

ORACLE  
PRESS

Also Covers Java SE 11 Developer Exam



# OCP

ORACLE CERTIFIED PROFESSIONAL

JAVA SE 17 Developer Exam 1Z0-829

# Programmer's Guide

Volume I

Khalid A. Mughal  
Vasily A. Strelnikov  
*Foreword by Simon Roberts*

ORACLE

FREE SAMPLE CHAPTER |



# **OCP**

**Oracle Certified Professional**

**Java SE 17 Developer**

**(Exam 1Z0-829)**

**Programmer's Guide**

*This page intentionally left blank*

# OCP

**Oracle Certified Professional**

**Java SE 17 Developer**

**(Exam 1Z0-829)**

**Programmer's Guide**

**Khalid A. Mughal**

**Vasily A. Strelnikov**



Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town  
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City  
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: your/Shutterstock

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The views expressed in this book are those of the author and do not necessarily reflect the views of Oracle.

Oracle America Inc. does not make any representations or warranties as to the accuracy, adequacy or completeness of any information contained in this work, and is not responsible for any errors or omissions.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

Visit us on the Web: [informit.com](http://informit.com)

Library of Congress Control Number: 2022951639

Copyright © 2023 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearson.com/permissions](http://www.pearson.com/permissions).

ISBN-13: 978-0-13-799364-2

ISBN-10: 0-13-799364-1

ScoutAutomatedPrintCode

## **Pearson's Commitment to Diversity, Equity, and Inclusion**

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at <https://www.pearson.com/report-bias.html>.

*This page intentionally left blank*

*To the loving memory of my mother, Zubaida Begum,  
and my father, Mohammed Azim.  
And to the future generation: Tobias Albert, Ronja Johanne  
and Serine Begum—with all my love.  
—K. A.M.*

*This page intentionally left blank*

# Contents Overview



<b>Figures</b>	<b>xxi</b>
<b>Tables</b>	<b>xxv</b>
<b>Examples</b>	<b>xxix</b>
<b>Foreword</b>	<b>xxxvii</b>
<b>Preface</b>	<b>xxxix</b>
<b>1 Basics of Java Programming</b>	<b>1</b>
<b>2 Basic Elements, Primitive Data Types, and Operators</b>	<b>29</b>
<b>3 Declarations</b>	<b>97</b>
<b>4 Control Flow</b>	<b>151</b>
<b>5 Object-Oriented Programming</b>	<b>189</b>
<b>6 Access Control</b>	<b>323</b>
<b>7 Exception Handling</b>	<b>363</b>
<b>8 Selected API Classes</b>	<b>423</b>
<b>9 Nested Type Declarations</b>	<b>489</b>
<b>10 Object Lifetime</b>	<b>531</b>
<b>11 Generics</b>	<b>563</b>
<b>12 Collections, Part I: <code>ArrayList&lt;E&gt;</code></b>	<b>643</b>

<b>13</b>	<b>Functional-Style Programming</b>	<b>673</b>
<b>14</b>	<b>Object Comparison</b>	<b>741</b>
<b>15</b>	<b>Collections: Part II</b>	<b>781</b>
<b>16</b>	<b>Streams</b>	<b>879</b>
<b>17</b>	<b>Date and Time</b>	<b>1023</b>
<b>18</b>	<b>Localization</b>	<b>1095</b>
<b>19</b>	<b>Java Module System</b>	<b>1161</b>
<b>20</b>	<b>Java I/O: Part I</b>	<b>1231</b>
<b>21</b>	<b>Java I/O: Part II</b>	<b>1285</b>
<b>22</b>	<b>Concurrency: Part I</b>	<b>1365</b>
<b>23</b>	<b>Concurrency: Part II</b>	<b>1419</b>
<b>24</b>	<b>Database Connectivity</b>	<b>1511</b>
<b>25</b>	<b>Annotations</b>	<b>1555</b>
<b>26</b>	<b>Secure Coding</b>	<b>1599</b>
<b>A</b>	<b>Taking the Java SE 17 and Java SE 11 Developer Exams</b>	<b>1615</b>
<b>B</b>	<b>Exam Topics: Java SE 17 Developer</b>	<b>1623</b>
<b>C</b>	<b>Exam Topics: Java SE 11 Developer</b>	<b>1629</b>
<b>D</b>	<b>Annotated Answers to Review Questions</b>	<b>1635</b>
<b>E</b>	<b>Mock Exam: Java SE 17 Developer</b>	<b>1709</b>
<b>F</b>	<b>Annotated Answers to Mock Exam</b>	<b>1737</b>
<b>G</b>	<b>Java Logging API Overview</b>	<b>1747</b>
	<b>Index</b>	<b>1753</b>

# Contents



<b>Figures</b>	<b>xxi</b>
<b>Tables</b>	<b>xxv</b>
<b>Examples</b>	<b>xxix</b>
<b>Foreword</b>	<b>xxxvii</b>
<b>Preface</b>	<b>xxxix</b>
Writing This Book	xxxix
About This Book	xl
Using This Book	xli
Book Website	xliii
Request for Feedback	xliii
About the Authors	xliv
Acknowledgments	xliv
<b>1 Basics of Java Programming</b>	<b>1</b>
1.1 The Java Ecosystem	2
1.2 Classes	5
1.3 Objects	8
1.4 Instance Members	9
1.5 Static Members	10
1.6 Inheritance	13
1.7 Aggregation	16
<i>Review Questions</i>	17
1.8 Sample Java Program	19
1.9 Program Output	24
<i>Review Questions</i>	26

<b>2</b>	<b>Basic Elements, Primitive Data Types, and Operators</b>	<b>29</b>
2.1	Basic Language Elements	30
2.2	Primitive Data Types	41
2.3	Conversions	43
2.4	Type Conversion Contexts	46
2.5	Precedence and Associativity Rules for Operators	50
2.6	Evaluation Order of Operands	52
2.7	The Simple Assignment Operator =	54
2.8	Arithmetic Operators: *, /, %, +, -	58
2.9	The Binary String Concatenation Operator +	67
2.10	Variable Increment and Decrement Operators: ++, --	69
	<i>Review Questions</i>	71
2.11	Boolean Expressions	74
2.12	Relational Operators: <, <=, >, >=	74
2.13	Equality	75
2.14	Boolean Logical Operators: !, ^, &,	78
2.15	Conditional Operators: &&,	80
2.16	Integer Bitwise Operators: ~, &,  , ^	82
2.17	Shift Operators: <<, >>, >>>	86
2.18	The Conditional Operator ?:	90
2.19	Other Operators: new, [], instanceof, ->	92
	<i>Review Questions</i>	93
<b>3</b>	<b>Declarations</b>	<b>97</b>
3.1	Class Declarations	99
3.2	Method Declarations	100
3.3	Statements	101
3.4	Variable Declarations	102
3.5	Instance Methods and the Object Reference this	106
3.6	Method Overloading	108
3.7	Constructors	109
3.8	Static Member Declarations	112
	<i>Review Questions</i>	115
3.9	Arrays	117
3.10	Parameter Passing	127
3.11	Variable Arity Methods	136
3.12	The main() Method	141
3.13	Local Variable Type Inference	142
	<i>Review Questions</i>	147
<b>4</b>	<b>Control Flow</b>	<b>151</b>
4.1	Selection Statements	152
4.2	The swi tch Statement	155
4.3	The swi tch Expression	164
	<i>Review Questions</i>	170

4.4	Iteration Statements	172
4.5	The while Statement	172
4.6	The do-while Statement	173
4.7	The for(;;) Statement	174
4.8	The for(:) Statement	176
4.9	Transfer Statements	179
4.10	Labeled Statements	179
4.11	The break Statement	180
4.12	The continue Statement	182
4.13	The return Statement	184
	<i>Review Questions</i>	185
<b>5</b>	<b>Object-Oriented Programming</b>	<b>189</b>
5.1	Implementing Inheritance	191
5.2	The Object Reference super	206
5.3	Chaining Constructors Using this() and super()	209
	<i>Review Questions</i>	215
5.4	Abstract Classes and Methods	218
5.5	Final Declarations	225
	<i>Review Questions</i>	234
5.6	Interfaces	237
	<i>Review Questions</i>	257
5.7	Arrays and Subtyping	259
5.8	Reference Values and Conversions	261
5.9	Reference Value Assignment Conversions	261
5.10	Method Invocation Conversions Involving References	265
5.11	Reference Casting and the instanceof Operator	269
5.12	Polymorphism	278
	<i>Review Questions</i>	283
5.13	Enum Types	287
5.14	Record Classes	299
5.15	Sealed Classes and Interfaces	311
	<i>Review Questions</i>	318
<b>6</b>	<b>Access Control</b>	<b>323</b>
6.1	Design Principle: Encapsulation	324
6.2	Java Source File Structure	325
6.3	Packages	326
6.4	Searching for Classes on the Class Path	337
	<i>Review Questions</i>	341
6.5	Access Modifiers	345
6.6	Scope Rules	352
6.7	Implementing Immutability	356
	<i>Review Questions</i>	360

<b>7</b>	<b>Exception Handling</b>	<b>363</b>
7.1	Stack-Based Execution and Exception Propagation	365
7.2	Exception Types	368
7.3	Exception Handling: try, catch, and finally	375
7.4	The throw Statement	386
7.5	The throws Clause	388
	<i>Review Questions</i>	392
7.6	The Multi-catch Clause	397
7.7	The try-with-resources Statement	407
7.8	Advantages of Exception Handling	416
	<i>Review Questions</i>	417
<b>8</b>	<b>Selected API Classes</b>	<b>423</b>
8.1	Overview of the java.lang Package	425
8.2	The Object Class	425
8.3	The Wrapper Classes	429
	<i>Review Questions</i>	438
8.4	The String Class	439
8.5	The StringBuilder Class	464
	<i>Review Questions</i>	471
8.6	The Math Class	478
8.7	The Random Class	482
8.8	Using Big Numbers	484
	<i>Review Questions</i>	487
<b>9</b>	<b>Nested Type Declarations</b>	<b>489</b>
9.1	Overview of Nested Type Declarations	491
9.2	Static Member Types	495
9.3	Non-Static Member Classes	501
	<i>Review Questions</i>	510
9.4	Local Classes	512
9.5	Static Local Types	519
9.6	Anonymous Classes	521
	<i>Review Questions</i>	527
<b>10</b>	<b>Object Lifetime</b>	<b>531</b>
10.1	Garbage Collection	533
10.2	Reachable Objects	533
10.3	Facilitating Garbage Collection	536
10.4	Invoking Garbage Collection Programmatically	537
	<i>Review Questions</i>	538
10.5	Initializers	540
10.6	Field Initializer Expressions	540
10.7	Static Initializer Blocks	545
10.8	Instance Initializer Blocks	551

10.9	Constructing Initial Object State	555
	<i>Review Questions</i>	558
<b>11</b>	<b>Generics</b>	<b>563</b>
11.1	Introducing Generics	565
11.2	Generic Types and Parameterized Types	567
11.3	Collections and Generics	578
11.4	Wildcards	579
11.5	Using References of Wildcard Parameterized Types	584
11.6	Bounded Type Parameters	591
11.7	Generic Methods and Constructors	593
11.8	Implementing a Simplified Generic Stack	598
	<i>Review Questions</i>	600
11.9	Wildcard Capture	604
11.10	Flexibility with Wildcard Parameterized Types	607
11.11	Type Erasure	613
11.12	Implications for Overloading and Overriding	615
11.13	Limitations and Restrictions on Generic Types	623
	<i>Review Questions</i>	636
<b>12</b>	<b>Collections, Part I: ArrayList&lt;E&gt;</b>	<b>643</b>
12.1	Lists	644
12.2	Declaring References and Constructing ArrayLists	646
12.3	Modifying an ArrayList<E>	651
12.4	Querying an ArrayList<E>	655
12.5	Iterating Over an ArrayList<E>	657
12.6	Converting an ArrayList<E> to an Array	658
12.7	Creating List Views	659
12.8	Arrays versus ArrayLists	662
	<i>Review Questions</i>	667
<b>13</b>	<b>Functional-Style Programming</b>	<b>673</b>
13.1	Functional Interfaces	675
13.2	Lambda Expressions	679
13.3	Lambda Expressions and Anonymous Classes	688
	<i>Review Questions</i>	693
13.4	Overview of Built-In Functional Interfaces	695
13.5	Suppliers	699
13.6	Predicates	703
13.7	Consumers	709
13.8	Functions	712
13.9	Two-Arity Specialization of Function<T, R>: BiFunction<T, U, R>	717
13.10	Extending Function<T, T>: UnaryOperator<T>	720
13.11	Extending BiFunction<T, T, T>: BinaryOperator<T>	721
13.12	Currying Functions	723

13.13	Method and Constructor References	724
13.14	Contexts for Defining Lambda Expressions	733
	<i>Review Questions</i>	735
<b>14</b>	<b>Object Comparison</b>	<b>741</b>
14.1	The Objects Class	743
14.2	Implementing the equals() Method	744
14.3	Implementing the hashCode() Method	753
14.4	Implementing the java.lang.Comparable<E> Interface	761
14.5	Implementing the java.util.Comparator<E> Interface	769
	<i>Review Questions</i>	774
<b>15</b>	<b>Collections: Part II</b>	<b>781</b>
15.1	The Java Collections Framework	783
15.2	Collections	790
15.3	Lists	801
15.4	Sets	804
15.5	Sorted Sets and Navigable Sets	810
15.6	Queues	814
15.7	Deque	821
	<i>Review Questions</i>	826
15.8	Maps	830
15.9	Map Implementations	840
15.10	Sorted Maps and Navigable Maps	845
	<i>Review Questions</i>	851
15.11	The Collections Class	856
15.12	The Arrays Class	864
	<i>Review Questions</i>	874
<b>16</b>	<b>Streams</b>	<b>879</b>
16.1	Introduction to Streams	881
16.2	Running Example: The CD Record Class	882
16.3	Stream Basics	884
16.4	Building Streams	890
16.5	Intermediate Stream Operations	905
16.6	The Optional Class	940
16.7	Terminal Stream Operations	946
16.8	Collectors	978
16.9	Parallel Streams	1009
	<i>Review Questions</i>	1016
<b>17</b>	<b>Date and Time</b>	<b>1023</b>
17.1	Date and Time API Overview	1024
17.2	Working with Dates and Times	1027

17.3	Using Temporal Units and Temporal Fields	1044
17.4	Working with Instants	1049
17.5	Working with Periods	1057
17.6	Working with Durations	1064
17.7	Working with Time Zones and Daylight Savings	1072
17.8	Converting Date and Time Values to Legacy Date	1088
	<i>Review Questions</i>	1089
<b>18</b>	<b>Localization</b>	<b>1095</b>
18.1	Using Locales	1096
18.2	Properties Files	1100
18.3	Bundling Resources	1102
	<i>Review Questions</i>	1112
18.4	Core API for Formatting and Parsing of Values	1115
18.5	Formatting and Parsing Number, Currency, and Percentage Values	1116
18.6	Formatting and Parsing Date and Time	1127
18.7	Formatting and Parsing Messages	1139
	<i>Review Questions</i>	1153
<b>19</b>	<b>Java Module System</b>	<b>1161</b>
19.1	Making the Case for Modules	1163
19.2	The Modular JDK	1164
19.3	Module Basics	1168
19.4	Overview of Module Directives	1177
19.5	Creating a Modular Application	1179
19.6	Compiling and Running a Modular Application	1186
19.7	Creating JAR Files	1189
19.8	Open Modules and the opens Directive	1191
19.9	Services	1196
19.10	Creating Runtime Images	1204
19.11	Categories of Modules	1205
19.12	Migrating to Modules	1209
19.13	Exploring Modules	1211
19.14	Summary of Selected Operations with the JDK Tools	1218
	<i>Review Questions</i>	1223
<b>20</b>	<b>Java I/O: Part I</b>	<b>1231</b>
20.1	Input and Output	1233
20.2	Byte Streams: Input Streams and Output Streams	1234
20.3	Character Streams: Readers and Writers	1241
20.4	The Console Class	1256
	<i>Review Questions</i>	1259
20.5	Object Serialization	1261
	<i>Review Questions</i>	1277

<b>21</b>	<b>Java I/O: Part II</b>	<b>1285</b>
21.1	Characteristics of a Hierarchical File System	1287
21.2	Creating Path Objects	1289
21.3	Working with Path Objects	1294
21.4	Operations on Directory Entries	1304
21.5	Reading and Writing Files Using Paths	1314
21.6	Managing File Attributes	1321
21.7	Creating Directory Entries	1339
21.8	Stream Operations on Directory Entries	1345
	<i>Review Questions</i>	1355
<b>22</b>	<b>Concurrency: Part I</b>	<b>1365</b>
22.1	Threads and Concurrency	1367
22.2	Runtime Organization for Thread Execution	1369
22.3	Creating Threads	1370
	<i>Review Questions</i>	1378
22.4	Thread Lifecycle	1380
22.5	Thread Issues	1408
	<i>Review Questions</i>	1415
<b>23</b>	<b>Concurrency: Part II</b>	<b>1419</b>
23.1	Utility Classes <code>TimeUnit</code> and <code>ThreadLocalRandom</code>	1421
23.2	The Executor Framework	1423
23.3	The Fork/Join Framework	1447
23.4	Writing Thread-Safe Code	1451
23.5	Special-Purpose Synchronizers	1470
23.6	Synchronized Collections and Maps	1477
23.7	Concurrent Collections and Maps	1482
	<i>Review Questions</i>	1504
<b>24</b>	<b>Database Connectivity</b>	<b>1511</b>
24.1	Introduction to Relational Databases	1512
24.2	Introduction to JDBC	1517
24.3	Establishing a Database Connection	1519
24.4	Creating and Executing SQL Statements	1522
24.5	Processing Query Results	1533
24.6	Customizing Result Sets	1539
24.7	Discovering Database and <code>ResultSet</code> Metadata	1543
24.8	Implementing Transaction Control	1545
	<i>Review Questions</i>	1548
<b>25</b>	<b>Annotations</b>	<b>1555</b>
25.1	Basics of Annotations	1557
25.2	Declaring Annotation Types	1558

25.3	Applying Annotations	1563
25.4	Meta-Annotations	1567
25.5	Selected Standard Annotations	1577
25.6	Processing Annotations	1587
	<i>Review Questions</i>	1593
<b>26</b>	<b>Secure Coding</b>	<b>1599</b>
26.1	Application Security Overview	1600
26.2	Security Threat Categories	1602
26.3	Java Security Policies	1608
26.4	Additional Security Guidelines	1610
	<i>Review Questions</i>	1611
<b>A</b>	<b>Taking the Java SE 17 and Java SE 11 Developer Exams</b>	<b>1615</b>
A.1	Preparing for the Exam	1615
A.2	Registering for the Online Proctored Exam	1616
A.3	How the Online Proctored Exam Is Conducted	1618
A.4	The Questions	1619
<b>B</b>	<b>Exam Topics: Java SE 17 Developer</b>	<b>1623</b>
<b>C</b>	<b>Exam Topics: Java SE 11 Developer</b>	<b>1629</b>
<b>D</b>	<b>Annotated Answers to Review Questions</b>	<b>1635</b>
<b>E</b>	<b>Mock Exam: Java SE 17 Developer</b>	<b>1709</b>
<b>F</b>	<b>Annotated Answers to Mock Exam</b>	<b>1737</b>
<b>G</b>	<b>Java Logging API Overview</b>	<b>1747</b>
G.1	Purpose of the Logging API	1747
G.2	Configuring Logging	1748
G.3	Writing Log Messages	1749
G.4	Applying Guarded Logging	1751
G.5	Summary	1751
	<b>Index</b>	<b>1753</b>

*This page intentionally left blank*

# Figures



1.1	UML Notation for Classes	6
1.2	UML Notation for Objects	9
1.3	Class Diagram Showing Static Members of a Class	11
1.4	Members of a Class	12
1.5	Class Diagram Depicting Inheritance Relationship	14
1.6	Class Diagram Depicting Aggregation	16
2.1	Primitive Data Types in Java	41
2.2	Widening Primitive Conversions	44
2.3	Overflow and Underflow in Floating-Point Arithmetic	60
2.4	Numeric Promotion in Arithmetic Expressions	65
3.1	Array of Arrays	126
3.2	Parameter Passing: Primitive Data Values	130
3.3	Parameter Passing: Reference Values	132
3.4	Parameter Passing: Arrays	133
4.1	Activity Diagram for <code>if</code> Statements	153
4.2	Form of the <code>switch</code> Statement with the Colon Notation	156
4.3	Activity Diagram for the <code>switch</code> Statement with the Colon Notation	157
4.4	Form of the <code>switch</code> Statement with the Arrow Notation	160
4.5	Activity Diagram for the <code>switch</code> Statement with the Arrow Notation	161
4.6	Activity Diagram for the <code>while</code> Statement	173
4.7	Activity Diagram for the <code>do-while</code> Statement	174
4.8	Activity Diagram for the <code>for</code> Statement	175
4.9	Enhanced <code>for</code> Statement	177
5.1	Inheritance Hierarchy	194
5.2	Inheritance Hierarchy for Example 5.2	197
5.3	Class Diagram for Example 5.8	220
5.4	Inheritance Hierarchies for Classes and Interfaces in Example 5.10	245
5.5	Inheritance Relationships for Interface Constants	255
5.6	Reference Type Hierarchy: Arrays and Subtype Covariance	259
5.7	Type Hierarchy to Illustrate Polymorphism	279
5.8	A Domain Model as Represented by Inheritance Hierarchy	312
5.9	Sealed Classes for Domain Modeling of Inheritance Hierarchy	313
5.10	Sealed Classes and Interfaces	316

6.1	Java Source File Structure	326
6.2	Package Structure	327
6.3	Compiling Code into Package Directories	336
6.4	Searching for Classes	338
6.5	Accessibility of Class Members	348
6.6	Block Scope	355
7.1	Normal Method Execution	367
7.2	Exception Propagation	368
7.3	Partial Exception Inheritance Hierarchy	369
7.4	The <code>try-catch-finally</code> Construct	376
7.5	Exception Handling (Scenario 1)	378
7.6	Exception Handling (Scenario 2)	380
7.7	Exception Handling (Scenario 3)	382
8.1	Partial Inheritance Hierarchy in the <code>java.lang</code> Package	425
8.2	Converting Values among Primitive, Wrapper, and String Types	430
8.3	Incidental Whitespace in Text Blocks	462
9.1	Overview of Type Declarations	491
9.2	Static Member Classes and Interfaces	499
9.3	Outer Object with Associated Inner Objects	503
9.4	Non-Static Member Classes and Inheritance	508
9.5	Local Classes and Inheritance Hierarchy	517
10.1	Memory Organization at Runtime	535
11.1	Extending Generic Types	574
11.2	No Subtype Covariance for Parameterized Types	580
11.3	Partial Type Hierarchy for <code>Node&lt;? extends Number&gt;</code>	581
11.4	Partial Type Hierarchy for <code>Node&lt;? super Integer&gt;</code>	582
11.5	Partial Type Hierarchy for Selected Parameterized Types of <code>Node&lt;E&gt;</code>	583
12.1	Partial <code>ArrayList</code> Inheritance Hierarchy	645
15.1	The Core Interfaces	784
15.2	The Core Collection Interfaces and Their Implementations	786
15.3	The Core Map Interfaces and Their Implementations	787
15.4	Bulk Operations on Collections	791
16.1	Data Processing with Streams	885
16.2	The Core Stream Interfaces	890
16.3	Filtering Stream Elements	913
16.4	Selecting Distinct Elements	916
16.5	Skipping Elements at the Head of a Stream	916
16.6	Truncating a Stream	917
16.7	Mapping	923
16.8	Incorrect Solution to the Query	925
16.9	Mapping a Stream of Streams	925
16.10	Flattening Streams	926
16.11	Reducing with an Initial Value	957
16.12	Parallel Functional Reduction	963
16.13	Sequential Mutable Reduction	966
16.14	Parallel Mutable Reduction	967

16.15	Collecting to a Map	982
16.16	Grouping	986
16.17	Multilevel Grouping as a Two-Dimensional Matrix	988
16.18	Partitioning	991
18.1	Core Classes for Formatting in the <code>java.text</code> Package	1115
19.1	Structure of JDK 17 Installation	1166
19.2	Partial Module Graph of Java SE 17 Modules	1167
19.3	Modules and Module Declarations	1170
19.4	Module Graph and Implicit Dependencies	1173
19.5	Module Declarations and Module Graph	1175
19.6	Implied Module Dependencies	1176
19.7	Qualified Export	1177
19.8	Module Graph of the <code>adviceApp</code> Application	1180
19.9	Application Directory Structure	1181
19.10	Exploded Module Structure for the <code>model</code> Module	1181
19.11	Module Graph and Module Declarations	1182
19.12	Compiling Exploded Modules Containing Source Code	1183
19.13	Compiling the <code>model</code> Module	1187
19.14	Creating Modular JARs	1188
19.15	Module Graph and Module Declarations for Services	1198
19.16	Bottom-Up Strategy for Code Migration	1210
19.17	Top-Down Strategy for Code Migration	1211
19.18	Module Graph Using the <code>jdeps</code> Tool	1217
20.1	Partial Byte Stream Inheritance Hierarchies in the <code>java.io</code> Package	1235
20.2	Stream Chaining for Reading and Writing Binary Values to a File	1239
20.3	Selected Character Streams in the <code>java.io</code> Package	1242
20.4	Setting Up a <code>PrintWriter</code> to Write to a File	1248
20.5	Setting Up Readers to Read Characters	1250
20.6	Buffered Writers	1251
20.7	Buffered Readers	1252
20.8	Keyboard and Display as Console	1256
20.9	Object Stream Chaining	1261
21.1	Creating Path Objects	1291
22.1	Runtime Data Areas	1369
22.2	Spawning Threads Using a <code>Runnable</code> Object	1372
22.3	Spawning Threads—Extending the <code>Thread</code> Class	1375
22.4	Thread States	1381
22.5	Starting a Thread	1386
22.6	Running and Yielding	1387
22.7	Blocked for Lock Acquisition	1388
22.8	Sleeping and Waking Up	1396
22.9	Timed Waiting and Notifying	1397
22.10	Thread Coordination	1398
22.11	Timed Joining of Threads	1403
22.12	Blocked for I/O	1405
22.13	Thread Termination	1405

22.14	Deadlock	1409
23.1	Executor Interfaces in the <code>java.util.concurrent</code> Package	1424
23.2	Executor Service Lifecycle	1428
23.3	Selected Interfaces and Classes in the Lock API	1461
23.4	Concurrent Collections in the <code>java.util.concurrent</code> Package	1485
23.5	Concurrent Maps in the <code>java.util.concurrent</code> Package	1490
23.6	Blocking Queues in the <code>java.util.concurrent</code> Package	1495
23.7	Copy-on-Write Collections in the <code>java.util.concurrent</code> Package	1501
24.1	Layered Database Connectivity	1517
25.1	API Documentation of a Deprecated Method	1582
25.2	Selected Types from the Reflection API	1588

# Tables



1.1	Terminology for Class Members	13
1.2	Format Specifier Examples	26
2.1	Keywords in Java	31
2.2	Contextual Keywords	31
2.5	Separators in Java	32
2.6	Examples of Literals	32
2.3	Reserved Keywords Not Currently in Use	32
2.4	Reserved Literals in Java	32
2.7	Examples of Decimal, Binary, Octal, and Hexadecimal Literals	33
2.8	Representing Signed byte Values Using Two's Complement	34
2.9	Examples of Character Literals	37
2.10	Escape Sequences	38
2.11	Examples of Escape Sequence \ddd	38
2.13	Range of Character Values	42
2.14	Range of Floating-Point Values	42
2.12	Range of Integer Values	42
2.15	Boolean Values	43
2.16	Summary of Primitive Data Types	43
2.17	Selected Conversion Contexts and Conversion Categories	47
2.18	Operator Summary	51
2.19	Examples of Truncated Values	58
2.20	Arithmetic Operators	59
2.21	Examples of Arithmetic Expression Evaluation	64
2.22	Arithmetic Compound Assignment Operators	67
2.23	Relational Operators	74
2.24	Primitive Data Value Equality Operators	75
2.25	Reference Equality Operators	76
2.26	Truth Values for Boolean Logical Operators	78
2.27	Boolean Logical Compound Assignment Operators	79
2.28	Conditional Operators	80
2.29	Truth Values for Conditional Operators	80
2.30	Integer Bitwise Operators	83
2.31	Result Table for Bitwise Operators	83

2.32	Examples of Bitwise Operations	84
2.33	Bitwise Compound Assignment Operators	85
2.34	Shift Operators	86
2.35	Shift Compound Assignment Operators	90
3.1	Default Values	103
3.2	Parameter Passing by Value	128
4.1	Comparing the <code>switch</code> Statement and the <code>switch</code> Expression	169
4.2	The <code>return</code> Statement	184
5.1	Overriding, Hiding, and Overloading of Methods	204
5.2	Summary of Member Declarations in an Interface (Part I)	239
5.3	Summary of Member Declarations in an Interface (Part II)	239
5.4	Types and Values	259
6.1	Compiling and Executing Non-Modular Java Code	341
6.2	Access Modifiers for Top-Level Reference Types (Non-Modular)	347
6.3	Accessibility of Members in a Class (Non-Modular)	352
6.4	Accessing Members within a Class	353
6.5	Examples of Immutable Classes	356
7.1	Comparing Checked and Unchecked Exceptions	392
8.1	Indenting Lines in a String	457
8.2	Incidental Whitespace in Text Blocks	463
8.3	Applying Rounding Functions	480
8.4	Big Number Classes	485
9.1	Various Aspects of Type Declarations	494
11.1	Summary of Subtyping Relationships for Generic Types	581
11.2	Get and Set Operations Using Parameterized References	587
11.3	Summary of Get and Set Operations Using Parameterized References	590
11.4	Examples of Type Erasure	614
11.5	Examples of Reifiable Types	624
11.6	Examples of Non-Reifiable Types	624
12.1	Summary of Arrays versus <code>ArrayLists</code>	662
13.1	Basic Functional Interfaces in the <code>java.util.function</code> Package	695
13.2	Built-In Functional Interfaces in the <code>java.util.function</code> Package	696
13.3	Summary of Built-In Functional Interfaces	698
13.4	Suppliers	699
13.5	Predicates	703
13.6	Consumers	709
13.7	Functions	712
13.8	Two-arity Functions	718
13.9	Unary Operators	720
13.10	Binary Operators	722
13.11	Method and Constructor References	725
15.1	Core Interfaces and Concrete Classes in the Collections Framework	784
15.2	Summary of Collection and Map Implementations	788
15.3	Bulk Operations and Set Logic	804
15.4	Summary of Methods in the Queue Interface	815
15.5	Summary of Deque Methods	822

15.6	Summary of Scenarios Using Advanced Key-Based Operations	836
15.7	Map Implementations	840
16.1	Summary of Stream Building Methods	904
16.2	Interoperability between Stream Types	934
16.3	Selected Aspects of Intermediate Stream Operations	938
16.4	Intermediate Stream Operations	939
16.5	Terminal Stream Operations	977
16.6	Additional Terminal Operations in the Numeric Stream Interfaces	978
16.7	Static Methods in the Collectors Class	1005
16.8	Method Comparison: The Stream Interfaces and the Collectors Class	1008
17.1	Fields in Selected Classes in the Date and Time API	1025
17.2	Selected Method Name Prefixes in the Temporal Classes	1026
17.3	Selected Common Methods in the Temporal Classes	1026
17.4	Some Differences between the Period Class and the Duration Class	1072
18.1	Selected Language Codes	1097
18.2	Selected Country/Region Codes	1097
18.3	Selected Predefined Locales for Languages	1097
18.4	Selected Predefined Locales for Countries	1097
18.5	Result and Parent Bundles (See Example 18.3)	1110
18.6	Compact Number Styles	1120
18.7	Formatting Numbers to Compact Form	1121
18.8	Parsing Compact Form to Numbers	1122
18.9	Selected Rounding Modes	1123
18.10	Selected Number Format Pattern Symbols for the United States	1125
18.11	Selected ISO-Based Predefined Formatters for Date and Time	1130
18.13	Using Style-Based Formatters	1132
18.12	Format Styles for Date and Time	1132
18.14	Using Style-Based Parsers	1133
18.15	Selected Date/Time Pattern Letters	1135
18.16	Format Type and Format Style Combinations for Format Elements	1140
19.1	Module Accessibility Rules	1174
19.2	Overview of Module Directives	1177
19.3	The open Modifier for Modules	1178
19.4	How JARs Are Handled Depends on the Path	1206
19.5	Module Comparison	1208
19.6	Selected Operations with the JDK Tools	1218
19.7	Selected Options for the javac Tool	1219
19.8	Selected Options for the java Tool	1220
19.9	Selected Options for the jar Tool	1221
19.10	Selected Options for the jdeps Tool	1222
19.12	Selected Common Shorthand Options for JDK Tools	1223
19.11	Selected Options for the jlink Tool	1223
20.1	Selected Input Streams	1235
20.2	Selected Output Streams	1236
20.3	The DataInput and DataOutput Interfaces	1238
20.4	Selected Readers	1242

20.5	Selected Writers	1243
20.6	Print Methods of the <code>PrintWriter</code> Class	1246
20.7	Correspondence between Selected Byte and Character Streams	1255
21.1	Using Current and Parent Directory Designators	1289
21.2	Converting to an Absolute Path	1298
21.3	Normalizing Paths	1299
21.4	Resolving Paths	1300
21.5	Constructing a Relative Path between Two Paths	1301
21.7	Converting to a Real Path	1302
21.6	Link Option	1302
21.8	Standard Copy Options	1308
21.9	Selected Standard Open Options	1314
21.10	POSIX File Permissions	1326
21.11	Interfaces for Read-Only Access to File Attributes	1330
21.12	Selected File Attribute Views	1334
21.13	File Visit Option	1349
22.1	Thread States Defined by the <code>Thread.State</code> Enum Type	1382
22.2	Selective Priorities Defined by the <code>Thread</code> Class	1385
23.1	Constants Defined by the <code>java.util.concurrent.TimeUnit</code> Enum Type	1421
23.2	Selected Executor Services Provided by the Executors Utility Class	1426
23.3	Task-Defining Classes	1448
23.4	Selected Atomic Classes	1457
23.5	Concurrent Collections in the <code>java.util.concurrent</code> Package	1486
23.6	Characteristics of Concurrent Collections	1486
23.7	Selected Methods in the <code>ConcurrentLinkedQueue</code> Class	1487
23.8	Selected Methods in the <code>ConcurrentLinkedDeque</code> Class	1487
23.9	Concurrent Maps in the <code>java.util.concurrent</code> Package	1491
23.10	Characteristics of Concurrent Maps	1491
23.11	Blocking Queues in the <code>java.util.concurrent</code> Package	1496
23.13	Selected Methods in the <code>BlockingQueue</code> Interface	1497
23.12	Characteristics of Blocking Queues	1497
23.14	Selected Methods in the <code>BlockingDeque</code> Interface	1499
23.15	Copy-on-Write Collections in the <code>java.util.concurrent</code> Package	1502
23.16	Characteristics of Copy-on-Write Collections	1502
24.1	Definition of the <code>compositions</code> Table	1513
24.2	Data in the <code>compositions</code> Table	1513
24.3	Selected Constants Defined in the <code>ResultSet</code> Interface	1539
25.1	Retention Policy Values for the <code>@Retention</code> Meta-Annotation	1568
25.2	The <code>ElementType</code> Values for the <code>@Target</code> Meta-Annotation	1569
25.3	Summary of Selected Standard Annotations	1586

