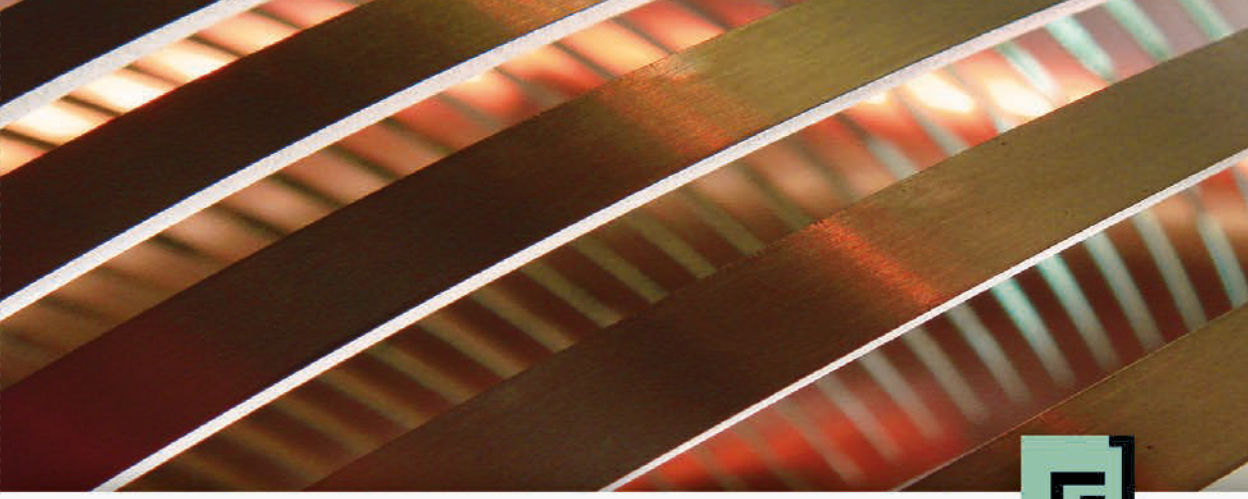


Broschüre zu Java-Threadpools und dem Future-Pattern mit einer Einführung in CompletableFutures

Jörg Hettel · Manh Tien Tran

Nebenläufige Programmierung mit Java

Konzepte und Programmiermodelle für Multicore-Systeme

dpunkt.verlag

Jörg Hettel studierte Theoretische Physik und promovierte am Institut für Informationsverarbeitung und Kybernetik an der Universität Tübingen. Nach seiner Promotion war er als Berater bei nationalen und internationalen Unternehmen tätig. Er begleitete zahlreiche Firmen bei der Einführung von objektorientierten Technologien und übernahm als Softwarearchitekt Projektverantwortung. Seit 2003 ist er Professor an der Hochschule Kaiserslautern am Standort Zweibrücken. Seine aktuellen Arbeitsgebiete sind u.a. verteilte internetbasierte Transaktionssysteme und die Multicore-Programmierung.
joerg.hettel@hs-kl.de.

Manh Tien Tran studierte Informatik an der TU Braunschweig. Von 1987 bis 1995 war er wissenschaftlicher Mitarbeiter am Institut für Mathematik der Universität Hildesheim, wo er 1995 promovierte. Von 1995 bis 1998 war er als Softwareentwickler bei BOSCH Blaupunkt beschäftigt. 1999 wechselte er zu Harman Becker und war dort bis 2000 für Softwarearchitekturen zuständig. Seit 2000 ist er Professor an der Hochschule Kaiserslautern am Standort Zweibrücken. Seine aktuellen Arbeitsgebiete sind Frameworks, Embedded-Systeme und die Multicore-Programmierung.
manhtien.tran@hs-kl.de.

Lektorat: Martin Wohlrab
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Jörg Hettel, Manh Tien Tran
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de

Diese Broschüre basiert auf dem Buch
»Nebenläufige Programmierung mit Java«, dpunkt.verlag, 2016 (ISBN 978-3-86490-369-4).

Copyright © 2017 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

Jörg Hettel · Manh Tien Tran

Java ExecutorService

Threadpools und das Future-Pattern

dpunkt.verlag

Vorwort

Ausgangspunkt der nebenläufigen Programmierung ist bei Java, wie bei vielen anderen Programmiersprachen, das Thread-Konzept. Die Einführung von Threads ist die Übertragung der programmflussbasierten Ablaufbeschreibung, wie sie bei sequenziellen Programmen benutzt wird, auf quasiparallele Abläufe. Zur Steuerung und Synchronisierung von nebenläufigen Threads stehen bei Java seit der Version 1.0 Low-Level-Mechanismen wie Synchronisierung (Schlüsselwort `synchronized`) und Bedingungsvariablen (Objektmethoden `wait` und `notify` bzw. `notifyAll`) zur Verfügung. Da moderne Rechner eine symmetrische Multiprozessor-(SMP-)Architektur mit verschiedenen Cache-Hierarchien aufweisen, spielt auch implizit das Thema »Sichtbarkeit« eine große Rolle. So wurden den Schlüsselwörtern `synchronized` und `volatile` auch Sichtbarkeitsgarantien zugewiesen (siehe z. B. [1],[2], [3]).

Der Umgang mit Low-Level-Konzepten ist aber sehr fehleranfällig, da die Beschreibung von Nebenläufigkeit auf Kontrollflussebene nicht intuitiv ist. So kommt es durch den verzahnten Ablauf von Programmflüssen zu einer exponentiell steigenden Anzahl von verschiedenen Zugriffsreihenfolgen auf gemeinsam benutzte Ressourcen. Darüber hinaus besitzen diese Low-Level-Konzepte einige Limitierungen, die die Benutzungsmöglichkeiten einschränken.

Da nun die nebenläufige Programmierung durch die Einführung von Multicore-Systemen in das Alltagsgeschäft der Softwareentwicklung Einzug gehalten hat, werden die Programmiersprachen um bessere und abstraktere Konzepte erweitert. Ziel ist es, den Umgang mit der Nebenläufigkeit zu vereinfachen und sicherer zu machen.

In dieser Broschüre werden die Abstraktionskonzepte *Executor*, *Future* und *CompletableFuture* vorgestellt. Die Einführung von `Executors` (Threadpools), und damit die Implementierung des Future-Patterns, ersetzt den rudimentären Thread durch eine Task-Abstraktion. Das Future-Pattern realisiert hierbei einen allgemeingültigen Rückgabemechanismus für Task-Ergebnisse. Die mit Java 8 eingeführte `CompletableFuture`-Klasse erlaubt es, nebenläufige Ablaufketten zu definieren und asynchron auszuführen, was der Einführung einer Beschreibung für Task-Parallelität entspricht.

Die Broschüre gibt lediglich einen Überblick über diese Konzepte. Detaillierte Informationen zu diesen beiden Themen und weitere wichtige Nebenläufigkeitskonzepte finden Sie in unserem Buch *Nebenläufige Programmierung mit Java. Konzepte und Programmiermodelle für Multicore-Systeme* [2].

Danksagungen

Ein herzliches Dankeschön geht an die Mitarbeiter des dpunkt.verlags und insbesondere an Frau Christa Preisendanz, Frau Ursula Zimpfer und Herrn Martin Wohlrab, die die Fertigstellung dieser Broschüre professionell begleitet haben.

Jörg Hettel und Manh Tien Tran
Zweibrücken, Januar 2017

http://www.hs-kl.de/java-concurrency

Inhalt

1 Der Umgang mit Threadpools

Zahlreiche Aufgaben, die nebenläufig ausgeführt werden sollen, sind oft nur von kurzer Dauer und treten nicht unbedingt regelmäßig auf. Würde man also für jede neue Aufgabe einen Thread erzeugen und starten, würde das Betriebssystem unnötig belastet werden. Es ist sinnvoller, Threads wiederzuverwenden.

Ein weiterer Punkt ist, dass sich eine große Anzahl von Threads negativ auf die Systemleistung auswirkt. Die maximale Anzahl von nebenläufigen Aktivitäten, die ein Prozess verwalten kann, ist nicht festgelegt und hängt von der Implementierung der JVM (Java Virtual Machine) und dem zugrunde liegenden Betriebssystem ab. Es ist daher wichtig, die Menge der erzeugten Threads zu beschränken.

In der Praxis wird man deshalb weniger mit rudimentären Thread-Objekten arbeiten, sondern mit sogenannten *Threadpools*. Java stellt verschiedene Realisierungen in dem Paket `java.util.concurrency` zur Verfügung, die wir im Folgenden besprechen.

1.1 Threadpools und die Klasse Executors

Ein Threadpool verwaltet eine gewisse Anzahl von Threads. Soll eine Aufgabe nebenläufig durchgeführt werden, so übergibt man dem Pool ein entsprechendes `Runnable`-Objekt[1]. Je nach Art des Pools wird es sofort einem Thread zugeteilt oder erst in eine Warteschlange (Queue) gestellt und später bearbeitet. Wenn der Thread das `Runnable` ausgeführt hat, wird er ohne zu terminieren zurück in den Pool gestellt und kann weitere noch wartende Aufgaben übernehmen. Es werden somit nicht dauernd neue Threads vom Betriebssystem angefordert und beendet.

[1] Ein Objekt, das die Schnittstelle `Runnable` implementiert und somit als Einstiegspunkt für einen nebenläufigen Kontrollfluss dient.

Java bietet für die Erzeugung verschiedener Typen von Threadpools die `Executors`-Klasse an. Die gebräuchlichsten Fabrikmethoden sind:

- `newCachedThreadPool()`
- `newFixedThreadPool(int nThreads)`
- `newScheduledThreadPool(int coreSize)`
- `newWorkStealingPool()`
- `newWorkStealingPool(int parallelism)`

Die Methoden liefern entweder eine `ExecutorService`- oder `ScheduledExecutorService`-Implementierung zurück (vgl. Abb. 1-1).

