Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Nils Schnabl

Thread Pools und deren Umsetzung in der Programmiersprache Java

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Nils Schnabl

Thread Pools und deren Umsetzung in der Programmiersprache Java

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Schäfers
Zweitgutachter: Prof. Dr. Axel Schmolitzky

Abgegeben am 22.08.2019

Nils Schnabl

Thema der Arbeit

Thread Pools und deren Umsetzung in der Programmiersprache Java

Stichworte

Task, Runnable, Callable, ThreadPoolExecutor, ForkJoin-Framework, Executors

Kurzzusammenfassung

Ziel dieser Abschlussarbeit ist das Thema Thread Pools und deren Umsetzung in der Programmiersprache Java zu beleuchten. Theoretisch werden viele Grundbegriffe und weitergehende Inhalte allgemein sowie Java-spezifisch im Kapitel Einführung und Grundlagen erklärt. Im weiteren Verlauf werden dann im Kapitel 3 die weiteren Details Java-spezifisch erklärt. Kapitel 4 enthält die Praxisbeispiele zum Thema Executor-Interface und der Executors-Klasse.

Nils Schnabl

Title of the paper

Thread pools and their implementation in the programming language Java

Keywords

Task, Runnable, Callable, ThreadPoolExecutor, ForkJoin-Framework, Executors

Abstract

The aim of this thesis is the topic of thread pools and its implementation in the Java programming language. Theoretically, many basic terms and further contents are described in general as well as Java-specific in the chapter Introduction and Basics. In the further course, the details will be explained Java-specific in Chapter 3. Chapter 4 contains the practical examples on the subject of the Executor-Interface and the Executors class, which are discussed in more detail.

Inhaltsverzeichnis

1 Einführung

1.1 Thread

In der Informatik ist ein Thread die kleinste Folge programmierter Anweisungen, z.B. ein Ausführungsstrang oder eine Ausführungsreihenfolge in der Abarbeitung eines Programms. Die Threads werden von einem Scheduler, der normalerweise ein Teil des Betriebssystems ist, unabhängig verwaltet. In den meisten Fällen ist ein Thread Bestandteil eines Prozesses. Innerhalb eines Prozesses können mehrere Threads vorhanden sein, die gleichzeitig ausgeführt werden und Ressourcen wie den Arbeitsspeicher gemeinsam nutzen. Insbesondere teilen sich die Threads eines Prozesses zu jeder Zeit ihren ausführbaren Code sowie die Werte seiner dynamisch zugewiesenen Variablen und nicht-lokalen Variablen [13].

Die Abbildung 1.1 zeigt einen Prozess mit zwei Ausführungsthreads, der auf einem Prozessor ausgeführt wird.

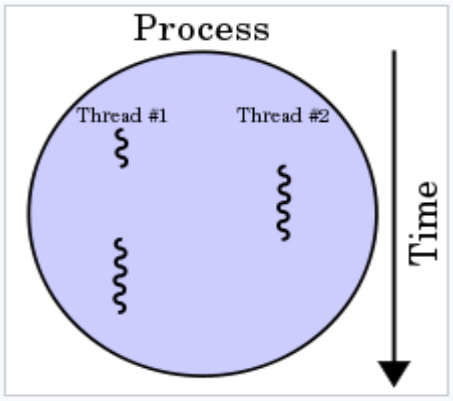


Abbildung 1.1: Prozess mit zwei Ausführungsthreads

1.2 Task

Ein „Task“ ist ein Arbeitsauftrag, der von einem konkreten Thread ausgeführt wird. Zu den alternativen Begriffen gehören unter anderem der Prozess und der Thread zur Ausführung. Die Arbeitseinheiten selbst oder die Threads, die die Arbeit ausführen, können als „Tasks“ bezeichnet werden. Im weiteren Verlauf wird von „Task“ gesprochen, wenn dieser als Arbeitseinheit/Arbeitsauftrag gemeint ist.

1.3 Basic Thread Management

Es gibt vier grundlegende Thread-Management-Operations: Thread-Creation, Thread-Termination, Thread-Join und Thread-Yield [6].

1.3.1 Thread Creation

Die Abbildung 1.2 zeigt einen übergeordneten Thread A und untergeordnete Threads [6].

Grundsätzlich können wir den Ausführungsthread in mehrere Teile splitten. So werden später im Verlauf mehrere Threads gleichzeitig ausgeführt. Der Erzeugende (parent thread) ist der übergeordnete Thread und die Erstellten (child threads) sind untergeordnete Threads. Jeder Thread, einschließlich des Hauptprogramms, welches beim Start ausgeführt wird, kann untergeordnete Threads erstellen.

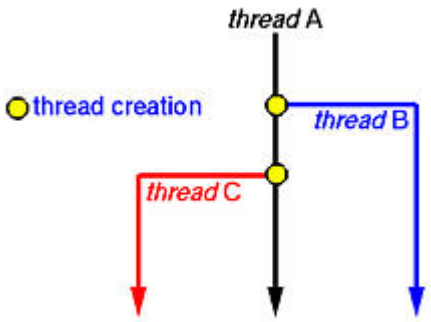


Abbildung 1.2: Thread Creation

1.3.2 Thread Termination

In den meisten Fällen werden Threads nicht ewig ausgeführt. Nach Beendigung der Arbeit werden die Threads beendet (sie terminieren) [6]. Die untergeordneten Threads teilen Ressourcen mit dem übergeordneten Thread, einschließlich Variablen. Deshalb können diese untergeordneten Threads terminieren, sobald der übergeordnete Thread sich beendet. Denn, wenn der übergeordnete Thread beendet

wird, können alle seine Variablen verloren gehen, so dass die untergeordneten Threads nicht auf die gemeinsamen Ressourcen zugreifen können. In Java können diese untergeordneten Threads (child threads) ihre Eltern überleben.

1.3.3 Thread Join

Ein Thread kann einen Thread-Join ausführen, um zu warten, bis ein anderer Thread beendet ist [6]. Zum Beispiel sollte der Haupt-Thread auf einen Thread-Join warten, bis ein untergeordneter Thread beendet wird. Im Allgemeinen ist der Thread-Join sinnvoll für einen übergeordneten Thread, der sich mit einem seiner untergeordneten Threads verbinden will.

1.3.4 Thread Yield

Thread Yield ist von Bedeutung, wenn beispielsweise mehrere Programme gleichzeitig auf einem Computer ausgeführt werden [6]. So ist es möglich, dass die Threads einiger Programme die CPU-Zyklen vollständig auslasten. Dieses bedeutet, dass andere Programme nicht effektiv laufen können, sie können verhungern (starvation). Dies kann ein Problem der Zeiteinteilung des Betriebssystems sein. Wenn man Programme mit mehreren Threads schreibt, muss sichergestellt werden, dass einige Threads die CPU nicht für immer oder für eine sehr lange Zeit belegen. Es kann die Situation eintreten in der ein oder zwei Threads weiterlaufen, während die Anderen einfach auf ihre Zeitslots warten. Das bedeutet, diese Threads werden auf eine sehr “freundliche” Weise ausgeführt, so dass ein Thread gelegentlich eine Pause einlegt, damit die CPU von anderen Threads verwendet werden kann. Dieses wird erreicht durch Thread Yield.

1.4 Thread Life Cycle Overhead

Als Overhead gelten in der elektronischen Datenverarbeitung Daten als auch Aufwand, die nicht primär zu den Nutzdaten zählen, sondern als Zusatzinformation zur Übermittlung oder Speicherung benötigt werden [4]. Dazu zählt aus dem Thread-Pool-Bereich z.B. der zusätzliche Aufwand durch die Erzeugung vieler Threads. Der Thread Life Cycle Overhead sind hierbei, die zur Erzeugung von einzelnen Threads benötigten Ressourcen. Genauer sind mit Ressourcen die übermäßige oder indirekte Rechenzeit, der Speicher oder andere Ressourcen, die zur Ausführung einer bestimmten Aufgabe erforderlich sind, gemeint. Diese zusätzlichen Merkmale (Zusatzinformationen zur Erzeugung einzelner Threads), die zur Erfüllung von Aufgaben erforderlich sind, werden als Thread Life Cycle Overhead bezeichnet.

1.5 Thread Pools

1.5.1 Motivation “für” Thread Pools

Bei Verwendung eines einfachen Modells zum Bauen einer Serveranwendung wird ein neuer Thread erzeugt, wenn eine Anforderung/Anfrage ankommt. Einen Ansatz dieser Art für eine Serveranwendung bereitzustellen hat signifikante Nachteile. Es wäre ein Worst-Case-Szenario, wenn für jede Anfrage ein neuer Thread erzeugt würde. Dies verbräuchte mehr Zeit und Systemressourcen, um Threads zu erzeugen und zu zerstören, als tatsächliche Benutzeranfragen zu bearbeiten. Zusätzlich zu diesem Overhead verbrauchen aktive Threads auch noch Systemressourcen. Das übermäßige Erzeugen von vielen Threads in einer Java Virtual Machine (JVM), kann unter anderem zu einem übermäßigen Speicherverbrauch führen. Um eine solche Ressourcenverschwendung zu vermeiden, benötigen solche Serveranwendungen die Möglichkeit, die Anzahl der Anfragen zu jedem Zeitpunkt zu limitieren.

1.5.2 Thread Pools als Lösung

Mit Hilfe von Thread Pools wird eine Lösung zur Reduzierung von Ressourcenverschwendung und eines Thread Life Cycle Overhead angeboten. Durch die Wiederverwendung von Threads für mehrere Tasks wird der Thread-Lifecycle-Overhead auf viele Tasks verteilt. Vereinfacht heißt dieses, dass die Threads bereits existieren und nicht neu erzeugt werden müssen. Die Tasks können beispielsweise sofort mit einem bereits existierenden Thread bearbeitet werden. Der Delay der Thread-Erzeugung wird somit eliminiert. Folgend kann die Anfrage ohne Verzögerung bedient werden. Weiterhin kann die Anzahl der Threads im Thread Pool richtig abgestimmt werden, so dass eine Ressourcenverschwendung verhindert wird.

Die Größe eines Thread-Pools ist die Anzahl der Threads, die für die Ausführung von Tasks reserviert sind. Dies ist normalerweise ein einstellbarer Parameter der Anwendung, der zur Optimierung der Programmleistung angepasst wird [8].

1.5.3 Erläuterung/Definition von Thread Pools

Ein Thread Pool ist ein Software Design Pattern [8] zum Erreichen einer Parallelität der Exekution (Ausführung). Es wird häufig als „Replicated Workers“ oder „Worker-Crew“ Model bezeichnet. Ein Thread Pool ist daher eine verwaltete Sammlung von Threads, die verfügbar sind Tasks auszuführen. Thread Pools bieten normalerweise:

- Eine verbesserte Leistung bei der Ausführung einer großen Zahl von Tasks durch einen reduzierten pro Task Aufruf Overhead.

- Ein Mittel zur Begrenzung der Ressourcen, einschließlich Threads, welche verbraucht werden, wenn eine Sammlung von Aufgaben ausgeführt wird.
- Die Anzahl der verfügbaren Threads ist auf die für das Programm verfügbaren Rechenressourcen abzustimmen, wie parallele Prozessoren, Cores, Speicher und Netzwerk-Sockets.

Es ist nicht Ziel des Thread Pools Rechenkerne ungenutzt zu lassen. Zusätzlich befreien Thread Pools von dem Verwalten des Lebenszyklusses von Threads. Diese Thread Pools erlauben, den Vorteil des Threading zu nutzen. So kann der Programmierer sich auf die Tasks (Aufgaben) konzentrieren, die die Threads ausführen sollen, anstelle sich mit der Thread-Problematik zu befassen.

1.5.4 Performance von Thread Pools

Die Anzahl der Threads kann während der Lebensdauer einer Applikation, basierend auf der Anzahl wartender Tasks, dynamisch angepasst werden. Ein Webserver kann Threads hinzufügen, wenn zahlreiche Webseitenanforderungen eingehen und ebenfalls Threads entfernen, wenn diese Anforderungen nachlassen. Ein größerer Thread-Pool erhöht die Ressourcennutzung. Der Algorithmus mit dem bestimmt wird wann Threads erstellt oder gelöscht werden, schlägt sich auf die Gesamtleistung durch.

Nachfolgend werden drei wichtige Punkte zur Performance aufgeführt [8]:

- Werden zu viele Threads erstellt, ist dies eine Verschwendung von Zeit und Ressourcen.
- Das Zerstören von zu vielen Threads erfordert nachfolgend, bei einer erneuten Erstellung, mehr Zeit.
- Wenn Threads zu langsam erstellt werden, kann dies auf der Client-Seite zu längeren Wartezeiten führen.