# Examples



1.1	Basic Elements of a Class Declaration	7
1.2	Static Members in Class Declaration	11
1.3	Defining a Subclass	15
1.4	Using Aggregation	17
1.5	A Sample Program	20
1.6	A Single-File Source-Code Program	22
2.1	Evaluation Order of Operands and Arguments	53
2.2	Numeric Promotion in Arithmetic Expressions	64
2.3	Short-Circuit Evaluation Involving Conditional Operators	81
2.4	Bitwise Operations	84
3.1	Default Values for Fields	104
3.2	Flagging Uninitialized Local Variables of Primitive Data Types	104
3.3	Flagging Uninitialized Local Reference Variables	105
3.4	Using the <code>this</code> Reference	107
3.5	Namespaces	110
3.6	Accessing Static Members in a Class	113
3.7	Using Arrays	121
3.8	Using Anonymous Arrays	123
3.9	Using Multidimensional Arrays	126
3.10	Passing Primitive Values	129
3.11	Passing Reference Values	131
3.12	Passing Arrays	132
3.13	Array Elements as Primitive Data Values	134
3.14	Array Elements as Reference Values	135
3.15	Calling a Variable Arity Method	138
3.16	Passing Program Arguments	142
3.17	Illustrating Local Variable Type Reference	143
4.1	Fall-Through in a <code>switch</code> Statement with the Colon Notation	158
4.2	Nested <code>switch</code> Statements with the Colon Notation	159
4.3	Nested <code>switch</code> Statements with the Arrow Notation	161
4.4	Strings in a <code>switch</code> Statement	162
4.5	Enums in a <code>switch</code> Statement	163
4.6	A <code>yield</code> Statement in a <code>switch</code> Expression with the Colon Notation	165

4.7	Statement Blocks in a <code>switch</code> Expression with the Arrow Notation	167
4.8	Expression Actions in a <code>switch</code> Expression with the Arrow Notation	168
4.9	Returning Multiple Values as a Record from a <code>switch</code> Expression	168
4.10	The <code>break</code> Statement	180
4.11	Labeled <code>break</code> Statement	182
4.12	The <code>continue</code> Statement	183
4.13	Labeled <code>continue</code> Statement	183
4.14	The <code>return</code> Statement	185
5.1	Extending Classes: Inheritance and Accessibility	192
5.2	Overriding, Overloading, and Hiding	198
5.3	Ambiguous Call to Overloaded Methods	202
5.4	Using the <code>super</code> Keyword	207
5.5	Constructor Overloading	209
5.6	The <code>this()</code> Constructor Call	210
5.7	The <code>super()</code> Constructor Call	212
5.8	Using Abstract Classes	222
5.9	Using the <code>final</code> Modifier	227
5.10	Implementing Interfaces	241
5.11	Default Methods in Interfaces	247
5.12	Inheriting Default Method Implementations from Superinterfaces	249
5.13	Inheriting Method Implementations from Supertypes	250
5.14	Static Methods in Interfaces	251
5.15	Private Methods in Interfaces	253
5.16	Constants in Interfaces	254
5.17	Inheriting Constants in Interfaces	256
5.18	Assigning and Passing Reference Values	262
5.19	Choosing the Most Specific Method (Simple Case)	266
5.20	Overloaded Method Resolution	268
5.21	The <code>instanceof</code> Type Comparison and Cast Operators	272
5.22	Using the <code>instanceof</code> Type Comparison Operator	273
5.23	Using Polymorphism	280
5.24	Using Enums	289
5.25	Declaring Enum Constructors and Members	291
5.26	Declaring Constant-Specific Class Bodies	295
5.27	The <code>CD</code> Class	299
5.28	The <code>CD Record</code> Class	301
5.29	Normal Canonical Record Constructor	305
5.30	Compact Record Constructor and Non-Canonical Constructor	306
5.31	Other Member Declarations in a Record Class	309
5.32	Declaring <code>sealed</code> , <code>non-sealed</code> , and <code>final</code> Classes	314
5.33	Using Sealed Classes	318
6.1	Defining Packages and Using Type Import	328
6.2	Single Static Import	331
6.3	Avoiding the Interface Constant Antipattern	332
6.4	Importing Enum Constants	333
6.5	Shadowing Static Import	333

6.6	Conflict in Importing a Static Method with the Same Signature	335
6.7	Access Modifiers for Top-Level Reference Types	345
6.8	Accessibility of Class Members	349
6.9	Class Scope	354
6.10	Implementing an Immutable Class	356
7.1	Method Execution	365
7.2	The try-catch Construct	378
7.3	Exception Propagation	380
7.4	The try-catch-finally Construct	383
7.5	The try-finally Construct	384
7.6	The finally Clause and the return Statement	385
7.7	Throwing Exceptions Programmatically	387
7.8	The throws Clause	389
7.9	Using Multiple catch Clauses	398
7.10	Using the Multi-catch Clause	399
7.11	Precise Rethrowing of Exceptions	401
7.12	Precise throws Clause	402
7.13	Conditions for Rethrowing Final Exceptions	404
7.14	Chaining Exceptions	405
7.15	Naive Resource Management	407
7.16	Explicit Resource Management	408
7.17	Using try with Automatic Resource Management (ARM)	411
7.18	Implementing the AutoCloseable Interface	413
8.1	Methods in the Object Class	427
8.2	Text Representation of Integers	436
8.3	String Construction and Internment	443
8.4	Reading Characters from a String	445
8.5	Controlling String Builder Capacity	470
9.1	Overview of Type Declarations	493
9.2	Static Member Types	495
9.3	Importing Static Member Types	498
9.4	Accessing Members in Enclosing Context (Static Member Classes)	499
9.5	Defining and Instantiating Non-Static Member Classes	501
9.6	Qualified this and Qualified Class Instance Creation Expression	505
9.7	Inheritance Hierarchy and Enclosing Context	507
9.8	Extending Inner Classes	508
9.9	Declaring Local Classes	513
9.10	Accessing Local Declarations in the Enclosing Block (Local Classes)	514
9.11	Accessing Members in the Enclosing Class (Local Classes)	516
9.12	Instantiating Local Classes	517
9.13	Defining Static Local Record Classes	520
9.14	Defining Anonymous Classes	522
9.15	Accessing Declarations in Enclosing Context (Anonymous Classes)	526
10.1	Initializer Expression Order and Method Calls	542
10.2	Exceptions in Initializer Expressions	544
10.3	Static Initializers and Forward References	547

10.4	Static Initializers and Order of Execution	548
10.5	Static Initializer Blocks and Exceptions	549
10.6	Instance Initializers and Forward References	552
10.7	Instance Initializer Block in Anonymous Class	553
10.8	Exception Handling in Instance Initializer Blocks	554
10.9	Object State Construction	556
10.10	Initialization Anomaly under Object State Construction	557
11.1	A Legacy Class	566
11.2	A Generic Class for Nodes	568
11.3	A Generic Interface and Its Implementation	572
11.4	A Non-Generic Class Implementing a Generic Interface	573
11.5	Extending Generic Types	574
11.6	Unchecked Warnings	577
11.7	Illustrating Get and Set Operations Using Parameterized References	587
11.8	Implementing the Comparable<E> Interface	592
11.9	Declaring Generic Methods	593
11.10	Implementing a Simplified Generic Stack	598
11.11	Using Recursive Bounds	612
11.12	Using the @Override Annotation	618
11.13	Subsignatures	619
11.14	Overriding from Generic Supertype	619
11.15	Missing Supertype Parameterization	620
11.16	Genericity Cannot be Added to Inherited Methods	621
11.17	Restrictions on Exception Handling	632
11.18	Type Parameter in throws Clause	633
11.19	Generic Nested Types	634
11.20	Instantiating Generic Nested Classes	635
11.21	Enum Implements Parameterized Generic Interface	635
12.1	Using an ArrayList	663
13.1	Accessing Members in the Enclosing Object	686
13.2	Accessing Local Variables in the Enclosing Method	688
13.3	Filtering an ArrayList<E>	689
13.4	Implementing Suppliers	701
13.5	Implementing the BiPredicate<T, U> Functional Interface	707
13.6	Implementing Functions	714
14.1	Not Overriding the Object.equals() Method	744
14.2	Implications of Not Overriding the Object.equals() Method	745
14.3	Implementing the equals() Method	748
14.4	Implications of Implementing the equals() Method	751
14.5	Implications of Not Overriding the Object.hashCode() Method	755
14.6	Implementing the hashCode() Method	757
14.7	Implications of Implementing the hashCode() Method	759
14.8	Implementing the compareTo() Method of the Comparable<E> Interface	763
14.9	Implications of Implementing the compareTo() Method	766
14.10	Natural Ordering and Total Orderings	770
14.11	Using a Comparator for Version Numbers	772

15.1	Iterating Over a Collection	792
15.2	Converting Collections to Arrays	799
15.3	Using Lists	803
15.4	Iterating Over Sets	807
15.5	Using Sets	809
15.6	Using Navigable Sets	813
15.7	Using Priority Queues	818
15.8	Demonstrating Deque Operations	823
15.9	Using Deques as a LIFO Stack and as a FIFO Queue	825
15.10	Using Maps	844
15.11	Using Navigable Maps	849
16.1	The CD Example Classes	883
16.2	Data Processing Using Streams	888
16.3	Stream Mapping and Loop Fusion	906
16.4	Order of Intermediate Operations	909
16.5	Filtering	918
16.6	Examining Stream Elements	920
16.7	Sorting Streams	930
16.8	Using Optionals	941
16.9	Using Numerical Optionals	945
16.10	Identity Matrix Test	951
16.11	Implementing Functional Reductions	958
16.12	Implementing Mutable Reductions	968
16.13	Flat mapping	996
16.14	Benchmarking	1012
17.1	Creating Local Dates and Local Times	1031
17.2	Using Local Dates and Local Times	1037
17.3	Temporal Arithmetic	1043
17.4	Valid Temporal Units and Temporal Fields	1047
17.5	Period-Based Loop	1062
17.6	More Temporal Arithmetic	1063
17.7	Duration-Based Loop	1071
17.8	Flight Time Information	1080
17.9	Adjusting for DST Crossovers	1084
17.10	Converting to Legacy Date	1089
18.1	Understanding Locales	1099
18.2	The BasicResources Bundle Family (See Also Example 18.3)	1104
18.3	Using Resource Bundles (See Also Example 18.2)	1105
18.4	Locating Resource Bundles	1111
18.5	Rounding Modes	1124
18.6	Using the DecimalFormat class	1126
18.7	Formatting and Parsing with Letter Patterns	1137
18.8	Formatting Compound Messages	1143
18.9	Using Choice Pattern	1147
18.10	Using the ChoiceFormat Class	1149
19.1	Source Code Files for the adviceApp Application	1184

19.2	Selected Source Code Files in the adviceOpen Application	1193
19.3	Specifying the Advice Service Interface	1197
19.4	Implementing the Advice Service Providers	1199
19.5	Implementing the Advice Service Locator	1201
19.6	Implementing of the Advice Service Consumer	1203
20.1	Copying a File Using a Byte Buffer	1237
20.2	Reading and Writing Binary Values	1240
20.3	Copying a File Using a Character Buffer	1245
20.4	Demonstrating Readers and Writers, and Character Encoding	1253
20.5	Changing Passwords	1258
20.6	Object Serialization	1263
20.7	Non-Serializable Objects	1266
20.8	Customized Serialization	1269
20.9	Serialization and Inheritance	1272
20.10	Class Versioning	1276
21.1	Querying Path Objects	1296
21.2	Reading and Writing Text Files	1316
21.3	Reading and Writing Bytes	1319
21.4	Accessing Individual Attributes	1321
21.5	Traversing the Directory Hierarchy	1350
22.1	Implementing the Runnable Interface	1374
22.2	Extending the Thread Class	1376
22.3	Thread States	1384
22.4	Mutual Exclusion	1389
22.5	Interrupt Handling	1394
22.6	Waiting and Notifying	1400
22.7	Joining of Threads	1404
22.8	Thread Termination	1407
22.9	Deadlock	1409
22.10	Livelock	1411
22.11	Starvation	1413
23.1	Executor Lifecycle	1429
23.2	Submitting Tasks and Handling Task Results	1438
23.3	Executing Scheduled One-Shot Tasks	1441
23.4	Executing Scheduled Periodic Tasks	1443
23.5	Task Cancellation	1446
23.6	Filtering Values Using Fork/Join Framework	1449
23.7	Testing Counter Implementations	1451
23.8	Visibility of Shared Data	1454
23.9	Volatile Counter	1456
23.10	Atomic Counter	1458
23.11	Synchronized Counter	1460
23.12	Reentrant Lock Counter	1465
23.13	Reentrant Read-Write Lock Counter	1467
23.14	Cyclic Barrier	1473
23.15	Count-Down Latch	1475

23.16	Serial Access in Synchronized Views of Collections	1479
23.17	Compound Mutually Exclusive Operation	1481
23.18	Concurrent Collections	1488
23.19	Using a Concurrent Map	1493
23.20	Linked Blocking Queue	1500
23.21	Copy-on-Write Array List	1503
24.1	Connecting to the musicDB Database	1521
24.2	Executing a Prepared Statement	1528
24.3	Prepared Statement to Execute INSERT, UPDATE, and DELETE	1529
24.4	Stored Procedures and Functions	1532
24.5	Processing a ResultSet	1537
24.6	Using ResultSet Options	1542
24.7	Discovering Metadata for the Database and ResultSet	1544
24.8	Controlling Transactions	1546
25.1	Using the @Deprecated Annotation	1580
25.2	Using the @SuppressWarnings Annotation	1583
25.3	Using the @SafeVarargs Annotation	1585
25.4	Processing Annotations	1589
25.5	Annotation Processor	1592

*This page intentionally left blank*

# Foreword



In over 25 years of teaching Java, one of the recurring questions I am asked is why one would bother to pursue Java certification. Obviously one answer is that it might be helpful in pursuit of a job, or a better job, as a Java programmer. But whether that is relevant is dependent on the job market. There is another answer that does not depend on the ever-changing ratio of applicants to vacancies, and it is the answer that I have always preferred. That answer is that it is good for anyone who claims to take a skill seriously to do two things. One is to be measured by the standards of one's peers, and the other is to push oneself to learn and improve continuously. Being satisfied with "good enough" must surely be a sign of some degree of mediocrity.

These days Java training is widely available from many sources (including video-based training that I offer). Each has its pros and cons, but what makes Khalid and Vasily's book stand out is probably the thoroughness of the treatment that is offered in these pages. Khalid has a long history teaching at the university level, and good teaching is a very different skill from merely being an expert. Far too much of what we are offered as training or self-study material these days is, on closer analysis, simply an enumeration of facts. Good teaching requires building not just knowledge but understanding, layer upon layer, so that it sticks.

One aspect of teaching and learning that is sadly little understood is how we learn. If you want to get better at lifting heavy weights, most of us would recognize that somewhere in the process, you have to do just that: lift heavy weights. That is essentially true of all human skills. We must practice (and practice diligently) those things at which we wish to succeed. In the case of learning, what we want to be able to do is to recall facts and apply them on demand. That means, quite literally, that we must practice the act of recalling those facts. To this end, you will find extensive review questions between these covers. These will provide you with the opportunity to practice the act of recall and thereby solidify your knowledge and understanding. That way, not only can you expect to pass what is undoubtedly a challenging exam, but you can also expect to become a better Java programmer.

Good luck in your study!

Simon Roberts  
President, Dancing Cloud Services, LLC

*This page intentionally left blank*

# Preface



## Writing This Book

---

Java has been around for over 25 years. During this time, the Java ecosystem has evolved to become the platform of choice for developing software systems, and the Java certification has evolved with it. The goal of this book is twofold: to provide a comprehensive guide, not only for learning Java, and also to nail the Java certification exams.

This book provides extensive coverage for the following Java certifications:

- *Oracle Certified Professional (OCP), Java SE 17 Developer*, and its required exam, *Java SE 17 Developer (1Z0-829)* (for details, see Appendix B)
- *Oracle Certified Professional (OCP), Java SE 11 Developer*, and its required exam, *Java SE 11 Developer (1Z0-819)* (for details, see Appendix C)

The exam syllabus is defined by a set of *exam objectives*. The exam objectives have evolved to become more high-level, with a smattering of specific topics, leaving plenty of room for interpretation as to which topics are covered by an exam objective. The scope of the exam objectives is thus very difficult to define, and entails exhaustive coverage of the Java language features and its many APIs to be on the safe side. This factor is the major cause of the considerable size of this book.

Our approach to writing this book has not changed from the one we employed for our previous books, mainly because it has proved successful. The emphasis remains on analyzing code scenarios, rather than esoteric syntax of individual language constructs. The exam continues to require actual experience with the language, not just mere recitation of facts. We still claim that proficiency in the language is the key to success.

Since the exam emphasizes the core features of Java, this book provides in-depth coverage of topics related to those features. This book is also no different from our previous books in one other important aspect: It is a one-stop guide, providing a mixture of theory and practice that enables readers to prepare for the exam. It can be used to learn Java and to prepare for the exam. After the exam is passed, it can also be used as a handy reference for the Java language.

All elements found in our previous books (e.g., examples, figures, tables, review questions, mock exam questions) can be found in this one as well. We continue to use UML (Unified Modeling Language) extensively to illustrate concepts and language constructs, and all numbered examples continue to be complete Java programs ready for experimentation.

With that, dear reader, we wish you all the best should you decide to go down the path of Java certification. May your loops terminate and your exceptions get caught!

## About This Book

---

This book provides extensive coverage of the core features of the Java programming language and its core application programming interface (API). The book is primarily intended for professionals who want to prepare for the *Java SE 17* or *Java SE 11 Developer Exams*, but it is readily accessible to any programmer who wants to master the language. For both purposes, it provides in-depth coverage of essential features of the language and its core API.

The demand for well-trained and highly skilled Java programmers remains unabated. Oracle offers many Java certifications that professionals can take to validate their skills (see <https://education.oracle.com>). The certification provides members of the IT industry with a standard to use when hiring such professionals, and it allows professionals to turn their Java skills into credentials that are important for career advancement.

The book provides extensive coverage of all the objectives defined by Oracle for both the *Java SE 17* and *Java SE 11 Developer Exams*. The exam objectives are selective, however, and do not include many of the essential features of Java. This book covers many additional topics that every Java programmer should master to be truly proficient. In this regard, the book is a comprehensive primer for learning the Java programming language. After mastering the language by working through this book, the reader can confidently sit for the exams.

This book is *not* a complete reference for Java, as it does not attempt to list every member of every class from the Java SE Platform API documentation. The purpose is not to document the Java SE Platform API. The emphasis is more on the Java programming language features—their syntax and correct usage through code examples—and less on teaching programming techniques.

The book assumes little background in programming. We believe the exam is accessible to any programmer who works through the book. A Java programmer can easily skip over material that is well understood and concentrate on parts that need reinforcing, whereas a programmer new to Java will find the concepts explained from basic principles.

Each topic is explained and discussed thoroughly with examples, and is backed by review questions to reinforce the concepts. The book is not biased toward any particular platform, but provides platform-specific details where necessary.

## Using This Book

The reader can choose a linear or a nonlinear route through the book, depending on their programming background. Non-Java programmers wishing to migrate to Java can read Chapter 1, which provides a short introduction to object-oriented programming concepts, and the procedure for compiling and running Java applications. For those preparing for any of the Developer exams, the book has a separate appendix (Appendix A) providing all the pertinent information on preparing for and taking the exams.

Cross-references are provided where necessary to indicate the relationships among the various constructs of the language. To understand a language construct, all pertinent details are provided where the construct is covered, but in addition, cross-references are provided to indicate its relationship to other constructs. Sometimes it is necessary to postpone discussion of certain aspects of a topic if they depend on concepts that have not yet been covered in the book. A typical example is the consequences of object-oriented programming concepts (e.g., inheritance) on the member declarations that can occur in a class. This approach can result in forward references in the initial chapters of the book.

The table of contents; listings of tables, examples, and figures; and a comprehensive index facilitate locating topics discussed in the book.

In particular, we draw attention to the following features of the book:



### Chapter Topics

Each chapter starts with a short summary of the topics covered in the chapter, pointing out the major concepts that are introduced.

Developer Exam Objectives	
[0.1]	Exam objectives that are covered in each chapter are stated clearly at the beginning of every chapter.
[0.2]	The number in front of the objective identifies the exam objective, as defined by Oracle. The objectives are organized into major sections, detailing the curriculum for the exam.
[0.3]	The objectives for the <i>Java SE 17</i> and <i>Java SE 11 Developer Exams</i> are reproduced verbatim in Appendix B and Appendix C, respectively. These appendices also map each exam objective to relevant chapters and sections in the book.
Supplementary Topics	
•	Supplementary topics are Java topics that are <i>not</i> on the exam per se, but which the candidate is expected to know.
•	Any supplementary topic is listed as a bullet at the beginning of the chapter.



## Review Questions

Review questions are provided after every major topic to test and reinforce the material. The review questions predominantly reflect the kind of multiple-choice questions that can be asked on the actual exam. On the exam, the exact number of answers to choose for each question is explicitly stated. The review questions in this book follow that practice.

Many questions on the actual exam contain code snippets with line numbers to indicate that complete implementation is not provided, and that the necessary missing code to compile and run the code snippets can be assumed. The review questions in this book provide complete code implementations where possible, so that the code can be readily compiled and run.

Annotated answers to the review questions are provided in Appendix D.

### Example 0.1 *Example Source Code*

We encourage readers to experiment with the code examples to reinforce the material in the book. The source code for the examples can be downloaded from the companion book website (see p. xliii), and readily imported into Eclipse IDE 2022-06.

Java code in the book is presented in a monospaced font. Lines of code in the examples or in code snippets are referenced in the text by a number, which is specified by using a single-line comment in the code. For example, in the following code snippet, the call to the method `doSomethingInteresting()` at (1) does something interesting:

```
// ...
doSomethingInteresting();           // (1)
// ...
```

Names of classes and interfaces start with an uppercase letter. Names of packages, variables, and methods start with a lowercase letter. Constants are in all uppercase letters. Interface names begin with the prefix `I`, when it makes sense to distinguish them from class names. Coding conventions are followed, except when we have had to deviate from these conventions in the interest of space or clarity.

### *Mock Exam*

The mock exam for Java SE 17 in Appendix E should be attempted when the reader feels confident about the topics on the exam. It is highly recommended to read Appendix A before attempting the mock exam, as Appendix A contains pertinent information about the questions to expect on the actual exam. Each multiple-choice question in the mock exam explicitly states how many answers are applicable for a given question, as is the case on the actual exam. Annotated answers to the questions in the mock exam are provided in Appendix F.

## Java SE Platform API Documentation

A vertical gray bar is used to highlight methods and fields found in the classes of the Java SE Platform API.

Any explanation following the API information is also similarly highlighted.

To obtain the maximum benefit from using this book in preparing for the *Java SE 17* and *Java SE 11 Developer Exams*, we strongly recommend installing the latest version (Release 17 or newer) of the JDK and its accompanying API documentation. The book focuses solely on Java SE 17 and Java SE 11, and does not acknowledge other versions.

## Book Website

---

This book is backed by a website:

<https://www.mughal.no/jse17ocp/>

Auxiliary material on the website includes the following:

- Source code for all the examples in the book
- Annotated answers to the reviews questions in the book
- Annotated answers to the mock exam in the book
- Table of contents, sample chapters, and index from the book
- Content specific for the *Java SE 17* and *Java SE 11 Developer Exams*
- Errata for the book
- Links to miscellaneous Java resources (e.g., certification, discussion groups, and tools)

Information about the Java Standard Edition (SE) and its documentation can be found at the following website:

[www.oracle.com/technetwork/java/javase/overview/index.html](http://www.oracle.com/technetwork/java/javase/overview/index.html)

The current authoritative technical reference for the Java programming language, *The Java® Language Specification: Java SE 17 Edition*, can be found at this website:

<http://docs.oracle.com/javase/specs/index.html>

## Request for Feedback

---

Considerable effort has been made to ensure the accuracy of the content of this book. All code examples (including code fragments) have been compiled and tested on various platforms. In the final analysis, any errors remaining are the sole responsibility of the principal author.

Any questions, comments, suggestions, and corrections are welcome. Let us know whether the book was helpful (or not) for your purpose. Any feedback is valuable.

The principal author and the co-author can be reached at the following email addresses, respectively:

khalid@mughal.no  
vasiliy.a.strelnikov@oracle.com

## About the Authors

---

### *Khalid A. Mughal*

Khalid A. Mughal is the principal author of this book, primarily responsible for writing the material covering the Java topics. He is also the principal author of three other books on previous versions of the Java certification exam: Java SE 8 OCA (1Z0-808), Java SE 6 (1Z0-851), and SCPJ2 1.4 (CX-310-035).

Khalid is an associate professor emeritus at the Department of Informatics at the University of Bergen, Norway, where he was responsible for designing and implementing various courses in informatics. Over the years, he has taught programming (primarily Java), software engineering (object-oriented system development), databases (data modeling and database management systems), compiler techniques, web application development, and software security courses. For 15 years, he was responsible for developing and running web-based programming courses in Java, which were offered to off-campus students. He has also given numerous courses and seminars at various levels in object-oriented programming and system development using Java and Java-related technologies, both at the University of Bergen and various other universities in Norway and East Africa, and also for the IT industry.

### *Vasily A. Strelnikov*

Vasily Strelnikov is primarily responsible for developing new review questions for the chapters contained in this book.

Vasily is a senior principal OCI (Oracle Cloud Infrastructure) solution specialist, working at Oracle for more than 26 years. He is a co-author of the Java EE 7, Java SE 8, Java SE 11, and Java SE 17 Certification exams. He has designed multiple Java courses that are offered by Oracle: Java SE 17 Programming Complete, Java SE 11 Programming Complete, and Java SE 8 Certification Preparation Seminar. He has also created the Developing Applications for the Java EE 7 Platform training at Oracle. Vasily has over 20 years of experience in Java. He specializes in Java middleware application development and web services.

## Acknowledgments

---

First of all, we would like to acknowledge the contribution that Rolf W. Rasmussen at vizrt made in earlier Java certification books that we wrote together. Some of that contribution still permeates this edition as well.

At Pearson, senior executive editor Greg Doench was once again at the helm. He managed the long process of publishing this book. Senior content producer Julie Nahil was again the in-house contact at Pearson, professionally and efficiently managing production of the book. Audrey Doyle did a truly marvelous job copy-editing the book. She scrutinized every sentence with a meticulous eye—and taught us a thing or two about the English language. The folks at codeMantra did the critical proofreading, endorsing the book with a quality assurance stamp. Rob Mauhar at The CIP Group performed the typesetting wizardry necessary to give the book its professional look. Our sincere thanks to Greg, Julie, Audrey, Rob, the folks at codeMantra, and all those behind the scenes at Pearson, who helped to make this publication see the light of day.

For the technical review of the book, we were lucky to have two Java gurus who agreed to take on the task.

- Mikalai Zaikin is a lead Java developer at IBA Lithuania, and is currently located in Vilnius. He has helped Oracle with development of Java certification exams and has also been a technical reviewer of several Java certification books. He also contributes to the Java Quiz column for Oracle’s Java Magazine in collaboration with Simon Roberts.
- Ankit Garg is currently Vice President at Morgan-Stanley, Bengaluru, India, and a Sheriff (i.e., Moderator) for various Java Forums at [coderanch.com](http://coderanch.com). He is a Java developer with many years’ experience in several Java-related technologies (e.g., Java SE, Java EE, and Spring). He is also a certified Java Programmer and Web Component Developer.

Without doubt, both Mikalai and Ankit have an eye for detail. It is no exaggeration to say that their feedback has been invaluable in improving the quality of this book at all levels. Our most sincere thanks to both Mikalai and Ankit for the many excellent comments and suggestions, and above all, for weeding out numerous pesky errors in the manuscript.

Kristian Berg and Morten Nygaard Åsnes were kind enough to review two chapters each and provide useful feedback, for which we are very grateful.

Great effort has been made to eliminate mistakes and errors in this book. We accept full responsibility for any remaining oversights. We hope that when our diligent readers find any, they will bring them to our attention.

Without family support this edition would still be a fantasy, and we are very grateful to our families for putting up with us when we were burning the midnight oil. Hopefully we can get our family life back now that this book is out the door.

—Khalid A. Mughal  
December 2, 2022.  
Bergen, Norway.

*This page intentionally left blank*



## Chapter Topics

- Understanding the construction of a stream pipeline
- Understanding various aspects of streams: sequential or parallel, ordered or unordered, finite or infinite, and object or numeric
- Creating object streams from various sources; for example, assorted collections, arrays, strings, and I/O classes
- Creating infinite numeric streams using generator functions
- Understanding the various aspects of intermediate stream operations: stream mapping, lazy execution, short-circuit evaluation, and stateless or stateful operations
- Understanding the implications of operation order, and non-interfering and stateless behavioral parameters of intermediate stream operations
- Filtering, skipping, and examining stream elements
- Selecting distinct elements and truncating a stream
- Understanding mapping and flattening a stream
- Sorting stream elements
- Changing the execution mode of a stream and marking a stream as unordered
- Understanding interoperability between stream types
- Understanding the role of the `Optional` class
- How to create, query, filter, map, and flatten optionals
- Using numeric optionals
- Understanding the implication of invoking a terminal operation on a stream
- Applying consumer actions to elicit side effects in a stream
- Using terminal operations to match, find, and count stream elements

- Understanding functional and mutable reduction, both sequential and parallel
- Collecting stream results in lists, sets, and arrays
- Using functional reduction on numeric streams, including statistical operations
- Understanding the role of a collector in stream execution
- Collecting to a collection, list, set, map, and concurrent map
- Using a collector to join strings
- Using collectors that group and partition stream elements
- Using downstream collectors for functional reduction: counting, finding min/max, summing, averaging, and summarizing
- How to implement collectors for customized reduction
- How to use map-reduce, filtering, flat mapping, and finishing adapters for downstream collectors
- Understanding how to build and execute a parallel stream
- Understanding factors that can affect parallel stream execution
- Understanding the importance of benchmarking parallel stream execution

Java SE 17 Developer Exam Objectives	
[6.1] Use Java object and primitive Streams, including lambda expressions implementing functional interfaces, to supply, filter, map, consume, and sort data <ul style="list-style-type: none"> <li>○ Streams are covered in this chapter.</li> <li>○ For lambda expressions implementing functional interfaces, see Chapter 13, p. 673.</li> </ul>	§16.3, p. 884 §16.4, p. 890 §16.5, p. 905 §16.7, p. 946
[6.2] Perform decomposition, concatenation and reduction, and grouping and partitioning on sequential and parallel streams	§16.7, p. 946 §16.8, p. 978 §16.9, p. 1009
Java SE 11 Developer Exam Objectives	
[6.2] Use Java Streams to filter, transform and process data	§16.3, p. 884 §16.4, p. 890 §16.5, p. 905 §16.7, p. 946
[6.3] Perform decomposition and reduction, including grouping and partitioning on sequential and parallel streams	§16.7, p. 946 §16.8, p. 978 §16.9, p. 1009

The Stream API brings a new programming paradigm to Java: a *declarative* way of processing data using *streams*—expressing *what* should be done to the values and not *how* it should be done. More importantly, the API allows programmers to harness the power of multicore architectures for *parallel* processing of data.

We strongly suggest reviewing the following topics which we consider essential prerequisites for learning about streams:

- Functional-style programming (Chapter 13, p. 673); specially, functional interfaces, lambda expressions, method references, and built-in functional interfaces
- Comparing objects (Chapter 14, p. 741); in particular, the `Comparator<E>` functional interface (§14.4, p. 761)

## 16.1 Introduction to Streams

---

A *stream* allows aggregate operations to be performed on a sequence of elements. An *aggregate operation* performs a task on the stream as a whole rather than on an individual element of the stream. In the context of streams, these aggregation operations are called *stream operations*. Such operations utilize behavior parameterization implemented by functional interfaces for actions performed on the stream elements.

Examples of stream operations accepting implementation of functional interfaces include:

- Generating elements of the stream using a `Supplier`
- Converting the elements in the stream according to a mapping defined by a `Function`
- Filtering the elements in the stream according to some criteria defined by a `Predicate`
- Sorting the elements in the stream using a `Comparator`
- Performing actions for each of the elements in the stream with the help of a `Consumer`

Streams can be produced from a variety of sources. Collections and arrays are typical examples of sources for streams. The `Collection<E>` interface and the `Arrays` utility class both provide a `stream()` method that builds a stream from the elements of a collection or an array.

In the loop-based solution below, elements from the `values` list are processed using a `for(:)` loop to test whether a year is after the year 2000. The strings in the list are parsed to a `Year` object before being tested in an `if` statement.

```
// Loop-based solution:  
List<String> values = List.of("2001", "1999", "2021");  
for (String s : values) {  
    Year y = Year.parse(s);  
    if (y.isAfter(Year.of(2000))) {
```

```

        System.out.print(s + " ");           // 2001 2021
    }
}

// Stream-based solution:
List<String> values2 = List.of("2001", "1999", "2021");
values2.stream()                          // (1)
    .map(s -> Year.parse(s))                // (2)
    .filter(y -> y.isAfter(Year.of(2000))) // (3)
    .forEach(y -> System.out.print(y + " ")); // (4) 2001 2021

```

A stream-based solution for the same problem is also presented above. The `stream()` operation at (1) generates a stream based on the elements from the collection. The `map()` operation at (2) parses the string elements to a `Year` object, as defined by the lambda expression that implements the `Function` interface. The `filter()` operation at (3) performs a filtering of the elements in the stream that are after the year 2000, as defined by a lambda expression that implements the `Predicate` interface. The `forEach()` operation at (4) performs an action on each stream element, as defined by a lambda expression that implements the `Consumer` interface.

The loop-based solution specifies how the operations should be performed. The stream-based solution states what operations should be performed, qualified by the implementation of an appropriate functional interface. Stream-based solutions to many problems can be elegant and concise compared to their iteration-based counterparts.