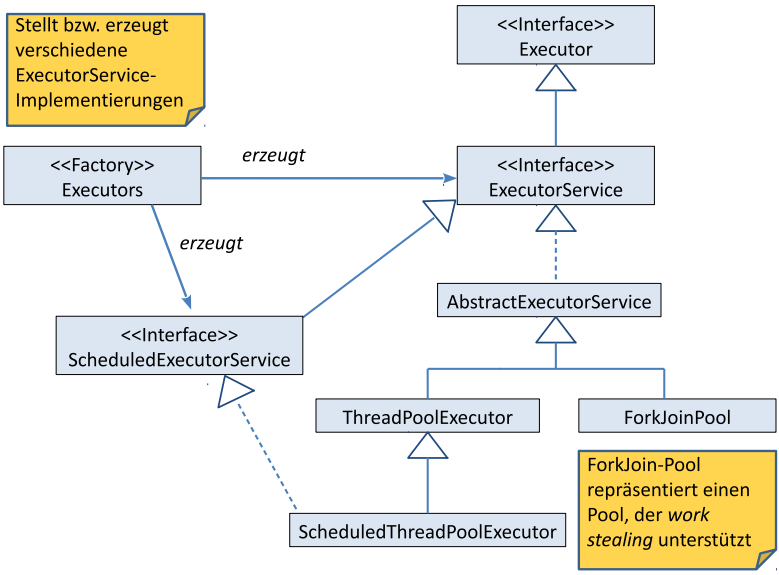


Abbildung 1-1: Klassenhierarchie des Poolkonzepts

Mit `newCachedThreadPool` erhalten wir einen Pool, der bei Bedarf neue Threads erzeugt. Unbenutzte Threads bleiben für 60 Sekunden erhalten und werden danach, falls sie zwischenzeitlich nicht benötigt wurden, terminiert. Pools dieser Art werden typischerweise zur Performance-Verbesserung von Programmen eingesetzt, die im Programmverlauf immer wieder kurzlebige, asynchrone Tasks benötigen.

Mit `newFixedThreadPool(int nThreads)` wird ein Pool mit einer festen Anzahl von Threads erzeugt. Pools dieser Art haben eine unbeschränkte Warteschlange für übergebene Aufgaben. Zu jedem Zeitpunkt sind höchstens `nThreads` Threads tätig. Durch die angegebene Obergrenze wird sichergestellt, dass Tasks in diesem Pool nur die zugewiesene Rechenkapazität (`nThreads` Threads) zur Verfügung steht. Dieser Pool kann

typischerweise für lang laufende Task eingesetzt werden, die eine niedrige Priorität besitzen und deren Ergebnis nicht sofort benötigt wird.

Die Fabrikmethode `newScheduledThreadPool(int coreSize)` liefert einen `ScheduledExecutorService`, mit dessen Hilfe Aufgaben nach einer gegebenen Verzögerung bzw. periodisch ausgeführt werden können. Mit Java 8 wurden noch zwei Fabrikmethoden für den `ForkJoinPool` eingeführt, der das sogenannte *Work-Stealing*-Verfahren unterstützt. *Work-Stealing*-Pools kommen insbesondere bei der Parallelisierung von *Divide-and-Conquer*-Verfahren zum Einsatz und bilden die Grundlage des ForkJoin-Frameworks und der parallelen Array- und Stream-Verarbeitung.

Alle Implementierungen des funktionalen Interface `Executor` stellen die `execute`-Methode bereit:

```
public interface Executor
{
  void execute(Runnable command);
}
```

Somit können allen Threadpools `Runnable`-Objekte bzw. entsprechende Lambda-Ausdrücke zur Ausführung übergeben werden:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
executor.execute( () -> System.out.println("Hallo Welt") );
executor.shutdown();
```

Das Interface `ExecutorService` stellt noch weitere wichtige Methoden für den Einsatz von Threadpools zur Verfügung.

Die Threads von `ScheduledThreadPoolExecutor` bzw. `ThreadPoolExecutor` sind standardmäßig sogenannte User-Threads. Da sie nicht von sich aus terminieren, muss der Threadpool mit `shutdown` kontrolliert beendet werden. Nach dem `shutdown` werden die zugewiesenen Aufgaben noch abgearbeitet, neue werden aber abgewiesen. Zu beachten ist, dass `shutdown` kein blockierender Aufruf ist. Mit `isShutdown` kann man den aktuellen Status abfragen und mit `isTerminated` erhält man die Auskunft, ob der Pool terminiert ist.

Das Herunterfahren eines Pools kann mit `shutdownNow` erzwungen werden. Hierbei werden alle aktiven Tasks des Pools mit `interrupt` zum Aufhören aufgefordert. Die in der Bearbeitungsqueue liegenden, wartenden Aufträge werden zurückgegeben. Diese Methode garantiert nicht, dass sich der Pool sofort beendet, da die Reaktion der Tasks auf `interrupt` implementierungsabhängig ist. Neben der nicht blockierenden Methode `shutdown` gibt es noch die blockierende `awaitTermination(long`

`timeout, TimeUnit unit)`, die erst zurückkehrt, wenn alle Threads terminiert sind oder die angegebene Zeit abgelaufen ist.

1.2 Executors mit eigener ThreadFactory

Bei einigen der Fabrikmethoden kann eine `ThreadFactory` als Argument übergeben werden. Codebeispiel 1.1 zeigt die Erzeugung eines *CachedThreadPool*, dessen Threads alle die *Daemon*-Eigenschaft und eine geringe Priorität besitzen. Dieser Pool muss daher nicht mit `shutdown` beendet werden.

```
final ExecutorService executor = Executors.newCachedThreadPool(
   new ThreadFactory()
   {
     @Override
     public Thread newThread(Runnable r)
     {
       Thread th = new Thread(r, "MyFactoryThread");
       th.setPriority(Thread.MIN_PRIORITY);
       th.setDaemon(true);
       return th;
     }
   });
```

Codebeispiel 1.1: Ein Threadpool mit eigener Factory

1.3 Explizite ThreadPoolExecutor-Erzeugung

Man kann auch direkt einen Threadpool erzeugen, ohne die Fabrikmethoden von `Executors` zu benutzen. Für die direkte Erzeugung eines `ThreadPoolExecutor` steht unter anderem folgender Konstruktor zur Verfügung:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)
```

Über die Parameter `corePoolSize` und `maximumPoolSize` kann die Anzahl der zur Verfügung gestellten Threads gesteuert werden. Es wird sichergestellt, dass mindestens `corePoolSize` Threads existieren. Bei Bedarf werden bis `maximumPoolSize` Threads erzeugt. Sind `corePoolSize`

und `maximumPoolSize` gleich, so handelt es sich um einen *fixed-size* Pool. Die beiden Parameter können bei den meisten Pools auch zur Laufzeit geändert werden. Mit `keepAliveTime` und `unit` wird die Zeit angegeben, wie lange unbenutzte Threads gehalten werden. Die `workQueue` wird benutzt, um übergebene Tasks ggf. zwischenzuspeichern. Es gibt dafür verschiedene Implementierungen. Mit `handler` wird angegeben, wie mit Tasks bei einer vollen Queue zu verfahren ist. Zur Verfügung stehen hier folgende Strategien:

- `AbortPolicy`: Entspricht dem Defaultverhalten. Es wird eine `RejectedExecutionException` geworfen.
- `CallerRunsPolicy`: Hier wird der Task abgelehnt. Er wird aber von dem `execute` aufrufenden Thread ausgeführt. Es findet in diesem Fall ein blockierender Aufruf statt.
- `DiscardOldestPolicy`: Der am längsten wartende Task wird zugunsten des neuen verdrängt.
- `DiscardPolicy`: Hier wird der übergebene Task ohne eine spezielle Meldung, z. B. über eine Exception, ignoriert.

2 Das Future-Pattern

Die Benutzung von Threadpools entlastet den Entwickler vom direkten Umgang mit Threads. Einem Threadpool werden direkt Task-Objekte übergeben, die nebenläufig ausgeführt werden[1]. Da die `run`-Methode des `Runnable`-Interface keine Rückgabe vorsieht, nebenläufige Tasks aber oft eine besitzen, wurde mit den Threadpools zusammen auch eine Implementierung des Future-Patterns[2] eingeführt.

2.1 Callable, Future und ExecutorService

Die von einem `Executor` angebotene Methode `execute` erwartet ein `Runnable`-Objekt. Soll ein Task eine Rückgabe liefern, kann man das über die Verwendung eines speziellen *Rückgabe-Attributs* realisieren. Codebeispiel 2.1 zeigt eine mögliche Implementierung.

```
public class RunnableWithReturn<T> implements Runnable
{
  private T returnValue;
  private volatile Thread self;
  // ...

  public void run()
  {
    self = Thread.currentThread();
    // Berechnung und Ergebnis in returnValue abspeichern
  }

  // Blockierende Abfrage des Return-Werts
  public T get()
  {
    self.join();
    return returnValue;
  }
}
```

Codebeispiel 2.1: Aufbau eines `Runnable` mit Rückgabe

[1] Eigentlich sollte man nicht von Threadpools, sondern tatsächlich von »Executors« reden, da sie zusätzlich zur Thread-Verwaltung auch die Ausführung von Tasks übernehmen.

[2] Das Future-Pattern entspricht hier dem *Active Object Pattern* [4].

Die Lösung beinhaltet aber verschiedene Probleme: `self` kann `null` sein und der Task hat keine Möglichkeit, auftretende Ausnahmen während der Ausführung zurückzugeben. Eine einfache Lösung dafür ist die Verwendung eines sogenannten `Future`-Objekts.

