

7 Multithreading

Das Thema Multithreading und Parallelverarbeitung mehrerer Aufgaben gewinnt durch Computer mit mehreren Prozessoren (CPUs) oder Prozessoren mit mehreren Rechenkernen (Multicores) zunehmend an Bedeutung. Ziel ist es, komplexe Aufgaben innerhalb von Programmen in voneinander unabhängige Teilaufgaben zu untergliedern, die parallel abgearbeitet werden können. Ist dies der Fall, spricht man auch von Nebenläufigkeit. Java bietet zwar einen einfachen Zugang zur Programmierung mit Threads, allerdings verleitet dies manchmal dazu, Multithreading einzusetzen, ohne die resultierenden Konsequenzen zu beachten. Dadurch kommt es in der Praxis aber immer wieder zu Problemen. Dieses Kapitel soll helfen, Multithreading mit seinen Möglichkeiten (aber auch Fallstricken) kennenzulernen.

Eine kurze Einführung in das Thema Multithreading und Nebenläufigkeit mit den Klassen `Thread` und `Runnable` gibt Abschnitt 7.1. Wenn in einem Programm Aufgaben auf mehrere Threads aufgeteilt werden, müssen deren Berechnungsergebnisse an verschiedenen Stellen innerhalb des Programmablaufs abgeglichen oder zusammengeführt werden. Dies ist Thema in Abschnitt 7.2. Auch die Kommunikation zwischen Threads hat Einfluss auf eine erfolgreiche Zusammenarbeit. Darauf gehe ich in Abschnitt 7.3 ein. Hintergründe zum Java-Memory-Modell und diesbezüglich einige wichtige Details beim Zugriff auf Attribute werden in Abschnitt 7.4 erläutert. Das Themengebiet Threads und Nebenläufigkeit schließt Abschnitt 7.5 mit der Vorstellung verschiedener Besonderheiten beim Einsatz von Threads ab. Im Speziellen widmen wir uns dem Beenden von Threads sowie der zeitgesteuerten Ausführung von Aufgaben mithilfe der Klassen `Timer` und `TimerTask`. Viele Aufgaben im Bereich von Multithreading werden mit den in JDK 5 eingeführten Concurrency Utilities deutlich erleichtert. Abschnitt 7.6 geht darauf ein. Nachdem damit wichtige Grundlagen und Bestandteile der nebenläufigen Programmierung erläutert sind, fehlt nur noch ein wichtiges Thema: Multithreading und Swing. Eine detaillierte Behandlung anhand einiger Beispiele erfolgt nicht in diesem Kapitel, sondern in Abschnitt 8.6.

Grundlagen zu Parallelität und Nebenläufigkeit

Moderne Betriebssysteme beherrschen sogenanntes *Multitasking* und führen mehrere Programme gleichzeitig aus oder vermitteln dem Benutzer zumindest die Illusion, dass verschiedene Programme gleichzeitig ausgeführt würden. Wie eingangs erwähnt, besitzen modernere Computer zum Teil mehrere Prozessoren oder Prozessoren mit mehreren

Kernen und erlauben so, dass das Betriebssystem verschiedene Programme direkt auf alle verfügbaren Prozessoren bzw. Kerne aufteilen kann. Allerdings kann zu jedem Zeitpunkt jeder Prozessor nur genau ein Programm ausführen. Sollen mehrere Programme parallel ausgeführt werden, dann müssen diese intelligent auf die jeweiligen Prozessoren aufgeteilt werden. Bei Systemen mit nur einer CPU ist dies bereits bei zwei Programmen der Fall. Ganz allgemein gilt für eine n -Prozessor-Maschine¹, dass eine Verteilung bei mehr als n Programmen notwendig wird. Ein sogenannter *Scheduler* bestimmt anhand verschiedener Kriterien, welches Programm auf welchem Prozessor ausgeführt wird, und führt bei Bedarf eine Umschaltung des gerade ausgeführten Programms auf ein anderes durch. Etwas später wird das zuvor unterbrochene Programm an gleicher Stelle fortgesetzt. Diese Vorgehensweise sorgt dafür, dass die Ausführung jeweils in kleinen Schritten erfolgt, und erlaubt es, andere Programme minimal zeitlich versetzt und damit scheinbar parallel ausführen zu können.

Laufende Programme können wiederum aus »schwergewichtigen« *Prozessen* und »leichtgewichtigen« *Threads* bestehen. Prozesse sind im Gegensatz zu Threads bzgl. des verwendeten Speichers voneinander abgeschottet und belegen dort unterschiedliche Bereiche. Dadurch beeinflusst ein abstürzender Prozess andere Prozesse nicht. Allerdings erschwert dies auch die Zusammenarbeit und Kommunikation der Prozesse untereinander. Dazu verwendet man eine spezielle Form der Kommunikation, die Interprozesskommunikation. Diese wird durch die sogenannte *Middleware* geregelt, die den Transport von Daten zwischen Anwendungen, das sogenannte *Messaging*, ermöglicht. Die Interprozesskommunikation kann nur über Standards erfolgen, nicht aber durch normale Methodenaufrufe. Über Rechner- und Betriebssystemgrenzen hinweg setzt man dazu RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) usw. ein. Auf demselben Rechner existieren leichtgewichtige Alternativen aus dem Webservices-Umfeld, etwa SOAP (Simple Object Access Protocol) oder REST-Architektur (REpresentational State Transfer).

Ein Java-Programm wird im Betriebssystem durch einen Prozess einer JVM ausgeführt. Ein solcher Prozess kann wiederum verschiedene eigene Threads starten. Alle Threads in einer JVM teilen sich dann den gleichen Adressraum und Speicher. Änderungen an Variablen sind dadurch theoretisch für alle Threads sichtbar. In der Realität ist es durch das Java-Memory-Modell jedoch etwas komplizierter. Abschnitt 7.4 geht detailliert darauf ein. Damit sich zwei Threads nicht gegenseitig stören, müssen sich diese abstimmen, wenn sie auf dieselben Variablen oder Ressourcen zugreifen wollen. Dazu kann man sogenannte *kritische Bereiche* definieren, die zu einer Zeit exklusiv immer nur von einem Thread betreten werden können. Damit vermeidet man konkurrierende Zugriffe auf gemeinsame Daten durch verschiedene Threads und erreicht so eine *Synchronisierung*. Allerdings birgt dies die Gefahr, dass sich Threads gegenseitig behindern oder sogar blockieren. Eine solche Situation nennt man *Verklemmung* (oder *Deadlock*) und muss vom Entwickler selbst verhindert werden. Strategien dazu stellt Abschnitt 7.2.1 vor.

¹Vereinfachend wird hier kein Unterschied zwischen n CPUs und n Rechenkernen gemacht.

7.1 Threads und Runnables

Jeder Thread entspricht in Java einer Instanz der Klasse `Thread` oder einer davon abgeleiteten Klasse. Wenn man über Threads spricht, ist es wichtig, zwei Dinge zu unterscheiden:

1. Den tatsächlichen Thread, der eine Aufgabe ausführt und vom Betriebssystem erzeugt und verwaltet wird.
2. Das zugehörige `Thread`-Objekt, das den zuvor genannten Thread des Betriebssystems repräsentiert und Steuerungsmöglichkeiten auf diesen bietet.

Die JVM erzeugt und startet automatisch einen speziellen Thread, den man `main-Thread` nennt, da dieser die `main()`-Methode der Applikation ausführt. Ausgehend von diesem Thread können weitere Threads erzeugt und gestartet werden.

Betrachten wir zunächst, wie wir eine auszuführende Aufgabe beschreiben, bevor wir uns dem Ausführen von Threads zuwenden.

7.1.1 Definition der auszuführenden Aufgabe

Zur Beschreibung der auszuführenden Aufgaben eines Threads nutzt man die Klasse `Thread` selbst oder das Interface `Runnable`. Beide bieten eine `run()`-Methode, die mit »Leben gefüllt« werden muss. Das kann auf zwei Arten geschehen:

1. Ableiten von der Klasse `Thread` und implementieren der `run()`-Methode
2. Implementieren des Interface `Runnable` und der `run()`-Methode

Betrachten wir beide Varianten anhand zweier einfacher Klassen `CountingThread` und `DatePrinter`. Abbildung 7-1 zeigt das zugehörige Klassendiagramm.

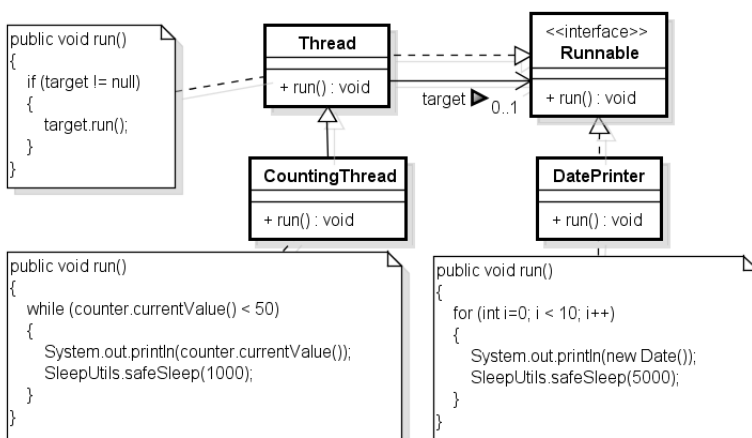


Abbildung 7-1 Thread und Runnable

Die Klasse `CountingThread` erweitert die Klasse `Thread`. Die Implementierung der `run()`-Methode erhöht im Abstand von einer Sekunde einen Zähler und gibt dessen Wert anschließend auf der Konsole aus. Die Klasse `DatePrinter` implementiert das Interface `Runnable` und gibt im Takt von fünf Sekunden das aktuelle Datum aus. Beide Klassen verwenden zum Warten eine Utility-Klasse `SleepUtils`, die wir später in Abschnitt 7.1.3 kennenlernen werden.

In beiden Realisierungen wird zunächst die `run()`-Methode des Threads ausgeführt. Wird einem `Thread`-Objekt ein `Runnable`-Objekt übergeben, so delegiert die `run()`-Methode des `Thread`-Objekts die Ausführung an die `run()`-Methode des `Runnable`-Objekts. Diese zweite Variante ist zu bevorzugen, da sie OO-technisch sauberer ist: Es findet keine Ableitung von einer Utility-Klasse statt und die Funktionalität ist als Einheit gekapselt. Allerdings muss zur Ausführung ein neues `Thread`-Objekt als Laufzeitcontainer für ein `Runnable`-Objekt erzeugt oder bereitgestellt werden.

7.1.2 Start, Ausführung und Ende von Threads

Starten von Threads

Die Konstruktion eines neuen `Thread`-Objekts führt nicht dazu, dass ein neuer Ausführungspfad abgespalten wird. Erst durch Aufruf der `start()`-Methode entsteht ein eigenständiger Thread: Das Programm arbeitet nach diesem Aufruf weiter, ohne blockierend auf das Ende des gestarteten Threads zu warten. In folgendem Beispiel werden die zwei vorgestellten unterschiedlichen Realisierungen von Threads jeweils erzeugt und gestartet:

```
public static void main(final String[] args)
{
    final Thread derivedThread = new CountingThread();
    derivedThread.start();

    final Thread threadWithRunnable = new Thread(new DatePrinter());
    threadWithRunnable.start();
}
```

Listing 7.1 Ausführbar als 'THREADEXAMPLE'

Ausführen von Threads

Nach Aufruf von `start()` kommt es zu einer Verarbeitung und zur Ausführung des Threads durch den Thread-Scheduler. In der Folgezeit wird die `run()`-Methode so lange ausgeführt, bis deren letztes Statement erreicht ist (oder ein Abbruch erfolgt, etwa durch Auslösen einer Exception). Bis dahin wird der Thread normalerweise bei der Abarbeitung der in der `run()`-Methode aufgeführten Anweisungen einige Male vom Thread-Scheduler unterbrochen, um andere Threads auszuführen. Über die Methode `isAlive()` kann man ein `Thread`-Objekt befragen, ob es noch aktiv ist und vom Thread-Scheduler verwaltet wird. Die Methode liefert dann den Wert `true`.

Ein `Thread`-Objekt führt seine durch die `run()`-Methode beschriebene Aufgabe allerdings nur genau einmal nebenläufig aus. Eine Wiederholung ist nicht möglich: Ein erneuter Aufruf von `start()` löst eine `java.lang.IllegalThreadStateException` aus.

Achtung: Versehentlicher direkter Aufruf von `run()`

Manchmal sieht man Sourcecode, der die Methode `run()` direkt aufruft, um einen neuen Thread zu erzeugen und die Aufgabe parallel auszuführen. **Das funktioniert jedoch nicht!** Stattdessen werden die Anweisungen der Methode `run()` einfach synchron im momentan aktiven Thread wie jede normale andere Methode ausgeführt. Dieser Fehler ist relativ schwierig zu finden, da die gewünschten Aktionen ausgeführt werden — in diesem Fall allerdings nicht nebenläufig.

Ende der Ausführung von Threads

Nachdem das letzte Statement der `run()`-Methode ausgeführt wurde, endet auch der Thread. Das zugehörige `Thread`-Objekt bleibt jedoch weiter erhalten, stellt aber nur noch ein Objekt wie jedes andere dar. Man kann weiterhin auf Attribute des Threads zugreifen, um beispielsweise die Ergebnisse einer Berechnung zu ermitteln. Ein Aufruf von `isAlive()` liefert dann den Wert `false`.

Zum Teil sollen Bearbeitungen, d. h. die Ausführung der `run()`-Methode, zu einem beliebigen Zeitpunkt abgebrochen werden. Leider lassen sich Threads nicht so einfach beenden wie starten. Es gibt zwar im API eine zu `start()` korrespondierende Methode `stop()`, doch diese ist als `deprecated` markiert, da sie verschiedene Probleme auslösen kann. Zum Verständnis der Details müssen wir zunächst einige andere Themen besprechen. Abschnitt 7.5.3 greift das Thema »Beenden von Threads« erneut auf und geht detaillierter auf zugrunde liegende Probleme und mögliche Lösungen ein.

Thread-Prioritäten

Jeder Thread besitzt eine Ausführungspriorität in Form eines Zahlenwerts im Bereich von 1 bis 10, entsprechend den Konstanten `Thread.MIN_PRIORITY` und `Thread.MAX_PRIORITY`. Diese Priorität beeinflusst, wie der Thread-Scheduler zu aktivierende Threads auswählt. Threads höherer Priorität werden normalerweise bevorzugt, allerdings wird dies nicht garantiert, und jede JVM oder das Betriebssystem können dies anders handhaben. Dadurch können beispielsweise Threads mit der Priorität n und $n + 1$ durch den Thread-Scheduler ohne Unterschied zur Ausführung ausgewählt werden.

Beim Erzeugen übernimmt ein Thread die Priorität des erzeugenden Threads, die in der Regel dem Wert `Thread.NORM_PRIORITY` (5) entspricht. Diese kann nachträglich über die Methoden `getPriority()` und `setPriority(int)` abgefragt und verändert werden. Dabei sind allerdings nur Werte aus dem durch die obigen Konstanten

definierten Wertebereich erlaubt.² Häufig muss die Priorität nicht angepasst werden. Hierbei gibt es zwei Ausnahmen: Für im Hintergrund zu erledigende, nicht zeitkritische Aufgaben kann beispielsweise der Wert 3 verwendet werden. Für zeitkritische Berechnungen bietet sich eine Priorität nahe `MAX_PRIORITY` an, etwa der Wert 8. Doug Lea gibt in seinem Buch »Concurrent Programming in Java« [46] folgende Empfehlungen:

Tabelle 7-1 Empfehlungen für Thread-Prioritäten

Priorität	Verwendungszweck
10	Krisenmanagement
7 – 9	Interaktive, ereignisgesteuerte Aufgaben
4 – 6	Normalfall und I/O-abhängige Aufgaben
2 – 3	Berechnungen im Hintergrund
1	Unwichtige Aufgaben

7.1.3 Lebenszyklus von Threads und Thread-Zustände

Threads haben einen definierten Lebenszyklus, der durch eine festgelegte Menge an Zuständen beschrieben wird. Gültige Zustände sind als innere `enum`-Aufzählung `State` in der Klasse `Thread` definiert. Durch Aufruf der Methode `getState()` kann man den momentanen Thread-Zustand ermitteln. Mögliche Rückgabewerte sind in Tabelle 7-2 aufgelistet und kurz beschrieben.

Tabelle 7-2 Thread-Zustände

Thread-Zustand	Beschreibung
<code>NEW</code>	Der Thread wurde erzeugt, ist aber noch nicht gestartet.
<code>RUNNABLE</code>	Der Thread ist lauffähig oder wird gerade ausgeführt.
<code>BLOCKED</code>	Der Thread ist in seiner Ausführung blockiert und wartet auf den Zutritt in einen kritischen Bereich (vgl. Abschnitt 7.2.1).
<code>WAITING</code>	Der Thread wartet unbestimmte Zeit auf eine Benachrichtigung durch einen anderen Thread (vgl. Abschnitt 7.3).
<code>TIMED_WAITING</code>	Identisch mit <code>WAITING</code> , allerdings wird maximal eine angegebene Zeitdauer auf eine Benachrichtigung gewartet.
<code>TERMINATED</code>	Der Thread ist beendet.

²Ein Versuch, einen Wert außerhalb dieses Wertebereichs zu setzen, führt zu einer `IllegalArgumentException`.

Für das Verständnis der Zusammenarbeit von Threads ist es wichtig, den Lebenszyklus von Threads und die dabei eingenommenen Thread-Zustände zu kennen. Im Folgenden stelle ich daher die Bedingungen und Auslöser für Zustandswechsel vor.

Nachdem ein Thread erzeugt wurde, wechselt dieser durch den Aufruf seiner `start()`-Methode in den Zustand `RUNNABLE`. Meistens gibt es mehrere Threads in diesem Zustand. Da auf einer 1-Prozessor-Maschine jedoch jeweils nur ein Thread ausgeführt werden kann, ist es Aufgabe des *Thread-Schedulers*, alle vorhandenen Threads zu überwachen und den nächsten auszuführenden zu bestimmen. Dazu wählt er einen Thread aus allen denjenigen aus, die sich im Zustand `RUNNABLE` befinden. Der Gewählte wird dann aktiv (dies wird nicht durch einen eigenen Zustand beschrieben, man kann sich allerdings einen »künstlichen« Unterzustand `ACTIVE` im Zustand `RUNNABLE` vorstellen). Auf einer 1-Prozessor-Maschine existiert immer genau ein aktiver Thread. Auf einer Mehrprozessormaschine kann es jedoch – abhängig von der JVM – auch mehrere aktive Threads geben. In beiden Fällen gibt es jedoch beliebig viele Threads, die zur Ausführung bereit sind (Zustand `RUNNABLE`). Abbildung 7-2 stellt die definierten Thread-Zustände und mögliche Zustandswechsel dar.

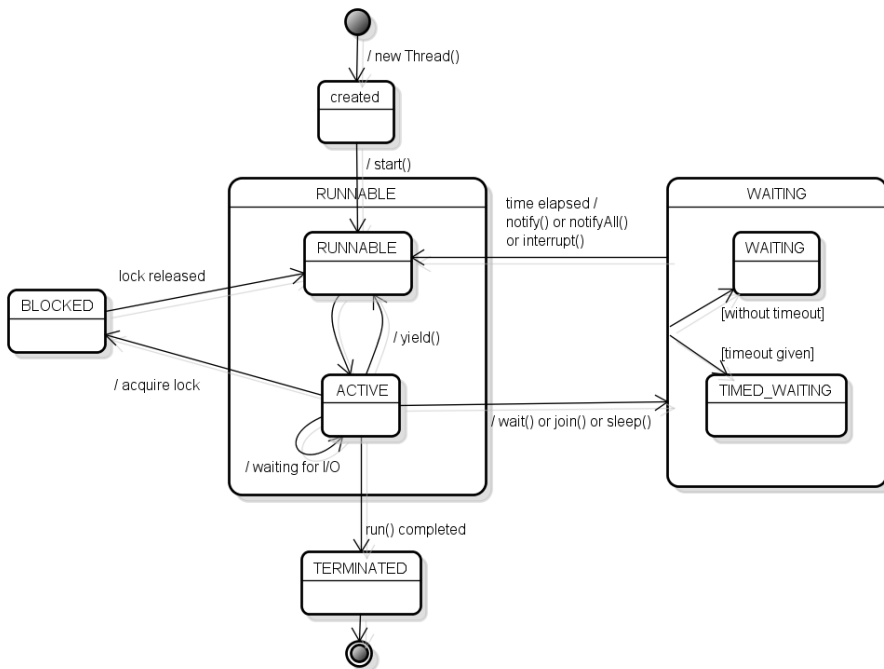


Abbildung 7-2 Thread-Zustände

Nicht jeder Thread im Zustand `RUNNABLE` ist aber wirklich immer lauffähig. Er kann in einem speziellen »blockiert«-Zustand verharren: Ein Thread, der gerade läuft, kann durch das Warten auf eine Ressource, etwa einen Dateizugriff, am Weiterarbeiten gehindert werden und damit in seiner Ausführung blockiert sein. Der Zustand `RUNNABLE`

wird dabei jedoch nicht verlassen. Diese Situation ist nicht mit dem Zustand `BLOCKED` zu verwechseln, der durch Warten auf die Zuteilung des Zutritts zu einem momentan durch einen anderen Thread belegten kritischen Bereich ausgelöst wird. Im Zustand `BLOCKED` erhält ein Thread keine Rechenzeit und konkurriert dadurch nicht mit anderen Threads um Zuteilung des Prozessors. Gleiches gilt für die beiden Wartezustände `WAITING` und `TIMED_WAITING`.

Nehmen wir eine normale, nicht blockierende Abarbeitung der Kommandos des aktiven Threads an, so können durch Aufruf der nun vorgestellten Methoden spezielle Zustandswechsel selbst initiiert werden.

yield() Der aktive Thread kann über den Aufruf der statischen Methode `yield()` in den Zustand `RUNNABLE` zurückversetzt werden. Dadurch wird anderen Threads die Gelegenheit zur Ausführung gegeben. Besonders für lang dauernde Berechnungen scheint es sinnvoll, diese Methode von Zeit zu Zeit aufzurufen. Da die Wahl des nächsten auszuführenden Threads durch den Thread-Scheduler prioritätsorientiert, aber zufällig erfolgt und der gerade angehaltene Thread sofort wieder gewählt werden kann, ist jedoch der Aufruf der im Folgenden beschriebenen Methode `sleep()` zu bevorzugen.

sleep() Durch Aufruf der statischen Methode `sleep()` wird der momentan aktive Thread für eine angegebene Zeitdauer »schlafen« gelegt. Er wechselt in den Zustand `TIMED_WAITING`. Dadurch verbraucht er keine CPU-Zeit und erlaubt anderen Threads deren Ausführung. Nachdem die angegebene Wartezeit abgelaufen ist, wechselt der ruhende Thread automatisch in den Zustand `RUNNABLE`, was jedoch nicht zum sofortigen Wiederaufnehmen seiner Ausführung führt. Gegebenenfalls werden zunächst noch andere Threads im Zustand `RUNNABLE` ausgeführt.

Achtung: Statische Methoden nicht über Objekte aufrufen

An den beiden statischen Methoden `yield()` und `sleep()` kann man sehr schön verdeutlichen, warum es irreführend ist, wenn man statische Methoden so ausführt, als ob sie Objektmethoden wären. Nehmen wir an, der `main`-Thread wäre aktiv und ein Thread `threadX` lauffähig. Es werden folgende Aufrufe ausgeführt:

```
threadX.sleep(2000);
threadX.yield();
```

Auf den ersten Blick meint man, der Thread `threadX` würde zunächst zwei Sekunden pausieren und danach kurz die Kontrolle abgeben. Tatsächlich wirken sich beide Methoden jedoch auf den zum Zeitpunkt des Methodenaufrufs aktiven Thread aus. Dies ist zunächst der laufende `main`-Thread. Auf welchen der beiden Threads sich der Aufruf von `yield()` auswirkt, ist zufällig und nicht vorhersagbar.

wait () Bei der Zusammenarbeit mehrerer Threads kann einer von diesen seine Ausführung unterbrechen wollen, bis eine spezielle Bedingung erfüllt ist. Dies ist beispielsweise der Fall, wenn Berechnungsergebnisse zur sinnvollen Weiterarbeit benötigt werden. Durch einen Aufruf der Objektmethode `wait ()` erfolgt dann ein Wechsel in den Zustand `WAITING` bzw. `TIMED_WAITING`, je nachdem, ob eine maximale Wartezeit übergeben wurde oder nicht. In diesem Zustand verweilt der Thread, bis entweder ein anderer Thread die Methode `notify ()` bzw. `notifyAll ()` aufruft, um das Eintreten einer Bedingung zu signalisieren, oder eine optional angegebene Wartezeit verstrichen ist.

Die in diesem Absatz genannten Methoden `wait ()`, `notify ()` bzw. `notifyAll ()` zur Kommunikation von Threads stammen nicht, wie man zunächst vermuten könnte, aus der Klasse `Thread`, sondern aus der Klasse `Object`. Dies ist dadurch begründet, dass mehrere Threads auf zu schützenden Daten eines Objekts arbeiten und die Threads untereinander abgestimmt werden müssen (vgl. Abschnitt 7.2).