In this chapter we will cover many stream operations in detail, as well as discover other use cases and benefits of using streams.

## 16.2 Running Example: The CD Record Class

---

We will use the `CD` record class and the `Genre` enum type from Example 16.1 in many of the examples throughout this chapter. The classes are intentionally kept simple for the purposes of exposition, but they will suffice in illustrating the vast number of topics covered in this chapter.

The `CD` record class declares fields for the following information about a CD: an artist name, a title, a fixed number of tracks, the year the CD was released, and a musical genre. The compiler provides the relevant get methods to access the fields in a `CD` record, but `boolean` methods are explicitly defined to determine its musical genre. The compiler also provides implementations to override the `toString()`, `hashCode()`, and `equals()` methods from the `Object` class. However, the record class provides its own implementation of the `toString()` method. The record class implements the `Comparable<CD>` interface with the following comparison order for the fields: artist name, title, number of tracks, year released, and musical genre.

It is important to note the static field declarations in the `CD` record class, as they will be used in subsequent examples. These include the static references `cd0`, `cd1`, `cd2`, `cd3`, and `cd4` that refer to five different instances of the `CD` record class. In addition,

the fixed-size list `cdList` contains these five ready-made CDs, as does the array `cdArray`. The output from Example 16.1 shows the state of the CDs in the `cdList`. We recommend consulting this information in order to verify the results from examples presented in this chapter.

The simple enum type `Genre` is used to indicate the style of music on a CD.

**Example 16.1** *The CD Example Classes*

```

.....
// The different genres in music.
public enum Genre {POP, JAZZ, OTHER}
.....

import java.time.Year;
import java.util.Comparator;
import java.util.List;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,
                Year year, Genre genre) implements Comparable<CD> {

    // Additional get methods:
    public boolean isPop() { return this.genre == Genre.POP; }
    public boolean isJazz() { return this.genre == Genre.JAZZ; }
    public boolean isOther() { return this.genre == Genre.OTHER; }

    // Provide own implementation of the toString() method.
    @Override public String toString() {
        return String.format("<%=, \"%s\", %d, %s, %s>",
            this.artist, this.title, this.noOfTracks, this.year, this.genre);
    }

    /** Compare by artist, by title, by number of tracks, by year, and by genre. */
    @Override public int compareTo(CD other) {
        return Comparator.comparing(CD::artist)
            .thenComparing(CD::title)
            .thenComparing(CD::noOfTracks)
            .thenComparing(CD::year)
            .thenComparing(CD::genre)
            .compare(this, other);
    }

    // Some ready-made CDs.
    public static final CD cd0
        = new CD("Jaav", "Java Jive", 8, Year.of(2017), Genre.POP);
    public static final CD cd1
        = new CD("Jaav", "Java Jam", 6, Year.of(2017), Genre.JAZZ);
    public static final CD cd2
        = new CD("Funkies", "Lambda Dancing", 10, Year.of(2018), Genre.POP);
    public static final CD cd3
        = new CD("Genericos", "Keep on Erasing", 8, Year.of(2018), Genre.JAZZ);
    public static final CD cd4
        = new CD("Genericos", "Hot Generics", 10, Year.of(2018), Genre.JAZZ);

```

```

// A fixed-size list of CDs.
public static final List<CD> cdList = List.of(cd0, cd1, cd2, cd3, cd4);

// An array of CDs.
public static final CD[] cdArray = {cd0, cd1, cd2, cd3, cd4};
}
.....
import java.util.List;

public final class CDAdmin {
    public static void main(String[] args) {
        List<CD> cdList = CD.cdList;
        System.out.println("    Artist    Title           No. Year Genre");
        for(int i = 0; i < cdList.size(); ++i) {
            CD cd = cdList.get(i);
            String cdToString = String.format("%-10s%-16s%-4d%-5s%-5s",
                cd.artist(), cd.title(), cd.noOfTracks(),
                cd.year(), cd.genre());
            System.out.printf("cd%d: %s\n", i, cdToString);
        }
    }
}

```

Output from the program:

	Artist	Title	No.	Year	Genre
cd0:	Jaav	Java Jive	8	2017	POP
cd1:	Jaav	Java Jam	6	2017	JAZZ
cd2:	Funkies	Lambda Dancing	10	2018	POP
cd3:	Genericos	Keep on Erasing	8	2018	JAZZ
cd4:	Genericos	Hot Generics	10	2018	JAZZ

.....

## 16.3 Stream Basics

---

In this section we introduce the terminology and the basic concepts required to work with streams, and provide an overview of the Stream API.

An example of an aggregate operation is filtering the elements in the stream according to some criteria, where all the elements of the stream must be examined in order to determine the result. We saw a simple example of filtering a list in the previous section. Figure 16.1a shows an example of a *stream-based* solution for filtering a list of CDs to find all pop music CDs. A stream of CDs is created at (1). The stream is filtered at (2) to find pop music CDs. Each pop music CD that is found is accumulated into a result list at (3). Note how the method calls are chained, which is typical of processing elements in a stream. We will fill in the details as we use Figure 16.1 to introduce the basics of data processing using streams.

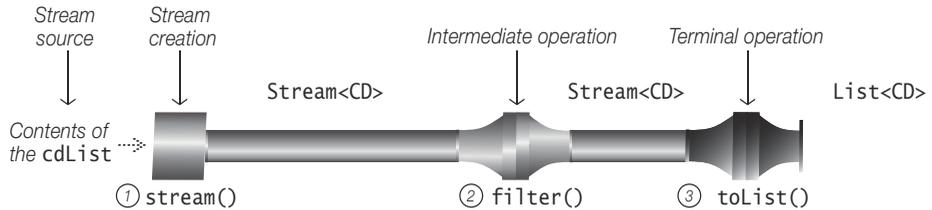
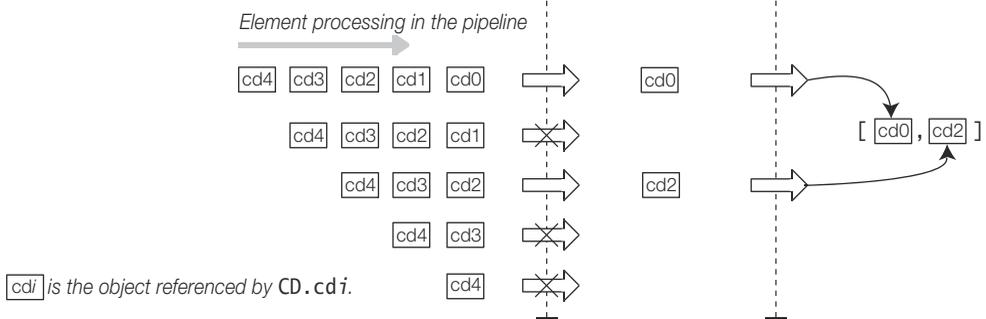
A stream must be built from a *data source* before operations can be performed on its elements. Streams come in two flavors: those that process *object references*, called

**Figure 16.1** *Data Processing with Streams*

```

// Query to create a list of all CDs with pop music.
List<CD> cdList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd3, CD.cd4);
List<CD> popCDs = cdList
  ① .stream()                               // Stream creation.
  ② .filter(CD::isPop)                       // Intermediate operation.
  ③ .toList();                               // Terminal operation.

```

(a) *A query for data processing with streams*(b) *A stream pipeline*(c) *Execution of a stream pipeline*

*object streams*; and those that process numeric values, called *numeric streams*. In Figure 16.1a, the call to the `stream()` method on the list `cdList` creates an object stream of CD references with `cdList` as the data source. Building streams from data sources is explored in §16.4, p. 890.

Stream operations are characterized either as *intermediate operations* (§16.5, p. 905) or *terminal operations* (§16.7, p. 946). In Figure 16.1a, there are two stream operations. The methods `filter()` and `collect()` implement an intermediate operation and a terminal operation, respectively.

An intermediate operation always returns a new stream—that is, it transforms its *input stream* to an *output stream*. In Figure 16.1a, the `filter()` method is called at (2) on its input stream, which is the initial stream of CDs returned by the `stream()` method at (1). The method reference `CD::isPop` passed as an argument to the `filter()` method implements the `Predicate<CD>` functional interface to filter the CDs in the stream. The `filter()` method returns an output stream, which is a stream of pop music CDs.

A terminal operation either causes a side effect or computes a result. The method `collect()` at (3) implements a terminal operation that computes a result (§16.7, p. 964). This method is called on the output stream of the `filter()` operation, which is now the input stream of the terminal operation. Each CD that is determined to be a pop music CD by the `filter()` operation at (2) is accumulated into a result list by the `toList()` method at (3). The `toList()` method (p. 972) creates an empty list and accumulates elements from the input stream—in this case, CDs with pop music.

The code below splits the chain of method calls in Figure 16.1a, but produces the same result. The code explicitly shows the streams that are created and how an operation is invoked on the stream returned by the preceding operation. However, this code is a lot more verbose and not as easy to read as the code in Figure 16.1a—so much so that it is frowned upon by stream aficionados.

```
Stream<CD> stream1 = cdList.stream();           // (1a) Stream creation.
Stream<CD> stream2 = stream1.filter(CD::isPop); // (2a) Intermediate operation.
List<CD> popCDs2 = stream2.toList();          // (3a) Terminal operation.
```

## Composing Stream Pipelines

Stream operations can be chained together to compose a *stream pipeline* in which stream components are specified in the following order:

- An operation on a data source for building the initial stream
- Zero or more intermediate operations to transform one stream into another
- A single mandatory terminal operation in order to execute the pipeline and produce some result or side effect

Composition into a pipeline is possible because stream creation and intermediate operations return a stream, allowing method calls to be chained as we have seen in Figure 16.1a.

The chain of method calls in Figure 16.1a forms the stream pipeline in Figure 16.1b, showing the components of the pipeline. A stream pipeline is analogous to an assembly line, where each operation depends on the result of the previous operation as parts are assembled. In a pipeline, an intermediate operation consumes elements made available by its input stream to produce elements that form its output stream. The terminal operation produces the final result from its input stream. Creating a stream pipeline can be regarded as a fusion of stream operations, where only a single pass of the elements is necessary to process the stream.

Stream operations are typically customized by behavior parameterization that is specified by functional interfaces and implemented by lambda expressions. That is why understanding built-in functional interfaces and writing method references (or their equivalent lambda expressions) is essential. In Figure 16.1a, the `Predicate` argument of the `filter()` operation implements the behavior of the `filter()` operation.

A stream pipeline formulates a query on the elements of a stream created from a data source. It expresses *what* should be done to the stream elements, and not *how*

it should be done—analogous to a database query. One important advantage of composing stream pipelines is that the compiler can freely optimize the operations—for example, for parallel execution—as long as the same result is guaranteed.

## Executing Stream Pipelines

Apart from the fact that an intermediate operation always returns a new stream and a terminal operation never does, another crucial difference between these two kinds of operations is the way in which they are executed. Intermediate operations use *lazy execution*; that is, they are executed on demand. Terminal operations use *eager execution*; that is, they are executed immediately when the terminal operation is invoked on the stream. This means the intermediate operation will never be executed unless a terminal operation is invoked on the output stream of the intermediate operation, whereupon the intermediate operations will start to execute and pull elements from the stream created on the data source.

The execution of a stream pipeline is illustrated in Figure 16.1c. The `stream()` method just returns the initial stream whose data source is `cdList`. The CD objects in `cdList` are processed in the stream pipeline in the same order as in the list, designated as `cd0`, `cd1`, `cd2`, `cd3`, and `cd4`. The way in which elements are successively processed by the operations in the pipeline is shown *horizontally* for each element. The execution of the pipeline only starts when the terminal operation `toList()` is invoked, and proceeds as follows:

1. `cd0` is selected by the `filter()` operation as it is a pop music CD, and the `collect()` operation places it in the list created to accumulate the results.
2. `cd1` is discarded by the `filter()` operation as it is not a pop music CD.
3. `cd2` is selected by the `filter()` operation as it is a pop music CD, and the `collect()` operation places it in the list created to accumulate the results.
4. `cd3` is discarded by the `filter()` operation as it is not a pop music CD.
5. `cd4` is discarded by the `filter()` operation as it is not a pop music CD.

In Figure 16.1c, when the stream is exhausted, execution of the `collect()` terminal operation completes and execution of the pipeline stops. Note that there was only *one* pass over the elements in the stream. From Figure 16.1c, we see that the resulting list contains only `cd0` and `cd2`, which is the result of the query. Printing the resulting `popCDs` list produces the following output:

```
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>]
```

A stream is considered *consumed* once a terminal operation has completed execution. A stream that has been consumed *cannot* be reused, and any attempt to use it will result in a nasty `java.lang.IllegalStateException`.

The code presented in this subsection is shown in Example 16.2.

**Example 16.2** *Data Processing Using Streams*

```

import java.util.List;
import java.util.stream.Stream;

public class StreamPipeLine {
    public static void main(String[] args) {

        List<CD> cdList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd3, CD.cd4);

        // (A) Query to create a list of all CDs with pop music.
        List<CD> popCDs = cdList.stream()           // (1) Stream creation.
            .filter(CD::isPop)                     // (2) Intermediate operation.
            .toList();                             // (3) Terminal operation.
        System.out.println(popCDs);

        // (B) Equivalent to (A).
        Stream<CD> stream1 = cdList.stream();      // (1a) Stream creation.
        Stream<CD> stream2 = stream1.filter(CD::isPop); // (2a) Intermediate operation.
        List<CD> popCDs2 = stream2.toList();      // (3a) Terminal operation.
        System.out.println(popCDs2);
    }
}

```

Output from the program:

```

[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>]
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>]

```

## Comparing Collections and Streams

It is important to understand the distinction between collections and streams. Streams are not collections, and vice versa, but a stream can be created with a collection as the data source (p. 897).

Collections are data structures that can be used to store and retrieve elements. Streams are data structures that do not store their elements, but process them by expressing computations on them through operations like `filter()` and `collect()`.

Typically, operations are provided to add or remove elements from a collection. However, no elements can be added or removed from a stream—that is, streams are immutable. Because of their functional nature, if a stream operation does remove or discard an element in a stream, a new stream is returned with the remaining elements. A stream operation does not mutate its data source.

A collection can be used in the program as long as there is a reference to it. However, a stream cannot be reused once it is consumed. It must be re-created on the data source in order to be reused.

Operations on a collection are executed immediately, whereas streams can define intermediate operations that are executed on demand—that is, by lazy execution.

Collections are iterable, but streams are *not* iterable. Streams do not implement the `Iterable<T>` interface, and therefore, a `for(:)` loop cannot be used to iterate over a stream.

Mechanisms for iteration over a collection are based on an iterator defined by the `Collection` interface, but must be explicitly used in the program to iterate over the elements; this is called *external iteration*. On the other hand, iteration over stream elements is implicitly handled by the API; this is called *internal iteration* and it occurs when the stream operations are executed.

Collections have a finite size, but streams can be unbounded; these are called *infinite streams*. Special stream operations, such as `limit()`, exist to compute with infinite streams.

Some collections, such as lists, allow positional access of their elements with an index. However, this is not possible with streams, as only aggregate operations are permissible.

Note also that streams supported by the Stream API are not the same as those supported by the File I/O APIs (§20.1, p. 1233).

## Overview of API for Data Processing Using Streams

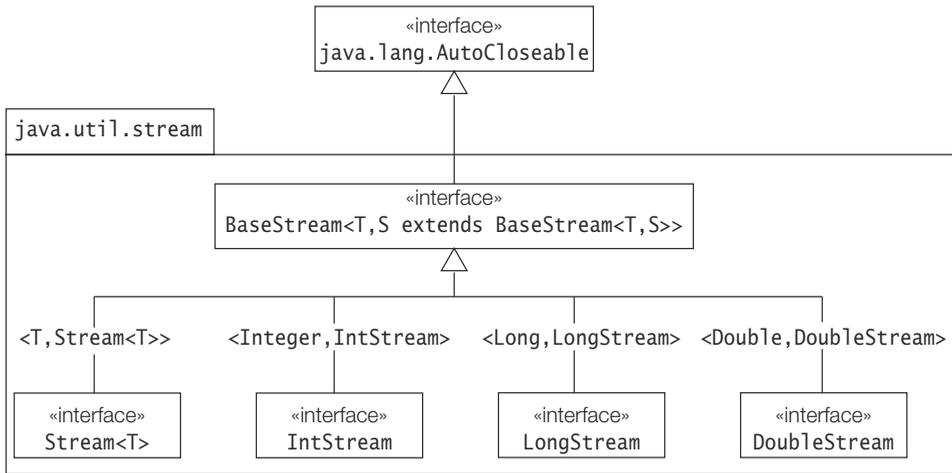
In this subsection we present a brief overview of new interfaces and classes that are introduced in this chapter. We focus mainly on the Stream API in the `java.util.stream` package, but we also discuss utility classes from the `java.util` package.

### *The Stream Interfaces*

Figure 16.2 shows the inheritance hierarchy of the core stream interfaces that are an important part of the Stream API defined in the `java.util.stream` package. The generic interface `Stream<T>` represents *a stream of object references*—that is, *object streams*. The interfaces `IntStream`, `LongStream`, and `DoubleStream` are specializations to *numeric streams* of type `int`, `long`, and `double`, respectively. These interfaces provide the static factory methods for creating streams from various sources (p. 890), and define the intermediate operations (p. 905) and the terminal operations (p. 946) on streams.

The interface `BaseStream` defines the basic functionality offered by all streams. It is recursively parameterized with a stream element type `T` and a subtype `S` of the `BaseStream` interface. For example, the `Stream<T>` interface is a subtype of the parameterized `BaseStream<T, Stream<T>>` interface, and the `IntStream` interface is a subtype of the parameterized `BaseStream<Integer, IntStream>` interface.

All streams implement the `AutoCloseable` interface, meaning they should be closed after use in order to facilitate resource management during execution. However, this is not necessary for the majority of streams. Only resource-backed streams need to be closed—for example, a stream whose data source is a file. Such resources are best managed automatically with the `try-with-resources` statement (§7.7, p. 407).

Figure 16.2 *The Core Stream Interfaces*

### *The Collectors Class*

A collector encapsulates the machinery required to perform a reduction operation (p. 978). The `java.util.stream.Collector` interface defines the functionality that a collector must implement. The `java.util.stream.Collectors` class provides a rich set of predefined collectors for various kinds of reductions.

### *The Optional Classes*

Instances of the `java.util.Optional<T>` class are containers that may or may not contain an object of type `T` (p. 940). An `Optional<T>` instance can be used to represent the absence of a value of type `T` more meaningfully than the `null` value. The numeric analogues are `OptionalInt`, `OptionalLong`, and `OptionalDouble` that can encapsulate an `int`, a `long`, or a `double` value, respectively.

### *The Numeric Summary Statistics Classes*

Instances of the `IntSummaryStatistics`, `LongSummaryStatistics`, and `DoubleSummaryStatistics` classes in the `java.util` package are used by a group of reduction operations to collect summarizing statistics like the count, sum, average, min, and max of the values in a numeric stream of type `int`, `long`, and `double`, respectively (p. 974, p. 1001).

## 16.4 Building Streams

A stream must have a data source. In this section we will explore how streams can be created from various data sources: collections, arrays, specified values, generator functions, strings, and I/O channels, among others.

## Aspects to Consider When Creating Streams

When creating a stream from a data source, certain aspects to consider include whether the stream is:

- Sequential or parallel
- Ordered or unordered
- Finite or infinite
- Object or numeric

### *Sequential or Parallel Stream*

A *sequential stream* is one whose elements are processed sequentially (as in a for loop) when the stream pipeline is executed by a single thread. Figure 16.1 illustrates the execution of a sequential stream, where the stream pipeline is executed by a single thread.

A *parallel stream* is split into multiple substreams that are processed in parallel by multiple instances of the stream pipeline being executed by multiple threads, and their intermediate results combined to create the final result. Parallel streams are discussed in detail later (p. 1009).

The different ways to create a stream on a data source that are illustrated in this section result in a sequential stream. A parallel stream can only be created directly on a collection by invoking the `Collection.parallelStream()` method (p. 897).

The sequential or parallel mode of an existing stream can be modified by calling the `BaseStream.sequential()` and `BaseStream.parallel()` intermediate operations, respectively (p. 933). A stream is executed sequentially or in parallel depending on the execution mode of the stream on which the terminal operation is initiated.

### *Ordered or Unordered Stream*

The *encounter order* of a stream refers to the way in which a stream makes its elements available for processing to an operation in a pipeline. For such data sources as a list, the encounter order of the initial stream is the same as the order of the elements in the list, whereas a stream created with a set of values does not have an encounter order, as the elements of a set are considered to be unordered.

The encounter order of a stream may be changed by an intermediate operation. For example, the `sorted()` operation may impose an encounter order on an unordered stream (p. 929), and the `unordered()` operation may designate a stream unordered (p. 932). Also, some terminal operations might choose to ignore the encounter order; an example is the `forEach()` operation (p. 948).

For ordered sequential streams, an identical result is produced when identical stream pipelines are executed on an identical data source—that is, the execution is deterministic. This guarantee does not hold for unordered sequential streams, as the results produced might be different.

Processing of unordered parallel streams may have better performance than for ordered parallel streams in identical stream pipelines when the ordering constraint is removed, as maintaining the order might carry a performance penalty.

### *Finite or Infinite Stream*

The size of a stream can be finite or infinite depending on how the stream is created. The `generate()` and `iterate()` methods of the core stream interfaces create streams with an infinite number of elements (p. 894). Such a stream is said to be *unbounded*. The overloaded `ints()`, `longs()`, and `doubles()` methods of the `java.util.Random` class create streams with an effectively unlimited number of pseudorandom values (p. 900). An infinite stream must be truncated before the terminal operation is initiated; otherwise, the stream pipeline will never terminate (p. 917).

### *Object or Numeric Stream*

The interface `Stream<T>` defines the contract for streams of object references—that is, *object streams*. The specialized interfaces `IntStream`, `LongStream`, and `DoubleStream` represent streams of `int`, `long`, and `double` values, respectively—that is, *numeric streams*. The various ways to create streams discussed here will always result in a stream whose element type is either a reference type or a numeric type (`int`, `long`, or `double`). Conversion between these stream types is discussed in §16.5, p. 934.

Table 16.1, p. 904, summarizes selected methods for building streams from various data sources.

The following static factory methods for building streams are defined in the `Stream<T>` class. Counterparts to these methods are also provided by the `IntStream`, `LongStream`, and `DoubleStream` interfaces for creating numeric streams, unless otherwise noted:

```
static <T> Stream<T> empty()
static <T> Stream<T> of(T t)
static <T> Stream<T> of(T... varargs)
```

The first method creates an *empty* stream—that is, a stream that has no elements.

The second method creates a *singleton* stream—that is, a stream that has a single element.

The third method creates a *finite sequential ordered* stream whose elements are the values specified by the variable arity parameter `varargs`.

The second and last methods throw a `NullPointerException` if the argument is `null`.

```
static <T> Stream<T> ofNullable(T t)           Only in the Stream<T> interface.
Creates a singleton stream that has a single element, if the argument t is non-
null; otherwise, it returns an empty stream.
```

```
static <T> Stream<T> generate(Supplier<T> supplier)
```

Creates an *infinite sequential unordered* stream where each element is generated by the specified supplier. Typically, this is used for constant streams and streams with random elements.

```
static <T> Stream<T> iterate(T s, UnaryOperator<T> uop)
```

Creates an *infinite sequential ordered* stream produced by the iterative application of the function `uop` to an initial element seed `s`, producing a stream consisting of `s`, `e1=uop.apply(s)`, `e2=uop.apply(e1)`, and so on; that is, `uop` is applied to the previous element.

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

Creates a stream whose elements are all elements of the first stream followed by all elements of the second stream.

The resulting stream is ordered only if both input streams are ordered. If either input stream is parallel, the resulting stream is parallel. As one would expect, the resulting stream is finite only if both input streams are finite.

## The Empty Stream

An *empty stream* can be obtained by calling the `empty()` method of the core stream interfaces. As the name implies, such a stream has no elements.

```
Stream<CD> cdStream = Stream.empty();           // Empty stream of CD.
System.out.println("Count: " + cdStream.count()); // Count: 0
IntStream iStream = IntStream.empty();         // Empty stream of int.
System.out.println("Count: " + iStream.count()); // Count: 0
```

The `count()` method is a terminal operation in the `Stream<T>` interface (p. 953). It returns the number of elements processed through the stream pipeline.

Using the `null` value to indicate that a stream is empty may result in a `NullPointerException`. Therefore, using an explicit empty stream is highly recommended.

## Streams from Specified Values

The two overloaded `of()` methods in the core stream interfaces create finite sequential ordered streams from data values that are specified as arguments.

In the code below, the single-argument `of()` method is called at (1), and the variable arity `of()` method is called at (2), both creating a stream of element type `CD`. The size of the streams created at (1) and (2) is 1 and 3, respectively. The stream pipeline comprising (3) and (4) filters the pop music CDs and prints their title at (4). The `forEach()` terminal operation at (4) applies its `Consumer` action to each pop music `CD`.

```
// From specified objects.
Stream<CD> cdStream1 = Stream.of(CD.cd0);           // (1) Single-arg call.
Stream<CD> cdStream2 = Stream.of(CD.cd0, CD.cd1, CD.cd2); // (2) Varargs call.
cdStream2.filter(CD::isPop)                       // (3)
    .forEach(cd -> System.out.println(cd.title())); // (4)
```

The code below shows examples of using numeric values to create streams. The values specified at (1) and (2) are autoboxed to create a stream of objects. The declaration statements at (3) and (4) avoid the overhead of autoboxing when streams of numeric values are created. However, at (4), an implicit numeric conversion to double is applied to the non-double values.

```
// From specified numeric values.
Stream<Integer> integerStream1 = Stream.of(2017, 2018, 2019); // (1)
Stream<? extends Number> numStream = Stream.of(100, 3.14D, 5050L); // (2)
IntStream intStream1 = IntStream.of(2017, 2018, 2019); // (3)
DoubleStream doubleStream = DoubleStream.of(100, 3.14D, 5050L); // (4)
```

The variable arity of() method can be used to create a stream whose source is an array. Equivalently, the overloaded Arrays.stream() method can be used for the same purpose. In all cases below, the size of the stream is the same as the size of the array, except at (7). An int array is an object that is passed to the single-argument Stream.of() method (creating a Stream<int[]>), and not the variable arity Stream.of() method. The int array is, however, passed to the variable arity IntStream.of() method at (8). Creating a stream from a numeric array is safer with the numeric stream interfaces or the Arrays.stream() method than the Stream.of() method.

```
// From an array of CDs.
Stream<CD> cdStream3 = Stream.of(CD.cdArray); // (1)
Stream<CD> cdStream4 = Arrays.stream(CD.cdArray); // (2)

// From an array of Integer.
Integer[] integerArray = {2017, 2018, 2019}; // (3)
Stream<Integer> integerStream2 = Stream.of(integerArray); // (4)
Stream<Integer> integerStream3 = Arrays.stream(integerArray); // (5)

// From an array of int.
int[] intArray = {2017, 2018, 2019}; // (6)
Stream<int[]> intArrayStream = Stream.of(intArray); // (7) Size is 1.
IntStream intStream2 = IntStream.of(intArray); // (8) Size is 3.
IntStream intStream3 = Arrays.stream(intArray); // (9) Size is 3.
```

The Stream.of() methods throw a NullPointerException if the argument is null. The ofNullable() method, on the other hand, returns an empty stream if this is the case; otherwise, it returns a singleton stream.

## Using Generator Functions to Build Infinite Streams

The generate() and iterate() methods of the core stream interfaces can be used to create infinite sequential streams that are unordered or ordered, respectively.

Infinite streams need to be truncated explicitly in order for the terminal operation to complete execution, or the operation will not terminate. Some stateful intermediate operations must process all elements of the streams in order to produce their results—for example, the sort() intermediate operation (p. 929) and the reduce() terminal operation (p. 955). The limit(maxSize) intermediate operation can be used to limit the number of elements that are available for processing from a stream (p. 917).

## Generate

The `generate()` method accepts a *supplier* that generates the elements of the infinite stream.

```
IntSupplier supplier = () -> (int) (6.0 * Math.random()) + 1; // (1)
IntStream diceStream = IntStream.generate(supplier);           // (2)
diceStream.limit(5)                                           // (3)
    .forEach(i -> System.out.print(i + " "));                 // (4) 2 4 5 2 6
```

The `IntSupplier` at (1) generates a number between 1 and 6 to simulate a dice throw every time it is executed. The supplier is passed to the `generate()` method at (2) to create an infinite unordered `IntStream` whose values simulate throwing a dice. In the pipeline comprising (3) and (4), the number of values in the `IntStream` is limited to 5 at (3) by the `limit()` intermediate operation, and the value of each dice throw is printed by the `forEach()` terminal operation at (4). We can expect five values between 1 and 6 to be printed when the pipeline is executed.

## Iterate

The `iterate()` method accepts a *seed* value and a *unary operator*. The method generates the elements of the infinite ordered stream *iteratively*: It applies the operator to the previous element to generate the next element, where the first element is the seed value.

In the code below, the seed value of 1 is passed to the `iterate()` method at (2), together with the unary operator `uop` defined at (1) that increments its argument by 2. The first element is 1 and the second element is the result of the unary operator applied to 1, and so on. The `limit()` operation limits the stream to five values. We can expect the `forEach()` operation to print the first five odd numbers.

```
IntUnaryOperator uop = n -> n + 2; // (1)
IntStream oddNums = IntStream.iterate(1, uop); // (2)
oddNums.limit(5)
    .forEach(i -> System.out.print(i + " ")); // 1 3 5 7 9
```

The following stream pipeline will really go bananas if the stream is not truncated by the `limit()` operation:

```
Stream.iterate("ba", b -> b + "na")
    .limit(5)
    .forEach(System.out::println);
```

## Concatenating Streams

The `concat()` method creates a resulting stream where the elements from the first argument stream are followed by the elements from the second argument stream. The code below illustrates this operation for two unordered sequential streams. Two sets are created at (1) and (2) based on lists of strings that are passed to the set constructors. The two streams created at (3) and (4) are unordered, since they are created from sets (p. 897). These unordered streams are passed to the `concat()`

method at (5). The resulting stream is processed in the pipeline comprising (5) and (6). The `forEachOrdered()` operation at (6) respects the encounter order of the stream if it has one—that is, if it is ordered (p. 948). The output confirms that the resulting stream is unordered.

```

Set<String> strSet1                                // (1)
    = Set.of("All", " objects", " are", " equal");
Set<String> strSet2                                // (2)
    = Set.of(" but", " some", " are", " more", " equal", " than", " others.");
Stream<String> unorderedStream1 = strSet1.stream(); // (3)
Stream<String> unorderedStream2 = strSet2.stream(); // (4)
Stream.concat(unorderedStream1, unorderedStream2) // (5)
    .forEachOrdered(System.out::print);           // (6)
// objectsAll equal are some are others. than equal more but

```

The resulting stream is ordered if both argument streams are ordered. The code below illustrates this operation for two ordered sequential streams. The two streams created at (1) and (2) below are ordered. The ordering is given by the specification order of the strings as arguments to the `Stream.of()` method. These ordered streams are passed to the `concat()` method at (3). The resulting stream is processed in the pipeline comprising (3) and (4). The output confirms that the resulting stream is ordered.

```

Stream<String> orderedStream1 = Stream.of("All", " objects", // (1)
    " are", " equal");
Stream<String> orderedStream2 = Stream.of(" but", " some", " are", " more", // (2)
    " equal", " than", " others.");
Stream.concat(orderedStream1, orderedStream2) // (3)
    .forEachOrdered(System.out::print); // (4)
// All objects are equal but some are more equal than others.

```

As far as the mode of the resulting stream is concerned, it is parallel if at least one of the constituent streams is parallel. The code below illustrates this behavior.

The `parallel()` intermediate operation used at (1) returns a possibly parallel stream (p. 933). The call to the `isParallel()` method confirms this at (2). We pass one parallel stream and one sequential stream to the `concat()` method at (3). The call to the `isParallel()` method at (4) confirms that the resulting stream is parallel. The print-out from (5) shows that it is also unordered. Note that new streams are created on the sets `strSet1` and `strSet2` at (1) and (3), respectively, as we cannot reuse the streams that were created earlier and consumed.

```

Stream<String> pStream1 = strSet1.stream().parallel(); // (1)
System.out.println("pStream1 is parallel: " + pStream1.isParallel()); // (2) true
Stream<String> rStream = Stream.concat(pStream1, strSet2.stream()); // (3)
System.out.println("rStream is parallel: " + pStream1.isParallel()); // (4) true
rStream.forEachOrdered(System.out::print); // (5)
// objectsAll equal are some are others. than equal more but

```



```

        .stream()                // (3)
        .count();              // (4) 4

```

In the examples in this subsection, the call to the `stream()` method can be replaced by a call to the `parallelStream()` method. The stream will then execute in parallel, without the need for any additional synchronization code (p. 1009).

## Streams from Arrays

We have seen examples of creating streams from arrays when discussing the variable arity of `()` method of the stream interfaces and the overloaded `Arrays.stream()` methods earlier in the chapter (p. 893). The sequential stream created from an array has the same order as the positional order of the elements in the array. As far as numeric streams are concerned, only an `int`, `long`, or `double` array can act as the data source of such a stream.

The code below illustrates creating a stream based on a subarray that is given by the half-open interval specified as an argument to the `Array.stream()` method, as shown at (1). The stream pipeline at (2) calculates the length of the subarray.

```

Stream<CD> cdStream = Arrays.stream(cdArray, 1, 4);    // (1)
long noOfElements = cdStream.count();                // (2) 3

```

The following overloaded static methods for building sequential ordered streams from arrays are defined in the `java.util.Arrays` class:

```

static <T> Stream<T> stream(T[] array)
static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)

```

Create a *finite sequential ordered* `Stream<T>` with the specified array as its source. The stream created by the second method comprises the range of values given by the specified half-open interval.

```

static NumTypeStream stream(numtype[] array)
static NumTypeStream stream(numtype[] array,
                             int startInclusive, int endExclusive)

```

`NumType` is `Int`, `Long`, or `Double`, and the corresponding `numtype` is `int`, `long`, or `double`.

Create a *finite sequential ordered* `NumTypeStream` (which is either `IntStream`, `LongStream`, or `DoubleStream`) with the specified array as its source. The stream created by the second method comprises the range of primitive values given by the specified half-open interval.

## Building a Numeric Stream with a Range

The overloaded methods `range()` and `rangeClosed()` can be used to create *finite ordered* streams of integer values based on a range that can be *half-open* or *closed*, respectively. The increment size is always 1.