Mit dem `ExecutorService` wird das Interface `Future` eingeführt, mit dessen Hilfe die Ergebnisrückgabe einer asynchronen Berechnung einfach und einheitlich realisiert wird. Über das `Future`-Objekt kann neben dem Ergebnis auch der Status der Berechnung abgefragt werden.

Ein `Callable` wird einem `ExecutorService` über die Methode `submit` zur Ausführung übergeben, die ein `Future`-Objekt zurückliefert. Über dieses kann die Rückgabe erfragt werden. Das Codebeispiel 2.2 demonstriert eine Verwendung.

```
public static void main(String[] args)
{
  Callable<Integer> callable = new Callable<Integer>()                    ❶
  {
    @Override
    public Integer call() throws Exception                                ❷
    {
      return 42;
    }
  };

  ExecutorService executor = Executors.newCachedThreadPool();
  Future<Integer> future = executor.submit( callable );                   ❸

  try
  {
    Integer result = future.get();                                        ❹
    System.out.println( result );
  }
  catch (InterruptedException | ExecutionException e)                     ❺
  {
    // ...
  }
}
```

Codebeispiel 2.2: Ein Beispiel mit einem `Callable` und `Future`

Statt eines `Runnable`-Objekts wird jetzt ein parametrisiertes `Callable`-Objekt benutzt (❶). Die zu implementierende `call`-Methode hat eine typisierte Rückgabe (❷). Das `Callable` wird über `submit` dem Threadpool übergeben und man erhält ein `Future`-Objekt (❸). Da `Callable` auch ein funktionales Interface ist, kann man auch schreiben:

```
Future<Integer> future = executor.submit( () -> 42 );
```

Das Ergebnis der asynchronen Berechnung kann nun über die `get`-Methode des `Future`-Objekts erfragt werden (❹). Dabei bleibt `get` so lange blockiert, bis das Ergebnis vorliegt. Die `get`-Methode kann die beiden Ausnahmen `InterruptedException` und `ExecutionException` werfen (❺). Die Fehlerbehandlung im Zusammenhang mit `Callable` und `Future` besprechen wir später noch genauer. Abbildung 2-1 zeigt den Ablauf im Sequenzdiagramm.

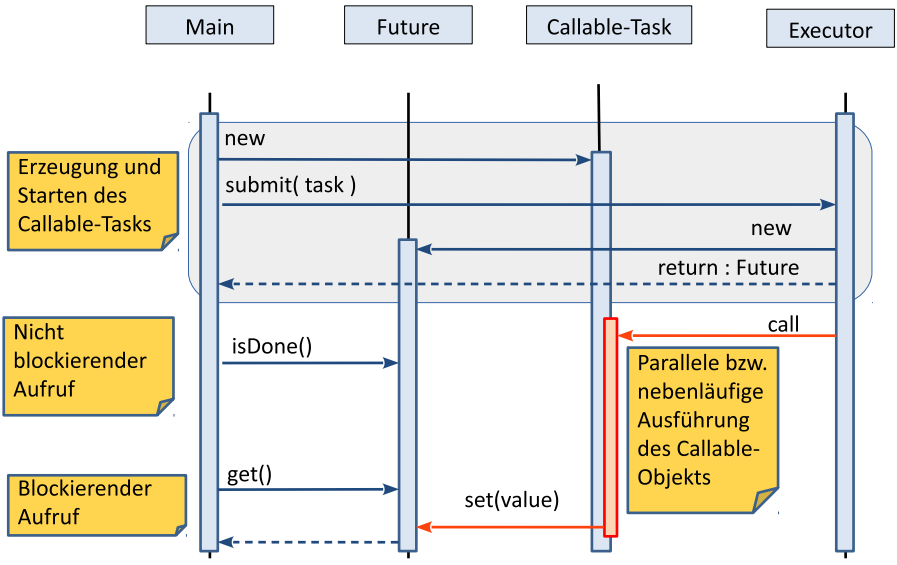


Abbildung 2-1: Funktionsweise des Future-Patterns

Ein `Future<V>` bietet neben `get` noch weitere Methoden an. Durch `get(long timeout, TimeUnit unit)` kann der Aufrufer eine maximale Wartezeit angeben. Ist das Ergebnis nach der vorgegebenen Zeit nicht verfügbar, wird eine `TimeOutException` geworfen. Mit `isDone` kann der Bearbeitungsstatus abgefragt werden. Zum Abbrechen kann `cancel(boolean mayInterruptIfRunning)` benutzt werden. Ist der Task noch nicht gestartet, wird er nicht ausgeführt. Befindet er sich mitten in der Abarbeitung, wird in Abhängigkeit vom aufrufenden Parameter dem ausführenden Thread ggf. ein *Interrupt* gesendet. Der Task muss in dem Fall dann so implementiert sein, dass er den *Interrupt* berücksichtigt. Mit `isCancelled` kann geprüft werden, ob der Task abgebrochen wurde.

2.2 Callable und ThreadPoolExecutor

An einen `ExecutorService` kann man einen Task vom Typ `Runnable` oder `Callable` senden, der vom Pool möglichst bald ausgeführt wird. Alle folgenden Methoden kehren nach dem Aufruf unmittelbar zurück, sodass der aufrufende Thread seine Tätigkeit nebenläufig ausführen kann:

- `Future<?> submit(Runnable task)`: Das zurückgegebene Objekt wird verwendet, um `isDone`, `cancel` und `isCancelled` aufzurufen. Der `get`-Aufruf liefert bei Fertigstellung nur den Wert `null`.
- `Future<T> submit(Runnable task, T result)`: Im Vergleich zum obigen `submit` liefert `get` das vorgegebene `result`-Objekt als Ergebnis zurück.
- `Future<T> submit(Callable<T> task)`: In dieser Version wird ein `Future`-Objekt zurückgeliefert, über das das Ergebnis der Berechnung abgeholt werden kann.

Schauen wir uns die Verwendung von `ExecutorService`, `Callable` und `Future` etwas näher an. Die Klasse `FindWordInFiles` aus Codebeispiel 2.3 realisiert ein `Callable` zur asynchronen Suche nach einem Wort in einer Datei.

```
class FindWordInFiles implements Callable<List<String>>          ❶
{
  private final Pattern searchPattern;
  private final Path path;   // Dateipfad

  public FindWordInFiles(Path path, String search)
  {
    this.path = path;
    this.searchPattern = Pattern.compile(".*\\b"+search+"\\b.*");
  }



  public List<String> call() throws IOException                   ❷
  {
    List<String> result = new ArrayList<>();
    List<String> lines = Files.readAllLines(path,
                                      StandardCharsets.UTF_8);

    int count = 0;
    for (String line : lines)
    {
      count++;
      if ( searchPattern.matcher(line).matches() )
      {
        result.add( path + " " + count + " : " + line);
      }
    }
```

```
    return result;
  }
}
```

Codebeispiel 2.3: Beispiel für eine Suche nach einem Wort in einer Datei

Die Rückgabe des asynchron ausgeführten Tasks ist `List<String>` (❶,❷), wobei jeder Eintrag der Liste die Zeile enthält, in der das Suchwort vorkommt. Im Codebeispiel 2.4 wird das Wort »Haus« parallel in drei Textdateien gesucht.

```
public class FindWordBeispiel
{
  public static void main(String[] args)
  {
    ExecutorService pool = Executors.newCachedThreadPool();
    String search = "Haus";

    Callable<List<String>> task1 = new FindWordInFiles(                ❶
                            Paths.get("Text1.txt"), search);
    Callable<List<String>> task2 = new FindWordInFiles(
                            Paths.get("Text2.txt"), search);
    Callable<List<String>> task3 = new FindWordInFiles(
                            Paths.get("Text3.txt"), search);

    Future<List<String>> task1Future = pool.submit(task1);             ❷
    Future<List<String>> task2Future = pool.submit(task2);
    Future<List<String>> task3Future = pool.submit(task3);

    try
    {
      List<String> task1Liste = task1Future.get();                     ❸
      List<String> task2Liste = task2Future.get();
      List<String> task3Liste = task3Future.get();


      task1Liste.forEach(System.out::println);                         ❹
      task2Liste.forEach(System.out::println);
      task3Liste.forEach(System.out::println);
    }
    catch (InterruptedException | ExecutionException e)
    {
      e.printStackTrace();
    }
    pool.shutdown();
  }
}
```

Codebeispiel 2.4: Beispiel für eine parallele Suche

Zuerst werden die drei `Callable`-Objekte erzeugt (❶) und dann an den Threadpool mit `submit` einzeln übergeben (❷). Mit `get` wird anschließend auf das Ende des jeweiligen Tasks gewartet (❸), bevor die Ergebnisse auf die Konsole ausgegeben werden (❹).

