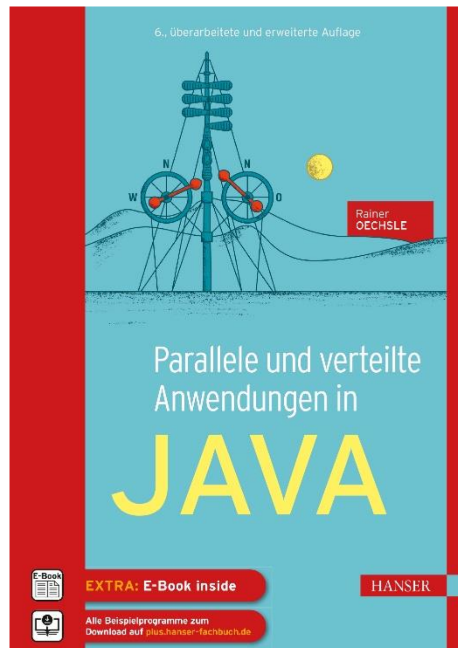
HANSER



Leseprobe

zu

Parallele und verteilte Anwendungen in Java

von Rainer Oechsle

Print-ISBN: 978-3-446-46919-8
E-Book-ISBN: 978-3-446-47348-5
E-Pub-ISBN: 978-3-446-47504-5

Weitere Informationen und Bestellungen unter
https://www.hanser-kundencenter.de/fachbuch/artikel/9783446469198
sowie im Buchhandel

© Carl Hanser Verlag, München

Vorwort zur 6. Auflage

Dieses Buch handelt von der Entwicklung paralleler und verteilter Anwendungen in Java. Nach einem einleitenden Kapitel, in dem wichtige Begriffe wie Programme, Prozesse und Threads auch anhand einer Metapher aus dem täglichen Leben erläutert werden, lassen sich die folgenden acht Kapitel in drei Teile gruppieren:

- Entwicklung paralleler Anwendungen in Java (Kapitel 2 und 3): Das Buch beginnt mit einer relativ ausführlichen Einführung in das Gebiet der parallelen Programmierung. Die Sachverhalte sind für Neulinge oft anspruchsvoll, denn Programmcode, der bei rein sequenzieller Ausführung korrekt ist, kann im Fall einer parallelen Nutzung fehlerbehaftet sein. Der Einstieg in das Thema Parallelität wird im Zusammenhang mit Objektorientierung für viele noch problematischer. Denn zum einen muss man verstehen, dass es Thread-Objekte gibt, dass diese aber nicht identisch mit den parallelen Aktivitäten, den Threads selbst, sind. Zum anderen muss man begreifen lernen, dass es mehrere Objekte einer Klasse geben kann, dass aber ein einziges Objekt (quasi) gleichzeitig von mehreren Threads verwendet werden kann, d. h., dass dieselbe und unterschiedliche Methoden auf einem Objekt gleichzeitig parallel ausgeführt werden können. In diesem Buch wird in die Gedankenwelt der Parallelität mit zahlreichen Programmbeispielen behutsam eingeführt (Kapitel 2). Es werden dann aber auch die grundlegenden Ideen weiterer anspruchsvoller Konzepte aus der Java-Concurrent-Bibliothek wie das Fork-Join-Framework, sequenzielles und paralleles Data-Streaming sowie CompletableFutures behandelt, ohne auf alle Details dieser Konzepte einzugehen (Kapitel 3).
- Entwicklung von Anwendungen mit grafischer Benutzeroberfläche in Java (Kapitel 4): Grafische Benutzeroberflächen scheinen auf den ersten Blick nichts mit parallelen und verteilten Anwendungen zu tun zu haben. Bei näherem Hinsehen erkennt man aber durchaus Zusammenhänge. So wird zum Beispiel in diesem Buch ausführlich erläutert, welche negativen Effekte es bei naiver Programmierung für Anwendungen mit grafischer Benutzeroberfläche gibt, wenn eine länger dauernde Aktivität aufgrund einer Interaktion mit der grafischen Benutzeroberfläche gestartet wird. Insbesondere bei verteilten Anwendungen können länger dauernde Aktivitäten immer bei einer Kommunikation zwischen einem Client und einem Server über das Internet vorkommen. Die Probleme lassen sich mithilfe von Parallelität lösen. Da Client-Programme oft und Server-Programme manchmal eine grafische Benutzeroberfläche haben, spielt also das Thema Parallelität bei verteilten Anwendungen mit grafischer Benutzeroberfläche eine Rolle. Aber auch wichtige Strukturierungsprinzipien für lokale Programme mit grafischer Benutzeroberfläche wie das MVP-Architekturmuster (MVP: Model View Presenter) lassen sich auf verteilte Anwendungen übertragen.

- Entwicklung verteilter Anwendungen in Java (Kapitel 5 bis 9): Verteilte Anwendungen folgen häufig dem Client-Server-Prinzip. Auch hier besteht wieder ein enger Zusammenhang zur Parallelität, denn auf Server-Seite ist fast immer Parallelität notwendig, um mehrere Clients (quasi) gleichzeitig zu bedienen und somit die Bedienung eines Clients nicht beliebig lange durch die Bearbeitung eines länger dauernden Auftrags eines anderen Clients zu verzögern. Um die parallele Bearbeitung von Client-Aufträgen zu erreichen, müssen die Threads bei der Programmierung eines Servers auf Socket-Basis (Kapitel 5) selbst explizit erzeugt werden. Wenn RMI (Kapitel 6) oder Servlets und Java Server Faces (Kapitel 8) benutzt werden, dann werden Threads implizit (d. h. nicht im Programmcode der Anwendung) erzeugt. Dies muss man wissen und den Umgang damit beherrschen, wenn man korrekte Server-Programme schreiben will. Wenn die Kommunikation zwischen den Rechnern nicht mehr direkt, sondern über einen Vermittler (Broker) erfolgt, erreicht man eine losere Kopplung zwischen den kommunizierenden Parteien mit einigen Vorteilen (Kapitel 7). Viele verteilte Anwendungen nutzen heutzutage Cloud-Dienste. Es werden abschließend einige Cloud-Anwendungen unter Nutzung der Cloud-Dienste von Amazon präsentiert (Kapitel 9).

In vielen Lehrbüchern werden Parallelitätsaspekte bei Programmen mit grafischer Benutzeroberfläche oder bei Server-Programmen nicht genügend oder überhaupt nicht berücksichtigt. So habe ich einige Beispiele in Lehrbüchern gefunden, die das Thema Parallelität vollständig ignorieren. Folglich sind solche Programme mit fehlender Synchronisation oftmals nicht korrekt, was beim oberflächlichen Ausprobieren in der Regel (zum Glück oder leider?) nicht auffällt. In diesem Buch wird dagegen durchgängig für alle Anwendungen ein besonderes Augenmerk auf Parallelitätsaspekte gelegt.

Dieses Buch ist weder ein Handbuch mit allen Details, die man bei der Software-Entwicklung benötigt, noch ist es ein Überblicksbuch, in dem eine Fülle von Themen nur angerissen wird. Stattdessen versucht es seinem Charakter als Lehrbuch gerecht zu werden, indem es die Grundprinzipien zentraler Konzepte herausarbeitet. Der Fokus liegt auf den beiden eng miteinander verzahnten Themen Parallelität (Nebenläufigkeit) und Verteilung. Bei dem Thema parallele Programmierung ist es in der Praxis sicher ratsam, nach Möglichkeit die Klassen aus der Java-Concurrent-Klassenbibliothek zu verwenden, die in Kapitel 3 auch behandelt werden. Allerdings wird die Mehrzahl der Beispiele aus didaktischen Gründen ohne Nutzung dieser Bibliothek entwickelt. Ähnliches trifft auf Servlets und JSF (Java Server Faces) zu: In der Praxis wird man eher JSF nutzen, im Buch werden die meisten Beispiele mit Servlets entwickelt, um den Leserinnen und Lesern besser verständlich zu machen, was bei der Ausführung des Programms passiert. Auch in dieser Auflage behandle ich immer noch RMI sehr intensiv. Man mag der Meinung sein, dass RMI inzwischen veraltet ist, aber aus meiner Sicht ist RMI weiterhin eine sehr elegante und konsequent zu Ende gedachte Realisierung einer Client-Server-Kommunikation. Ich bin überzeugt davon, dass die intensive Beschäftigung mit RMI elementar wichtige Aspekte der Informatik wie zum Beispiel den Unterschied zwischen Call-by-value und Call-by-result gut verständlich macht. So ist die Beschäftigung mit den hier vorhandenen Inhalten nicht nur dazu da, um aktuell notwendige Kenntnisse und Fertigkeiten für die Berufswelt zu erlernen, sondern vor allem zum Erlernen grundlegender Informatikkonzepte. Aus diesem Grund ist die hier verwendete Programmiersprache Java auch nur ein Vehikel zur Darstellung unterschiedlicher Aspekte aus dem Bereich der Programmierung paralleler und verteilter Anwendungen. So wie dieselben Inhalte dieses Buchs statt in Deutsch in einer anderen Sprache wie Englisch oder

Französisch vermittelt werden könnten, so ließen sich viele der vorgestellten Konzepte auch durch Programmbeispiele in anderen Programmiersprachen wie etwa C++ oder C# illustrieren.

Bei der Auswahl des Lehrstoffs ist es immer auch schmerzlich, viele interessante und relevante Themen nicht tiefer oder gar nicht behandeln zu können aufgrund des begrenzten Umfangs des Buchs. So könnte das neue Kapitel 9 zum Thema Cloud Computing auch leicht so ausgedehnt werden, dass es ein ganzes Buch füllen würde. Zu den leider gar nicht behandelten Themen gehören beispielsweise die Bereiche Spring Boot und Kubernetes. Bei der Stoffauswahl muss man eben Kompromisse eingehen.