The following static factory methods for building numeric streams are defined only in the `IntStream` and `LongStream` interfaces in the `java.util.stream` package.

```
static NumTypeStream range(numtype startInclusive, numtype endExclusive)
static NumTypeStream rangeClosed(numtype startInclusive,
                                  numtype endInclusive)
```

*NumType* is *Int* or *Long*, and the corresponding *numtype* is *int* or *long*.

Both methods return a *finite sequential ordered NumTypeStream* whose elements are a sequence of numbers, where the first number in the stream is the start value of the range *startInclusive* and increment length of the sequence is 1. For a *half-open interval*, as in the first method, the end value of the range *endExclusive* is excluded. For a *closed interval*, as in the second method, the end value of the range *endInclusive* is included.

The `range(startInclusive, endExclusive)` method is equivalent to the following `for(;;)` loop:

```
for (int i = startInclusive; i < endExclusive; i++) {
    // Loop body.
}
```

When processing with ranges of integer values, the `range()` methods should also be considered on par with the `for(;;)` loop.

The stream pipeline below prints all the elements in the `CD` array in reverse. Note that no terminating condition or increment expression is specified. As range values are always in increasing order, a simple adjustment can be done to reverse their order.

```
IntStream.range(0, CD.cdArray.length) // (1)
    .forEach(i -> System.out.println(cdArray[CD.cdArray.length - 1 - i]));
```

The following example counts the numbers that are divisible by a specified divisor in a given range of values.

```
int divisor = 5;
int start = 2000, end = 3000;
long divisibles = IntStream
    .rangeClosed(start, end) // (1)
    .filter(number -> number % divisor == 0) // (2)
    .count(); // (3)
System.out.println(divisibles); // 201
```

The next example creates an `int` array that is filled with increment values specified by the range at (1) below. The `toArray()` method is a terminal operation that creates an array of the appropriate type and populates it with the values in the stream (p. 971).

```
int first = 10, len = 8;
int[] intArray = IntStream.range(first, first + len).toArray(); // (1)
System.out.println(intArray.length + ": " + Arrays.toString(intArray));
//8: [10, 11, 12, 13, 14, 15, 16, 17]
```

The example below shows usage of two nested ranges to print the multiplication tables. The inner `arrange` is executed 10 times for each value in the outer range.

```

IntStream.rangeClosed(1, 10)                // Outer range.
    .forEach(i -> IntStream.rangeClosed(1, 10) // Inner range.
        .forEach(j -> System.out.printf("%2d * %2d = %2d%n",
            i, j, i * j)));
}

```

We cordially invite the inquisitive reader to code the above examples in the imperative style using explicit loops. Which way is better is not always that clear-cut.

## Numeric Streams Using the Random Class

The following methods for building *numeric unordered* streams are defined in the `java.util.Random` class:

*NumType* is `Int`, `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`. The corresponding overloaded *numtypes()* methods are `ints()`, `longs()`, and `doubles()`.

```

NumTypeStream numtypes()
NumTypeStream numtypes(numtype randomNumberOrigin,
    numtype randomNumberBound)
NumTypeStream numtypes(long streamSize)
NumTypeStream numtypes(long streamSize, numtype randomNumberOrigin,
    numtype randomNumberBound)

```

The first two methods generate an *effectively unlimited sequential unordered* stream of pseudorandom *numtype* values. For the zero-argument `doubles()` method, the double values are between 0.0 (inclusive) and 1.0 (exclusive). For the second method, the *numtype* values generated are in the *half-open* interval defined by the origin and the bound values.

The last two methods generate a *finite sequential unordered* stream of pseudorandom *numtype* values, where the length of the stream is limited by the specified `streamSize` parameter.

The examples below illustrate using a *pseudorandom number generator* (PRNG) to create numeric streams. The same PRNG can be used to create multiple streams. The PRNG created at (1) will be used in the examples below.

```

Random rng = new Random();                // (1)

```

The `int` stream created at (2) is an *effectively unlimited unordered* stream of `int` values. The size of the stream is limited to 3 by the `limit()` operation. However, at (3), the maximum size of the stream is specified in the argument to the `ints()` method. The values in both streams at (2) and (3) can be any random `int` values. The contents of the array constructed in the examples will, of course, vary.

```

IntStream iStream = rng.ints();            // (2) Unlimited, any int value
int[] intArray = iStream.limit(3).toArray(); // Limits size to 3
//[[-1170441471, 1070948914, 264046613]

```

```
intArray = rng.ints(3).toArray();           // (3) Size 3, any int value
//[1011448344, -974832344, 816809715]
```

The unlimited unordered stream created at (4) simulates the dice throw we implemented earlier using the `generate()` method (p. 895). The values are between 1 and 6, inclusive. The `limit()` method must be used explicitly to limit the stream. The finite unordered stream created at (5) incorporates the size and the value range.

```
intArray = rng.ints(1, 7)                   // (4) Unlimited, [1, 6]
                .limit(3)                   // Limits size to 3
                .toArray();                 // [5, 2, 4]

intArray = rng.ints(3, 1, 7)                // (5) Size 3, [1, 6]
                .toArray();                 // [1, 4, 6]
```

The zero-argument `doubles()` method and the single-argument `doubles(streamSize)` method generate an unlimited and a limited unordered stream, respectively, whose values are between 0.0 and 1.0 (exclusive).

```
DoubleStream dStream = rng.doubles(3);      // (6) Size 3, [0.0, 1.0]
double[] dArray = dStream.toArray();
//[0.9333794789872794, 0.7037326827186609, 0.2839257522887708]
```

## Streams from a CharSequence

The `CharSequence.chars()` method creates a *finite sequential ordered* `IntStream` from a sequence of char values. The `IntStream` must be transformed to a `Stream<Character>` in order to handle the values as `Characters`. The `IntStream.mapToObj()` method can be used for this purpose, as shown at (2). A cast is necessary at (2) in order to convert an int value to a char value which is autoboxed in a `Character`. Conversion between streams is discussed in §16.5, p. 934.

```
String strSource = "banananana";
IntStream iStream = strSource.chars();      // (1)
iStream.forEach(i -> System.out.print(i + " ")); // Prints ints.
// 98 97 110 97 110 97 110 97 110 97

strSource.chars()
    .mapToObj(i -> (char)i)                 // (2) Stream<Character>
    .forEach(System.out::print);           // Prints chars.
// banananana
```

The following default method for building `IntStreams` from a sequence of char values (e.g., `String` and `StringBuilder`) is defined in the `java.lang.CharSequence` interface (§8.4, p. 444):

```
default IntStream chars()
Creates a finite sequential ordered stream of int values by zero-extending the
char values in this sequence.
```

## Streams from a String

The following method of the `String` class can be used to extract *text lines* from a string:

```
Stream<String> lines()
Returns a stream of lines extracted from this string, separated by line terminators.
```

In the code below, the string at (1) contains three text lines separated by the line terminator (`\n`). A stream of element type `String` is created using this string as the source at (2). Each line containing the word "mistakes" in this stream is printed at (3).

```
String inputLines = "Wise men learn from their mistakes.\n" // (1)
                  + "But wiser men learn from the mistakes of others.\n"
                  + "And fools just carry on.";
Stream<String> lStream = inputLines.lines(); // (2)
lStream.filter(l -> l.contains("mistakes")).forEach(System.out::println); // (3)
```

Output from the code:

```
Wise men learn from their mistakes.
But wiser men learn from the mistakes of others.
```

## Streams from a BufferedReader

A `BufferedReader` allows contents of a text file to be read as lines. A *line* is a sequence of characters terminated by a *line terminator sequence*. Details of using a `BufferedReader` are covered in §20.3, p. 1251. A simple example of creating streams on text files using a `BufferedReader` is presented below.

At (1) and (2) in the header of the try-with-resources statement (§7.7, p. 407), a `BufferedReader` is created to read lines from a given file, and a stream of `String` is created at (3) by the `lines()` method provided by the `BufferedReader` class. These declarations are permissible since both the buffered reader and the stream are `AutoCloseable`. Both will be automatically closed after the try block completes execution. A terminal operation is initiated at (4) on this stream to count the number of lines in the file. Of course, each line from the stream can be processed depending on the problem at hand.

```
try ( FileReader fReader = new FileReader("CD_Data.txt"); // (1)
      BufferedReader bReader = new BufferedReader(fReader); // (2)
      Stream<String> lStream = bReader.lines() ) { // (3)
    System.out.println("Number of lines: " + lStream.count()); // (4) 13
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

The following method for building a `Stream<String>` from a text file is defined in the `java.io.BufferedReader` class:

```
Stream<String> lines()
```

Returns a *finite sequential ordered* Stream of Strings, where the elements are text lines read by this `BufferedReader`.

The reader position in the file is not guaranteed after the stream terminal operation completes.

The result of the terminal stream operation is undefined if the reader is operated upon during the execution of this operation.

Any operation on a Stream returned by a `BufferedReader` that has already been closed will throw an `UncheckedIOException`.

## Streams from Factory Methods in the Files Class

A detailed discussion of the NIO2 File API that provides the classes for creating the various streams for reading files, finding files, and walking directories in the file system can be found in Chapter 21, p. 1285.

Analogous to the `lines()` method in the `BufferedReader` class, a static method with the same name is provided by the `java.nio.file.Files` class that creates a stream for reading the file content as lines.

In the example below, a `Path` is created at (1) to represent a file on the file system. A stream is created to read lines from the path at (2) in the header of the `try-with-resources` statement (§7.7, p. 407). As streams are `AutoCloseable`, such a stream is automatically closed after the try block completes execution. As no character set is specified, bytes from the file are decoded into characters using the UTF-8 charset. A terminal operation is initiated at (3) on this stream to count the number of lines in the file. Again, each line in the stream can be processed as desired.

```
Path path = Paths.get("CD_Data.txt"); // (1)
try (Stream<String> lStream = Files.lines(path)) { // (2)
    System.out.println("Number of lines: " + lStream.count()); // (3) 13
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

The following static methods for building a `Stream<String>` from a text file are defined in the `java.nio.file.Files` class:

```
static Stream<String> lines(Path path)
static Stream<String> lines(Path path, Charset cs)
```

Return a *finite sequential ordered* Stream of String, where the elements are text lines read from a file given by the specified path. The first method decodes the bytes into characters using the UTF-8 charset. The charset to use can be explicitly specified, as in the second method.

## Summary of Stream Building Methods

Selected methods for building streams from various data sources are listed in Table 16.1. The first column lists the method names and the reference type that provides them. For brevity, the parameters of the methods are omitted. Note that some methods are overloaded. The prefix *NumType* stands for *Int*, *Long*, or *Double*. A reference is also provided where details about the method can be found. The remaining four columns indicate various aspects of a stream: the type of stream returned by a method, whether the stream is finite or infinite, whether it is sequential or parallel, and whether it is ordered or unordered (p. 891).

**Table 16.1** Summary of Stream Building Methods

Method	Returned stream type	Finite/ Infinite	Sequential/ Parallel	Ordered/ Unordered
<code>Stream.empty()</code> <code>NumTypeStream.empty()</code> (p. 893)	<code>Stream&lt;T&gt;</code> <code>NumTypeStream</code>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>
<code>Stream.of()</code> <code>Stream.ofNullable()</code> <code>NumTypeStream.of()</code> (p. 893)	<code>Stream&lt;T&gt;</code> <code>Stream&lt;T&gt;</code> <code>NumTypeStream</code>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>
<code>Stream.generate()</code> <code>NumTypeStream.generate()</code> (p. 895)	<code>Stream&lt;T&gt;</code> <code>NumTypeStream</code>	<i>Infinite</i>	<i>Sequential</i>	<i>Unordered</i>
<code>Stream.iterate()</code> <code>NumTypeStream.iterate</code> (p. 895)	<code>Stream&lt;T&gt;</code> <code>NumTypeStream</code>	<i>Infinite</i>	<i>Sequential</i>	<i>Ordered</i>
<code>Stream.concat()</code> <code>NumTypeStream.concat()</code> (p. 895)	<code>Stream&lt;T&gt;</code> <code>NumTypeStream</code>	<i>Finite if both finite</i>	<i>Parallel if either parallel</i>	<i>Ordered if both ordered</i>
<code>Collection.stream()</code> (p. 897)	<code>Stream&lt;T&gt;</code>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered if collection ordered</i>
<code>Collection.parallelStream()</code> (p. 897)	<code>Stream&lt;T&gt;</code>	<i>Finite</i>	<i>Parallel</i>	<i>Ordered if collection ordered</i>
<code>Arrays.stream()</code> (p. 898)	<code>Stream&lt;T&gt;</code> <code>NumTypeStream</code>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>
<code>IntStream.range()</code> <code>IntStream.rangeClosed()</code> <code>LongStream.range()</code> <code>LongStream.rangeClosed()</code> (p. 898)	<code>IntStream</code> <code>IntStream</code> <code>LongStream</code> <code>LongStream</code>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>

**Table 16.1** *Summary of Stream Building Methods (Continued)*

Method	Returned stream type	Finite/ Infinite	Sequential/ Parallel	Ordered/ Unordered
Random.ints() Random.longs() Random.doubles() (p. 900)	IntStream LongStream DoubleStream	<i>Finite or infinite, depending on parameters</i>	<i>Sequential</i>	<i>Unordered</i>
CharSequence.chars() (p. 901)	IntStream	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>
String.lines() (p. 902)	Stream<String>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>
BufferedReader.lines() (p. 902)	Stream<String>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>
Files.lines() (p. 903)	Stream<String>	<i>Finite</i>	<i>Sequential</i>	<i>Ordered</i>

## 16.5 Intermediate Stream Operations

A stream pipeline is composed of stream operations. The stream operations process the elements of the stream to produce some result. After the creation of the initial stream, the elements of the stream are processed by zero or more intermediate operations before the mandatory terminal operation reduces the elements to some final result. The initial stream can undergo several *stream transformations* (technically called *mappings*) by the intermediate operations as the elements are processed through the pipeline.

Intermediate operations *map* a stream to a new stream, and the terminal operation *reduces* the final stream to some result. Because of the nature of the task they perform, the operations in a stream pipeline are also called *map-reduce transformations*.

### Aspects of Streams, Revisited

We now take a closer look at the following aspects pertaining to streams:

- Stream mapping
- Lazy execution
- Short-circuit evaluation
- Stateless and stateful operations
- Order of intermediate operations
- Non-interfering and stateless behavioral parameters of stream operations

Table 16.3, p. 938, summarizes certain aspects of each intermediate operation. Table 16.4, p. 939, summarizes the intermediate operations provided by the Stream API.

### *Stream Mapping*

Each intermediate operation returns a new stream—that is, it maps the elements of its input stream to an output stream. Intermediate operations can thus be easily recognized. Having a clear idea of the *type* of the new stream an intermediate operation should produce aids in customizing the operation with an appropriate implementation of its behavioral parameters. Typically, these behavioral parameters are functional interfaces.

Because intermediate operations return a new stream, calls to methods of intermediate operations can be *chained*, so much so that code written in this *method chaining* style has become a distinct hallmark of expressing queries with streams.

In Example 16.3, the stream pipeline represents the query to create a list with titles of pop music CDs in a given list of CDs. Stream mapping is illustrated at (1). The initial stream of CDs (`Stream<CD>`) is first transformed by an intermediate operation (`filter()`) to yield a new stream that has only pop music CDs (`Stream<CD>`), and this stream is then transformed to a stream of CD titles (`Stream<String>`) by a second intermediate operation (`map()`, p. 921). The stream of CD titles is reduced to the desired result (`List<CD>`) by the terminal operation (`collect()`).

In summary, the *type of the output stream* returned by an intermediate operation need not be the same as the *type of its input stream*.

#### Example 16.3 *Stream Mapping and Loop Fusion*

```

import java.util.List;

public class StreamOps {
    public static void main(String[] args) {

        // Query: Create a list with titles of pop music CDs.

        // (1) Stream Mapping:
        List<CD> cdList1 = CD.cdList;
        List<String> popCDs1 = cdList1
            .stream()                // Initial stream:      Stream<CD>
            .filter(CD::isPop)       // Intermediate operation: Stream<CD>
            .map(CD::title)          // Intermediate operation: Stream<String>
            .toList();               // Terminal operation: List<String>
        System.out.println("Pop music CDs: " + popCDs1);
        System.out.println();

        // (2) Lazy Evaluation:
        List<CD> cdList2 = CD.cdList;
        List<String> popCDs2 = cdList2
            .stream()                // Initial stream:      Stream<CD>
    
```

```

        .filter(cd -> {
            // Intermediate operation: Stream<CD>
            System.out.println("Filtering: " + cd // (3)
                + (cd.isPop() ? " is pop CD." : " is not pop CD."));
            return cd.isPop();
        })
        .map(cd -> {
            // Intermediate operation: Stream<String>
            System.out.println("Mapping: " + cd.title()); // (4)
            return cd.title();
        })
        .toList(); // Terminal operation: List<String>
    System.out.println("Pop music CDs: " + popCDs2);
}
}

```

Output from the program:

```

Pop music CDs: [Java Jive, Lambda Dancing]

Filtering: <Jaav, "Java Jive", 8, 2017, POP> is pop CD.
Mapping: Java Jive
Filtering: <Jaav, "Java Jam", 6, 2017, JAZZ> is not pop CD.
Filtering: <Funkies, "Lambda Dancing", 10, 2018, POP> is pop CD.
Mapping: Lambda Dancing
Filtering: <Genericos, "Keep on Erasing", 8, 2018, JAZZ> is not pop CD.
Filtering: <Genericos, "Hot Generics", 10, 2018, JAZZ> is not pop CD.
Pop music CDs: [Java Jive, Lambda Dancing]

```

### Lazy Execution

A stream pipeline does not execute until a terminal operation is invoked. In other words, its intermediate operations do not start processing until their results are needed by the terminal operation. Intermediate operations are thus *lazy*, in contrast to the terminal operation, which is *eager* and executes when it is invoked.

An intermediate operation is *not* performed on all elements of the stream before performing the next operation on all elements resulting from the previous stream. Rather, the intermediate operations are performed back-to-back on *each* element in the stream. In a sense, the loops necessary to perform each intermediate operation on *all* elements successively are fused into a single loop (technically called *loop fusion*). Thus only a single pass is required over the elements of the stream.

Example 16.3 illustrates loop fusion resulting from lazy execution of a stream pipeline at (2). The intermediate operations now include print statements to announce their actions at (3) and (4). Note that we do not advocate this practice for production code. The output shows that the elements are processed *one at a time* through the pipeline when the terminal operation is executed. A CD is filtered first, and if it is a pop music CD, it is mapped to its title and the terminal operation includes this title in the result list. Otherwise, the CD is discarded. When there are no more CDs in the stream, the terminal operation completes, and the stream is consumed.

### *Short-circuit Evaluation*

The lazy nature of streams allows certain kinds of optimizations to be performed on stream operations. We have already seen an example of such an optimization that results in loop fusion of intermediate operations.

In some cases, it is not necessary to process all elements of the stream in order to produce a result (technically called *short-circuit execution*). For instance, the `limit()` intermediate operation creates a stream of a specified size, making it unnecessary to process the rest of the stream once this limit is reached. A typical example of its usage is to turn an infinite stream into a finite stream. Another example is the `takeWhile()` intermediate operation that short-circuits stream processing once its predicate becomes false.

Certain terminal operations (`anyMatch()`, `allMatch()`, `noneMatch()`, `findFirst()`, `findAny()`) are also short-circuit operations, since they do not need to process all elements of the stream in order to produce a result (p. 949).

### *Stateless and Stateful Operations*

An *stateless* operation is one that can be performed on a stream element without taking into consideration the outcome of any processing done on previous elements or on any elements yet to be processed. In other words, the operation does not retain any *state* from processing of previous elements in order to process a new element. Rather, the operation can be performed on an element independently of how the other elements are processed.

A *stateful* operation is one that needs to retain state from previously processed elements in order to process a new element.

The intermediate operations `distinct()`, `dropWhile()`, `limit()`, `skip()`, `sorted()`, and `takeWhile()` are *stateful* operations. All other intermediate operations are *stateless*. Examples of stateless intermediate operations include the `filter()` and `map()` operations.

### *Order of Intermediate Operations*

The order of intermediate operations in a stream pipeline can impact the performance of a stream pipeline. If intermediate operations that reduce the size of the stream can be performed earlier in the pipeline, fewer elements need to be processed by the subsequent operations.

Moving intermediate operations such as `filter()`, `distinct()`, `dropWhile()`, `limit()`, `skip()`, and `takeWhile()` earlier in the pipeline can be beneficial, as they all decrease the size of the input stream. Example 16.4 implements two stream pipelines at (1) and (2) to create a list of CD titles, but skipping the first three CDs. The `map()` operation transforms each CD to its title, resulting in an output stream with element type `String`. The example shows how the number of elements processed by the `map()` operation can be reduced if the `skip()` operation is performed before the `map()` operation (p. 921).

**Example 16.4** *Order of Intermediate Operations*

```

import java.util.List;

public final class OrderOfOperations {
    public static void main(String[] args) {

        List<CD> cdList = CD.cdList;

        // Map before skip.
        List<String> cdTitles1 = cdList
            .stream() // (1)
            .map(cd -> { // Map applied to all elements.
                System.out.println("Mapping: " + cd.title());
                return cd.title();
            })
            .skip(3) // Skip afterwards.
            .toList();
        System.out.println(cdTitles1);
        System.out.println();

        // Skip before map preferable.
        List<String> cdTitles2 = cdList
            .stream() // (2)
            .skip(3) // Skip first.
            .map(cd -> { // Map not applied to the first 3 elements.
                System.out.println("Mapping: " + cd.title());
                return cd.title();
            })
            .toList();
        System.out.println(cdTitles2);
    }
}

```

Output from the program:

```

Mapping: Java Jive
Mapping: Java Jam
Mapping: Lambda Dancing
Mapping: Keep on Erasing
Mapping: Hot Generics
[Keep on Erasing, Hot Generics]

Mapping: Keep on Erasing
Mapping: Hot Generics
[Keep on Erasing, Hot Generics]

```

*Non-interfering and Stateless Behavioral Parameters*

One of the main goals of the Stream API is that the code for a stream pipeline should execute and produce the same results whether the stream elements are processed sequentially or in parallel. In order to achieve this goal, certain constraints are placed on the *behavioral parameters*—that is, on the lambda expressions and

method references that are implementations of the functional interface parameters in stream operations. These behavioral parameters, as the name implies, allow the behavior of a stream operation to be customized. For example, the predicate supplied to the `filter()` operation defines the criteria for filtering the elements.

Most stream operations require that their behavioral parameters are *non-interfering* and *stateless*. A *non-interfering behavioral parameter* does not change the stream data source during the execution of the pipeline, as this might not produce deterministic results. The exception to this is when the data source is *concurrent*, which guarantees that the source is thread-safe. A *stateless behavioral parameter* does not access any state that can change during the execution of the pipeline, as this might not be thread-safe.

If the constraints are violated, all bets are off, resulting in incorrect results being computed, which causes the stream pipeline to fail. In addition to these constraints, care should be taken to introduce side effects via behavioral parameters, as these might introduce other concurrency-related problems during parallel execution of the pipeline.

The aspects of intermediate operations mentioned in this subsection will become clearer as we fill in the details in subsequent sections.

## Filtering

Filters are stream operations that select elements based on some criteria, usually specified as a *predicate*. This section discusses different ways of filtering elements, selecting unique elements, skipping elements at the head of a stream, and truncating a stream.

The following methods are defined in the `Stream<T>` interface, and analogous methods are also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

```
// Filtering using a predicate.
Stream<T> filter(Predicate<? super T> predicate)
```

Returns a stream consisting of the elements of this stream that match the given non-interfering, stateless predicate.

This is a stateless intermediate operation that changes the stream size, but not the stream element type or the encounter order of the stream.

```
// Taking and dropping elements using a predicate.
default Stream<T> takeWhile(Predicate<? super T> predicate)
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

The `takeWhile()` method puts an element from the input stream into its output stream, if it matches the predicate—that is, if the predicate returns the value `true` for this element. In this case, we say that the `takeWhile()` method *takes* the element.

The `dropWhile()` method discards an element from its input stream, if it matches the predicate—that is, if the predicate returns the value `true` for this element. In this case, we say that the `dropWhile()` method *drops* the element.

*For an ordered stream:*

The `takeWhile()` method takes elements from the input stream as long as an element matches the predicate, after which it short-circuits the stream processing.

The `dropWhile()` method drops elements from the input stream as long as an element matches the predicate, after which it passes through the remaining elements to the output stream.

In short, both methods find the *longest prefix of elements* to take or drop from the input stream, respectively.

*For an unordered stream, where the predicate matches some but not all elements in the input stream:*

The elements taken by the `takeWhile()` method or dropped by the `dropWhile()` method are *nondeterministic*; that is, any subset of matching elements can be taken or dropped, respectively, including the empty set.

*If the predicate matches all elements in the input stream, regardless of whether the stream is ordered or unordered:*

The `takeWhile()` method takes all elements; that is, the result is the same as the input stream.

The `dropWhile()` method drops all elements; that is, the result is the empty stream.

*If the predicate matches no elements in the input stream, regardless of whether the stream is ordered or unordered:*

The `takeWhile()` method takes no elements; that is, the result is the empty stream.

The `dropWhile()` method drops no elements; that is, the result is the same as the input stream.

Note that the `takeWhile()` method is a *short-circuiting* stateful intermediate operation, whereas the `dropWhile()` method is a stateful intermediate operation.

```
// Selecting distinct elements.  
Stream<T> distinct()
```

Returns a stream consisting of the distinct elements of this stream, where no two elements are equal according to the `Object.equals()` method; that is, the method assumes that the elements override the `Object.equals()` method. It also uses the `hashCode()` method to keep track of the elements, and this method should also be overridden from the `Object` class.

For ordered streams, the *first* occurrence of a duplicated element is selected in the encounter order—called the *stability guarantee*. This stateful operation is particularly expensive for a parallel ordered stream which entails buffering overhead to ensure the stability guarantee. There is no such guarantee for an unordered stream: Which occurrence of a duplicated element will be selected is not guaranteed.

This stateful intermediate operation changes the stream size, but not the stream element type.

```
// Skipping elements.
Stream<T> skip(long n)
```

Returns a stream consisting of the remaining elements of this stream after discarding the first *n* elements of the stream in encounter order. If this stream has fewer than *n* elements, an empty stream is returned.

This stateful operation is expensive for a parallel ordered stream which entails keeping track of skipping the *first* *n* elements.

This is a stateful intermediate operation that changes the stream size, but not the stream element type.

```
// Truncating a stream.
Stream<T> limit(long maxSize)
```

Returns a stream consisting of elements from this stream, truncating the length of the returned stream to be no longer than the value of the parameter *maxSize*.

This stateful operation is expensive for a parallel ordered stream which entails keeping track of passing the *first* *n* elements from the input stream to the output stream.

This is a short-circuiting, stateful intermediate operation.

### Filtering Using a Predicate

We have already seen many examples of filtering stream elements in this chapter. The first example of using the `Stream.filter()` method was presented in Figure 16.1, p. 885.

Filtering a collection using the `Iterator.remove()` method and the `Collection.removeIf()` method is discussed in §13.3, p. 691, and §15.2, p. 796, respectively.

The `filter()` method can be used on both object and numeric streams. The `Stream.filter()` method accepts a `Predicate<T>` as an argument. The predicate is typically implemented as a lambda expression or a method reference defining the selection criteria. It yields a stream consisting of elements from the input stream that satisfy the predicate. The elements that do not match the predicate are discarded.

In Figure 16.3, Query 1 selects those CDs from a list of CDs (`CD.cdList`) whose titles are in a set of popular CD titles (`popularTitles`). The `Collection.contains()` method is used in the predicate to determine if the title of a CD is in the set of popular CD titles. The execution of the stream pipeline shows there are only two such CDs (`cd0`, `cd1`). CDs that do not satisfy the predicate are discarded.

We can express the same query using the `Collection.removeIf()` method, as shown below. The code computes the same result as the stream pipeline in Figure 16.3. Note that the predicate in the `remove()` method call is a negation of the predicate in the `filter()` operation.

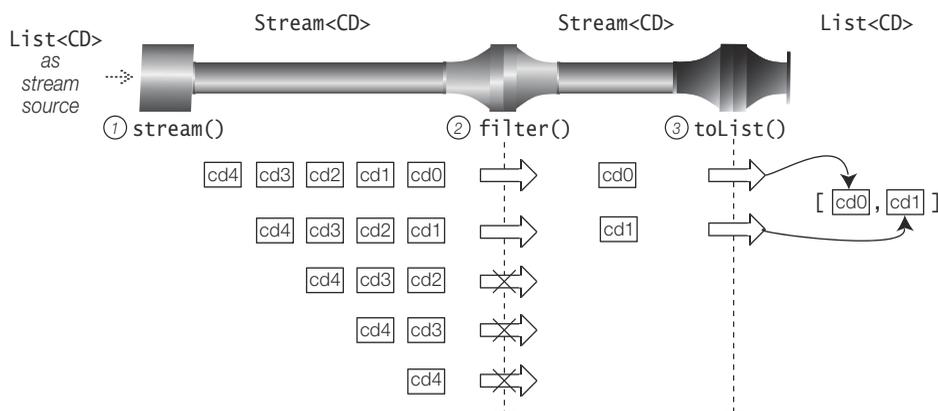
```
List<CD> popularCDs2 = new ArrayList<>(CD.cdList);
popularCDs2.removeIf(cd -> !popularTitles.contains(cd.title()));
System.out.println("Query 1b: " + popularCDs2);
//Query 1b: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ>]
```

In summary, the `filter()` method implements a stateless intermediate operation. It can change the size of the stream, since elements are discarded. However, the element type of the output stream returned by the `filter()` method is the same as that of its input stream. In Figure 16.3, the input and output stream type of the `filter()` method is `Stream<CD>`. Also, the encounter order of the stream remains unchanged. In Figure 16.3, the encounter order in the output stream returned by the `filter()` method is the same as the order of the elements in the input stream—that is, the insertion order in the list of CDs.

**Figure 16.3** *Filtering Stream Elements*

```
// Query 1: Find CDs whose titles are in the set of popular CD titles.
Set<String> popularTitles = Set.of("Java Jive", "Java Jazz", "Java Jam");
List<CD> popularCDs1 = CD.cdList
① .stream()
② .filter(cd -> popularTitles.contains(cd.title()))
③ .toList();
```

(a) Query to filter stream elements



(b) Execution of stream pipeline

### Taking and Dropping Elements Using Predicates

Both the `takeWhile()` and the `dropWhile()` methods find the longest prefix of elements to take or drop from the input stream, respectively.

The code below at (1) and (2) illustrates the case for ordered streams. The `takeWhile()` method takes odd numbers from the input stream until a number is not odd, and short-circuits the processing of the stream—that is, it truncates the rest of the stream based on the predicate. The `dropWhile()` method, on the other hand, drops odd numbers from the input stream until a number is not odd, and passes the remaining elements to its output stream; that is, it skips elements in the beginning of the stream based on the predicate.

```
// Ordered stream:
Stream.of(1, 3, 5, 7, 8, 9, 11) // (1)
```

```

        .takeWhile(n -> n % 2 != 0)           // Takes longest prefix: 1 3 5 7
        .forEach(n -> System.out.print(n + " ")); // 1 3 5 7

Stream.of(1, 3, 5, 7, 8, 9, 11)           // (2)
    .dropWhile(n -> n % 2 != 0)           // Drops longest prefix: 1 3 5 7
    .forEach(n -> System.out.print(n + " ")); // 8 9 11

```

Given an unordered stream, as shown below at (3), both methods return nondeterministic results: Any subset of matching elements can be taken or dropped, respectively.

```

// Unordered stream:
Set<Integer> iSeq = Set.of(1, 9, 4, 3, 7); // (3)
iSeq.stream()
    .takeWhile(n -> n % 2 != 0)           // Takes any subset of elements.
    .forEach(n -> System.out.print(n + " ")); // Nondeterministic: 1 9 7

iSeq.stream()
    .dropWhile(n -> n % 2 != 0)           // Drops any subset of elements.
    .forEach(n -> System.out.print(n + " ")); // Nondeterministic: 4 3

```

Regardless of whether the stream is ordered or unordered, if *all* elements match the predicate, the `takeWhile()` method takes all the elements and the `dropWhile()` method drops all the elements, as shown below at (4) and (5).

```

// All match in ordered stream:           (4)
Stream.of(1, 3, 5, 7, 9, 11)
    .takeWhile(n -> n % 2 != 0)           // Takes all elements.
    .forEach(n -> System.out.print(n + " ")); // Ordered: 1 3 5 7 9 11

Stream.of(1, 3, 5, 7, 9, 11)
    .dropWhile(n -> n % 2 != 0)           // Drops all elements.
    .forEach(n -> System.out.print(n + " ")); // Empty stream

// All match in unordered stream:         (5)
Set<Integer> iSeq2 = Set.of(1, 9, 3, 7, 11, 5);
iSeq2.stream()
    .takeWhile(n -> n % 2 != 0)           // Takes all elements.
    .forEach(n -> System.out.print(n + " ")); // Unordered: 9 11 1 3 5 7

iSeq2.stream()
    .dropWhile(n -> n % 2 != 0)           // Drops all elements.
    .forEach(n -> System.out.print(n + " ")); // Empty stream

```

Regardless of whether the stream is ordered or unordered, if *no* elements match the predicate, the `takeWhile()` method takes no elements and the `dropWhile()` method drops no elements, as shown below at (6) and (7).