Anstatt die Tasks einzeln an den Threadpool zu übergeben, können diese auch in eine `Collection` aufgenommen und dann mit `invokeAll` auf einmal übergeben werden (vgl. Codebeispiel 2.5). Hier ist zu beachten, dass `invokeAll` blockiert und erst zurückkommt, wenn alle Tasks beendet sind.

```
List<Callable<List<String>>> tasks = new ArrayList<>();
tasks.add(new FindWordInFiles(Paths.get("Text1.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text2.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text3.txt"), search));

try
{
  List<Future<List<String>>> tasksFuture=pool.invokeAll(tasks);

  for( Future<List<String>> future : tasksFuture )
  {
    future.get().forEach(System.out::println);
  }
}
catch (InterruptedException | ExecutionException e)
{
  e.printStackTrace();
}
```

Codebeispiel 2.5: Beispiel für die Verwendung von `invokeAll`

In den beiden Codebeispielen 2.4 und 2.5 musste mit der Ausgabe immer so lange gewartet werden, bis auch der langsamste Task fertig war. Das kann zu unnötigen Wartezeiten führen, da man mit der Veröffentlichung der Ergebnisse beginnen könnte, wenn der erste Task zu Ende ist. Um diese Limitierung zu umgehen, kann ein `CompletionService` eingesetzt werden. Ein `CompletionService` verwaltet eine interne Queue, in die die `Future`-Objekte eingestellt werden, sobald die zugehörigen Tasks beendet sind. Codebeispiel 2.6 demonstriert dies.

```
ExecutorService pool = Executors.newCachedThreadPool();
String search = "Haus";
List<Callable<List<String>>> tasks = new ArrayList<>();
tasks.add(new FindWordInFiles(Paths.get("Text1.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text2.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text3.txt"), search));

CompletionService<List<String>> completionService =                ❶
                        new ExecutorCompletionService<>(pool);
tasks.forEach(completionService::submit);                          ❷
```

```
try
{
  for (int i = 0; i < tasks.size(); i++)
  {
    Future<List<String>> future = completionService.take();      ❸
    future.get().forEach(System.out::println);
  }
}
catch (InterruptedException | ExecutionException e)
{
  e.printStackTrace();
}
pool.shutdown();
```

Codebeispiel 2.6: Beispiel für die Verwendung von `CompletionService`

Bei der Erzeugung eines `ExecutorCompletionService`, einer Implementierung von `CompletionService`, wird der zu benutzende Threadpool angegeben (❶). Danach werden ihm die Tasks mit `submit` übergeben (❷). Sobald ein Task beendet ist, kann dessen `Future`-Objekt mit `take` aus der internen Queue des `CompletionService` entnommen werden (❸).

Hinweis

Wird ein `ExecutorService` nicht mehr benötigt, sollte dessen `shutdown`-Methode aufgerufen werden, damit die belegten Ressourcen an das Betriebssystem zurückgegeben werden.

2.3 Callable und ScheduledThreadPoolExecutor

Für Tasks, die mehrfach bzw. periodisch ausgeführt werden sollen, steht die Klasse `ScheduledThreadPoolExecutor` mit dem Interface `ScheduledExecutorService` zur Verfügung. Instanzen können wie bei `ThreadPoolExecutor` am bequemsten über die Fabrikmethoden `newScheduledThreadPool` bzw. `newSingleThreadScheduledExecutor` der `Executors`-Klasse erhalten werden. Das `ScheduledExecutorService`-Interface ist von `ExecutorService` abgeleitet und stellt zusätzliche Methoden zur periodischen Ausführung von Tasks bereit.

Mit den `schedule`-Methoden kann ein `Runnable`- bzw. `Callable`-Task nach der angegebenen Zeit einmal ausgeführt werden. Für die periodische Ausführung kann `scheduleAtFixedRate` verwendet werden. Nach einer

Anfangsverzögerung wird der Task periodisch gestartet. Wenn für die Ausführung einer Wiederholung länger als die angegebene Periode benötigt wird, werden die folgenden entsprechend später ablaufen. Es wird garantiert, dass sich Aktivitäten nie überlappen.

Codebeispiel 2.7 zeigt, wie der `ScheduledExecutorService` eingesetzt werden kann. Es wird ein Task gestartet, der jede Sekunde einen Signalton ausgibt (❶). Parallel dazu wird ein Task eingestellt, der nach 10 Sekunden den Signalton stoppt und den Pool herunterfährt (❷).

```
ScheduledExecutorService scheduler =
               Executors.newScheduledThreadPool(1);
ScheduledFuture<?> beeperHandle =
         scheduler.scheduleAtFixedRate(                          ❶
                     Toolkit.getDefaultToolkit()::beep,
                     0, 1, TimeUnit.SECONDS);

scheduler.schedule( () -> { beeperHandle.cancel(true);
                            scheduler.shutdown();                ❷
                          },
                    10, TimeUnit.SECONDS);
```

Codebeispiel 2.7: Beispiel für geplante Ausführungen

2.4 Callable und ForkJoinPool

In Java 7 wurde zusammen mit dem ForkJoin-Framework der `ForkJoinPool` eingeführt, der in Java 8 noch mal überarbeitet wurde. Das ist der Standardpool, der für die Java-internen Parallelisierungen, wie z. B. bei den parallelen Array-Methoden und *parallel Streams*, eingesetzt wird.

Der `ForkJoinPool` benutzt für seine interne Verwaltung ein *Work-Stealing*-Verfahren. Bei diesem Verfahren besitzt im Prinzip jeder Thread eine eigene Task-Queue, aus der er seine Aufträge holt bzw. in die er Aufträge, die er generiert, hineinstellt. Ist seine Queue leer, holt er sich vom Ende anderer Task-Queues Aufgaben und bearbeitet diese. Ein *Work-Stealing*-Pool kommt insbesondere mit einer vorher nicht abschätzbaren, hohen Anzahl von Tasks mit azyklischen Abhängigkeiten[3] zurecht, wie sie typischerweise in *Divide-and-Conquer*-Algorithmen auftreten. Sind die Tasks dagegen unabhängig voneinander, wie in unseren bisherigen Beispielen, besitzt er gegenüber einem `ThreadPoolExecutor` keine Vorteile, weil in diesem Fall insgesamt nur eine Queue benötigt wird.

[3] Es gibt keine gegenseitige Abhängigkeit zwischen Tasks. Ihre Beziehungen können durch eine Baumstruktur beschrieben werden.

Einen `ForkJoinPool` kann man sich entweder über die Fabrikmethoden der `Executors`-Klasse erzeugen oder durch den direkten Aufruf eines Konstruktors.

Der CommonPool

Um das ständige Erzeugen und Schließen von Pools zu vermeiden, benutzt Java einen globalen Threadpool, der bei der ersten Verwendung von Java-eigenen Parallelisierungskonzepten angelegt wird. Zugriff auf diesen Pool erhält man mit `ForkJoinPool.commonPool`. Möchte man den *CommonPool* konfigurieren, so kann man dies über das Setzen von System-Properties[4] bewerkstelligen:

- `java.util.concurrent.ForkJoinPool.common.parallelism`
- `java.util.concurrent.ForkJoinPool.common.threadFactory`
- `java.util.concurrent.ForkJoinPool.common.exceptionHandler`

Der Defaultwert für `parallelism` ist in der Regel `Runtime.getRuntime().availableProcessors() - 1`, falls mehrere Kerne zur Verfügung stehen. Die Defaultkonfiguration kann auch innerhalb der Anwendung geändert werden. Hierbei muss beachtet werden, dass dies vor dem ersten Aufruf von `ForkJoinPool.commonPool` geschieht. Der folgende Code zeigt, wie man die Anzahl der verwendeten Threads setzen kann:

```
System.setProperty(
 "java.util.concurrent.ForkJoinPool.common.parallelism", "4");
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

Bitte beachten Sie, dass Java selbst für die internen parallelen Konzepte `ForkJoinPool.commonPool` aufruft.

Hinweis

Benutzt man Threads aus dem `ForkJoinPool.commonPool` z.B. für Aufgaben, die lange laufen und oft blockiert sind, stehen diese Threads nicht mehr den Java-internen Parallelisierungen zur Verfügung.

[4] Aufrufparameter beim Starten der virtuellen Maschine.

2.5 Exception-Handling

Eine wichtige Frage ist, wie mit Fehlern umgegangen wird, die in nebenläufig ausgeführten Tasks auftreten. Betrachten wir ein Beispiel, in dem wir einen Task mit einer Division durch null an einen Pool senden:

```
ExecutorService executor = Executors.newCachedThreadPool();
executor.execute( () -> System.out.println(1 / 0) );
```

Wir erhalten daraufhin die Meldung

```
Exception in thread "pool-1-thread-1" java.lang.ArithmeticException: /
    by zero
  at kap6.threadpool.ExceptionBeispiel1.lambda$0(ExceptionBeispiel1.
     java:12)
    ...
```

Der Pool-Thread wird durch die Ausnahme terminiert und der Default-Handler wird in diesem Fall aufgerufen. Wenn anstatt `execute` nun `submit` verwendet wird, also

```
 ExecutorService executor = Executors.newCachedThreadPool();
 executor.submit( () -> System.out.println(1/0) );
```

erscheint keine Ausgabe auf der Konsole. Der Grund ist, dass bei der Verwendung von `submit` jede nicht behandelte Ausnahme von `Runnable` oder `Callable` abgefangen wird:

```
public void run()
{
  Throwable thrown = null;
  try
  {
    while (!isInterrupted())
    {
      runTask(getTaskFromWorkQueue());
    }
  }
  catch (Throwable e)
  {
    thrown = e;
  }
  finally
  {
    threadExited(this, thrown);
  }
}
```

Die Ausnahme wird erst bei einem Zugriff mit `get` auf das zurückgegebene `Future`-Objekt ausgelöst. Mit

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<?> future = executor.submit(
                          () -> System.out.println(1 / 0) );
try
{
  future.get();
}
catch (InterruptedException | ExecutionException e)
{
  e.printStackTrace();
}
```

erhält man auf der Konsole folgende Ausgabe:

```
java.util.concurrent.ExecutionException: java.lang.ArithmeticException:
     / by zero
   at java.util.concurrent.FutureTask.report(Unknown Source)
   ...
```

Alternativ kann man die Exception im Task abfangen und loggen. Damit der Aufrufende über die Ausnahme in Kenntnis gesetzt wird, sollte die Exception weitergegeben werden:

```
future = executor.submit(() ->
   {
     try
     {
       System.out.println(1 / 0);
     }
     catch (Exception ex)
     {
        // Eigene Fehlerbehandlung etwa Loggen
        System.out.println("Ausführungsfehler = " + ex);
        throw ex; // Damit man über get die Ausnahme noch sieht
     }
   });
```

2.6 Tipps für das Arbeiten mit Threadpools

Im Folgenden sind einige nützliche Tipps zusammengestellt, die sich in der Praxis bewährt haben.