Für diese sechste Auflage wurden neben der Korrektur von Fehlern, die in der fünften Auflage bemerkt wurden, folgende wesentlichen Änderungen vorgenommen:

- Das Kapitel 7 zur indirekten Kommunikation wurde neu geschrieben.
- Außerdem ist Kapitel 9 zum Thema Cloud Computing neu dazugekommen.
- In Kapitel 5 wird im neu geschriebenen Abschnitt 5.7 nun auch verschlüsselte Kommunikation über SSL (Secure Socket Layer) bzw. TLS (Transport Layer Security) behandelt.
- Da Java EE (Enterprise Edition) von der Firma Oracle nicht mehr weiterentwickelt wird, sondern dies von der Eclipse Foundation übernommen wurde und nun den Namen Jakarta EE trägt, wurden Bezüge auf die Enterprise Edition von Java EE in Jakarta EE geändert.
- Der Abschnitt 6.8 (Laden von Klassen über das Netz) im RMI-Kapitel der fünften Auflage wurde ersatzlos gestrichen, da der dort benutzte Security Manager seit der Java-Version 17 „deprecated" ist (d. h. nicht mehr benutzt werden sollte, da er in einer späteren Version nicht mehr vorhanden sein könnte).
- Ferner wurden mehrere kleinere Anpassungen vorgenommen wie z. B. die Ergänzung des Abschnitts 2.1.4 über parallele Abläufe oder die Ergänzung des Abschnitts 5.6.4 über horizontale Skalierung.

Die Beispielprogramme folgen gängigen Programmierkonventionen für Java bezüglich der Groß- und Kleinschreibung von Bezeichnern und dem Einrücken. Alle Bezeichner für Klassen, Schnittstellen, Methoden und Attribute sind einheitlich in Englisch geschrieben. Die Ausgaben, die von den Programmen erzeugt werden, sind jedoch alle in deutscher Sprache. In den abgedruckten Programmen wurden alle Package-Anweisungen entfernt (bis auf eine Ausnahme in Kapitel 9, Listing 9.6, weil bei diesem Beispiel auf den Package-Namen explizit Bezug genommen wird). Beachten Sie aber bitte, dass in der elektronischen Version, die Sie von einer der Webseiten zum Buch *www.hochschule-trier.de/puva6* (puva: **p**arallele **u**nd **v**erteilte **A**nwendungen) und *plus.hanser-fachbuch.de* beziehen können, die Klassen und Schnittstellen kapitelweise in unterschiedliche Packages gruppiert wurden (chapter2, chapter3 usw.). Alle Java-Programme wurden mit einem Java-Compiler der Version 14 übersetzt und ausprobiert.

Von den soeben bereits erwähnten Webseiten *www.hochschule-trier.de/puva6* und *plus.hanser-fachbuch.de* können Sie nicht nur alle Programme des Buchs in Form einer ZIP-Datei herunterladen. Auch nachträglich entdeckte Fehler werde ich mitsamt ihren Richtigstellungen und den Namen der Entdeckenden wie für die vorhergehende Auflage auf dieser Seite veröffentlichen. Ich habe zwar für diese Auflage alle entdeckten Fehler korrigiert, aber es ist nicht auszuschließen, dass bisher unentdeckte alte Fehler noch zutage treten werden,

und ich bin mir sehr sicher, dass ich bei der Überarbeitung der alten Texte und dem Schreiben der neuen Texte unabsichtlich neue Fehler eingebaut habe. Ich bin allen Leserinnen und Lesern dankbar für alle Arten von Fehlermeldungen, sowohl für die Meldung gravierender Fehler als auch einfacher Komma-, Tipp- oder Formatierungsfehler. Kommentare, Verbesserungsvorschläge und weitere Programmbeispiele, die Sie mir gerne senden können, werde ich ebenfalls auf dieser Webseite veröffentlichen, sofern sie mir für einen größeren Kreis interessant erscheinen.

Meinen Wunsch, geschlechtsneutrale Formulierungen zu verwenden, habe ich so umgesetzt, dass ich an manchen Stellen die männliche und weibliche Form angebe, an anderen Stellen nur die männliche oder nur die weibliche Form. Ich hoffe, dass sich dadurch Lesende beiderlei Geschlechts in gleicher Weise angesprochen fühlen.

Sollten Sie tiefer in die Thematik dieses Buches einsteigen wollen, dann empfehle ich Ihnen, das Modul „Fortgeschrittene Programmiertechniken (FOPT)" im Rahmen des Informatik-Fernstudiums an der Hochschule Trier zu belegen. Hier können Sie zu den Themen dieses Buches Einsendeaufgaben bearbeiten, an zusätzlichen Tutorien (per Videokonferenz) teilnehmen sowie ein einwöchiges Präsenzpraktikum absolvieren. Nähere Informationen hierzu, insbesondere über die Voraussetzungen für die Belegung, über die Kosten sowie über die weiteren Module des Fernstudiums, finden Sie auf den Webseiten des Fachbereichs Informatik der Hochschule Trier (*www.hochschule-trier.de/informatik*).

Diese sechste Auflage hätte ohne die Hilfe der nachfolgend genannten Personen nicht bzw. nicht in dieser Form erscheinen können. Ich bedanke mich daher sehr gerne

- bei den für dieses Buch verantwortlichen Mitarbeiterinnen des Hanser-Verlags, Natalia Silakova und Christina Kubiak, für das Ergreifen der Initiative zur sechsten Auflage dieses Buchs, für die Möglichkeit der Erweiterung des Umfangs des Buchs um ca. 20 %, für die Umsetzung meines Vorschlags eines Semaphors als Titelbild sowie für die jederzeit schnelle Klärung aller meiner Fragen,
- bei Daniel Aggintus, Andreas Daum, Fabian Gibert, Robin Grell, Yanik Kaypinger, Marc Kutscher, Frank Seeger, Gunnar Sperveslage, Timo Vollmert und Thomas Zehrer für ihre Hinweise auf entdeckte Fehler und ihre Verbesserungsvorschläge, die alle auf der Webseite *www.hochschule-trier.de/puva5* veröffentlicht und in dieser sechsten Auflage berücksichtigt wurden,
- bei Stefan Bodenschatz, der mit mir zusammen an der Hochschule Trier eine Lehrveranstaltung zum Thema Cloud Computing aufgebaut und mehrfach durchgeführt hat, für seine zahlreichen Verbesserungsvorschläge zu einer früheren Fassung von Kapitel 9,
- und schließlich bei meiner Frau Ingrid für die gewährte Zeit zur Überarbeitung des Buchs.

Über positive und negative Bemerkungen zu diesem Buch, Hinweise auf Fehler und Verbesserungsvorschläge würde ich mich auch dieses Mal wieder freuen. Senden Sie Ihre Kommentare bitte in Form einer elektronischen Post an *oechsle@hochschule-trier.de*.

Konz-Oberemmel, im Februar 2022

Rainer Oechsle

Inhaltsverzeichnis

Vorwort zur 6. Auflage .... **V**

1 Einleitung **1**

1.1 Parallelität, Nebenläufigkeit und Verteilung 1

1.2 Programme, Prozesse und Threads 2

2 Grundlegende Synchronisationskonzepte in Java **6**

2.1 Erzeugung und Start von Java-Threads 6

2.1.1 Ableiten der Klasse Thread 6

2.1.2 Implementieren der Schnittstelle Runnable 8

2.1.3 Einige Beispiele 11

2.1.4 Parallele Abläufe 18

2.2 Probleme beim Zugriff auf gemeinsam genutzte Objekte 19

2.2.1 Erster Lösungsversuch 22

2.2.2 Zweiter Lösungsversuch 23

2.3 synchronized und volatile 25

2.3.1 Synchronized-Methoden 25

2.3.2 Synchronized-Blöcke 27

2.3.3 Wirkung von synchronized 28

2.3.4 Notwendigkeit von synchronized 30

2.3.5 volatile 31

2.3.6 Regel für die Nutzung von synchronized 31

2.4 Ende von Java-Threads 33

2.4.1 Asynchrone Beauftragung mit Abfragen der Ergebnisse 34

2.4.2 Zwangsweises Beenden von Threads 40

2.4.3 Asynchrone Beauftragung mit befristetem Warten 45

2.4.4 Asynchrone Beauftragung mit Rückruf (Callback) 47
2.4.5 Asynchrone Beauftragung mit Rekursion 50
2.5 wait und notify 54
2.5.1 Erster Lösungsversuch 55
2.5.2 Zweiter Lösungsversuch 55
2.5.3 Dritter Lösungsversuch 57
2.5.4 Korrekte und effiziente Lösung mit wait und notify 58
2.6 NotifyAll 67
2.6.1 Erzeuger-Verbraucher-Problem mit wait und notify 67
2.6.2 Erzeuger-Verbraucher-Problem mit wait und notifyAll 71
2.6.3 Faires Parkhaus mit wait und notifyAll 74
2.7 Prioritäten von Threads 76
2.8 Thread-Gruppen 84
2.9 Vordergrund- und Hintergrund-Threads 88
2.10 Weitere „gute“ und „schlechte“Thread-Methoden 90
2.11 Thread-lokale Daten 91
2.12 Zusammenfassung 94

3 Fortgeschrittene Synchronisationskonzepte in Java 99

3.1 Semaphore 100
3.1.1 Einfache Semaphore 100
3.1.2 Einfache Semaphore für den gegenseitigen Ausschluss 101
3.1.3 Einfache Semaphore zur Herstellung vorgegebener Ausführungsreihenfolgen 103
3.1.4 Additive Semaphore 106
3.1.5 Semaphorgruppen 109
3.2 Message Queues 112
3.2.1 Verallgemeinerung des Erzeuger-Verbraucher-Problems 112
3.2.2 Übertragung des erweiterten Erzeuger-Verbraucher-Problems auf Message Queues 114
3.3 Pipes 117
3.4 Philosophen-Problem 120
3.4.1 Lösung mit synchronized – wait – notifyAll 121
3.4.2 Naive Lösung mit einfachen Semaphoren 124