Unterbrechungswünsche durch Aufruf von `interrupt ()` äußern

Bei der Kommunikation von Threads, auf die ich in Abschnitt 7.3 detaillierter eingehe, kann ein Thread einen anderen Thread in seiner Abarbeitung unterbrechen bzw. beenden wollen. Dazu dient die Methode `interrupt ()` der Klasse `Thread`. Diese hat jedoch keine unmittelbare unterbrechende Wirkung, sondern ist lediglich als Aufforderung oder Hinweis zu verstehen. Durch die JVM wird beim empfangenden Thread ein spezielles `Interrupted`-Flag gesetzt.

Ein Empfänger dieser Aufforderung muss geeignet auf diese Situation reagieren. Einem gerade aktiven Thread ist dies unmittelbar möglich. Dazu prüft er über einen Aufruf der Methode `isInterrupted ()` der Klasse `Thread`, ob er eine solche Aufforderung empfangen hat, d. h., ob das Flag gesetzt ist. In diesem Fall sollte die Abarbeitung der Anweisungen in der `run ()`-Methode beendet werden. Die dazu notwendige Prüfung ist gegebenenfalls mehrfach innerhalb der `run ()`-Methode auszuführen, um eine zeitnahe Reaktion auf einen Unterbrechungswunsch zu garantieren.

Komplizierter wird dies, wenn ein Thread momentan nicht aktiv ist, weil er eine bestimmte Zeitspanne pausiert (`sleep ()`) oder aber auf ein bestimmtes Ereignis wartet (`wait ()`). Kommt es während der Wartezeit zu einem Aufruf von `interrupt ()`, so wird zwar in beiden Fällen das Flag gesetzt, eine Reaktion ist jedoch nicht sofort möglich, da der Thread noch nicht aktiv ist. Daher wird von der JVM das Warten abgebrochen und der Thread wechselt in den Zustand `RUNNABLE`. Bei einer anschließenden Aktivierung durch den Thread-Scheduler wird von der JVM eine `java.lang.InterruptedException` ausgelöst.³ Da es sich um eine `Checked Exception` handelt, muss diese propagiert oder mit einem `catch`-Block behandelt werden (vgl. Abschnitt 4.7.1). Dabei ist ein Implementierungsdetail zu beachten.

³Daher wird diese Exception in den Signaturen der Methoden `sleep ()` und `wait ()` definiert.

Behandlung von `interrupt()` und `InterruptedException` Existiert ein korrespondierender `catch (InterruptedException ex)`-Block, so wird dieser ausgeführt. Leider sieht man dort häufig in etwa folgende »Behandlung«:

```
try
{
    Thread.sleep(duration);
}
catch (final InterruptedException e)
{
    // ACHTUNG: unzureichende Behandlung
    e.printStackTrace();
}
```

Dieses Vorgehen ist nahezu immer problematisch. Die JVM löscht nämlich bei Eintritt in den `catch`-Block das zuvor gesetzte `InterruptedException`-Flag! Dadurch führt eine mit einem »leeren« `catch`-Block abgefangene `InterruptedException` dazu, dass ein Thread einen Unterbrechungswunsch nicht mehr erkennen kann und daher auch nicht terminiert.⁴

Wie geht man also mit der Situation um? Eine konsistente Abfrage des `InterruptedException`-Flags mit der Methode `isInterrupted()` wird ermöglicht, wenn man im `catch`-Block über einen expliziten Aufruf der Methode `interrupt()` dieses Flag erneut setzt:

```
while (!Thread.currentThread().isInterrupted())
{
    doSomething();

    try
    {
        Thread.sleep(duration);
    }
    catch (final InterruptedException e)
    {
        // Flag erneut setzen und dadurch Abbruch ermöglichen
        Thread.currentThread().interrupt();
    }
}
```

Die gezeigte Schleifenkonstruktion kann man dazu einsetzen, Threads gezielt zu beenden. Abschnitt 7.5.3 beschreibt gebräuchliche Alternativen.

⁴Wenn man den `catch`-Block derart implementiert, so kommt es zu einem unterschiedlichen Programmverhalten abhängig vom momentanen Thread-Zustand, wartend oder aktiv. Eine solche Inkonsistenz ist zu vermeiden. Hängt das Programmverhalten von der zeitlichen Abfolge der Anweisungen ab, so spricht man auch von *Race Conditions*. Details beschreibt Abschnitt 7.2.1.

Entwurf einer Utility-Klasse

Da Aufrufe von `Thread.sleep()` in der Praxis immer wieder zum Verzögern oder Warten eingesetzt werden, und dabei fälschlicherweise häufig der `catch`-Block leer implementiert wird, bietet sich der Entwurf einer Utility-Klasse `SleepUtils` an. Diese stellt zwei überladene Methoden `safeSleep()` bereit, die für eine korrekte Verarbeitung der Exception sorgen und das `Interrupted`-Flag erneut setzen:

```
public final class SleepUtils
{
    public static void safeSleep(final TimeUnit timeUnit, final long duration)
    {
        safeSleep(timeUnit.toMillis(duration));
    }

    public static void safeSleep(final long durationInMilliSecs)
    {
        try
        {
            Thread.sleep(durationInMilliSecs);
        }
        catch (final InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }

    private SleepUtils()
    {
    }
}
```

Hinweis: Leere `catch`-Blöcke für `InterruptedException`

Ruft man die statische Methode `sleep(long)` der Klasse `Thread` auf, um die Ausführung des eigenen Threads kurzfristig zu unterbrechen, so kann der `catch`-Block unter Umständen leer implementiert werden. Dies gilt allerdings nur, wenn keine Zusammenarbeit und Kommunikation mit anderen Threads über `interrupt()` erfolgt. Der Grund ist einfach: Kennen sich Threads gegenseitig nicht, können sich diese auch nicht anhalten wollen. Eine `InterruptedException` kann nicht auftreten, muss aber bekanntermaßen behandelt oder propagiert werden.

Kommunizieren Threads miteinander, deutet eine `InterruptedException` darauf hin, dass ein Thread einen anderen unterbrechen und evtl. sogar stoppen möchte, und darf somit natürlich nicht ignoriert werden. Möchte man auf Nummer sicher gehen, kann man generell die zuvor gezeigte Lösung verwenden, um Problemen aus dem Weg zugehen. Eine Abfrage des `Interrupted`-Flags ist somit jederzeit bei Bedarf konsistent möglich.

7.2 Zusammenarbeit von Threads

Bei der Zusammenarbeit von Threads und dem Zugriff auf gemeinsam benutzte Daten müssen einige Dinge beachtet werden. Wir betrachten zunächst Situationen, in denen es zu Zugriffsproblemen und Zweideutigkeiten kommt. Zu deren Vermeidung werden anschließend Locks, Monitore und kritische Bereiche vorgestellt.

7.2.1 Konkurrierende Datenzugriffe

Der Zugriff auf gemeinsame Daten muss beim Einsatz von Multithreading immer aufeinander abgestimmt erfolgen, um Konsistenzprobleme zu vermeiden. Ein einziger unsynchronisierter, konkurrierender Datenzugriff kann bereits zu massiven Problemen führen. Meistens sind solche Situationen schwer reproduzierbar, wodurch eine Fehlersuche sehr mühselig wird. Zudem können bereits minimale Änderungen im Zusammenspiel von Threads zu einem Verschwinden der Probleme oder zu ihrem erstmaligen Auftreten führen. Wenn das Ergebnis einer Berechnung vom zeitlichen Verhalten bestimmter Anweisungen abhängt, so wird dies als *Race Condition* bezeichnet.

Betrachten wir dies beispielhaft für den Start einer Applikation, die aus mehreren Komponenten besteht, die während ihrer Startphase auf eine Klasse, realisiert gemäß dem SINGLETON-Muster (vgl. Abschnitt 12.1.4), zugreifen. Der Zugriff geschieht auf folgende problematische `getInstance()`-Methode:

```
// ACHTUNG: SCHLECHT !!!
public static BadSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BadSingleton();
    }
    return INSTANCE;
}
```

Leider sieht man diese Art der Implementierung selbst in Produktivcode häufiger. Untersuchen wir nun, was daran problematisch ist. Nehmen wir dazu vereinfachend an, während des Starts der Applikation würden zwei Komponenten (Thread 1 und Thread 2) auf diese `getInstance()`-Methode zugreifen. Erfolgt der Aufruf nahezu zeitgleich, so besteht die Gefahr, dass einer der aufrufenden Threads direkt nach der `null`-Prüfung durch den Thread-Scheduler unterbrochen und anschließend der andere Thread aktiviert wird. Beide Threads »sehen« damit einen `null`-Wert für die Variable `INSTANCE`. Dadurch erfolgt der Konstruktoraufruf und die Zuweisung mehrfach. Für dieses Beispiel erhält man demnach zwei Instanzen eines Singletons: Beide Threads besitzen dann ihre eigene Instanz! Dies widerspricht natürlich vollständig dem Gedanken des Singletons.

Die Wahrscheinlichkeit für eine solche *Race Condition* durch einen gemeinsamen Zugriff erhöht sich mit der Verweildauer eines Threads im Konstruktor der Klasse `BadSingleton`, beispielsweise, wenn dort aufwendige Initialisierungen stattfinden. In diesem Szenario ruft Thread 2 die Methode `getInstance()` auf, während Thread 1 noch

im Konstruktor beschäftigt ist: Die Variable `INSTANCE` ist dann aber noch nicht zugewiesen und hat damit immer noch den Wert `null`.

Durch das Java-Memory-Modell, dessen Implikationen wir in Abschnitt 7.4 genauer betrachten werden, ist es jedoch prinzipiell egal, ob Thread 1 bereits einen Wert `INSTANCE != null` produziert hat. Selbst wenn Thread 1 bereits seit Stunden erfolgreich mit einer solchen Referenz arbeitet, ist durch die JLS nicht sichergestellt, dass Thread 2 die Änderungen von Thread 1 an der Variablen `INSTANCE` je mitbekommt. Das liegt daran, dass Threads normalerweise einige Variablen in Thread-lokalen Caches zwischenspeichern und die Werte nur zu definierten Zeitpunkten mit dem Hauptspeicher abgleichen. Da für Thread 2 ohne das Schlüsselwort `synchronized` kein Zwang besteht, sich beim Zugriff auf `getInstance()` neu mit dem Hauptspeicher zu synchronisieren, ist es durchaus möglich, dass er mit einem alten Wert der Variablen `INSTANCE` arbeitet. Die geschilderte Situation wird im Java-Memory-Modell als **Sichtbarkeit (Visibility)** bezeichnet.

Anhand dieses Beispiels sehen wir, wie wichtig es ist, den Zugriff auf von mehreren Threads gemeinsam benutzte Variablen zu synchronisieren. Dies ist durch die Definition sogenannter **kritischer Bereiche** möglich, zu denen jeweils nur ein Thread zur gleichen Zeit »Zutritt« hat. Im einfachsten Fall kann man dazu das Schlüsselwort `synchronized` auf Methodenebene wie folgt nutzen:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

Eine Thread-sichere Implementierung, die zudem Nebenläufigkeit unterstützt, wird in Abschnitt 12.1.4 detailliert vorgestellt.

7.2.2 Locks, Monitore und kritische Bereiche

Betrachten wir nun sogenannte Locks und Monitore, die bei der Funktion des Schlüsselworts `synchronized` eine entscheidende Rolle spielen.

Um konkurrierende Zugriffe mehrerer Threads zu verhindern, existiert eine Zugangskontrolle, auch **Monitor** genannt. Ein solcher Monitor sorgt dafür, dass der Zutritt zu einem kritischen Bereich immer nur einem einzelnen Thread gewährt wird. Dazu wird eine spezielle Sperre (**Lock**) genutzt, die durch das Schlüsselwort `synchronized` gesteuert wird. In Java verwaltet jedes Objekt zwei Sperren: Eine davon bezieht sich auf die Objektreferenz (`this`) und dient dazu, Objektmethoden zu synchronisieren. Die andere wirkt auf die Klassenreferenz (`class`-Referenz), womit die Synchronisation für statische Methoden möglich ist.

Vor Eintritt in einen mit `synchronized` markierten kritischen Bereich wird von der JVM automatisch geprüft, ob der angeforderte Lock verfügbar ist. In diesem Fall

wird der Lock vergeben und der anfragende Thread erhält Zutritt zu dem kritischen Bereich. Nach Abarbeitung des `synchronized`-Blocks wird der Lock ebenfalls automatisch wieder freigegeben. Kann dagegen ein Lock nicht akquiriert werden, so muss der anfragende, momentan aktive Thread darauf warten, wird inaktiv und in eine spezielle Warteliste des Locks eingetragen. Es findet ein gegenseitiger Ausschluss statt: **Ein durch einen kritischen Bereich geschützter Zugriff auf eine gemeinsame Ressource serialisiert die Abarbeitung mehrerer Threads**. Statt einer parallelen Abarbeitung kommt es zu einer zeitlich versetzten Bearbeitung. Je mehr Threads um den Zugriff auf einen Lock konkurrieren, desto mehr leidet die Nebenläufigkeit. Abbildung 7-3 zeigt dies symbolisch für drei Threads T_1 , T_2 und T_3 , die eine ungeschützte und eine durch `synchronized` geschützte Methode ausführen. Im ersten Fall kann die Abarbeitung parallel erfolgen (oder zumindest quasi parallel durch mehrfache Thread-Wechsel). Im zweiten Fall müssen Threads warten und werden sequenziell abgearbeitet (aber nicht unbedingt in der Reihenfolge des versuchten Eintritts in den kritischen Bereich).

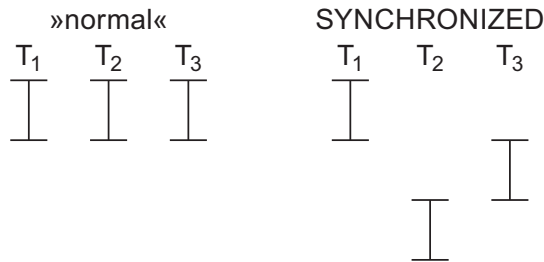


Abbildung 7-3 Serialisierung der Abarbeitung durch Locks

Ein Monitor sorgt sowohl dafür, dass ein verfügbarer Lock vergeben wird, als auch dafür, dass ein anfragender Thread bei Nichtverfügbarkeit in die Liste der auf diesen Lock wartenden Threads aufgenommen wird. Wird ein Lock freigegeben, so erhält ihn ein beliebiger Thread aus der Warteliste dieses Locks. Dieser Thread kann daraufhin seine Ausführung fortsetzen. Diese Aktionen laufen für den Entwickler unsichtbar ab.

Tipp: Locks und die Analogie zum Wartezimmer

Das Vorgehen beim Synchronisieren mit Locks und Monitoren kann man sich mithilfe der Analogie eines Wartezimmers beim Arzt klarmachen. Der Arzt stellt die durch den Lock geschützte Ressource dar, die exklusiv von einem Patienten (Thread) belegt wird. Kommen weitere Patienten zum Arzt, so müssen diese zunächst im Wartezimmer (Warteliste) Platz nehmen und gehen dort in einen Wartezustand, einen Zustand der Blockierung für andere Dinge. Die Arzthelferin (Monitor) gewährt den Zutritt zum Doktor, sobald dieser wieder Zeit hat, also der vorherige Patient den Lock zurückgegeben hat.

Kritische Bereiche und das Schlüsselwort `synchronized`

Zur Definition kritischer Bereiche mit dem Schlüsselwort `synchronized` existieren zwei Varianten, die ich im Folgenden vorstelle.

`synchronized`-Methode Die bekannteste und bereits beschriebene Variante zur Definition eines kritischen Bereichs ist das *Synchronisieren der gesamten Methode* durch Erweitern der Signatur um das Schlüsselwort `synchronized`:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

Ein Thread muss zur Ausführung der Methode dann zunächst den entsprechenden Lock zugeteilt bekommen, ansonsten wird er in der weiteren Ausführung angehalten und wartet auf die Freigabe des benötigten Locks, um diesen zunächst zu akquirieren und anschließend mit seiner Berechnung fortfahren zu können.

In der Regel greifen weitere Methoden auf ein zu schützendes Attribut zu. In diesem Fall müssen alle diese Methoden synchronisiert werden. Aufgrund der Arbeitsweise der Locks wissen wir, dass dadurch alle `synchronized`-Methoden des Objekts für Zugriffe durch andere Threads gesperrt sind. *Diese Art der Synchronisierung erschwert somit die Nebenläufigkeit.* Die parallele Ausführung weiterer als `synchronized` deklarierter Methoden eines Objekts ist nicht möglich.

Was kann an dem `synchronized` für eine Methode außerdem noch problematisch sein? Ist beispielsweise die Ausführung einer geschützten Methode zeitintensiv, so müssen andere Threads lange warten, um den Lock zu erhalten. Damit ist klar, dass ein Synchronisieren über Methoden bei dem Wunsch nach viel Parallelität nicht optimal ist. Betrachten wir nun eine andere, elegantere Synchronisierungsvariante.

Achtung: Verlust der Synchronisierung durch Überschreiben

`synchronized`-Methoden können so überschrieben werden, dass diese nicht mehr `synchronized` sind. Das synchronisierte Verhalten betrifft dann nur die deklarierende Klasse.

Synchronisationsobjekt Die unbekanntere, aber oft sinnvollere Möglichkeit zur Definition eines kritischen Bereichs ist das *Synchronisieren eines Abschnitts innerhalb von Methoden*. Man nutzt dazu ein sogenanntes *Synchronisationsobjekt*. Prinzipiell kann jedes beliebige Objekt dazu verwendet werden, seinen Lock zur Verfügung zu stellen. Um Anwendungsfehler zu vermeiden und die Semantik eines solchen Synchronisationsobjekts besonders herauszustellen, sollte jedoch bevorzugt eine finale Objektreferenz vom Typ `Object` genutzt werden, im folgenden Beispiel `LOCK` genannt.

Zum Eintritt in einen durch `synchronized` (LOCK) geschützten kritischen Bereich muss ein Thread den Lock dieses speziellen Objekts erhalten.

Synchronisationsobjekte nutzt man, um Sperrungen auf feingranularer Ebene durchzuführen, d. h., um kleine Blöcke innerhalb von Methoden zu schützen. Am Beispiel der folgenden Methoden `calcValue1()` und `calcValue2()` zeige ich dies anhand von Berechnungen, die sich zu verschiedenen Zeitpunkten in ihrer Abarbeitung gegenseitig ausschließen sollen. Statt beide Methoden über `synchronized` exklusiv auszuführen und damit Nebenläufigkeit zu verhindern, werden in diesen Methoden nur die wirklich kritischen Abschnitte über `synchronized` (LOCK) vor einer gleichzeitigen Ausführung geschützt:

```
// Gemeinsame genutzte und daher zu schützende Daten
private final List<Integer> sharedData = new ArrayList<Integer>();

// Synchronisationsobjekt
private final Object LOCK = new Object();

public int calcValue1()
{
    // Blockiert calcValue2()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    // Aktionen ohne Lock, calcValue2() kann ausgeführt werden

    return 1;
}

public int calcValue2()
{
    // Blockiert calcValue1()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    // Aktionen ohne Lock, calcValue1() kann ausgeführt werden

    // Blockiert calcValue1()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    return 2;
}
```

Mehr Nebenläufigkeit kann man erreichen, wenn man mehrere solcher Lock-Objekte für alle diejenigen Attribute einführt, die geschützt werden sollen. Dadurch werden konkurrierende Zugriffe auf andere gemeinsame Attribute nicht mehr blockiert und jeweils immer nur der benötigte Zugriff exklusiv ausgeführt. Eine Idee ist, das zu schützende Objekt selbst als Synchronisationsobjekt zu verwenden. Allerdings ist dies nur möglich, wenn sichergestellt ist, dass sich die Referenz darauf nicht ändert, diese also `final` ist.

Ansonsten kann es zu einer `java.lang.IllegalMonitorStateException` kommen. Darauf gehe ich in Abschnitt 7.3 genauer ein.

Analogie von Synchronisationsobjekt und `synchronized`-Methode

Die Möglichkeiten durch den Einsatz von Synchronisationsobjekten sind mächtiger als die von `synchronized`-Methoden. Daher kann man das Verhalten von `synchronized`-Methoden durch den Einsatz von Synchronisationsobjekten abbilden, die die `this`-Referenz verwenden, und den synchronisierten Abschnitt auf die komplette Methode ausdehnen. Die folgenden Zeilen

```
void method()
{
    synchronized (this)
    {
        // ...
    }
}
```

entsprechen dem Verhalten des Schlüsselworts `synchronized` auf Methodenebene:

```
synchronized void method()
{
    // ...
}
```

Für statische Methoden wird der Lock durch das Synchronisieren auf die `class`-Variable beschrieben. Folgende Varianten sind damit äquivalent:

```
static void staticMethod()
{
    synchronized (SynchronizationExample.class)
    {
        // ...
    }
}
```

und

```
static synchronized void staticMethod()
{
    // ...
}
```

Tipp: Fallstricke beim Synchronisieren

Ein Schutz vor konkurrierenden Zugriffen muss immer vollständig erfolgen. **Ein einziger nicht synchronisierter Zugriff auf ein zu schützendes Attribut reicht aus, um Multithreading-Probleme zu verursachen.** Dies kann schnell geschehen, wenn man verschiedene Synchronisationsvarianten nicht konsequent, sondern beispielsweise gemischt verwendet und somit potenziell den Lock auf das »falsche« Synchronisationsobjekt hält.

7.2.3 Deadlocks und Starvation

Wie bereits eingangs erwähnt, sind *Deadlocks* Situationen, in denen Threads sich gegenseitig blockieren. Unter *Starvation* versteht man Situationen, in denen es für einen oder mehrere Threads kein Vorankommen im Programmfluss mehr gibt. Beim Einsatz von Multithreading kann beides leicht auftreten.

Deadlocks

Wenn ein beliebiger Thread 1 den Lock auf Objekt A belegt und versucht, den Lock auf Objekt B zu erhalten, kann es zu einem sogenannten *Deadlock* kommen, wenn ein anderer Thread 2 bereits den Lock auf Objekt B hält und seinerseits wiederum versucht, den Lock auf Objekt A zu bekommen. Beide warten daraufhin endlos. In der Informatik ist die exklusive Belegung von Ressourcen auch als das *Philosophenproblem* bekannt. Etwas vereinfacht treffen sich zwei Philosophen P1 und P2 zum Essen an einem Tisch, allerdings haben sie nur eine Gabel und ein Messer. Dies zeigt Abbildung 7-4.

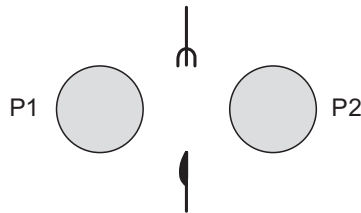


Abbildung 7-4 Ressourcenproblem

Wählt Philosoph P1 immer zuerst die Gabel und nimmt Philosoph P2 immer zunächst das Messer, so kann es bei ungünstiger zeitlicher Abfolge (nahezu zeitgleichem Zugriff auf die Besteck-Ressource) dazu kommen, dass beide niemals etwas essen, da ihnen die jeweils andere Ressource fehlt. Halten sich beide an eine vorgegebene Reihenfolge erst Gabel, dann Messer oder umgekehrt, so kann es niemals zu einer gegenseitigen Blockierung kommen.

Tipp: Vermeidung von Deadlocks

Deadlocks lassen sich am besten dadurch vermeiden, dass man Locks und Ressourcen immer in der gleichen Reihenfolge belegt, etwa gemäß dem folgenden Schema: Lock A, Lock B, Unlock B, Unlock A.

Hält man sich an diese Regel, so vermeidet man Situationen, in denen ein Thread auf einen anderen wartet und keiner von beiden ausgeführt werden kann, weil beide jeweils auf eine bereits belegte Ressource zugreifen wollen.

Starvation

Wenn der Besitzer einer Ressource diese nicht wieder freigibt, obwohl er sie nicht weiter benötigt, kann es zu dem *Starvation* genannten Phänomen kommen. Andere auf diese Ressource wartende Threads sind endlos blockiert, da sie vergeblich auf den Erhalt der benötigten Ressource warten. Verhalten sich alle Beteiligten allerdings fair, so muss zwar manchmal einer auf den anderen warten (wenn dieser gerade die benötigten Ressourcen besitzt), aber alle können ihre Aufgabe auf jeden Fall nacheinander bzw. abwechselnd ausführen.

Java bietet mit dem Schlüsselwort `synchronized` leider keine Unterbrechbarkeit oder Time-outs beim Zugriff auf Locks. Durch die mit JDK 5 eingeführten und im Anschluss vorgestellten Lock-Klassen wird dieses Manko beseitigt.