Temporäre Änderung des Thread-Namens

Beim Debugging ist es äußerst hilfreich, wenn man Threads über sinnvolle Namen identifizieren kann. Das Defaultschema für die Pool-Thread-Benennung ist `pool-N-thread-M`, wobei `N` die Poolnummer und `M` die Thread-Nummer ist.

Eine einfache Lösung, mit der Thread-Namen temporär geändert werden können, zeigt die folgende Hilfsmethode. Dabei wird das übergebene `Callable`-Objekt in einem Wrapper gekapselt. Der aktuelle Thread-Name wird vor der Ausführung der Aktivität abgespeichert, geändert (❶) und am Ende wieder hergestellt (❷).

```
public static <T> Future<T> submit(ExecutorService service,
    Callable<T> task, String name)
{
  return service.submit(() ->
   {
     Thread current = Thread.currentThread();
     String oldname = current.getName();
     current.setName(name);                                    ❶
     try
     {
       return task.call();
     } finally
     {
       current.setName(oldname);                               ❷
     }
   });
}
```

Anzahl der Pool-Threads

Neben der Frage, ob man Daemon-Threads nutzen möchte oder nicht, sollte man sich auch Gedanken über die Poolgröße machen. Eine angemessene Poolgröße hängt einerseits von der Anzahl und andererseits von der Art der zu bearbeitenden Tasks ab. Die Frage ist also, ob sie z. B. eher IO-intensiv oder rechenintensiv sind.

Goetz et al. [1] geben folgende Faustregel für die Anzahl der Pool-Threads an

$$N_{threads} = N_{cpu} \cdot U_{cpu} \cdot \left(1 + \frac{W}{C}\right)$$

mit

$$\begin{aligned} N_{cpu} &= \text{Anzahl der zur Verfügung stehenden Kerne} \\ U_{cpu} &= \text{Auslastung der CPU, } 0 \leq U_{cpu} \leq 1 \\ \frac{W}{C} &= \text{Verhältnis zwischen Warte- und Rechenzeit} \end{aligned}$$

Für rechenintensive Tasks und Vollauslastung sollte

$$\begin{aligned} N_{threads} &= N_{cpu} + 1 \\ &= \texttt{Runtime.getRuntime().availableProcessors()} + 1 \end{aligned}$$

gewählt werden, da selbst bei rechenintensiven Aufgaben es gelegentlich *page faults* und somit Unterbrechungen bzw. Wartezeiten gibt.

3 CompletableFuture

Mithilfe des `Future`-Konzepts ist es sehr einfach, einen asynchronen Task zu starten und dessen Ergebnis zu einem späteren Zeitpunkt über entsprechende Methoden abzufragen. Möchte man z.B. mehrere intern voneinander abhängige Tasks ausführen, so muss der Ablauf mit den bisherigen Möglichkeiten aber explizit koordiniert werden.

3.1 CompletableFuture als Erweiterung des Future-Patterns

Der `CompletableFuture`-Mechanismus erlaubt, solche Ablaufketten einfach zu formulieren und asynchron auszuführen. Die Klasse `CompletableFuture` erweitert das bestehende Konzept. Man erkennt dies daran, dass sie neben dem `Future`- auch das neu eingeführte `CompletionStage`-Interface implementiert (vgl. Abb. 3-1).

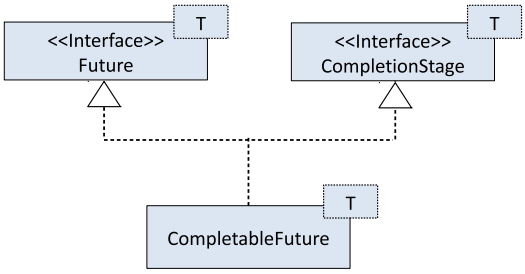


Abbildung 3-1: Klasse `CompletableFuture` mit ihren Interfaces

Bei der Verwendung von `CompletableFuture`-Objekten kommen im Wesentlichen die funktionalen Interfaces aus dem Paket `java.util.function` zum Einsatz, sodass in vielen Fällen Lambda-Ausdrücke verwendet werden können. Das folgende Codebeispiel 3.1 zeigt die Umsetzung der `Future`-Funktionalität mithilfe des `CompletableFuture`-Mechanismus, wobei hier bewusst keine Lambda-

Syntax benutzt wurde, um die Ähnlichkeiten mit dem `Future`-Konzept besser zu verdeutlichen (vgl. hierzu auch Codebeispiel 2.2 in Abschnitt 2.1).

```
public class SimpleCompletableFuture
{
  static class Task implements Supplier<Integer>                    ❶
  {
    @Override
    public Integer get()
    {
       return 42;
    }
  }

  public static void main(String[] args)
  {
    Future<Integer>  future  =
           CompletableFuture.supplyAsync( new Task() );             ❷
    // ...
    try
    {
      System.out.println( future.get() );                           ❸
    }
    catch (InterruptedException | ExecutionException e)
    {
      e.printStackTrace();
    }
  }
}
```

Codebeispiel 3.1: Umsetzung der `Future`-Funktionalität mit `CompletableFuture`

In dem Beispiel übernimmt ein `Supplier` die Rolle des `Callable` (❶). Nachdem das Task-Objekt erzeugt wurde, wird es mithilfe der Klassenmethode `CompletableFuture.supplyAsync` asynchron zur Ausführung gebracht (❷), die ein `CompletableFuture<Integer>`-Objekt zurückliefert. Der Task wird mithilfe eines Threads aus dem CommonPool ausgeführt. Es gibt eine überladene `supplyAsync`-Methode, bei der explizit ein Threadpool angegeben werden kann, der dann für die Ausführung der asynchronen Aktivitäten verantwortlich ist (später dazu mehr). Über die üblichen `Future`-Methoden kann nun auf das Ergebnis wie gewohnt zugegriffen werden (❸).

Schaut man sich die `CompletableFuture`-Klasse an, so erscheint das API überfrachtet. Bei näherer Betrachtung erkennt man, dass es mehrere Benutzungsrollen bedient. Abbildung 3-2 zeigt schematisch verschiedene Kategorien und die ihnen zugeordneten Methoden.

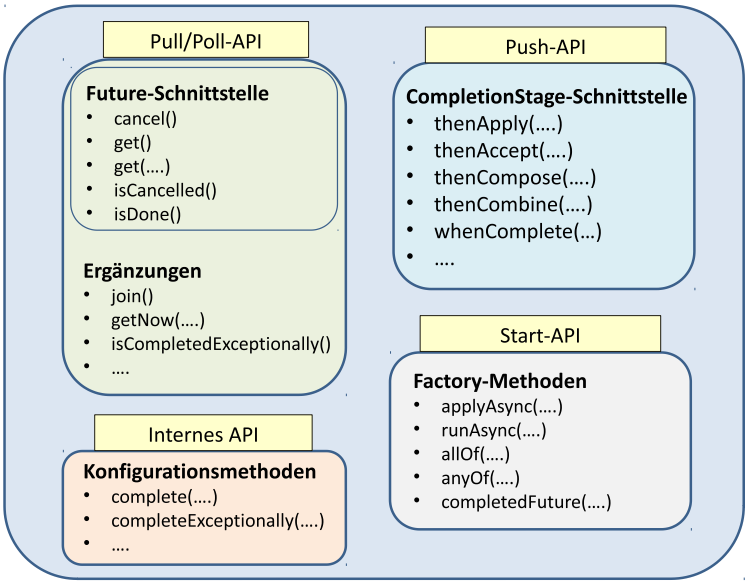


Abbildung 3-2: Aufteilung des `CompletableFuture`-APIs in verschiedene Kategorien

Die in der Kategorie »Internes API« zusammengefassten Methoden dienen zum internen Umgang mit `CompletableFutures`. Die Kategorie »Push-API« wird verwendet, um komplexe Verarbeitungsketten zu formulieren, was in Abschnitt 3.3 näher erläutert wird. Zum Starten asynchroner Verarbeitungen sind die Fabrikmethoden von »Start-API« zuständig.

Die Kategorie »Pull/Poll-API« beinhaltet das `Future`-Interface und definiert noch weitere Methoden, über die aktiv der Wert oder Status einer asynchronen Berechnung bzw. Berechnungskette abgefragt werden kann. Die wichtigsten hinzugekommenen Methoden sind:

- `T join()`: Liefert das Ergebnis der asynchronen Verarbeitung, wirft aber im Gegensatz zu `get` keine Exception. Die Methode blockiert, bis das Ergebnis vorliegt, oder wirft im Fehlerfall entweder die von `RuntimeException` abgeleitete `CancellationException` oder `CompletionException`.
- `T getNow(T valueIfAbsent)`: Fragt nach dem Ergebnis. Ist (noch) keines vorhanden, wird `valueIfAbsent` zurückgegeben. Auch diese Methode wirft im Fehlerfall ebenfalls `CancellationException` oder `CompletionException`.
- `boolean isCompletedExceptionally()`: Liefert `true`, falls während der Verarbeitung eine Ausnahme aufgetreten ist.

Im Gegensatz zu den im Interface `Future` definierten Methoden verwenden die zusätzlich aufgenommenen Methoden keine checked Exceptions, was der besseren Formulierung mit Lambda-Ausdrücken zugutekommt.