1.5.5 Wann sollten keine Thread Pools verwendet werden?

- Sind genügend Prozessor-/Rechenkerne für die gestarteten Threads vorhanden, sind Thread Pools als Lösung nicht notwendig.
- Zählt nur das "Endergebnis" und bricht das System nicht unter Last der Threads (Speicher, Thread-Wechsel) zusammen, sind Thread Pools ebenfalls nicht notwendig.

1.5.6 keepAliveTime (ThreadPoolExecutor)

Wenn der Pool mehr Threads enthält, als die Core-Pool-Größe zur Verfügung stellt (siehe Absatz 1.5.7), werden überschüssige Threads beendet, wenn sie länger als die

Keep-Alive-Time im Leerlauf waren. Auf diese Weise können Sie den Ressourcenverbrauch reduzieren. Wenn die Core-Pool-Größe nicht ausreicht, werden zu einem späteren Zeitpunkt neue Threads erstellt.

1.5.7 Core- und Maximum-Pool Size

Die Core-Pool-Größe, die maximale Poolgröße und die Keep-alive-Time bestimmen die Erstellung und den Abbau von Threads. Die Core-Pool-Größe (Core-Pool Size) ist die Zielgröße. Die Implementierung versucht, den Pool auf dieser Größe zu halten, auch wenn keine Aufgaben ausgeführt werden müssen. Es werden nur dann mehr Threads erstellt, wenn die Working Queue voll ist. Die maximale Poolgröße (Maximum-Pool Size) ist die Obergrenze für die Anzahl der Pool-Threads, die gleichzeitig aktiv sein können. Ein Thread, der länger als die Keep-alive-Time im Leerlauf war, ist ein Kandidat, der beendet wird, wenn die aktuelle Pool-Größe die Core-Pool-Größe überschreitet.

1.5.8 Varianten von Thread Pools

Es gibt Thread Pools …

- die so viele Threads wie benötigt erzeugen.
- die die Anzahl parallel laufender Threads begrenzen.
- die nur einen Thread zurzeit zu lassen.
- die die verzögerte und periodische Ausführung von Aufgaben bei einer begrenzten Anzahl von parallel laufender Threads unterstützen.
- die das Work-Stealing-Verfahren für das ForkJoin-Pattern realisieren (siehe Absatz 3.2.4).

2 Grundlagen

In diesem Kapitel werden einige wichtige Fachwörter eingeführt und erklärt.

2.1 Worker Thread

Ein Worker-Thread arbeitet für das System in dem er sich befindet. Dieser besondere Thread kann beispielsweise durch Clientanforderungen aktiviert werden.

Zum Beispiel:

- Ihr Server überwacht einen Port.
- Eine Anforderung kommt an diesem Port an.
- Ein Listener-Thread[1] nimmt diese Anforderung an und sendet sie an einen Worker-Thread, der die Anforderung abschließt.

Wenn zwei Client-Anforderungen gleichzeitig eingehen, werden zwei Worker-Threads zugewiesen und die Tasks werden gleichzeitig ausgeführt.

2.2 Single Threading

Single-Threading bedeutet die Verarbeitung von einem Befehl nach dem anderen zu jeweils einem Zeitpunkt. Das Gegenteil von Single-Threading ist Multithreading.

2.3 Multithreading

Multithreading wird hauptsächlich in Multitasking-Betriebssystemen gefunden. Multithreading ist ein weit verbreitetes Programmier- und Ausführungsmodell, mit dem mehrere Threads im Kontext eines Prozesses vorhanden sind. Diese Threads verwenden die Ressourcen des Prozesses gemeinsam, können jedoch unabhängig voneinander ausgeführt werden. Das Threading-Programmiermodell bietet Entwicklern eine nützliche Abstraktion der gleichzeitigen Ausführung. Multithreading kann auch auf einen oder mehrere Prozesse angewendet werden, um die parallele Ausführung auf einem Multiprozessorsystem zu ermöglichen. In der JVM (Java Virtual Machine) gibt es nur einen Prozess.

[1] Der Listener-Thread ist ein System-Thread, der die Aufgabe eines Listeners übernimmt.

2.4 Task Queue

Üblicherweise ist eine Task Queue eine Datenstruktur, die von der Task-Scheduling-Software verwaltet wird und die auszuführenden Tasks enthält. Später bei den Thread Pools findet man die Task Queues wieder. Benutzer senden ihre Programme, die sie ausführen möchten (Tasks) zur Verarbeitung in diese Warteschlange (Task Queue). Diese Task Queue wird für die Tasks genutzt, die bewältigt werden müssen. Der Scheduler kann mehrere dieser Warteschlangen verwenden, um die Tasktypen in Abhängigkeit von folgenden Parametern zu unterscheiden:

- job priority
- geschätzte Execution Time
- Ressourcenanforderungen

2.5 Hooks

Der Begriff "Hooking" umfasst eine Reihe von Techniken die verwendet werden, um das Verhalten von komplexen Systemen bzw. Anwendungen oder anderer Softwarekomponenten zu kontrollieren, indem Funktionsaufrufe oder Nachrichten oder Ereignisse abgefangen werden, die zwischen Softwarekomponenten übertragen werden. Code, der solche abgefangenen Funktionsaufrufe, Ereignisse oder Nachrichten behandelt, wird als Hook bezeichnet.

2.6 Wichtige Java Interfaces/Klassen

2.6.1 Interface Runnable

Das Runnable-Interface sollte von jeder Klasse, die bereits von einer anderen abgeleitet wurde und als eigener Thread (parallel zu anderen Klassen) laufen soll, implementiert werden. Die Klasse muss die run()-Methode ohne Argumente überschreiben. Runnable wird von der Klasse Thread implementiert. Das Runnable-Interface ist deshalb die primäre Vorlage für jedes Objekt, dass von einem Thread ausgeführt werden soll. In den meisten Fällen sollte das Runnable-Interface verwendet werden, wenn nur die run() -Methode und keine anderen Thread-Methoden überschrieben werden sollen.

2.6.2 Klasse Thread

Die “Java-Class Thread” *implements* Runnable.

Die Klasse Thread ist ein Thread zur Ausführung in einem Programm. Die Java Virtual Machine ermöglicht einer Anwendung die gleichzeitige Ausführung mehrerer Ausführungsthreads. Es gibt zwei Möglichkeiten, einen neuen Ausführungsthread zu erstellen. Eine Möglichkeit ist, eine Klasse als eine Unterklasse von Thread zu deklarieren. Diese Unterklasse muss die run()-Methode der Klasse Thread überschreiben. Eine Instanz der Unterklasse kann dann zugewiesen und gestartet werden.

2.6.3 Interface Callable<V>

Dieses Interface ist ein Task, der ein Ergebnis zurückgibt und möglicherweise eine Exception auslöst. Programmierer definieren eine einzelne Methode ohne Argumente, genannt call(). Das Callable-Interface ähnelt Runnable, da beide für Klassen konzipiert sind, deren Instanzen möglicherweise von einem anderen Thread ausgeführt werden. Ein Runnable liefert jedoch kein Ergebnis und kann keine geprüften Exceptions auslösen. Die Executors-Klasse enthält Utility-Methoden zum konvertieren von anderen gebräuchlichen Formen in Callable Klassen [12].

2.6.4 Interface Future<V>

Ein Future repräsentiert das Ergebnis einer asynchronen Verarbeitung. Es werden Methoden bereitgestellt um zu überprüfen, ob die Verarbeitung abgeschlossen ist, sowie auf deren Abschluss zu warten und dann das Ergebnis der Berechnung abzurufen (genaueres folgt in Kapitel 3.3).

2.6.5 Varianten der Task-Übergabe

Die folgende Tabelle 2.1 zeigt eine Methodenübersicht von run(), call() und execute():

run()	**Interface Runnable**	Wenn ein Objekt verwendet wird, das die Schnittstelle Runnable implementiert, um einen Thread zu erstellen, wird beim Starten des Threads die *run()*-Methode des Objekts in dem separat ausgeführten Thread aufgerufen [12].
call()	**Interface Callable<V>**	Berechnet ein Ergebnis oder löst eine Exception aus, wenn eine Ergebnisrückgabe nicht möglich ist [12].
execute(Runnable command)	**Interface Executor**	Führt den angegebenen Befehl zu einem späteren Zeitpunkt aus [12].

Tabelle 2.1: Methodenübersicht

2.7 Die wichtigsten Ideen zu Thread Pools im Detail

Die Abbildung 2.1 zeigt einen Beispiel-Thread-Pool (hell-grüne Kästchen) mit den wartenden Aufgaben/Task Queue (hell-lila-blau) und den abgeschlossenen Aufgaben/Completed Tasks (gelb) [8].

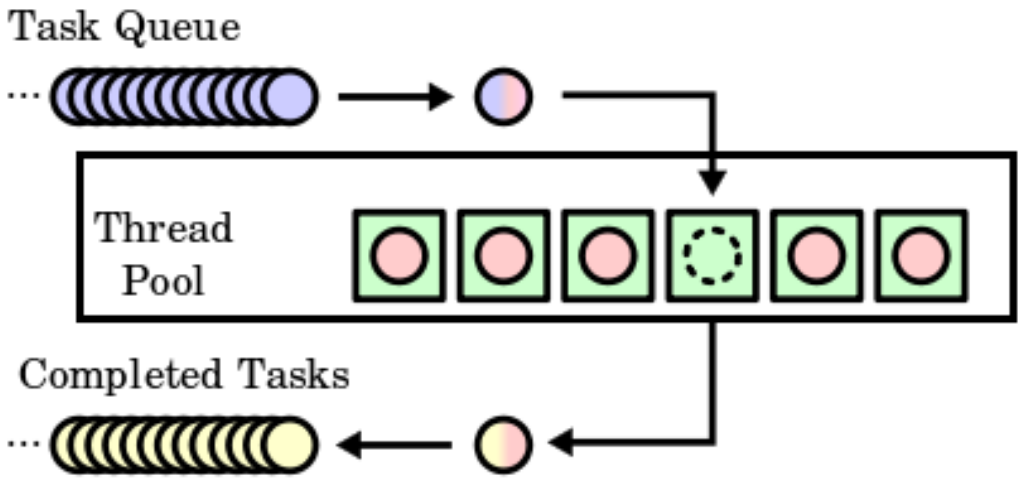


Abbildung 2.1: Beispiel-Thread-Pool

Es folgt eine Auflistung zu Thread Pools im Detail:

- Ein Thread-Pool dient der Abarbeitung von Tasks, in der Regel aus einer Warteschlange heraus, durch einen Pool aus Threads. Der Thread-Pool wird hierbei von einem Scheduler verwaltet [16].
- Eine Anzahl von Worker-Threads (also der Thread Pool) steht im Thread Pool bereit und wartet darauf Tasks abzuarbeiten.
- Die einzelnen Tasks werden in geeigneter Form (z.B. Runnable) dem Thread-Pool übergeben.
- Die Tasks werden in einer Work-Queue organisiert und warten dort auf ihre Bearbeitung.
- Ein unbeschäftigter Worker-Thread greift sich den Task und arbeitet ihn ab.
- Nach der Abarbeitung eines Tasks wartet der nun freie Thread im Thread-Pool wieder auf neue Tasks, die abgearbeitet werden sollen.
- Die Threads werden somit recycelt und nicht vernichtet, um dann später wieder neu eingerichtet zu werden.

2.8 Task-Submission

Task-Submission bedeutet unter anderem, eine Kontrolle über Tasks auf einem System zu haben. Ein Endbenutzer in einem Time-Sharing-System könnte einen Task

interaktiv von seinem Remote-Terminal aus übermitteln (submit) und mit den Bedienern kommunizieren.

2.9 Die Execution Policy

Die Execution Policy bestimmt/beschreibt folgende Punkte genauer [1]:

- Die Reihenfolge des Anstartens der Tasks z.B. als FIFO oder LIFO wird gesteuert.
- Die Anzahl der parallel abgearbeiteten Tasks/Threads wird festgelegt.
- Die Festlegung von einer maximalen Anzahl wartender Tasks, die auf ihre Abarbeitung warten, bevor eine weitere Annahme verweigert wird.
- Wenn die Last zu groß ist, muss die Entscheidung welche Tasks verworfen werden können, inklusive einer Benachrichtigung an eben diese Tasks, gewährleistet sein.

Das Festlegen was vor und/oder nach der Bearbeitung eines Tasks getan werden soll, ist ebenfalls Teil der Execution Policy.

3 Thread Pools unter Java

Die Klassenbibliothek von Java stellt flexible Thread Pool Implementierungen bereit. Damit können einige nützliche vordefinierte Konfigurationen eingestellt werden. Ein Thread Pool kann, indem eine der statischen Factory Methods von Executors aufgerufen wird, erzeugt werden [1, 2].