3.4.3 Einschränkende Lösung mit gegenseitigem Ausschluss 125
3.4.4 Gute Lösung mit einfachen Semaphoren 126
3.4.5 Lösung mit Semaphorgruppen 130
3.5 Leser-Schreiber-Problem 132
3.5.1 Lösung mit synchronized – wait – notifyAll 133
3.5.2 Lösung mit additiven Semaphoren 136
3.6 Schablonen zur Nutzung der Synchronisationsprimitive und Konsistenzbetrachtungen 138
3.7 Concurrent-Klassenbibliothek aus Java 5 142
3.7.1 Executors 143
3.7.2 Locks und Conditions 149
3.7.3 Atomic-Klassen 157
3.7.4 Synchronisationsklassen 161
3.7.5 Queues 164
3.8 Das Fork-Join-Framework von Java 7 165
3.8.1 Grenzen von ThreadPoolExecutor 165
3.8.2 ForkJoinPool und RecursiveTask 167
3.8.3 Beispiel zur Nutzung des Fork-Join-Frameworks 169
3.9 Das Data-Streaming-Framework von Java 8 171
3.9.1 Einleitendes Beispiel 172
3.9.2 Sequenzielles Data-Streaming 174
3.9.3 Paralleles Data-Streaming 177
3.10 Die CompletableFutures von Java 8 179
3.11 Ursachen für Verklemmungen 185
3.11.1 Beispiele für Verklemmungen mit synchronized 186
3.11.2 Beispiele für Verklemmungen mit Semaphoren 190
3.11.3 Bedingungen für das Eintreten von Verklemmungen 191
3.12 Vermeidung von Verklemmungen 192
3.12.1 Anforderung von Betriebsmitteln „auf einen Schlag“ 195
3.12.2 Anforderung von Betriebsmitteln gemäß einer vorgegebenen Ordnung 196
3.12.3 Weitere Verfahren 197
3.13 Zusammenfassung 199

4 Parallelität und grafische Benutzeroberflächen ... 200

4.1 Einführung in die Programmierung grafischer Benutzeroberflächen mit JavaFX ... 201

4.1.1 Allgemeines zu grafischen Benutzeroberflächen ... 201

4.1.2 Erstes JavaFX-Beispiel ... 202

4.1.3 Ereignisbehandlung ... 203

4.2 Properties, Bindings und JavaFX-Collections ... 207

4.2.1 Properties ... 207

4.2.2 Bindings ... 210

4.2.3 JavaFX-Collections ... 211

4.3 Elemente von JavaFX ... 212

4.3.1 Container ... 212

4.3.2 Interaktionselemente ... 215

4.3.3 Grafikprogrammierung ... 217

4.3.4 Weitere Funktionen von JavaFX ... 223

4.4 MVP ... 224

4.4.1 Prinzip von MVP ... 224

4.4.2 Beispiel zu MVP ... 226

4.5 Threads und JavaFX ... 232

4.5.1 Threads für JavaFX ... 232

4.5.2 Länger dauernde Ereignisbehandlungen ... 234

4.5.3 Beispiel Stoppuhr ... 239

4.5.4 Tasks und Services in JavaFX ... 244

4.6 Zusammenfassung ... 253

5 Verteilte Anwendungen mit Sockets ... 254

5.1 Einführung in das Themengebiet der Rechnernetze ... 255

5.1.1 Schichtenmodell ... 255

5.1.2 IP-Adressen und DNS-Namen ... 259

5.1.3 Das Transportprotokoll UDP ... 259

5.1.4 Das Transportprotokoll TCP ... 261

5.2 Socket-Schnittstelle ... 262

5.2.1 Socket-Schnittstelle zu UDP ... 262

5.2.2 Socket-Schnittstelle zu TCP 263
5.2.3 Socket-Schnittstelle für Java 266
5.3 Kommunikation über UDP mit Java-Sockets 267
5.4 Multicast-Kommunikation mit Java-Sockets 276
5.5 Kommunikation über TCP mit Java-Sockets 280
5.6 Sequenzielle und parallele Server 292
5.6.1 TCP-Server mit dynamischer Parallelität 293
5.6.2 TCP-Server mit statischer Parallelität 297
5.6.3 Sequenzieller, „verzahnt" arbeitender TCP-Server 302
5.6.4 Horizontale Skalierung mit Lastbalancierung 305
5.7 Verschlüsselte Kommunikation über TLS 306
5.8 Zusammenfassung 310

6 Verteilte Anwendungen mit RMI 311
6.1 Prinzip von RMI 311
6.2 Einführendes RMI-Beispiel 314
6.2.1 Basisprogramm 314
6.2.2 RMI-Client mit grafischer Benutzeroberfläche 318
6.2.3 RMI-Registry 323
6.3 Parallelität bei RMI-Methodenaufrufen 327
6.4 Wertübergabe für Parameter und Rückgabewerte 331
6.4.1 Serialisierung und Deserialisierung von Objekten 332
6.4.2 Serialisierung und Deserialisierung bei RMI 336
6.5 Referenzübergabe für Parameter und Rückgabewerte 341
6.6 Transformation lokaler in verteilte Anwendungen 356
6.6.1 Rechnergrenzen überschreitende Synchronisation mit RMI 357
6.6.2 Asynchrone Kommunikation mit RMI 359
6.6.3 Verteilte MVP-Anwendungen mit RMI 360
6.7 Dynamisches Umschalten zwischen Wert- und Referenzübergabe – Migration von Objekten 361
6.7.1 Das Exportieren und „Unexportieren" von Objekten 361
6.7.2 Migration von Objekten 364
6.7.3 Eintrag eines Nicht-Stub-Objekts in die RMI-Registry 371

6.8 Realisierung von Stubs und Skeletons 372
6.8.1 Realisierung von Skeletons 373
6.8.2 Realisierung von Stubs 374
6.9 Verschiedenes 376
6.10 Zusammenfassung 377

7 Verteilte Anwendungenmit indirekter Kommunikation 378
7.1 Prinzip der indirekten Kommunikation 379
7.2 Kommunikationsmodelle 381
7.2.1 Kommunikationsmodell Queue 381
7.2.2 Kommunikationsmodell Topic 382
7.3 Nutzung der indirekten Kommunikation in Java 383
7.4 Unidirektionale Kommunikation 385
7.5 Bidirektionale Kommunikation mithilfe eines Rückkanals 391
7.6 Empfangsbestätigungen 396
7.7 Transaktionen 397
7.8 Verschiedenes 398

8 Webbasierte Anwendungen mit Servlets und JSF 401
8.1 HTTP und HTML 402
8.1.1 GET 403
8.1.2 Formulare in HTML 406
8.1.3 POST 408
8.1.4 Format von HTTP-Anfragen und -Antworten 409
8.2 Einführende Servlet-Beispiele 409
8.2.1 Allgemeine Vorgehensweise 409
8.2.2 Erstes Servlet-Beispiel 411
8.2.3 Zugriff auf Formulardaten 413
8.2.4 Zugriff auf die Daten der HTTP-Anfrage und -Antwort 414
8.3 Parallelität bei Servlets 416
8.3.1 Demonstration der Parallelität von Servlets 416
8.3.2 Paralleler Zugriff auf Daten 418
8.3.3 Anwendungsglobale Daten 421

8.4 Sessions und Cookies ... 425
8.4.1 Sessions ... 425
8.4.2 Realisierung von Sessions mit Cookies ... 430
8.4.3 Direkter Zugriff auf Cookies ... 433
8.4.4 Servlets mit länger dauernden Aufträgen ... 434
8.5 Asynchrone Servlets ... 439
8.6 Filter ... 444
8.7 Übertragung von Dateien mit Servlets ... 445
8.7.1 Herunterladen von Dateien ... 445
8.7.2 Hochladen von Dateien ... 447
8.8 JSF (Java Server Faces) ... 450
8.8.1 Einführendes Beispiel ... 451
8.8.2 Managed Beans und deren Scopes ... 457
8.8.3 MVP-Prinzip mit JSF ... 461
8.8.4 AJAX mit JSF ... 463
8.9 RESTful WebServices ... 467
8.9.1 Definition von RESTful WebServices ... 468
8.9.2 JSON ... 469
8.9.3 Beispiel ... 471
8.10 WebSockets ... 476
8.11 Zusammenfassung ... 480

9 Verteilte Anwendungen in der Cloud ... 483
9.1 Cloud Computing ... 483
9.2 AWS (Amazon Web Services) ... 487
9.2.1 AWS-Infrastruktur ... 487
9.2.2 AWS-Dienste ... 488
9.2.3 Nutzung der AWS-Dienste ... 492
9.3 Nutzung der AWS-Dienste von außerhalb der Cloud ... 494
9.3.1 Nutzung des AWS-Dienstes S3 ... 496
9.3.2 Nutzung des AWS-Dienstes DynamoDB ... 501
9.3.3 Nutzung des AWS-Dienstes Translate ... 507
9.4 Nutzung von EC2 als Server ... 512

9.5 Nutzung von ECS als Server 518
9.5.1 Isolationsstufen 518
9.5.2 Linux-Grundlagen für die Realisierung von Containern 520
9.5.3 Docker 523
9.5.4 ECS 528
9.6 Nutzung von Lambda als Server 529
9.6.1 Idee der zu entwickelnden Anwendung 531
9.6.2 Lambda-Funktion 532
9.6.3 API Gateway 537
9.6.4 Kommandozeilenbasierter Client 540
9.6.5 Java-basierter Client mit grafischer Benutzeroberfläche 542

Literatur **553**

Index **555**

1 Einleitung

Computer-Nutzer dürften mit großer Wahrscheinlichkeit sowohl mit parallelen Abläufen auf ihrem eigenen Rechner als auch verteilten Anwendungen vertraut sein. So ist jeder Benutzer eines PC heutzutage gewohnt, dass z. B. gleichzeitig eine größere Video-Datei kopiert, ein Musikstück aus einer MP3-Datei abgespielt, ein Java-Programm übersetzt und ein Dokument in einem Editor oder Textverarbeitungsprogramm bearbeitet werden kann. Aufgrund der Tatsache, dass die Mehrzahl der genutzten Computer an das Internet angeschlossen ist und fast alle Menschen ein Handy nutzen, sind heute auch nahezu alle den Umgang mit verteilten Anwendungen wie der elektronischen Post, Messenger-Diensten oder dem World Wide Web gewohnt.