Damit kann das Codebeispiel 3.1 prägnanter formuliert werden, wie im Codebeispiel 3.2 dargestellt. Zu beachten ist, dass die Rückgabe diesmal vom Typ `CompletableFuture` ist (❶). Da der Wert mit `join` abgefragt wird (❷), ist eine explizite Fehlerbehandlung jetzt nicht mehr notwendig.

```
public class SimpleCompletableFuture
{
   public static void main(String[] args)
   {
      CompletableFuture<Integer>  future  =                          ❶
         CompletableFuture.supplyAsync( () -> 42 );

      // ...
      System.out.println( future.join() );                           ❷
   }
}
```

Codebeispiel 3.2: Ein einfaches Programm mit einem CompletableFuture

3.2 Asynchrone Verarbeitung: Task-Parallelität

Ein Hauptanwendungsgebiet von `CompletableFutur` ist die Konfiguration und asynchrone Ausführung komplexer Verarbeitungsketten.

3.2.1 Das Starten einer asynchronen Verarbeitung

Die `CompletableFuture`-Klasse hat verschiedene statische Methoden, die als Einstiegspunkte für asynchrone Verarbeitungsketten benutzt werden können. Im Codebeispiel 3.1 bzw. 3.2 wurde bereits `supplyAsync` verwendet. Im Folgenden sind die zur Verfügung stehenden Startmethoden aufgelistet:

- `CompletableFuture<U> supplyAsync(Supplier<U> supplier)`: Der übergebene `Supplier` wird durch einen Thread aus dem CommonPool asynchron ausgeführt, das Ergebnis wird in Form eines `CompletableFuture<U>` dem Aufrufer zurückgegeben.
- `CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)`: Wie vorhin mit dem Unterschied, dass explizit der übergebene `executor` verwendet wird.

- `CompletableFuture<Void> runAsync(Runnable runnable)`: Das übergebene `Runnable`-Objekt wird asynchron von einem Thread aus dem CommonPool ausgeführt. Da die `run`-Methode des `Runnable`-Interface als Rückgabe `void` hat, wird ein `CompletableFuture<Void>`-Objekt zurückgeliefert.
- `CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)`: Hat dieselbe Funktionalität, nur dass zur Ausführung des `Runnable`-Objekts ein Thread aus dem übergebenen Threadpool (Argument `executor`) übernommen wird.

3.2.2 Definition einer asynchronen Verarbeitungskette

Das `CompletionStage`-Interface stellt verschiedene, sogenannte *Push*-Methoden zur Verkettung von Aktivitäten bereit. Die Grundidee ist hier, dass jeder Task ein `CompletableFuture` als Ergebnis liefert, sodass eine Ablaufkette über ein *Fluent Programming Style* gebildet werden kann. Die einzelnen Tasks werden hierbei allerdings durch Funktionen bzw. Lambda-Ausdrücke formuliert.

Abbildung 3-3 zeigt schematisch ein Beispiel eines asynchronen, sequenziellen Ablaufs aus vier Tasks.

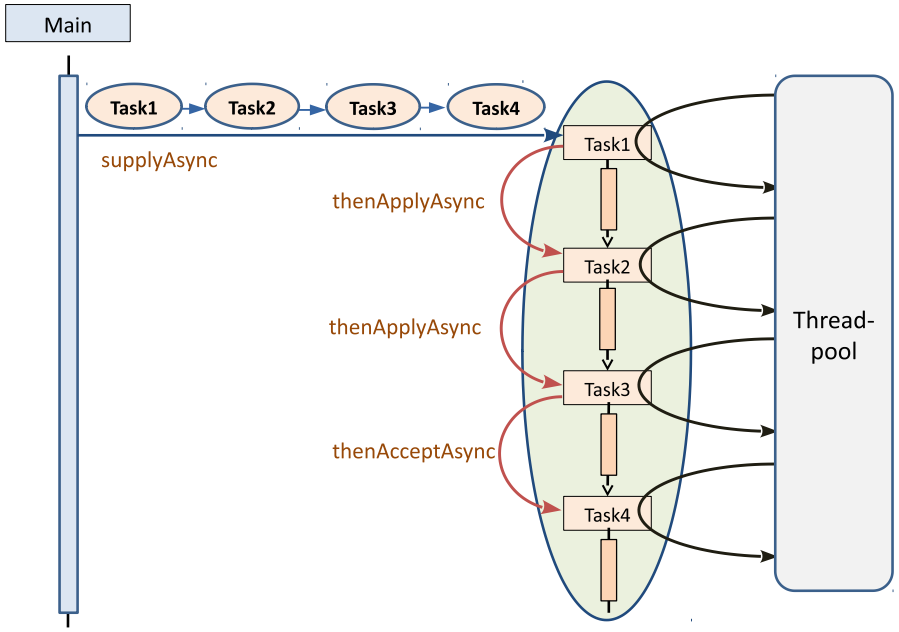


Abbildung 3-3: Eine asynchrone Verarbeitung von vier Tasks mithilfe des `CompletableFuture`-Konzepts

3.3 Das Arbeiten mit CompletableFutures

Mit `CompletableFuture`-Objekten können nicht nur einfache asynchrone Abläufe definiert und ausgeführt werden. Im Folgenden wird das von `CompletableFuture` zur Verfügung gestellte API kurz vorgestellt und verschiedene Ablauf-Patterns aufgezeigt. Detailliertere Informationen und Codebeispiele sind in [2] zu finden.

3.3.1 Das Konzept des CompletionStage

Über die Methoden des `CompletionStage`-Interface können Ablaufsequenzen zusammengestellt werden. Hierbei kann für jeden Folgeschritt entschieden werden, wie er ausgeführt werden soll. Bei einer expliziten asynchronen Ausführung kann noch alternativ ein Threadpool (`Executor`) angegeben werden, falls nicht der CommonPool benutzt werden soll. Diese Variabilität führt dazu, dass für jede Methode in der Regel drei Varianten existieren. So gibt es z.B. für die *thenApply*-Methode:

1. `CompletableFuture<U> thenApply(Function<? super T, ? extends U> fn)`
2. `CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn)`
3. `CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn, Executor executor)`

Die an `thenApply` übergebene Funktion wird im Thread des »Vorgängers« ausgeführt, falls dieser das entsprechende `CompletableFuture` noch nicht abgeschlossen hat. Hier tritt die *Push*-Funktionalität zutage. Beim internen Abschluss eines `CompletableFuture` wird geprüft, ob eine weitere Aktivität angehängt ist. Falls ja, wird sie von demselben Thread bzw. von einem Thread aus dem Threadpool ausgeführt.

Die an `thenApplyAsync` übergebene Funktion wird immer asynchron durch einen separaten Thread ausgeführt.

3.3.2 Lineare Kompositionsmöglichkeiten

Über das `CompletionStage`-Interface lassen sich verschiedene Ablaufszenarien bilden. Für eine lineare Verknüpfung von Operationen stellt das `CompletionStage`-Interface verschiedene Methoden zur Verfügung.

Codebeispiel 3.3 zeigt ein Beispiel für eine einfache Sequenz, die mit `thenAcceptAsync` abgeschlossen wird[1].

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

Codebeispiel 3.3: Codeschema für eine asynchrone sequenzielle Aufrufkette

Die Methode `compose` kann benutzt werden, wenn eine Berechnung selbst ein `CompletableFuture`-Objekt zurückliefert und eigentlich mit dessen Wert weiter gerechnet werden soll (vgl. Abb. 3-4)[2]. Ohne die Methode `compose` würde man in dem Fall ein »geschachteltes« `CompletableFuture`-Objekt erhalten.

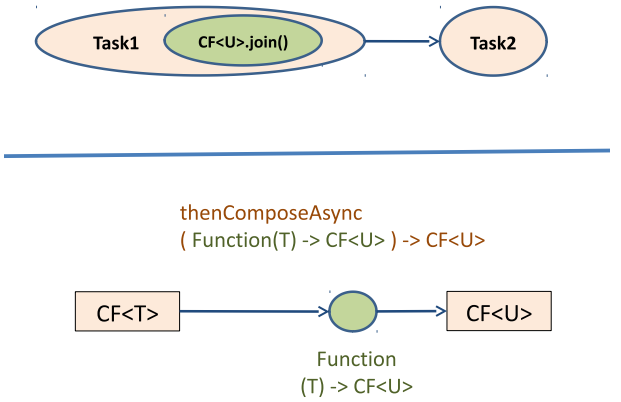


Abbildung 3-4: Komposition von zwei Tasks

3.3.3 Verzweigen und Vereinen

Neben den in Abschnitt 3.3.2 beschriebenen linearen Kompositionsmöglichkeiten enthält das `CompletionStage`-API noch Methoden zur Verzweigung und Vereinigung eines Verarbeitungsablaufs.

[1]Die Argumente der Methoden sind schematisch angegeben, d.h., die Ko- und Kontravarianzangaben sind weggelassen. Es gilt im Prinzip beim Input-Parameter immer `super` und beim Return-Typ `extends`.

[2]Die Methode `thenApply` entspricht eigentlich einer `map`-Operation und `compose` einer `flatMap`. Bei `Stream` wurden die Methoden auch so benannt, bei der Klasse `CompletableFuture` bedauerlicherweise nicht.

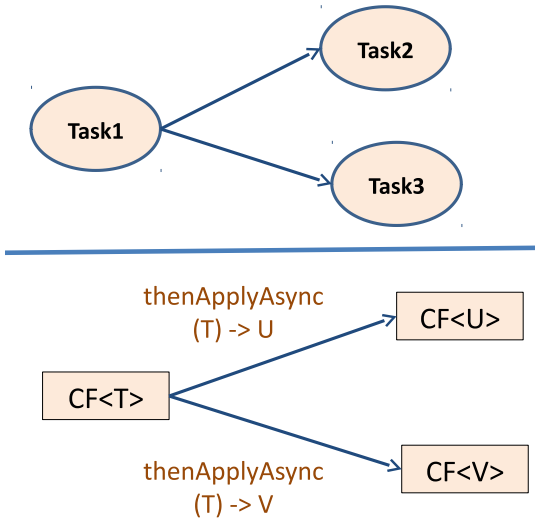


Abbildung 3-5: Aufteilen in zwei asynchrone Tasks (Split-Pattern)

Abbildung 3-5 zeigt schematisch die Aufteilung eines Tasks bzw. das Abspalten eines neuen asynchronen Tasks. Hierzu wird auf einem `CompletableFuture`-Objekt zweimal die `thenApplyAsync`-Methode aufgerufen. Codebeispiel 3.4 zeigt das zugehörige Idiom. Auf dem Objekt `cf` vom Typ `CompletableFuture` wird zweimal die `thenApplyAsync`-Methode aufgerufen, wobei jeder Aufruf der Beginn einer Verarbeitungskette darstellen kann.