Tipp: Regeln zur Verwendung von `synchronized`

Folgende Dinge sollten beim Programmieren mit Threads und Synchronisierung beachtet werden, um Deadlocks und andere Probleme möglichst auszuschließen:

- Ein Thread kann den Lock von einem oder die Locks von verschiedenen Objekten besitzen. Er erhält Zutritt zu den derart geschützten kritischen Bereichen.
- Ein über `synchronized` geschützter Bereich sollte möglichst kurz sein. Ideal ist es, wenn nur die wirklich zu schützenden Operationen synchronisiert werden. Nicht zu schützende Teile sollten in andere Methoden verlagert oder außerhalb des geschützten Bereichs durchgeführt werden.
- Aus einem mit `synchronized` geschützten Bereich sollten keine blockierenden Aufrufe (z. B. `read()` eines `InputStream`) erfolgen, da der Thread den Lock während des Wartens nicht abgibt. Auch Aufrufe von `Thread.sleep()` sind zu vermeiden, da auch hier keine Freigabe des Locks erfolgt. Andere Threads werden ansonsten möglicherweise länger blockiert.
- Aus dem vorherigen Punkt ergibt sich indirekt folgender Tipp: ***Wenn man einen Lock hält, sollte man möglichst keine Methoden anderer Objekte aufrufen (oder dabei zumindest große Vorsicht walten lassen)***. Dadurch vermeidet man Aufrufe, die zu Deadlocks führen können.
- Ein Objekt kann mehrere Methoden `synchronized` definieren. Besitzt ein Thread den Lock, können diese Methoden sich gegenseitig aufrufen, ohne dass bei jedem Methodenaufruf erneut versucht wird, den Lock zu bekommen. Wäre dies nicht so, würde sich ein Thread selbst blockieren, wenn er versuchen würde, andere `synchronized`-Methoden aufzurufen. Locks nennt man daher eintrittsinvariant oder auch ***reentrant***^a. Basierend darauf können `synchronized`-Methoden rekursiv aufgerufen werden.
- Nicht durch `synchronized` geschützte Methoden können jederzeit von jedem Thread aufgerufen werden.

^aMan bezeichnet eine Methode als reentrant, wenn sie von mehreren Threads problemlos ohne gegenseitige Beeinflussung gleichzeitig ausgeführt werden kann.

7.2.4 Kritische Bereiche und das Interface Lock

Zur Beschreibung kritischer Bereiche stehen seit JDK 5 ergänzend zu den impliziten Sperren über `synchronized` explizite Möglichkeiten über Locks zur Verfügung. Die für diese Funktionalität wichtigsten Interfaces sind `Lock`, `ReadWriteLock` und `Condition` sowie die beiden Klassen `ReentrantLock` und `ReentrantReadWriteLock` aus dem Package `java.util.concurrent.locks`. Durch deren Einsatz können einige Schwächen des in die Sprache integrierten Lock- und Synchronisationsmechanismus umgangen bzw. behoben werden. Die wichtigsten neuen Möglichkeiten sind in folgender Aufzählung genannt:

- Ein Thread kann unterbrechbar versuchen, auf einen Lock zuzugreifen.
- Ein Thread kann nicht nur unbeschränkt lange auf den Zugriff eines Locks warten, wie dies auch bei `synchronized` geschieht, sondern auch mit einer maximalen Wartezeit.
- Es werden unterschiedliche Arten von Locks unterstützt. Beispielsweise sind dies sogenannte Read-Write-Locks, die mehrere parallele Lesezugriffe erlauben, solange der zugehörige Write-Lock nicht vergeben wurde.
- Lock-Objekte können nun nicht nur einen gewissen Block aufspannen, sondern beliebige Bereiche. Dadurch können sie im Gegensatz zu `synchronized` in verschiedenen Klassen über Block- und Methodengrenzen hinweg eingesetzt werden.

Grundlagen von Locks

Lock-Objekte stellen eine Erweiterung von Synchronisationsobjekten dar. Letztere verwendet man bekanntermaßen wie folgt:

```
synchronized (lockObj)
{
    // Kritischer Bereich
}
```

Da Lock-Objekte nicht auf den in die JVM integrierten Mechanismen der Monitore beruhen, muss die Definition eines kritischen Bereichs explizit über Methodenaufrufe ausprogrammiert werden. Zum Sperren ruft man die Methode `lock()` auf einem Lock-Objekt auf. Soll der kritische Bereich beendet werden, muss explizit ein Methodenaufruf von `unlock()` erfolgen:

```
final Lock lockObj = new ReentrantLock();
lockObj.lock();
try
{
    // Kritischer Bereich
}
finally
{
    lockObj.unlock();
}
```

Dieses Beispiel zeigt den Einsatz des Interface `Lock` und der konkreten Realisierung `ReentrantLock`, um ein zu `synchronized` kompatibles Verhalten zu erreichen, das Eintrittsinvarianz bietet und rekursive Methodenaufrufe erlaubt. Die Implementierungen von Locks garantieren zudem ein bzgl. der Sichtbarkeit von Änderungen zu `synchronized` kompatibles Verhalten. Darauf geht der folgende Praxistipp ein.

Tipp: Konsistenz von Locks und `synchronized`

Wenn man das Java-Memory-Modell (vgl. Abschnitt 7.4) kennt, verwundert zunächst, dass Locks die Sichtbarkeit von Änderungen analog zu `synchronized` herstellen, da hier keine explizite Synchronisierung zu sehen ist. Diese sorgt neben dem gegenseitigen Ausschluss von Threads auch für Konsistenz bzgl. der verwendeten Attribute und für die Einhaltung der sogenannten **Happens-before**-Ordnung. Das bedeutet vereinfacht, dass Wertänderungen von Attributen innerhalb eines `synchronized`-Blocks anschließend für andere lesend zugreifende Threads sichtbar sind. Die Realisierungen von Locks nutzen `volatile`-Attribute, die ebenfalls eine Happens-before-Ordnung garantieren und somit Konsistenz von Änderungen sicherstellen.

Parallelität durch den Einsatz von Read-Write-Locks

Häufig werden deutlich mehr Lese- als Schreibzugriffe erfolgen. Für diese Anwendungsfälle existieren sogenannte Read-Write-Locks, die beide Arten von Zugriffen schützen können. Sie ermöglichen mehreren Threads, Read-Locks zu halten, um parallele Lesezugriffe zu erlauben, solange kein Thread das Write-Lock akquiriert hat. Das beschriebene Verhalten wird durch die Klasse `ReentrantReadWriteLock` realisiert, die das Interface `ReadWriteLock` implementiert. Dieses bietet Zugriff auf zwei spezielle Locks und ist folgendermaßen definiert:

```
public interface ReadWriteLock
{
    public Lock readLock();
    public Lock writeLock();
}
```

Während ein Thread das Write-Lock besitzt, können andere Threads das Read-Lock nicht akquirieren. Finden andererseits noch Leseaktionen statt, während ein Thread Zugriff auf das Write-Lock bekommen möchte, muss dieser warten, bis alle lesenden Threads ihre Read-Locks freigegeben haben. Dies kann dazu führen, dass Schreibzugriffe stark verzögert werden, wenn es sehr viele Lesezugriffe gibt. Ist ein derartiges Verhalten nicht gewünscht, so kann man bei der Konstruktion der Klasse `ReentrantReadWriteLock` einen Parameter übergeben, der dafür sorgt, dass die Klasse sich »fair« verhält. In diesem Modus werden Threads daran gehindert, das Read-Lock zu akquirieren, solange es noch Threads gibt, die auf das Write-Lock warten.

Beispiel: synchronized durch Locks ersetzen

Dieser Abschnitt zeigt, wie man einen Benachrichtigungsmechanismus gemäß dem BEOBACHTER-Muster (vgl. Abschnitt 12.3.8) mit gegenseitigem Ausschluss per `synchronized` auf die Möglichkeiten der Lock-Klassen umstellen kann.

Nehmen wir dazu an, dass sich verschiedene Beobachter vom Typ `ChangeListener` für Änderungsmitteilungen an- bzw. abmelden können. Die entsprechenden `add/removeChangeListener(ChangeListener)`-Methoden sind jeweils `synchronized` definiert. Zur Darstellung von Veränderungen am Objektzustand ist hier exemplarisch lediglich eine Methode `changeState(int)` definiert. Diese löst eine durch `synchronized` geschützte Änderung des Objektzustands aus, die an alle angemeldeten Beobachter propagiert werden soll. Dazu dient die synchronisierte Methode `notifyChangeListeners()`.

Im Folgenden ist eine gebräuchliche, Thread-sichere, aber bzgl. Nebenläufigkeit nicht optimale Realisierung mit einer `ArrayList<ChangeListener>` und synchronisierten Methoden gezeigt:

```
private final List<ChangeListener> listeners = new ArrayList<ChangeListener>();

public synchronized void addChangeListener(final ChangeListener listener)
{
    listeners.add(listener);
}

public synchronized void removeChangeListener(final ChangeListener listener)
{
    listeners.remove(listener);
}

private synchronized void notifyChangeListeners()
{
    final Iterator<ChangeListener> it = listeners.iterator();
    while (it.hasNext())
    {
        final ChangeListener listener = it.next();

        listener.update();
    }
}

public synchronized void changeState(final int newValue)
{
    state = newValue;
    notifyChangeListeners();
}
```

Listing 7.2 Ausführbar als 'OBSERVERWITHSYNCHRONIZEDEXAMPLE'

Durch die Synchronisierung auf Methodenebene erfolgt der gegenseitige Ausschluss für Lese- und Schreiboperationen gleichermaßen. Änderungen bzw. Schreibzugriffe auf das Attribut `listeners` sind nach einer initialen Registrierungsphase in der Praxis eher selten. Mitteilungen über Zustandsänderungen an angemeldete Beobachter sind dagegen die Regel. Sie erfordern lediglich lesenden Zugriff auf die Liste der gespeicherten

Beobachter. Reine Lesezugriffe sind voneinander unabhängig, blockieren sich bei dieser Art der Umsetzung aber gegenseitig.

Wünschenswert wäre es daher, wenn Lesezugriffe parallel erfolgen würden. Ein Schreibzugriff muss dagegen exklusiv ausgeführt werden: Gleichzeitig dürfen weder andere Schreib- noch Lesezugriffe erfolgen. Diese Anforderung mit `synchronized` umzusetzen ist nicht trivial. Verwenden wir stattdessen die Klasse `ReentrantReadWriteLock`, so wird die Realisierung der Aufgabe relativ einfach: Über die Methode `readLock()` kann ein Lock angefordert werden, der parallele Lesezugriffe erlaubt und lediglich blockiert, wenn parallel ein Lock zum Schreiben angefordert wurde. Mit der Methode `writeLock()` erhält man exklusiven Schreib- und Lesezugriff.

Für die Realisierung des Benachrichtigungsmechanismus ergeben sich die in den folgenden Listings gezeigten Änderungen. Erstens werden Schreibzugriffe auf das Attribut `listeners` durch einen speziellen Lock (ermittelt über Aufruf der Methode `writeLock()`) sowohl beim Hinzufügen als auch beim Löschen von Beobachtern gesichert:

```
private final List<ChangeListener> listeners = new ArrayList<ChangeListener>();

private final ReadWriteLock lockObj = new ReentrantReadWriteLock();
private final Lock readLock = lockObj.readLock();
private final Lock writeLock = lockObj.writeLock();

public void addChangeListener(final ChangeListener listener)
{
    writeLock.lock();
    try
    {
        listeners.add(listener);
    }
    finally
    {
        writeLock.unlock();
    }
}

public void removeChangeListener(final ChangeListener listener)
{
    writeLock.lock();
    try
    {
        listeners.remove(listener);
    }
    finally
    {
        writeLock.unlock();
    }
}
```

Zweitens benötigt eine Benachrichtigung mit `notifyChangeListeners()` nur lesen Zugriff. Daher nutzt man hier die Methode `readLock()` zum Zugriff auf ein Lock-Objekt. Schließlich verbleibt die Methode `changeState(int)`, die keine Zugriffe auf das Attribut `listeners` ausführt. Sie wird weiterhin über das Lock des Objekts synchronisiert, wodurch man Inkonsistenzen im eigentlichen Objektzustand verhindert. Es

wird hier jedoch nur noch der wirklich kritische Bereich der Zuweisung über einen `synchronized-Block` geschützt:

```
private void notifyChangeListeners()
{
    readLock.lock();
    try
    {
        for (final ChangeListener listener : listeners)
        {
            listener.update();
        }
    }
    finally
    {
        readLock.unlock();
    }
}

public void changeState(final int newValue)
{
    synchronized(this)
    {
        state = newValue;
    }
    notifyChangeListeners();
}
```

Diese Umsetzung erlaubt Nebenläufigkeit, erfordert aber den Einsatz mehrerer Locks. Datenstrukturen, auf denen deutlich häufiger Lese- als Schreibzugriffe erfolgen, sollte man nicht, wie hier zur Demonstration der Arbeitsweise von Locks geschehen, selbst implementieren. Dafür existieren mit den Klassen `CopyOnWriteArrayList<E>` und `CopyOnWriteArraySet<E>` vorgefertigte Bausteine, deren Arbeitsweise in Abschnitt 7.6.1 genauer beschrieben wird.

7.3 Kommunikation von Threads

Bis jetzt haben wir Möglichkeiten kennengelernt, Thread-sicher auf gemeinsam benutzte Daten zuzugreifen. In der Zusammenarbeit von Threads benötigt man aber häufig außerdem Möglichkeiten zur Abstimmung zwischen Threads. Ein Beispiel ist das sogenannte **Producer-Consumer-Problem**. Hierbei geht es darum, dass ein Erzeuger (Producer) gewisse Daten herstellt und ein Konsument (Consumer) diese verbraucht.

Im Folgenden stelle ich verschiedene Formen der Realisierung von `Producer`- und `Consumer`-Klassen vor, um mögliche Probleme bei der Kommunikation von Threads und deren Lösungen nachvollziehen zu können. Die vermittelten Grundlagen erleichtern das Verständnis der Zusammenhänge beim Einsatz von Multithreading.

7.3.1 Kommunikation mit Synchronisation

Zunächst betrachten wir, wie eine Kommunikation über eine gemeinsam genutzte Datenstruktur und den Einsatz von Synchronisation gelöst werden kann.

In diesem Beispiel erzeugt der Producer periodisch im Takt von einer Sekunde irgendwelche Daten vom Typ `Item` und legt sie in der gemeinsam genutzten Datenstruktur `items` vom Typ `List<Item>` ab:

```
public static class Producer implements Runnable
{
    private final List<Item> items;
    private final long      sleepTime;

    public Producer(final List<Item> items, final long sleepTime)
    {
        this.items = items;
        this.sleepTime = sleepTime;
    }

    public void run()
    {
        int counter = 0;

        while (!Thread.currentThread().isInterrupted())
        {
            // Erzeugen eines Items
            final Item newItem = new Item("Item " + counter);
            System.out.println("Producing ... " + newItem);

            SleepUtils.safeSleep(sleepTime);

            // Lock akquirieren, dann exklusiv zugreifen und hinzufügen
            synchronized (items)
            {
                items.add(newItem);
                System.out.println("Produced " + newItem);
            }
            // Lock wird automatisch freigeben

            counter++;
        }
    }
}
```

Der Consumer schaut zyklisch nach, ob bereits Daten für ihn vorliegen, und holt diese dann aus der Liste `items` heraus. Dieses Verfahren des aktiven Wartens wird auch **Busy Waiting** genannt und ist in der Regel zu vermeiden – später dazu mehr.

```

public static class Consumer implements Runnable
{
    private final List<Item> items;
    private final long      sleepTime;

    public Consumer(final List<Item> items, final long sleepTime)
    {
        this.items = items;
        this.sleepTime = sleepTime;
    }

    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            // Status-Flag ist als lokale Variable immer Thread-sicher
            boolean noItems = true;
            while (noItems)
            {
                // Lock akquirieren, dann exklusiv zugreifen und auslesen
                synchronized (items)
                {
                    noItems = (items.size() == 0);
                    if (noItems)
                    {
                        System.out.println("Consumer waiting for items ...");
                    }
                }
                // Lock wird automatisch freigegeben

                // sleep() nicht in synchronized aufrufen, Lock wird
                // nicht freigegeben => Producer könnte so niemals
                // während der Wartezeit Items erzeugen und ablegen
                SleepUtils.safeSleep(sleepTime);
            }

            System.out.println("Consuming " + items.remove(0));
        }
    }
}

```

Sowohl Producer als auch Consumer sind als Runnables realisiert und werden wie folgt gestartet:

```

public static void main(final String[] args)
{
    final List<Item> items = new LinkedList<Item>();

    new Thread(new Producer(items, 1000)).start();
    new Thread(new Consumer(items, 500)).start();
}

```

Listing 7.3 Ausführbar als 'PRODUCERCONSUMERSYNCHRONISATIONEXAMPLE'

Betrachten wir die Ausgabe, um das Programm zu analysieren:

```
Producing ... [Item] Item 0
Consumer waiting for items ...
Consumer waiting for items ...
Produced [Item] Item 0
Producing ... [Item] Item 1
Consuming [Item] Item 0
Consumer waiting for items ...
Produced [Item] Item 1
Producing ... [Item] Item 2
Consuming [Item] Item 1
Consumer waiting for items ...
Produced [Item] Item 2
Producing ... [Item] Item 3
Consuming [Item] Item 2
Consumer waiting for items ...
usw.
```

Anhand dieser Ausgabe erkennen wir folgende Dinge:

1. Producer und Consumer laufen synchron ab. Außerdem wird bei dieser Realisierung der produzierte Gegenstand immer sofort konsumiert, sobald er verfügbar ist.
2. Das aktive Warten führt zu ständigen Prüfungen. Es kostet Rechenzeit und sollte möglichst vermieden werden, da es elegantere und performantere Lösungsmöglichkeiten gibt.⁵
3. Andere Wartezeiten für Producer und Consumer sind denkbar. Würde man die Wartezeiten vertauschen, so würden in einer Sekunde zwei Gegenstände produziert, der Consumer würde aber nur jeweils einen Gegenstand konsumieren. Diese Wartezeiten simulieren den Fall, dass der Producer schneller arbeitet, als der Consumer die produzierten Gegenstände verarbeiten kann. Der Zwischenspeicher würde irgendwann überlaufen. Es ist daher ratsam, dessen Größe zu beschränken. Wünschenswert ist es, dass ein Producer die Arbeit einstellt, wenn ein maximaler Füllgrad erreicht ist, und ein Consumer so lange konsumiert, bis keine Gegenstände mehr vorhanden sind. Dies lässt sich mit aktivem Warten nur noch schwer adäquat ausdrücken.

Die Kommunikation lässt sich eleganter durch die im Folgenden vorgestellten Methoden `wait()`, `notify()` und `notifyAll()` lösen.

⁵Wird in einfachen Berechnungen allerdings häufig genug `Thread.sleep(long)` aufgerufen, ist der Einsatz nicht ganz so tragisch. Geht die Wartezeit jedoch gegen 0, so erhöht sich die Prozessorlast deutlich.

7.3.2 Kommunikation über die Methoden `wait()`, `notify()` und `notifyAll()`

Zur Kommunikation zwischen Threads existieren die Methoden `wait()`, `notify()` und `notifyAll()`, die nicht auf Threads, sondern auf Objekten aufgerufen werden. In diesem Fall kontrolliert das Objekt über seinen Lock indirekt den Thread. Wie beim Schlüsselwort `synchronized` werden die Verwaltung des Locks und die Warte- und Aufweckarbeiten automatisch durch die JVM erledigt.

- `wait()` – Versetzt den aktiven Thread in den Zustand `WAITING` und der belegte Lock wird freigegeben. Um ein endloses Warten auf ein evtl. nicht eintretendes Ereignis zu verhindern, kann beim Aufruf von `wait()` eine maximale Wartezeit mitgegeben werden, der Thread wechselt dadurch in den Zustand `TIMED_WAITING`.
- `notify()` – Informiert einen beliebigen wartenden Thread und versetzt diesen in den Zustand `RUNNABLE`. Unschönerweise kann man nicht kontrollieren, welcher Thread benachrichtigt wird. Daher sollte man bei der Kommunikation mehrerer Threads immer das nachfolgend beschriebene `notifyAll()` verwenden.
- `notifyAll()` – Informiert alle auf diesen Lock wartenden Threads aus dem Wartesaal und versetzt diese in den Zustand `RUNNABLE`.

Nach dieser etwas theoretischen Erklärung möchte ich das Zusammenspiel kurz erläutern: Wenn ein Thread *A* während seiner Bearbeitung Daten benötigt oder auf das Eintreten einer speziellen Bedingung warten möchte, so kann er die Methode `wait()` eines Objekts `obj` abrufen. Ein anderer Thread *B* kann dadurch bei Bedarf den benötigten Lock erhalten, Berechnungen durchführen und Ergebnisse produzieren. Danach informiert er einen oder mehrere auf den Lock dieses Objekts `obj` wartende Threads durch Aufruf der Methode `notify()` oder bevorzugt `notifyAll()` auf der Objektreferenz `obj`. Einer dieser Threads erhält dann den Lock des Objekts `obj`: Dies kann der wartende Thread *A* oder ein beliebiger anderer sein. Was dies im Einzelnen bedeutet, zeigen die folgenden Abschnitte.

Producer-Consumer mit `wait()`, `notify()` und `notifyAll()`

Nachdem wir nun die Methoden zur Steuerung der Kommunikation von Threads kennengelernt haben, nutzen wir sie, um das Producer-Consumer-Beispiel zu vereinfachen.

Zunächst setze ich die Methode `wait()` bewusst etwas naiv ein, um ein Problem beim Aufruf ohne vorherige und/oder nachfolgende Prüfung zeigen zu können. In jedem Fall lässt sich das Busy Waiting vermeiden. Zunächst wird die `run()`-Methode der `Consumer`-Klasse folgendermaßen modifiziert:

```
// ACHTUNG: Problematische Realisierung !!!
public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        synchronized (items)
        {
            try
            {
                System.out.println("Consumer waiting ...");
                items.wait();
                // ACHTUNG: Unsicherer Zugriff
                System.out.println("Consuming " + items.remove(0));
            }
            catch (final InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
        }

        SleepUtils.safeSleep(sleepTime);
    }
}
}
```

Damit der Consumer nicht endlos wartet, muss der Producer einen Aufruf der Methode `notify()` ausführen, der zum Aufwachen des Consumers führt. Die `run()`-Methode im Producer wird wie folgt realisiert:

```
public void run()
{
    int counter = 0;

    while (!Thread.currentThread().isInterrupted())
    {
        final Item newItem = new Item("Item " + counter);
        System.out.println("Producing ... " + newItem);

        SleepUtils.safeSleep(sleepTime);

        synchronized (items)
        {
            items.add(newItem);
            System.out.println("Produced " + newItem);
            // Informiere wartende Threads
            items.notifyAll();
        }

        counter++;
    }
}
}
```

Listing 7.4 Ausführbar als 'PRODUCERCONSUMEREXAMPLE'

Wie man sieht, ist die Kommunikation nun klarer und nicht nur mit einigen Tricks wie beim Busy Waiting möglich.

Nachdem ein wartender Thread aufgeweckt wurde, sollte er prüfen, ob tatsächlich die erwartete Situation eingetreten ist. *Dies ist notwendig, da man beim Aufruf von `wait()` und `notify()` bzw. `notifyAll()` keine Bedingung angeben kann.* Warten

mehrere Threads auf unterschiedliche Dinge, könnte es zu Problemen kommen, wenn ein Aufwachen als Eintritt der erwarteten Situation gewertet würde. Um dieses Problem zu umgehen, sollte der aufgeweckte Thread eine erneute Prüfung vornehmen. Dabei hat sich folgendes Idiom als hilfreich herausgestellt:

```
while (!condition)
    wait();
```

Doug Lea schlägt in seinem Buch »Concurrent Programming in Java« [46] vor, die Prüfung von Bedingungen in eigene Methoden auszulagern. Dieses Vorgehen ist sehr empfehlenswert und trägt deutlich zur Lesbarkeit bei. Um beispielsweise im Consumer zu prüfen, ob Daten zur Verarbeitung bereitgestellt wurden, kann folgende Methode `waitForItemsAvailable(List<Item>)` realisiert werden:

```
private static void waitForItemsAvailable(final List<Item> items) throws
    InterruptedException
{
    while (items.size() == 0)
        items.wait();
}
```

Neben der besseren Lesbarkeit gibt es noch einen triftigen Grund für diese Art der Umsetzung: Erfolgt eine Benachrichtigung mit `notify()`, bevor darauf mit `wait()` gewartet wird, so geht diese Benachrichtigung verloren und ein Thread wartet dann eventuell bis zum Programmende vergeblich auf das Eintreffen einer Benachrichtigung. Auch dies könnte man über eine `if`-Abfrage vor dem `wait()` lösen. Allerdings verbleibt noch das Problem der Unterscheidbarkeit von Benachrichtigungen. Wird an einigen Stellen auf das Eintreten verschiedener Bedingungen gewartet, so können die Benachrichtigungen auf unterschiedliche Ereignisse über `notify()` bzw. `notifyAll()` nicht unterschieden werden. Daher muss man auch nach dem Aufruf von `wait()` die gewünschte Bedingung erneut prüfen. Eleganter als zwei Prüfungen über `if`-Abfragen ist somit der Einsatz der gezeigten Prüfung in einer Schleife.