3.1 Thread – Pool – Arten in Java

Um Thread Pools zu benutzen, wird eine Implementation des ExecutorService-Interface instanziiert und eine Reihe von Aufgaben werden übergeben. Die Wahlmöglichkeiten von konfigurierbaren Thread Pool Implementationen sind der ForkJoinPool, der ThreadPoolExecutor und der ScheduledThreadPoolExecutor. Diese Implementationen erlauben die Core- und Maximum-Pool Size festzulegen, die benötigte Art der Datenstruktur zu nutzen und wie mit abgelehnten Aufgaben umzugehen ist. Die Art und Weise wie Threads erzeugt und terminiert werden, kann ebenfalls eingestellt werden.

Es wird empfohlen, die bequemeren Factory-Methoden der Klasse Executors zu verwenden (genaueres folgt in Kapitel 3.4). Mit diesen Methoden werden Einstellungen für die häufigsten Anwendungsszenarien vorkonfiguriert.

Es wurden zwei Richtlinien zum Ausführen von Tasks untersucht. Die eine Richtlinie führt Tasks sequenziell in einem einzelnen Thread aus und die andere Richtlinie führt jeden Task in einem eigenen Thread aus. Beide haben ernsthafte Einschränkungen:

- Der “sequentielle” Ansatz leidet unter “schlechter Reaktionsfähigkeit” und “schlechtem Durchsatz”.
- Der “Thread-per-Task”-Ansatz leidet unter mangelhafter Ressourcenverwaltung.

3.2 Executor-Interface, ExecutorService

Die folgende Abbildung 3.1 zeigt die Klassenhierarchie des Poolkonzeptes:

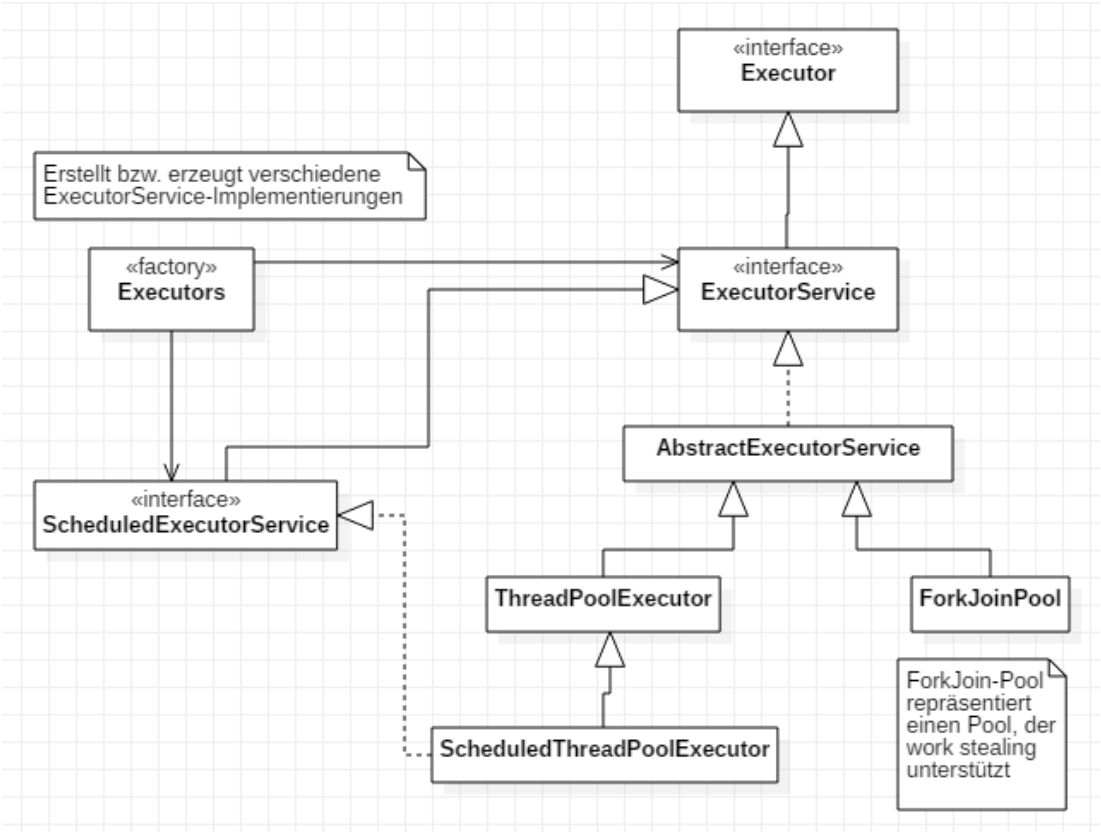


Abbildung 3.1: Klassenhierarchie des Poolkonzeptes

3.2.1 Das Executor-Interface und ExecutorService

Wie bereits ausgeführt, sind Tasks logische Arbeitseinheiten und Threads sind ein Mechanismus mit denen Tasks asynchron laufen können [1].

Der Executor kann eine einfache Schnittstelle (Interface) sein, bildet aber auch die Basis für ein flexibles und leistungsfähiges Framework für asynchrone Aufgaben, die eine Vielzahl von Execution Policies für eben diese Ausführung von Aufgaben unterstützt. Dieses Interface stellt ein „Standardmittel“ zur Entkopplung von „Task Submission“ von „Task Execution“ zur Verfügung, wobei Tasks mit „Runnable“ beschrieben werden. Die Executor-Implementierungen bieten auch einen Lifecycle Support an und zusätzlich auch noch Hooks (Haken) für das Hinzufügen von Statistiken und das Sammeln und Verwalten von Anwendungen.

3.2.2 Der ThreadPoolExecutor

Der ThreadPoolExecutor ist ein ExecutorService, der jede übermittelte Task unter Verwendung eines von möglicherweise mehreren zusammengefassten Threads ausführt, die normalerweise mit den Factory-Methoden von Executors konfiguriert werden [12].

Thread-Pools betreffen zwei unterschiedliche Probleme: Sie bieten normalerweise eine verbesserte Leistung, wenn eine große Anzahl asynchroner Aufgaben ausgeführt wird, da der Aufwand für das Aufrufen pro Task geringer ist. Außerdem bieten sie die Möglichkeit, Ressourcen und Threads, die beim Ausführen einer Auflistung verwendet werden, zu begrenzen und Aufgaben zu verwalten. Jeder ThreadPoolExecutor verwaltet außerdem einige grundlegende Statistiken, z. B. die Anzahl der erledigten Aufgaben.

Der ThreadPoolExecutor stellt die Basisimplementierung für die Factory Executors bereit [1], die u. a. von den Factories *newCachedThreadPool()*, *newFixedThreadPool()* und *newScheduledThreadExecutor()* in Executors zurückgegeben werden.

ThreadPoolExecutor ist eine flexible, robuste Pool-Implementierung, die eine Vielzahl von Anpassungen ermöglicht. Wenn die Standard (default) Execution Policy nicht ihren Anforderungen entspricht, kann ein ThreadPoolExecutor über seinen Konstruktor instanziiert und nach Bedarf anpasst werden. Man kann den Quellcode für Executors heranziehen, um die Execution Policies für die Standardkonfigurationen anzuzeigen und sie als Ausgangspunkt zu verwenden. Der ThreadPoolExecutor verfügt über mehrere Konstruktoren, von denen der allgemeinste Konstruktor in der folgenden Abbildung 3.2 gezeigt wird:

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

Abbildung 3.2: Allgemeinster Konstruktor für den ThreadPoolExecutor

Die Core-Pool-Größe, die maximale Pool-Größe und die Keep-alive-Time bestimmen das Erstellen und Herunterfahren von Threads [1]. Die Core-Pool-Größe ist hierbei die Zielgröße. Die Implementierung versucht, den Thread Pool auf dieser Größe zu halten, auch wenn keine auszuführenden Tasks vorhanden sind. Außerdem werden nur Threads erstellt, wenn die Work Queue voll ist. Die maximale Pool-Größe ist die Obergrenze dafür, wie viele Pool-Threads gleichzeitig aktiv sein können. Ein Thread, der länger als die Keep-alive-Time inaktiv war, kann beendet werden, wenn die aktuelle Pool-Größe die Core-Pool-Größe überschreitet.

Indem man die Core-Pool-Größe und die Keep-alive-Time optimiert, kann der Pool dazu befähigen Ressourcen freizugeben, die von ansonsten inaktiven Threads verwendet werden und sie für nützlichere Arbeiten verfügbar machen [1].

Die *newFixedThreadPool()*-Factory legt sowohl die Core-Pool-Größe als auch die maximale Pool-Größe für die angeforderte Pool-Größe fest, wodurch der Effekt einer unbegrenzten Zeitüberschreitung entsteht [1]. Die *newCachedThreadPool()*-Factory setzt die maximale Pool-Größe auf *Integer.MAX_VALUE* und die Core-Pool-Größe auf *Null* mit einer Zeitüberschreitung von einer Minute. Dadurch entsteht ein unendlich erweiterbarer Thread-Pool, der sich bei sinkender Nachfrage wieder zusammenzieht. Auch andere Kombinationen sind mit diesem eindeutigen ThreadPoolExecutor-Konstruktor möglich.

3.2.3 Der ScheduledThreadPoolExecutor

Der ScheduledThreadPoolExecutor kann zusätzlich Befehle einplanen, die nach einer bestimmten Verzögerung oder periodisch ausgeführt werden. Diese Klasse ist dem Timer vorzuziehen, wenn mehrere Arbeitsthreads erforderlich sind oder wenn die zusätzliche Flexibilität oder die zusätzlichen Funktionen von ThreadPoolExecutor (um die diese Klasse erweitert wird) erforderlich sind [12].

Verzögerte Tasks werden frühestens ausgeführt, wenn sie aktiviert sind, aber ohne Echtzeitgarantie. Tasks, die für genau dieselbe Ausführungszeit geplant sind, werden in der FIFO-Reihenfolge der Übermittlung (First-In-First-Out) aktiviert.

3.2.4 Das ForkJoin-Framework

Dieses Framework kann insbesondere für die Parallelisierung von Divide-and-Conquer-Algorithmen eingesetzt werden [2]. Es verwendet intern einen Threadpool, der ein Work-Stealing-Verfahren implementiert. Dieses sorgt dafür, dass die verfügbaren Ressourcen entsprechend ausgenutzt werden. Das ForkJoin-Framework setzt so das ForkJoin-Pattern, welches in Abbildung 3.3 dargestellt ist, um.

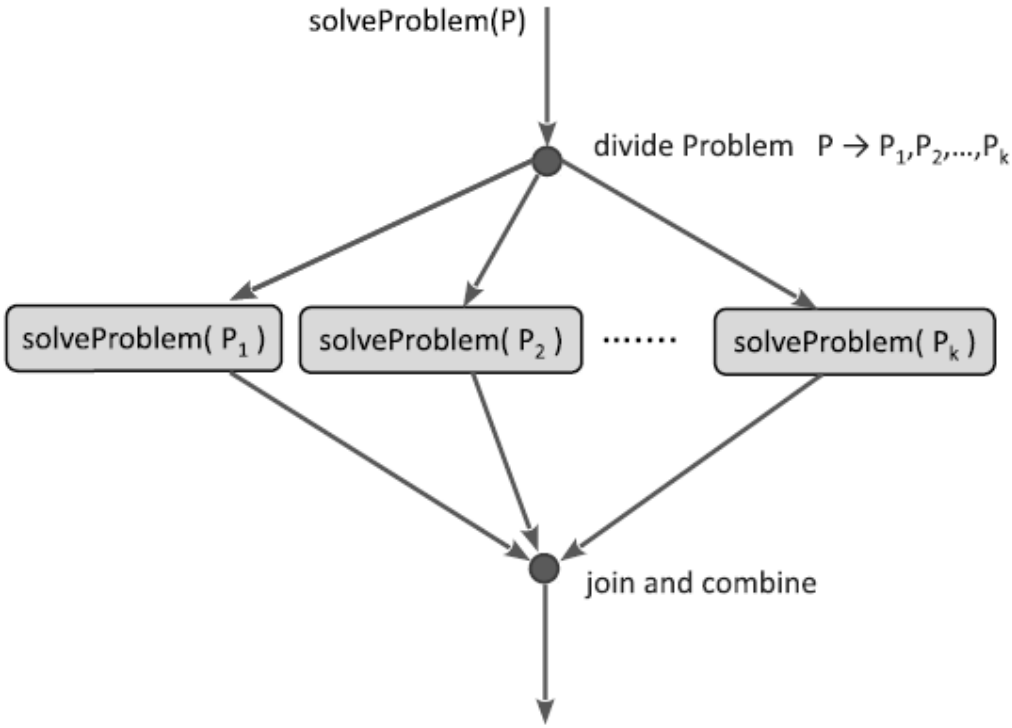


Abbildung 3.3: Der Kontrollfluss des ForkJoin-Patterns

Das Grundprinzip des ForkJoin-Patterns [2]:

Beim ForkJoin-Pattern wird der Kontrollfluss an einer dedizierten Stelle in mehrere nebenläufige Flüsse aufgeteilt (fork), die an einer späteren Stelle alle wieder vereint (join) werden (siehe Abbildung 3.3). Die Vereinigung entspricht einem Synchronisationspunkt. Wenn alle Teilaufgaben erledigt sind, wird das Programm danach fortgesetzt. Die Stärke bzw. die eigentliche Anwendung des ForkJoin-Patterns tritt bei der Umsetzung rekursiver Divide-and-Conquer-Algorithmen zutage.