Dieses Buch handelt allerdings nicht von der Benutzung, sondern von der Entwicklung paralleler und verteilter Anwendungen mit Java. In diesem ersten einleitenden Kapitel werden zunächst einige wichtige Begriffe wie Parallelität, Nebenläufigkeit, Verteilung, Prozesse und Threads geklärt.

1.1 Parallelität, Nebenläufigkeit und Verteilung

Wenn mehrere Vorgänge gleichzeitig auf einem Rechner ablaufen, so sprechen wir von Parallelität oder Nebenläufigkeit (engl. concurrency). Diese Vorgänge können dabei echt gleichzeitig oder nur scheinbar gleichzeitig ablaufen: Wenn ein Rechner mehrere Prozessoren bzw. einen Mehrkernprozessor (Multicore-Prozessor) besitzt, dann ist echte Gleichzeitigkeit möglich. Man spricht in diesem Fall auch von echter Parallelität. Besitzt der Rechner aber nur einen einzigen Prozessor mit einem einzigen Kern, so wird die Gleichzeitigkeit der Abläufe nur vorgetäuscht, indem in sehr hoher Frequenz von einem Vorgang auf den nächsten umgeschaltet wird. Man spricht in diesem Fall von Pseudoparallelität oder Nebenläufigkeit. Die Begriffe Parallelität und Nebenläufigkeit werden in der Literatur nicht einheitlich verwendet: Einige Autoren verwenden den Begriff Nebenläufigkeit als Oberbegriff für echte Parallelität und Pseudoparallelität, für andere Autoren sind Nebenläufigkeit und Pseudoparallelität Synonyme. In diesem Buch wird der Einfachheit halber nicht zwischen Nebenläufigkeit und Parallelität unterschieden; mit beiden Begriffen sollen sowohl die echte als auch die Pseudoparallelität gemeint sein.

Wenn das gleichzeitige Ablaufen von Vorgängen auf mehreren Rechnern betrachtet wird, wobei die Rechner über ein Rechnernetz gekoppelt sind und darüber miteinander kommunizieren, spricht man von Verteilung (verteilte Systeme, verteilte Anwendungen).

Wir unterscheiden also, ob die Vorgänge auf einem Rechner oder auf mehreren Rechnern gleichzeitig ablaufen; im ersten Fall sprechen wir von Parallelität, im anderen Fall von Verteilung. Die Mehrzahl der Leserinnen und Leser dürfte vermutlich mit dieser Unterscheidung zufrieden sein. In manchen Fällen ist es aber gar nicht so einfach, zu entscheiden, ob ein gegebenes System einen einzigen Rechner oder eine Vielzahl von Rechnern darstellt. Betrachten Sie z.B. ein System zur Steuerung von Maschinen, wobei dieses System in einem Schaltschrank untergebracht ist, in dem sich mehrere Einschübe mit Prozessoren befinden. Handelt es sich hier um einen oder um mehrere kommunizierende Rechner? Zur Klärung dieser Frage wollen wir uns hier an die allgemein übliche Unterscheidung zwischen eng und lose gekoppelten Systemen halten: Ein eng gekoppeltes System ist ein Rechnersystem bestehend aus mehreren gekoppelten Prozessoren, wobei diese auf einen gemeinsamen Speicher (Hauptspeicher) zugreifen können. Ein lose gekoppeltes System (auch verteiltes System genannt) besteht aus mehreren gekoppelten Prozessoren ohne gemeinsamen Speicher (Hauptspeicher), die über ein Kommunikationssystem Nachrichten austauschen. Ein eng gekoppeltes System sehen wir als einen einzigen Rechner, während wir ein lose gekoppeltes System als einen Verbund mehrerer Rechner betrachten.

Parallelität und Verteilung schließen sich nicht gegenseitig aus, sondern hängen im Gegenteil eng miteinander zusammen: In einem verteilten System laufen auf jedem einzelnen Rechner mehrere Vorgänge parallel (echt parallel oder pseudoparallel) ab. Wie auch in diesem Buch noch ausführlich diskutiert wird, arbeitet ein Server im Rahmen eines Client-Server-Szenarios häufig parallel, um mehrere Clients gleichzeitig zu bedienen. Außerdem können verteilte Anwendungen, die für den Ablauf auf unterschiedlichen Rechnern vorgesehen sind, im Spezialfall auf einem einzigen Rechner parallel ausgeführt werden.

Sowohl Parallelität als auch Verteilung werden durch Hard- und Software realisiert. Bei der Software spielt das Betriebssystem eine entscheidende Rolle. Das Betriebssystem verteilt u.a. die auf einem Rechner gleichzeitig möglichen Abläufe auf die vorhandenen Prozessoren bzw. die vorhandenen Kerne des Rechners. Auf diese Art vervielfacht also das Betriebssystem die Anzahl der vorhandenen Prozessoren bzw. der vorhandenen Kerne virtuell. Diese Virtualisierung ist eines der wichtigen Prinzipien von Betriebssystemen, die auch für andere Ressourcen realisiert wird. So wird z.B. durch das Konzept des virtuellen Speichers ein größerer Hauptspeicher vorgegaukelt als tatsächlich vorhanden. Erreicht wird dies, indem immer die gerade benötigten Daten vom Hintergrundspeicher (Platte) in den Hauptspeicher transferiert werden.

■ 1.2 Programme, Prozesse und Threads

Im Zusammenhang mit Parallelität bzw. Nebenläufigkeit und Verteilung muss zwischen den Begriffen Programm, Prozess und Thread (Ausführungsfaden) unterschieden werden. Da es einen engen Zusammenhang zu den Themen Betriebssysteme, Rechner und verteilte Systeme gibt, sollen alle diese Begriffe anhand einer Metapher verdeutlicht werden:

- Ein Programm entspricht einem Rezept in einem Kochbuch. Es ist statisch. Es hat keine Wirkung, solange es nicht ausgeführt wird. Dass man von einem Rezept nicht satt wird, ist hinlänglich bekannt.
- Einen Prozess kann man sich vorstellen als eine Küche und einen Thread als einen Koch. Ein Koch kann nur in einer Küche existieren, aber nie außerhalb davon. Umgekehrt muss sich in einer Küche immer mindestens ein Koch befinden. Alle Köche gehen streng nach Rezepten vor, wobei unterschiedliche Köche nach demselben oder nach unterschiedlichen Rezepten kochen können. Jede Küche hat ihre eigenen Pfannen, Schüsseln, Herde, Waschbecken, Messer, Gewürze, Lebensmittel usw. Köche in unterschiedlichen Küchen können sich gegenseitig nicht in die Quere kommen, wohl aber die Köche in einer Küche. Diese müssen den Zugriff auf die Materialien und Geräte der Küche koordinieren.
- Ein Rechner ist in dieser Metapher ein Haus, in dem sich mehrere Küchen befinden.
- Ein Betriebssystem lässt sich mit einem Hausmeister eines Hauses vergleichen, der dafür sorgt, dass alles funktioniert (z. B. dass immer Strom für den Herd da ist). Der Hausmeister übernimmt u. a. auch die Rolle eines Boten zwischen den Küchen, um Gegenstände oder Informationen zwischen den Küchen auszutauschen. Auch kann er eine Küche durch einen Anbau vergrößern, wenn eine Küche zu klein geworden ist.

Ein verteiltes System besteht entsprechend aus mehreren solcher Häuser mit Küchen, wobei die Hausmeister der einzelnen Häuser z. B. über Telefon oder über hin- und herlaufende Boten untereinander kommunizieren können. Somit können Köche, die in unterschiedlichen Häusern arbeiten, Gegenstände oder Informationen austauschen, indem sie ihre jeweiligen Hausmeister damit beauftragen.

Diese Begriffe und ihre Beziehung sind in Bild 1.1 zusammenfassend dargestellt.

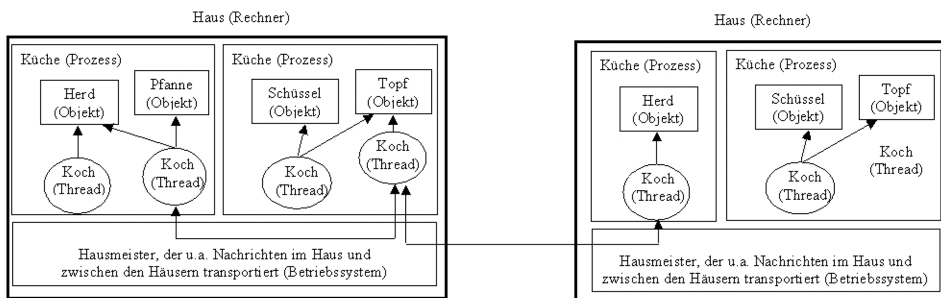


Bild 1.1 Häuser, Küchen, Köche und Hausmeister als Metapher für Rechner, Prozesse, Threads und Betriebssysteme

Am Beispiel der Programmiersprache Java und der Ausführung von Java-Programmen lässt sich diese Metapher nun auf die Welt der Informatik übertragen:

- Ein Programm (Kochrezept) ist in einer Datei abgelegt: als Quelltext in einer oder mehreren Java-Dateien und als übersetztes Programm (Byte-Code) in einer oder mehreren Class-Dateien.
- Zum Ausführen eines Programms mithilfe des Kommandos java wird eine JVM (Java Virtual Machine) gestartet. Bei jedem Erteilen des Java-Kommandos wird ein neuer Prozess

(Küche) erzeugt. Ein Prozess stellt im Wesentlichen einen Adressraum für den Programmcode und die Daten dar. Der Programmcode, der sich in einer oder mehreren Dateien befindet, wird in den Adressraum des Prozesses geladen. Es ist möglich, mehrere JVMs zu starten, sodass die entsprechenden Prozesse alle gleichzeitig existieren, wobei jeder Prozess seinen eigenen Adressraum besitzt.