Erweiterung auf mehrere Consumer

Ich erweitere das vorherige Beispiel auf mehrere unabhängige Consumer, um auf ein spezielles Problem bei `wait()` und anschließenden Zugriffen auf die gemeinsamen Daten einzugehen. Betrachten wir dazu zunächst folgende Implementierung der `run()`-Methode, die wie bisher nach dem Aufruf von `wait()` ohne Prüfung einer Bedingung einen Zugriff auf das Attribut `items` ausführt:

```

public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        synchronized (items)
        {
            try
            {
                System.out.println(consumerName + " waiting ...");
                items.wait();
                // ACHTUNG FALSCH: Zugriff ohne Prüfung
                final Item item = items.remove(0);
                System.out.println(consumerName + " consuming " + item);
            }
            catch (final InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
        }

        SleepUtils.safeSleep(sleepTime);
    }
}

```

Um die Problematik zu verdeutlichen, schreiben wir folgende `main()`-Methode:

```

public static void main(final String[] args)
{
    final List<Item> items = new LinkedList<Item>();

    new Thread(new Producer(items, 1000)).start();
    new Thread(new Consumer(items, 100, "Consumer 1")).start();
    new Thread(new Consumer(items, 100, "Consumer 2")).start();
    new Thread(new Consumer(items, 100, "Consumer 3")).start();
}

```

Listing 7.5 Ausführbar als 'PRODUCERMULTICONSUMERWRONG1EXAMPLE'

Startet man das Programm durch Aufruf des Ant-Targets `PRODUCERMULTICONSUMERWRONG1EXAMPLE`, treten schnell zwei `IndexOutOfBoundsException` auf. Die Ursache dafür ist, dass bei einem Aufruf von `notifyAll()` alle Consumer aufgeweckt werden. Alle konkurrieren dann um den einen Lock des `items`-Objekts, den sie beim Aufruf von `wait()` abgegeben haben. Der erste Consumer, der den Lock erhält, führt die Zeile

```
final Item item = items.remove(0);
```

aus. Selbst das ist nur dadurch garantiert, dass `wait()` in einem `synchronized`-Block aufgerufen wird. Die beiden anderen Consumer führen diese Zeile später bei Erhalt des Locks auch aus. Da durch den ersten Aufruf die gespeicherten Daten entnommen wurden, schlägt jeder weitere Aufruf `items.remove(0)` mit einer `Exception` fehl, wodurch zwei der drei Consumer beendet werden und anschließend nur noch ein Consumer aktiv ist.

Dieses Beispiel verdeutlicht die Problematik, dass beim Warten und Benachrichtigen keine Bedingungen angegeben werden können. Folglich kann auch nicht garantiert werden, dass die Bedingung, auf die von einem speziellen `wait()` gewartet wurde, nach dem Aufwachen tatsächlich eingetreten ist.

Aber selbst wenn die Bedingung, auf die gewartet wurde, eingetreten ist, so kann man nicht sicher sein, dass beim Erhalt des Locks die Bedingung noch gilt. In diesem Beispiel mit den drei Consumern wird dies sehr deutlich. Wenn der Producer `notifyAll()` aufruft, dann sind auf jeden Fall Daten in der Datenstruktur `items` vorhanden. Dies gilt aber nicht mehr, wenn ein beliebiger Consumer aktiviert wird und vor ihm ein anderer an der Reihe war. Andere Threads können den Lock bereits erhalten und Modifikationen an den Daten durchgeführt haben. Diese können dazu führen, dass die erwartete Bedingung nicht mehr erfüllt ist. Die erste Idee ist, die Bedingung anschließend erneut zu prüfen:

```
System.out.println(consumerName + " waiting ...");

items.wait();

if (items.size() > 0)
    System.out.println(consumerName + " consuming " + items.remove(0));
else
    System.out.println(consumerName + " --- item already consumed by " +
        "other consumer!");

SleepUtils.safeSleep(sleepTime);
```

Listing 7.6 Ausführbar als 'PRODUCERMULTICONSUMERWRONG2EXAMPLE'

Allerdings werden durch diese Art der Realisierung immer alle Consumer ausgewertet und in der Abarbeitung fortgesetzt. Dadurch muss zusätzlich eine Fehlerbehandlung in den Sourcecode integriert werden, die sich negativ auf die Verständlichkeit auswirkt. Folgende Ausgabe verdeutlicht die zuvor gemachten Aussagen:

```
Consumer 3 consuming [Item] Item 3
Consumer 3 waiting ...
Consumer 1 -- item already consumed by other consumer!
Consumer 1 waiting ...
Consumer 2 -- item already consumed by other consumer!
Consumer 2 waiting ...
```

Um die genannten Probleme zu adressieren, sollte man `wait()` besser, wie anfangs vorgeschlagen, innerhalb einer `while`-Schleife ausführen und dort erneut die Bedingung prüfen. Dazu nutzen wir die bereits vorgestellte Methode `waitForItemsAvailable(List<Item>)` wie folgt:

```
System.out.println(consumerName + " waiting ...");

waitForItemsAvailable(items);

// Zugriff immer durch waitForItemsAvailable() sicher
final Item item = items.remove(0);
System.out.println(consumerName + " consuming " + item);

SleepUtils.safeSleep(sleepTime);
```

Listing 7.7 Ausführbar als 'PRODUCERMULTICONSUMEREXAMPLE'

Als Folge warten anfangs alle Consumer auf das Eintreffen einer Benachrichtigung. Es werden zwar alle Consumer geweckt, allerdings erhält einer von diesen zunächst den Lock und konsumiert. Erhalten in der Folgezeit die anderen Consumer den Lock, prüfen diese zunächst wieder die Bedingung und warten sofort wieder auf das Eintreten der nun nicht mehr gültigen Bedingung.

Erweiterung auf eine Größenbeschränkung

Bisher haben wir lediglich den Fall betrachtet, dass die Consumer genauso schnell verbrauchen, wie der Producer Dinge herstellen kann. Für den Fall, dass der Producer allerdings wesentlich schneller ist, kämen die Consumer nicht mehr hinterher und die Datenstruktur würde immer voller, bildlich gesprochen: Sie würde überlaufen.

Ein Zwischenspeicher begrenzter Kapazität ist daher sinnvoll. Wird dessen Kapazität erreicht, so wartet der Producer darauf, dass die Consumer wieder Platz schaffen. Ist dies der Fall wird er wieder aktiv. Umgekehrt gilt: Sind die Consumer schneller als der Producer, so sollen diese so lange warten, bis wieder Dinge zu konsumieren sind.

Wir erweitern das Beispiel. Wenig objektorientiert wäre es, die Anforderungen in den Klassen selbst zu realisieren. Stattdessen definieren wir eine typsichere Containerklasse, deren Anforderungen durch folgendes Interface beschrieben sind (dadurch sind wir in der Lage, Realisierungen auszutauschen oder miteinander zu vergleichen):

```
public interface FixedSizeContainer<T>
{
    /**
     * put the passed item into this container, blocks if container reached
     * its capacity (the queue is full)
     */
    void putItem(final T item) throws InterruptedException;

    /**
     * returns the next item from this container, blocks if there are no
     * items available (the queue is empty)
     */
    T takeItem() throws InterruptedException;
}
```

Eine erste Realisierung stellt die Klasse `FixedSizeListContainer<T>` dar, die mit einer Liste arbeitet:

```
public static class FixedSizeListContainer<T> implements FixedSizeContainer<T>
{
    private final List<T> queuedItems;
    private final int    maxSize;

    public FixedSizeListContainer(final int maxSize)
    {
        this.queuedItems = new LinkedList<T>();
        this.maxSize = maxSize;
    }

    public synchronized void putItem(final T item) throws InterruptedException
    {
        waitWhileQueueFull();
        // aufwecken von waitWhileQueueEmpty()
        notify();

        queuedItems.add(item);
    }

    public synchronized T takeItem() throws InterruptedException
    {
        waitWhileQueueEmpty();
        // aufwecken von waitWhileQueueFull()
        notify();

        return queuedItems.remove(0);
    }

    private void waitWhileQueueFull() throws InterruptedException
    {
        while (queuedItems.size() == maxSize)
            wait();
    }

    private void waitWhileQueueEmpty() throws InterruptedException
    {
        while (queuedItems.size() == 0)
            wait();
    }
}
```

Die beiden Methoden `putItem(T)` und `takeItem()` zum Einfügen und zum Abholen müssen synchronisiert sein, um gleichzeitige Zugriffe durch Consumer und Producer korrekt bearbeiten zu können. Wenn es keinen Platz mehr für ein neu produziertes Element gibt, wartet der Producer mit `wait()`, bis es wieder Platz gibt. Der Consumer ruft beim Abholen `notify()` auf, so dass danach der Producer informiert wird und prüfen kann, ob wieder Platz vorhanden ist. Beide Wartebedingungen werden, wie zuvor vorgeschlagenen, über die Methoden `waitWhileQueueFull()` bzw. `waitWhileQueueEmpty()` sichergestellt.

In folgendem Beispiel erzeugen wir wieder drei Consumer, die von einem Producer versorgt werden. Die Ausgangssituation entspricht exakt den vorherigen Beispielen – es wird lediglich die neue Datenstruktur mit einer willkürlichen Beschränkung auf maximal sieben Elemente zur Kommunikation verwendet.

```

public static void main(String[] args)
{
    final int MAX_QUEUE_SIZE = 7;
    final FixedSizeContainer<Item> items = new FixedSizeListContainer<Item>(
        MAX_QUEUE_SIZE);

    new Thread(new Producer(items, 1000)).start();

    new Thread(new Consumer(items, 100, "Consumer 1")).start();
    new Thread(new Consumer(items, 100, "Consumer 2")).start();
    new Thread(new Consumer(items, 100, "Consumer 3")).start();
}

```

Listing 7.8 Ausführbar als 'PRODUCERMULTICONSUMERSTIZEDEXAMPLE'

Startet man das Ant-Target PRODUCERMULTICONSUMERSTIZEDEXAMPLE, so erhält man tatsächlich genau die gleiche Ausgabe wie zuvor.

Drehen wir nun etwas an den Stellschrauben und lassen den Producer schneller arbeiten. Damit wir die Größenbeschränkung nachvollziehen können, wird hier zudem eine künstliche Pause von zwei Sekunden nach Erzeugung des Producers eingelegt:

```

public static void main(String[] args)
{
    final int MAX_QUEUE_SIZE = 7;
    final FixedSizeContainer<Item> items = new FixedSizeListContainer<Item>(
        MAX_QUEUE_SIZE);

    new Thread(new Producer(items, 100)).start();

    // warte 2 Sekunden, dadurch sieht man die Größenbeschränkung
    SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

    new Thread(new Consumer(items, 1000, "Consumer 1")).start();
    new Thread(new Consumer(items, 1000, "Consumer 2")).start();
    new Thread(new Consumer(items, 1000, "Consumer 3")).start();
}

```

Listing 7.9 Ausführbar als 'PRODUCERMULTICONSUMERSTIZEDEXAMPLE2'

Bedingungen und das Interface Condition

Wir haben zum Ersatz von `synchronized` bereits das Interface `Lock` kennengelernt. Neben den vorgestellten Möglichkeiten erlaubt es außerdem, Bedingungen zu formulieren. Wir haben bisher einem Aufruf von `notify()` keine Wartebedingung zuordnen können. Durch den Einsatz von `Condition`-Objekten ist dies möglich. Diese kann man von einem `Lock`-Objekt erhalten und damit auf das Eintreten verschiedener Bedingungen warten. Die zuvor vorgestellte größenbeschränkte Containerklasse schreiben wir derart um, dass die `synchronized`-Blöcke durch Aufrufe von `lock()` und `unlock()` realisiert werden. Zuvor wurden Bedingungen ausschließlich anhand von Abfragen an die Daten speichernde Liste ermittelt. Wir nutzen in diesem Beispiel zusätzlich zwei `Condition`-Objekte `notEmpty` bzw. `notFull`. Über deren Methoden `await()` und `signal()` bzw. `signalAll()` kann man das Warten auf und Eintreten von Bedin-

gungen klarer als lediglich über Aufrufe der Objektmethoden `wait()`, `notify()` und `notifyAll()` folgendermaßen modellieren:

```
public final class BlockingFixedSizeBuffer<T> implements FixedSizeContainer<T>
{
    private final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    private final List<T> queuedItems;
    private final int maxSize;

    public BlockingFixedSizeBuffer(final int maxSize)
    {
        this.queuedItems = new LinkedList<T>();
        this.maxSize = maxSize;
    }

    public void putItem(final T item) throws InterruptedException
    {
        lock.lock();
        try
        {
            while (maxSize == queuedItems.size())
                notFull.await();

            queuedItems.add(item);
            notEmpty.signal();
        }
        finally
        {
            lock.unlock();
        }
    }

    public T takeItem() throws InterruptedException
    {
        lock.lock();
        try
        {
            while (queuedItems.size() == 0)
                notEmpty.await();

            return queuedItems.remove(0);
        }
        finally
        {
            lock.unlock();
        }
    }

    private void waitWhileQueueFull() throws InterruptedException
    {
        while (maxSize == queuedItems.size())
            notFull.await();
    }

    private void waitWhileQueueEmpty() throws InterruptedException
    {
        while (queuedItems.size() == 0)
            notEmpty.await();
    }
}
```

Trotz des Einsatzes von `Condition`-Objekten ist die Umsetzung noch unübersichtlich und umständlich, da die Implementierung auf einem relativ tiefen Abstraktionsniveau mit vielen Details geschieht. Beim Einsatz von Multithreading sind gemeinsam genutzte, größenbeschränkte Datenstrukturen elementar, um die Kommunikation verschiedener Threads zu steuern.

Abschließend zeige ich den Einsatz der zuvor entwickelten Klasse `BlockingFixedSizeBuffer<T>`. Führt man das Ant-Target `PRODUCERMULTICONSUMER-SIZEDEXAMPLE3` aus, so sieht man, dass diese Realisierung vollständig kompatibel zur Klasse `FixedSizeListContainer<T>` ist.

```
public static void main(String[] args)
{
    final int MAX_QUEUE_SIZE = 7;
    final FixedSizeContainer<Item> items = new BlockingFixedSizeBuffer<Item>(
        MAX_QUEUE_SIZE);

    new Thread(new Producer(items, 100)).start();

    // warte 2 Sekunden, dadurch sieht man die Größenbeschränkung
    SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

    new Thread(new Consumer(items, 1000, "Consumer 1")).start();
    new Thread(new Consumer(items, 1000, "Consumer 2")).start();
    new Thread(new Consumer(items, 1000, "Consumer 3")).start();
}
}
```

Listing 7.10 Ausführbar als 'PRODUCERMULTICONSUMER-SIZEDEXAMPLE3'

Da solche Containerklassen häufig benötigt werden, wurden sie in die `Concurrent Collections` aufgenommen. In Abschnitt 7.6.1 betrachten wir exemplarisch das Interface `java.util.concurrent.BlockingQueue<E>` und einige konkrete Realisierungen, die teilweise in ihrer Funktionalität ähnlich zu der Klasse `BlockingFixedSizeBuffer<T>` sind.

7.3.3 Abstimmung von Threads

Bei der Zusammenarbeit mehrerer Threads sollen manchmal Aufgaben in parallele Teile aufgespalten und später an Synchronisationspunkten wieder zusammengeführt werden. Dies ist mithilfe der Methode `join()` der Klasse `Thread` möglich. Mit JDK 5 wurden als Alternative spezielle Klassen in die später vorgestellten `Concurrency Utilities` integriert (vgl. Abschnitt 7.6).

Die Methoden `isAlive()` und `join()`

Aus den einleitenden Abschnitten ist bereits bekannt, dass man mit der Methode `isAlive()` ein `Thread`-Objekt fragen kann, ob dieses durch einen Aufruf an `start()` als neuer Ausführungspfad gestartet wurde und die Methode `run()` abgearbeitet wird. Vor und nach der Abarbeitung von `run()` stellt ein `Thread` nur ein ganz normales Objekt dar. Ein Aufruf von `isAlive()` liefert dann den Wert `false`.

Startet man Threads zur Erledigung bestimmter Aufgaben und möchte auf deren Ergebnisse warten, so ist dies durch Aufruf der Methode `join()` möglich. Repräsentiert beispielsweise `workerThread` einen Thread, so wartet der momentan aktive Thread durch den Aufruf

```
workerThread.join();
```

synchron, d. h. blockierend, auf die Beendigung des Threads `workerThread`. Erst danach wird der zuvor aktive Thread fortgesetzt.

Achtung: Verwirrende Syntax von `join()`

Diese Schreibweise beim Aufruf von `join()` ist verwirrend, aber trotzdem korrekt. Besser verständlich und objektorientierter wäre in etwa folgende Schreibweise gewesen: `currentThread.join(workerThread);`

Um nicht endlos zu warten, falls der andere Thread niemals endet, kann optional eine Time-out-Zeit angegeben werden, nach der das Warten auf das Ende des Threads abgebrochen wird. Die Realisierung der Methode `join()` kann man sich wie folgt vorstellen:

```
while (isAlive())
    wait(TIMEOUT);
```

Zwei Dinge sind zu beachten: Zum einen sollte ein Aufruf von `join()` nur erfolgen, wenn der Thread, auf den gewartet werden soll, bereits gestartet wurde. Ansonsten würde `isAlive()` den Wert `false` liefern, und es würde dadurch nicht gewartet. Zum anderen kehrt ein Aufruf von `join()` sofort zurück, falls der zu überprüfende Thread bereits beendet ist. Der aufrufende Thread wird augenblicklich fortgesetzt. Verdeutlichen wir uns die Arbeitsweise durch folgendes Programm, ausführbar als Ant-Target `PRODUCERJOINEXAMPLE`:

```
public static void main(String[] args)
{
    final List<Item> items = new LinkedList<Item>();
    final Thread producerThread = new Thread(new Producer(items, 1000));
    producerThread.start();

    try
    {
        // aktueller Thread wird für 5 Sekunden angehalten
        producerThread.join(TimeUnit.SECONDS.toMillis(5));
        System.out.println("after join");
        // 1000 ms Produktionszeit und 5000 ms Wartezeit => ca. 5 Items
        System.out.println("Item-Count after join(): " + items.size());
        // der Producer arbeitet noch 2 Sekunden weiter ...
        SleepUtils.safeSleep(TimeUnit.SECONDS, 2);
    }
    catch (final InterruptedException e)
    {
        Thread.currentThread().interrupt();
    }
}
```

```

// der Producer wird aufgefordert, nun anzuhalten ...
producerThread.interrupt();

// 1000 ms Produktionszeit und 7 s Wartezeit => ca. 7 Items
System.out.println("Item-Count after interrupt(): " + items.size());
}

```

Listing 7.11 Ausführbar als 'PRODUCERJOINEXAMPLE'

In diesem Beispiel wartet der aktuelle Thread fünf Sekunden auf das Ende des `Producer`-Threads. Dieser ist dann aber noch nicht terminiert. Die Ausgabe der Anzahl der produzierten Elemente stellt daher nur einen Zwischenstand dar. Der `Producer`-Thread setzt seine Arbeit fort und wird nach weiteren zwei Sekunden über einen Aufruf von `interrupt()` zum Anhalten aufgefordert.

Wesentlich häufiger soll ein Thread auf das Ende mehrerer Threads warten, bevor weitere Aufgaben ausgeführt werden. Aufgrund der Tatsache, dass `join()` sofort zurückkehrt, wenn ein zu überprüfender Thread beendet ist, kann man einfach mehrere `join()`-Aufrufe in beliebiger Reihenfolge hintereinander ausführen. Die maximale Wartezeit ist nur durch den Thread mit der längsten Ausführungszeit bestimmt.

Die Abstimmung der Ablaufreihenfolge mehrerer Threads lässt sich zwar über Aufrufe an `join()` realisieren, etwa indem abhängige Threads erst nach den Threads gestartet werden, auf deren Beendigung sie warten sollen. Sinnvoller ist es aber, die bereits vordefinierten Klassen aus den Concurrency Utilities (vgl. Abschnitt 7.6) zu verwenden. Diese Klassen liegen nicht im Fokus dieses Buchs. Eine kurze informelle Einführung in deren Arbeitsweise gibt jedoch der folgende Praxistipp »Weitere Synchronizer des Packages `java.util.concurrent`«. Detailliertere Informationen finden Sie u. a. im Buch »Java Concurrency in Practice« von Brian Goetz [26].

Kommunikation über einen Semaphor

Ein häufiger Anwendungsfall bei Multithreading ist, eine begrenzte Anzahl an Ressourcen auf eine größere Anzahl parallel arbeitender Threads zu verteilen. Ein Beispiel sind etwa anfragende Klienten an einen Webserver. Nur eine gewisse Anzahl dieser Anfragen werden tatsächlich parallel mithilfe von Threads bearbeitet. Nicht sofort verarbeitbare Anfragen müssen zunächst warten und werden zur späteren Ausführung zwischengespeichert.

Eine Möglichkeit, diese Funktionalität bereitzustellen, besteht darin, einen sogenannten *Semaphor* zu benutzen. Dieser verwaltet einen Zähler, der mit der Anzahl zur Verfügung stehender Ressourcen initialisiert wird und die momentan verfügbare Anzahl an Ressourcen beschreibt. Jeder anfragende Thread ruft dazu die Methode `acquire()` auf, die den Zähler um eins reduziert, sofern noch Ressourcen verfügbar sind. Ist dies nicht der Fall, so werden diese und alle folgenden Anfragen so lange blockiert, bis wieder mindestens eine Ressource bereitgestellt werden kann. Nach der Bearbeitung sollte ein Thread durch Aufruf der Methode `release()` seine Ressource wieder frei-

geben, und der Zähler wird um eins erhöht. Eine Implementierung als Klasse `SimpleSemaphore` ist gemäß den zuvor beschriebenen Ideen leicht möglich:

```
public final class SimpleSemaphore
{
    private int count;

    public SimpleSemaphore(final int n)
    {
        this.count = n;
    }

    public synchronized void acquire() throws InterruptedException
    {
        waitWhileNoResources();
        count--;
    }

    public synchronized void release()
    {
        count++;
        notify();
    }

    private void waitWhileNoResources() throws InterruptedException
    {
        while (count == 0)
            wait();
    }
}
```

Eine funktionale Erweiterung dieser einfachen Implementierung stellt die Klasse `java.util.concurrent.Semaphore` aus den Concurrency Utilities dar.

Tipp: Weitere Synchronizer des Packages `java.util.concurrent`

Barrieren Müssen sich mehrere Threads abstimmen, so kann man dies durch sogenannte Barrieren realisieren. Diese kann man sich wie Treffpunkte bei einer Radtour vorstellen, die in verschiedene Etappen eingeteilt ist. Jedes Etappenziel besitzt einen Sammelpunkt. Verlieren sich Tourteilnehmer, so treffen sie sich alle wieder an diesen Sammelpunkten und starten gemeinsam von dort die neue Etappe. Mit der Klasse `CyclicBarrier` kann man derartige Treffpunkte mit der zum Weiterfahren erforderlichen Anzahl von Teilnehmern definieren.

Latches Latches lassen sich mit folgender Analogie beschreiben: Bei einem Marathon treffen sich diverse Teilnehmer am Start und warten dort auf den Startschuss, der nach einem Countdown erfolgt. Mit der Klasse `CountDownLatch` kann man dieses Verhalten nachbilden: Ein oder mehrere Threads können sich durch Aufruf einer `await()`-Methode in einen Wartezustand versetzen. Bei der Konstruktion eines `CountDownLatch`-Objekts gibt man die Anzahl der Schritte bis zum Startschuss an. Durch Aufruf der Methode `countDown()` wird der verwendete Zähler schrittweise bis auf den Wert null heruntergezählt. Dann erfolgt der Startschuss und alle darauf wartenden Threads können weiterarbeiten.

Exchanger Zum Teil besteht die Kommunikation zweier Threads nur daraus, Daten miteinander austauschen. Erst wenn beide Threads am Austauschpunkt »anwesend« sind, kann der Tausch stattfinden. Jeder Thread bietet jeweils ein Element an und nutzt dazu die Klasse `Exchanger<V>` und die Methode `exchange()`. Haben beide Threads ihre Elemente angeboten, so erhalten sie jeweils das Gegenstück des anderen Threads als Tauschobjekt. Erscheint nur ein Thread am Austauschpunkt, wartet dieser, bis der andere Thread dort auftaucht und `exchange()` aufruft. Um ein endloses Warten zu verhindern, kann eine maximale Wartezeit spezifiziert werden.

7.3.4 Unerwartete `IllegalMonitorStateExceptions`

Beim Einsatz von Multithreading und der Kommunikation von Threads treten zum Teil unerwartet `java.lang.IllegalMonitorStateExceptions` auf. Viele Entwickler sind dann ratlos.

Eine solche Exception wird dadurch ausgelöst, dass der Thread, der die Methoden `wait()`, `notify()` bzw. `notifyAll()` aufruft, während seiner Ausführung nicht den Lock des zugehörigen Objekts besitzt. Diese Tatsache kann zur Kompilierzeit nicht geprüft werden und führt erst zur Laufzeit zu der genannten Exception.