```

// No match in ordered stream:           (6)
Stream.of(2, 4, 6, 8, 10, 12)
    .takeWhile(n -> n % 2 != 0)           // Takes no elements.
    .forEach(n -> System.out.print(n + " ")); // Empty stream

Stream.of(2, 4, 6, 8, 10, 12)
    .dropWhile(n -> n % 2 != 0)           // Drops no elements.
    .forEach(n -> System.out.print(n + " ")); // Ordered: 2 4 6 8 10 12

```

```
// No match in unordered stream: (7)
Set<Integer> iSeq3 = Set.of(2, 10, 8, 12, 4, 6);
iSeq3.stream()
    .takeWhile(n -> n % 2 != 0) // Takes no elements.
    .forEach(n -> System.out.print(n + " ")); // Empty stream

iSeq3.stream()
    .dropWhile(n -> n % 2 != 0) // Drops no elements.
    .forEach(n -> System.out.print(n + " ")); // Unordered: 8 10 12 2 4 6
```

### *Selecting Distinct Elements*

The `distinct()` method removes all duplicates of an element from the input stream, resulting in an output stream with only unique elements. Since the `distinct()` method must be able to distinguish the elements from one another and keep track of them, the stream elements must override the `equals()` and the `hashCode()` methods of the `Object` class. The CD objects comply with this requirement (Example 16.1, p. 883).

In Figure 16.4, Query 2 creates a list of unique CDs with pop music. The `filter()` operation and the `distinct()` operation in the stream pipeline select the CDs with pop music and those that are unique, respectively. The execution of the stream pipeline shows that the resulting list of unique CDs with pop music has only one CD (cd0).

In Figure 16.4, interchanging the stateless `filter()` operation and the stateful `distinct()` operation in the stream pipeline gives the same results, but then the more expensive `distinct()` operation is performed on *all* elements of the stream, rather than on a shorter stream which is returned by the `filter()` operation.

### *Skipping Elements in a Stream*

The `skip()` operation slices off or discards a specified number of elements from the head of a stream before the remaining elements are made available to the next operation. It preserves the encounter order if the input stream has one. Not surprisingly, skipping more elements than are in the input stream returns the empty stream.

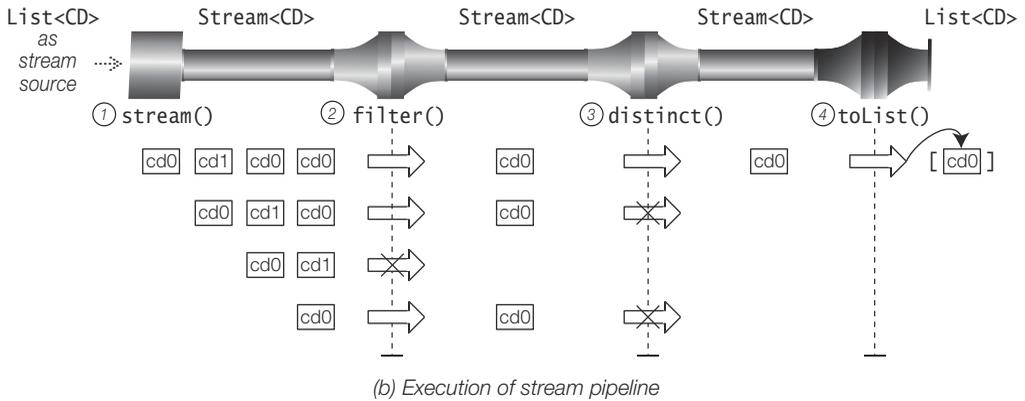
In Figure 16.5, Query 3a creates a list of jazz music CDs after skipping the first two CDs in the stream. The stream pipeline uses a `skip()` operation first to discard two CDs (one of them being a jazz music CD) and a `filter()` operation afterward to select any CDs with jazz music. The execution of this stream pipeline shows that the resulting list contains two CDs (cd3, cd4).

In the stream pipeline in Figure 16.5, the `skip()` operation is before the `filter()` operation. Switching the order of the `skip()` and `filter()` operations as in Query 3b in Example 16.5 does not solve the same query. It will skip the first two jazz music CDs selected by the `filter()` operation.

**Figure 16.4** *Selecting Distinct Elements*

```
// Query 2: Create a list of unique CDs with pop music.
List<CD> miscCDList = List.of(CD.cd0, CD.cd0, CD.cd1, CD.cd0);
List<CD> uniquePopCDs1 = miscCDList
    ① .stream()
    ② .filter(CD::isPop)
    ③ .distinct()
    ④ .toList();
```

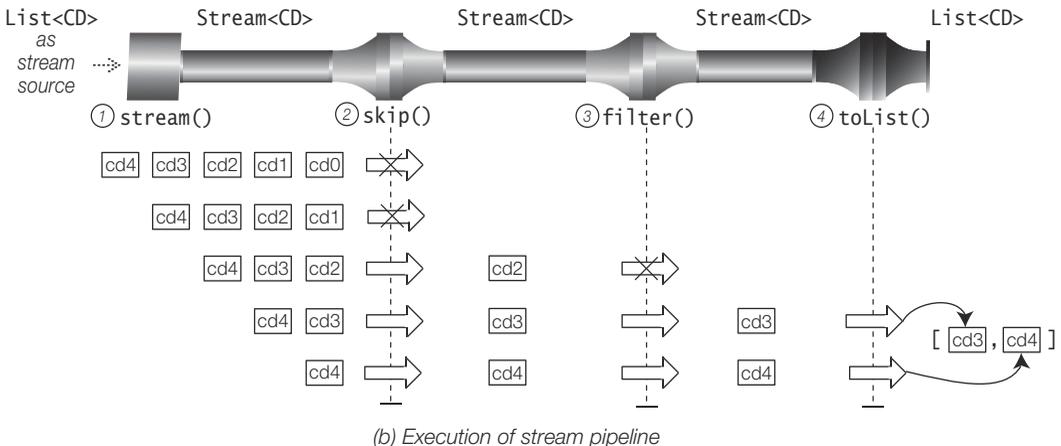
(a) *Selecting distinct elements in a stream*



**Figure 16.5** *Skipping Elements at the Head of a Stream*

```
// Query 3a: Create a list of jazz CDs, after skipping the first two CDs.
List<CD> jazzCDs1 = CD.cdList
    ① .stream()
    ② .skip(2)
    ③ .filter(CD::isJazz)
    ④ .toList();
```

(a) *Skipping elements in a stream*



## Truncating a Stream

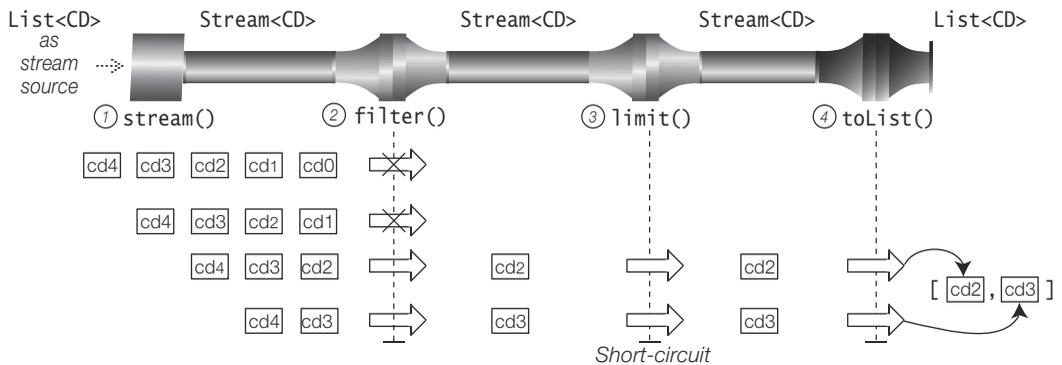
The `limit()` operation returns an output stream whose maximum size is equal to the max size specified as an argument to the method. The input stream is only truncated if its size is greater than the specified max size.

In Figure 16.6, Query 4 creates a list with the first two CDs that were released in 2018. The stream pipeline uses a `filter()` operation first to select CDs released in 2018, and the `limit()` operation truncates the stream, if necessary, so that, at most, only two CDs are passed to its output stream. The short-circuit execution of this stream pipeline is illustrated in Figure 16.6, showing the resulting list containing two CDs (`cd2`, `cd3`). The execution of the stream pipeline terminates after the `limit()` operation has reached its limit if there are no more elements left to process. In Figure 16.6, we can see that the limit was reached and execution was terminated. Regardless of the fact that the last element in the initial stream was not processed, the stream cannot be reused once the execution of the pipeline terminates due to a short-circuiting operation.

**Figure 16.6** *Truncating a Stream*

```
// Query 4: Create a list with the first 2 CDs that were released in 2018.
List<CD> twoFirstCDs2018 = CD.cdList
① .stream()
② .filter(cd -> cd.year().equals(Year.of(2018)))
③ .limit(2)
④ .toList();
```

(a) *Truncating a stream*



(b) *Execution of stream pipeline*

The `limit()` operation is ideal for turning an infinite stream into a finite stream. Numerous examples of using the `limit()` operation with the `iterate()` and `generate()` methods can be found in §16.4, p. 894, and with the `Random.ints()` method in §16.4, p. 900.

For a given value `n`, `limit(n)` and `skip(n)` are complementary operations on a stream, as `limit(n)` comprises the first `n` elements of the stream and `skip(n)` comprises the

remaining elements in the stream. In the code below, the `resultList` from processing the resulting stream from concatenating the two substreams is equal to the stream source `CD.cdList`.

```
List<CD> resultList = Stream
    .concat(CD.cdList.stream().limit(2), CD.cdList.stream().skip(2))
    .toList();
System.out.println(CD.cdList.equals(resultList));           // true
```

The `skip()` operation can be used in conjunction with the `limit()` operation to process a substream of a stream, where the `skip()` operation can be used to skip to the start of the substream and the `limit()` operation to limit the size of the substream. The substream in the code below starts at the second element and comprises the next three elements in the stream.

```
List<CD> substream = CD.cdList
    .stream()
    .skip(1)
    .limit(3)
    .toList();
System.out.println("Query 5: " + substream);
// Query 5: [<Jaav, "Java Jam", 6, 2017, JAZZ>,
//           <Funkies, "Lambda Dancing", 10, 2018, POP>,
//           <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]
```

The `limit()` operation is a short-circuiting stateful intermediate operation, as it needs to keep state for tracking the number of elements in the output stream. It changes the stream size, but not the stream element type or the encounter order. For an ordered stream, we can expect the elements in the resulting stream to have the same order, but we cannot assume any order if the input stream is unordered.

Example 16.5 contains the code snippets presented in this subsection.

---

### Example 16.5 *Filtering*

```
import java.time.Year;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Stream;

public final class Filtering {
    public static void main(String[] args) {

        // Query 1: Find CDs whose titles are in the set of popular CD titles.
        Set<String> popularTitles = Set.of("Java Jive", "Java Jazz", "Java Jam");

        // Using Stream.filter().
        List<CD> popularCDs1 = CD.cdList
            .stream()
            .filter(cd -> popularTitles.contains(cd.title()))
            .toList();
        System.out.println("Query 1a: " + popularCDs1);
```

```
// Using Collection.removeIf().
List<CD> popularCDs2 = new ArrayList<>(CD.cdList);
popularCDs2.removeIf(cd -> !(popularTitles.contains(cd.title())));
System.out.println("Query 1b: " + popularCDs2);

// Query 2: Create a list of unique CDs with pop music.
List<CD> miscCDList = List.of(CD.cd0, CD.cd0, CD.cd1, CD.cd0);
List<CD> uniquePopCDs1 = miscCDList
    .stream()
    .filter(CD::isPop)
    .distinct() // distinct() after filter()
    .toList();
System.out.println("Query 2: " + uniquePopCDs1);

// Query 3a: Create a list of jazz CDs, after skipping the first two CDs.
List<CD> jazzCDs1 = CD.cdList
    .stream()
    .skip(2) // skip() before filter().
    .filter(CD::isJazz)
    .toList();
System.out.println("Query 3a: " + jazzCDs1);

// Query 3b: Create a list of jazz CDs, but skip the first two jazz CDs.
List<CD> jazzCDs2 = CD.cdList // Not equivalent to Query 3
    .stream()
    .filter(CD::isJazz)
    .skip(2) // skip() after filter().
    .toList();
System.out.println("Query 3b: " + jazzCDs2);

// Query 4: Create a list with the first 2 CDs that were released in 2018.
List<CD> twoFirstCDs2018 = CD.cdList
    .stream()
    .filter(cd -> cd.year().equals(Year.of(2018)))
    .limit(2)
    .toList();
System.out.println("Query 4: " + twoFirstCDs2018);

// limit(n) and skip(n) are complementary.
List<CD> resultList = Stream
    .concat(CD.cdList.stream().limit(2), CD.cdList.stream().skip(2))
    .toList();
System.out.println(CD.cdList.equals(resultList));

// Query 5: Process a substream by skipping 1 and limiting the size to 3.
List<CD> substream = CD.cdList
    .stream()
    .skip(1)
    .limit(3)
    .toList();
System.out.println("Query 5: " + substream);
}
}
```

Output from the program (*formatted to fit on the page*):

```

Query 1a: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ>]
Query 1b: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ>]
Query 2: [<Jaav, "Java Jive", 8, 2017, POP>]
Query 3a: [<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
          <Genericos, "Hot Genericos", 10, 2018, JAZZ>]
Query 3b: [<Genericos, "Hot Genericos", 10, 2018, JAZZ>]
Query 4: [<Funkies, "Lambda Dancing", 10, 2018, POP>,
          <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]

true
Query 5: [<Jaav, "Java Jam", 6, 2017, JAZZ>,
          <Funkies, "Lambda Dancing", 10, 2018, POP>,
          <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]

```

## Examining Elements in a Stream

The `peek()` operation allows stream elements to be examined at the point where the operation is used in the stream pipeline. It does not affect the stream in any way, as it only facilitates a side effect via a non-interfering *consumer* specified as an argument to the operation. It is primarily used for debugging the pipeline by examining the elements at various points in the pipeline.

The following method is defined in the `Stream<T>` interface, and an analogous method is also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

```
Stream<T> peek(Consumer<? super T> action)
```

Returns a stream consisting of the same elements as those in this stream, but additionally performs the provided non-interfering action on each element as elements are processed from this stream.

This is a stateless intermediate operation that does not change the stream size, the stream element type, or the encounter order.

By using the `peek()` method, we can dispense with explicit print statements that were inserted in the implementation of the behavioral parameter of the `map()` operation in Example 16.4, p. 909. Example 16.6 shows how the `peek()` operation can be used to trace the processing of elements in the pipeline. A `peek()` operation after each intermediate operation prints pertinent information which can be used to verify the workings of the pipeline. In Example 16.6, the output shows that the `skip()` operation before the `map()` operation can improve performance, as the `skip()` operation shortens the stream on which the `map()` operation should be performed.

### Example 16.6 Examining Stream Elements

```

import java.util.List;

public final class OrderOfOperationsWithPeek {
    public static void main(String[] args) {

```

```

System.out.println("map() before skip()");
List<String> cdTitles1 = CD.cdList
    .stream()
    .map(CD::title)
    .peek(t -> System.out.println("After map: " + t))
    .skip(3)
    .peek(t -> System.out.println("After skip: " + t))
    .toList();
System.out.println(cdTitles1);
System.out.println();

System.out.println("skip() before map()");           // Preferable.
List<String> cdTitles2 = CD.cdList
    .stream()
    .skip(3)
    .peek(cd -> System.out.println("After skip: " + cd))
    .map(CD::title)
    .peek(t -> System.out.println("After map: " + t))
    .toList();
System.out.println(cdTitles2);
}
}

```

Output from the program:

```

map() before skip():
After map: Java Jive
After map: Java Jam
After map: Lambda Dancing
After map: Keep on Erasing
After skip: Keep on Erasing
After map: Hot Generics
After skip: Hot Generics
[Keep on Erasing, Hot Generics]

skip() before map():
After skip: <Genericos, "Keep on Erasing", 8, 2018, JAZZ>
After map: Keep on Erasing
After skip: <Genericos, "Hot Generics", 10, 2018, JAZZ>
After map: Hot Generics
[Keep on Erasing, Hot Generics]

```

## Mapping: Transforming Streams

The `map()` operation has already been used in several examples (Example 16.3, p. 906, Example 16.4, p. 909, and Example 16.6, p. 920). Here we take a closer look at this essential intermediate operation for data processing using a stream. It maps one type of stream (`Stream<T>`) into another type of stream (`Stream<R>`); that is, *each* element of type `T` in the input stream is mapped to an element of type `R` in the output stream by the function (`Function<T, R>`) supplied to the `map()` method. It defines a *one-to-one mapping*. For example, if we are interested in the *titles* of CDs in the CD

stream, we can use the `map()` operation to transform each CD in the stream to a `String` that represents the title of the CD by applying an appropriate function:

```
Stream<String> titles = CD.cdList
    .stream()                // Input stream: Stream<CD>.
    .map(CD::title);        // Lambda expression: cd -> cd.title()
```

The following methods are defined in the `Stream<T>` interface, and analogous methods are also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Returns a stream consisting of the result of applying the given non-interfering, stateless function to the elements of this stream—that is, it creates a new stream (`Stream<R>`) from the results of applying the mapper function to the elements of this stream (`Stream<T>`).

This is an intermediate operation that does not change the stream size, but it can change the stream element type and does not guarantee to preserve the encounter order of the input stream.

```
// Converting Stream<T> to a Numeric Stream
IntStream  mapToInt(ToIntFunction<? super T> mapper)
LongStream mapToLong(ToLongFunction<? super T> mapper)
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

Return the numeric stream consisting of the results of applying the given non-interfering, stateless function to the elements of this stream—that is, they create a stream of numeric values that are the results of applying the mapper function to the elements of this stream (`Stream<T>`).

These operations are all intermediate operations that transform an object stream to a numeric stream. The stream size is not affected, but there is no guarantee that the encounter order of the input stream is preserved.

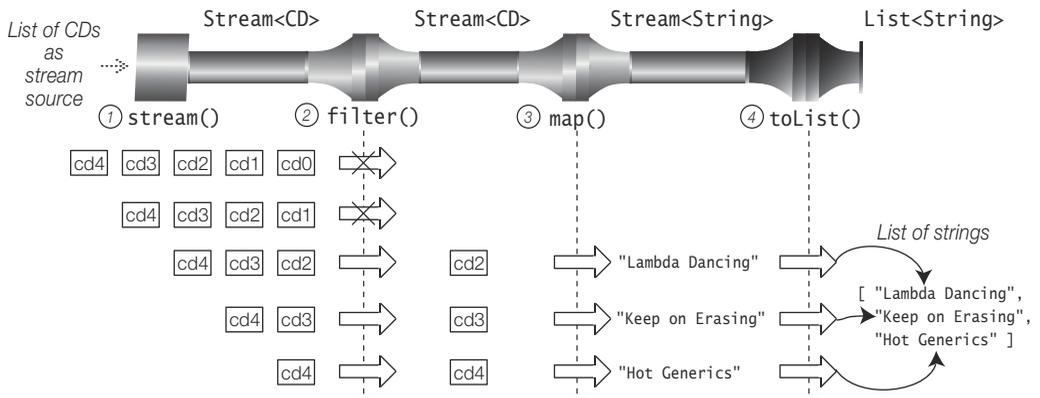
In Figure 16.7, the query creates a list with CD titles released in 2018. The stream pipeline uses a `filter()` operation first to select CDs released in 2018, and the `map()` operation maps a CD to its title (`String`). The input stream is transformed by the `map()` operation from `Stream<CD>` to `Stream<String>`. The execution of this stream pipeline shows the resulting list (`List<String>`) containing three CD titles.

The query below illustrates transforming an object stream to a numeric stream. When executed, the stream pipeline prints the years in which the CDs were released. Note the transformation of the initial stream, `Stream<CD>`. The `map()` operation first transforms it to a `Stream<Year>` and the `distinct()` operation selects the unique years. The `mapToInt()` operation transforms the stream from `Stream<Year>` to `IntStream`—that is, a stream of ints whose values are then printed.

```
CD.cdList.stream()                // Stream<CD>
    .map(CD::year)                 // Stream<Year>
    .distinct()                   // Stream<Year>
    .mapToInt(Year::getValue)     // IntStream
    .forEach(System.out::println); // 2017
                                    // 2018
```

**Figure 16.7** Mapping

```
// Query: Create a list of CD titles released in 2018.
List<String> listOfCDNames = CD.cdList
① .stream() // Stream<CD>
② .filter(cd -> cd.year().equals(Year.of(2018))) // Stream<CD>
③ .map(CD::title) // Stream<String>
④ .toList(); // List<String>
```

(a) Query using the `Stream.map()` method

(b) Execution of stream pipeline

In the example below, the `range()` method generates an `int` stream for values in the half-open interval specified by its arguments. The values are generated in *increasing* order, starting with the lower bound of the interval. In order to generate them in *decreasing* order, the `map()` operation can be used to reverse the values. In this case, the input stream and output stream of the `map()` operation are both `IntStreams`.

```
int from = 0, to = 5;
IntStream.range(from, to) // [0, 5)
    .map(i -> to + from - 1 - i) // Reverse the stream values
    .forEach(System.out::print); // 43210
```

The stream pipeline below determines the number of times the dice value is 6. The `generate()` method generates a value between 1 and 6, and the `limit()` operation limits the max size of the stream. The `map()` operation returns the value 1 if the dice value is 6; otherwise, it returns 0. In other words, the value of the dice throw is mapped either to 1 or 0, depending on the dice value. The terminal operation `sum()` sums the values in the streams, which in this case are either 1 or 0, thus returning the correct number of times the dice value was 6.

```
long sixes = IntStream
    .generate(() -> (int) (6.0 * Math.random()) + 1) // [1, 6)
    .limit(2000) // Number of throws.
    .map(i -> i == 6 ? 1 : 0) // Dice value mapped to 1 or 0.
    .sum();
```

## Flattening Streams

The `flatMap()` operation first maps each element in the input stream to a *mapped stream*, and then *flattens* the mapped streams to a *single stream*—that is, the elements of each mapped stream are incorporated into a single stream when the pipeline is executed. In other words, each element in the input stream may be mapped to many elements in the output stream. The `flatMap()` operation thus defines a *one-to-many mapping* that flattens a multilevel stream by one level.

The following method is defined in the `Stream<T>` interface, and an analogous method is also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

```
<R> Stream<R> flatMap(
    Function<? super T,? extends Stream<? extends R>> mapper)
```

The mapper function maps each element of type `T` in this stream to a *mapped stream* (`Stream<R>`). The method returns an output stream (`Stream<R>`) which is the result of replacing each element of type `T` in this stream with the *elements* of type `R` from its mapped stream.

If the result of the mapper function is `null`, the empty stream is used as the mapped stream.

This is an intermediate operation that changes the stream size and the element type of the stream, and does not guarantee to preserve the encounter order of the input stream.

The methods below are defined only in the `Stream<T>` interface. No counterparts exist in the `IntStream`, `LongStream`, or `DoubleStream` interfaces:

```
IntStream    flatMapToInt(Function<? super T,? extends IntStream> mapper)
LongStream   flatMapToLong(Function<? super T,? extends LongStream> mapper)
DoubleStream flatMapToDouble(
    Function<? super T,? extends DoubleStream> mapper)
```

The mapper function maps each element of type `T` in this stream to a *mapped numeric stream* (`NumTypeStream`). The method returns an output stream (`NumTypeStream`), which is the result of replacing each element of type `T` in this stream with the *values* from its mapped numeric stream. The designation `NumType` stands for `Int`, `Long`, or `Double`.

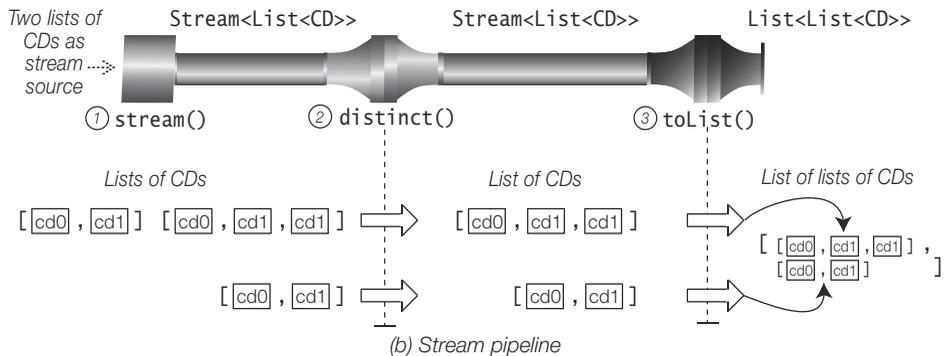
To motivate using the `flatMap()` operation, we look at how to express the query for creating a list of unique CDs from two given lists of CDs. Figure 16.8 shows an attempt to express this query by creating a stream of lists of CDs, `Stream<List<CD>>`, and selecting the unique CDs using the `distinct()` method. This attempt fails miserably as the `distinct()` method distinguishes between elements that are lists of CDs, and not individual CDs. Figure 16.8 shows the execution of the stream pipeline resulting in a list of lists of CDs, `List<List<CD>>`.

The next attempt to express the query uses the `map()` operation as shown in Figure 16.9. The idea is to map each list of CDs (`List<CD>`) to a stream of CDs (`Stream<CD>`), and select the unique CDs with the `distinct()` operation. The mapper

**Figure 16.8** *Incorrect Solution to the Query*

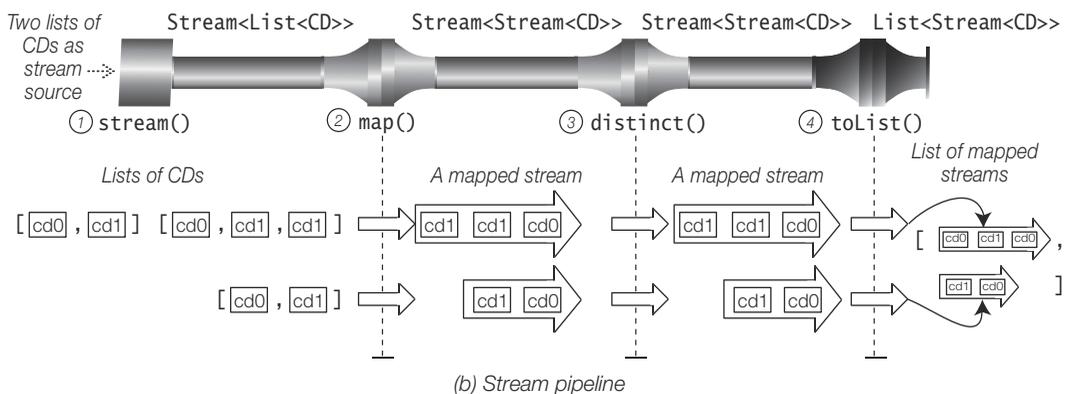
```
// Query: Create a list of unique CDs from two given lists of CDs.
List<List<CD>> listOfListOfCDs =
  ① Stream.of(cdList1, cdList2)           // Stream<List<CD>>
  ② .distinct()                           // Stream<List<CD>>
  ③ .toList();                             // List<List<CD>>
```

(a) *Incorrect solution using the Stream.distinct() method*

**Figure 16.9** *Mapping a Stream of Streams*

```
// Query: Create a list of unique CDs from two given lists of CDs.
List<Stream<CD>> listOfStreamOfCD =
  ① Stream.of(cdList1, cdList2)           // Stream<List<CD>>
  ② .map(List::stream)                    // Stream<Stream<CD>>
  ③ .distinct()                           // Stream<Stream<CD>>
  ④ .toList();                             // List<Stream<CD>>
```

(a) *Incorrect solution using the Stream.map() method*



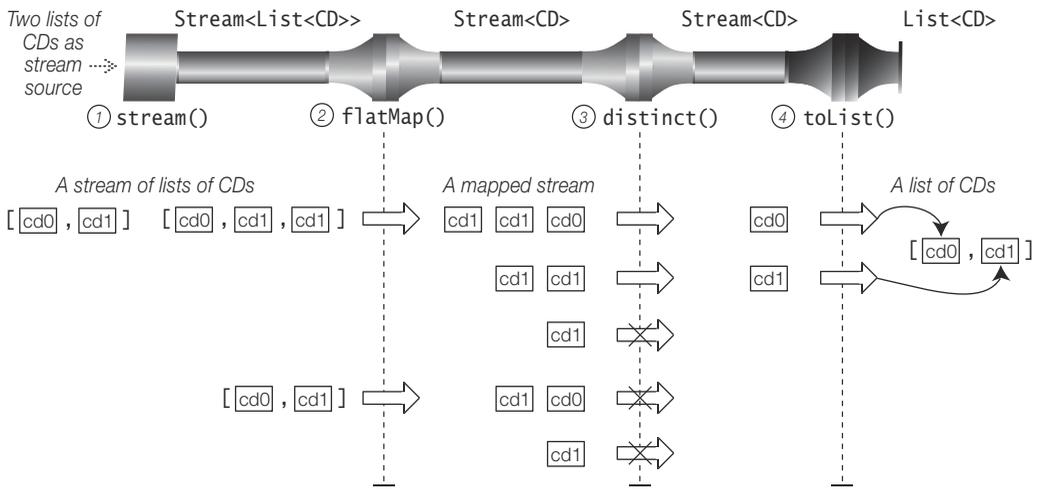
function of the `map()` operation maps each list of CDs to a *mapped stream* that is a stream of CDs, `Stream<CD>`. The resulting stream from the `map()` operation is a stream of streams of CDs, `Stream<Stream<CD>>`. The `distinct()` method distinguishes between elements that are mapped streams of CDs. Figure 16.9 shows the execution of the stream pipeline resulting in a list of mapped streams of CDs, `List<Stream<CD>>`.

The flatMap() operation provides the solution, as it flattens the contents of the mapped streams into a single stream so that the distinct() operation can select the unique CDs individually. The stream pipeline using the flatMap() operation and its execution are shown in Figure 16.10. The mapper function of the flatMap() operation maps each list of CDs to a mapped stream that is a stream of CDs, Stream<CD>. The contents of the mapped stream are flattened into the output stream. The resulting stream from the flatMap() operation is a stream of CDs, Stream<CD>. Note how each list in the initial stream results in a flattened stream whose elements are processed by the pipeline. The result list of CDs contains the unique CDs from the two lists.

Figure 16.10 Flattening Streams

```
//Query: Create a list of unique CDs from two given lists of CDs.
List<CD> listOfCD =
  ① Stream.of(cdList1, cdList2)           // Stream<List<CD>>
  ②   .flatMap(List::stream)             // Stream<CD>
  ③   .distinct()                       // Stream<CD>
  ④   .toList();                       // List<CD>
```

(a) Solution using the Stream.flatMap() method



(b) Stream pipeline

The code below flattens a two-dimensional array to a one-dimensional array. The Arrays.stream() method call at (1) creates an object stream, Stream<int[]>, whose elements are arrays that are rows in the two-dimensional array. The mapper of the flatMapToInt() operation maps each row in the Stream<int[]> to a stream of ints (IntStream) by applying the Array.stream() method at (2) to each row. This would result in a stream of mapped streams of ints (Stream<IntStream>), but it is flattened by the flatMapToInt() operation to a final stream of ints (IntStream). The terminal operation toArray() creates an appropriate array in which the int values of the final stream are stored (p. 971).

```

int[][] twoDimArray = { {2017, 2018}, {1948, 1949} };
int[] intArray = Arrays
    .stream(twoDimArray) // (1) Stream<int[]>
    .flatMapToInt(row -> Arrays.stream(row)) // (2) mapper: int[] -> IntStream,
    // flattens Stream<IntStream> to IntStream.
    .toArray(); // [2017, 2018, 1948, 1949]

```

## Replacing Each Element of a Stream with Multiple Elements

The `mapMulti()` intermediate operation applies a *one-to-many transformation* to the elements of the stream and flattens the result elements into a new stream. The functionality of the `mapMulti()` method is very similar to that of the `flatMap()` method. Whereas the latter uses a `Function<T, Stream<R>>` mapper to create a *mapping stream* for each element and then flattens the stream, the former applies a `BiConsumer<T, Consumer<R>>` mapper to each element. The mapper calls the `Consumer` to accept the replacement elements that are incorporated into a single stream when the pipeline is executed.

The `mapMulti()` method can be used to perform filtering, mapping, and flat mapping of stream elements, all depending on the implementation of the `BiConsumer` mapper passed to the method.

The code below shows a *one-to-one transformation* of the stream elements. A `BiConsumer` is defined at (1) that first filters the stream for pop music CDs at (2), and maps each CD to a string that contains its title and its number of tracks represented by an equivalent number of "\*" characters. The resulting string is submitted at (3) to the consumer (supplied by the `mapMulti()` method). Each value passed to the `accept()` method of the consumer replaces the current element in the stream. Note that the body of the `BiConsumer` is implemented in an *imperative* manner using an `if` statement. The `BiConsumer` created at (1) is passed to the `mapMulti()` method at (5) to process the CDs of the stream created at (4). The `mapMulti()` method passes an appropriate `Consumer` to the `BiConsumer` that accepts the replacement elements.

```

// One-to-one
BiConsumer<CD, Consumer<String>> bcA = (cd, consumer) -> { // (1)
    if (cd.genre() == Genre.POP) { // (2)
        consumer.accept(String.format("%-15s: %s", cd.title(), // (3)
            "*".repeat(cd.noOfTracks())));
    }
};

CD.cdList.stream() // (4)
    .mapMulti(bcA) // (5)
    .forEach(System.out::println);

```

Output from the code:

```

Java Jive      : *****
Lambda Dancing : *****

```

The code below shows a *one-to-many* transformation of the stream elements. The `BiConsumer` at (1) iterates through a list of CDs and maps each CD in the list to its title. Each list of CDs in the stream will thus be replaced with the titles of the CDs in the list. The `mapMulti()` operation with the `BiConsumer` at (1) is applied at (3) to a stream of list of CDs (`Stream<List<CD>>`) created at (2). The `mapMulti()` operation in this case is analogous to the `flatMap()` operation to achieve the same result.

```
// One-to-many
List<CD> cdList1 = List.of(CD.cd0, CD.cd1, CD.cd1);
List<CD> cdList2 = List.of(CD.cd0, CD.cd1);
BiConsumer<List<CD>, Consumer<String>> bcB = (lst, consumer) -> {      // (1)
    for (CD cd : lst) {
        consumer.accept(cd.title());
    }
};
List<String> listOfCDTitles = Stream.of(cdList1, cdList2) // (2) Stream<List<CD>>
    .mapMulti(bcB) // (3)
    .distinct()
    .toList();
System.out.println(listOfCDTitles); // [Java Jive, Java Jam]
```

The previous two code snippets first defined the `BiConsumer` with all relevant types specified explicitly, and then passed it to the `mapMulti()` method. The code below defines the implementation of the `BiConsumer` in the call to the `mapMulti()` method. We consider three alternative implementations as exemplified by (2a), (2b), and (2c).

Alternative (2a) results in a compile-time error. The reason is that the compiler cannot unequivocally infer the actual type parameter `R` of the consumer parameter of the lambda expression. It can only infer that the type of the `lst` parameter is `List<CD>` as it denotes an element of stream whose type is `Stream<List<CD>>`. The compiler makes the safest assumption that the type parameter `R` is `Object`. With this assumption, the resulting list is of type `List<Object>`, but this cannot be assigned to a reference of type `List<String>`, as declared in the assignment statement. To avoid the compile-time error in this case, we can change the type of the reference to `Object` or to the wildcard `?`.

Alternative (2b) uses the type witness `<String>` in the call to the `mapMulti()` method to explicitly corroborate the actual type parameter of the consumer.