- Jeder Prozess und damit auch jede JVM hat als Aktivitätsträger mindestens einen Thread (Koch). Neben den sogenannten Hintergrund-Threads, die z. B. für die Speicherbereinigung (Garbage Collection) zuständig sind, gibt es einen Thread, der die Main-Methode der im Java-Kommando angegebenen Klasse ausführt. Dieser Thread kann durch Aufruf entsprechender Methoden weitere Threads starten. Die Threads innerhalb desselben Prozesses können auf dieselben Objekte (Gegenstände in einer Küche wie Herd, Pfannen, Töpfe, Schüsseln usw.) lesend und schreibend zugreifen, nicht aber auf die Objekte, die sich in anderen Prozessen befinden.
- Das Betriebssystem (Hausmeister) verwaltet die Adressräume der Prozesse und teilt den Threads abwechselnd die vorhandenen Prozessoren bzw. den vorhandenen Kernen zu. Das Betriebssystem garantiert gemeinsam mit der Hardware, dass ein Prozess keinen Zugriff auf den Adressraum eines anderen Prozesses auf demselben Rechner besitzt. Damit sind die Prozesse eines Rechners voneinander isoliert. Zwei Prozesse auf unterschiedlichen Rechnern sind ebenfalls voneinander isoliert, da ein verteiltes System laut Definition ein lose gekoppeltes System ist, das keinen gemeinsamen Speicher hat.

Die Isolierung der Prozessadressräume kann durch Inanspruchnahme von Leistungen des Betriebssystems über Systemaufrufe in kontrollierter Weise durchbrochen werden. Damit können die Prozesse miteinander interagieren. Betriebssysteme bieten Dienste zur Synchronisation und Kommunikation zwischen Prozessen sowie zur gemeinsamen Nutzung von speziellen Speicherbereichen an. Ferner stellen Betriebssysteme Funktionen bereit, um über ein Rechnernetz Daten an Prozesse anderer Rechner zu senden oder eingetroffene Nachrichten entgegenzunehmen. Durch Systemaufrufe kann das Betriebssystem auch beauftragt werden, den Adressraum eines Prozesses zu vergrößern.

In diesem Buch geht es um zwei wesentliche Aspekte:

- Parallelität innerhalb eines Prozesses: Die Leserinnen und Leser sollen das Konzept der Parallelität innerhalb eines Prozesses aus Sicht einer Programmiererin bzw. eines Programmierers mit Java-Threads beherrschen lernen. Sie sollen erkennen, welche Probleme entstehen, wenn mehrere Threads auf dieselben Objekte zugreifen und wie diese Probleme gelöst werden können.
- Verteilung: Darüber hinaus zeigt das Buch, wie verteilte Anwendungen mit Java entwickelt werden. Wir unterscheiden dabei eigenständige Client-Server-Anwendungen und webbasierte Anwendungen. Bei eigenständigen Client-Server-Anwendungen entwickeln wir sowohl die Client- als auch die Server-Programme selbst. Client und Server kommunizieren dabei über die Socket-Schnittstelle, über RMI (Remote Method Invocation) oder indirekt über einen Vermittler. Bei webbasierten Anwendungen benutzen wir als Client einen Browser. Die Server-Seite besteht aus einem Web-Server, der durch selbst entwickelte Programme erweitert werden kann. Sowohl bei den eigenständigen Client-Server-Anwendungen als auch bei den webbasierten Anwendungen spielt die Parallelität insbesondere auf Server-Seite eine wichtige Rolle. Immer wichtiger wird die Nutzung von Cloud-Diensten durch verteilte Anwendungen. Auch dieses Thema wird behandelt.

Wir betrachten hier nicht gesondert die Parallelität, Interaktion und Synchronisation von Threads unterschiedlicher Prozesse desselben Rechners. Dies liegt vor allem daran, dass es hierzu keine speziellen Java-Klassen gibt. Dies bedeutet aber keine Einschränkung, denn alle in diesem Buch vorgestellten Kommunikationskonzepte zwischen dem Client- und Server-Prozess einer verteilten Anwendung können auch angewendet werden, wenn sich Client und Server auf demselben Rechner befinden. Das heißt: Bezüglich der Kommunikation zwischen Threads unterschiedlicher Prozesse unterscheiden wir nicht, ob sich die Prozesse auf demselben oder auf unterschiedlichen Rechnern befinden.

Die Synchronisations- und Kommunikationskonzepte, die anhand von Java-Threads innerhalb eines Prozesses vorgestellt werden, gibt es in ähnlicher Weise auch für das Zusammenspiel von Threads unterschiedlicher Prozesse auf einem Rechner. Wie schon erwähnt, gibt es zwar hierfür keine spezielle Java-Schnittstelle, aber die erlernten Konzepte wie Semaphore, Message Queues und Pipes bilden eine gute Grundlage für das Verständnis der Dienste, die ein Betriebssystem wie Linux zur Synchronisation und Kommunikation zwischen unterschiedlichen Prozessen anbietet.

2 Grundlegende Synchronisationskonzepte in Java

In diesem Kapitel geht es um die grundlegenden Synchronisationskonzepte in Java. Diese bestehen im Wesentlichen aus dem Schlüsselwort synchronized sowie den Methoden wait, notify und notifyAll der Klasse Object. Es wird erläutert, welche Wirkung synchronized, wait, notify und notifyAll haben und wie sie eingesetzt werden sollen. Außerdem spielt die Klasse Thread eine zentrale Rolle. Diese Klasse wird benötigt, um Threads zu erzeugen und zu starten.

2.1 Erzeugung und Start von Java-Threads

Wie schon im einleitenden Kapitel erläutert wurde, wird beim Start eines Java-Programms (z. B. mittels des Kommandos java) ein Prozess erzeugt, der u. a. einen Thread enthält, der die Main-Methode der angegebenen Klasse ausführt. Der Programmcode weiterer vom Anwendungsprogrammierer definierter Threads muss sich in Methoden namens *run* befinden:

```
public void run ()
{
    // Code, der in eigenem Thread ausgeführt wird
}
```

Es gibt zwei Möglichkeiten, in welcher Art von Klasse diese Run-Methode definiert wird.

2.1.1 Ableiten der Klasse Thread

Die erste Möglichkeit besteht darin, aus der Klasse *Thread*, die bereits eine leere Run-Methode besitzt, eine neue Klasse abzuleiten und darin die Run-Methode zu überschreiben. Die Klasse Thread ist (wie String) eine Klasse des Package java.lang und kann deshalb ohne Import-Anweisung in jedem Java-Programm verwendet werden. Hat man eine derartige Klasse definiert, so muss noch ein Objekt dieser Klasse erzeugt und dieses Objekt (das ja ein Thread ist, da es von Thread abgeleitet wurde) mit der *Start-Methode* gestartet werden. Das Programm in Listing 2.1 zeigt dies anhand eines Beispiels.

Listing 2.1

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Hallo Welt");
    }
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

An diesem ersten Programmbeispiel mag auf den ersten Blick verwirrend sein, dass in der Klasse MyThread zwar eine Run-Methode definiert wird, dass aber in der Main-Methode eine Methode namens *start* auf das Objekt der Klasse MyThread angewendet wird. Die Methode start ist in der Klasse Thread definiert und wird somit auf die Klasse MyThread vererbt.

```
public class Thread
{
    public void start () {...}
    ...
}
```

Natürlich könnte man statt start auch die Methode run auf das erzeugte Objekt anwenden. Der Benutzer würde keinen Unterschied zwischen den beiden Programmen feststellen können, denn in beiden Fällen wird „Hallo Welt“ ausgegeben. Allerdings ist der Ablauf in beiden Fällen deutlich verschieden: In der Metapher der Küchen und Köche passiert bei dem oben angegebenen Programm Folgendes: Der bereits vorhandene Koch, der nach dem Rezept der Main-Methode kocht, erzeugt einen neuen Koch und erweckt diesen mithilfe der Start-Methode zum Leben. Dieser neue Koch geht nach dem Rezept der entsprechenden Run-Methode vor und gibt „Hallo Welt“ aus. Würde dagegen der Aufruf der Start-Methode durch einen Aufruf der Run-Methode in obigem Programm ersetzt, so wäre dies ein gewöhnlicher Methodenaufruf, wie Sie das aus der bisherigen sequenziellen Programmierung bereits kennen. Die Ausgabe „Hallo Welt“ erfolgt also in diesem Fall durch den Thread, der die Main-Methode ausführt, und nicht durch einen neuen Thread. In der Metapher der Küchen und Köche könnte man einen Methodenaufruf so sehen wie einen Hinweis in einem Kochbuch, in dem in einem Rezept die Anweisung „Hefeteig zubereiten“ (s. Seite 456) steht. Derselbe Koch, der diese Anweisung liest, würde dann auf die Seite 456 blättern, die dort stehenden Anweisungen befolgen und anschließend zum ursprünglichen Rezept zurückkehren.

