Vorwort

Warum ist das Schreiben von nebenläufiger Software so schwer? Zeitgleiche Abläufe beherrschen doch unseren Alltag. Wir arbeiten in Teams, koordinieren unsere Termine und übernehmen oder verteilen Aufgaben. In der Regel kommen wir mit dieser Art der Parallelität ganz gut zurecht.

Die uns vertraute Parallelität erweist sich bei der Entwicklung von Softwaresystemen als schwer zugänglich. Das liegt sicherlich mit daran, dass wir dabei immer das Ganze im Blick haben und Abläufe immer wieder über neue Koordinationsregeln steuern müssen. Darüber hinaus haben wir es technikbedingt mit einer anderen Art der Beschreibung von Parallelität zu tun.

Die Abstraktion der nebenläufigen Programmierung ist bei vielen Konzepten der *Thread*, ein Kontrollfluss bzw. -faden, der unabhängig von anderen agiert und durch einen Programmcode gesteuert wird. Diese Beschreibungsweise hat ihren Ursprung in der sequenziellen Programmierung, bei der es genau einen Ablaufstrang gibt. Leider ist es für uns auch im normalen Leben unmöglich, viele gleichzeitige, obwohl sequenzielle Vorgänge zu bewältigen. Diese Parallelitätsabstraktion ist intuitiv nicht leicht zugänglich; wir denken im Alltag nicht in *Threads*.

Der Umgang mit Threads birgt deshalb zahlreiche Fehlerquellen. Viele Multithreaded-Anwendungen enthalten Anomalien, die erst nach Monaten oder Jahren auftreten (siehe z. B. [36]). Nicht reproduzierbare Programmabstürze oder Verklemmungen, die häufig erst spät im Produktivbetrieb auftreten, sind typische Symptome dafür.

Um einfache, sichere Programmiermodelle zu ermöglichen, versucht man auf den in der Sprache vorhandenen primitiven Mechanismen *Abstraktionskonzepte* und *Frameworks* aufzubauen. Auf diesem Gebiet hat sich in den letzten Jahren sehr viel getan. Insbesondere wurde die Programmiersprache Java um viele solcher Konzepte erweitert.

Die nebenläufige Programmierung ist keine neue Domäne und es existiert auch schon viel Literatur hierzu. Einen guten Überblick über diesen komplexen Themenbereich findet man z. B. in dem Buch *Multicore-Software* von Urs Gleim und Tobias Schüle [15]. Im Bereich der Java-Programmierung gilt nach wie vor das Buch von Doug Lea *Concurrent Programming in Java: Design Principles and Patterns* [34] als Standard-

werk. Viele Ideen aus diesem Buch wurden sukzessive in die einzelnen Java-Versionen übernommen. Als Fortsetzung dieses Werks gilt das 2005 erschienene Buch *Java Concurrency in Practice* von Brian Goetz et al. [16], das ausführlich das Java 5 *Concurrency-API* diskutiert. Gute Beiträge zu vielen einzelnen Themen gibt es z. B. von Klaus Kreft und Angelika Langer [30] oder Heinz Kabutz [27].

Mit dem vorliegenden Buch möchten wir an diese Literatur anknüpfen und eine fundierte Einführung in die nebenläufige Programmierung mit Java geben und insbesondere auch die in den letzten Jahren eingeführten Konzepte und Frameworks detailliert beschreiben. Das Buch richtet sich vor allem an erfahrene Softwareentwickler sowie fortgeschrittene Studenten, die nebenläufige Konzepte in Projekten einsetzen möchten.

Wir hoffen, dass Ihnen das Buch *Nebenläufige Programmierung mit Java* gefällt und vor allem, dass es Ihnen ein guter Ratgeber ist.

Aufbau des Buches

Das Buch besteht aus fünf Teilen. Im ersten Teil werden die für die nebenläufige Programmierung grundlegenden Konzepte besprochen. Es wird der Thread-Mechanismus eingeführt und die Koordinierung nebenläufiger Programmflüsse durch rudimentäre *Low-Level*-Synchronisationsmechanismen erläutert. Im Wesentlichen sind dies die Verfahrensweisen, die seit Einführung von Java im Sprachumfang zur Verfügung stehen. Die Basiskonzepte bilden die Grundlage für die weiteren Teile des Buches.

Mit dem Aufkommen von Multicore-Prozessoren und den damit verbundenen Möglichkeiten ist die nebenläufige Programmierung immer mehr in den Fokus der Anwendungsentwicklung gerückt. Da die rudimentären Konzepte sehr leicht zu fehleranfälligen Implementierungen führen, wurde mit Java 5 ein umfangreiches *Concurrency-API* eingeführt, das mit den folgenden Versionen immer wieder ausgebaut wurde und wird.