Die folgende Abbildung 3.4 zeigt schematisch die rekursive Verzweigungs- und Vereinigungsstruktur. In der ersten Phase wird das Problem immer wieder zerkleinert (Divide-Phase). Ist eine entsprechende Problemgröße erreicht, werden die Teilaufgaben gelöst (Work-Phase) und anschließend das Ergebnis zusammengesetzt (Combine-Phase).

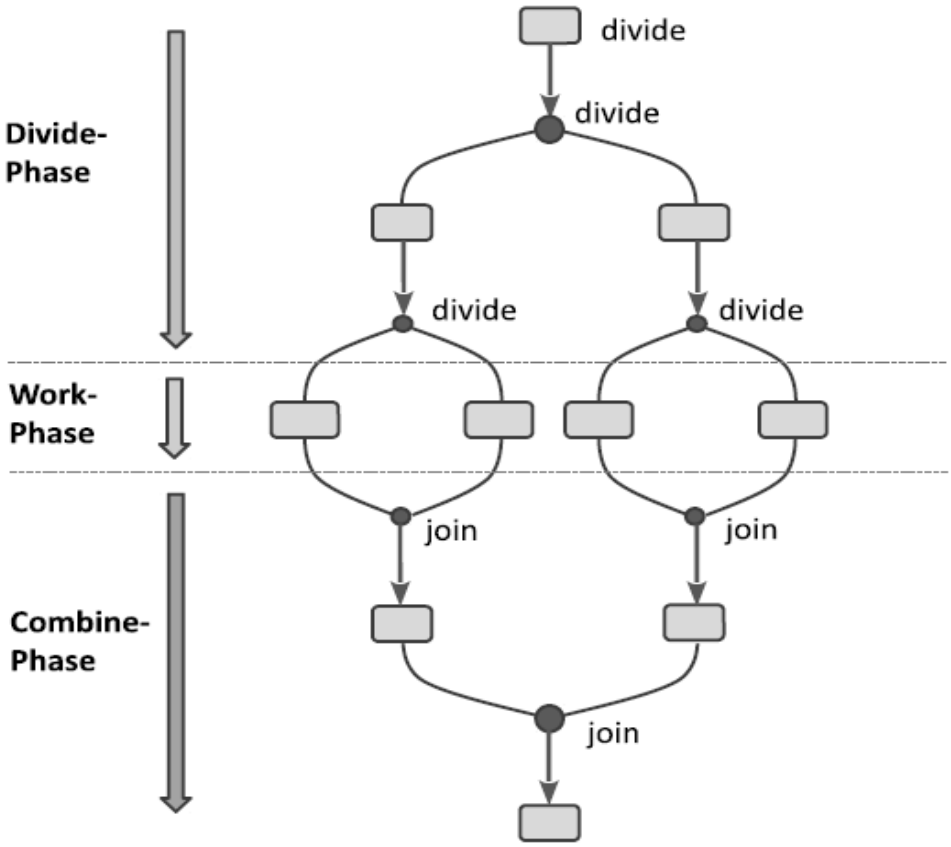


Abbildung 3.4: Rekursive Verwendung des ForkJoin-Patterns

Der ForkJoinTask ist eine abstrakte Basisklasse für Aufgaben, die in einem ForkJoinPool ausgeführt werden. Er ist zusätzlich eine threadähnliche Entität, die viel leichter als ein normaler Thread ist. Eine große Anzahl von Aufgaben und Unteraufgaben kann von einigen wenigen tatsächlichen Threads in einem ForkJoinPool organisiert werden, allerdings mit einigen Einschränkungen bei der Nutzung. Da der ForkJoinTask ein Interface ist, folgt Genaueres zu den abgeleiteten Klassen RecursiveTask und RecursiveAction in Kapitel 3.3.

Der ForkJoinPool unterscheidet sich von anderen Arten von ExecutorService hauptsächlich durch das Anwenden von Work-Stealing[2]. Dies ermöglicht eine effiziente Verarbeitung, wenn die meisten Aufgaben andere Unteraufgaben erzeugen (wie die meisten ForkJoinTasks) und wenn viele kleine Aufgaben von externen Clients an den Pool übermittelt werden.

[2] Alle Threads im Pool versuchen in den Pool gesendete Tasks zu finden und auszuführen oder sie werden von anderen aktiven Tasks erstellt (warten auf Arbeit, wenn keine vorhanden sind).

3.2.5 Das Programmiermodell des ForkJoin-Framework

Die zentralen Komponenten des ForkJoin-Frameworks bestehen aus dem ForkJoinPool-Threadpool und den von ForkJoinTask abgeleiteten abstrakten Klassen RecursiveAction und RecursiveTask (siehe hierzu Abbildung 3.5)[3]. Die Basisklasse für Tasks ohne Rückgabe ist RecursiveAction(). Soll ein Wert zurückgeliefert werden, müssen die Tasks von der Klasse RecursiveTask ableiten. Der ForkJoinPool besitzt die Konstruktoren ForkJoinPool() und ForkJoinPool(int parallelism). Zusätzlich werden explizit eine ThreadFactory, ein UncaughtExceptionHandler und der Ausführungsmodus angegeben [2].

Die für den Umgang mit dem ForkJoin-Framework wichtigen Methoden sind:

- **void execute(ForkJoinTask<?> task)**
- **T invoke(ForkJoinTask<T> task)**
- **ForkJoinTask<T> submit(ForkJoinTask<T> task)**

Die folgende Tabelle 3.1 listet die gebräuchlichen Methoden auf, wobei unterschieden wird, wann und wo sie verwendet werden können. Die Methoden execute(), invoke(), submit() dienen als Startpunkte. Dagegen werden fork() und invoke() innerhalb der compute-Methode aufgerufen und realisieren somit rekursive asynchrone bzw. synchrone Aufrufe [2].

	Aufruf außerhalb eines ForkJoin-Tasks	**Aufruf innerhalb eines ForkJoin-Tasks**
Führt den übergebenen Task asynchron aus.	execute (ForkJoinTask)	ForkJoinTask.fork ()
Startet die Ausführung des Tasks, wobei gewartet wird, bis er fertig ist (synchrone Ausführung).	invoke (ForkJoinTask)	ForkJoinTask.invoke ()
Führt den übergebenen Task asynchron aus und liefert ein ForkJoinTask-Objekt zurück, das auch ein Future ist und mit dem man z.B. auf den Rückgabewert zugreifen kann.	submit (ForkJoinTask)	ForkJoinTask.fork ()

Tabelle 3.1: Aufrufe außerhalb und innerhalb eines ForkJoinTasks

[3] Diese abstrakten Klassen RecursiveAction und RecursiveTask werden unter Kapitel 3.3 erklärt.

3.2.6 Der CommonPool [2]

Um das ständige Erzeugen und Schließen von Pools zu vermeiden, verwendet Java einen globalen Thread Pool, der bei der ersten Verwendung von Java-eigenen Parrallelisierungskonzepten angelegt wird. Zugriff auf diesen Pool erhält man mit *ForkJoinPool.commonPool*. Möchte man diesen CommonPool konfigurieren, so kann man dies über das Setzen von Aufrufparametern beim Starten der JVM realisieren [2]:

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

Der voreingestellte Defaultwert für *parallelism* ist in der Regel *Runtime.getRuntime().availableProcessors()* „**-1**“, falls mehrere Kerne zur Verfügung stehen. Diese Defaulteinstellung 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();
```

3.2.7 Auffangen von Exceptions mit dem UncaughtExceptionHandler

Um einen *UncaughtExceptionHandler* für Pool-Threads festzulegen, stellt man dem ThreadPoolExecutor-Konstruktor eine ThreadFactory zur Verfügung [1]. Die Thread-Manipulationen sollte nur der Eigentümer des Threads beeinflussen dürfen. Die Standard-Thread Pools ermöglichen eine Ausnahme für nicht erfasste Tasks, um den Pool-Thread zu beenden. Verwendung findet hierbei besser ein *try-finally-Block*, um benachrichtigt zu werden, wenn diese Beendigung passiert. Ohne einen nicht abgefangenen Exception-Handler oder einen anderen Fehlerbenachrichtigungs-mechanismus können Aufgaben unbemerkt fehlschlagen. Wenn man benachrichtigt werden möchte, wenn eine Aufgabe aufgrund einer Ausnahme fehlschlägt, schließt man den Task z.B. mit einem ausführbaren/aufrufbaren Element ab, welches die Ausnahme abfängt. Es ist auch möglich, den *afterExecute*-Hook in ThreadPoolExecutor zu überschreiben.

Diese Uncaught-Exceptions werden nur von Tasks, die mit *execute()* übergeben wurden, ausgelöst. Jede Exception wird als Teil des Return-Status des Task betrachtet.

3.3 Future, RecursiveTask und RecursiveAction

Die folgende Abbildung 3.5 zeigt die Hierarchie der Task-Klassen:

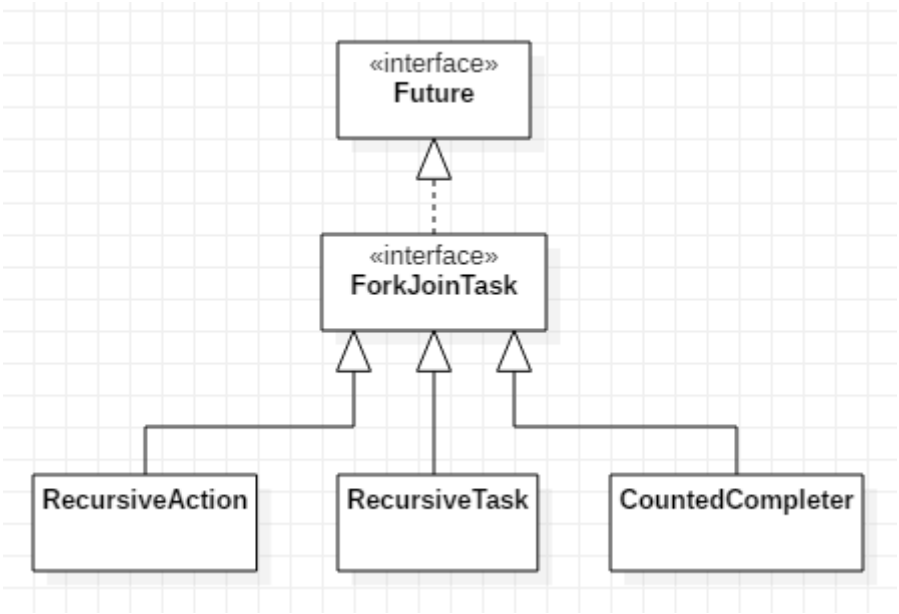


Abbildung 3.5: Hierarchie der Task-Klassen

3.3.1 <<Interface>> Future

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 warden. Ein Callable wird einem ServiceExecutor über die Methode "*submit(callable)*" zur Ausführung übergeben, die ein Future-Objekt zurückliefert. Über dieses Future-Objekt kann die Rückgabe erfragt werden [2].

Statt eines Runnable-Objektes wird hierbei ein Callable-Objekt verwendet. Die überschriebene call-Methode hat eine typisierte Rückgabe. Das Callable-Objekt wird mit "*submit(callable)*" dem Threadpool übergeben. Damit liefert die Rückgabe ein Future-Objekt.

3.3.2 ForkJoinTaskTask<V>

ForkJoinTask<V> ist die Superklasse von RecursiveTask <V> und RecursiveAction. *fork()* und *join()* sind Methoden, die in dieser abstrakten Klasse definiert sind. Diese abstrakte Klasse wird nicht direkt verwendet, aber es ist die Klasse mit den meisten

nützlichen Javadoc-Dokumentationen, falls man mehr über zusätzliche Methoden erfahren möchte [17].

3.3.3 RecursiveTask<V>

Mit RecursiveTask<V> wird eine Unterklasse davon in einem Pool ausgeführt und lässt diese Unterklasse ein Ergebnis zurückgeben [17]. Soll durch eine parallele Bearbeitung ein Ergebnis ermittelt werden, kann RecursiveTask<V> eingesetzt werden. Man spricht in dem Zusammenhang auch oft von einer reduzierten Arbeitsweise [2].

3.3.4 RecursiveAction

RecursiveAction funktioniert genau wie RecursiveTask<V>, außer dass diese kein Ergebnis zurück gibt [17]. RecursiveAction ist deshalb ein rekursiver ergebnisloser ForkJoinTask. Diese Klasse ermöglicht ergebnislose Aktionen als (void)-ForkJoinTasks. Da Null der einzige gültige Wert des Typs “Void” ist, geben Methoden wie *join()* nach der Vollendung immer Null zurück. RekursiveAction muss nicht vollständig rekursiv sein, solange sie den grundlegenden Divide-and-Conquer-Ansatz beibehält.

3.4 Static Factory Methods in Executers

3.4.1 Die Klasse Executors

Executors ist eine Factory bzw. ein Werkzeug für Executor, ExecutorService, ScheduledExecutorService, ThreadFactory und aufrufbare Klassen aus dessen Package.

Der Executors basiert auf dem Producer-Consumer Pattern (Produzenten-Konsumenten-Muster/Modell), wo die Aktivitäten, die die Tasks übernehmen, die „Produzenten“ sind. Die Threads, die die Tasks ausführen, sind die „Konsumenten“ *(die diese Einheiten verbrauchen)*. Der Executor ist in der Regel der einfachste Weg, um ein Thread Pool in einer Applikation zu erzeugen.

Die *newFixedThreadPool()*- und *newCachedThreadPool()*- Factories liefern Instanzen von dem general-purpose ThreadPoolExecutor, der auch direkt verwendet werden kann, um weitere spezialisierte Executors zu erstellen [1].

Ebenfalls öffnen sich mit einem "Executor" die Türen für alle Arten von zusätzlichen Möglichkeiten, wie z.B. für die Optimierung, Verwaltung, Überwachung, Protokollierung und Fehlerberichterstattung. Dieses Ganze wäre weitaus schwieriger, ohne ein Task-Execution-Framework [1].

3.4.2 Executors Thread Pool Factory Methods

Die folgende Tabelle 3.2 zeigt die Auflistung der wichtigsten Fabrikmethoden [1, 2]:

Klassenmethoden	**Beschreibung**
newFixedThreadPoll (int nThreads)	begrenzt Anzahl parallel laufender Threads
newCachedThreadPool ()	erzeugt so viele Threads wie benötigt
newSingleThreadExecutor ()	lässt nur einen Thread (zurzeit) zu
newSingleThreadScheduledExecutor()	erstellt einen Singlethread-Executor, der Befehle so einplanen kann, dass sie nach einer bestimmten Verzögerung oder in regelmäßigen Abständen ausgeführt werden
newScheduledThreadPool (int coreSize)	unterstützt die verzögerte und periodische Ausführung von Aufgaben bei einer begrenzten Anzahl parallel laufender Threads
newWorkStealingPool ()	Unterstützt das sogenannte Work-Stealing-Verfahren für den ForkJoinPool
newWorkStealingPool (int parallelism)	Unterstützt das sogenannte Work-Stealing-Verfahren für den ForkJoinPool

Tabelle 3.2: Auflistung der wichtigsten Fabrikmethoden

newFixedThreadPool(int nThreads):

Der “Fixed-size” Thread Pool erstellt Threads, wenn Tasks übermittelt werden. “Fixed-size” heißt, dass bis zu einer maximalen Pool-Größe Threads erstellt werden. Hierbei wird, falls ein Thread unerwartet stirbt, die Pool-Größe konstant gehalten. Dies bedeutet, dass bis zum erreichen der “Fixed-size” jedes Mal wieder ein neuer Thread hinzugefügt wird.

newCachedThreadPool():

Ein “Cached” Thread Pool hat mehr Flexibilität, um im Leerlauf befindliche Threads abzuschöpfen, wenn die aktuelle Größe des Pools den Verarbeitungsbedarf übersteigt. Bei Bedarfzunahme werden zusätzlich neue Threads hinzugefügt. Der Größe der Pools sind hierbei keine Grenzen gesetzt.

newSingleThreadExecutor():

Ein Single-Threaded-Executor erstellt jedes Mal nur einen einzelnen Worker-Thread zum Verarbeiten von Tasks. Dieser Worker-Thread wird ersetzt falls er unerwartet stirbt. Die Tasks werden entsprechend der Reihenfolge garantiert sequentiell abgearbeitet, wie dies von der Task-Queue festgelegt wird (FIFO, LIFO, priority order).

newSingleThreadScheduledExecutor():

Ein Single-Thread-Scheduled-Executor kann Befehle so einplanen, dass sie nach einer bestimmten Verzögerung oder in regelmäßigen Abständen ausgeführt werden.

newScheduledThreadPool(int coreSize):

Der “newScheduledThreadPool()” ist ein “Fixed-size” Thread Pool, der die verzögerte und periodische Ausführung von Aufgaben unterstützt.

newWorkStealingPool() / newWorkStealingPool(int parallism):

Mit Java 8 wurden noch zwei Fabrikmethoden für den ForkJoinPool eingeführt, der das sogenannte Work-Stealing-Verfahren unterstützt, dessen Arbeitsweise unter Kapitel 3.2 besprochen wurde.

4 Beispielimplementierungen

4.1 Beispiele in Java mit Runnable

4.1.1 Executors.*newFixedThreadPool*(threadPoolSize)

Die Abbildung 4.1 zeigt einen ausführbaren Task. Dieser Task führt Arbeit aus und berichtet dann regelmäßig, wie viel Prozent der Arbeit abgeschlossen wurde [9]:

Task.java ThreadPoolTest01.java

```
1  package examples;
2
4  * Task.java
10
11 public class Task implements Runnable {
12     private int taskNumber;
13
14     Task(int number) {
15         taskNumber = number;
16     }
17
18     public void run() {
19         for (int i = 0; i <= 100; i += 20) {
20             // Perform some work ...
21             System.out.println("Task number: " + taskNumber + ", Thread ID: " + Thread.currentThread().getId() + ", percent complete: " + i);
22             try {
23                 Thread.sleep((int) (Math.random() * 1000));
24             } catch (InterruptedException e) {
25             }
26         }
27     }
28 }
```

Abbildung 4.1: Java-Code-Runnable-Task

In ThreadPoolTest01 kann die Anzahl der zu erstellenden Arbeitsthreads (numTasks) und die Größe des Thread-Pools (treadPoolSize) angegeben werden. Diese werden zum Ausführen der Threads verwendet. In diesem Beispiel wird ein fester Thread-Pool verwendet, damit die Auswirkungen des Programms mit weniger Threads als Tasks beobachtet werden können:

Task.java ThreadPoolTest01.java

```
1  package examples;
2
4  * ThreadPoolTest01.java
11
12 import java.util.concurrent.*;
13
14 public class ThreadPoolTest01 {
15     public static void main(String[] args) {
16         int numTasks = 4;
17         int threadPoolSize = 2;
18
19         ExecutorService tpes = Executors.newFixedThreadPool(threadPoolSize);
20
21         Task[] tasks = new Task[numTasks];
22         for (int i = 0; i < numTasks; i++) {
23             tasks[i] = new Task(i);
24             tpes.execute(tasks[i]);
25         }
26         tpes.shutdown();
27     }
28 }
```

Abbildung 4.2: Java-Code-Testklasse

Die Abbildungen 4.3 und 4.4 zeigen ein Beispiel-Ergebnis des Tests mit 4 Tasks und einer ThreadPoolSize -> 2.

Diese folgende Ausgabe ist so nicht zwingend reproduzierbar:

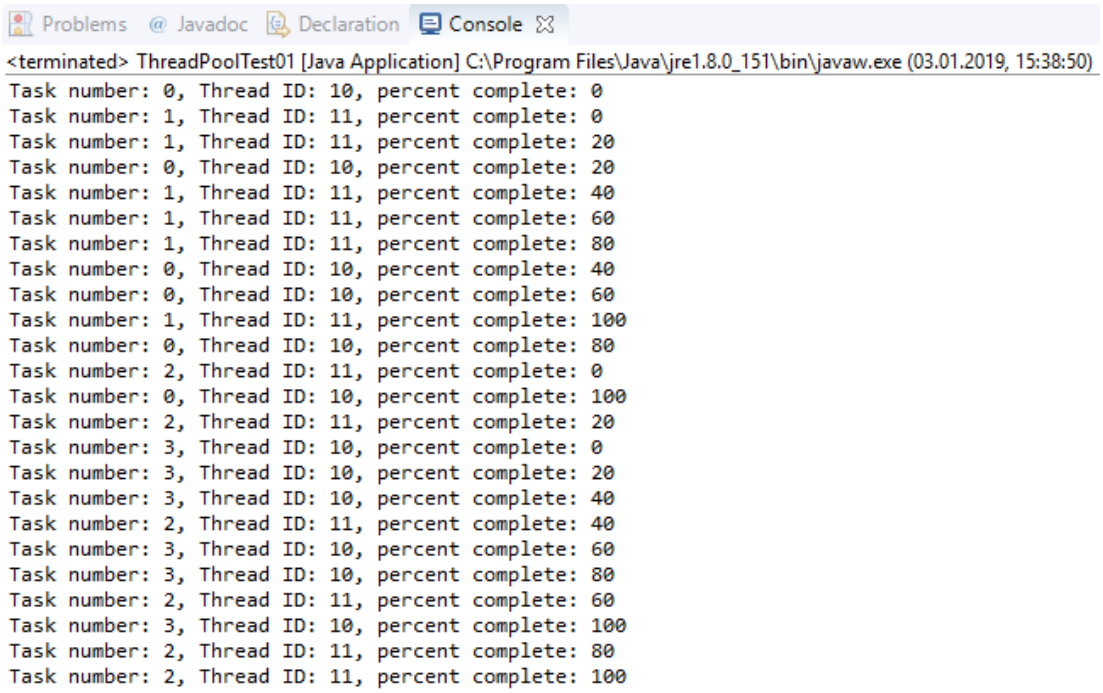


Abbildung 4.3: Ergebnis-Konsolenausgabe

Abbildung 4.4: Ergebnisdarstellung

4.1.2 Executors.*newCachedThreadPool*()

In ThreadPoolTest02 kann die Anzahl der zu erstellenden Arbeits-Tasks (numTasks) angegeben werden. Die Größe des Thread Pools weist hier somit etwas mehr Flexibilität auf. In diesem Beispiel wird kein von der Größe fester Thread-Pool verwendet, damit man die Auswirkungen des Programms mit weniger Threads als Tasks beobachten kann [9]:

Task.java | ThreadPoolTest01.java | ThreadPoolTest02.java