Ein kurzes Beispiel hilft dabei, einige Erkenntnisse zu gewinnen. In folgendem Listing wird über das Objekt `lock` synchronisiert und auch der Aufruf von `notify()` geschieht augenscheinlich auf diesem Objekt:

```
static Integer lock = new Integer(1);

public static void main(String[] args)
{
    synchronized (lock)
    {
        System.out.println("lock is " + lock);
        lock++;
        System.out.println("lock is " + lock);
        lock.notifyAll();
    }
    System.out.println("notWorking");
}
```

Listing 7.12 Ausführbar als `'ILLEGALMONITORSTATEEXAMPLE'`

Beim Ausführen des Ant-Targets `ILLEGALMONITORSTATEEXAMPLE` kommt es zu einer `IllegalMonitorStateException` und man erhält in etwa folgende Ausgabe auf der Konsole:

```
lock is 1
lock is 2
Exception in thread "main" java.lang.IllegalMonitorStateException
    at java.lang.Object.notifyAll(Native Method)
    at multithreading.IllegalMonitorStateExample.main(IllegalMonitorStateExample
        .java:14)
```

Tritt eine solche `IllegalMonitorStateException` auf, kann man folgendermaßen vorgehen, um die Ursache zu finden:

1. Prüfe, ob die Aufrufe an `wait()`, `notify()` bzw. `notifyAll()` innerhalb eines `synchronized`-Blocks ausgeführt werden.
 - (a) Ist dies der Fall, weiter mit Punkt 2.
 - (b) Erfolgt ein Aufruf ohne `synchronized`, so muss die Aufrufhierarchie der Methode verfolgt werden. Falls dort ein `synchronized` gefunden wird, geht es weiter mit Punkt 2. Ansonsten ist an geeigneter Stelle eine Synchronisierung einzufügen.
2. Prüfe, ob das korrekte Objekt zum Synchronisieren verwendet wird.

Für das Beispiel ist der erste Punkt offensichtlich gegeben. Anschließend prüft man, ob auch das korrekte Synchronisationsobjekt verwendet wird. Auf den ersten Blick scheint dies ebenfalls gegeben zu sein. Doch der Operator `'++'` und das Auto-Boxing führen hier dazu, dass wir nicht auf derselben Instanz des `Integer`-Objekts `lock` synchronisieren, auf der wir auch `notifyAll()` aufrufen. Somit kommt es zur `IllegalMonitorStateException`.

Tipp: Regeln zu Synchronisationsobjekten

Beim Einsatz von Synchronisationsobjekten helfen folgende Hinweise:

1. Verwende als Synchronisationsobjekt möglichst die Basisklasse `Object`.
2. Synchronisiere immer auf unveränderlichen Referenzen.
3. Vermeide den Einsatz von Objekten vom Typ `Integer`, `Long` und `String` als Synchronisationsobjekte. Das verhindert Effekte durch Auto-Boxing oder String-manipulationen.

Lock-Klassen und IllegalMonitorStateExceptions

Beim Einsatz der bereits vorgestellten `Lock`-Klassen kann es gleichermaßen zu `IllegalMonitorStateExceptions` kommen. Man muss dann sicherstellen, dass `await()`, `signal()` und `signalAll()` innerhalb eines durch einen `Lock` geschützten Bereichs aufgerufen werden. Zudem muss man dann prüfen, ob das korrekte `Condition`-Objekt des `Lock`-Objekts verwendet wird.

7.4 Das Java-Memory-Modell

Bis hierher haben wir bereits einige Fallstricke beim Zugriff auf gemeinsam verwendete Datenstrukturen beim Einsatz von Multithreading kennengelernt. Das Verständnis des Java-Memory-Modells (JMM) hilft, verlässlichere Multithreading-Applikationen

zu schreiben. Es beschreibt, wie Programme, im Speziellen Threads, Daten in den Hauptspeicher schreiben und wieder daraus lesen. Im Folgenden gehe ich auf die wichtigsten Punkte ein, die Kapitel 17 der JLS [28] im Detail beschreibt. Ich versuche, diese hier möglichst anschaulich und weniger mathematisch bzw. theoretisch als in der JLS darzustellen.

Das JMM regelt die Ausführungsreihenfolge und Unterbrechbarkeit von Operationen sowie den Zugriff auf den Speicher und bestimmt damit die Sichtbarkeit von Wertänderungen gemeinsamer Variablen verschiedener Threads. Dabei müssen drei Dinge beachtet werden:

1. **Sichtbarkeit** – Variablen können in Thread-lokalen Caches zwischengespeichert werden, wodurch ihre aktuellen Werte für andere Threads nicht sichtbar sind. Diese Speicherung erfolgt jedoch nicht immer für alle Variablen.
2. **Atomarität** – Lese- und Schreibzugriffe auf Variablen werden für die 64-Bit-Datentypen `long` und `double` *nicht atomar* in einem »Rutsch« ausgeführt, sondern durch Abarbeitung mehrerer Bytecode-Anweisungen, die daher unterbrochen werden können.
3. **Reorderings** – Befehle werden gegebenenfalls in einer von der statischen Reihenfolge im Sourcecode abweichenden Reihenfolge ausgeführt. Dies ist immer dann der Fall, wenn der Compiler einige sogenannte Peephole-Optimierungen durchgeführt hat. Details dazu beschreibt Abschnitt ??.

7.4.1 Sichtbarkeit

Konzeptionell laufen alle Threads einer JVM parallel und greifen dabei auf gemeinsam verwendete Variablen im Hauptspeicher zu. Aufgrund der Zwischenspeicherung von Werten in einem eigenen Cache eines Threads ist nicht in jedem Fall eine konsistente Sicht aller Threads auf diese Variablen gegeben. Findet kein Abgleich der gecachten Daten mit dem Hauptspeicher statt, so sind dadurch Änderungen eines Threads an einer Variablen für andere Threads nicht sichtbar.⁶

Das JMM garantiert einen solchen Abgleich von Daten mit dem Hauptspeicher lediglich zu folgenden Zeitpunkten:

- **Thread-Start** – Beim Start eines Threads erfolgt das initiale Einlesen der verwendeten Variablen aus dem Hauptspeicher, d. h., der lokale Cache eines Threads wird mit deren Werten belegt.
- **Thread-Ende** – Erst beim Ende der Ausführung eines Threads erfolgt in jedem Fall ein Abgleich des Cache mit dem Hauptspeicher. Dieser Vorgang ist zwingend notwendig, damit modifizierte Daten für andere Threads sichtbar werden.

⁶Dieser Abgleich ist auch unter dem Begriff *Cache-Kohärenz* bekannt.

Während der Abarbeitung eines Threads erfolgen in der Regel keine Hauptspeicherezugriffe mehr, sondern es wird mit den Daten des Cache gearbeitet. Der Thread »lebt isoliert« in seiner eigenen Welt. Ein expliziter Abgleich von Daten kann über die Definition eines kritischen Bereichs mit `synchronized` sowie über das Schlüsselwort `volatile` durchgeführt werden. Die Verwendung `volatile` garantiert, dass sowohl der lesende als auch der schreibende Datenzugriff direkt auf dem Hauptspeicher erfolgen. Dadurch ist garantiert, dass ein `volatile`-Attribut nie einen veralteten Wert aufweist. Beim Einsatz muss man allerdings beachten, dass die im folgenden beschriebene Atomarität sichergestellt wird.

7.4.2 Atomarität

Wie bereits erwähnt, erfolgt ein Zugriff auf Variablen der 64-Bit-Datentypen `long` und `double` nicht atomar. Für Multithreading ist dies insbesondere für nicht gecachte Variablen zu beachten, da folgendes Szenario entstehen kann: Nachdem die ersten 32 Bit zugewiesen wurden, kann ein anderer Thread aktiviert werden und »sieht« dann möglicherweise einen ungültigen Zwischenzustand der Variablen, hier am Beispiel der Zuweisung eines Werts an die Variable `val` und den Zeitpunkten `t1` und `t2` illustriert:

```
long val = 0x6789ABCD;
t1: long val = 0x0000ABCD;
t2: long val = 0x6789ABCD;
```

Will man den Zugriff atomar ausführen, nutzt man das Schlüsselwort `volatile`. Damit ist zudem die Sichtbarkeit in anderen Threads gegeben. Man könnte daher auf die Idee kommen, nur noch `volatile`-Attribute zum Datenaustausch bei Multithreading einzusetzen. Zur Vermeidung von Synchronisation könnte man beispielsweise die `increment()`-Methode eines Zählers wie folgt schreiben:

```
private volatile long counter = 0;

public void increment()
{
    // Achtung: ++ ist nicht atomar
    counter++;
}
```

Warum ist das weder sinnvoll noch ausreichend? Die Antwort ist überraschend einfach: Über `volatile` wird *kein* kritischer Abschnitt definiert, sondern lediglich *ein* Schreib- oder Lesevorgang atomar ausgeführt. Bereits das Post-Increment `counter++` besteht tatsächlich aus folgenden Einzelschritten (vgl. Abschnitt 3.1.2):

```
final long temp = counter;
counter = counter + 1;
```

Somit ist es möglich, dass andere Threads zu einem beliebigen Zeitpunkt der Ausführung der obigen Anweisungsfolge aktiviert werden. Dadurch ist keine Atomarität und

keine Thread-Sicherheit mehr gegeben. Zum Schutz vor Race Conditions und den im nächsten Abschnitt detailliert vorgestellten möglichen Folgen von Reorderings muss daher jede Folge von Anweisungen, die exklusiv durch einen Thread ausgeführt werden soll, als kritischer Abschnitt geschützt werden.

Atomare Variablen als Lösung?

Mit JDK 5 wurden verschiedene Klassen eingeführt, die atomare Read-Modify-Write-Sequenzen, wie das obige `counter++`, ermöglichen. Unter anderem sind dies die Klassen `AtomicInteger` und `AtomicLong`, die von der Klasse `Number` abgeleitet sind.⁷ Diese nutzen eine sogenannte *Compare-and-Swap-Operation* (kurz CAS). Die Besonderheit ist, dass diese zunächst einen Wert aus dem Speicher lesen und mit einem erwarteten Wert vergleichen, bevor sie eine Zuweisung mit einem übergebenen Wert durchführen. Dies geschieht allerdings nur, wenn ein erwarteter Wert gespeichert ist. Übertragen auf das vorherige Beispiel des Zählers würde dies unter Verwendung der Klasse `AtomicLong` wie folgt aussehen:

```
private AtomicLong counter = new AtomicLong();

public long incrementAndGetUsingCAS()
{
    long oldValue = counter.get();
    // Unterbrechung möglich
    while (!counter.compareAndSet(oldValue, oldValue + 1))
    {
        // Unterbrechung möglich
        oldValue = counter.get();
        // Unterbrechung möglich
    }
    return oldValue + 1;
}
```

Wie man sieht, ist selbst das einfache Hochzählen eines Werts deutlich komplizierter als ein Einsatz von `synchronized`. Das liegt vor allem daran, dass in einer Schleife geprüft werden muss, ob das Setzen des neuen Werts korrekt erfolgt ist. Diese Komplexität entsteht dadurch, dass zwischen dem Lesen des Werts und dem Setzen ein anderer Thread den Wert verändert haben könnte. In einem solchen Fall wird die Schleife so lange wiederholt, bis eine Inkrementierung erfolgreich ist.

Auf einer solch niedrigen Abstraktionsebene möchte man in der Regel nicht arbeiten. Zur Vereinfachung werden die Details durch verschiedenste Methoden der atomaren Klassen gekapselt: Ein sicheres Inkrementieren erfolgt beispielsweise mit der Methode `incrementAndGet()`. Weitere Details zu den atomaren Klassen finden Sie in den Büchern »Java Concurrency in Practice« von Brian Goetz [26] und »Java Threads« von Scott Oaks und Henry Wong [55].

Wie bei `volatile` gilt, dass sobald mehrere Attribute konsistent zueinander geändert werden sollen, eine Definition eines kritischen Abschnitts über `synchronized` oder Locks zwingend notwendig wird.

⁷Für Gleitkommatypen gibt es keine derartigen Klassen.

7.4.3 Reorderings

Die JVM darf gemäß der JLS zur Optimierung beliebige Anweisungen in ihrer Ausführungsreihenfolge umordnen, wenn dabei die Semantik des Programms nicht verändert wird. Bei der Entwicklung von Singlethreading-Anwendungen muss man den sogenannten *Reorderings* keine Beachtung schenken. Für Multithreading gibt es jedoch einige Dinge zu berücksichtigen.

Betrachten wir zur Verdeutlichung eine Klasse `ReorderingExample` mit den Attributen `x1` und `x2` und folgenden Anweisungen:

```
public class ReorderingExample
{
    int x1 = 0;
    int x2 = 0;

    void method()
    {
        // Thread 1
        x1 = 1; // #1
        x2 = 2; // #2
        System.out.println("x1 = " + x1 + " / x2 = " + x2 ); // #3
    }
}
```

Werden die obigen Anweisungen ausgeführt, so kann der Compiler die Anweisungen 1 und 2 der Methode `method()` vertauschen, ohne dass dies Auswirkung auf die nachfolgende Anweisung 3, hier die Ausgabe, hat. Anweisung 3 kann jedoch nicht mit Anweisung 1 oder 2 getauscht werden, da die Ausgabe lesend auf die zuvor geschriebenen Variablen zugreift. Reorderings sind nur dann erlaubt, wenn sichergestellt werden kann, dass es keine Änderungen an den Wertzuweisungen der sequenziellen Abarbeitung der Befehle gibt. Es kommt also immer zu der Ausgabe von `'x1 = 1 / x2 = 2'`.

Für Multithreading wird die Situation komplizierter: Würde parallel zu den obigen Anweisungen in einem separaten Thread etwa folgende Methode `otherMethod()` dieser Klasse abgearbeitet, so kann es zu unerwarteten Ergebnissen kommen.

```
void otherMethod()
{
    // Thread 2
    int y1 = x2; // #1
    int y2 = x1; // #2
    System.out.println("x1 = " + x1 + " / x2 = " + x2 ); // #3
    System.out.println("y1 = " + y1 + " / y2 = " + y2 ); // #4
}
```

Betrachten wir zunächst den einfachen Fall ohne Reorderings. Wird Thread 1 vor Thread 2 ausgeführt, also `method()` komplett vor `otherMethod()` abgearbeitet, dann gilt offensichtlich $x1 = 1 = y2$ und $x2 = 2 = y1$. Dies ergibt sich daraus, dass alle Schreiboperationen in Thread 1 bereits ausgeführt wurden, bevor die Leseoperationen in Thread 2 durchgeführt werden. Werden die beiden Threads allerdings abwechselnd, beliebig ineinander verwoben, ausgeführt, so kann es zu merkwürdigen Ausgaben kommen. Denkbar ist etwa eine Situation, in der zwar wie vermutet $x1 = 1$ und $x2 = 2$

gilt, aber auch $y1 = 0$ und $y2 = 1$. Dies ist der Fall, wenn zunächst die Ausführung von Thread 1 (`method()`) begonnen und nach der Zuweisung $x1 = 1$ unterbrochen wird und anschließend Thread 2 ausgeführt wird, wie dies im Folgenden beispielhaft dargestellt ist:

```
int x1 = 0;
int x2 = 0;

// Thread 1
x1 = 1;

x2 = 2;
// System.out: x1 = 1 / x2 = 2

// Thread 2
y1 = x2 = 0;
y2 = x1 = 1;
// System.out: x1 = 1 / x2 = 0
// System.out: y1 = 0 / y2 = 1
```

Wie man leicht sieht, kann es bereits beim abwechselnden Ausführen (*Interleaving*) von Threads zu Inkonsistenzen kommen. War obige Ausgabe noch mit etwas Nachdenken intuitiv verständlich, ist jedoch auch folgende, unerwartete Ausführungsreihenfolge möglich, wenn die Anweisungen umgeordnet werden:

```
int x1 = 0;
int x2 = 0;

// Thread 1
x2 = 2;

x1 = 1;
// System.out: x1 = 1 / x2 = 2

// Thread 2
y1 = x2 = 0;

y2 = x1 = 0;
// System.out: x1 = 0 / x2 = 2
// System.out: y1 = 0 / y2 = 0
```

Man erkennt, dass einige Besonderheiten beim Zusammenspiel von Threads und gemeinsamen Variablen zu beachten sind: Es kann zu Inkonsistenzen durch Reorderings und Interleaving kommen. Für Singlethreading analysiert und ordnet die JVM die Folge von Schreib- und Lesezugriffen automatisch, sodass diese Probleme gar nicht erst entstehen. Bei Multithreading ist die zeitliche Reihenfolge der Abarbeitung nicht im Vorhinein bekannt, sodass keine definierte Reihenfolge bzgl. des Schreibens und Lesens durch verschiedene Threads gegeben ist und die JVM keine Konsistenz sicherstellen kann. **Die Folge ist, dass ein Programm zufällige Resultate liefert (häufig sogar korrekte) und im schlimmsten Fall unbrauchbar wird.** Bei Multithreading ist es daher Aufgabe des Entwicklers, Hinweise zu geben, in welcher Reihenfolge die Abarbeitungen erfolgen sollen bzw. welche Bereiche kritisch sind und zu welchen Zeitpunkten keine Reorderings stattfinden dürfen.

Für Multithreading ist dazu eine spezielle Ordnung *Happens-before* (*hb*) definiert, die für zwei Anweisungen *A* und *B* beliebiger Threads besagt, dass für *B* alle Änderungen von Variablen einer Anweisung *A* sichtbar sind, wenn $hb(A, B)$ gilt. Durch

diese Ordnung wird indirekt festgelegt, in welchen Rahmen Reorderings stattfinden dürfen, da durch $hb(A, B)$ «Abstimmungspunkte» oder «Synchronisationspunkte» von Multithreading-Applikationen definiert werden. Zwischen diesen Abstimmungspunkten sind Reorderings allerdings beliebig möglich. An solchen Abstimmungspunkten im Programm weiß man dann jedoch sicher, dass alle Änderungen stattgefunden haben und diese für andere Threads sichtbar sind. Beispielsweise kann über Synchronisation eine gewisse Abarbeitungsreihenfolge erreicht werden: Zwei über dasselbe Lock synchronisierte kritische Abschnitte schließen sich gegenseitig aus. Laut $hb(A, B)$ gilt, dass nachdem einer von beiden abgearbeitet wurde, alle Modifikationen für den nachfolgenden sichtbar sind. Über die Reihenfolge der Ausführung innerhalb eines `synchronized`-Blocks kann allerdings keine Aussage getroffen werden.

Besteht diese Happens-before-Ordnung zwischen zwei Anweisungen jedoch nicht, so darf die JVM den Ablauf von Befehlen beliebig umordnen. In Multithreading-Programmen muss man daher als Programmierer dafür sorgen, eine Happens-before-Ordnung herzustellen, um mögliche Fehler durch Reorderings zu verhindern. Auch ohne sämtliche Details dieser Ordnung zu verstehen, kann man Thread-sichere Programme schreiben, wenn man sich an folgende Grundregeln hält:

1. **Korrekte, minimale, aber vollständige Synchronisierung** – Greifen mehrere Threads auf ein gemeinsam benutztes Attribut zu, so müssen immer *alle* Zugriffe über *dasselbe* Lock-Objekt synchronisiert werden. Für semantische Einheiten von Attributen kann auch ein und dasselbe Lock-Objekt verwendet werden.
2. **Beachtung von Atomarität** – Zuweisungen erfolgen in der Regel atomar. Für die 64-Bit-Datentypen muss dies explizit über das Schlüsselwort `volatile` sichergestellt werden. Mehrschrittoperationen (beispielsweise `++`) können so nicht geschützt werden und müssen zwingend synchronisiert werden.

Hintergrundinformationen zur Ordnung hb

Zum besseren Verständnis des Ablaufs bei Multithreading ist es hilfreich zu wissen, für welche Anweisungen eine Happens-before-Ordnung gilt. Dies sind unter anderem:

- Zwei Anweisungen A und B stehen in $hb(A, B)$, wenn B nach A im selben Thread in der sogenannten Program Order steht. Vereinfacht bedeutet dies, dass jeder Schreibzugriff für folgende Lesezugriffe auf ein Attribut sichtbar ist. Erfolgen Lese- und Schreibzugriffe nicht auf gleiche Attribute, stehen Anweisungen also nicht in einer Lese-Schreib-Beziehung, dürfen diese vom Compiler beliebig umgeordnet werden. Bei Singlethreading macht sich dies nicht bemerkbar, da trotz Reorderings die Bedeutung nicht verändert wird.
- Es gilt $hb(A, B)$, wenn zwei Anweisung A und B über denselben Lock synchronisiert sind.
- Für einen Schreibzugriff A auf ein `volatile`-Attribut und einen späteren Lesezugriff B gilt $hb(A, B)$.

- Für den Aufruf von `start()` (A) eines Threads und alle Anweisungen B , die in diesem Thread ausgeführt werden, gilt $hb(A, B)$.
- Für alle Anweisungen A eines Threads vor dessen Ende B ist $hb(A, B)$ erfüllt, d. h., alle nachfolgenden Threads können diese Änderungen sehen – allerdings nur, wenn, diese über den gleichen Lock synchronisiert sind oder `volatile`-Attribute verwenden.

Tipp: Eigenschaften der Ordnung hb

Die Ordnung hb besitzt für beliebige Anweisungen A und B folgende Eigenschaften:

- **Irreflexiv** – $!hb(A, A)$ – Keine Anweisung A »sieht« ihre eigenen Änderungen.
- **Transitiv** – Wenn $hb(A, B)$ und $hb(B, C) \Rightarrow hb(A, C)$ – Die Transitivität stellt die Sichtbarkeit gemäß der Ablaufreihenfolge der Anweisungen im Sourcecode sicher.
- **Antisymmetrisch** – Wenn $hb(A, B)$ und $hb(B, A) \Rightarrow A = B$ oder für diesen Fall intuitiver: Wenn $hb(A, B)$ und $A \neq B \Rightarrow !hb(B, A)$ – Die Antisymmetrie stellt sicher, dass eine Anweisung A nicht die Modifikationen einer Nachfolganweisung B »sieht«.

7.5 Besonderheiten bei Threads

Nachdem wir ausführlich den Lebenszyklus und die Zusammenarbeit von Threads kennengelernt haben, geht dieser Abschnitt auf Besonderheiten bei Threads ein. Im Folgenden stelle ich zunächst verschiedene Arten von Threads vor. Anschließend betrachten wir die Auswirkungen von Exceptions in Threads. Danach greife ich das Thema »Beenden von Threads« erneut auf, das mit dem bisher gesammelten Wissen mit all seinen Tücken nachvollzogen werden kann. Abschließend stelle ich Möglichkeiten zur zeitgesteuerten Ausführung von Aufgaben vor.

7.5.1 Verschiedene Arten von Threads

In Java unterscheidet man zwei Arten von Threads, sogenannte User- und Daemon-Threads.

main-Thread und User-Threads

Wie bereits erwähnt, erzeugt die JVM beim Start einen speziellen Thread, den man `main-Thread` nennt, weil dieser statt der `run()`-Methode die `main()`-Methode des Programms ausführt. Nehmen wir an, eine Applikation würde mehrere Threads aus dem `main-Thread` erzeugen und starten. Diese Threads nennt man *User-Threads*.

Die Ausführung des `main`-Threads endet zwar, wenn die letzte Anweisung der `main()`-Methode abgearbeitet wird, allerdings bleibt die JVM weiter aktiv, solange noch vom `main`-Thread abgespaltene User-Threads existieren und deren `run()`-Methode ausgeführt wird.

Daemon-Threads

Manchmal ist dieses Verhalten allerdings nicht gewünscht und eine JVM soll beendet werden können, selbst wenn noch gewisse Aufgaben im Hintergrund ablaufen. Dazu existieren sogenannte *Daemon-Threads*.

Eine Terminierung der JVM ist von solchen Threads unabhängig. Das bekannteste Beispiel für einen solchen Daemon-Thread ist der Garbage Collector. Über die Methode `setDaemon(boolean)` kann man einen beliebigen Thread vor seiner Ausführung zu einem Daemon-Thread erklären. Ein Programm bzw. die zugehörige JVM terminiert, nachdem alle normalen Threads beendet sind. Für dann noch aktive Daemon-Threads bedeutet dies, dass sie abrupt beendet werden, d. h. irgendwo in der Abarbeitung ihrer `run()`-Methode. Von Daemon-Threads belegte Ressourcen müssen daher entweder in einer eigenen `finalize()`-Methode (vgl. Abschnitt 8.7.5) oder einem Shut-down-Hook (vgl. Abschnitt 10.1.3) freigegeben werden.