Im Teil zwei werden verschiedene weiterführende Konzepte, wie *Threadpools*, *Futures*, *Atomic*-Variablen und *Locks*, vorgestellt.

Im dritten Teil werden ergänzende Synchronisationsmechanismen zur Koordinierung mehrerer Threads eingeführt. Neben dem `Exchanger` sind dies die Klassen `CountDownLatch`, `CyclicBarrier` und `Phaser`.

Teil vier bespricht die Parallelisierungsframeworks, mit denen auf einfache Art und Weise nebenläufige Programme erstellt werden können. Die Frameworks übernehmen hier im Wesentlichen die Thread-Koordination und -Synchronisation. Im Einzelnen werden das *ForkJoin*-Framework, die *Parallel Streams* und die Klasse `CompletableFuture` besprochen. Das ForkJoin-Framework erlaubt die Parallelisierung von *Divide-and-Conquer*-Algorithmen und parallele Streams die zeitgleiche Verarbeitung von Datensammlungen, wie z. B. Collections. Die Klasse `CompletableFuture` ent-

spricht einem Framework zur Erstellung von asynchronen Abläufen und ist eine Erweiterung des *Future*-Mechanismus um sogenannte *push*-Methoden.

Der fünfte Teil widmet sich der Anwendung der vorgestellten Konzepte und Klassen. Hierbei wurden die Beispiele aus verschiedenen Anwendungsgebieten ausgewählt. Des Weiteren gehen wir kurz auf das Thread-Konzept von JavaFX und Android ein. Abschließend stellen wir das Programmiermodell mit Aktoren vor, wobei hier das Akka-Framework benutzt wird, da im Java-Standard selbst (noch) kein solches Framework vorhanden ist.

Im Anhang geben wir der Vollständigkeit halber einen kurzen Ausblick auf Java 9, das bezüglich des *Concurrency*-API kleine Neuerungen bringt.

Vorausgesetzt werden gute Java-Kenntnisse, und erste Erfahrungen im Umgang mit *Lambda*-Ausdrücken wären wünschenswert. Als ergänzende Literatur empfehlen wir das Buch von Michael Inden [25], von dem wir auch einige Praxistipps übernommen haben. Die Streams von Java 8 und die in dem Zusammenhang benötigten funktionalen Interfaces werden in Kapitel 14 eingeführt.

Bemerkungen zu den Codebeispielen

Wir haben versucht, die Beispiele »so einfach wie möglich und so komplex wie notwendig« zu halten. Insbesondere sind die Fallbeispiele im fünften Teil noch nicht »voll praxistauglich«. Der benutzte *Coding Style* ist zum großen Teil der Buchform angepasst, was z. T. herausfordernd ist, da hier die Zeilenbreite sehr beschränkt ist. Das macht insbesondere die Darstellung von `Stream`- und `CompletableFuture`-Operationen oft schwierig. Wenn möglich, haben wir für das Verständnis nicht relevanten Code weggelassen. Insbesondere wird stets auf die `import`-Anweisungen verzichtet. Alle Codebeispiele findet man auch auf unserer Webseite zum Download. Bei den besprochenen APIs haben wir keinen Wert auf Vollständigkeit gelegt, sondern versucht, das Wesentliche zu extrahieren. Mit dem hier erworbenen Verständnis sollte man keine Probleme haben, die API-Dokumentation zu verstehen. Ein Blick in die Dokumentation ist immer zu empfehlen, da mittlerweile auch Verwendungsbeispiele aufgenommen wurden.

Danksagungen

Ein herzliches Dankeschön geht an die Mitarbeiter des dpunkt.verlags und insbesondere an Frau Christa Preisendanz, die die Fertigstellung des Buches professionell begleitet haben. Wir möchten uns auch bei unseren Studenten und den Reviewern, insbesondere Prof. Dr. Schiedermeier und Michael Inden, für die kritische Prüfung und kompetenten Hinweise bedanken. Zu guter Letzt geht auch ein Dank an unsere Familien für die Unterstützung und die Geduld.

Trotz sorgfältiger Prüfung wird das Buch wahrscheinlich leider noch Fehler enthalten, für die ausschließlich die Autoren verantwortlich sind. Falls Sie Fehler finden, lassen Sie es uns bitte wissen.

Jörg Hettel und Manh Tien Tran
Zweibrücken, Juni 2016

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