Alternative (2c) explicitly specifies the types for the parameters of the lambda expression.

```
List<String> listOfCDTitles2 = Stream.of(cdList1, cdList2) // (1) Stream<List<CD>>
// .mapMulti((lst, consumer) -> { // (2a) Compile-time error!
// .<String>mapMulti((lst, consumer) -> { // (2b) OK.
    .mapMulti((List<CD> lst, Consumer<String> consumer) -> { // (2c) OK.
        for (CD cd : lst) {
            consumer.accept(cd.title());
        }
    })
    .distinct()
    .toList();
System.out.println(listOfCDTitles2); // [Java Jive, Java Jam]
```

The `mapMulti()` method is preferable to the `flatMap()` method under the following circumstances:

- When an element is to be replaced with a small number of elements, or none at all. The `mapMulti()` method avoids the overhead of creating a mapped stream for each element, as done by the `flatMap()` method.
- When an *imperative approach* for creating replacement elements is easier than using a stream.

The following default method is defined in the `Stream<T>` interface, and an analogous method is also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

```
default <R> Stream<R> mapMulti(
    BiConsumer<? super T,? super Consumer<R>> mapper)
```

Returns a stream that is a result of replacing each element of this stream with multiple elements, specifically zero or more elements.

The specified `mapper` is applied to each element in conjunction with a *consumer* that *accepts* replacement elements. The `mapper` calls the consumer zero or more times to accept the replacement elements.

Note that the consumer is supplied by the `mapMulti()` method, and called by the `mapper` to accept replacement elements. An element of type `T` is replaced with zero or more elements of type `R`.

This is an intermediate operation that changes the stream size and the element type of the stream, and does not guarantee to preserve the encounter order of the input stream.

The following default methods are defined only in the `Stream<T>` interface. No counterparts exist in the `IntStream`, `LongStream`, or `DoubleStream` interfaces:

```
default IntStream
    mapMultiToInt(BiConsumer<? super T,? super IntConsumer> mapper)
default LongStream
    mapMultiToLong(BiConsumer<? super T,? super LongConsumer> mapper)
default DoubleStream
    mapMultiToDouble(BiConsumer<? super T,? super DoubleConsumer> mapper)
```

Return an `IntStream`, `LongStream`, and `DoubleStream`, respectively, consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements.

## Sorted Streams

The `sorted()` intermediate operation can be used to enforce a specific encounter order on the elements of the stream. It is important to note that the data source is *not* sorted; only the order of the elements in the stream is affected when a stream is sorted. It is an expensive stateful operation, as state must be kept for all elements in the stream before making them available in the resulting stream.

The following methods are defined in the `Stream<T>` interface, but only the first method is defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> cmp)
```

Return a stream containing the elements of this stream, sorted according to natural order or according to total order defined by the specified comparator, respectively.

The first method requires that the elements implement the `Comparable<T>` interface.

The sorting operation provides a *stability guarantee* for ordered streams only—that is, duplicates of an element will be in their encounter order in the resulting stream.

This is a stateful intermediate operation that does not change the size of the stream or the stream element type, and enforces the sort order to be the encounter order of the resulting stream.

The `Comparable<E>` and `Comparator<E>` interfaces are covered in §14.4, p. 761, and §14.5, p. 769, respectively.

Example 16.7 illustrates the `sorted()` operation on streams. Printing the array at (1) and executing the stream pipeline at (2) shows that the order of the elements in the array and in the stream is *positional order*, as one would expect. The zero-argument `sorted()` method sorts in *natural order*, as in the pipeline at (3). It expects the stream elements to implement the `Comparable<CD>` interface. The `sorted()` method in the pipeline at (4) uses the *reverse natural order* to sort the elements.

The pipeline at (5) represents the query to find all jazz music CDs and sort them by their title. A comparator to compare by title is passed to the `sorted()` method. Finally, the pipeline at (6) finds CDs with eight or more tracks, and sorts them according to the number of tracks. An appropriate comparator that compares by the number of tracks is passed to the `sorted()` method.

It is instructive to compare the output showing the results from each pipeline in Example 16.7. The comparators in Example 16.7 are also implemented as lambda expressions, in addition to their implementation by the methods in the `Comparator<E>` interface.

---

#### Example 16.7 *Sorting Streams*

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Sorting {
    public static void main(String[] args) {

        System.out.println("(1) Positional order in the array:");
```

```

    CD[] cdArray = CD.cdArray;
    System.out.println(Arrays.toString(cdArray));           // (1)

    System.out.println("(2) Positional order in the stream:");
    List<CD> cdsByPositionalOrder =                       // (2)
        Arrays.stream(cdArray)
            .toList();
    System.out.println(cdsByPositionalOrder);

    System.out.println("(3) Natural order:");
    List<CD> cdsByNaturalOrder =                         // (3)
        Arrays.stream(cdArray)
            .sorted()
            .toList();
    System.out.println(cdsByNaturalOrder);

    System.out.println("(4) Reversed natural order:");
    List<CD> cdsByRNO =                                  // (4)
        Arrays.stream(cdArray)
//        .sorted((c1, c2) -> -c1.compareTo(c2))
//        .sorted(Comparator.reverseOrder())
        .toList();
    System.out.println(cdsByRNO);

    System.out.println("(5) Only Jazz CDs, ordered by title:");
    List<String> jazzCDsByTitle =                       // (5)
        Arrays.stream(cdArray)
            .filter(CD::isJazz)
//        .sorted((c1, c2) -> c1.title().compareTo(c2.title()))
//        .sorted(Comparator.comparing(CD::title))
//        .map(CD::title)
        .toList();
    System.out.println(jazzCDsByTitle);

    System.out.println("(6) No. of tracks >= 8, ordered by number of tracks:");
    List<CD> cds =                                       // (6)
        Arrays.stream(cdArray)
            .filter(d -> d.noOfTracks() >= 8)
//        .sorted((c1, c2) -> c1.noOfTracks() - c2.noOfTracks())
//        .sorted(Comparator.comparing(CD::noOfTracks))
        .toList();
    System.out.println(cds);
}
}

```

Output from the program (*formatted to fit on the page*):

```

(1) Positional order in the array:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Genericos, "Hot Genericos", 10, 2018, JAZZ>]
(2) Positional order in the stream:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,

```

```

<Funkies, "Lambda Dancing", 10, 2018, POP>,
<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
<Genericos, "Hot Generics", 10, 2018, JAZZ>]
(3) Natural order:
[<Funkies, "Lambda Dancing", 10, 2018, POP>,
<Genericos, "Hot Generics", 10, 2018, JAZZ>,
<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
<Jaav, "Java Jam", 6, 2017, JAZZ>,
<Jaav, "Java Jive", 8, 2017, POP>]
(4) Reversed natural order:
[<Jaav, "Java Jive", 8, 2017, POP>,
<Jaav, "Java Jam", 6, 2017, JAZZ>,
<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
<Genericos, "Hot Generics", 10, 2018, JAZZ>,
<Funkies, "Lambda Dancing", 10, 2018, POP>]
(5) Only Jazz CDs, ordered by title:
[Hot Generics, Java Jam, Keep on Erasing]
(6) No. of tracks >= 8, ordered by number of tracks:
[<Jaav, "Java Jive", 8, 2017, POP>,
<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
<Funkies, "Lambda Dancing", 10, 2018, POP>,
<Genericos, "Hot Generics", 10, 2018, JAZZ>]

```

## Setting a Stream as Unordered

The `unordered()` intermediate operation does not actually reorder the elements in the stream to make them unordered. It just removes *the ordered constraint* on a stream if this constraint is set for the stream, indicating that stream operations can choose to ignore its encounter order. Indicating the stream to be unordered can improve the performance of some operations. For example, the `limit()`, `skip()`, and `distinct()` operations can improve performance when executed on unordered parallel streams, since they can process *any* elements by ignoring the encounter order. The removal of the ordered constraint can impact the performance of certain operations on parallel streams (p. 1015).

It clearly makes sense to call the `unordered()` method on an ordered stream only if the order is of no consequence in the final result. There is no method called `ordered()` to impose an order on a stream. However, the `sorted()` intermediate operation can be used to enforce a sort order on the output stream.

In the stream pipeline below, the `unordered()` method clears the ordered constraint on the stream whose elements have the same order as in the data source—that is, the positional order in the list of CDs. The outcome of the execution shows that the titles in the result list are in the same order as they are in the data source; this is the same result one would get without the `unordered()` operation. It is up to the stream operation to take into consideration that the stream is unordered. The fact that the result list retains the order does not make it invalid. After all, since the stream is set as unordered, it indicates that ignoring the order is at the discretion of the stream operation.

```
//Query: Create a list with the first 2 Jazz CD titles.
List<String> first2JazzCDTitles = CD.cdList
    .stream()
    .unordered()                // Don't care about ordering.
    .filter(CD::isJazz)
    .limit(2)
    .map(CD::title)
    .toList();                 // [Java Jam, Keep on Erasing]
```

The following method is inherited by the `Stream<T>` interface from its superinterface `BaseStream`. Analogous methods are also inherited by the `IntStream`, `LongStream`, and `DoubleStream` interfaces from the superinterface `BaseStream`.

```
Stream<T> unordered()
```

Returns an unordered sequential stream that has the same elements as this stream. The method returns itself if this stream is already unordered. The method can only indicate that the encounter order of this stream can be ignored, and an operation might not comply to this request.

This is an intermediate operation that does not change the stream size or the stream element type. It only indicates that the encounter order can be ignored.

## Execution Mode of a Stream

The two methods `parallel()` and `sequential()` are intermediate operations that can be used to set the execution mode of a stream—that is, whether it will execute sequentially or in parallel. Only the `Collection.parallelStream()` method creates a parallel stream from a collection, so the default mode of execution for most streams is sequential, unless the mode is specifically changed by calling the `parallel()` method. The execution mode of a stream can be switched between sequential and parallel execution at any point between stream creation and the terminal operation in the pipeline. However, it is the *last* call to any of these methods that determines the execution mode for the *entire* pipeline, regardless of how many times these methods are called in the pipeline.

The declaration statements below show examples of both sequential and parallel streams. No stream pipeline is executed, as no terminal operation is invoked on any of the streams. However, when a terminal operation is invoked on one of the streams, the stream will be executed in the mode indicated for the stream.

```
Stream<CD> seqStream1
    = CD.cdList.stream().filter(CD::isPop);           // Sequential
Stream<CD> seqStream2
    = CD.cdList.stream().sequential().filter(CD::isPop); // Sequential
Stream<CD> seqStream3
    = CD.cdList.stream().parallel().filter(CD::isPop).sequential(); // Sequential
Stream<CD> paraStream1
    = CD.cdList.stream().parallel().filter(CD::isPop); // Parallel
Stream<CD> paraStream2
    = CD.cdList.stream().filter(CD::isPop).parallel(); // Parallel
```

The `isParallel()` method can be used to determine the execution mode of a stream. For example, the call to the `isParallel()` method on `seqStream3` below shows that

this stream is a sequential stream. It is the call to the `sequential()` method that occurs last in the pipeline that determines the execution mode.

```
System.out.println(seqStream3.isParallel()); // false
```

Parallel streams are explored further in §16.9, p. 1009.

The following methods are inherited by the `Stream<T>` interface from its superinterface `BaseStream`. Analogous methods are also inherited by the `IntStream`, `LongStream`, and `DoubleStream` interfaces from the superinterface `BaseStream`.

```
Stream<T> parallel()
Stream<T> sequential()
```

Set the execution mode of a stream. They return a parallel or a sequential stream that has the same elements as this stream, respectively. Each method will return itself if this stream is already parallel or sequential, respectively.

These are intermediate operations that do not change the stream size, the stream element type, or the encounter order.

```
boolean isParallel()
```

Returns whether this stream would execute in parallel when the terminal operation is invoked. The method might yield unpredictable results if called after a terminal operation has been invoked.

It is *not* an intermediate operation.

## Converting between Stream Types

Table 16.2 provides a summary of interoperability between stream types—that is, transforming between different stream types. Where necessary, the methods are shown with the name of the built-in functional interface required as a parameter. Selecting a naming convention for method names makes it easy to select the right method for transforming one stream type to another.

**Table 16.2** *Interoperability between Stream Types*

Stream types	To Stream<R>	To IntStream	To LongStream	To DoubleStream
<b>From Stream&lt;T&gt;</b>	<code>map(Function)</code>	<code>mapToInt(ToIntFunction)</code>	<code>mapToLong(ToLongFunction)</code>	<code>mapToDouble(ToDoubleStream)</code>
	<code>flatMap(Function)</code>	<code>flatMapToInt(Function)</code>	<code>flatMapToLong(Function)</code>	<code>flatMapToDouble(Function)</code>
<b>From IntStream</b>	<code>mapToObj(IntFunction)</code>	<code>map(IntUnaryOperator)</code>	<code>mapToLong(IntToLongFunction)</code>	<code>mapToDouble(IntToDoubleFunction)</code>
	<code>Stream&lt;Integer&gt; boxed()</code>	<code>flatMap(IntFunction)</code>	<code>asLongStream()</code>	<code>asDoubleStream()</code>

**Table 16.2** *Interoperability between Stream Types (Continued)*

Stream types	To Stream<R>	To IntStream	To LongStream	To DoubleStream
<b>From LongStream</b>	mapToObj( LongFunction)	mapToInt( LongToInt- Function)	map( LongUnary- Operator)	mapToDouble( LongToDouble- Function)
	Stream<Long> boxed()		flatMap( LongFunction)	asDoubleStream()
<b>From DoubleStream</b>	mapToObj( DoubleFunction)	mapToInt( DoubleToInt- Function)	mapToLong( DoubleToLong- Function)	map( DoubleUnary- Operator)
	Stream<Double> boxed()			flatMap( DoubleFunction)

### Mapping between Object Streams

The `map()` and `flatMap()` methods of the `Stream<T>` interface transform an object stream of type `T` to an object stream of type `R`. Examples using these two methods can be found in §16.5, p. 921, and §16.5, p. 924, respectively.

### Mapping an Object Stream to a Numeric Stream

The `mapToNumType()` methods in the `Stream<T>` interface transform an object stream to a stream of the designated numeric type, where *NumType* is either `Int`, `Long`, or `Double`.

The query below sums the number of tracks for all CDs in a list. The `mapToInt()` intermediate operation at (2) accepts an `IntFunction` that extracts the number of tracks in a CD, thereby transforming the `Stream<CD>` created at (1) into an `IntStream`. The terminal operation `sum()`, as the name implies, sums the values in the `IntStream` (p. 973).

```
int totalNumOfTracks = CD.cdList
    .stream()                // (1) Stream<CD>
    .mapToInt(CD::noOfTracks) // (2) IntStream
    .sum();                 // 42
```

The `flatMapToNumType()` methods are only defined by the `Stream<T>` interface to flatten a multilevel object stream to a numeric stream, where *NumType* is either `Int`, `Long`, or `Double`.

Earlier we saw an example of flattening a two-dimensional array using the `flatMapToInt()` method (p. 924).

The query below sums the number of tracks for all CDs in *two* CD lists. The `flatMapToInt()` intermediate operation at (1) accepts a `Function` that maps each `List<CD>` in a `Stream<List<CD>>` to an `IntStream` whose values are the number of tracks in a CD contained in the list. The resulting `Stream<IntStream>` from the mapper function is flattened into an `IntStream` by the `flatMapToInt()` intermediate operation, thus



### Mapping between Numeric Streams

In contrast to the methods in the `Stream<T>` interface, the `map()` and the `flatMap()` methods of the numeric stream interfaces transform a numeric stream to a numeric stream of the *same* primitive type; that is, they do *not* change the type of the numeric stream.

The `map()` operation in the stream pipeline below does not change the type of the initial `IntStream`.

```
IntStream.rangeClosed(1, 3)           // IntStream
    .map(i -> i * i)                   // IntStream
    .forEach(n -> System.out.printf("%d ", n)); // 1 4 9
```

The `flatMap()` operation in the stream pipeline below also does not change the type of the initial stream. Each `IntStream` created by the mapper function is flattened, resulting in a single `IntStream`.

```
IntStream.rangeClosed(1, 3)           // IntStream
    .flatMap(i -> IntStream.rangeClosed(1, 4)) // IntStream
    .forEach(n -> System.out.printf("%d ", n)); // 1 2 3 4 1 2 3 4 1 2 3 4
```

Analogous to the methods in the `Stream<T>` interface, the `mapToNumType()` methods in the numeric stream interfaces transform a numeric stream to a stream of the designated numeric type, where *NumType* is either `Int`, `Long`, or `Double`.

The `mapToDouble()` operation in the stream pipeline below transforms the initial `IntStream` into a `DoubleStream`.

```
IntStream.rangeClosed(1, 3)           // IntStream
    .mapToDouble(i -> Math.sqrt(i))    // DoubleStream
    .forEach(d -> System.out.printf("%.2f ", d)); // 1.00 1.41 1.73
```

The methods `asLongStream()` and `asDoubleStream()` in the `IntStream` interface transform an `IntStream` to a `LongStream` and a `DoubleStream`, respectively. Similarly, the method `asDoubleStream()` in the `LongStream` interface transforms a `LongStream` to a `DoubleStream`.

The `asDoubleStream()` operation in the stream pipeline below transforms the initial `IntStream` into a `DoubleStream`. Note how the range of `int` values is thereby transformed to a range of `double` values by the `asDoubleStream()` operation.

```
IntStream.rangeClosed(1, 3)           // IntStream
    .asDoubleStream()                  // DoubleStream
    .map(d -> Math.sqrt(d))            // DoubleStream
    .forEach(d -> System.out.printf("%.2f ", d)); // 1.00 1.41 1.73
```

In the stream pipeline below, the `int` values in the `IntStream` are first boxed into `Integers`. In other words, the initial `IntStream` is transformed into an *object stream*, `Stream<Integer>`. The `map()` operation transforms the `Stream<Integer>` into a `Stream<Double>`. In contrast to using the `asDoubleStream()` in the stream pipeline above, note the boxing/unboxing that occurs in the stream pipeline below in the evaluation of the `Math.sqrt()` method, as this method accepts a `double` as a parameter and returns a `double` value.

```

IntStream.rangeClosed(1, 3)           // IntStream
        .boxed()                       // Stream<Integer>
        .map(n -> Math.sqrt(n))       // Stream<Double>
        .forEach(d -> System.out.printf("%.2f ", d)); // 1.00 1.41 1.73

```

## Summary of Intermediate Stream Operations

Table 16.3 summarizes selected aspects of the intermediate operations.

**Table 16.3** Selected Aspects of Intermediate Stream Operations

Intermediate operation	Stateful/ Stateless	Can change stream size	Can change stream type	Encounter order
distinct (p. 915)	<i>Stateful</i>	<i>Yes</i>	<i>No</i>	<i>Unchanged</i>
dropWhile (p. 913)	<i>Stateful</i>	<i>Yes</i>	<i>No</i>	<i>Unchanged</i>
filter (p. 910)	<i>Stateless</i>	<i>Yes</i>	<i>No</i>	<i>Unchanged</i>
flatMap (p. 921)	<i>Stateless</i>	<i>Yes</i>	<i>Yes</i>	<i>Not guaranteed</i>
limit (p. 917)	<i>Stateful, short-circuited</i>	<i>Yes</i>	<i>No</i>	<i>Unchanged</i>
map (p. 921)	<i>Stateless</i>	<i>No</i>	<i>Yes</i>	<i>Not guaranteed</i>
mapMulti (p. 927)	<i>Stateless</i>	<i>Yes</i>	<i>Yes</i>	<i>Not guaranteed</i>
parallel (p. 933)	<i>Stateless</i>	<i>No</i>	<i>No</i>	<i>Unchanged</i>
peek (p. 920)	<i>Stateless</i>	<i>No</i>	<i>No</i>	<i>Unchanged</i>
sequential (p. 933)	<i>Stateless</i>	<i>No</i>	<i>No</i>	<i>Unchanged</i>
skip (p. 915)	<i>Stateful</i>	<i>Yes</i>	<i>No</i>	<i>Unchanged</i>
sorted (p. 929)	<i>Stateful</i>	<i>No</i>	<i>No</i>	<i>Ordered</i>
takeWhile (p. 913)	<i>Stateful, short-circuited</i>	<i>Yes</i>	<i>No</i>	<i>Unchanged</i>
unordered (p. 932)	<i>Stateless</i>	<i>No</i>	<i>No</i>	<i>Not guaranteed</i>

The intermediate operations of the `Stream<T>` interface (including those inherited from its superinterface `BaseStream<T, Stream<T>>`) are summarized in Table 16.4. The type parameter declarations have been simplified, where any bounds `<? super T>` or `<? extends T>` have been replaced by `<T>`, without impacting the intent of a method. A reference is provided to each method in the first column. Any type parameter and return type declared by these methods are shown in column two.

The last column in Table 16.4 indicates the function type of the corresponding parameter in the previous column. It is instructive to note how the functional interface parameters provide the parameterized behavior of an operation. For example,

the `filter()` method returns a stream whose elements satisfy a given predicate. This predicate is defined by the functional interface `Predicate<T>` that is implemented by a lambda expression or a method reference, and applied to each element in the stream.

The interfaces `IntStream`, `LongStream`, and `DoubleStream` also define analogous methods to those shown in Table 16.4, except for the `flatMapToNumType()` methods, where `NumType` is either `Int`, `Long`, or `Double`. A summary of additional methods defined by these numeric stream interfaces can be found in Table 16.2.

**Table 16.4** *Intermediate Stream Operations*

Method name	Any type parameter + return type	Functional interface parameters	Function type of parameters
<code>distinct</code> (p. 915)	<code>Stream&lt;T&gt;</code>	<code>()</code>	
<code>dropWhile</code> (p. 913)	<code>Stream&lt;T&gt;</code>	<code>(Predicate&lt;T&gt; predicate)</code>	<code>T -&gt; boolean</code>
<code>filter</code> (p. 910)	<code>Stream&lt;T&gt;</code>	<code>(Predicate&lt;T&gt; predicate)</code>	<code>T -&gt; boolean</code>
<code>flatMap</code> (p. 921)	<code>&lt;R&gt; Stream&lt;R&gt;</code>	<code>(Function&lt;T,Stream&lt;R&gt;&gt; mapper)</code>	<code>T -&gt; Stream&lt;R&gt;</code>
<code>flatMapToDouble</code> (p. 921)	<code>DoubleStream</code>	<code>(Function&lt;T,DoubleStream&gt; mapper)</code>	<code>T -&gt; DoubleStream</code>
<code>flatMapToInt</code> (p. 921)	<code>IntStream</code>	<code>(Function&lt;T,IntStream&gt; mapper)</code>	<code>T -&gt; IntStream</code>
<code>flatMapToLong</code> (p. 921)	<code>LongStream</code>	<code>(Function&lt;T,LongStream&gt; mapper)</code>	<code>T -&gt; LongStream</code>
<code>limit</code> (p. 917)	<code>Stream&lt;T&gt;</code>	<code>(long maxSize)</code>	
<code>map</code> (p. 921)	<code>&lt;R&gt; Stream&lt;R&gt;</code>	<code>(Function&lt;T,R&gt; mapper)</code>	<code>T -&gt; R</code>
<code>mapMulti</code> (p. 927)	<code>&lt;R&gt; Stream&lt;R&gt;</code>	<code>(BiConsumer&lt;T,Consumer&lt;R&gt;&gt; mapper)</code>	<code>(T, Consumer&lt;R&gt;) -&gt; void</code>
<code>mapToDouble</code> (p. 921)	<code>DoubleStream</code>	<code>(ToDoubleFunction&lt;T&gt; mapper)</code>	<code>T -&gt; double</code>
<code>mapToInt</code> (p. 921)	<code>IntStream</code>	<code>(ToIntFunction&lt;T&gt; mapper)</code>	<code>T -&gt; int</code>
<code>mapToLong</code> (p. 921)	<code>LongStream</code>	<code>(ToLongFunction&lt;T&gt; mapper)</code>	<code>T -&gt; long</code>
<code>parallel</code> (p. 933)	<code>Stream&lt;T&gt;</code>	<code>()</code>	
<code>peek</code> (p. 920)	<code>Stream&lt;T&gt;</code>	<code>(Consumer&lt;T&gt; action)</code>	<code>T -&gt; void</code>
<code>sequential</code> (p. 933)	<code>Stream&lt;T&gt;</code>	<code>()</code>	
<code>skip</code> (p. 915)	<code>Stream&lt;T&gt;</code>	<code>(long n)</code>	
<code>sorted</code> (p. 929)	<code>Stream&lt;T&gt;</code>	<code>()</code>	
<code>sorted</code> (p. 929)	<code>Stream&lt;T&gt;</code>	<code>(Comparator&lt;T&gt; cmp)</code>	<code>(T,T) -&gt; int</code>

Table 16.4 *Intermediate Stream Operations (Continued)*

Method name	Any type parameter + return type	Functional interface parameters	Function type of parameters
takeWhile (p. 913)	Stream<T>	(Predicate<T> predicate)	T -> boolean
unordered (p. 932)	Stream<T>	()	

## 16.6 The Optional Class

When a method returns a null value, it is not always clear whether the null value represents a valid value or the *absence of a value*. Methods that can return null values invariably force their callers to check the returned value explicitly in order to avoid a `NullPointerException` before using the returned value. For example, method chaining, which we have seen for composing stream pipelines, becomes cumbersome if each method call must be checked to see whether it returns a null value before calling the next method, resulting in a cascade of conditional statements.

The concept of an `Optional` object allows the absence of a value to be handled in a systematic way, making the code robust by enforcing that a consumer of an `Optional` must also deal with the case when the value is absent. Taking full advantage of `Optional` wrappers requires using them the right way, primarily to handle situations where the value returned by a method is absent.

The generic class `Optional<T>` provides a wrapper that represents either the presence or absence of a non-null value of type `T`. In other words, the wrapper either contains a non-null value of type `T` or no value at all.

Example 16.8 illustrates using objects of the `Optional<T>` class.

### Declaring and Returning an `Optional`

Example 16.8 illustrates declaring and returning an `Optional`. A book is represented by the `Book` class that has an optional blurb of type `String`; that is, a book may or may not have a blurb. The `Optional<T>` class is parameterized with the type `String` in the declaration, and so is the return type of the method that returns the optional blurb.

```
class Book {
    private Optional<String> optBlurb;

    public Optional<String> getOptBlurb() { return optBlurb; }

    //...
}
```

## Creating an Optional

The `Optional<T>` class models the absence of a value by a special singleton returned by the `Optional.empty()` method. In contrast to the `null` value, this singleton is a viable `Optional` object on which methods of the `Optional` class can be invoked without a `NullPointerException` being thrown.

```
static <T> Optional<T> empty()
static <T> Optional<T> of(T nonNullValue)
static <T> Optional<T> ofNullable(T value)
```

The `empty()` method returns an empty `Optional` instance; that is, it indicates *the absence of a value*.

The `of()` method returns an `Optional` with the specified value, if this value is non-`null`. Otherwise, a `NullPointerException` is thrown.

The `ofNullable()` method returns an `Optional` with the specified value, if this value is non-`null`. Otherwise, it returns an empty `Optional`.

The static `Optional.of()` factory method creates an `Optional` that encapsulates the non-`null` argument specified in the method call, as in the first declaration below. However, if the argument is a `null` value, a `NullPointerException` is thrown at run-time, as in the second declaration.

```
Optional<String> blurb0 = Optional.of("Java Programmers tell all!");
Optional<String> xblurb = Optional.of(null); // NullPointerException
```

The static `Optional.ofNullable()` factory method also creates an `Optional` that encapsulates the non-`null` argument specified in the method call, as in the first declaration below. However, if the argument is a `null` value, the method returns an *empty* `Optional`, as in the second declaration—which is effectively the same as the third declaration below.

```
Optional<String> blurb1 = Optional.ofNullable("Program like a Java Pro!");
Optional<String> noBlurb2 = Optional.ofNullable(null); // Optional.empty()
Optional<String> noBlurb3 = Optional.empty();
```

The blurbs above are used to initialize two `Book` objects (`book0`, `book1`) in Example 16.8. These `Book` objects with optional blurbs will be used to illustrate how to use `Optional` objects.

---

### Example 16.8 Using Optionals

```
// File: OptionalUsage.java
import java.util.Optional;

// A book can have an optional blurb.
class Book {
    private String bookName;
    private Optional<String> optBlurb;

    public String getBookName() { return bookName; }
    public Optional<String> getOptBlurb() { return optBlurb; }
```

```

public Book(String bookName, Optional<String> optBlurb) {
    this.bookName = bookName;
    this.optBlurb = optBlurb;
}
}

// A course can have an optional book.
class Course {
    private Optional<Book> optBook;
    public Optional<Book> getOptBook() { return optBook; }
    public Course(Optional<Book> optBook) { this.optBook = optBook; }
}

public class OptionalUsage {
    public static void main(String[] args) {

        // Creating an Optional:
        Optional<String> blurb0 = Optional.of("Java Programmers tell all!");
        //Optional<String> xblurb = Optional.of(null); // NullPointerException
        Optional<String> blurb1 = Optional.ofNullable("Program like a Java Pro!");
        Optional<String> noBlurb2 = Optional.ofNullable(null); // Optional.empty()
        Optional<String> noBlurb3 = Optional.empty();

        // Create some books:
        Book book0 = new Book("Embarrassing Exceptions", blurb0);
        Book book1 = new Book("Dancing Lambdas", noBlurb2); // No blurb.

        // Querying an Optional:
        if (book0.getOptBlurb().isPresent()) {
            System.out.println(book0.getOptBlurb().get()); //Java Programmers tell all!
        }

        book0.getOptBlurb()
            .ifPresent(System.out::println); //Java Programmers tell all!

// System.out.println(book1.getOptBlurb().get()); // NoSuchElementException

        String blurb = book0.getOptBlurb()
            .orElse("No blurb"); // "Java Programmers tell all!"
        System.out.println(blurb);

        blurb = book1.getOptBlurb().orElse("No blurb"); // "No blurb"
        System.out.println(blurb);

        blurb = book1.getOptBlurb().orElseGet(() -> "No blurb"); // "No blurb"
        System.out.println(blurb);

        //blurb = book1.getOptBlurb() // RuntimeException
        // .orElseThrow(() -> new RuntimeException("No blurb"));
    }
}

```

Output from the program:

```

Java Programmers tell all!
Java Programmers tell all!

```

```

Java Programmers tell all!
No blurb
No blurb

```

## Querying an Optional

The presence of a value in an `Optional` can be determined by the `isPresent()` method, and the value can be obtained by calling the `get()` method—which is not much better than checking explicitly for the `null` value, but as we shall see, other methods in the `Optional` class alleviate this drudgery. The `get()` method throws a `NoSuchElementException` if there is no value in the `Optional`.

```

if (book0.getOptBlurb().isPresent()) {
    System.out.println(book0.getOptBlurb().get()); // Java Programmers tell all!
}
System.out.println(book1.getOptBlurb().get()); // NoSuchElementException

```

The idiom of determining the presence of a value and then handling the value is combined by the `ifPresent()` method that accepts a `Consumer` to handle the value if one is present. The `ifPresent()` method does nothing if there is no value present in the `Optional`.

```

book0.getOptBlurb().ifPresent(System.out::println); //Java Programmers tell all!

```

**T** `get()`

If a value is present in this `Optional`, the method returns that value; otherwise, it throws a `NoSuchElementException`.

```

boolean isPresent()
void ifPresent(Consumer<? super T> consumer)

```

The first method returns `true` if there is a value present in this `Optional`; otherwise, it returns `false`.

If a value is present in this `Optional`, the second method invokes the specified consumer with the value; otherwise, it does nothing.

```

T orElse(T other)
T orElseGet(Supplier<? extends T> other)
<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
    throws X extends Throwable

```

If a value is present in this `Optional`, all three methods return this value.

They differ in their action when a value is not present in this `Optional`. The first method returns the other value. The second method invokes the specified supplier `other` and returns its result. The third method invokes the specified supplier `exceptionSupplier` and throws the exception created by this supplier.

Note that the type of argument in the first two methods must be compatible with the parameterized type of the `Optional` on which the method is invoked, or the compiler will issue an error.

Often, a default value should be supplied when an `Optional` does *not* contain a value. The `orElse()` method returns the value in the `Optional` if one is present; otherwise, it returns the value given by the argument specified in the method call.

The `orElse()` method in the statement below returns the blurb in the book referenced by the reference `book0`, as this book has a blurb.

```
String blurb = book0.getOptBlurb()
                .orElse("No blurb");           // "Java Programmers tell all!"
```

The book referenced by the reference `book1` has no blurb. Therefore, the `orElse()` method invoked on the optional `blurb` returns the argument in the method.

```
blurb = book1.getOptBlurb().orElse("No blurb"); // "No blurb"
```

For an `Optional` with a value, the `orElseGet()` method returns the value in the `Optional`. The `orElseGet()` method in the statement below returns the object supplied by the `Supplier` specified as an argument, since the book has no blurb.

```
blurb = book1.getOptBlurb().orElseGet(() -> "No blurb"); // "No blurb"
```

For an `Optional` with a value, the `orElseThrow()` method also returns the value in the `Optional`. The `orElseThrow()` method in the statement below throws the exception created by the `Supplier` specified as an argument, since the book has no blurb.

```
blurb = book1.getOptBlurb()                // RuntimeException
                .orElseThrow(() -> new RuntimeException("No blurb"));
```

## Numeric Optional Classes

An instance of the generic `Optional<T>` class can only encapsulate an object. To deal with optional numeric values, the `java.util` package also defines the following non-generic classes that can encapsulate primitive numeric values: `OptionalInt`, `OptionalLong`, and `OptionalDouble`. For example, an `OptionalInt` object encapsulates an `int` value.

The numeric optional classes provide methods analogous to the static factory methods of the `Optional` class to create a numeric optional from a numeric value and methods to query a numeric optional. The `filter()`, `map()`, and `flatMap()` methods are *not* defined for the numeric optional classes.

The following methods are defined in the `OptionalInt`, `OptionalLong`, and `OptionalDouble` classes in the `java.util` package. In the methods below, *NumType* is `Int`, `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`.

```
static OptionalNumType empty()
static OptionalNumType of(numtype value)
```

These two methods return an empty `OptionalNumType` instance and an `OptionalNumType` with the specified value, respectively.

```
boolean isPresent()
void    ifPresent(NumTypeConsumer consumer)
```

If there is a value present in this `Optional`, the first method returns `true`. Otherwise, it returns `false`.

If a value is present in this `Optional`, the specified consumer is invoked on the value. Otherwise, it does nothing.

```
NumType getAsNumType()
```

If a value is present in this `OptionalNumType`, the method returns the value. Otherwise, it throws a `NoSuchElementException`.

```
NumType orElse(NumType other)
NumType orElseGet(NumTypeSupplier other)
<X extends Throwable> NumType orElseThrow(Supplier<X> exceptionSupplier)
                                throws X extends Throwable
```

If a value is present in this `OptionalNumType`, all three methods return this value.

They differ in their action when there is no value present in this numeric optional. The first method returns the specified other value. The second method invokes the specified other supplier and returns the result. The third method throws an exception that is created by the specified supplier.