```
1  package examples;
2
4  * ThreadPoolTest02.java
11
12 import java.util.concurrent.*;
13
14 public class ThreadPoolTest02 {
15     public static void main(String[] args) {
16         int numTasks = 4;
17         //int threadPoolSize = 2;
18
19         ExecutorService tpes = Executors.newCachedThreadPool();
20
21         Task[] tasks = new Task[numTasks];
22         for (int i = 0; i < numTasks; i++) {
23             tasks[i] = new Task(i);
24             tpes.execute(tasks[i]);
25         }
26         tpes.shutdown();
27     }
28 }
```

Abbildung 4.5: Java-Code-Testklasse

Die Abbildungen 4.6 und 4.7 zeigen das Ergebnis des Tests mit 4 Tasks und einer flexiblen ThreadPoolSize -> (die flexible ThreadPoolSize beträgt hier 4).

Diese folgende Ausgabe ist so nicht zwingend reproduzierbar:

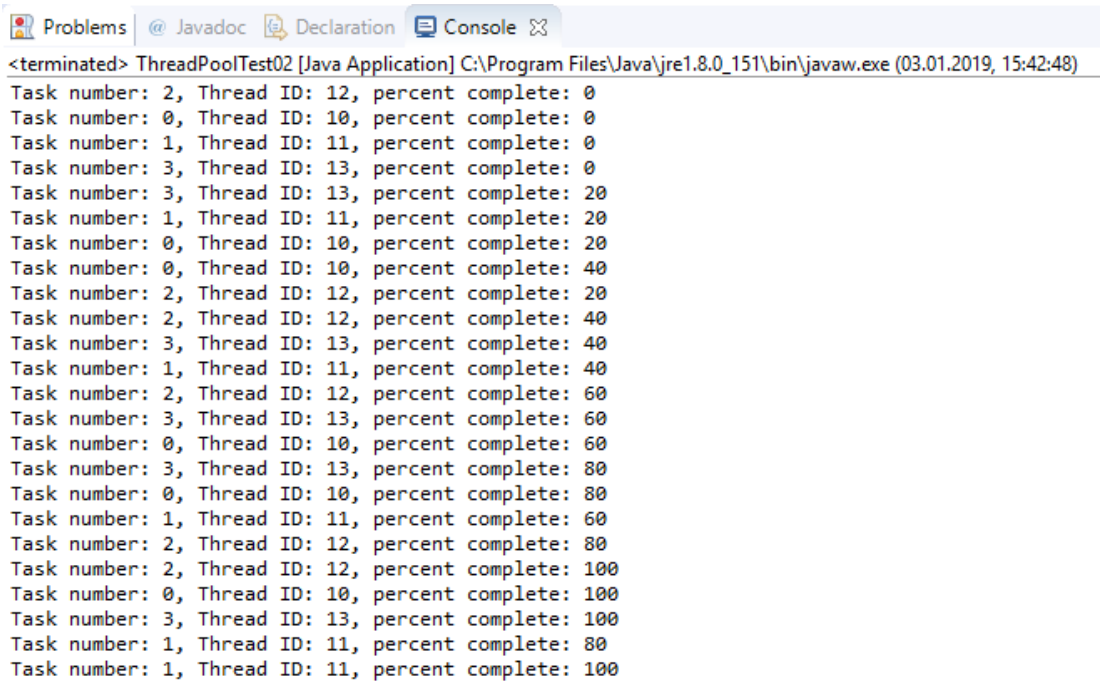


Abbildung 4.6: Ergebnis-Konsolenausgabe

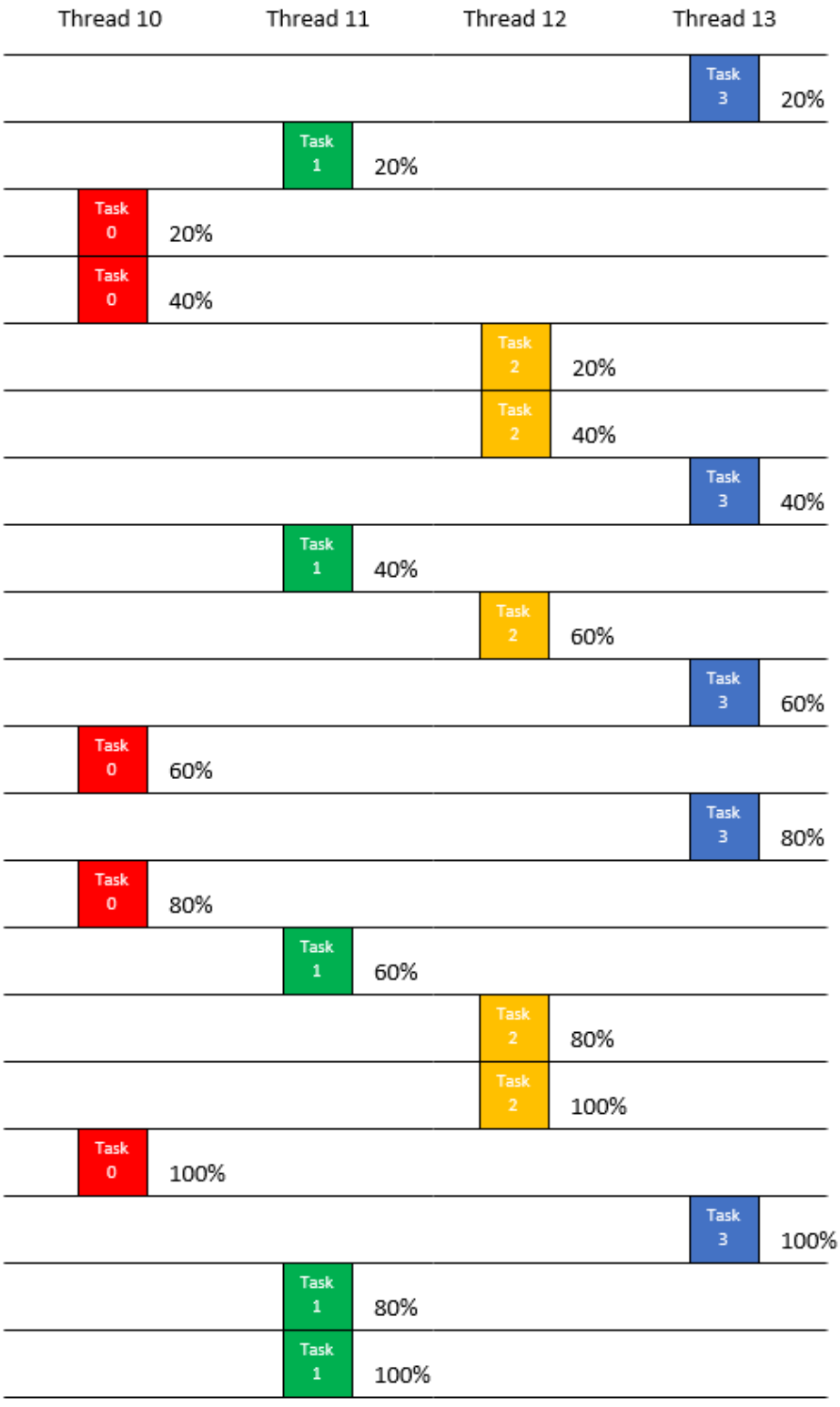


Abbildung 4.7: Ergebnisdarstellung

4.1.3 Executors.*newSingleThreadExecutor*()

In ThreadPoolTest03 erstellt ein Single-Threaded-Executor jedes Mal nur einen einzelnen Worker-Thread zum Verarbeiten von Tasks. Dieser Worker-Thread wird ersetzt, falls er unerwartet stirbt. Die Tasks werden entsprechend der Reihenfolge garantiert sequentiell abgearbeitet, wie dies von der Task-Queue festgelegt wird (FIFO, LIFO, priority order) [9].

Task.java | ThreadPoolTest01.java | ThreadPoolTest02.java | ThreadPoolTest03.java

```
1  package examples;
2
4  * ThreadPoolTest03.java
11
12 import java.util.concurrent.*;
13
14 public class ThreadPoolTest03 {
15     public static void main(String[] args) {
16         int numTasks = 4;
17         //int threadPoolSize = 2;
18
19         ExecutorService tpes = Executors.newSingleThreadExecutor();
20
21         Task[] tasks = new Task[numTasks];
22         for (int i = 0; i < numTasks; i++) {
23             tasks[i] = new Task(i);
24             tpes.execute(tasks[i]);
25         }
26         tpes.shutdown();
27     }
28 }
```

Abbildung 4.8: Java-Code-Testklasse

Die Abbildungen 4.9 und 4.10 zeigen das Ergebnis des Tests mit 4 Tasks und einer Benutzung von einem Single-Threaded-Executor.

Diese folgende Ausgabe ist so nicht zwingend reproduzierbar:

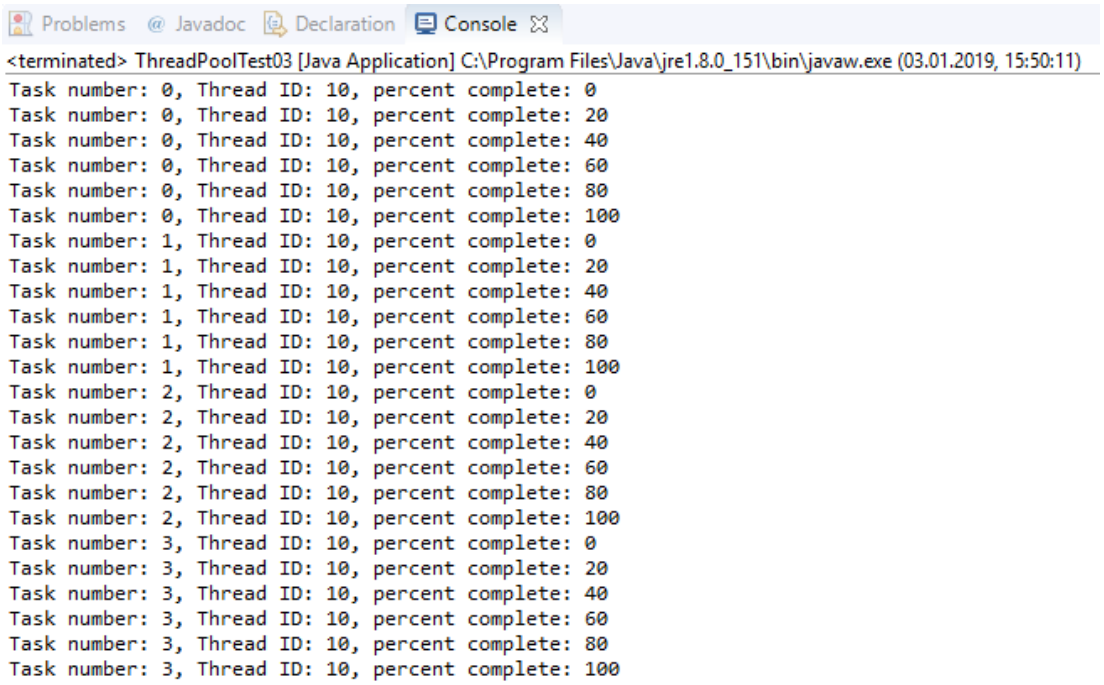


Abbildung 4.9: Ergebnis-Konsolenausgabe

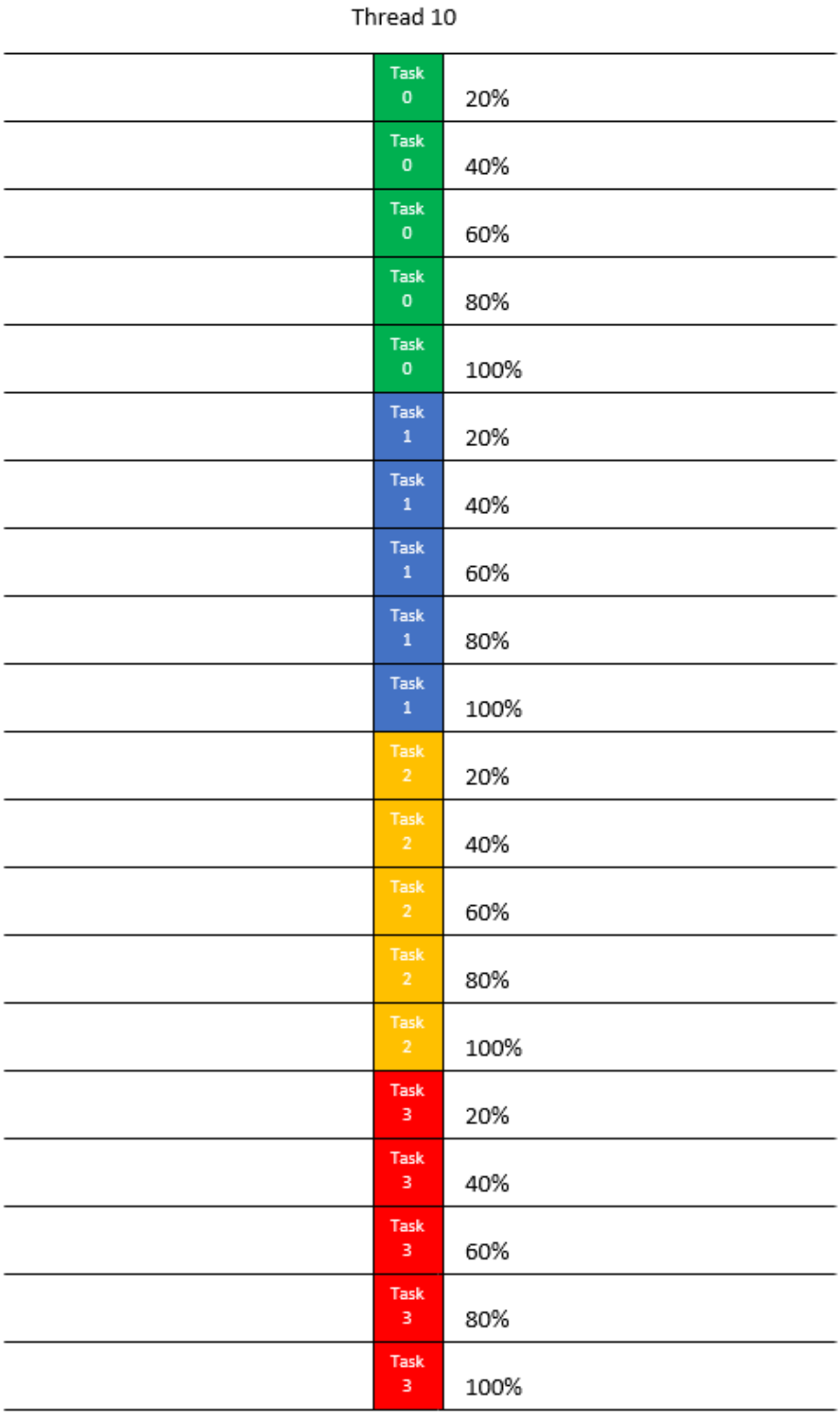


Abbildung 4.10: Ergebnisdarstellung

4.1.4 Executors.*newScheduledThreadPool*()

In ThreadPoolTest04 wird ein “newScheduledThreadPool()” erstellt, welcher ein “Fixed-size” Thread Pool ist, der die verzögerte und periodische Ausführung von Aufgaben unterstützt [9].

Task.java | ThreadPoolTest02.java | ThreadPoolTest03.java | ThreadPoolTest04.java