Der Einsatz von Daemon-Threads kann für Utility-Funktionalitäten sinnvoll eingesetzt werden, etwa die folgende: In Windows-Systemen können aufgrund eines Fehlers im JDK Aufrufe von `Thread.sleep(long)` mit kleinen Wartezeiten dazu führen, dass die Systemuhr beschleunigt wird. Die folgende als Daemon-Thread realisierte Klasse `SleepBugSolver` löst dieses Problem auf die von Sun vorgeschlagene Weise, indem in der `run()`-Methode in einer Endlosschleife hintereinander `sleep(long)` mit `Integer.MAX_VALUE` aufgerufen wird:

```
public final class SleepBugSolver extends Thread
{
    public SleepBugSolver()
    {
        setDaemon(true);
    }

    public void run()
    {
        while (true)
        {
            try
            {
                Thread.sleep(Integer.MAX_VALUE);
            }
            catch (final InterruptedException ex)
            {
                // Spezialfall: hier Exception bewusst ignorieren
            }
        }
    }
}
```

In diesem speziellen Fall leiten wir von der Utility-Klasse `Thread` ab, um die Daemon-Eigenschaft sicherzustellen. Würde die Klasse lediglich auf einem `Runnable` basieren, so müsste das Setzen der Daemon-Eigenschaft durch eine aufrufende Applikation geschehen. Zudem müsste man die Eigenschaft immer vor dem Start und damit vor dem Aufruf der `run()`-Methode setzen. Diesen Aufwand wollen wir vermeiden.

7.5.2 Exceptions in Threads

Nehmen wir an, eine Applikation würde mehrere Threads aus dem `main`-Thread starten. Tritt eine Exception in einem (Sub-)Thread auf, so wird diese bis zur `run()`- bzw. `main()`-Methode propagiert, wenn sie nicht explizit abgefangen wird. Eine vom Programm unbehandelte Exception wird über den Error-Stream `System.err` inklusive des kompletten Stacktrace ausgegeben und führt anschließend zu einer Beendigung des ausführenden Threads. Zur Demonstration des zuvor beschriebenen Verhaltens können Sie folgendes Ant-Target `EXCEPTIONINTHREADSEXAMPLE` ausführen:

```
// Achtung: Nur zur Demonstration des Exception Handlings
public static void main(final String[] args) throws InterruptedException
{
    exceptionalMethod();
}

static void exceptionalMethod() throws InterruptedException
{
    final Thread exceptional = new Thread()
    {
        public void run()
        {
            throw new IllegalStateException("run() failed");
        }
    };

    exceptional.start();
    Thread.sleep(1000);
    throw new IllegalStateException("exceptionalMethod() failed");
}
```

Listing 7.13 Ausführbar als 'EXCEPTIONINTHREADSEXAMPLE'

Ungefähr folgende Ausgaben (gekürzt) erscheinen auf der Konsole:

```
Exception in thread "Thread-0" java.lang.IllegalStateException: run() failed
    at multithreading.ExceptionInThreadsExample$1.run(ExceptionInThreadsExample.
        java:18)
```

Wird das abrupte Ende eines Threads nur auf die Konsole geschrieben, wird eine mögliche Fehlersuche sehr erschwert. Um eine Fehlersituation später nachvollziehen zu können, hilft es, Exceptions in eine Log-Datei zu schreiben.

Ab Java 5 existiert das innere Interface `UncaughtExceptionHandler` in der Klasse `Thread`. Realisierungen davon erlauben, ansonsten unbehandelte Exceptions zu behandeln. In einer Implementierung dieses Interface kann dann beispielsweise eine Ausgabe in eine Log-Datei erfolgen. Die folgende Klasse `LoggingUncaught-`

ExceptionHandler realisiert genau diese Funktionalität. Im Listing ist auch eine `main()`-Methode gezeigt, die zu Demonstrationszwecken Exceptions provoziert:

```
public final class LoggingUncaughtExceptionHandler implements Thread.
    UncaughtExceptionHandler
{
    private static final Logger log = Logger.getLogger(
        LoggingUncaughtExceptionHandler.class);

    private final Logger    loggerToUse;

    LoggingUncaughtExceptionHandler(final Logger loggerToUse)
    {
        this.loggerToUse = loggerToUse;
    }

    public void uncaughtException(final Thread thread, final Throwable throwable
    )
    {
        log.error("Unexpected exception occurred: ", throwable);
    }

    public static void main(final String[] args) throws InterruptedException
    {
        // Ausgabe auf Logger umleiten
        Thread.setDefaultUncaughtExceptionHandler(new
            LoggingUncaughtExceptionHandler(log));

        ExceptionInThreadsExample.exceptionMethod();
        throw new IllegalStateException("execute main() failed");
    }
}
```

Listing 7.14 Ausführbar als 'LOGGINGUNCAUGHTEXCEPTIONHANDLER'

Ein solcher `UncaughtExceptionHandler` kann, wie im obigen Listing, für alle Threads global wie folgt gesetzt werden:

```
Thread.setDefaultUncaughtExceptionHandler(new LoggingUncaughtExceptionHandler(
    loggerToUse));
```

Alternativ kann dies bei Bedarf für jeden Thread einzeln durch Aufruf der Methode `setUncaughtExceptionHandler(UncaughtExceptionHandler)` erfolgen.

Der Einsatz eines Exception Handlers empfiehlt sich in eigenen Applikationen, um im Fehlerfall Ressourcen freizugeben und Probleme zu protokollieren, was eine spätere Analyse erleichtert. In diesem Beispiel wird ein mögliches Problem zwar nicht weiter behandelt, aber zumindest in einer Log-Datei protokolliert.

7.5.3 Sicheres Beenden von Threads

Wie bereits erwähnt, lassen sich Threads leider nicht so einfach beenden wie starten. Die Methode `stop()` der Klasse `Thread` ist als `@deprecated` markiert. In verschiedenen Quellen wird als Grund für die Unzulänglichkeit genannt, dass beim Beenden eines Threads nicht alle Locks freigegeben werden. Das ist ein verbreiteter Irrtum. Definitiv werden beim Auftreten von Exceptions alle gehaltenen Locks durch die JVM

zurückgegeben. Der Grund für die Markierung als `@deprecated` ist vielmehr folgender: Wenn ein Thread durch Aufruf dieser Methode beendet wird, können dadurch die Daten, auf denen der Thread gerade gearbeitet hat, in einen inkonsistenten Zustand gebracht werden. Insbesondere gilt dies, wenn der Thread innerhalb einer eigentlich atomaren Anweisungsfolge unterbrochen wird: Erfolgt ein Aufruf von `stop()` mitten innerhalb eines über `synchronized` definierten kritischen Bereichs, wird dieser irgendwo unterbrochen und nicht mehr atomar ausgeführt. Eine mögliche Inkonsistenz im Objektzustand ist die Folge.

Um Konsistenz zu wahren, müssen wir andere Wege finden, einen Thread korrekt zu beenden. Zwei Möglichkeiten, dies sauber zu lösen, sind die folgenden:

1. Einführen einer Hilfsklasse
2. Beenden durch Aufruf der Methode `interrupt()`

Hilfsklasse zum Beenden

Eine mögliche Lösung zum Beenden von Threads besteht darin, eine Hilfsklasse zu implementieren und dort periodisch ein Flag `shouldStop` abzufragen. Hierfür sieht man in der Praxis und in manchem Buch folgende denkbare, aber falsche Umsetzung:⁸

```
// Achtung: Nicht Thread-sicher
public class BaseStoppableThread extends Thread
{
    private boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public void shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // Kein Aufruf von requestStop() und
            // keine Schreibzugriffe auf shouldStop
        }
    }
}
```

Auf den ersten Blick ist – abgesehen von der ungeschickten Ableitung von der Utility-Klasse `Thread` (vgl. folgenden Hinweis »Ableitung von `Thread`«) – kein Fehler zu

⁸In der Praxis wird das Flag häufig `stopped` genannt. Zudem heißen die Zugriffsmethoden auf das Flag etwa `setStopped(boolean)` und `isStopped()`. Dies entspricht zwar der Intention des Beendens, allerdings wird hier eher ein Stoppwunsch geäußert. Daher werden die Methoden in diesem Beispiel bewusst `requestStop()` und `shouldStop()` genannt.

erkennen. Wieso ist diese Umsetzung trotzdem problematisch? Nach Lektüre von Abschnitt 7.4 über das Java-Memory-Modell sind wir bereits etwas sensibilisiert: Die JVM darf zur Optimierung Reorderings durchführen, sofern die Happens-before-Ordnung eingehalten wird. Ohne diese Ordnung beachtet der Compiler bei der Optimierung keine Multithreading-Aspekte: Er kann beispielsweise wiederholte Lesezugriffe auf sich nicht ändernde Variablen zu vermeiden versuchen, indem er eine `if`-Abfrage einfügt und anschließend die `while`-Schleife mit konstantem Wert erzeugt. Weitere Details zu derartigen Peephole-Optimierungen liefert Kapitel 16.

In dem Beispiel finden wir keinen Aufruf von `requestStop()` innerhalb der `run()`-Methode: Für Singlethreading ergibt sich daraus, dass sich das Attribut `shouldStop` in der Schleife nicht mehr ändert. Damit ist das Ergebnis der Bedingung konstant. Um den Methodenaufruf und die wiederholte Auswertung der Bedingung `!shouldStop()` einzusparen, kann der Sourcecode durch den Compiler und die JVM wie folgt optimiert und umgeordnet werden:

```
public void run()
{
    if (!shouldStop())
    {
        while (true)
        {
            // ...
        }
    }
}
```

Man erkennt die Komplexität der Fallstricke beim Einsatz von Multithreading und Reorderings gerade beim Beenden von Threads besonders gut. Zum korrekten Ablauf dieses Beispiels bei Multithreading ist daher durch den Entwickler die Happens-before-Ordnung sicherzustellen. Dies kann in diesem Fall entweder über die Deklaration des Attributs `shouldStop` als `volatile` oder einen synchronisierten Zugriff geschehen. Wir vermeiden zudem die Ableitung von der Klasse `Thread` und implementieren folgende Lösung basierend auf dem Interface `Runnable`:

```
abstract class AbstractStoppableRunnable implements Runnable
{
    private volatile boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public boolean shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // ...
        }
    }
}
```

Die Happens-before-Ordnung ist nun sichergestellt und die JVM darf als Folge keine Reorderings und keine Optimierung der Schleifenabfrage durchführen.

Hinweis: Ableitung von Thread

Ableitungen von Utility-Klassen sind in der Regel ungünstig. **Leider sieht man aber genau dies häufig beim Einsatz der Klasse `Thread`**. Schauen wir es uns an, wie so es zu Problemen kommen kann. Nehmen wir dazu an, eine Klasse `AbstractStoppableThread` sei durch Ableitung realisiert und die restliche Realisierung wäre analog zu der zuvor vorgestellten Klasse `AbstractStoppableRunnable`. Ersteres zeigt folgende Zeile:

```
public abstract class AbstractStoppableThread extends Thread
```

Als Folge käme es zu einem unerwarteten Effekt durch diesen Einsatz von Vererbung. Es wird das Substitutionsprinzip und die »is-a«-Eigenschaft (vgl. Kapitel 3) verletzt. Mit dieser von `Thread` abgeleiteten Klasse `AbstractStoppableThread` können keine Implementierungen von `Runnable` ausgeführt werden, da die `run()`-Methode der Basisklasse `Thread` durch eine eigene Implementierung überschrieben wird, die dies nicht erlaubt. Die Umsetzung ist OO-technisch unsauber, da sich die eigene Klasse nicht so verwenden lässt wie die Klasse `Thread`.

Beenden mit `interrupt()`

Statt einen Thread durch die Verwendung eigener Mechanismen in einer abgeleiteten Klasse, basierend auf `Thread` bzw. `Runnable` und dem Einsatz eines Stop-Flags, zu beenden, kann man über die Methode `interrupt()` der Klasse `Thread` die Beendigung der Ausführung eines anderen Threads anregen. Wie bereits bekannt, ist ein Aufruf der Methode `interrupt()` jedoch nur als Aufforderung zu sehen, sie besitzt aber keine unterbrechende Wirkung: Es wird lediglich ein Flag gesetzt. Die Bearbeitung und Auswertung dieses Flags mithilfe der Methode `isInterrupted()` ist Aufgabe des Entwicklers der `run()`-Methode:

```
public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        // ...
    }
}
```

Achtung: `Thread.interrupted()` vs. `isInterrupted()`

Bei der Abfrage des Flags muss man etwas Vorsicht walten lassen. Die Klasse `Thread` bietet die statische Methode `interrupted()`, die zwar das Flag prüft, dieses allerdings auch zurücksetzt. In der Regel soll eine Prüfung ohne Seiteneffekt erfolgen. Dazu ist immer die Objektmethode `isInterrupted()` zu verwenden.

Fazit

Beide gezeigten Lösungen zum Beenden von Threads sind funktional nahezu gleichwertig. Damit länger andauernde Aktionen in der `run()`-Methode tatsächlich abgebrochen werden können, müssen dort gegebenenfalls weitere Abfragen und Aufrufe von `shouldStop()` bzw. `isInterrupted()` erfolgen, um auch zwischen den ausgeführten Arbeitsschritten eine Reaktion auf Stoppwünsche zu ermöglichen.

7.5.4 Zeitgesteuerte Ausführung

Möchte man gewisse Aufgaben zu einem speziellen Zeitpunkt oder periodisch ausführen, ist dies mit `Thread`-Objekten und deren `sleep(long)`-Methode zwar möglich, aber umständlich zu realisieren.

Eine einmalige Ausführung zu einem Zeitpunkt in der Zukunft kann durch den Aufruf von `Thread.sleep(long)` vor der eigentlichen Funktionalität realisiert werden. Dazu berechnet man die Differenz von dem aktuellen Zeitpunkt bis zu dem gewünschten Zeitpunkt der Ausführung und wartet vor Ausführung der Aufgabe die zuvor berechnete Zeitdauer. Eine periodische Ausführung erreicht man, wenn die zuvor beschriebene Logik wiederholt innerhalb einer Schleife ausgeführt wird. Kompliziert wird dies, wenn mehrere Aufgaben zeitgesteuert abgearbeitet werden sollen.

Statt dies umständlich von Hand zu programmieren, ist es sinnvoller, die Utility-Klassen `Timer` und `TimerTask` zu verwenden. Darauf gehe ich im Folgenden genauer ein.

Ausführung einer Aufgabe mit den Klassen `Timer` und `TimerTask`

Die Klasse `Timer` ist für die Verwaltung und Ausführung von `TimerTask`-Objekten zuständig, die durchzuführende Aufgaben implementieren. Man erzeugt zunächst ein `Timer`-Objekt und übergibt diesem ein oder mehrere `TimerTask`-Objekte. Deren Realisierung erfolgt durch Ableitung von der abstrakten Klasse `TimerTask`, die das `Runnable`-Interface erfüllt und folgende Methoden bietet:

- `void run()` – Hier wird analog zu `Thread` und `Runnable` die Implementierung der auszuführenden Aufgabe vorgenommen.
- `boolean cancel()` – Beendet den `TimerTask`. Weitere Ausführungen werden verhindert.
- `long scheduledExecutionTime()` – Liefert den Zeitpunkt der letzten Ausführung und kann z. B. zur Kontrolle der Ausführungsdauer genutzt werden.

Zur Demonstration eines `TimerTasks` und der Verarbeitung mit einem `Timer` definieren wir folgende Klasse `SampleTimerTask`, die lediglich einen im Konstruktor übergebenen Text ausgibt:

```

public static class SampleTimerTask extends TimerTask
{
    private final String message;

    SampleTimerTask(final String message)
    {
        this.message = message;
    }

    public void run()
    {
        System.out.println(message);
    }
}

```

Einmalige Ausführung Zur Ausführung werden `TimerTask`-Objekte an einen `Timer` übergeben und dort eingeplant. Der Zeitpunkt der Ausführung kann relativ in Millisekunden bis zur Ausführung oder absolut in Form eines `Date` angegeben werden. Das folgende Beispiel zeigt die Einplanung von `SampleTimerTasks` per `schedule(TimerTask, long)` einmalig sofort und zu einem gewissen Zeitpunkt in Form der Ausgaben "OnceImmediately" sowie "OnceAfter5s". Die Ausgabe von "OnceAfter1min" zeigt die Variante von `schedule(TimerTask, Date)`, die den Ausführungszeitpunkt als `Date`-Objekt übergeben bekommt und die Ausführung nach einer Minute startet. Dieser Zeitpunkt wird über eine eigene Hilfsmethode `oneMinuteFromNow()` berechnet:

```

public static void main(String[] args)
{
    final Timer timer = new Timer();

    // Sofortige Ausführung
    final long NO_DELAY = 0;
    timer.schedule(new SampleTimerTask("OnceImmediately"), NO_DELAY);

    // Ausführung nach fünf Sekunden
    final long INITIAL_DELAY_FIVE_SEC = 5000;
    timer.schedule(new SampleTimerTask("OnceAfter5s"), INITIAL_DELAY_FIVE_SEC);

    // Ausführung nach einer Minute
    final Date ONE_MINUTE_AS_DATE = oneMinuteFromNow();
    timer.schedule(new SampleTimerTask("OnceAfter1min"), ONE_MINUTE_AS_DATE);
}

```

Listing 7.15 Ausführbar als 'TIMERTASKEXAMPLE'

Periodische Ausführung Möchte man eine Aufgabe periodisch ausführen, bietet die Klasse `Timer` hierfür zwei Varianten an: Zum einen kann dies mit festem Intervall durch einen Aufruf von `schedule(TimerTask, long, long)` und zum anderen mit festem Takt durch einen Aufruf der Methode `scheduleAtFixedRate(TimerTask, long, long)` geschehen. Optional kann eine Verzögerung bis zur ersten Ausführung

angegeben werden. Liegt der Ausführungszeitpunkt in der Vergangenheit oder wird eine Verzögerung von 0 angegeben, so startet die Ausführung sofort.

Abbildung 7-5 macht den Unterschied der beiden Varianten deutlich: Bei festem Intervall ist die Pause zwischen zwei Ausführungen immer gleich lang. Bei festem Takt ist der Startzeitpunkt zweier Ausführungen immer durch die angegebene Taktrate bestimmt.

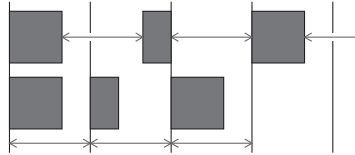


Abbildung 7-5 Ausführung eines `TimerTask` mit festem Takt und festem Intervall

Anhand der Grafik wird klar, dass *eine periodische Ausführung mit festem Takt nur dann funktionieren kann, wenn der angegebene Takt deutlich größer ist als die Dauer der Ausführung der einzelnen `TimerTasks`.*

Folgende Programmzeilen zeigen beide Varianten und sorgen dafür, dass jeweils nach 30 Sekunden die Meldung "PeriodicRefreshing" und in einem festen Takt von 10 Sekunden die Meldung "10s FixedRateExecution" ausgegeben werden.

```
public static void main(String[] args)
{
    final Timer timer = new Timer();

    // Eingeplante Ausführung mit INITIAL_DELAY und Verzögerung DELAY
    final long INITIAL_DELAY_ONE_SEC = 1000;
    final long DELAY_30_SECS = 30000;
    timer.schedule(new SampleTimerTask("PeriodicRefreshing"),
        INITIAL_DELAY_ONE_SEC, DELAY_30_SECS);

    // Eingeplante Ausführung mit INITIAL_DELAY und Takt RATE
    final long RATE_10_SECS = 10000;
    timer.scheduleAtFixedRate(new SampleTimerTask("10s FixedRateExecution"),
        INITIAL_DELAY_ONE_SEC, RATE_10_SECS);
}
```

Listing 7.16 Ausführbar als 'TIMERTASKPERIODICEXAMPLE'

Achtung: Fallstricke beim Einsatz von `Timer` und `TimerTask`

Ausführung als User-Thread Ein `Timer` besitzt zur Ausführung *genau einen* Thread. Dieser läuft in der Regel als User-Thread. Eine JVM wird jedoch nicht terminiert, solange noch mindestens ein User-Thread läuft. **Daher muss bei Programmende jeder `Timer` explizit mit der Methode `cancel()` angehalten werden, um eine Terminierung der JVM zu erlauben.** Um sich nicht um solche Details kümmern zu müssen, ist es sinnvoller, einen `Timer` und dessen Thread als Daemon-Thread wie folgt zu starten:

```
final Timer daemonTimer = new Timer("DaemonTimer", true);
```

Gegenseitige Beeinflussung von `TimerTasks` Die Genauigkeit der angegebenen Ausführungszeiten kann gewissen Schwankungen unterliegen, da alle `TimerTasks` in einem gemeinsamen Thread des `Timers` ausgeführt werden. Weiterhin beendet eine nicht gefangene oder explizit ausgelöste Exception sowohl den `Timer` als auch alle enthaltenen `TimerTasks`.

Sollen sich Ausführungszeiten oder Fehler in einzelnen `TimerTasks` nicht auf die Ausführung anderer `TimerTasks` auswirken, so lässt sich dies durch die Ausführung in eigenen `Timer`-Objekten und damit separaten Threads lösen. Es bietet sich alternativ der Einsatz sogenannter Thread-Pools an. Eine derartige Ausführung durch mehrere Threads sollte man nicht selbst realisieren, sondern die in Abschnitt 7.6.2 beschriebenen Möglichkeiten des mit JDK 5 eingeführten Executor-Frameworks nutzen.

Fazit

Wie man sieht, handelt es sich bei den bisher vorgestellten Techniken vorwiegend um (komplizierte) Konstrukte mit einem niedrigen Abstraktionsgrad, die einiges an Komplexität in eine Applikation einführen und in der Praxis eher umständlich zu nutzen sind. Mit dieser etwas schwierigeren Kost haben wir nun die elementaren Grundlagen für das Schreiben sicherer Multithreading-Anwendungen kennengelernt.

Ein Ansatz, die Probleme beim Multithreading eleganter zu lösen, bestand lange Zeit darin, die Klassenbibliothek `util.concurrent` von Doug Lea zu verwenden. Sie wurde durch den JSR 166 in Form der sogenannten »Concurrency Utilities« ins JDK 5 aufgenommen. Der folgende Abschnitt stellt diese vor.

7.6 Die Concurrency Utilities

Die bis einschließlich JDK 1.4 vorhandenen und zuvor vorgestellten Sprachmittel, wie die Schlüsselwörter `volatile` und `synchronized` sowie die Methoden `wait()`, `notify()` und `notifyAll()`, erlauben es zwar, ein Programm auf mehrere Threads aufzuteilen und zu synchronisieren, allerdings lassen sich viele Aufgabenstellungen so nur relativ umständlich realisieren.

Die mit JDK 5 eingeführten Concurrency Utilities erleichtern die Entwicklung von Multithreading-Anwendungen, da sie für viele zuvor nur mühsam zu lösende Probleme bereits fertige Bausteine bereitstellen. Durch deren Einsatz wird Komplexität aus der Anwendung in das Framework verlagert, was die Lesbarkeit und Verständlichkeit des Applikationscodes deutlich erhöht: Für Locks, Conditions und atomare Variablen haben wir bereits gesehen, dass sich Ideen und Konzepte klarer ausdrücken lassen.

Im Package `java.util.concurrent` befinden sich die Bausteine (Klassen und Interfaces) der Concurrency Utilities. Man kann folgende Kernfunktionalitäten unterscheiden:

- **Concurrent Collections** – Die Concurrent Collections enthalten auf Parallelität spezialisierte Implementierungen der Interfaces `List`, `Set`, `Map`, `Queue` und `Deque`, wodurch ein einfaches Ersetzen »normaler« Datenstrukturen durch deren Concurrent-Pendants wesentlich erleichtert wird, falls Nebenläufigkeit erforderlich ist.
- **Executor-Framework** – Das Executor-Framework unterstützt bei der Ausführung asynchroner Aufgaben durch Bereitstellung von Thread-Pools und ermöglicht es, verschiedene Abarbeitungsstrategien zu verwenden sowie die Bearbeitung abubrechen und Ergebnisse abzufragen.
- **Locks und Conditions** – Verschiedene Ausprägungen von Locks und Conditions vereinfachen die zum Teil umständliche Kommunikation und Koordination mit den Methoden `wait()`, `notify()` und `notifyAll()` sowie den Schutz kritischer Bereiche durch `synchronized`. Eine kurze Einführung in die Thematik habe ich bereits als Abschluss der Diskussion über kritische Bereiche und das Schlüsselwort `synchronized` in Abschnitt 7.2.2 gegeben.
- **Atomare Variablen** – Atomare Aktionen auf Attributen waren vor JDK 5 nur über Hilfsmittel, etwa die Schlüsselwörter `volatile` und `synchronized`, zu erreichen. Die mit JDK 5 neu eingeführten atomaren Variablen haben wir bereits bei der Vorstellung der Atomarität des JMM in Abschnitt 7.4.2 kennengelernt. Sie erlauben eine atomare Veränderung ohne Einsatz von Synchronisation.
- **Synchronizer** – Manchmal sollen Aktivitäten an speziellen Stellen in parallele Teile aufgespalten und später an Synchronisationspunkten wieder zusammengeführt werden. Lange Zeit war die Koordination mehrerer Threads lediglich auf unterster Sprachebene unter Verwendung der Methode `join()` möglich. Mit Java 5 wurden unter anderem die Klassen `CountDownLatch`, `CyclicBarrier`, `Exchanger` und `Semaphore` eingeführt, die dabei helfen, Synchronisationspunkte zu realisieren. Eine kurze Vorstellung der Arbeitsweise wurde bereits in Abschnitt 7.3 gegeben.

Bei der täglichen Arbeit sind vor allem die Concurrent Collections und das Executor-Framework wichtige Bausteine. Beide werden in den folgenden Abschnitten 7.6.1 und 7.6.2 behandelt.