Example 16.9 illustrates using numeric optional values. A recipe has an optional number of calories that are modeled using an `OptionalInt` that can encapsulate an `int` value. Declaring, creating, and querying `OptionalInt` objects is analogous to that for `Optional` objects.

---

#### Example 16.9 Using Numerical Optionals

```
// File: NumericOptionalUsage.java
import java.util.OptionalInt;

class Recipe {
    private String recipeName;
    private OptionalInt calories;    // Optional number of calories.

    public String getRecipeName() { return recipeName; }
    public OptionalInt getCalories() { return calories; }

    public Recipe(String recipeName, OptionalInt calories) {
        this.recipeName = recipeName;
        this.calories = calories;
    }
}

public final class NumericOptionalUsage {
    public static void main(String[] args) {
        // Creating an OptionalInt:
        OptionalInt optNOCO = OptionalInt.of(3500);
        OptionalInt optNOC1 = OptionalInt.empty();
    }
}
```

```

// Creating recipes with optional number of calories:
Recipe recipe0 = new Recipe("Mahi-mahi", optNOC0);
Recipe recipe1 = new Recipe("Loco moco", optNOC1);

// Querying an Optional:
// System.out.println(recipe1.getCalories()
//                      .getAsInt());           // NoSuchElementException
System.out.println((recipe1.getCalories().isPresent()
    ? recipe1.getCalories().getAsInt()
    : "Unknown calories.));           // Unknown calories.

recipe0.getCalories().ifPresent(s -> System.out.println(s + " calories.));
System.out.println(recipe0.getCalories().orElse(0) + " calories.");
System.out.println(recipe1.getCalories().orElseGet(() -> 0) + " calories.");
// int noc = recipe1.getCalories()           // RuntimeException
//                      .orElseThrow(() -> new RuntimeException("Unknown calories.));
}
}

```

Output from the program:

```

Unknown calories.
3500 calories.
3500 calories.
0 calories.

```

.....

## 16.7 Terminal Stream Operations

---

A stream pipeline does not execute until a terminal operation is invoked on it; that is, a stream pipeline does not start to process the stream elements until a terminal operation is initiated. A terminal operation is said to be *eager* as it executes immediately when invoked—as opposed to an intermediate operation which is *lazy*. Invoking the terminal operation results in the intermediate operations of the stream pipeline to be executed. Understandably, a terminal operation is specified as the last operation in a stream pipeline, and there can only be one such operation in a stream pipeline. A terminal operation never returns a stream, which is always done by an intermediate operation. Once the terminal operation completes, the stream is consumed and cannot be reused.

Terminal operations can be broadly grouped into three groups:

- *Operations with side effects*

The Stream API provides two terminal operations, `forEach()` and `forEachOrdered()`, that are designed to allow side effects on stream elements (p. 948). These terminal operations do not return a value. They allow a Consumer action, specified as an argument, to be applied to every element, as they are consumed from the stream pipeline—for example, to print each element in the stream.

- *Searching operations*

These operations perform a search operation to determine a match or find an element as explained below.

All search operations are *short-circuit operations*; that is, the operation can terminate once the result is determined, whether or not all elements in the stream have been considered.

Search operations can be further classified into two subgroups:

- *Matching operations*

The three terminal operations `anyMatch()`, `allMatch()`, and `noneMatch()` determine whether stream elements match a given Predicate specified as an argument to the method (p. 949). As expected, these operations return a `boolean` value to indicate whether the match was successful or not.

- *Finding operations*

The two terminal operations `findAny()` and `findFirst()` find any element and the first element in a stream, respectively, if such an element is available (p. 952). As the stream might be empty and such an element might not exist, these operations return an `Optional`.

- *Reduction operations*

A *reduction* operation computes a result from combining the stream elements by successively applying a combining function; that is, the stream elements are *reduced* to a result value. Examples of reductions are computing the sum or average of numeric values in a numeric stream, and accumulating stream elements into a collection.

We distinguish between two kinds of reductions:

- *Functional reduction*

A terminal operation is a functional reduction on the elements of a stream if it reduces the elements to a *single immutable value* which is then returned by the operation.

The overloaded `reduce()` method provided by the Stream API can be used to implement customized functional reductions (p. 955), whereas the terminal operations `count()`, `min()`, and `max()` implement specialized functional reductions (p. 953).

Functional reductions on numeric streams are discussed later in this section (p. 972).

- *Mutable reduction*

A terminal operation performs a *mutable reduction* on the elements of a stream if it uses a *mutable container*—for example, a list, a set, or a map—to accumulate values as it processes the stream elements. The operation returns the mutable container as the result of the operation.

The Stream API provides two overloaded `collect()` methods that perform mutable reduction (p. 964). One overloaded `collect()` method can be used to

implement customized mutable reductions by specifying the functions (*supplier*, *accumulator*, *combiner*) required to perform such a reduction. A second `collect()` method accepts a `Collector` that is used to perform a mutable reduction. A *collector* encapsulates the functions required for performing a mutable reduction. The Stream API provides built-in collectors that allow various containers to be used for performing mutable reductions (p. 978). When a terminal operation performs a mutable reduction using a specific container, it is said to *collect to* this container.

The `toArray()` method implements a specialized mutable reduction that returns an array with the accumulated values (p. 971); that is, the method collects to an array.

## Consumer Action on Stream Elements

We have already used both the `forEach()` and `forEachOrdered()` terminal operations to print elements when the pipeline is executed. These operations allow side effects on stream elements.

The `forEach()` method is defined for both streams and collections. In the case of collections, the method iterates over all the elements in the collection, whereas it is a terminal operation on streams.

Since these terminal operations perform an action on each element, the input stream to the operation must be *finite* in order for the operation to terminate.

Counterparts to the `forEach()` and `forEachOrdered()` methods for the primitive numeric types are also defined by the numeric stream interfaces.

```
void forEach(Consumer<? super T> action)
```

This terminal operation performs an action on each element of this stream. This method should not be relied upon to produce deterministic results, as the order in which the elements are processed is not guaranteed.

```
void forEachOrdered(Consumer<? super T> action)
```

This terminal operation performs an action on each element of this stream, but in the encounter order of the stream if the stream has one.

The difference in behavior of the `forEach()` and `forEachOrdered()` terminal operations is that the `forEach()` method does not guarantee to respect the encounter order, whereas the `forEachOrdered()` method always does, if there is one.

Each operation is applied to both an ordered sequential stream and an ordered parallel stream to print CD titles with the help of the consumer `printStr`:

```
Consumer<String> printStr = str -> System.out.print(str + "|");

CD.cdList.stream().map(CD::title).forEach(printStr);           // (1a)
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|
```

```
CD.cdList.stream().parallel().map(CD::title).forEach(printStr); // (1b)
//Lambda Dancing|Hot Generics|Keep on Erasing|Java Jam|Java Jive|
```

The behavior of the `forEach()` operation is nondeterministic, as seen at (1a) and (1b). The output from (1a) and (1b) shows that the `forEach()` operation respects the encounter order for an ordered sequential stream, but not necessarily for an ordered parallel stream. Respecting the encounter order for an ordered parallel stream would incur overhead that would impact performance, and is therefore ignored.

On the other hand, the `forEachOrdered()` operation always respects the encounter order in both cases, as seen below from the output at (2a) and (2b). However, it is important to note that, in the case of the ordered parallel stream, the terminal action on the elements can be executed in different threads, but guarantees that the action is applied to the elements in encounter order.

```
CD.cdList.stream().map(CD::title).forEachOrdered(printStr); // (2a)
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|
```

```
CD.cdList.stream().parallel().map(CD::title).forEachOrdered(printStr); // (2b)
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|
```

The discussion above also applies when the `forEach()` and `forEachOrdered()` terminal operations are invoked on numeric streams. The nondeterministic behavior of the `forEach()` terminal operation for `int` streams is illustrated below. The terminal operation on the sequential `int` stream at (3a) seems to respect the encounter order, but should not be relied upon. The terminal operation on the parallel `int` stream at (3b) can give different results for different runs.

```
IntConsumer printInt = n -> out.print(n + "|");
IntStream.of(2018, 2019, 2020, 2021, 2022).forEach(printInt); // (3a)
//2018|2019|2020|2021|2022|
```

```
IntStream.of(2018, 2019, 2020, 2021, 2022).parallel().forEach(printInt); // (3b)
//2020|2019|2018|2021|2022|
```

## Matching Elements

The match operations determine whether any, all, or none of the stream elements satisfy a given Predicate. These operations are not reductions, as they do not always consider all elements in the stream in order to return a result.

Analogous match operations are also provided by the numeric stream interfaces.

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

These three terminal operations determine whether *any*, *all*, or *no* elements of this stream match the specified predicate, respectively.

The methods may not evaluate the predicate on all elements if it is not necessary for determining the result; that is, they are *short-circuit* operations.

If the stream is empty, the predicate is *not* evaluated.

The `anyMatch()` method returns `false` if the stream is empty.

The `allMatch()` and `noneMatch()` methods return `true` if the stream is empty.

There is no guarantee that these operations will terminate if applied to an infinite stream.

The queries at (1), (2), and (3) below determine whether any, all, or no CDs are jazz music CDs, respectively. At (1), the execution of the pipeline terminates as soon as any jazz music CD is found—the value `true` is returned. At (2), the execution of the pipeline terminates as soon as a non-jazz music CD is found—the value `false` is returned. At (3), the execution of the pipeline terminates as soon as a jazz music CD is found—the value `false` is returned.

```
boolean anyJazzCD = CD.cdList.stream().anyMatch(CD::isJazz); // (1) true
boolean allJazzCds = CD.cdList.stream().allMatch(CD::isJazz); // (2) false
boolean noJazzCds = CD.cdList.stream().noneMatch(CD::isJazz); // (3) false
```

Given the following predicates:

```
Predicate<CD> eq2015 = cd -> cd.year().compareTo(Year.of(2015)) == 0;
Predicate<CD> gt2015 = cd -> cd.year().compareTo(Year.of(2015)) > 0;
```

The query at (4) determines that no CDs were released in 2015. The queries at (5) and (6) are equivalent. If all CDs were released after 2015, then none were released in or before 2015 (negation of the predicate `gt2015`).

```
boolean noneEQ2015 = CD.cdList.stream().noneMatch(eq2015); // (4) true
boolean allGT2015 = CD.cdList.stream().allMatch(gt2015); // (5) true
boolean noneNotGT2015 = CD.cdList.stream().noneMatch(gt2015.negate()); // (6) true
```

The code below uses the `anyMatch()` method on an `int` stream to determine whether any year is a leap year.

```
IntStream yrStream = IntStream.of(2018, 2019, 2020);
IntPredicate isLeapYear = yr -> Year.of(yr).isLeap();
boolean anyLeapYear = yrStream.anyMatch(isLeapYear);
out.println("Any leap year: " + anyLeapYear); // true
```

Example 16.10 illustrates using the `allMatch()` operation to determine whether a square matrix—that is, a two-dimensional array with an equal number of columns as rows—is an *identity matrix*. In such a matrix, all elements on the main diagonal have the value 1 and all other elements have the value 0. The methods `isIdentityMatrixLoops()` and `isIdentityMatrixStreams()` at (1) and (2) implement this test in different ways.

The method `isIdentityMatrixLoops()` at (1) uses nested loops. The outer loop processes the rows, whereas the inner loop tests that each row has the correct values. The outer loop is a labeled loop in order to break out of the inner loop if an element in a row does not have the correct value—effectively achieving short-circuit execution.

The method `isIdentityMatrixStreams()` at (2) uses nested numeric streams, where the outer stream processes the rows and the inner stream processes the elements in a row. The `allMatch()` method at (4) in the inner stream pipeline determines that all elements in a row have the correct value. It short-circuits the execution of the inner stream if that is not the case. The `allMatch()` method at (3) in the outer stream pipeline also short-circuits its execution if its predicate to process a row returns the value `false`. The stream-based implementation for the identity matrix test expresses the logic more clearly and naturally than the loop-based version.

---

**Example 16.10** *Identity Matrix Test*

```
import static java.lang.System.out;

import java.util.Arrays;
import java.util.stream.IntStream;

public class IdentityMatrixTest {
    public static void main(String[] args) {
        // Matrices to test:
        int[][] sqMatrix1 = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
        int[][] sqMatrix2 = { {1, 1}, {1, 1} };
        isIdentityMatrixLoops(sqMatrix1);
        isIdentityMatrixLoops(sqMatrix2);
        isIdentityMatrixStreams(sqMatrix1);
        isIdentityMatrixStreams(sqMatrix2);
    }

    private static void isIdentityMatrixLoops(int[][] sqMatrix) {           // (1)
        boolean isCorrectValue = false;
        outerLoop:
        for (int i = 0; i < sqMatrix.length; ++i) {
            for (int j = 0; j < sqMatrix[i].length; ++j) {
                isCorrectValue = j == i ? sqMatrix[i][i] == 1
                    : sqMatrix[i][j] == 0;
                if (!isCorrectValue) break outerLoop;
            }
        }
        out.println(Arrays.deepToString(sqMatrix)
            + (isCorrectValue ? " is " : " is not ") + "an identity matrix.");
    }

    private static void isIdentityMatrixStreams(int[][] sqMatrix) {       // (2)
        boolean isCorrectValue =
            IntStream.range(0, sqMatrix.length)
                .allMatch(i -> IntStream.range(0, sqMatrix[i].length) // (3)
                    .allMatch(j -> j == i // (4)
                        ? sqMatrix[i][i] == 1
                        : sqMatrix[i][j] == 0));
        out.println(Arrays.deepToString(sqMatrix)
            + (isCorrectValue ? " is " : " is not ") + "an identity matrix.");
    }
}
```

Output from the program:

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] is an identity matrix.
[[1, 1], [1, 1]] is not an identity matrix.
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] is an identity matrix.
[[1, 1], [1, 1]] is not an identity matrix.
```

## Finding the First or Any Element

The `findFirst()` method can be used to find the first element that is available in the stream. This method respects the encounter order, if the stream has one. It always produces a stable result; that is, it will produce the same result on identical pipelines based on the same stream source. In contrast, the behavior of the `findAny()` method is nondeterministic. Counterparts to these methods are also defined by the numeric stream interfaces.

`Optional<T> findFirst()`

This terminal operation returns an `Optional` describing the *first* element of this stream, or an empty `Optional` if the stream is empty.

This method may return any element if this stream does not have any encounter order.

It is a *short-circuit* operation, as it will terminate the execution of the stream pipeline as soon as the first element is found.

This method throws a `NullPointerException` if the element selected is `null`.

`Optional<T> findAny()`

This terminal operation returns an `Optional` describing *some* element of the stream, or an empty `Optional` if the stream is empty. This operation has nondeterministic behavior.

It is a *short-circuit* operation, as it will terminate the execution of the stream pipeline as soon as any element is found.

In the code below, the encounter order of the stream is the positional order of the elements in the list. The first element returned by the `findFirst()` method at (1) is the first element in the CD list.

```
Optional<CD> firstCD1 = CD.cdList.stream().findFirst();           // (1)
out.println(firstCD1.map(CD::title).orElse("No first CD.));      // (2) Java Jive
```

Since such an element might not exist—for example, the stream might be empty—the method returns an `Optional<T>` object. At (2), the `Optional<CD>` object returned by the `findFirst()` method is mapped to an `Optional<String>` object that encapsulates the title of the CD. The `orElse()` method on this `Optional<String>` object returns the CD title or the argument string if there is no such CD.

If the encounter order is not of consequence, the `findAny()` method can be used, as it is nondeterministic—that is, it does not guarantee the same result on the same

stream source. On the other hand, it provides maximal performance on parallel streams. At (3) below, the `findAny()` method is free to return any element from the parallel stream. It should not come as a surprise if the element returned is not the first element in the list.

```
Optional<CD> anyCD2 = CD.cdList.stream().parallel().findAny(); // (3)
out.println(anyCD2.map(CD::title).orElse("No CD.")); // Lambda Dancing
```

The `match` methods only determine whether any elements satisfy a `Predicate`, as seen at (5) below. Typically, a `find` terminal operation is used to find the first element made available to the terminal operation after processing by the intermediate operations in the stream pipeline. At (6), the `filter()` operation will filter the jazz music CDs from the stream. However, the `findAny()` operation will return the first jazz music CD that is filtered and then short-circuit the execution.

```
boolean anyJazzCD = CD.cdList.stream().anyMatch(CD::isJazz); // (5)
out.println("Any Jazz CD: " + anyJazzCD); // Any Jazz CD: true

Optional<CD> optJazzCD = CD.cdList.stream().filter(CD::isJazz).findAny(); // (6)
optJazzCD.ifPresent(out::println); // <Jaav, "Java Jam", 6, 2017, JAZZ>
```

The code below uses the `findAny()` method on an `IntStream` to find whether any number is divisible by 7.

```
IntStream numStream = IntStream.of(50, 55, 65, 70, 75, 77);
OptionalInt intOpt = numStream.filter(n -> n % 7 == 0).findAny();
intOpt.ifPresent(System.out::println); // 70
```

The `find` operations are guaranteed to terminate when applied to a finite, albeit empty, stream. However, for an infinite stream in a pipeline, at least one element must be made available to the `find` operation in order for the operation to terminate. If the elements of an initial infinite stream are all discarded by the intermediate operations, the `find` operation will not terminate, as in the following pipeline:

```
Stream.generate(() -> 1).filter(n -> n == 0).findAny(); // Never terminates.
```

## Counting Elements

The `count()` operation performs a functional reduction on the elements of a stream, as each element contributes to the count which is the single immutable value returned by the operation. The `count()` operation reports the number of elements that are made available to it, which is not necessarily the same as the number of elements in the initial stream, as elements might be discarded by the intermediate operations.

The code below finds the total number of CDs in the streams, and how many of these CDs are jazz music CDs.

```
long numOfCDS = CD.cdList.stream().count(); // 5
long numOfJazzCDS = CD.cdList.stream().filter(CD::isJazz).count(); // 3
```

The `count()` method is also defined for the numeric streams. Below it is used on an `IntStream` to find how many numbers between 1 and 100 are divisible by 7.

```
IntStream numStream = IntStream.rangeClosed(1, 100);
long divBy7 = numStream.filter(n -> n % 7 == 0).count();           // 14
```

```
long count()
```

This terminal operation returns the count of elements in this stream—that is, the length of this stream.

This operation is a special case of a functional reduction.

The operation does not terminate when applied to an infinite stream.

## Finding Min and Max Elements

The `min()` and `max()` operations are functional reductions, as they consider all elements of the stream and return a single value. They should only be applied to a finite stream, as they will not terminate on an infinite stream. These methods are also defined by the numeric stream interfaces for the numeric types, but without the specification of a comparator.

```
Optional<T> min(Comparator<? super T> cmp)
Optional<T> max(Comparator<? super T> cmp)
```

These terminal operations return an `Optional` with the minimum or maximum element of this stream according to the provided `Comparator`, respectively, or an empty `Optional` if this stream is empty. It throws a `NullPointerException` if the minimum element is `null`.

These operations are a special case of a functional reduction.

These operations do not terminate when applied to an infinite stream.

Both methods return an `Optional`, as the minimum and maximum elements might not exist—for example, if the stream is empty. The code below finds the minimum and maximum elements in a stream of CDs, according to their natural order. The artist name is the most significant field according to the natural order defined for CDs (p. 883).

```
Optional<CD> minCD = CD.cdList.stream().min(Comparator.naturalOrder());
minCD.ifPresent(out::println);           // <Funkies, "Lambda Dancing", 10, 2018, POP>
out.println(minCD.map(CD::artist).orElse("No min CD."));           // Funkies

Optional<CD> maxCD = CD.cdList.stream().max(Comparator.naturalOrder());
maxCD.ifPresent(out::println);           // <Jaav, "Java Jive", 8, 2017, POP>
out.println(maxCD.map(CD::artist).orElse("No max CD."));           // Jaav
```

In the code below, the `max()` method is applied to an `IntStream` to find the largest number between 1 and 100 that is divisible by 7.

```
IntStream iStream = IntStream.rangeClosed(1, 100);
OptionalInt maxNum = iStream.filter(n -> n % 7 == 0).max(); // 98
```

If one is only interested in the minimum and maximum elements in a collection, the overloaded methods `min()` and `max()` of the `java.util.Collections` class can be more convenient to use.

## Implementing Functional Reduction: The `reduce()` Method

A *functional reduction* combines all elements in a stream to produce a *single immutable value* as its result. The reduction process employs an *accumulator* that repeatedly computes a new partial result based on the current partial result and the current element in the stream. The stream thus gets shorter by one element. When all elements have been combined, the last partial result that was computed by the accumulator is returned as the final result of the reduction process.

The following terminal operations are special cases of functional reduction:

- `count()`, p. 953.
- `min()`, p. 954.
- `max()`, p. 954.
- `average()`, p. 1000.
- `sum()`, p. 1001.

The overloaded `reduce()` method can be used to implement new forms of functional reduction.

`Optional<T> reduce(BinaryOperator<T> accumulator)`

This terminal operation returns an `Optional` with the *cumulative* result of applying the accumulator on the elements of this stream:  $e_1 \oplus e_2 \oplus e_3 \dots$ , where each  $e_i$  is an element of this stream and  $\oplus$  is the accumulator. If the stream is empty, an empty `Optional` is returned.

The accumulator must be *associative*—that is, the result of evaluating an expression is the same, regardless of how the operands are grouped to evaluate the expression. For example, the grouping in the expression below allows the sub-expressions to be evaluated in parallel and their results combined by the accumulator:

$$e_i \oplus e_j \oplus e_k \oplus e_l == (e_i \oplus e_j) \oplus (e_k \oplus e_l)$$

where  $e_i$ ,  $e_j$ ,  $e_k$ , and  $e_l$  are operands, and  $\oplus$  is the accumulator. For example, numeric addition, `min`, `max`, and string concatenation are associative operations, whereas subtraction and division are nonassociative.

The accumulator must also be a *non-interfering* and *stateless* function (p. 909).

Note that the method reduces a `Stream` of type `T` to a result that is an `Optional` of type `T`.

A counterpart to the single-argument `reduce()` method is also provided for the numeric streams.

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

This terminal operation returns the *cumulative* result of applying the accumulator on the elements of this stream:  $\text{identity} \oplus e_1 \oplus e_2 \oplus e_3 \dots$ , where  $e_i$  is an element of this stream, and  $\oplus$  is the accumulator. The *identity* value is the initial value to accumulate. If the stream is empty, the *identity* value is returned.

The *identity* value must be an *identity* for the accumulator—for all  $e_i$ ,  $\text{identity} \oplus e_i == e_i$ . The accumulator must be *associative*.

The accumulator must also be a *non-interfering* and *stateless* function (p. 909).

Note that the method reduces a Stream of type T to a result of type T.

A counterpart to the two-argument `reduce()` method is also provided for the numeric streams.

```
<U> U reduce(
    U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
```

This terminal operation returns the *cumulative* result of applying the accumulator on the elements of this stream, using the *identity* value of type U as the initial value to accumulate. If the stream is empty, the *identity* value is returned.

The *identity* value must be an *identity* for the combiner function. The accumulator and the combiner function must also satisfy the following relationship for all  $u$  and  $t$  of type U and T, respectively:

$$u \odot (\text{identity} \oplus t) == u \oplus t$$

where  $\odot$  and  $\oplus$  are the accumulator and combiner functions, respectively.

The combiner function combines two values during stream processing. It may not be executed for a sequential stream, but for a parallel stream, it will combine cumulative results of segments that are processed concurrently.

Both the accumulator and the combiner must also be *non-interfering* and *stateless* functions (p. 909).

Note that the accumulator has the function type  $(U, T) \rightarrow U$ , and the combiner function has the function type  $(U, U) \rightarrow U$ , where the type parameters U and T are always the types of the partial result and the stream element, respectively. This method reduces a Stream of type T to a result of type U.

There is *no* counterpart to the three-argument `reduce()` method for the numeric streams.

The idiom of using a loop for calculating the sum of a finite number of values is something that is ingrained into all aspiring programmers. A loop-based solution to calculate the total number of tracks on CDs in a list is shown below, where the variable `sum` will hold the result after the execution of the `for(:)` loop:

```
int sum = 0; // (1) Initialize the partial result.
for (CD cd : CD.cdList) { // (2) Iterate over the list.
    int numOfTracks = cd.numOfTracks(); // (3) Get the current value.
    sum = sum + numOfTracks; // (4) Calculate new partial result.
}
```

Apart from the `for(:)` loop at (2) to iterate over all elements of the list and read the number of tracks in each CD at (3), the two necessary steps are:

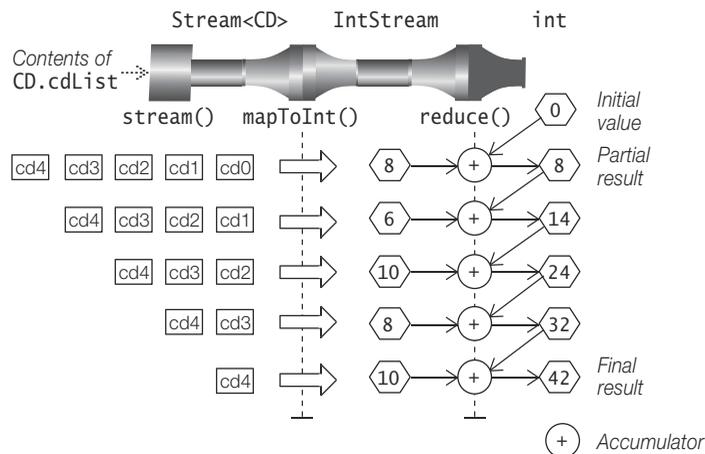
- Initialization of the variable `sum` at (1)
- The accumulative operation at (4) that is applied repeatedly to compute a new partial result in the variable `sum`, based on its previous value and the number of tracks in the current CD

The loop-based solution above can be translated to a stream-based solution, as shown in Figure 16.11. All the code snippets can be found in Example 16.11.

**Figure 16.11** Reducing with an Initial Value

```
// Query: Find the total number of CD tracks.
int totNumOfTracks = CD.cdList // (5)
    .stream() // (6)
    .mapToInt(CD::noOfTracks) // (7)
    .reduce(0, // (8)
        (sum, numOfTracks) -> sum + numOfTracks); // (9)
```

(a) Using the `reduce()` method with an initial value



(b) Stream pipeline

In Figure 16.11, the stream created at (6) internalizes the iteration over the elements. The `mapToInt()` intermediate operation maps each CD to its number of tracks at (7)—the `Stream<CD>` is mapped to an `IntStream`. The `reduce()` terminal operation with two arguments computes and returns the total number of tracks:

- Its first argument at (8) is the *identity* element that provides the initial value for the operation and is also the default value to return if the stream is empty. In this case, this value is 0.
- Its second argument at (9) is the *accumulator* that is implemented as a lambda expression. It repeatedly computes a new partial sum based on the previous partial sum and the number of tracks in the current CD, as evident from

Figure 16.11. In this case, the accumulator is an `IntBinaryOperator` whose functional type is `(int, int) -> int`. Note that the parameters of the lambda expression represent the partial sum and the current number of tracks, respectively.

The stream pipeline in Figure 16.11 is an example of a *map-reduce transformation* on a sequential stream, as it maps the stream elements first and then reduces them. Typically, a filter operation is also performed before the map-reduce transformation.

Each of the following calls can replace the `reduce()` method call in Figure 16.11, as they are all equivalent:

```
reduce(0, (sum, noOfTracks) -> Integer.sum(sum, noOfTracks))
reduce(0, Integer::sum) // Method reference
sum() // Special functional reduction, p. 1001.
```

In Example 16.11, the stream pipeline at (10) prints the actions taken by the accumulator which is now augmented with print statements. The output at (3) shows that the accumulator actions correspond to those in Figure 16.11.

The single-argument `reduce()` method only accepts an accumulator. As no explicit default or initial value can be specified, this method returns an `Optional`. If the stream is not empty, it uses the first element as the initial value; otherwise, it returns an empty `Optional`. In Example 16.11, the stream pipeline at (13) uses the single-argument `reduce()` method to compute the total number of tracks on CDs. The return value is an `OptionalInt` that can be queried to extract the encapsulated `int` value.

```
OptionalInt optSumTracks0 = CD.cdList // (13)
    .stream()
    .mapToInt(CD::noOfTracks)
    .reduce(Integer::sum); // (14)
out.println("Total number of tracks: " + optSumTracks0.orElse(0)); // 42
```

We can again augment the accumulator with print statements as shown at (16) in Example 16.11. The output at (5) shows that the number of tracks from the first CD was used as the initial value before the accumulator is applied repeatedly to the rest of the values.

---

#### Example 16.11 *Implementing Functional Reductions*

```
import static java.lang.System.out;

import java.util.Comparator;
import java.util.Optional;
import java.util.OptionalInt;
import java.util.function.BinaryOperator;

public final class FunctionalReductions {
    public static void main(String[] args) {

// Two-argument reduce() method:
    {
```

```

    out.println("(1) Find total number of tracks (loop-based version):");
    int sum = 0; // (1) Initialize the partial result.
    for (CD cd : CD.cdList) { // (2) Iterate over the list.
        int numOfTracks = cd.noOfTracks(); // (3) Get the next value.
        sum = sum + numOfTracks; // (4) Calculate new partial result.
    }
    out.println("Total number of tracks: " + sum);
}

out.println("(2) Find total number of tracks (stream-based version):");
int totNumOfTracks = CD.cdList // (5)
    .stream() // (6)
    .mapToInt(CD::noOfTracks) // (7)
    .reduce(0, // (8)
        (sum, numOfTracks) -> sum + numOfTracks); // (9)
// .reduce(0, (sum, noOfTracks) -> Integer.sum(sum, noOfTracks));
// .reduce(0, Integer::sum);
// .sum();
out.println("Total number of tracks: " + totNumOfTracks);
out.println();

out.println("(3) Find total number of tracks (accumulator logging): ");
int totNumOfTracks1 = CD.cdList // (10)
    .stream()
    .mapToInt(CD::noOfTracks)
    .reduce(0, // (11)
        (sum, noOfTracks) -> { // (12)
            int newSum = sum + noOfTracks;
            out.printf("Accumulator: sum=%2d, noOfTracks=%2d, newSum=%2d%n",
                sum, noOfTracks, newSum);
            return newSum;
        }
    );
out.println("Total number of tracks: " + totNumOfTracks1);
out.println();

// One-argument reduce() method:

out.println("(4) Find total number of tracks (stream-based version):");
OptionalInt optSumTracks0 = CD.cdList // (13)
    .stream()
    .mapToInt(CD::noOfTracks)
    .reduce(Integer::sum); // (14)
out.println("Total number of tracks: " + optSumTracks0.orElse(0));
out.println();

out.println("(5) Find total number of tracks (accumulator logging): ");
OptionalInt optSumTracks1 = CD.cdList // (15)
    .stream()
    .mapToInt(CD::noOfTracks)
    .reduce((sum, noOfTracks) -> { // (16)
        int newSum = sum + noOfTracks;
        out.printf("Accumulator: sum=%2d, noOfTracks=%2d, newSum=%2d%n",
            sum, noOfTracks, newSum);
        return newSum;
    });
}

```

```

    out.println("Total number of tracks: " + optSumTracks1.orElse(0));
    out.println();

// Three-argument reduce() method:

    out.println("(6) Find total number of tracks (accumulator + combiner): ");
    Integer sumTracks5 = CD.cdList // (17)
// .stream() // (18a)
    .parallelStream() // (18b)
    .reduce(Integer.valueOf(0), // (19) Initial value
            (sum, cd) -> sum + cd.noOfTracks(), // (20) Accumulator
            (sum1, sum2) -> sum1 + sum2); // (21) Combiner
    out.println("Total number of tracks: " + sumTracks5);
    out.println();

    out.println("(7) Find total number of tracks (accumulator + combiner): ");
    Integer sumTracks6 = CD.cdList // (22)
// .stream() // (23a)
    .parallelStream() // (23b)
    .reduce(0,
            (sum, cd) -> { // (24) Accumulator
                Integer noOfTracks = cd.noOfTracks();
                Integer newSum = sum + noOfTracks;
                out.printf("Accumulator: sum=%2d, noOfTracks=%2d, "
                        + "newSum=%2d%n", sum, noOfTracks, newSum);
                return newSum;
            },
            (sum1, sum2) -> { // (25) Combiner
                Integer newSum = sum1 + sum2;
                out.printf("Combiner: sum1=%2d, sum2=%2d, newSum=%2d%n",
                        sum1, sum2, newSum);
                return newSum;
            }
    );
    out.println("Total number of tracks: " + sumTracks6);
    out.println();

// Compare by CD title.
    Comparator<CD> cmpByTitle = Comparator.comparing(CD::title); // (26)
    BinaryOperator<CD> maxByTitle =
        (cd1, cd2) -> cmpByTitle.compare(cd1, cd2) > 0 ? cd1 : cd2; // (27)

// Query: Find maximum Jazz CD by title:
    Optional<CD> optMaxJazzCD = CD.cdList // (28)
        .stream()
        .filter(CD::isJazz)
        .reduce(BinaryOperator.maxBy(cmpByTitle)); // (29a)
// .reduce(maxByTitle); // (29b)
// .max(cmpByTitle); // (29c)
    optMaxJazzCD.map(CD::title).ifPresent(out::println);// Keep on Erasing
}
}

```

Possible output from the program:

```
(1) Find total number of tracks (loop-based version):
Total number of tracks: 42
(2) Find total number of tracks (stream-based version):
Total number of tracks: 42

(3) Find total number of tracks (accumulator logging):
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Accumulator: sum= 8, noOfTracks= 6, newSum=14
Accumulator: sum=14, noOfTracks=10, newSum=24
Accumulator: sum=24, noOfTracks= 8, newSum=32
Accumulator: sum=32, noOfTracks=10, newSum=42
Total number of tracks: 42

(4) Find total number of tracks (stream-based version):
Total number of tracks: 42

(5) Find total number of tracks (accumulator logging):
Accumulator: sum= 8, noOfTracks= 6, newSum=14
Accumulator: sum=14, noOfTracks=10, newSum=24
Accumulator: sum=24, noOfTracks= 8, newSum=32
Accumulator: sum=32, noOfTracks=10, newSum=42
Total number of tracks: 42

(6) Find total number of tracks (accumulator + combiner):
Total number of tracks: 42

(7) Find total number of tracks (accumulator + combiner):
Accumulator: sum= 0, noOfTracks=10, newSum=10
Accumulator: sum= 0, noOfTracks=10, newSum=10
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Combiner: sum1= 8, sum2=10, newSum=18
Combiner: sum1=10, sum2=18, newSum=28
Accumulator: sum= 0, noOfTracks= 6, newSum= 6
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Combiner: sum1= 8, sum2= 6, newSum=14
Combiner: sum1=14, sum2=28, newSum=42
Total number of tracks: 42
```

Keep on Erasing

.....

The single-argument and two-argument `reduce()` methods accept a *binary operator* as the accumulator whose arguments and result are of the *same* type. The three-argument `reduce()` method is more flexible and can only be applied to objects. The stream pipeline below computes the total number of tracks on CDs using the three-argument `reduce()` method.