```
1  package examples;
2
4  * ThreadPoolTest04.java
11
12 import java.util.concurrent.*;
13
14 public class ThreadPoolTest04 {
15     public static void main(String[] args) {
16         int numTasks = 4;
17         int corePoolSize = 2;
18
19         ExecutorService tpes = Executors.newScheduledThreadPool(corePoolSize);
20
21         Task[] tasks = new Task[numTasks];
22         for (int i = 0; i < numTasks; i++) {
23             tasks[i] = new Task(i);
24             tpes.execute(tasks[i]);
25         }
26         tpes.shutdown();
27     }
28 }
```

Abbildung 4.11: Java-Code-Testklasse

Die Abbildungen 4.12 und 4.13 zeigen das Ergebnis des Tests mit 4 Tasks und einer Benutzung von einem New-Scheduled-Threaded-Pool.

Diese folgende Ausgabe ist so nicht zwingend reproduzierbar:

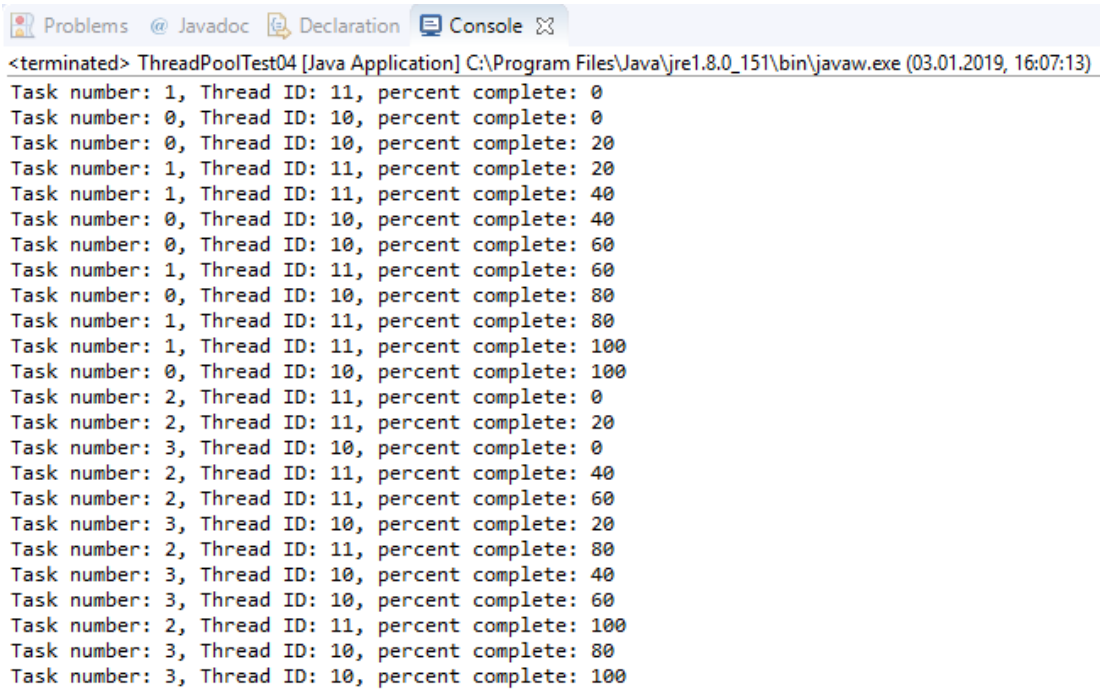


Abbildung 4.12: Ergebnis-Konsolenausgabe

Abbildung 4.13: Ergebnisdarstellung

4.2 Beispiele in Java mit Callable<Integer>

Eine andere Möglichkeit zum Erstellen von Tasks besteht in der Implementierung des Callable-Interface. Ein Callable ist flexibler als ein Runnable, da es einen Wert zurückgeben und eine Exception auslösen kann. Um ein Callable zu implementieren, überschreibt man die call()-Methode, die einen Wert zurückgibt, in diesem Fall einen Integer, die die Task-Nummer darstellt.

4.2.1 Executors.*newCachedThreadPool*()

CallableTask.java | ThreadPoolTest05.java

```
1  package examplesCallable;
2
4  * CallableTask.java
10
11 import java.util.concurrent.*;
12
13 public class CallableTask implements Callable<Integer> {
14     private int taskNumber;
15
16     CallableTask(int number) {
17         taskNumber = number;
18     }
19
20     public Integer call() {
21         for (int i = 0; i <= 100; i += 20) {
22             // Perform some work ...
23             System.out.println("Task number: " + taskNumber + ", Thread ID: " + Thread.currentThread().getId() + ", percent complete: " + i);
24             try {
25                 Thread.sleep((int)(Math.random() * 1000));
26             } catch (InterruptedException e) {
27             }
28         }
29         return(taskNumber);
30     }
31 }
```

Abbildung 4.14: Java-Code-Callable-Task

ThreadPoolTest05 verwendet den CachedThreadPool-Executor-Service, der beliebig viele Threads erstellt, zuvor erstellte Threads jedoch wiederverwendet, sofern verfügbar. Man verwendet die Submit-Methode, um einen Executor-Service zu bitten ein Callable auszuführen. Diese Methode gibt ein Future-Objekt zurück, mit dem man die Aufgabe steuert. Mit Future kann man dann das Ergebnis der Ausführung des Tasks abrufen, die Tasks überwachen und die Aufgabe abbrechen. Um beispielsweise auf das Ergebnis zuzugreifen, rufen Sie einfach die Methode get auf [9].

CallableTask.java ThreadPoolTest05.java

```
1  package examplesCallable;
2
4  * ThreadPoolTest05.java
11
12 import java.util.concurrent.*;
13
14 public class ThreadPoolTest05 {
15     public static void main(String[] args) {
16         int numTasks = 4;
17
18         ExecutorService tpes =
19             Executors.newCachedThreadPool();
20         CallableTask tasks[] =
21             new CallableTask[numTasks];
22         Future futures[] = new Future[numTasks];
23
24         for (int i = 0; i < numTasks; i++) {
25             tasks[i] = new CallableTask(i);
26             futures[i]=tpes.submit(tasks[i]);
27         }
28         for (int i = 0; i < numTasks; i++) {
29             try {
30                 System.out.println("Ending Task: " + futures[i].get());
31             } catch (Exception e) {}
32         }
33     }
34 }
```

Abbildung 4.15: Java-Code-Testklasse

Die Abbildungen 4.16 und 4.17 zeigen das Ergebnis der Ausführung von ThreadPoolTest05. Zu beachten ist, wie jedem Task sofort ein Thread zur Ausführung zugewiesen wird. Nach Abschluss jedes Tasks wird der Identifier zurückgegeben, den ThreadPoolTest05 von den Task-Futures abruft.

Diese folgende Ausgabe ist so nicht zwingend reproduzierbar:

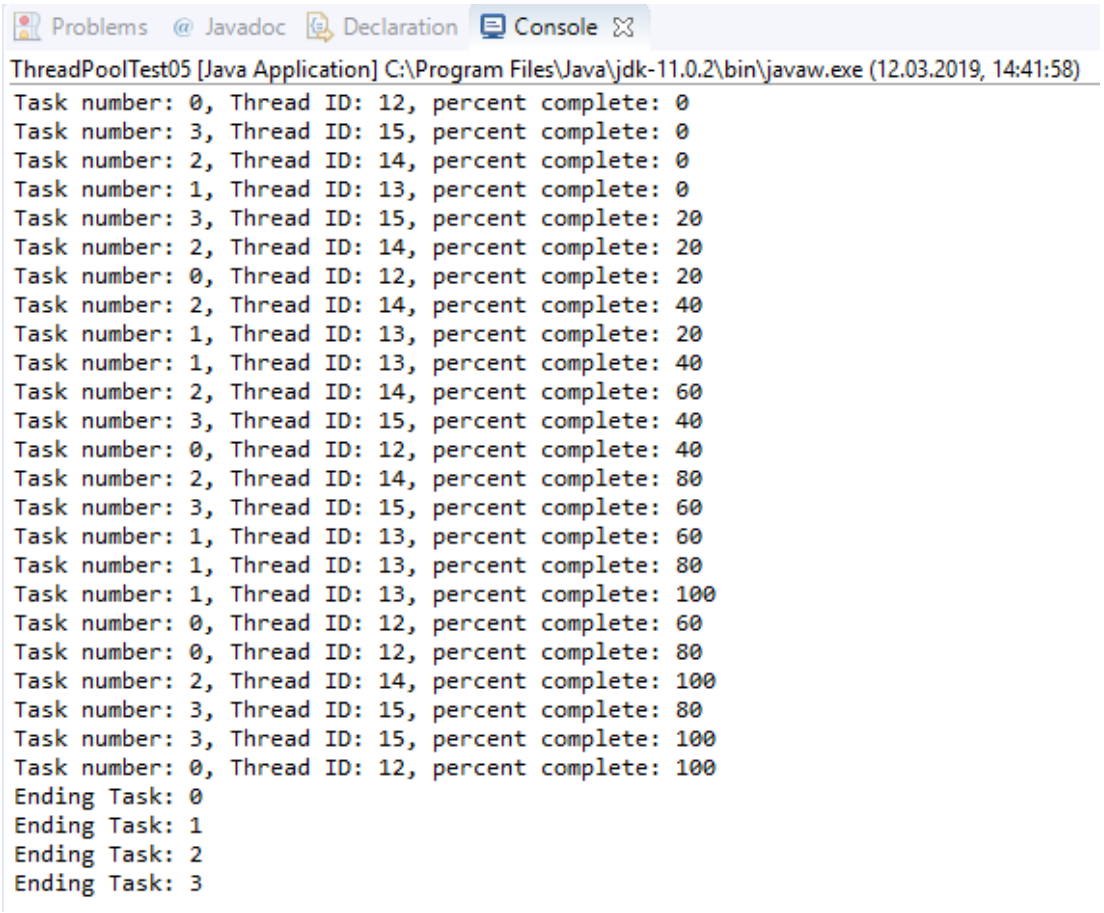


Abbildung 4.16: Ergebnis-Konsolenausgabe

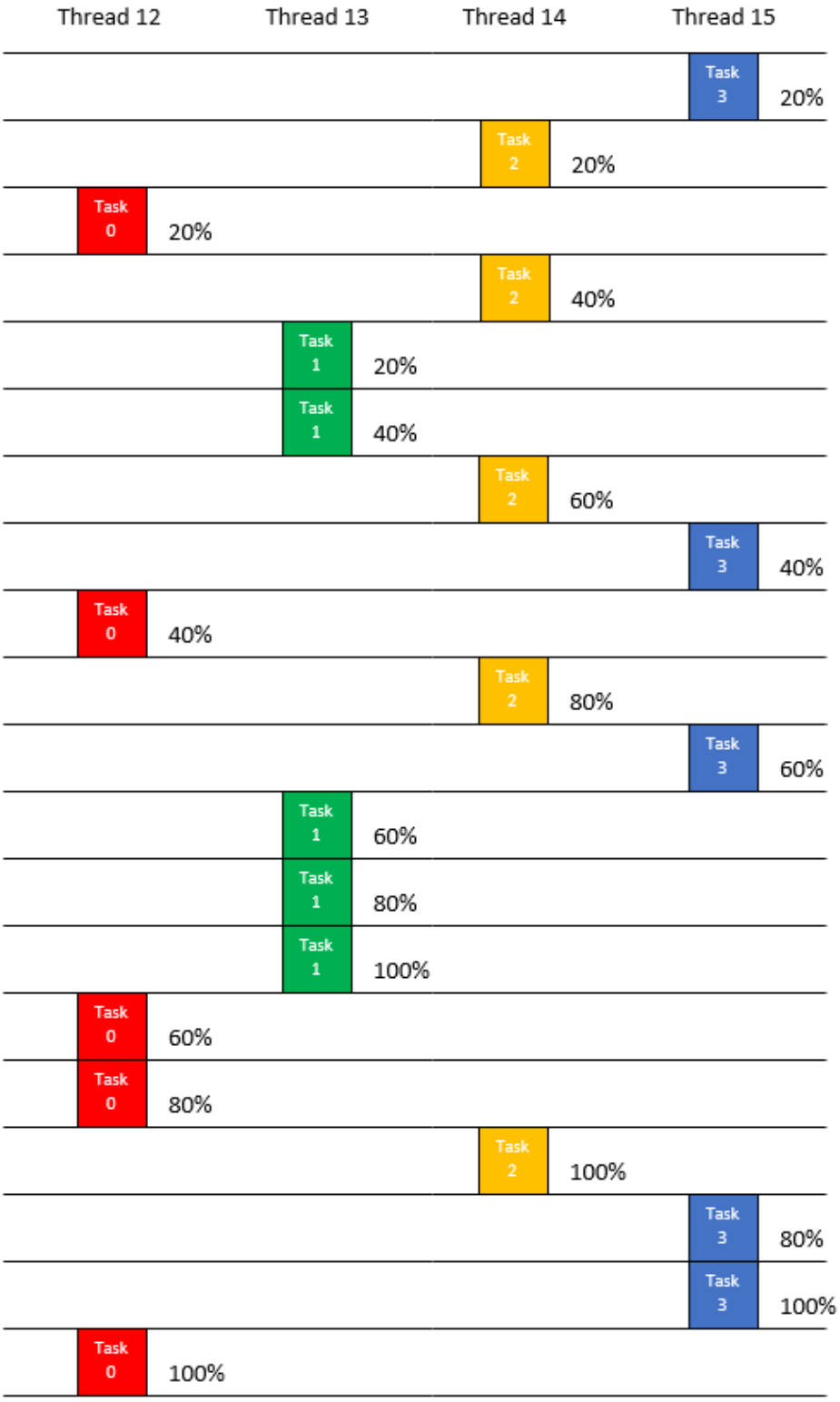


Abbildung 4.17: Ergebnisdarstellung

4.2.2 Executors.*newFixedThreadPool*(threadPoolSize)

In ThreadPoolTest06 können wir die Anzahl der zu erstellenden Arbeitsthreads (numWorkers) und die Größe des Thread-Pools (treadPoolSize) angeben, der zum Ausführen der Threads verwendet wird. In diesem Beispiel wird ein fester Thread-Pool verwendet, damit die Auswirkungen des Programms mit weniger Threads als Tasks beobachtet werden können [9]:

CallableTask.java | ThreadPoolTest05.java | ThreadPoolTest06.java