7.6.1 Concurrent Collections

Wenn mehrere Threads parallel verändernd auf eine Datenstruktur zugreifen, kann es leicht zu Inkonsistenzen kommen. Dies gilt insbesondere, da die meisten Containerklassen des Collections-Frameworks nicht Thread-sicher sind. In diesem Abschnitt betrachten wir kurz typische Probleme in Multithreading-Anwendungen beim Einsatz dieser »normalen« Collections. Anschließend werden wir zur Lösung dieser Probleme die Concurrent Collections nutzen. Diese können bei Bedarf nach Parallelität stellvertretend für die »Original«-Container eingesetzt werden. Die Grundlage für diese Austauschbarkeit bilden die gemeinsamen Interfaces, beispielsweise `List`, `Set` und `Map`.

Wenn tatsächlich Nebenläufigkeit und viele parallele Zugriffe unterstützt werden müssen, können die Concurrent Collections ihre Stärken ausspielen. In Singlethreading-Umgebungen oder bei sehr wenigen konkurrierenden Zugriffen ist ihr Einsatz gut abzuwägen, da in den Containern selbst einiges an Aufwand betrieben wird, um sowohl Thread-Sicherheit als auch Parallelität zu gewährleisten.

Thread-Sicherheit und Parallelität mit »normalen« Collections

Die `synchronized`-Wrapper des Collections-Frameworks ermöglichen einen Thread-sicheren Zugriff durch Synchronisierung aller Methoden (vgl. Abschnitt 5.3.2). Allerdings erfordern Iterationen – wie wir im Verlauf dieses Abschnitts sehen werden – eine zusätzliche Synchronisierung, da es sich um sogenannte Mehrschrittoperationen handelt. Betrachten wir folgende synchronisierte Liste von `Person`-Objekten als Ausgangsbasis unserer Diskussion:

```
final List<Person> syncPersons = Collections.synchronizedList(personList);
```

Bei parallelen Zugriffen auf diese Liste können sich Threads gegenseitig blockieren und stören. Die Synchronisierung stellt einen Engpass dar und serialisiert die Zugriffe: Es kommt dadurch zu (stark) eingeschränkter Parallelität. Eine derartige Ummantelung schützt zudem nicht vor möglichen Inkonsistenzen, wenn man mehrere für sich Thread-sichere Methoden hintereinander aufruft. Eine atomare Ausführung als kritischer Bereich ist dadurch nicht garantiert. Dies habe ich bereits bei der Beschreibung der `synchronized`-Wrapper in Abschnitt 5.3.2 und bei der Darstellung eines Singletons und dessen `getInstance()`-Methode in Abschnitt 7.2.1 diskutiert.

Greifen wir hier das Thema erneut auf, um mithilfe von einfachen Beispielen die Problematik zu verdeutlichen und die Arbeitsweise und Vorteile der Concurrent Collections kennenzulernen.

Datenzugriff Jeder einzelne Methodenaufruf einer wie oben synchronisierten Collection ist für sich gesehen Thread-sicher. Für eine benutzende Komponente sind solche feingranularen Sperren aber häufig uninteressant. Vielmehr sollen Operationen mit mehreren Schritten atomar und Thread-sicher ausgeführt werden. Solche Mehrschrittoperationen sind etwa »`testAndGet()`« oder »`putIfAbsent()`«, die zunächst prüfen, ob ein gewisses Element enthalten ist, und nur dann einen Zugriff bzw. eine Modifikation ausführen:

```
// ACHTUNG: nicht Thread-sicher
public Person testAndGet(final int index)
{
    if (index < syncPersons.size())
    {
        return syncPersons.get(index);
    }
    return null;
}
```

Für Maps würde man analog Folgendes schreiben:

```
// ACHTUNG: nicht Thread-sicher
public Person putIfAbsent(final String key, final Person newPerson)
{
    if (!syncMap.containsKey(key))
    {
        return syncMap.put(key, newPerson);
    }

    return syncMap.get(key);
}
```

Auch wenn man zunächst auf die etwas naive Idee kommen könnte, die Mehrschrittoperationen für Listen und Maps wie gezeigt durch den Einsatz Thread-sicherer Einzelmethode zu realisieren, so ist dieser Ansatz doch nicht Thread-sicher: In ihrer Kombination sind Thread-sichere Methode nicht Thread-sicher, da nach jedem geschützten Methodenaufwurf ein Thread-Wechsel möglich ist. Erfolgt die Unterbrechung nach einer Leseoperation, in obigen Beispielen etwa direkt nach Ausführung der `if`-Bedingungen, und verändert ein aktivierter Thread die Datenstruktur, so kann dies verheerende Auswirkungen bei Wiederaufnahme eines unterbrochenen Threads haben: Bei nachfolgenden Zugriffen kommt es entweder zu Inkonsistenzen oder Exceptions.

Daher müssen beim Bearbeiten von Daten zusätzliche Synchronisierungsschritte ausgeführt werden. Man kann dazu ein Synchronisationsobjekt verwenden. In diesem Fall bietet sich die finale Referenz `syncPersons` an. Wenn alle Zugriffsmethoden über den Lock dieses Synchronisationsobjekts geschützt werden, kann eine korrekt synchronisierte Version der `testAndGet(int)`-Methode folgendermaßen implementiert werden:

```
public Person testAndGet(final int index)
{
    // Kritischer Bereich für Mehrschrittoperationen
    synchronized (syncPersons)
    {
        if (index < syncPersons.size())
        {
            return syncPersons.get(index);
        }
    }
    return null;
}
```

Werden jedoch nicht alle Zugriffe über denselben Lock geschützt, kann es leicht zu Inkonsistenzen oder Deadlocks kommen.

Iteration Eine weitere gebräuchliche Mehrschrittoperation ist die Iteration über eine Datenstruktur. Die Iteratoren der »normalen« Container sind »fail-fast«, d. h., sie prüfen sehr streng, ob möglicherweise während einer Iteration eine Veränderung an der Datenstruktur vorgenommen wurde, und reagieren darauf mit einer `ConcurrentModificationException`.

Zur Thread-sicheren Iteration über eine synchronisierte Liste ist dieser Vorgang zusätzlich durch einen `synchronized`-Block zu schützen:

```
// Blockiert andere Zugriffe auf syncPersons während der Iteration
synchronized (syncPersons)
{
    for (final Person person : syncPersons)
    {
        person.doSomething();
    }
}
```

Auf diese Weise ist zwar eine korrekt synchronisierte Iteration möglich, allerdings auf Kosten von Nebenläufigkeit: Durch den `synchronized`-Block wird ein kritischer Bereich definiert und *die Liste ist während des Iterierens für mögliche andere Zugriffe auf die Datenstruktur `syncPersons` blockiert. Nebenläufigkeit wird dadurch stark behindert.*

Varianten der Iteration Betrachten wir das Ganze etwas konkreter anhand der Benachrichtigung von Listnern im BEOBACHTER-Muster (vgl. Abschnitt 12.3.8) und der folgenden Methode `notifyChangeListeners()`:

```
private synchronized void notifyChangeListeners()
{
    for (final ChangeListener listener : listeners)
    {
        listener.update(this);
    }
}
```

Auch hier gilt, dass eine Blockierung für parallele Zugriffe auf die Datenstruktur `listeners` erfolgt. Zudem werden über die Methode `update()` wiederum Aktionen in den Listnern ausgelöst. All dies erfolgt synchron. Eine lang andauernde Verarbeitung eines Listners in `update()` bremst alle anderen Zugriffe aus.

Es empfiehlt sich daher folgender Trick: Zur Entkopplung der Vorgänge »Benachrichtigung« sowie »An- und Abmeldung« von Beobachtern wird die Methode `notifyChangeListeners()` leicht modifiziert. Statt direkt auf der Liste der Beobachter zu arbeiten, legt man sich eine lokale Kopie dieser Liste an. Die Benachrichtigung der Listener erfolgt anschließend durch eine Iteration über diese lokale Kopie der Liste:

```
private void notifyChangeListeners()
{
    final List<ChangeListener> copyOfListeners;
    synchronized(this)
    {
        copyOfListeners = new LinkedList<ChangeListener>(listeners);
    }

    for (final ChangeListener listener : copyOfListeners)
    {
        listener.update(this);
    }
}
```

Diese Realisierung ist durch die Zugriffe auf die lokalen Daten automatisch Thread-sicher und sorgt zudem für mehr Nebenläufigkeit, da An- oder Abmeldevorgänge anderer Beobachter nur für den Zeitraum der Listenkopie ausgeschlossen sind. Es verbleibt allerdings das Problem einer lang andauernden Verarbeitung eines Listeners in `update()`.

Fazit Bei Realisierungen mit `synchronized` sind immer alle auf diesen Lock wartenden Aktionen blockiert, bis die momentan ausgeführte Aktion beendet ist und den gehaltenen Lock freigibt. Wie gezeigt, gibt es zwar Möglichkeiten, die Zeitdauer der Sperrung zu minimieren, in vielen Fällen ist jedoch eine solche exklusive Sperre für parallele Zugriffe gar nicht erforderlich. Dies gilt insbesondere, wenn viele Lesezugriffe und nur wenige Schreibzugriffe erfolgen sollen.

Thread-Sicherheit und Parallelität mit den Concurrent Collections

Wenn mehrere Threads gemeinsam ausschließlich lesend auf einer Liste arbeiten, muss nicht immer die gesamte Datenstruktur gesperrt und dadurch der Zugriff für andere Threads blockiert werden.

Häufig ist die Anzahl der Leseoperationen deutlicher höher als die der Schreibzugriffe. Lesezugriffe sollten sich gegenseitig nicht blockieren. Lese- und Schreibzugriffe müssen dagegen aufeinander abgestimmt werden. Dazu existieren verschiedene Verfahren. Zwei davon stelle ich kurz vor:

- **Kopieren beim Schreiben** – Die dahinterliegende Idee ist, vor jedem Schreibzugriff die Datenstruktur zu kopieren und dann das Element hinzuzufügen. Andere Threads können dadurch lesend zugreifen, ohne durch das Schreiben gestört zu werden. Diese Variante realisieren die Klassen `CopyOnWriteArrayList<E>` und `CopyOnWriteArraySet<E>` für Listen sowie Sets.
- **Lock-Striping / Lock-Splitting** – Bei diesem Verfahren werden verschiedene Teile eines Objekts oder einer Datenstruktur mithilfe mehrerer Locks geschützt. Dadurch sinkt die Wahrscheinlichkeit für gleichzeitige Zugriffe auf jeden einzelnen Lock und Threads werden seltener durch das Warten auf Locks am Weiterarbeiten gehindert. Als Folge davon steigt die Parallelität. Zur Realisierung der parallelen `ConcurrentHashMap<K, V>` wird genau dieses Verfahren angewendet, das auf die spezielle Arbeitsweise von Hashcontainern mit Buckets abgestimmt ist. Statt die gesamte `HashMap<K, V>` zu synchronisieren, wird stattdessen nur ein kleiner Ausschnitt der Datenstruktur geschützt.

In den verschiedenen Concurrent Collections werden diese Techniken eingesetzt, um neben Thread-Sicherheit auch für mehr Parallelität als bei den `synchronized`-Wrappern zu sorgen.

Datenzugriff in den Klassen `CopyOnWriteArrayList`/`-Set` Die Implementierung der Klasse `CopyOnWriteArraySet<E>` nutzt die Klasse `CopyOnWriteArrayList<E>`. Diese wiederum verwendet ein Array zur Speicherung von Elementen und passt dieses ähnlich der in Abschnitt ?? beschriebenen Größenänderung von Arrays an. Jede Änderung der Daten erzeugt eine Kopie des zugrunde liegenden Arrays. Dadurch können Threads ungestört davon parallel lesen.

Allerdings sollte man folgende zwei Dinge beim Einsatz bedenken: Für kleinere Datenmengen (< 1.000 Elemente) ist das Kopieren und die mehrfache Datenhaltung in der Regel vernachlässigbar. Der negative Einfluss steigt mit der Anzahl zu speichernder Elemente linear an. Aufgrund der gewählten Strategie des Kopierens bieten sich diese Datenstrukturen daher vor allem dann an, wenn deutlich mehr Lese- als Schreibzugriffe erfolgen und die zu speichernden Datenvolumina nicht zu groß sind.

Datenzugriff in der Klasse `ConcurrentHashMap` Betrachten wir die Klasse `ConcurrentHashMap<K, V>`, die eine Realisierung einer `Map<K, V>` ist und mehreren Threads paralleles Lesen und Schreiben ermöglicht. Die Realisierung garantiert, dass sich Lesezugriffe nicht gegenseitig blockieren. Für Schreibzugriffe kann man zum Konstruktionszeitpunkt bestimmen, wie viel Nebenläufigkeit unterstützt werden soll, indem man die Anzahl der Schreibsperrern festlegt.

Streng genommen ist die Klasse `ConcurrentHashMap<K, V>` lediglich ein Ersatz für die Klasse `Hashtable<K, V>` und nicht für die Klasse `HashMap<K, V>`, wie es der Name andeutet. Das liegt daran, dass die Klasse `ConcurrentHashMap<K, V>` im Gegensatz zur Klasse `HashMap<K, V>` keine `null`-Werte für Schlüssel und Werte unterstützt. Der Wert `null` drückt stattdessen aus, dass ein gesuchter Eintrag fehlt. In vielen Fällen wird diese Unterstützung nicht benötigt, und man kann die Klasse `ConcurrentHashMap<K, V>` dann problemlos auch als Ersatz für eine `HashMap<K, V>` verwenden.

Das Interface `ConcurrentMap<K, V>` deklariert vier gebräuchliche Mehrschrittoperationen. Das Besondere daran ist, dass diese von den konkreten Realisierungen `ConcurrentHashMap<K, V>` und `ConcurrentSkipListMap<K, V>` atomar ausgeführt werden müssen. Die folgende Aufzählung nennt die Methoden und zeigt zur Verdeutlichung der Funktionalität eine schematische Pseudoimplementierung. Die tatsächliche Implementierung ist wesentlich komplizierter, da sie Parallelität ohne Blockierung gewährleistet.

- `V putIfAbsent(K key, V value)` – Erzeugt einen neuen Eintrag zu diesem Schlüssel und dem übergebenen Wert, falls kein Eintrag zu dem Schlüssel existiert. Liefert den zuvor gespeicherten Wert zurück, falls bereits ein Eintrag zu dem Schlüssel vorhanden ist, oder `null`, wenn dies nicht der Fall war.

```
if (!map.containsKey(key))
{
    return map.put(key, value);
}
return map.get(key);
```

- `boolean remove(Object key, Object value)` – Entfernt den Eintrag zu diesem Schlüssel, falls der gespeicherte Wert mit dem übergebenen Wert übereinstimmt. Liefert `true`, falls es einen solchen Eintrag gab, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(oldValue))
    {
        map.remove(key);
        return true;
    }
}
return false;
```

- `V replace(K key, V value)` – Ersetzt zu diesem Schlüssel den gespeicherten durch den übergebenen Wert. Liefert den zuvor mit dem Schlüssel assoziierten Wert zurück oder `null`, wenn es keinen Eintrag zu dem Schlüssel gab:

```
if (map.containsKey(key))
{
    return map.put(key, newValue);
}
return null;
```

- `boolean replace(K key, V oldValue, V newValue)` – Ersetzt den Eintrag zu diesem Schlüssel mit dem neuen Wert, falls der gespeicherte Wert mit dem alten Wert übereinstimmt. Liefert `true`, falls dem so ist, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(oldValue))
    {
        map.put(key, newValue);
        return true;
    }
}
return false;
```

Die Aufrufe von `map.containsKey(key)` sind notwendig, zu verhindern, dass bei einem fehlenden Eintrag eine `NullPointerException` ausgelöst wird.

Iteration Um eine parallele Verarbeitung zu unterstützen, wurde das Verhalten der Iteratoren so angepasst, dass diese weder `ConcurrentModificationExceptions` werfen noch eine Synchronisierung benötigen. Allerdings sind die Iteratoren auch nur »schwach« konsistent oder »weakly consistent«, d. h., eine Iteration liefert nicht immer die aktuellste Zusammensetzung der Datenstruktur, sondern den Zustand zum Zeitpunkt der Erzeugung des Iterators. Nachfolgende Änderungen an der Zusammensetzung können bei der Iteration berücksichtigt werden, müssen es aber nicht. Dadurch können Iterationsvorgänge parallel zu Veränderungen erfolgen.

Blockierende Warteschlangen und das Interface `BlockingQueue`

Zum Austausch von Daten oder Nachrichten zwischen verschiedenen Programmkomponenten können Implementierungen des Interface `Queue<E>` (vgl. Abschnitt 5.4) verwendet werden. Das Interface `BlockingQueue<E>` erweitert das Interface `Queue<E>` um folgende Methoden:

- `boolean offer(E element, long time, TimeUnit unit)` – Fügt ein Element in die Queue ein, falls keine Größenbeschränkung existiert oder die Queue nicht voll ist. Ansonsten wartet der Aufruf blockierend maximal die angegebene Zeitspanne, dass eine andere Programmkomponente ein Element entfernt, so dass als Folge ein Einfügen möglich wird. Liefert `true`, wenn das Einfügen erfolgreich war, ansonsten `false`.
- `E poll(long time, TimeUnit unit)` – Gibt das erste Element zurück, und entfernt es aus der Queue. Wenn die Queue leer ist, wird maximal die angegebene Zeitspanne auf das Einfügen eines Elements durch eine andere Programmkomponente gewartet und nach einem erfolglosen Warten `null` zurückgegeben.
- `void put(E element)` – Fügt ein Element in die Queue ein. Dieser Aufruf erfolgt blockierend. Das bedeutet, dass gewartet werden muss, wenn die Queue eine Größenbeschränkung besitzt und zum Zeitpunkt des Einfügens voll ist.
- `E take()` – Gibt das erste Element zurück und entfernt es aus der Queue. Der Aufruf blockiert, solange die Queue leer ist.

Diese Methoden blockieren beim Schreiben in eine volle Queue bzw. beim Lesen aus einer leeren Queue, wodurch eine fehleranfällige Synchronisierung und Benachrichtigung über die Methoden `wait()` und `notify()` bzw. `notifyAll()` entfällt.

Die durch das Interface `BlockingQueue<E>` beschriebene Schnittstelle erinnert an das in Abschnitt 7.3 entwickelte Interface `FixedSizeContainer<T>` und die implementierende Klasse `BlockingFixedSizeBuffer<T>`. Ähnliche Funktionalität wird durch folgende Klassen der Concurrent Collections realisiert:

- `ArrayBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff, besitzt eine Größenbeschränkung und nutzt ein Array zur Speicherung.
- `LinkedBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff und verwendet eine `LinkedList<E>`, wodurch keine Größenbeschränkung gegeben ist.⁹ Vorteilhaft ist dies, wenn die Anzahl der zu speichernden Elemente dynamisch ist und im Vorhinein nur schlecht abgeschätzt werden kann. Das wurde ausführlich in Abschnitt 5.1.2 für Arrays und Listen diskutiert.
- `PriorityBlockingQueue<E>` – Diese Realisierung nutzt ein Sortierkriterium, um die Reihenfolge der Elemente innerhalb der Queue zu bestimmen. Elemente gemäß der höchsten Priorität stehen am Anfang der Queue und werden somit bei Lesezugriffen zuerst zurückgeliefert.

⁹Eine Begrenzung ergibt sich lediglich aus dem der JVM zugeteilten Speicher.

Zusätzlich gibt es noch zwei besondere Implementierungen:

- `SynchronousQueue<E>` – Diese Queue ist ein Spezialfall einer größenbeschränkten Queue, allerdings mit der Größe 0. Zunächst klingt dies unsinnig, ist aber für solche Anwendungsfälle geeignet, die erfordern, dass zwei Threads unmittelbar aufeinander warten. Bezogen auf das Producer-Consumer-Beispiel bedeutet dies, dass ein Producer erst sein Produkt »speichern« kann, wenn ein Consumer dieses direkt abholt. Andersherum gilt: Möchte ein Consumer Daten aus der Queue entnehmen, muss er warten, bis ein Producer Daten ablegt.
- `DelayQueue<E>` – Diese Art von Queue ist ähnlich zu einer `PriorityBlockingQueue<E>`. Zu speichernde Elemente beschreiben ihre Sortierung dadurch, dass sie das Interface `java.util.concurrent.Delayed` erfüllen, das das Interface `Comparable<Delayed>` erweitert und über die Methode `getDelay(java.util.concurrent.TimeUnit unit)` einen Ablaufzeitpunkt bis zur Aktivierung angibt. In einer solchen Queue sind die Elemente nach ihrem »Verfallsdatum« geordnet.

Beispiel: Producer-Consumer mit `BlockingQueue`

Das in Abschnitt 7.3 vorgestellte und dort ständig weiterentwickelte Beispiel des Producer-Consumer-Problems kann durch den Einsatz des Interface `BlockingQueue<E>` leicht auf die Concurrent Collections umgestellt werden. Dieser Umstieg ist möglich, da bereits eine konzeptionell ähnliche Realisierung existiert. Es ist lediglich ein Implementierungsdetail anzupassen (Einsatz eines anderen Interface).

```
public static void main(String[] args)
{
    final int MAX_QUEUE_SIZE = 7;
    final BlockingQueue<Item> items = new LinkedBlockingQueue<Item>(
        MAX_QUEUE_SIZE);

    new Thread(new Producer(items, 100)).start();

    // warte 2 Sekunden, dadurch sieht man die Größenbeschränkung
    SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

    new Thread(new Consumer(items, 1000, "Consumer 1")).start();
    new Thread(new Consumer(items, 1000, "Consumer 2")).start();
    new Thread(new Consumer(items, 1000, "Consumer 3")).start();
}
```

Listing 7.17 Ausführbar als `'PRODUCERCONSUMERBLOCKINGQUEUEEXAMPLE3'`

Info: Erweiterungen im Package `java.util.concurrent` mit JDK 6

Im JDK 6 sind folgende Interfaces neu hinzugekommen:

- `BlockingDeque<E>` – Dieses Interface beschreibt eine `Deque<E>` mit Methoden, die beim Holen eines Elements bzw. beim Speichern eines Elements solange warten, bis die gewünschte Aktion möglich ist. Dieses Interface erweitert die Interfaces `Deque<E>` und `BlockingQueue<E>`.
- `ConcurrentNavigableMap<K, V>` – Dieses Interface ist eine Kombination aus den Interfaces `ConcurrentMap<K, V>` und `NavigableMap<K, V>`. Letzteres ist eine Erweiterung der `SortedMap<K, V>` um die Möglichkeit, passende Elemente für übergebene Schlüsselwerte zu finden. Die Klasse `ConcurrentSkipListMap<K, V>` implementiert das Interface `ConcurrentNavigableMap<K, V>` und stellt einen auf Parallelität der Zugriffe ausgelegten Ersatz für die Klasse `TreeMap<K, V>` dar.

7.6.2 Das Executor-Framework

Das Executor-Framework vereinfacht den Umgang mit Threads und die Verarbeitung von Ergebnissen asynchroner Aufgaben, sogenannter *Tasks*. Es erfolgt eine Trennung der Beschreibung eines Tasks und dessen tatsächlicher Ausführung. Dabei wird vollständig von den Details abstrahiert, etwa wie und wann eine Aufgabe von welchem Thread ausgeführt wird. Die Kernidee ist, Tasks an sogenannte Executors zur Ausführung zu übergeben.

Zunächst stelle ich die wichtigsten und zentralen Interfaces und Klassen vor, bevor diese in einigen kleineren Beispielanwendungen genutzt werden.

Das Interface `Executor`

Das Interface `java.util.concurrent.Executor` dient dazu, Tasks (Aufgaben bzw. Arbeitsblöcke) auszuführen, die das Interface `Runnable` implementieren. Diese könnten im einfachsten Fall durch den Einsatz der Klasse `Thread` ausgeführt werden, wie dies bereits bekannt ist und durch folgende Anweisungen realisiert wird:

```
new Thread(runnableTask).start();
```

Statt explizit mit Threads zu arbeiten, kann man alternativ eine Realisierung des `Executor`-Interface nutzen, wodurch von den konkreten Aktionen »Erzeugen« und »Starten« abstrahiert wird. Das Interface `Executor` ist dazu wie folgt definiert:

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

Damit lässt sich die obige Ausführung folgendermaßen schreiben:

```
executor.execute(runnableTask);
```

Implementierung des Interface `Executor` Das Interface `Executor` gibt keine konkrete Art der Ausführung vor. Die Details werden durch die gewählte konkrete `Executor`-Implementierung spezifiziert, beispielsweise wie viele Aufgaben parallel ausgeführt werden und wann bzw. in welcher Reihenfolge dies erfolgt.