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

cf.thenApplyAsync(  (E e) ->  T );
cf.thenApplyAsync(  (E e) ->  U );
```

Codebeispiel 3.4: Aufspaltung in zwei Folgetasks (Split-Pattern)

Die Vereinigung von zwei Verarbeitungsketten und deren Fortführung erfolgt durch die Methode `thenCombine`. Abbildung 3-6 zeigt schematisch den Vorgang. Hierbei wird eines der beiden `CompletableFuture`-Objekte als Argument der Methode `thenCombineAsync` benutzt, die noch zusätzlich einen Verknüpfungsoperator (`BiFunction`) erhält. Im Codebeispiel 3.5 ist das zugehörige Idiom abgebildet.

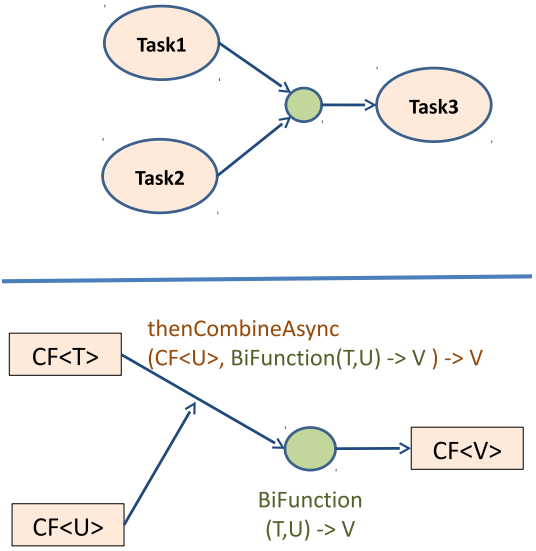


Abbildung 3-6: Zusammenführen von zwei Tasks

```
CompletableFuture<E> cf = CompletableFuture.supplyAsync( () -> E );
// split
CompletableFuture<T> task1 = cf.thenApplyAsync(  (E e) ->  T );
CompletableFuture<U> task2 = cf.thenApplyAsync(  (E e) ->  U );
// join
CompletableFuture<V> result =
              task1.thenCombineAsync(task2, (T t, U u) -> V );
```

Codebeispiel 3.5: Schema für das Zusammenführen von zwei nebenläufigen Tasks

Codebeispiel 3.6 zeigt das Zusammenführen zweier asynchroner Tasks, wobei der erste ein `String` und der zweite ein `Integer` als Ergebnis liefert. Die Resultate werden verkettet. Auf der Konsole wird hier `Hello 42` ausgegeben.