Dieses kleine, nur wenige Zeilen umfassende Beispielprogramm enthält noch ein weiteres Verständnisproblem für viele Neulinge: Warum muss ein Thread-Objekt (genauer: ein Objekt der aus Thread abgeleiteten Klasse MyThread) mit new erzeugt und warum muss dieses dann noch zusätzlich mit der Start-Methode gestartet werden? Diese Verständnisschwierigkeit kann beseitigt werden, indem man sich klar macht, dass es einen Unterschied zwischen einem Thread-Objekt und dem eigentlichen Thread im Sinne einer selbst-

ständig ablaufenden Aktivität gibt. In unserer Küchen-Köche-Metapher entspricht das Thread-Objekt dem Körper eines Kochs. Ein solcher Körper wird mit new erzeugt. Man kann bei diesem Objekt wie bei anderen Objekten üblich Attribute lesen und verändern, also z. B. Name, Personalnummer und Schuhgröße des Kochs. Dieses Objekt ist aber leblos wie andere Objekte bei der sequenziellen Programmierung auch. Erst durch Aufruf der Start-Methode wird dem Koch der Odem eingehaucht; er beginnt zu atmen und eigenständig gemäß seines Run-Rezepts zu handeln. Dieses Leben des Kochs ist als Objekt im Programm nicht repräsentiert, sondern lediglich der Körper des Kochs. Das Leben des Kochs ist beim Ablauf des Programms durch die vorhandene Aktivität zu erkennen.

Wie im richtigen Leben kann auf ein Thread-Objekt nur ein einziges Mal die Start-Methode angewendet werden. Wenn mehrere gleichartige Threads gestartet werden sollen, dann müssen entsprechend viele Thread-Objekte erzeugt werden (s. Abschnitt 2.1.3).

Ist das Run-Rezept eines Kochs abgehandelt (d. h. ist die Run-Methode zu Ende), so stirbt dieser Koch wieder (der Thread ist als Aktivität nicht mehr vorhanden). Damit muss aber der Körper des Kochs nicht auch verschwinden, sondern dieser kann weiter existieren (falls es noch Referenzen auf das entsprechende Thread-Objekt gibt, ist dieses Objekt noch vorhanden; die verbleibenden Threads können weitere Methoden auf dieses Objekt anwenden).

2.1.2 Implementieren der Schnittstelle Runnable

Falls sich im Rahmen eines größeren Programms die Run-Methode in einer Klasse befinden soll, die bereits aus einer anderen Klasse abgeleitet ist, so kann diese Klasse nicht auch zusätzlich aus Thread abgeleitet werden, da es in Java keine Mehrfachvererbung für Klassen gibt. Als Ersatz für die Mehrfachvererbung existieren in Java Schnittstellen (Interfaces). Es gibt eine Schnittstelle namens *Runnable* (wie die Klasse Thread im Package java.lang), die nur die schon oben vorgestellte Run-Methode enthält.

```
public interface Runnable
{
    public void run();
}
```

Will man nun die Run-Methode in einer nicht aus Thread abgeleiteten Klasse definieren, so sollte diese Klasse stattdessen die Schnittstelle Runnable implementieren. Wenn ein Objekt einer solchen Klasse, die diese Schnittstelle implementiert, dem Thread-Konstruktor als Parameter übergeben wird, dann wird die Run-Methode dieses Objekts nach dem Starten des Threads ausgeführt. Das Programm in Listing 2.2 zeigt diese Vorgehensweise anhand eines Beispiels.

Listing 2.2

```
public class SomethingToRun implements Runnable
{
    public void run()
    {
        System.out.println("Hallo Welt");
    }
```

```
    public static void main(String[] args)
    {
        SomethingToRun runner = new SomethingToRun();
        Thread t = new Thread(runner);
        t.start();
    }
}
```

Voraussetzung für die korrekte Übersetzung beider Beispielprogramme ist, dass die Klasse Thread u. a. folgende Konstruktoren besitzen muss:

```
public class Thread
{
    public Thread() {...}
    public Thread(Runnable r) {...}
   ...
}
```

Der zweite Konstruktor ist offenbar für das zweite Beispiel nötig. Die Nutzung des ersten Konstruktors im ersten Beispiel ist weniger offensichtlich. Da in der Klasse MyThread kein Konstruktor definiert wurde, ist automatisch der folgende Standardkonstruktor vorhanden:

```
public MyThread()
{
    super();
}
```

Der Super-Aufruf bezieht sich auf den parameterlosen Konstruktor der Basisklasse Thread. Einen solchen muss es geben, damit das Programm übersetzbar ist.

Auch für das zweite Beispiel gilt die Unterscheidung zwischen dem Thread-Objekt und dem eigentlichen Thread. Deshalb muss auch hier nach der Erzeugung des Thread-Objekts der eigentliche Thread noch gestartet werden.

Auch wenn wie oben beschrieben ein Thread nur einmal gestartet werden kann, kann hier dennoch dasselbe Runnable-Objekt mehrmals als Parameter an Thread-Konstruktoren übergeben werden. Es wird ja jedes Mal ein neues Thread-Objekt erzeugt, das dann nur einmal gestartet wird. Unter Umständen kann dies aber zu Synchronisationsproblemen führen (s. Abschnitt 2.2 und Abschnitt 2.3).

Seit Java 8 gibt es sogenannte *Lambda-Ausdrücke*. Die Definition der Klasse SomethingToRun in Listing 2.2, welche die Schnittstelle Runnable implementiert, sowie das Erzeugen eines Objekts dieser Klasse kann mit einem Lambda-Ausdruck durch eine einzige Anweisung ersetzt werden:

```
Runnable runner = () -> System.out.println("Hallo Welt");
```

Wenn man das Runnable-Objekt, das man dem Konstruktor von Thread übergibt, in keine lokale Variable speichern möchte, dann kann man die Thread-Erzeugung noch kürzer auch so schreiben:

```
Thread t = new Thread(() -> System.out.println("Hallo Welt"));
```

Und wenn man auf die lokale Thread-Variable t auch noch verzichten möchte, dann schrumpft der Inhalt der Main-Methode auf diese eine Zeile zusammen:

```
new Thread(() -> System.out.println("Hallo Welt")).start();
```

Lambda-Ausdrücke können im Programmcode immer dort angegeben werden, wo ein Objekt vom Typ einer sogenannten *funktionalen Schnittstelle* erwartet wird. Eine funktionale Schnittstelle (Functional Interface) ist eine Schnittstelle mit einer einzigen Methode (genauer müsste man sagen: eine Schnittstelle mit einer einzigen *abstrakten* Methode, denn seit Java 8 können Schnittstellen auch nicht-abstrakte Methoden, sogenannte Default-Methoden, besitzen, für die in der Schnittstelle eine Implementierung angegeben ist). Die Schnittstelle Runnable ist ganz offensichtlich eine funktionale Schnittstelle. Also kann auf der rechten Seite einer Zuweisung an eine Runnable-Variable (Runnable runner = ...) oder als Parameterwert eines Thread-Konstruktors mit Runnable-Parameter ein Lambda-Ausdruck eingesetzt werden.

Allgemein hat ein Lambda-Ausdruck folgende Form:

```
Parameterliste -> Code
```

Der Name der implementierten Methode der Schnittstelle muss (und darf auch) bei einem Lambda-Ausdruck nicht angegeben werden; der Typ des Lambda-Ausdrucks ist nämlich eine funktionale Schnittstelle mit einer einzigen abstrakten Methode, und genau diese Methode wird implementiert. Für die Parameter müssen Bezeichner und optional der jeweilige Typ angegeben werden. Sie können im Code-Teil verwendet werden. Betrachten wir dazu z. B. folgende funktionale Schnittstelle I1:

```
public interface I1
{
    public void m(String s, boolean b);
}
```

Nun könnte man beispielsweise schreiben:

```
I1 i11 = (String s, boolean b) -> System.out.println(s + ", " + b);
```

Oder auch kürzer durch Weglassen der Parametertypen, die sich wie der Methodenname eindeutig aus der funktionalen Schnittstelle I1 herleiten lassen:

```
I1 i12 = (s, b) -> System.out.println(s + ", " + b);
```

Mischformen (also ein Parameter mit Typangabe und ein anderer Parameter ohne Typangabe im selben Lambda-Ausdruck) sind nicht möglich.

Da die Methode run der Schnittstelle Runnable parameterlos ist, musste im obigen Thread-Beispiel der Lambda-Ausdruck mit einer leeren Klammer beginnen. Wenn die zu implementierende Methode genau einen Parameter besitzt, dann können im Lambda-Ausdruck die Klammern um den Parameter weggelassen werden, wenn man auch auf die Angabe des Typs verzichtet.