Betrachten wir zunächst zwei extreme Arten der Ausführung: alle Tasks synchron hintereinander und alle Tasks vollständig asynchron. Für Ersteres könnte man eine Klasse `SynchronousExecutor` wie folgt implementieren:

```
public class SynchronousExecutor implements Executor
{
    public void execute(final Runnable runnable)
    {
        runnable.run();
    }
}
```

In der Regel sollen Tasks nicht synchron im aufrufenden Thread, sondern parallel dazu ausgeführt werden. Eine Klasse `AsyncExecutor`, die für jeden Task einen eigenen Thread startet, könnte folgendermaßen realisiert werden:

```
public class AsyncExecutor implements Executor
{
    public void execute(final Runnable runnableTask)
    {
        new Thread(runnable).start();
    }
}
```

Sollen lediglich einige wenige Tasks ausgeführt werden, so ist kaum ein Vorteil durch den Einsatz eines Executors gegenüber dem Einsatz von Threads gegeben. Für Multithreading-Anwendungen, in denen diverse (unabhängige) Tasks von Threads ausgeführt werden sollen, kann eine derartige Abstraktion allerdings sehr vorteilhaft sein.

Motivation für Thread-Pools

Nehmen wir an, es seien viele Aufgaben zu erledigen. Die zuvor vorgestellten beiden möglichen Extreme bei der Umsetzung, d. h. die streng sequenzielle Ausführung aller Aufgaben durch einen einzigen Thread bzw. die maximale Parallelisierung durch Abspalten eines eigenen Threads pro Aufgabe, besitzen beide unterschiedliche Nachteile. Der erste Ansatz führt durch die Hintereinanderausführung zu langen Wartezeiten und schlechten Antwortzeiten. Es kommt zu einem schlechten Durchsatz. Der zweite Ansatz parallelisiert zwar die Ausführung und erreicht dadurch einen besseren Durchsatz. Etwas naiv könnte man auf die Idee kommen, einfach so viele Threads zu erzeugen, wie Aufgaben existieren, um dadurch die Parallelität der Abarbeitung zu maximieren

und für gute Antwortzeiten zu sorgen. Allerdings skaliert dieses Vorgehen nur begrenzt: Steigt die Anzahl der genutzten Threads deutlich über die Anzahl der verfügbaren CPUs bzw. Rechenkern, so sinkt der Leistungszuwachs. Dies ist dadurch bedingt, dass maximal so viele Threads gleichzeitig aktiv sein können, wie es CPUs bzw. Rechenkern gibt. Zudem steigt der Aufwand zur Verwaltung und Abstimmung der Threads und für Thread-Wechsel.

Weiterhin ist die Ausführung von Aufgaben durch Threads mit einem gewissen Overhead verbunden: Zum einen kostet das Erzeugen und Starten deutlich mehr Rechenzeit als ein Methodenaufruf. Zum anderen belegen Threads Betriebssystemressourcen und Speicher innerhalb und außerhalb der JVM. Sie können daher nur in begrenzter Anzahl¹⁰ erzeugt werden.

Einen sinnvollen Kompromiss erreicht man durch den Einsatz sogenannter *Thread-Pools*. Die Idee ist dabei, eine bestimmte Anzahl lauffähiger, aber pausierter Threads vorrätig zu halten und bei Bedarf zur Ausführung von Tasks zu aktivieren. Dadurch spart man sich die aufwendige Konstruktion neuer Threads. Thread-Pools ermöglichen zudem ein klares Programmdesign, da Verwaltungsaufgaben, d. h. das gesamte Thread-Management (Erzeugung, Zuteilung, Fehlerbehandlung usw.), aus der Applikation in den Thread-Pool verlagert wird. Ein solcher Thread-Pool kann auch dazu dienen, bei sehr hoher Last momentan nicht verarbeitbare Anfragen zu speichern und im Nachhinein zu bearbeiten. Alternativ kann ein Thread-Pool beim Auftreten von Belastungsspitzen versuchen, den Durchsatz zu erhöhen, indem temporär die Anzahl der Bearbeitungs-Threads erhöht wird.

Achtung: Nachteile von Thread-Pools

Thread-Pools bieten sich zwar an, wenn die auszuführenden Tasks möglichst unabhängig voneinander sind. Eine Mischung von langlaufenden und kurzen Tasks kann zu Problemen führen, wenn die »Langläufer« alle »Kurzläufer« blockieren. Die Ausführung kurzer Tasks wird dann eventuell unerwartet stark verzögert.

Wenn übergebene Tasks voneinander abhängig sind, kann der Einsatz eines Thread-Pools ungeeignet sein, da keine Abarbeitungsreihenfolge garantiert ist. Im Extremfall muss ein auszuführender Task auf das Ergebnis eines noch nicht durch den Thread-Pool ausgeführten Vorgänger-Tasks warten. Dadurch ist keine weitere Abarbeitung dieses Tasks mehr möglich.

Das Interface `ExecutorService`

Thread-Pools können im Executor-Framework über FABRIKMETHODEN (vgl. Abschnitt 12.1.2) der Utility-Klasse `java.util.concurrent.Executors` erzeugt werden. Diese Methoden geben Objekte vom Typ `ExecutorService` zurück. Dieser er-

¹⁰Die genaue Anzahl hängt von der verwendeten Stackgröße und dem zur Verfügung stehenden Hauptspeicher ab.

weitere das Interface `Executor` um die Möglichkeit, bereits laufende Aufgaben abbrechen zu können.

Mit folgenden Fabrikmethoden können spezielle Realisierungen des Interface `java.util.concurrent.ExecutorService` erzeugt werden:

- `newFixedThreadPool(int poolSize)` – Erzeugt einen Thread-Pool fester Größe.
- `newCachedThreadPool()` – Erzeugt einen Thread-Pool unbegrenzter Größe, der nach Bedarf wachsen und schrumpfen kann.
- `newSingleThreadExecutor()` – Erzeugt einen Thread-Pool, der lediglich einen Thread verwaltet und folglich übergebene Aufgaben sequenziell abarbeitet (dies jedoch parallel zur eigentlichen Applikation).
- `newScheduledThreadPool(int poolSize)` – Erzeugt einen Thread-Pool fester Größe, der eine verzögerte und periodische Ausführung unterstützt und damit als Ersatz für die Klassen `Timer` und `TimerTask` (vgl. Abschnitt 7.5.4) dienen kann.

Wenn lediglich gegen das Interface `ExecutorService` programmiert wird, können die verwendeten, konkreten Realisierungen ausgetauscht und auf den jeweiligen Einsatz abgestimmt gewählt werden. Ein weiterer Vorteil der eben genannten Thread-Pool-Implementierungen ist, dass diese eine Fehlerbehandlung durchführen: Kommt es bei der Abarbeitung eines Tasks zu einer Exception, so führt dies zum Ende des ausführenden Worker-Threads. Durch den Thread-Pool werden »sterbende« Worker-Threads automatisch ersetzt, sodass immer eine Abarbeitung garantiert wird. Dies ist ein entscheidender Vorteil. Beim Einsatz eines `Timers` führte eine Exception zum Abbruch aller angemeldeten `TimerTasks`.

Die erzeugten Thread-Pools stellen spezielle Ausprägungen der folgenden beiden Klassen dar:

- `ThreadPoolExecutor` – Diese Klasse nutzt einen Thread-Pool. Neu eintreffende Aufgaben werden von bereitstehenden Threads bearbeitet, d. h. von solchen, die momentan kein `Runnable` ausführen. Gibt es keinen freien Thread, so werden die Tasks in einer Warteliste (`BlockingQueue<Runnable>`) zur Abarbeitung vorgemerkt.
- `ScheduledThreadPoolExecutor` – Diese Klasse erlaubt es, Tasks zeitgesteuert zu bestimmten Zeitpunkten oder periodisch auszuführen und ähnelt damit der Klasse `Timer`.

Zur Ausführung von Tasks kann man diese durch Aufruf überladener `submit()`-Methoden an einen `ExecutorService` übergeben. Bevor ich auf die Details der Ausführung eingehe, stelle ich im Folgenden zwei Arten der Realisierung von Tasks vor.

Hinweis: Executor und ExecutorService selbst implementieren?

Wie bereits gesehen, ist es zwar möglich, eigene Implementierungen der Interfaces `Executor` und `ExecutorService` zu schreiben, normalerweise empfiehlt es sich jedoch, die in der obigen Aufzählung beschriebenen, bereits vorhandenen Klassen des Executor-Frameworks zu verwenden.

Callables und Runnables

Das bereits beschriebene Interface `Runnable` besitzt gewisse Beschränkungen: Um die Ergebnisse nebenläufiger Berechnungen wieder an den startenden Thread kommunizieren zu können, muss man sich gewisser Tricks bedienen. Ursache dafür ist, dass die `run()`-Methode weder ein Ergebniswert zurückgeben noch eine Checked Exception werfen kann. Ein Austausch von Ergebnissen ist daher lediglich dadurch möglich, dass diese in gemeinsam benutzten Datenstrukturen abgelegt werden. Dazu müssen in eigenen Realisierungen von `Runnable`s spezielle Attribute zur Speicherung von Rückgabewerten definiert werden, die nach Abschluss der Berechnung von Aufrufern ausgelesen werden können. Mögliche Probleme beim Zugriff mehrerer Threads habe ich bereits in Abschnitt 7.2 beschrieben. Daher wird der Zugriff hier synchronisiert:

```
public final class CalcDurationInMsAsRunnable implements Runnable
{
    private final TimeUnit timeUnit;
    private final long duration;
    private long result = -1;

    CalcDurationInMsAsRunnable(final TimeUnit timeUnit, final long duration)
    {
        this.timeUnit = timeUnit;
        this.duration = duration;
    }

    @Override
    public void run()
    {
        try
        {
            timeUnit.sleep(duration);
            synchronized(this)
            {
                result = timeUnit.toMillis(duration);
            }
        }
        catch (final InterruptedException e)
        {
            /* Initialisierungswert zurückgeben (-1) */
        }
    }

    public synchronized long getResult()
    {
        return result;
    }
}
```

Die Klasse `CalcDurationInMsAsRunnable` speichert zu Demonstrationszwecken lediglich eine übergebene Wartedauer in einer beliebigen angegebenen Zeiteinheit, schläft anschließend die Wartezeit und erlaubt durch Aufruf der Methode `getResult()` das Berechnungsergebnis (die Wartezeit in Millisekunden) abzufragen.¹¹

Der zusätzliche Implementierungsaufwand ist durch die Beschränkungen des `Runnable`-Interface begründet. Allerdings ist es weiterhin nicht immer einfach, das Ende einer Berechnung zu erkennen und erst im Anschluss daran Ergebnisse abzufragen. Aufgrund dieser Einschränkungen wurde das Interface `Callable<V>` eingeführt, womit man einem Aufrufer einen Rückgabewert übermitteln und außergewöhnliche Situationen über Exceptions kommunizieren kann. Dieses Interface ist folgendermaßen definiert:

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

Betrachten wir nun, wie sich die zuvor realisierte Klasse durch den Einsatz des `Callable<V>`-Interface klarer gestalten lässt. Das generische Interface erlaubt beliebige Rückgabetypen. Für dieses einfache Beispiel nutzen wir den Typ `Long`:

```
public class CalcDurationInMs implements Callable<Long>
{
    private final TimeUnit timeUnit;
    private final long duration;

    CalcDurationInMs(final TimeUnit timeUnit, final long duration)
    {
        this.timeUnit = timeUnit;
        this.duration = duration;
    }

    @Override
    public Long call() throws Exception
    {
        timeUnit.sleep(duration);

        return timeUnit.toMillis(duration);
    }
}
```

Wie man sieht, ist diese Realisierung kürzer und benötigt keine Hilfsvariable `result` zur Zwischenspeicherung der Berechnungsergebnisse mehr.

¹¹Fand eine Unterbrechung durch einen anderen Thread statt, so wird als Ergebnis der Wert -1 zurückgegeben.

Ausführen von Runnable und Callable: Das Future-Interface

Nachdem wir nun wissen, wie Tasks definiert werden, wenden wir uns deren asynchroner Ausführung durch einen `ExecutorService` zu. Dieser kann Tasks ausführen, die durch Implementierungen der Interfaces `Runnable` sowie `Callable<V>` realisiert sind. Wie bereits erwähnt, werden dazu überladene `submit()`-Methoden genutzt. Dies sind im Einzelnen:

- `<T> Future<T> submit(Callable<T> task)` – Es erfolgt eine Abarbeitung des `Callable` mit anschließender Möglichkeit, das Ergebnis auszuwerten.
- `Future<?> submit(Runnable task)` – Das übergebene `Runnable` wird von einem Thread des `ExecutorService` abgearbeitet. Aufgrund der zuvor beschriebenen Einschränkungen von `Runnables` ist es im Gegensatz zur Ausführung von `Callables` nur möglich, abzufragen, ob die Abarbeitung bereits beendet ist. Es wird jedoch kein Rückgabewert über `get()` geliefert, sondern immer der Wert `null`.
- `<T> Future<T> submit(Runnable task, T result)` – Erweitert den vorherigen Aufruf um die Möglichkeit, mit `get()` ein Ergebnis abfragen zu können. Nach der Abarbeitung des übergebenen Tasks wird der als Referenz übergebene Wert `result` vom Typ `T` zurückgegeben. Dies kann man nur dann sinnvoll nutzen, wenn dieser Typ veränderlich ist: Der berechnende Task erhält diese Referenz zum Konstruktionszeitpunkt und ändert sie dort als Folge seiner Berechnungen. Dies stellt zwar einen Seiteneffekt dar, dieser ist aber lokal begrenzt und daher vertretbar. Folgende Zeilen deuten die Abarbeitung für ein Ergebnis vom Typ `List<Integer>` und einen `Task ModifyingTask` an:

```
public static final class ModifyingTask implements Runnable
{
    private final List<Integer> result;

    ModifyingTask(final List<Integer> result)
    {
        this.result = result;
    }

    @Override
    public void run()
    {
        result.add(Integer.valueOf(4711));
    }

    public List<Integer> getResult()
    {
        return result;
    }
}

public static void main(String[] args)
{
    final int POOL_SIZE = 3;
    final ExecutorService executorService = Executors.newFixedThreadPool(
        POOL_SIZE);
```

```

final List<Integer> result = new ArrayList<Integer>();
final Future<List<Integer>> future = executorService.submit(new
    ModifyingTask(result), result);

try
{
    System.out.println("isDone? " + future.isDone());

    final List<Integer> calculatedResult = future.get();
    System.out.println("After Job: " + new Date());
    System.out.println(calculatedResult);

}
catch (final InterruptedException e)
{
    // Kann in diesem Beispiel nicht auftreten
}
catch (final ExecutionException e)
{
    // Kann in diesem Beispiel nicht auftreten, wird geworfen wenn
    // versucht wird, auf ein Ergebnis eines Tasks zuzugreifen, der
    // mit einer Exception beendet wurde
}

executorService.shutdown();
}

```

Listing 7.18 Ausführbar als 'FUTUREEXAMPLEWITHRUNNABLEANDRESULT'

Die von den Tasks gelieferten Ergebnisse werden asynchron berechnet und über das bisher nicht näher beschriebene Interface `java.util.concurrent.Future<T>` bereitgestellt. Da weder der genaue Ausführungszeitpunkt noch die Zeitdauer der einzelnen Tasks bekannt sind, ist es als Konsequenz unmöglich, direkt das Ende einer Berechnung festzustellen, um auf das Ergebnis zuzugreifen. Zum Verfolgen des Ausführungsfortschritts eines übergebenen Tasks erfolgt die Rückgabe eines `Future<T>`-Interface, das sowohl das Ergebnis einer Berechnung als auch deren Asynchronität kapselt. Mithilfe dieses Interface kann man auch den Lebenszyklus von Tasks abfragen. Überladene `get()`-Methoden ermitteln den Ergebniswert und blockieren so lange, bis das Ergebnis verfügbar ist oder eine angegebene Time-out-Zeit überschritten wurde. Das Interface `Future<V>` ist folgendermaßen definiert:

```

public interface Future<V>
{
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit) throws
        InterruptedException, ExecutionException, TimeoutException;
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}

```

Parallele Abarbeitung im ExecutorService

Nachdem die Verarbeitung über eine der überladenen `submit()`-Methoden angestoßen wurde, können parallel weitere Aufgaben gestartet werden. Jede Einzelne kann man über das zurückgelieferte `Future<T>`-Interface mit der Methode `isDone()` befragen, ob die Berechnung beendet ist. Folgendes Listing zeigt dies exemplarisch für zwei Berechnungen mit der bekannten Task `CalcDurationInMs`. Der erste Task wartet 5, der zweite 10 Sekunden:

```
public static void main(String[] args)
{
    final int POOL_SIZE = 3;
    final ExecutorService executorService = Executors.newFixedThreadPool(
        POOL_SIZE);

    // Definition und Start zweier Tasks
    final Future<Long> future1 = executorService.submit(new CalcDurationInMs(
        TimeUnit.SECONDS, 5));
    final Future<Long> future2 = executorService.submit(new CalcDurationInMs(
        TimeUnit.SECONDS, 10));

    System.out.println("Start: " + new Date());
    try
    {
        // synchron auf das Ende von Task 1 warten
        final Long result1 = future1.get();
        System.out.println("After Job 1: " + new Date());
        System.out.println(result1);

        // Zugriff nach 5s sollte false liefern
        System.out.println("isDone? Job 2: " + future2.isDone());

        // synchron auf das Ende von Task 2 warten
        final Long result2 = future2.get();
        System.out.println("After Job 2: " + new Date());
        System.out.println(result2);
    }
    catch (final InterruptedException e)
    {
        // Kann in diesem Beispiel nicht auftreten
    }
    catch (final ExecutionException e)
    {
        // Kann in diesem Beispiel nicht auftreten, s. o.
    }

    // Aufruf, um Thread-Pool zu beenden und JVM-Terminierung zu ermöglichen
    executorService.shutdown();
}
```

Listing 7.19 Ausführbar als 'EXECUTORSERVICEEXAMPLE'

Mehrere Tasks abarbeiten

Die Methode `submit()` vom `ExecutorService` nimmt zu einem Zeitpunkt genau ein `Callable<V>` an und führt es parallel aus. Häufig möchte man jedoch mehrere Tasks parallel ausführen. Das kann man durch sukzessive Aufrufe an `submit()` erreichen. Eine Alternative ist, die folgenden, allerdings blockierenden Methoden des `ExecutorService` zu nutzen:

- `invokeAll(Collection<Callable<T>> tasks)` – Führt alle Aufgaben aus und liefert eine Liste von `Future<T>`-Objekten, die Zugriff auf die Ergebnisse bieten.
- `invokeAny(Collection<Callable<T>> tasks)` – Führt alle Aufgaben aus und liefert nur das Ergebnis des Threads, der als Erster fertig ist.

Für beide Methoden existiert eine überladene Version, der man eine maximale Ausführungszeit mitgeben kann. Das berechnete Ergebnis wird nur geliefert, wenn es zu keiner Zeitüberschreitung kommt.

ScheduledExecutorService VS. Timer

Das Interface `ScheduledExecutorService` bietet Methoden, um Aufgaben zu bestimmten Zeiten bzw. wiederholt auszuführen, und wird von der Klasse `ScheduledThreadPoolExecutor` implementiert. Die Intention ist vergleichbar mit derjenigen der Klasse `Timer`. Im Gegensatz zu dieser werden jedoch mehrere Threads aus einem Pool zur Ausführung benutzt. Außerdem können Ausführungszeiten nicht nur in Millisekunden angegeben werden, sondern mit Hilfe der Klasse `TimeUnit` in beliebigen Zeiteinheiten. Dies trägt deutlich zur Lesbarkeit bei.

Kommen wir auf das Beispiel aus Abschnitt 7.5.4 zurück, das zur Demonstration der Arbeitsweise der Klassen `Timer` und `TimerTask` gedient hat. Statt eines `TimerTask` zur Ausgabe eines Textes auf der Konsole verwende ich hier folgende simple Implementierung eines `Runnable`s mit gleicher Aufgabe:

```
public static void main(String[] args)
{
    final int POOL_SIZE = 3;
    final ScheduledExecutorService executorService = Executors.
        newScheduledThreadPool(POOL_SIZE);

    // Sofortige Ausführung
    executorService.schedule(new SampleMessageTask("OnceImmediately"), 0,
        TimeUnit.SECONDS);

    // Ausführung nach fünf Sekunden
    executorService.schedule(new SampleMessageTask("OnceAfter5s"), 5,
        TimeUnit.SECONDS);

    // Ausführung nach einer Minute
    executorService.schedule(new SampleMessageTask("OnceAfter1min"), 1,
        TimeUnit.MINUTES);
}
```

```

public static class SampleMessageTask implements Runnable
{
    private final String message;

    SampleMessageTask(final String message)
    {
        this.message = message;
    }

    public void run()
    {
        System.out.println(message);
    }
}

```

Listing 7.20 Ausführbar als 'SCHEDULEDEXECUTOREXAMPLE'

Durch die Verwendung der Klasse `TimeUnit` lassen sich Zeitangaben lesbarer darstellen. Ansonsten existiert kaum ein Unterschied zu den Methoden der Klasse `Timer`. Allerdings kann, im Gegensatz zu dieser, keine Einplanung über die Angabe eines Zeitpunkts in Form der Klasse `Date` erfolgen. Dafür lässt sich die periodische Ausführung noch klarer als bei der Klasse `Timer` ausdrücken.

Die Methoden zur Ausführung mit festem Takt und festgelegter Verzögerung zwischen den Ausführungen besitzen sprechende Namen: `scheduleAtFixedRate()` und `scheduleWithFixedDelay()`. Letztere Ausführungsart wurde im `Timer` durch die Methode `schedule()` angestoßen. Besser lesbar ist der Methodenaufruf `scheduleWithFixedDelay()`. Folgendes Beispiel zeigt beide Varianten der wiederholten Ausführung:

```

public static void main(String[] args)
{
    final int POOL_SIZE = 3;
    final ScheduledExecutorService executorService = Executors.
        newScheduledThreadPool(POOL_SIZE);

    // Eingeplante Ausführung mit INITIAL_DELAY und Verzögerung DELAY
    final long INITIAL_DELAY_ONE_SEC = 1;
    final long DELAY_30_SECS = 30;
    executorService.scheduleWithFixedDelay(new SampleMessageTask(
        "PeriodicRefreshing"), INITIAL_DELAY_ONE_SEC, DELAY_30_SECS,
        TimeUnit.SECONDS);

    // Eingeplante Ausführung mit INITIAL_DELAY und Takt RATE
    final long RATE_10_SECS = 10;
    executorService.scheduleAtFixedRate(new SampleMessageTask(
        "10s FixedRateExecution"), INITIAL_DELAY_ONE_SEC, RATE_10_SECS,
        TimeUnit.SECONDS);
}

```

Listing 7.21 Ausführbar als 'SCHEDULEDEXECUTORSERVICEEXAMPLE'

Fazit

Durch das Executor-Framework kann man von vielen Details der Thread-Verwaltung abstrahieren und es gibt nahezu keinen Grund mehr, direkt mit Threads zu arbeiten. Wenn viele Tasks auszuführen sind, gelten diese Aussagen umso mehr. *In neuen Projekten sollten bevorzugt die verschiedenen **Executor-Klassen** eingesetzt werden, da sie eine bessere Abstraktion als die Klassen **Thread** und **Timer** bieten.* Dies empfiehlt auch Joshua Bloch in seinem Buch »Effective Java« [8] in Item 68.

7.7 Weiterführende Literatur

Da es sich bei dem Thema Nebenläufigkeit um ein komplexes Thema handelt, ist es sehr sinnvoll, weitere Quellen zu konsultieren. Weiterführende Informationen finden sich in folgenden Büchern:

- **»Parallele und verteilte Anwendungen in Java«** von Rainer Oechsle [56]
Dieses Buch gibt einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5.
- **»Java Threads«** von Scott Oaks und Henry Wong [55]
Ebenso wie das Buch von Rainer Oechsle bietet dieses Buch einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5. Durch den Fokus auf Threads erklärt es einige Dinge noch gründlicher.
- **»Java Concurrency in Practice«** von Brian Goetz et al. [26]
Dieses Buch gibt eine sehr umfassende und fundierte Beschreibung zum Thema Multithreading. Zudem ist dies derzeit die aktuellste Quelle zu diesem Thema, dort werden auch die Erweiterungen der Concurrent Collections in JDK 6 behandelt.
- **»Concurrent Programming in Java«** von Doug Lea [46]
Dieser Klassiker, der bereits einige Jahre auf dem Buckel hat, stammt vom Entwickler der Concurrency Utilities selbst. Viele der im Buch beschriebenen Klassen haben später Einzug in die Java-Klassenbibliothek gehalten. Es ist allerdings keine leichte Lektüre.
- **»Taming Java Threads«** von Allen Holub [33]
Dieses Buch gibt einen fundierten Einstieg und beschreibt sehr genau die Details von Multithreading inklusive diverser Fallstricke. Es werden auch fortgeschrittenere Themen wie Multithreading und Swing, Thread-Pools und blockierende Warteschlangen behandelt. Da dieses Buch noch auf Java 1.4 basiert, werden hier ähnliche Ideen und Klassen entwickelt wie bei Doug Lea.
- **»Fortgeschrittene Programmierung mit Java 5«** von Johannes Nowak [54]
Dieses Buch beschäftigt sich intensiv mit den in JDK 5 hinzugekommenen Sprach-Features Generics und den Concurrency Utilities. Es geht nicht so detailliert wie die zuvor genannten Bücher auf die Thematik ein, gibt aber einen guten Überblick.