```
CompletableFuture<String>  task1
          = CompletableFuture.supplyAsync( () -> "Hello ");
CompletableFuture<Integer> task2
          = CompletableFuture.supplyAsync( () ->  42 );
CompletableFuture<String> result
          = task1.thenCombine(task2, (t,r) -> t + r );
System.out.println( result.join() );
```

Codebeispiel 3.6: Zusammenführen von zwei nebenläufigen Tasks

Die Methoden `thenAcceptBoth` und `runAfterBoth` arbeiten ähnlich wie `thenCombine`, nur dass anstatt eines `BiFunction`-Ausdrucks ein `BiConsumer`- bzw. ein `Runnable`-Ausdruck ausgeführt wird. Beide haben als Rückgabe ein `CompletableFuture<Void>`.

Die Methode `applyToEither` wird aufgerufen, wenn der erste der beiden vorherigen Tasks abgeschlossen ist. Sie entspricht somit einer ODER-Vereinigung. Der Wert des zuerst abgeschlossenen `CompletableFuture` wird der `Function` als Argument übergeben. Abbildung 3-7 zeigt den schematischen Ablauf.

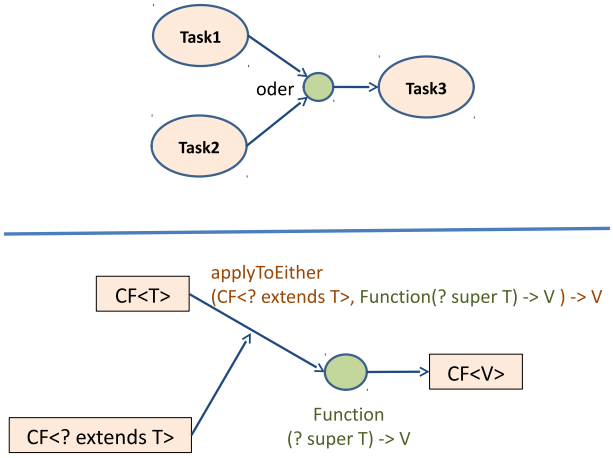


Abbildung 3-7: Zusammenführen von zwei Tasks durch ein ODER

3.3.4 Synchronisationsbarrieren

Es existieren noch die zwei statischen Methoden mit den Parametern `CompletableFuture<?>... cfs`

- `CompletableFuture<Void> allOf(...)`
- `CompletableFuture<Object> anyOf(...)`

mit denen beliebig viele asynchrone Abläufe koordiniert werden können. Mit `allOf` wird auf alle übergebenen Tasks gewartet, bis diese abgeschlossen sind. Sind sie beendet, dann ist auch das zurückgelieferte Objekt abgeschlossen.

Im Gegensatz zu `allOf` wird bei `anyOf` das zurückgelieferte Objekt abgeschlossen, wenn das erste der übergebenen `CompletableFuture` abgeschlossen ist. Das Ergebnis dieses Tasks kann über `get` bzw. `join` abgefragt werden, wobei die zurückgelieferte Referenz vom Typ `Objekt` ist und unter Umständen entsprechend gecastet werden muss.

In Abbildung 3-8 ist die Arbeitsweise der beiden Synchronisationsmethoden schematisch veranschaulicht. Links die von `allOf`, rechts die von `anyOf`.

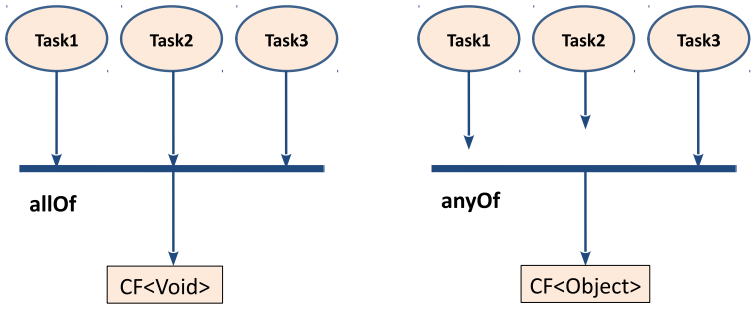


Abbildung 3-8: Die zwei Synchronisationsmöglichkeiten

3.4 Fehlerbehandlung und Abbruch einer Verarbeitung

Es bleibt nun noch die wichtige Frage: Was passiert, wenn bei der asynchronen Verarbeitung ein Fehler auftritt? Für den Umgang mit auftretenden Exceptions stehen im `CompletionStage`-Interface zwei Methoden zur Verfügung: `handle` oder `whenComplete` (vgl. Tab. 3-1). Auch für sie gibt es wieder die üblichen Varianten.

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

Tabelle 3-1: Methoden für die Fehlerbehandlung bei der Verwendung von `CompletableFuture`-Objekten. `CF` steht abkürzend für `CompletableFuture`.

Die grundlegende Idee ist, dass zur Behandlung einer ausgelösten Exception in der Verarbeitungskette nach einem entsprechenden Handler gesucht wird, der über `handle` oder `whenComplete` registriert ist. Abbildung 3-9 zeigt dies schematisch.

Da ein `CompletableFuture` auch das `Future`-Interface implementiert, steht dem Aufrufer auch die Methode `cancel(boolean mayInterruptIfRunning)` zur Verfügung. Bei gewöhnlichen `Future`-Objekten wird, wenn der Methode ein `true` übergeben wird, dem Thread, der den Task ausführt, ein `interrupt` gesendet.

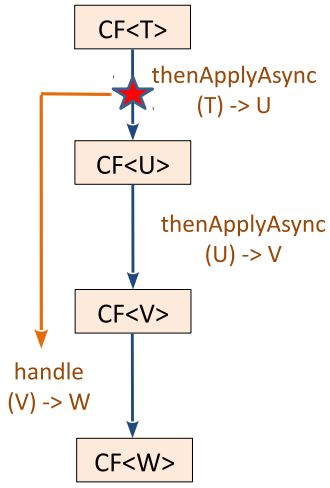


Abbildung 3-9: Verarbeitung einer aufgetretenen Exception

Die `cancel`-Methode beim `CompletableFuture` funktioniert anders. Auch wenn sie mit `true` aufgerufen wird, wird der ausführende Thread nicht unterbrochen. Alle anhängigen nicht abgeschlossenen `CompletableFuture`-Objekte erhalten eine `CancellationException` und werden hierdurch abgeschlossen (*complete exceptionally*). Die so ausgelöste `CancellationException` kann somit auch nicht mit `handle` oder `whenComplete` abgefangen werden.

4 Zusammenfassung

In dieser Broschüre haben wir uns lediglich auf die Threadpools und den Future-Mechanismus von Java konzentriert und das Thema »CompletableFutures« nur angerissen. Java bietet darüber hinaus noch weitere zahlreiche und interessante Abstraktionskonzepte für die nebenläufige Programmierung (siehe z.B. [2]).

Durch die Einführung von verschiedenen Lock-Klassen kann z. B. zwischen Schreib- und Lesesperren unterschieden werden und Locks können auch mehrere sogenannte Bedingungsvariablen zugeordnet werden. Durch die Verwendung von `StampLock` existiert auch die Möglichkeit, optimistisches Locking zu implementieren.

Mit den Atomic-Klassen wurde die Möglichkeit geschaffen, einzelne Variablen lockfrei und atomar zu ändern (*Compare-and-Set-Operation*). Hieraus ergibt sich die Option, lockfreie Datenstrukturen zu implementieren. So wurde folglich auch Java mit einigen sogenannten Concurrent-Collections ausgestattet.

Die Klassen `Semaphore`, `Exchanger`, `CountDownLatch`, `CyclicBarrier` und `Phaser` können zur Koordination mehrerer Threads genutzt werden. Hierdurch lassen sich z.B. Algorithmen, die Datenparallelität nutzen, einfacher implementieren. Die Aufteilung in Tasks und Ablaufkoordination müssen hier allerdings explizit implementiert werden.

Mit dem ForkJoin-Framework lassen sich rekursive Algorithmen bzw. Abläufe direkt und effizient parallelisieren. Der Entwickler muss hierzu lediglich entsprechende Task-Klassen zur Verfügung stellen, die von dem Framework für die Problemzerlegung und parallele Ausführung benutzt werden (*Divide-and-Conquer*).

Mit dem Stream-Konzept steht explizit die Möglichkeit der parallelen Collection-Verarbeitung zur Verfügung (Datenparallelität). Der Entwickler stellt hier lediglich nur noch Funktionsobjekte (Lambda-Ausdrücke) zur Verfügung, die durch interne Mechanismen auf die Stream-Elemente angewendet werden.

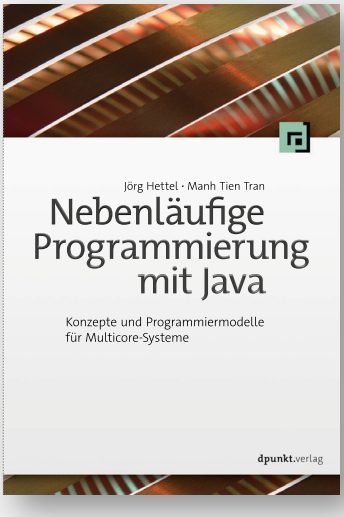
Literatur

[1] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes und D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[2] J. Hettel und M.T. Tran. *Nebenläufige Programmierung mit Java: Konzepte und Programmiermodelle für Multicore-Systeme*. dpunkt.verlag, 2016.

[3] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 2. Auflage, 1999.

[4] D. Schmidt, M. Stal, H. Rohnert und F. Buschmann. *Pattern-orientierte Software-Architektur. Muster für nebenläufige und vernetzte Objekte*. dpunkt.verlag, 2002.



2016,
378 Seiten, Broschur
€ 34,90 (D)
ISBN 978-3-86490-369-4

»Gute deutsche Bücher zur nebenläufigen Programmierung mit Java sind recht rar gestreut. Hier gilt das Buch von Oechsle als eines der Standardwerke. Das vorliegende Werk setzt einen etwas anderen Schwerpunkt und scheint für Einsteiger fast noch besser geeignet. Und es hat es nach Meinung des Rezensenten das Potenzial, ebenfalls in die Liga der Standardwerke vorzudringen.«

(Heise developer)

Jörg Hettel · Manh Tien Tran

Nebenläufige Programmierung mit Java

Konzepte und Programmiermodelle für Multicore-Systeme

Damit die Performance-Möglichkeiten moderner Multicore-Rechner effizient genutzt werden, muss die Software dafür entsprechend entworfen und entwickelt werden. Für diese Aufgabe bietet insbesondere Java vielfältige Konzepte an.

Das Buch bietet eine fundierte Einführung in die nebenläufige Programmierung mit Java. Der Inhalt gliedert sich dabei in fünf Teile: Im ersten Teil wird das grundlegende Thread-Konzept besprochen und die Koordinierung nebenläufiger Programmflüsse durch rudimentäre Synchronisationsmechanismen erläutert. Im zweiten Teil werden weiterführende Konzepte wie Threadpools, Futures, Atomic-Variablen und Locks vorgestellt. Ergänzende Synchronisationsmechanismen zur Koordinierung mehrerer Threads werden im dritten Teil eingeführt. Teil vier bespricht das ForkJoin-Framework, die Parallel Streams und die Klasse CompletableFuture, mit denen auf einfache Art und Weise nebenläufige Programme erstellt werden können. Im fünften Teil findet der Leser Beispiele für die Anwendung der vorgestellten Konzepte und Klassen. Dabei werden auch das Thread-Konzept von JavaFX und Android sowie das Programmiermodell mit Aktoren vorgestellt.

Der Anhang enthält einen Ausblick auf Java 9, das bezüglich des Concurrency-API kleine Neuerungen bringt. Alle Codebeispiele stehen auf der Webseite zum Buch zum Download bereit.

dpunkt.verlag
www.dpunkt.de

Softwarekonferenz für Parallel Programming, Concurrency, HPC und Multicore-Systeme

para//el 2017

Heidelberg, Print Media Academy, 29. – 31. März 2017

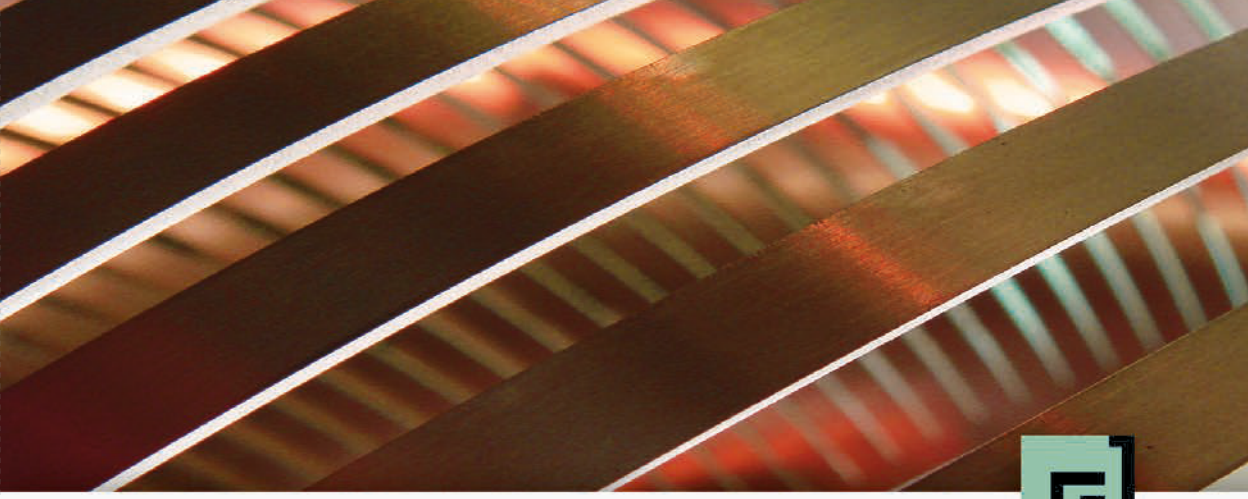
Die parallel 2017 ist das wichtigste deutsche Event zur Parallelprogrammierung. Die Konferenz bietet in ihrer sechsten Auflage erneut theoretisches Grundlagenwissen und Praxiserfahrungen zur Programmierung für Multicore- und Manycore-Architekturen – vorgestellt im Rahmen spezifischer Anwendungsfelder und Real-World-Szenarien.

Haben sich beim parallelen Rechnen die theoretischen Aspekte in den vergangenen Jahren gar nicht mehr so sehr verändert, ist hingegen gerade in jüngster Zeit die Durchdringung paralleler und nebenläufiger Programmierparadigmen in deutlich mehr Bereichen zu beobachten – so neben HPC- und GPU- auch in verteilten und Deep-Learning- bzw. Big-Data-Systemen, ganz zu schweigen von der Verbreitung von Multicores in Embedded-Systemen.

Teilnehmer der parallel 2017 bekommen wertvolle und praktische Ratschläge zum Einsatz von Produkten, Techniken und Mechanismen, mit denen sich das Potenzial unterschiedlicher Multicore-Plattformen besser ausschöpfen lässt.

www.parallelcon.de

weitere Veranstaltungen:
www.dpunkt.de/veranstaltungen.php



Jörg Hettel, Manh Tien Tran

Nebenläufige Programmierung mit Java

Damit die Performance-Möglichkeiten moderner Multicore-Rechner effizient genutzt werden, muss die Software dafür entsprechend entworfen und entwickelt werden. Für diese Aufgabe bietet insbesondere Java vielfältige Konzepte an.

Mit dieser Broschüre erhalten Sie einen Überblick über die Abstraktionskonzepte Executor, Future und CompletableFuture für die nebenläufige Programmierung in Java. Sie erfahren im Detail und anhand von vielen Codebeispielen, wie Sie Threadpools und den Future-Mechanismus von Java einsetzen, und lernen die mit Java 8 eingeführte CompletableFuture-Klasse kennen.

Die Broschüre basiert auf dem Buch »Nebenläufige Programmierung mit Java – Konzepte und Programmiermodelle für Multicore-Systeme« der Autoren (dpunkt.verlag, 2016).

»*Das Buch (...) hat wohl das Zeug dazu, in die Liga der Java-Standardwerke vorzudringen.*« (heise Developer)

http://www.hs-kl.de/java-concurrency

www.dpunkt.de