Der Codeteil bestand in den bisherigen Beispielen immer aus genau einer Anweisung. Im Allgemeinen können es mehrere Anweisungen sein, die dann aber als Java-Block in geschweiften Klammern zusammengefasst sein müssen:

```
I1 i13 = (s, b) -> {System.out.println(s); System.out.println(b);};
```

Index

Symbole

@ApplicationScoped 457
@Consumes 471
@ConversationScoped 458
@DELETE 471
@Dependent 458
@GET 471
@Managed Bean 452
@Named 453
@OnClose 478
@OnError 478
@OnMessage 478
@OnOpen 478
@Path 471
@PathParam 472
@POST 471
@Produces 471
@PUT 471
@RequestScoped 458
@ServerEndpoint 478
@SessionScoped 458
@ViewScoped 458
@WebFilter 444
@WebListener 424
@WebServlet 410
@XHTML 450

A

Abort 397
Abstract Window Toolkit 201
accept 176, 281
Access Key 493
acquire 101, 161
ActionEvent 206
activeCount 86
activeGroupCount 86
ActiveMQ Artemis 385
add 164, 203
addCookie 434
Adressenabbildung 425
AJAX 463
aktive Klasse 97
aktives Objekt 97
aktives Warten 24, 33, 55, 435
Alert 223
allMatch 177
allOf 184
AlreadyBoundException 323
Amazon Machine Image 512
Amazon Web Services 399, 485
AMI 512
AnchorPane 214
Animation 223
Annotation 422
Anwendungsprotokoll 272
Anwendungsschicht 258
Anycast 381
anyMatch 177
anyOf 184
API Gateway 530, 537
API Key 539
API-Schlüssel 539
Application 202
ApplicationContext 422
Application Layer 258
Application Load Balancer 490
applyToEitherAsync 184
Arc 218
Architekturmuster 224
ArrayBlockingQueue 164
ASCII-Protokoll 285, 290, 403
Asynchronous JavaScript And XML 463
atomar 30, 157
Atomarität 30, 397
Audio-Video-Konferenz 258, 260
Auftragnehmer 262
Auskunftsdienst 313
AutoCloseable 271
Auto Scaling 490
Auto Scaling Group 490
Availability 497
Availability Zone 487
average 174, 177

await 151, 152, 162
awaitTermination 146
AWS 399
AWT 201
AZ 487
Azure 485

B

Beobachter 207
Betriebsmittel 186
Betriebsmittelgraph 191
Betriebsmitteltyp 193
Betriebsmittelverwalter 193
Betriebssystem 3, 4
Big Data 171
bind 323
Binding 210
– bidirektional 211
– unidirektional 210
Bitübertragungsschicht 257
BlockingQueue 146, 164
BorderPane 213
Broker 379, 390
Bucket 496
BufferedInputStream 284
BufferedOutputStream 284
BufferedReader 284, 286
BufferedWriter 284, 285
Bühne 202
busy waiting 24
Button 201, 205, 215
ByteArrayInputStream 283
ByteArrayOutputStream 283

C

call 144, 246
Callable 144
Callback 47, 341
Call by Reference 330, 341
Call by Value 330, 331
cancel 144, 246
Cascading Style Sheets 223
CDN 487
Cgroup 520
changed 209
ChangeListener 216
Channel 302
CharacterArrayReader 283
CharacterArrayWriter 283
CheckBox 201, 215
ChoiceBox 216
Choice Button 215
ChoiceDialog 223
Circle 217
CLI 493
Client 262
Client-Server-Anwendung 262, 263
Clipping 222
close 268, 281
Closeable 271
Cloud Computing 483
Cloud Front 492
collect 177
Collection 211
Color 217
ColorPicker 216
ComboBox 216
Command Button 215
Command Line Interface 493
Commit 397
CommonPool 181
Community Cloud 485
Comparable 165
Comparator 165
CompletableFutures 179
complete 179
completeExceptionally 179
concurrency 1
Condition 151
connect 269
ConnectionFactory 383
Consumer 67, 176
Container 201, 523
Control 215
Control Group 520
Cookie 430
Cookie (Klasse) 433
count 177
countDown 162
CountDownLatch 162
createRegistry 325
createServerSocket 376
createSocket 376
CSS 223
CubicCurve 218
currentThread 80
currentTimeMillis 77
CyclicBarrier 162

D

Daemon 85
Daemon Threads 89
Datagramm 260
datagrammorientiert 260
Datagrammverlust 260
DatagramPacket 266, 267
DatagramSocket 266, 267, 268
DataInputStream 284
Data Link Layer 257
DataOutputStream 283

Data-Streaming 172
Data Warehouse 491
Datenstrom 171
datenstromorientiert 261, 285
datenstromorientierte Kommunikation 117
DatePicker 216
Decoupling 399
Dedicated Host 514
Dekomprimieren 284
Delayed 164
DelayQueue 164
Denial-of-Service 297, 530
deprecated 40, 85, 90
Deserialisierung 332, 470
Destination 384
destroy 410
Diagramm 223
Dialog 223
Diensterbringer 263
Dienstschnittstelle 255
DNS 259
Docker 518, 523
Docker Compose 528
Dockerfile 524
DockerHub 523, 528
Docker Registry 523
Document Object Model 463
doGet 410
DOM 463
Domain Name System 259
doPost 410
DoubleStream 174
down 101
Downcall 207
Drag and Drop 223
drahtloses Funknetz 257
dumpStack 90
Durability 496
DurableConsumer 399
DynamicProxy 374
Dynamic Web Project 410
DynamoDB 491, 501

E

EA-intensive Threads 83
EBS 489, 513
EC2 488, 512
Eclipse 409
Eclipse EE 409
ECR 528
ECS 488, 518, 528
Edge Location 487
EFS 489
Egress-Only Gateway 489
Eingabestrom 282
EJB 462
EKS 488
EL 450
Elastic Beanstalk 488
Elastic Block Storage 489, 513
Elastic Compute Cloud 488, 512
Elastic Container Registry 528
Elastic Container Service 488, 518
Elastic Container Service for Kubernetes 488
Elastic File System 489
Elastic IP Address 517
Elastic Load Balancing 488
elastische IP-Adresse 517
ELB 488
elektronische Post 258
Ellipse 217
Enterpreise Java Beans 462
Entschlüsseln 284
Entwurfsmuster 207
Erzeuger 67, 112
Erzeuger-Verbraucher-Prinzip 164
Erzeuger-Verbraucher-Problem 114, 152
Ethernet 257
EventHandler 206
exchange 162
Exchanger 162
execute 143
Executor 143, 181
ExecutorService 145
Exportieren 361
exportObject 362
Expression Language 450

F

FaaS 485, 529
fair 74, 151, 161
Fargate 488, 529
Fern-Methodenaufruf 311
FileChooser 216
FileInputStream 283
FileOutputStream 283
FileReader 283
FileWriter 283
filter 173
Firewall 267
Fließband 171
FlowPane 213
flush 285
Flusskontrolle 261
forEach 177
Fork-Join-Framework 165
ForkJoinPool 165
ForkJoinTask 167
Free Tier 492
fromJson 471
Function 180
Functional Interface 10

Function as a Service 485
funktionale Schnittstelle 10
Future 144, 179
FutureTask 179
FXML 223

G

gegenseitiger Ausschluss 101, 191
generate 174
getAllByName 267
getAttribute 422
getByName 266
getChildren 203
getCompletedTaskCount 147
getCookies 434
getDelay 164
getHeader 415
getHeaderNames 415
getHoldCount 151
getHostAddress 266
getHostName 266
getId 432
getInetAddress 281
getInputStream 282, 285
GET-Kommando 403
getLargestPoolSize 147
getLocalAddress 268
getLocalHost 267
getLocalPort 268, 281
getMethod 415
getName 14
getOutputStream 282, 285
getParameter 413
getParameterNames 414
getParameterValues 414
getPriority 76
getQueueLength 151
getRegistry 325
getRemoteAddr 415
getRemoteHost 415
getServletContext 422
getSession 426
getSoTimeout 268
getState 90
getThreadGroup 85, 86
getWriter 412
Glacier 488, 497
Global Content Delivery Network 487
Google Cloud 485
Google Web Toolkit 477
GridPane 213
GSON 470
GWT 477

H

handle 206
HBox 202, 213
Herunterladen von Dateien 445
Hibernate 462
Hintergrund-Threads 89
Hochladen von Dateien 445, 447
holdsLock 90
HTTP 285, 403
HTTP-Anfrage 403, 409
HTTP-Antwort 404, 409
HttpServlet 410
HttpServletRequest 414
HttpServletResponse 414
HttpSession 426
Hybrid Cloud 485
Hyperlink 215
HyperText Transfer Protocol 403
Hypervisor 512

I

IaaS 484
IAM 493
IBM Cloud 485
idempotent 276
Identity and Access Management 493
IGW 489
IllegalMonitorStateException 58, 71
ImagePattern 217
InetAddress 266
Infrastructure as a Service 484
Infrequent Access 488, 497
init 410
InitialContext 383
InputStream 282
InputStreamReader 285
Instance Store 513
IntelliJ 409
Interaktionselement 201
Internet Gateway 489
Internet Protocol 258
interrupt 42, 149
interrupted 43
InterruptedException 16, 34, 43
Interrupt-Flag 42
IntStream 174
invalidate 428
Invariante 40, 54, 138
InvocationHandler 374
invoke 374
invokeAll 145, 147
invokeAny 145
IP 258
IP-Adresse 258, 259
IPv4 259

IPv6 259
isAlive 33
isCancelled 145, 246
isDaemon 89
isDone 145
isInterrupted 42
isReachable 267
isRunning 249
iterate 175

J

Jakarta EE 383, 411
Jakarta Messaging 383
JavaFX 201
JavaFX Application Thread 233
JavaFX-Collection 211
Java Messaging Service 383
Java Naming and Directory Interface 383
JavaScript Object Notation 336, 468
Java Server Faces 450
Java Server Pages 450
JAX-RS 471
Jersey 471
JMS 383
JMSConsumer 384
JMSContext 384
JMSProducer 384
JNDI 383
join 33, 182
joinGroup 276
JSF 450
JSON 336, 468, 469, 531, 543
JSP 450

K

Kommunikationsprotokoll 255
Komprimieren 284
konsistenter Zustand 54, 138
Konsistenz 54, 138
Konsistenzbedingung 54, 138
Kopplung
– eng 378
– lose 379
kritischer Abschnitt 102
Kubernetes 528
Kunde 262

L

Label 201, 202, 215
Labeled 215
Lambda 488, 529, 532
Lambda-Ausdruck 9
Lastausgleich 305
Lastbalancierung 305, 380, 433
launch 202
Layout 201
leaveGroup 276
Leitungsschicht 257
Leser-Schreiber-Problem 132
Lesesperre 151
limit 176
Line 217
LinearGradient 217
LinkedBlockingQueue 164
list 324
ListView 216
Load Balancer 305, 490
Load Balancing 380
localhost 259
LocateRegistry 325
lock 149
Lock 149, 150
lock-free 161
lockInterruptibly 150
LongStream 174
lookup 316, 323