```
1  package examplesCallable;
2
4  * ThreadPoolTest06.java
11
12 import java.util.concurrent.*;
13
14 public class ThreadPoolTest06 {
15     public static void main(String[] args) {
16         int numTasks = 4;
17         int threadPoolSize = 2;
18
19         ExecutorService tpes = Executors.newFixedThreadPool(threadPoolSize);
20         CallableTask tasks[] = new CallableTask[numTasks];
21         Future futures[] = new Future[numTasks];
22
23         for (int i = 0; i < numTasks; i++) {
24             tasks[i] = new CallableTask(i);
25             futures[i] = tpes.submit(tasks[i]);
26         }
27         for (int i = 0; i < numTasks; i++) {
28             try {
29                 System.out.println("Ending Task: " + futures[i].get());
30             } catch (Exception e) {
31             }
32         }
33     }
34 }
```

Abbildung 4.18: Java-Code-Testklasse

Die Abbildungen 4.19 und 4.20 zeigen ein Beispiel-Ergebnis des Tests mit 4 Tasks und einer ThreadPoolSize -> 2.

Diese folgende Ausgabe ist so nicht zwingend reproduzierbar:

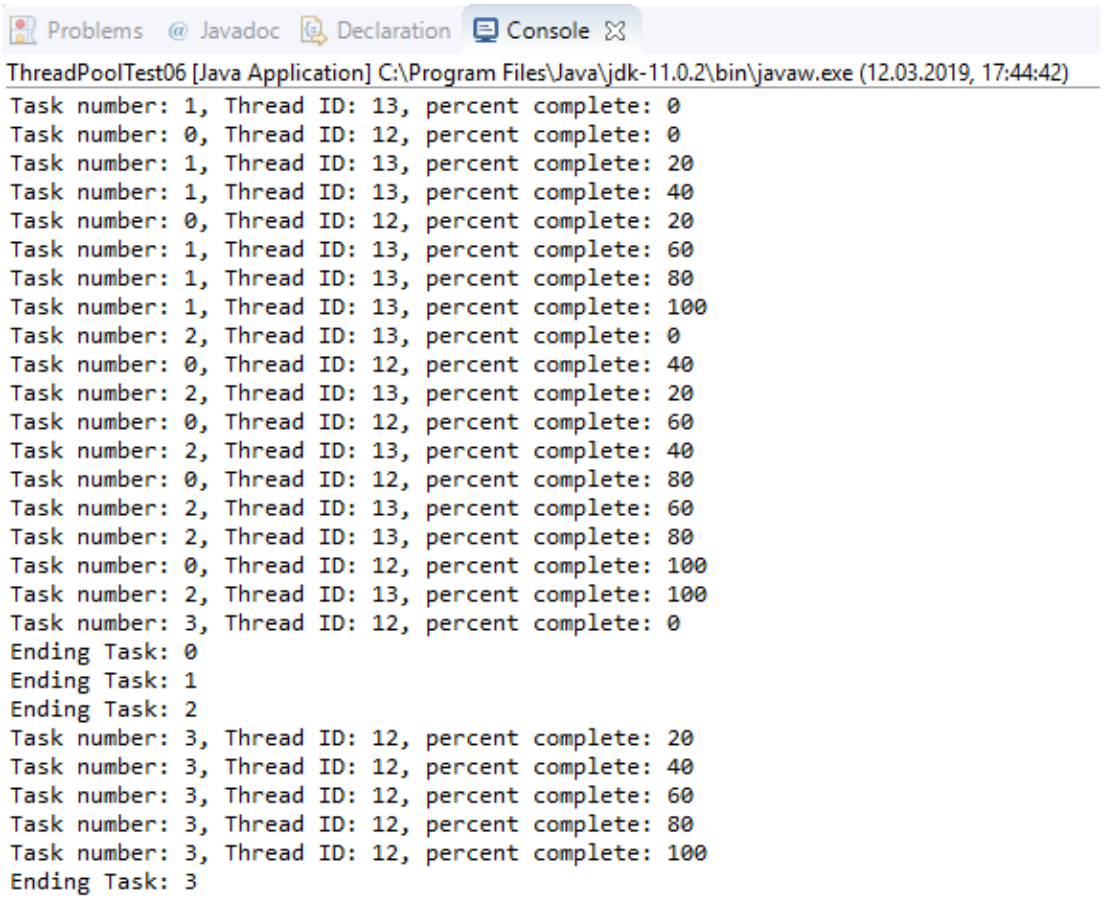


Abbildung 4.19: Ergebnis-Konsolenausgabe

Abbildung 4.20: Ergebnisdarstellung

5 Fazit

Der Thread Pool ist ein nützliches Werkzeug zum Organisieren von Serveranwendungen. Dieses Konzept des Thread Pools ist recht unkompliziert. Es gibt jedoch einige Probleme, die bei der Implementierung und Verwendung eines Problems zu beachten sind, wie etwa Deadlock, Ressourcen-Thrashing und die Komplexität von wait() und notify(). Wenn ein Thread-Pool für eine Anwendung benötigt wird, kann eine der Static-Factory-Methoden von Executors aus dem "java.util.concurrent" verwendet werden, anstatt ein Executor-Objekt von Grund auf neu zu schreiben. Ebenfalls ist bei der Erstellung von Threads, die kurzlebige Aufgaben erledigen, zu beachten, dass ein Thread-Pool verwendet werden sollte.

Das Optimieren der Größe eines Thread-Pools ist sehr wichtig [10]. Dieses dient dazu, zwei Fehler zu vermeiden: Zu wenige Threads oder zu viele Threads. Für die meisten Anwendungen ist der Mittelweg zwischen zu wenigen und zu vielen Threads ziemlich breit. Die optimale Größe eines Thread-Pools hängt von der Anzahl der verfügbaren Prozessoren und der Art der Aufgaben in einer Arbeitswarteschlange ab.

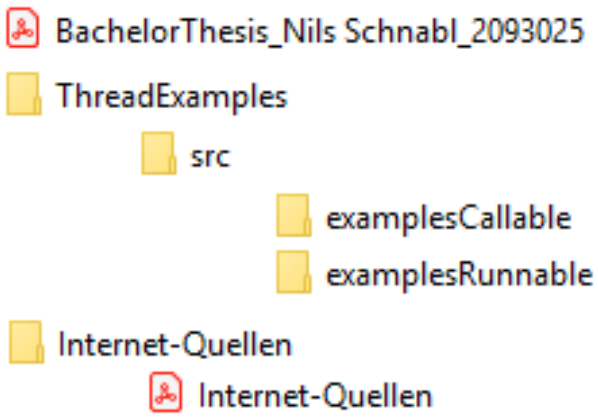
Abschließend ist festzustellen, dass bei einem N-Prozessor-System für eine Arbeitswarteschlange im Allgemeinen eine maximale CPU-Auslastung mit einem Thread-Pool von N oder N+1 Threads zu erzielen ist.

A Inhalt der CD

Auf der beiliegenden CD befindet sich im Ordner ThreadExamples (ein Eclipse-Projekt-Ordner/ ***Eclipse 2018-12***) der Quellcode zu dem Kapitel Beispielimplementierungen. Die Bespiele in Java mit Runnable befinden sich im Package examplesRunnable und die Bespiele in Java mit Callable befinden sich im Package examplesCallable. Um den Quellcode auszuführen muss dieser Projektordner ThreadExamples als Projekt von Eclipse importiert werden.

Ebenfalls auf der obersten Ebene befinden sich eine PDF-Datei dieser Bachelor-Thesis und der Ordner mit den flüchtigen Internet-Quellen.

Folgend wird die Struktur der CD aufgezeigt:



Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Bücher & Artikel

[1] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea (2006): Java Concurrency in Practice, Addison-Wesley Pearson Education

[2] Jörg Hettel, Manh Tien Tran (2016): Nebenläufige Programmierung mit Java, dpunkt.verlag GmbH

[3] Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft (2015): Java 8 in Action, Manning Publications Co.

Internet-Quellen

[4] Wikipedia: „Overhead (EDV)“. (2018). https://de.wikipedia.org/wiki/Overhead_(EDV) (Zugriffsdatum: 24.06.2019)

[5] Wikipedia: „Zyklische Redundanzprüfung“. (2019). https://de.wikipedia.org/wiki/Zyklische_Redundanzprüfung (Zugriffsdatum: 14.07.2019)

[6] C.-K. Shene: „Multithreaded Programming with ThreadMentor: A Tutorial“. (2014). http://pages.mtu.edu/~shene/NSF-3/e-Book/FUNDAMENTALS/thread-management.html (Zugriffsdatum: 24.06.2019)

[7] Rajeev Kumar Singh: „CalliCoder, computer science education blog: Java Thread and Runnable Tutorial“. (2017). https://www.callicoder.com/java-multithreading-thread-and-runnable-tutorial/ (Zugriffsdatum: 24.06.2019)

[8] Wikipedia: „Thread pool“. (2019). https://en.wikipedia.org/wiki/Thread_pool (Zugriffsdatum: 24.06.2019)

[9] „The Java Tutorial: Thread Pools“. (2005). https://www.math.uni-hamburg.de/doc/java/tutorial/essential/threads/group.html (Zugriffsdatum: 24.06.2019)

[10] Brian Goetz: „Thread pools and work queues“. (2002). https://www.ibm.com/developerworks/java/library/j-jtp0730/index.html (Zugriffsdatum: 24.06.2019)

[11] Eugen Paraschiv: „Runnable vs. Callable in Java“. https://www.baeldung.com/java-runnable-callable (Zugriffsdatum: 24.06.2019)

[12] Java-Api 10 - java.util.concurrent (Zugriffsdatum: 24.06.2019)

[13] Wikipedia: „Thread (computing)“. (2019). https://en.wikipedia.org/wiki/Thread_(computing) (Zugriffsdatum: 24.06.2019)

[14] „Java-Threads-Interdface Runnable“. www.kleinerorkan.de/SelfJAVA/java0902.htm (Zugriffsdatum: 03.08.2019)

[15] Michael Vitz: „Java 10-Evolution statt Revolution“. (2018). https://www.innoq.com/de/articles/2018/03/java-10/ (Zugriffsdatum: 24.06.2019)

[16] Wikipedia: „Pool (Informatik)“. (2019). https://de.wikipedia.org/wiki/Pool_(Informatik) (Zugriffsdatum: 24.06.2019)

[17] Dan Grossman: „Beginner‘s Introduction to Java's ForkJoin Framework“. (2016). https://homes.cs.washington.edu/~djg/teachingMaterials/spac/grossmanSPAC_forkJoinFramework.html (Zugriffsdatum: 29.06.2019)

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 22.08.2019 Nils Schnabl