M

MAC 257
MAC-Adresse 257
MapMessage 398
mapToInt 173
MAX_PRIORITY 76
Medium Access Control 257
Medium-Zugangskontrolle 257
Mehrkernprozessor 1
Message 384
MessageListener 388
Message-Oriented Middleware 360, 381
Message Queue 112, 382
Micro-Service-Architektur 528
Migration 364
migrieren 364
MIN_PRIORITY 76
Model View Controller 224
Model View Presenter 224
Model View ViewModel 224
MOM 360, 381
MouseEvent 219
Multicast 276, 382
Multicast-Adresse 259
MulticastSocket 276
Multicore-Prozessor 1
Mutex 101
mutual exclusion 101
MVC 224
MVP 224, 360
MVVM 224

N

nachrichtenorientierte Kommunikation 117
NACL 489, 512
Namespace 520
Naming 315, 316, 323
nanoTime 77
NAT 425
NAT-Gateway 489
Nebenläufigkeit IX, 1
NetBeans 409
Network Access Control List 489, 512
Network Address Translation 425
Network Layer 258
newCondition 150, 152
New Input/Output 302
newLine 285
NIO-Bibliothek 302
Node 212
NORM_PRIORITY 76
NoSQL 491, 501
Notification 382
notify 54, 58
notifyAll 67
Number 209

O

ObjectInputStream 334
ObjectMessage 398
ObjectOutputStream 334
Objekt-Relationale-Mapper 462
ObservableList 212
ObservableMap 212
ObservableSet 212
ObservableValue 209
Observer 207
offer 164
On Demand 513
OpenSSH 512
ORM 462
OutputStream 282
OutputStreamWriter 285

P

p 101
PaaS 485
Paint 217
Pane 212
parallel 177
Parallelität IX, 1
– dynamisch 293
– echt 1
– pseudoparallel 1
– statisch 292
parallelStream 178
passive Klasse 97
passives Objekt 97
PasswordField 217
peek 176
Philosophen-Problem 120
Physical Layer 257
ping 267
Pipe 117
Platform 236
Platform as a Service 485
Point-to-Point 381
poll 164
Polling 24, 435
Polly 511
Polygon 218
Polyline 218
POP 285
Portnummer 258, 260, 265, 267, 280
POST-Kommando 408
Predicate 173
print 412
println 412
PrintWriter 412
Prioritäten 76
PriorityBlockingQueue 165
Private Cloud 485
Produce-Consume 381
Producer 67
Programm 3
ProgressBar 217
ProgressIndicator 217
Property 208
Protokoll 255
Proxy 425
Prozess 3
Prozessorzuteilungsstrategie 76
Public Cloud 485
Publish-Subscribe 382
punktierte Dezimalnotation 259
put 146, 164
Putty 512

Q

QuadCurve 218
Queue 381, 384
QueueBrowser 399

R

RadialGradient 217
RadioButton 201, 215
RDS 491
read 282
Reader 282
readLine 286

readObject 334
ReadOnlyBooleanProperty 209
ReadOnlyBooleanWrapper 209
ReadOnlyIntegerProperty 209
ReadOnlyIntegerWrapper 209
ReadOnlyProperty 209
ReadOnlyWrapper 209
ReadWriteLock 151
rebind 315, 323
receive 268
rechenintensive Threads 83
Rechner 3, 4
Rectangle 217
RecursiveAction 167
RecursiveTask 167
Redshift 491
reduce 177
Reduzieroperationen 177
reentrant 30
ReentrantLock 151
ReentrantReadWriteLock 151
Referenzübergabe 341
Reflection API 373
Region (AWS) 487
Registry 325
Reihenfolgevertauschung 258, 260
Relational Database Service 491
release 101, 161
Remote 313
RemoteException 313
Remote Method Invocation 311
removeAttribute 422
Representational State Transfer 468
Request-Reply 382, 391
Reserved Instance 513
Reservierte Instanz 513
reset 249
REST 530
restart 249
RESTful WebServices 468
resume 85, 90
Return by Reference 331
Return by Value 330
RMI 311
rmic 317, 372
RMIClientSocketFactory 376
rmiregistry 318
RMI-Registry 314, 318, 324, 325
RMIServerSocketFactory 376
RMISocketFactory 376
Rollback 397
Root User 493
run 6
runLater 236
Runnable 8

S

S3 488, 496
SaaS 485
Sammeloperationen 177
Saving Plan 513
Scene 202
Scenebuilder 223
ScheduledExecutorService 149
Scheduled Instance 513
ScheduledService 244, 251
ScheduledThreadPoolExecutor 149
Scheduling 76
Schicht 255
Schichtenmodell 255
Schreibsperre 151
Secret Access Key 493
Secure Socket Layer 306, 376
Security Group 489, 512
select 303
Selector 302
Semaphor 100, 161
– additiv 107
– binär 106
Semaphorgruppe 109
send 268
serialisierbar 332
Serialisierung 332, 470
Serializable 332
Server 263
Serverless 529
ServerSocket 266, 280, 281
ServerSocketChannel 303
Service 244, 249
Servlet 409
ServletContext 422
Session 425
set 179, 208
setAttribute 422
setContentType 412
setDaemon 89
setDefaultUncaughtExceptionHandler 90
setDisable 238
setException 179
setHeader 415
setLayoutX 212
setLayoutY 212
setMaxAge 434
setMaxInactiveInterval 428
setMaxPriority 85
setName 14
setOnAction 205
setOnMouseDragged 219
setOnMouseMoved 219
setOnMousePressed 219
setOnMouseReleased 219
setPriority 76
setSoTimeout 268

setStatus 415
setText 207, 210
setUncaughtExceptionHandler 90
Shape 217
show 202
shutdown 146
shutdownInput 281
shutdownNow 146
shutdownOutput 281
signal 151
signalAll 151
SimpleBooleanProperty 208
SimpleDoubleProperty 208
SimpleIntegerProperty 208
SimpleLongProperty 208
Simple Notification Service 399
Simple Object Access Protocol 468
SimpleObjectProperty 208
Simple Queue Service 399
Simple Storage Service 488, 496
SimpleStringProperty 208
Skalierung
– horizontal 305, 380, 433
– vertikal 305, 380
Skeleton 312, 372, 373
sleep 16
Slider 201, 217
SMTP 285
SNS 399
SOAP 468
Socket 266, 280, 281
Socket-Schnittstelle 262
Software as a Service 485
sorted 179
Sperre 26, 149
Spot Instance 513
SQS 399
SSH-Client 512
SSL 306, 376
SslRMIClientSocketFactory 376
SslRMIServerSocketFactory 376
SSLServerSocket 308
SSLServerSocketFactory 308
SSLSocket 308
StackPane 213
Stage 202
Standard Window Toolkit 201
start 6, 202, 249
stateless 306, 380
stop 40, 85, 90
stream 173
Stream 173
StreamMessage 398
Stub 312, 372, 374
submit 145, 179
Suchoperationen 177
sum 174, 177
Supplier 180
supplyAsync 179
suspend 85, 90
Swing 201
SWT 201
synchronized 25
Synchronized-Block 27
SynchronousQueue 164
System
– eng gekoppelt 2
– lose gekoppelt 2, 4
– verteiltes 2, 3, 4
Szene 202

T

TableView 217
take 146, 164
Task 244, 246
TBB 171
TCP 258, 261, 280
TCPSocket 286
Text 218
TextArea 216
TextField 216
TextInputControl 216
TextMessage 385
thenApplyAsync 180
thenCombineAsync 182
Thread 3, 4, 293
– Gruppe 84
– (Klasse) 6
ThreadGroup 84
Threading Building Blocks 171
ThreadLocal 91
Thread-Pool 144, 146, 179
ThreadPoolExecutor 146
thread-safe 157
thread-sicher 157
TilePane 213
Time-To-Live 277
TimeUnit 145
TLS 306
ToggleButton 215
ToggleGroup 216
ToIntFunction 173
toJson 471
Tomcat 410
Topic 382, 384
Transaktion 397
Transcribe 511
transient 336
Translate 507
Transmission Control Protocol 258, 261
transparent 311
Transparenz 313
Transport Layer 258
Transport Layer Security 306

Transportschicht 258
Traversierungsoperationen 177
TreeTableView 217
TreeView 217
Treiber 258
tryLock 150
try-with-resources 269, 271
Two Factor Authentication 493

U

Überlastkontrolle 261
UDP 258, 260, 267
UI Control 201
unbind 324
uncaughtException 90
Unexportieren 363
unexportObject 363
Unicast-Adresse 259
UnicastRemoteObject 313, 361
Union File System 522
Universally Unique Identifier 500
Universal Resource Locator 403
unlock 149
unteilbar 30, 157
unzuverlässig 258
up 101
Upcall 207
updateMessage 245
updateProgress 245
updateTitle 245
updateValue 245
URL 310, 403
URLConnection 310
User Datagram Protocol 258, 260
User Threads 89
UUID 500

V

v 101
VBox 202, 213
verbindungslos 258, 260
verbindungsorientiert 261
Verbraucher 67, 112
Verklemmung 63, 109, 125, 185
Verlust 258
Vermittlungsschicht 258
Verschlüsseln 284
verteilte Anwendungen 2
verteilte Systeme 2
Verteilung 1
Verteilungstransparenz 312
Virtualisierung 2
Virtual Private Cloud 489, 512
volatile 31
Vordergrund-Threads 89
VPC 489, 512

W

wait 54, 58, 66
WebServices 467
WebSockets 477
well-known port number 263
Wertübergabe 331
Widget 201
Wiki 439
wohlbekannte Portnummer 263
Worker 244
World Wide Web (WWW) 258
write 282, 285
writeInt 283
writeObject 334
Writer 282
WWW 258

Y

yield 90

Z

Zone 487
zustandslos 306, 380, 433
Zustandsübergangsdiagramm 96
zuverlässig 261