```
Integer sumTracks5 = CD.cdList                                // (17)
    .stream()                                                // (18a)
// .parallelStream()                                       // (18b)
    .reduce(Integer.valueOf(0),                               // (19) Initial value
              (sum, cd) -> sum + cd.noOfTracks(),           // (20) Accumulator
              (sum1, sum2) -> sum1 + sum2);                 // (21) Combiner
```

The `reduce()` method above accepts the following arguments:

- An *identity* value: Its type is `U`. In this case, it is an `Integer` that wraps the value `0`. As before, it is used as the initial value. The type of the value returned by the `reduce()` method is also `U`.
- An *accumulator*: It is a `BiFunction<U,T,U>`; that is, it is a binary function that accepts an object of type `U` and an object of type `T` and produces a result of type `U`. In this case, type `U` is `Integer` and type `T` is `CD`. The lambda expression implementing the accumulator first reads the number of tracks from the current `CD` before the addition operator is applied. Thus the accumulator will calculate the sum of `Integers` which are, of course, unboxed and boxed to do the calculation. As we have seen earlier, the accumulator is repeatedly applied to sum the tracks on the `CDs`. Only this time, the mapping of a `CD` to an `Integer` is done when the accumulator is evaluated.
- A *combiner*: It is a `BinaryOperator<U>`; that is, it is a binary operator whose arguments and result are of the same type `U`. In this case, type `U` is `Integer`. Thus the combiner will calculate the sum of `Integers` which are unboxed and boxed to do the calculation.

In the code above, the combiner is not executed if the `reduce()` method is applied to a *sequential* stream. However, there is no guarantee that this is always the case for a sequential stream. If we uncomment (18b) and remove (18a), the combiner will be executed on the *parallel* stream.

That the combiner in the three-argument `reduce()` method is executed for a parallel stream is illustrated by the stream pipeline at (22) in Example 16.11, that has been augmented with print statements. There is no output from the combiner when the stream is sequential. The output at (7) in Example 16.11 shows that the combiner accumulates the partial sums created by the accumulator when the stream is parallel.

### *Parallel Functional Reduction*

Parallel execution is illustrated in Figure 16.12. Multiple instances of the stream pipeline are executed in parallel, where each pipeline instance processes a segment of the stream. In this case, only one `CD` is allocated to each pipeline instance. Each pipeline instance thus produces its partial sum, and the combiner is applied in parallel on the partial sums to combine them into a final result. No additional synchronization is required to run the `reduce()` operation in parallel.

Figure 16.12 also illustrates why the initial value must be an identity value. Say we had specified the initial value to be `3`. Then the value `3` would be added multiple times to the sum during parallel execution. We also see why both the accumulator and the combiner are associative, as this allows for any two values to be combined in any order.

When the single-argument and two-argument `reduce()` methods are applied to a parallel stream, the accumulator also acts as the combiner. The three-argument `reduce()` method can usually be replaced with a *map-reduce* transformation, making the combiner redundant.

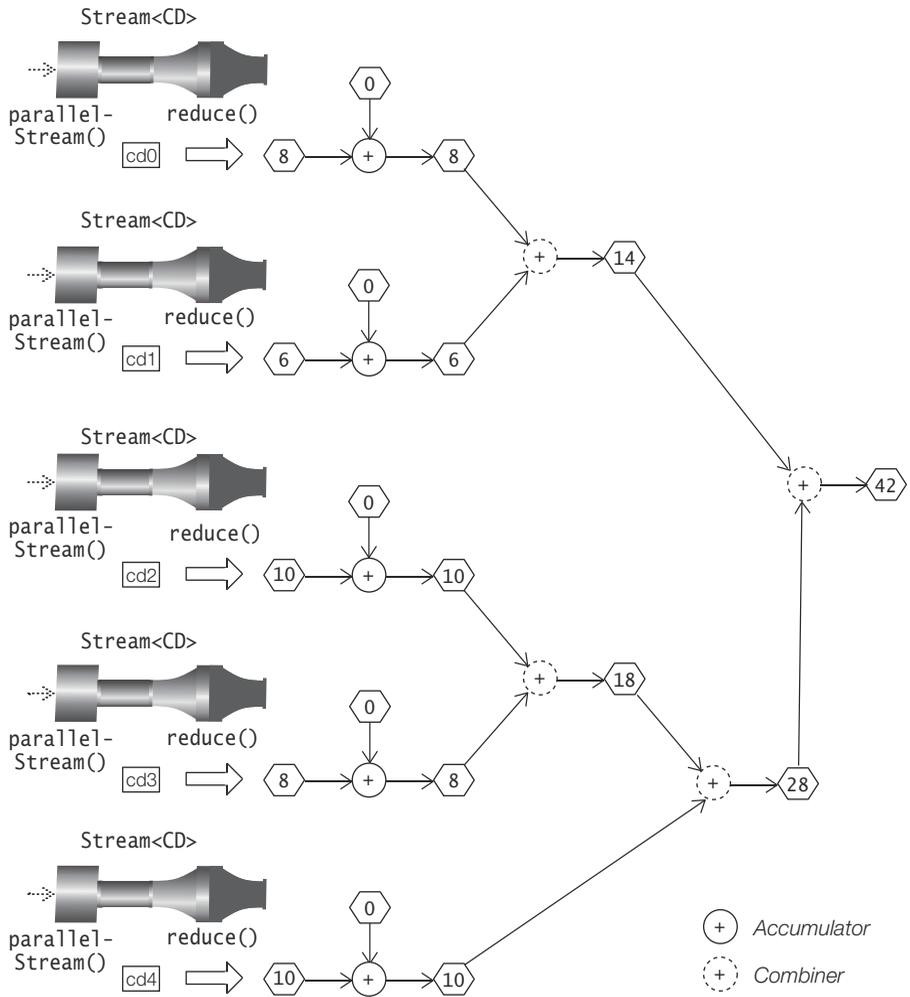
Figure 16.12 *Parallel Functional Reduction*

```

// Query: Find total number of CD tracks.
Integer sumTracks5 = CD.cdList                // (17)
    .parallelStream()                         // (18b)
    .reduce(Integer.valueOf(0),              // (19) Initial value
             (sum, cd) -> sum + cd.noOfTracks(), // (20) Accumulator
             (sum1, sum2) -> sum1 + sum2);    // (21) Combiner

```

(a) Using the `Stream.reduce()` method on a parallel stream



(b) Parallel functional reduction

We conclude the discussion on implementing functional reductions by implementing the `max()` method that finds the maximum element in a stream according to a given comparator. A comparator that compares by the CD title is defined at (26). A binary operator that finds the maximum of two CDs when compared by title is defined at (27). It uses the comparator defined at (26). The stream pipeline at (28) finds the maximum of all jazz music CDs by title. The method calls at (29a), (29b), and (29c) are equivalent.

```

Comparator<CD> cmpByTitle = Comparator.comparing(CD::title);    // (26)
BinaryOperator<CD> maxByTitle =
    (cd1, cd2) -> cmpByTitle.compare(cd1, cd2) > 0 ? cd1 : cd2; // (27)

Optional<CD> optMaxJazzCD = CD.cdList                          // (28)
    .stream()
    .filter(CD::isJazz)
    .reduce(BinaryOperator.maxBy(cmpByTitle));                // (29a)
// .reduce(maxByTitle);                                       // (29b)
// .max(cmpByTitle);                                          // (29c)
optMaxJazzCD.map(CD::title).ifPresent(out::println);         // Keep on Erasing

```

The accumulator at (29a), returned by the `BinaryOperator.maxBy()` method, will compare the previous maximum CD and the current CD by title to compute a new maximum jazz music CD. The accumulator used at (29b) is implemented at (27). It also does the same comparison as the accumulator at (29a). At (29c), the `max()` method also does the same thing, based on the comparator at (26). Note that the return value is an `Optional<CD>`, as the stream might be empty. The `Optional<CD>` is mapped to an `Optional<String>`. If it is not empty, its value—that is, the CD title—is printed.

The `reduce()` method does not terminate if applied to an infinite stream, as the method will never finish processing all stream elements.

## Implementing Mutable Reduction: The `collect()` Method

The `collect(Collector)` method accepts a *collector* that encapsulates the functions required to perform a mutable reduction. We discuss predefined collectors implemented by the `java.util.stream.Collectors` class in a later section (p. 978). The code below uses the collector returned by the `Collectors.toList()` method that accumulates the result in a list (p. 980).

```

List<String> titles = CD.cdList.stream()
    .map(CD::title).collect(Collectors.toList());
// [Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]

```

The `collect(supplier, accumulator, combiner)` generic method provides the general setup for implementing mutable reduction on stream elements using different kinds of *mutable containers*—for example, a list, a map, or a `StringBuilder`. It uses one or more mutable containers to accumulate partial results that are combined into a single mutable container that is returned as the result of the reduction operation.

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

This terminal operation performs a reduction operation on the elements of this stream using a `Collector` (p. 978).

A `Collector` encapsulates the functions required for performing the reduction.

The result of the reduction is of type `R`, and the type parameter `A` is the intermediate accumulation type of the `Collector`.

```
<R> R collect(
    Supplier<R>          supplier,
    BiConsumer<R,? super T> accumulator,
    BiConsumer<R,R>     combiner)
```

This terminal operation performs a mutable reduction on the elements of this stream. A counterpart to this method is also provided for numeric streams.

The `supplier` creates a new mutable container of type `R`—which is typically empty. Elements are incorporated into such a container during the reduction process. For a parallel stream, the `supplier` can be called multiple times, and the container returned by the `supplier` must be an *identity container* in the sense that it does not mutate any result container with which it is merged.

The `accumulator` incorporates additional elements into a result container: A stream element of type `T` is incorporated into a mutable container of type `R`.

The `combiner` merges two values that are mutable containers of type `R`. It must be compatible with the `accumulator`. There is no guarantee that the `combiner` is called if the stream is sequential, but definitely comes into play if the stream is parallel.

Both the `accumulator` and the `combiner` must also be *non-interfering* and *stateless* functions (p. 909).

With the above requirements on the argument functions fulfilled, the `collect()` method will produce the same result regardless of whether the stream is sequential or parallel.

We will use Figure 16.13 to illustrate mutable reduction performed on a sequential stream by the three-argument `collect()` method. The figure shows both the code and the execution of a stream pipeline to create a list containing the number of tracks on each CD. The stream of CDs is mapped to a stream of `Integers` at (3), where each `Integer` value is the number of tracks on a CD. The `collect()` method at (4) accepts three functions as arguments. They are explicitly defined as lambda expressions to show what the parameters represent and how they are used to perform mutable reduction. Implementation of these functions using method references can be found in Example 16.12.

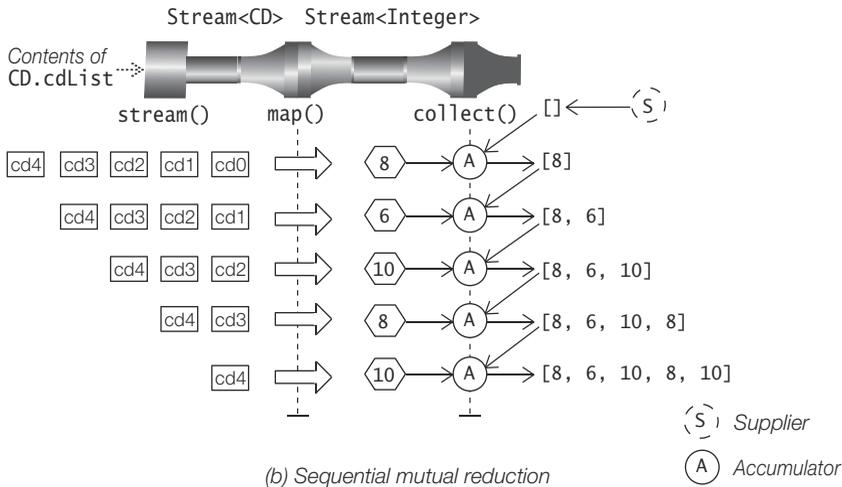
- *Supplier*: The `supplier` is a `Supplier<R>` that is used to create new instances of a mutable result container of type `R`. Such a container holds the results computed by the `accumulator` and the `combiner`. In Figure 16.13, the `supplier` at (4) returns an *empty* `ArrayList<Integer>` every time it is called.

- *Accumulator*: The accumulator is a `BiConsumer<R, T>` that is used to accumulate an element of type `T` into a mutable result container of type `R`. In Figure 16.13, type `R` is `ArrayList<Integer>` and type `T` is `Integer`. The accumulator at (5) mutates a container of type `ArrayList<Integer>` by repeatedly adding a new `Integer` value to it, as illustrated in Figure 16.13b. It is instructive to contrast this accumulator with the accumulator for sequential functional reduction illustrated in Figure 16.11, p. 957.
- *Combiner*: The combiner is a `BiConsumer<R, R>` that merges the contents of the second argument container with the contents of the first argument container, where both containers are of type `R`. As in the case of the `reduce(identity, accumulator, combiner)` method, the combiner is executed when the `collect()` method is called on a parallel stream.

Figure 16.13 Sequential Mutable Reduction

```
// Query: Create a list with the number of tracks on each CD.
List<Integer> tracks = CD.cdList // (1)
    .stream() // (2a)
    .map(CD::noOfTracks) // (3)
    .collect(() -> new ArrayList<>(), // (4) Supplier
            (cont, noOfTracks) -> cont.add(noOfTracks), // (5) Accumulator
            (cont1, cont2) -> cont1.addAll(cont2)); // (6) Combiner
```

(a) Using the `Stream.collect()` method on a sequential stream



### Parallel Mutable Reduction

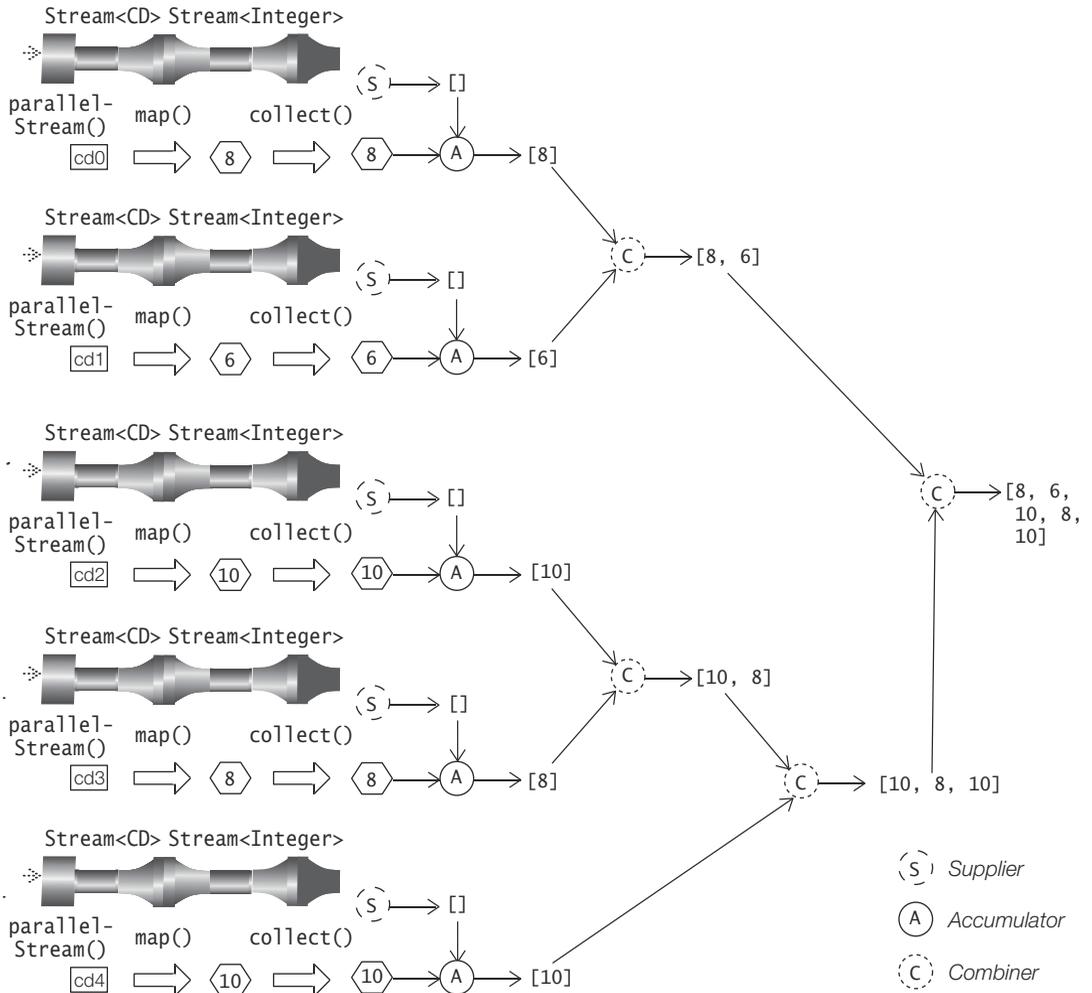
Figure 16.14 shows the stream pipeline from Figure 16.13, where the sequential stream (2a) has been replaced by a parallel stream (2b); in other words, the `collect()` method is called on a parallel stream. One possible parallel execution of the pipeline is also depicted in Figure 16.14b. We see five instances of the pipeline being executed in parallel. The supplier creates five empty `ArrayList`s that are used

**Figure 16.14** *Parallel Mutable Reduction*

```

// Query: Create a list with the number of tracks on each CD.
List<Integer> tracks = CD.cdList // (1)
    .parallelStream() // (2b)
    .map(CD::noOfTracks) // (3)
    .collect() -> new ArrayList<>(), // (4) Supplier
        (cont, noOfTracks) -> cont.add(noOfTracks), // (5) Accumulator
        (cont1, cont2) -> cont1.addAll(cont2)); // (6) Combiner
    
```

(a) Using the `Stream.collect()` method on a parallel stream



(b) Parallel mutable reduction

as partial result containers by the accumulator, and are later merged by the combiner to a final result container. The containers created by the supplier are mutated by the accumulator and the combiner to perform mutable reduction. The partial result containers are also merged in parallel by the combiner. It is instructive to contrast this combiner with the combiner for parallel functional reduction that is illustrated in Figure 16.12, p. 963.

In Example 16.12, the stream pipeline at (7) also creates a list containing the number of tracks on each CD, where the stream is parallel, and the lambda expressions implementing the argument functions of the `collect()` method are augmented with print statements so that actions of the functions can be logged. The output from this parallel mutable reduction shows that the combiner is executed multiple times to merge partial result lists. The actions of the argument functions shown in the output are the same as those illustrated in Figure 16.14b. Of course, multiple runs of the pipeline can show different sequences of operations in the output, but the final result is the same. Also note that the elements retain their relative position in the partial result lists as these are combined, preserving the encounter order of the stream.

Although a stream is executed in parallel to perform mutable reduction, the merging of the partial containers by the combiner can impact performance if this is too costly. For example, merging mutable maps can be costly compared to merging mutable lists. This issue is further explored for parallel streams in §16.9, p. 1009.

---

**Example 16.12** *Implementing Mutable Reductions*

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.Stream;

public final class Collecting {
    public static void main(String[] args) {

        // Query: Create a list with the number of tracks on each CD.
        System.out.println("Sequential Mutable Reduction:");
        List<Integer> tracks = CD.cdList // (1)
            .stream() // (2a)
            // .parallelStream() // (2b)
            .map(CD::noOfTracks) // (3)
            .collect(() -> new ArrayList<>(), // (4) Supplier
                (cont, noOfTracks) -> cont.add(noOfTracks), // (5) Accumulator
                (cont1, cont2) -> cont1.addAll(cont2)); // (6) Combiner
            // .collect(ArrayList::new, ArrayList::add, ArrayList::addAll); // (6a)
            // .toList();
        System.out.println("Number of tracks on each CD (sequential): " + tracks);
        System.out.println();

        System.out.println("Parallel Mutable Reduction:");
        List<Integer> tracks1 = CD.cdList // (7)
```

```

//      .stream()                                // (8a)
      .parallelStream()                          // (8b)
      .map(CD::noOfTracks)                       // (9)
      .collect(                                  // (10)
        () -> {                                  // (11) Supplier
            System.out.println("Supplier: Creating an ArrayList");
            return new ArrayList<>();
        },
        (cont, noOfTracks) -> {                  // (12) Accumulator
            System.out.printf("Accumulator: cont:%s, noOfTracks:%s",
                               cont, noOfTracks);
            cont.add(noOfTracks);
            System.out.printf(", mutCont:%s%n", cont);
        },
        (cont1, cont2) -> {                     // (13) Combiner
            System.out.printf("Combiner: con1:%s, cont2:%s", cont1, cont2);
            cont1.addAll(cont2);
            System.out.printf(", mutCont:%s%n", cont1);
        });
    System.out.println("Number of tracks on each CD (parallel): " + tracks1);
    System.out.println();

    // Query: Create an ordered set with CD titles, according to natural order.
    Set<String> cdTitles = CD.cdList              // (14)
        .stream()
        .map(CD::title)
        .collect(TreeSet::new, TreeSet::add, TreeSet::addAll); // (15)
    System.out.println("CD titles: " + cdTitles);
    System.out.println();

    // Query: Go bananas.
    StringBuilder goneBananas = Stream          // (16)
        .iterate("ba", b -> b + "na")         // (17)
        .limit(5)
        .peek(System.out::println)
        .collect(StringBuilder::new,          // (18)
            StringBuilder::append,
            StringBuilder::append);
    System.out.println("Go bananas: " + goneBananas);
}
}

```

Possible output from the program:

```

Sequential Mutable Reduction:
Number of tracks on each CD (sequential): [8, 6, 10, 8, 10]

```

```

Parallel Mutable Reduction:
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:8, mutCont:[8]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:6, mutCont:[6]
Combiner: con1:[8], cont2:[6], mutCont:[8, 6]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:10, mutCont:[10]
Supplier: Creating an ArrayList

```

```

Accumulator: cont:[], noOfTracks:8, mutCont:[8]
Combiner: con1:[10], cont2:[8], mutCont:[10, 8]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:10, mutCont:[10]
Combiner: con1:[10, 8], cont2:[10], mutCont:[10, 8, 10]
Combiner: con1:[8, 6], cont2:[10, 8, 10], mutCont:[8, 6, 10, 8, 10]
Number of tracks on each CD (parallel): [8, 6, 10, 8, 10]

CD titles: [Hot Generics, Java Jam, Java Jive, Keep on Erasing, Lambda Dancing]

ba
bana
banana
bananana
banananana
Go bananas: babanabanabanananabanananana

```

Example 16.12 also shows how other kinds of containers can be used for mutable reduction. The stream pipeline at (14) performs mutable reduction to create an ordered set with CD titles. The supplier is implemented by the constructor reference `TreeSet::new`. The constructor will create a container of type `TreeSet<String>` that will maintain the CD titles according to the natural order for `Strings`. The accumulator and the combiner are implemented by the method references `TreeSet::add` and `TreeSet::addAll`, respectively. The accumulator will add a title to a container of type `TreeSet<String>` and the combiner will merge the contents of two containers of type `TreeSet<String>`.

In Example 16.12, the mutable reduction performed by the stream pipeline at (16) uses a mutable container of type `StringBuilder`. The output from the `peek()` method shows that the strings produced by the `iterate()` method start with the initial string "ba" and are iteratively concatenated with the postfix "na". The `limit()` intermediate operation truncates the infinite stream to five elements. The `collect()` method appends the strings to a `StringBuilder`. The supplier creates an empty `StringBuilder`. The accumulator and the combiner append a `CharSequence` to a `StringBuilder`. In the case of the accumulator, the `CharSequence` is a `String`—that is, a stream element—in the call to the `append()` method. But in the case of the combiner, the `CharSequence` is a `StringBuilder`—that is, a partial result container when the stream is parallel. One might be tempted to use a string instead of a `StringBuilder`, but that would not be a good idea as a string is immutable.

Note that the accumulator and combiner of the `collect()` method do not return a value. The `collect()` method does not terminate if applied to an infinite stream, as the method will never finish processing all the elements in the stream.

Because mutable reduction uses the same mutable result container for accumulating new results by changing the state of the container, it is more efficient than a functional reduction where a new partial result always replaces the previous partial result.

## Collecting to an Array

The overloaded method `toArray()` can be used to collect or accumulate into an array. It is a special case of a mutable reduction, and as the name suggests, the mutable container is an array. The numeric stream interfaces also provide a counterpart to the `toArray()` method that returns an array of a numeric type.

```
Object[] toArray()
```

This terminal operation returns an array containing the elements of this stream. Note that the array returned is of type `Object[]`.

```
<A> A[] toArray(IntFunction<A[]> generator)
```

This terminal operation returns an array containing the elements of this stream. The provided generator function is used to allocate the desired array. The type parameter `A` is the element type of the array that is returned. The size of the array (which is equal to the length of the stream) is passed to the generator function as an argument.

The zero-argument method `toArray()` returns an array of objects, `Object[]`, as generic arrays cannot be created at runtime. The method needs to store all the elements before creating an array of the appropriate length. The query at (1) finds the titles of the CDs, and the `toArray()` method collects them into an array of objects, `Object[]`.

```
Object[] objArray = CD.cdList.stream().map(CD::title)
                                     .toArray();           // (1)
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

The `toArray(IntFunction<A>)` method accepts a generator function that creates an array of type `A`, (`A[]`), whose length is passed as a parameter by the method to the generator function. The array length is determined from the number of elements in the stream. The query at (2) also finds the CD titles, but the `toArray()` method collects them into an array of strings, `String[]`. The method reference defining the generator function is equivalent to the lambda expression `(len -> new String[len])`.

```
String[] cdTitles = CD.cdList.stream().map(CD::title)
                                     .toArray(String[]::new); // (2)
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

Examples of numeric streams whose elements are collected into an array are shown at (3) and (4). The `limit()` intermediate operation at (3) converts the infinite stream into a finite one whose elements are collected into an `int` array.

```
int[] intArray1 = IntStream.iterate(1, i -> i + 1).limit(5).toArray(); // (3)
// [1, 2, 3, 4, 5]
int[] intArray2 = IntStream.range(-5, 5).toArray();           // (4)
// [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Not surprisingly, when applied to infinite streams the operation results in a fatal `OutOfMemoryError`, as the method cannot determine the length of the array and keeps storing the stream elements, eventually running out of memory.

```
int[] intArray3 = IntStream.iterate(1, i -> i + 1)           // (5)
                        .toArray();                       // OutOfMemoryError!
```

If the sole purpose of using the `toArray()` operation in a pipeline is to convert the data source collection to an array, it is far better to use the overloaded `Collection.toArray()` methods. For one thing, the size of the array is easily determined from the size of the collection.

```
CD[] cdArray1 = CD.cdList.stream().toArray(CD[]::new);      // (6) Preferred.
CD[] cdArray2 = CD.cdList.toArray(new CD[CD.cdList.size()]); // (7) Not efficient.
```

Like any other mutable reduction operation, the `toArray()` method does not terminate when applied to an infinite stream, unless it is converted into a finite stream as at (3) above.

## Collecting to a List

The method `Stream.toList()` implements a terminal operation that can be used to collect or accumulate the result of processing a stream into a list. Compared to the `toArray()` instance method, the `toList()` method is a default method in the `Stream` interface. The default implementation returns an *unmodifiable* list; that is, elements cannot be added, removed, or sorted. This unmodifiable list is created from the array into which the elements are accumulated first.

If the requirement is an unmodifiable list that allows `null` elements, the `Stream.toList()` is the clear and concise choice. Many examples of stream pipelines encountered so far in this chapter use the `toList()` terminal operation.

```
List<String> titles = CD.cdList.stream().map(CD::title).toList();
// [Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
titles.add("Java Jingles"); // UnsupportedOperationException!
```

Like any other mutable reduction operation, the `toList()` method does not terminate when applied to an infinite stream, unless the stream is converted into a finite stream.

```
default List<T> toList()
```

Accumulates the elements of this stream into a `List`, respecting any encounter order the stream may have. The returned `List` is *unmodifiable* (§12.2, p. 649), and calls to any mutator method will always result in an `UnsupportedOperationException`. The unmodifiable list returned allows `null` values.

See also the `toList()` method in the `Collectors` class (p. 980).

The `Collectors.toCollection(Supplier)` method is recommended for greater control.

## Functional Reductions Exclusive to Numeric Streams

In addition to the counterparts of the methods in the `Stream<T>` interface, the following functional reductions are exclusive to the numeric stream interfaces

`IntStream`, `LongStream`, and `DoubleStream`. These reduction operations are designed to calculate various statistics on numeric streams.

In the methods below, *NumType* is `Int`, `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`. These statistical operations do not terminate when applied to an infinite stream:

```
numtype sum()
```

This terminal operation returns the sum of elements in this stream. It returns zero if the stream is empty.

```
OptionalDouble average()
```

This terminal operation returns an `OptionalDouble` that encapsulates the arithmetic mean of elements of this stream, or an empty `Optional` if this stream is empty.

```
NumTypeSummaryStatistics summaryStatistics()
```

This terminal operation returns a `NumTypeSummaryStatistics` describing various summary data about the elements of this stream.

### Summation

The `sum()` terminal operation is a special case of a functional reduction that calculates the sum of numeric values in a stream. The stream pipeline below calculates the total number of tracks on the CDs in a list. Note that the stream of `CD` is mapped to an `int` stream whose elements represent the number of tracks on a `CD`. The `int` values are cumulatively added to compute the total number of tracks.

```
int totNumOfTracks = CD.cdList
    .stream()                // Stream<CD>
    .mapToInt(CD::noOfTracks) // IntStream
    .sum();                  // 42
```

The query below sums all even numbers between 1 and 100.

```
int sumEven = IntStream
    .rangeClosed(1, 100)
    .filter(i -> i % 2 == 0)
    .sum();                // 2550
```

The `count()` operation is equivalent to mapping each stream element to the value 1 and adding the 1s:

```
int numOfCDs = CD.cdList
    .stream()
    .mapToInt(cd -> 1)    // CD => 1
    .sum();              // 5
```

For an empty stream, the sum is always zero.

```
double total = DoubleStream.empty().sum(); // 0.0
```

## Averaging

Another common statistics to calculate is the average of values, defined as the sum of values divided by the number of values. A loop-based solution to calculate the average would explicitly sum the values, count the number of values, and do the calculation. In a stream-based solution, the `average()` terminal operation can be used to calculate this value. The stream pipeline below computes the average number of tracks on a CD. The CD stream is mapped to an `int` stream whose values are the number of tracks on a CD. The `average()` terminal operation adds the number of tracks and counts the values, returning the average as a `double` value encapsulated in an `OptionalDouble`.

```
OptionalDouble optAverage = CD.cdList
    .stream()
    .mapToInt(CD::noOfTracks)
    .average();
System.out.println(optAverage.orElse(0.0));    // 8.4
```

The reason for using an `Optional` is that the average is not defined if there are no values. The absence of a value in the `OptionalDouble` returned by the method means that the stream was empty.

## Summarizing

The result of a functional reduction is a single value. This means that for calculating different results—for example, count, sum, average, min, and max—requires separate reduction operations on a stream.

The method `summaryStatistics()` does several common reductions on a stream in a single operation and returns the results in an object of type `NumTypeSummaryStatistics`, where `NumType` is `Int`, `Long`, or `Double`. An object of this class encapsulates the count, sum, average, min, and max values of a stream.

The classes `IntSummaryStatistics`, `LongSummaryStatistics`, and `DoubleSummaryStatistics` in the `java.util` package define the following constructor and methods, where `NumType` is `Int` (but it is `Integer` when used as a type name), `Long`, or `Double`, and the corresponding `numtype` is `int`, `long`, or `double`:

```
NumTypeSummaryStatistics()
```

Creates an empty instance with zero count, zero sum, a min value as `NumType.MAX_VALUE`, a max value as `NumType.MIN_VALUE`, and an average value of zero.

```
double getAverage()
```

Returns the arithmetic mean of values recorded, or zero if no values have been recorded.

```
long getCount()
```

Returns the count of values recorded.

```
numtype getMax()
```

Returns the maximum value recorded, or *NumType*.MIN\_VALUE if no values have been recorded.

```
numtype getMin()
```

Returns the minimum value recorded, or *NumType*.MAX\_VALUE if no values have been recorded.

```
numtype getSum()
```

Returns the sum of values recorded, or zero if no values have been recorded. The method in the *IntSummaryStatistics* and *LongSummaryStatistics* classes returns a long value. The method in the *DoubleSummaryStatistics* class returns a double value.

```
void accept(numtype value)
```

Records a new value into the summary information, and updates the various statistics. The method in the *LongSummaryStatistics* class is overloaded and can accept an int value as well.

```
void combine(NumTypeSummaryStatistics other)
```

Combines the state of another *NumTypeSummaryStatistics* into this one.

The *summaryStatistics()* method is used to calculate various statistics for the number of tracks on two CDs processed by the stream pipeline below. Various *get* methods are called on the *IntSummaryStatistics* object returned by the *summaryStatistics()* method, and the statistics are printed.

```
IntSummaryStatistics stats1 = List.of(CD.cd0, CD.cd1)
    .stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();
System.out.println("Count=" + stats1.getCount());      // Count=2
System.out.println("Sum=" + stats1.getSum());          // Sum=14
System.out.println("Min=" + stats1.getMin());         // Min=6
System.out.println("Max=" + stats1.getMax());         // Max=8
System.out.println("Average=" + stats1.getAverage()); // Average=7.0
```

The default format of the statistics printed by the *toString()* method of the *IntSummaryStatistics* class is shown below:

```
System.out.println(stats1);
//IntSummaryStatistics{count=2, sum=14, min=6, average=7.000000, max=8}
```

Below, the *accept()* method records the value 10 (the number of tracks on CD.cd2) into the summary information referenced by *stats1*. The resulting statistics show the new count is 3 (=2 +1), the new sum is 24 (=14+10), and the new average is 8.0 (=24.0/3.0). However, the min value was not affected but the max value has changed to 10.

```
stats1.accept(CD.cd2.noOfTracks()); // Add the value 10.
System.out.println(stats1);
//IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}
```

The code below creates another `IntSummaryStatistics` object that summarizes the statistics from two other CDs.

```
IntSummaryStatistics stats2 = List.of(CD.cd3, CD.cd4)
    .stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();
System.out.println(stats2);
//IntSummaryStatistics{count=2, sum=18, min=8, average=9.000000, max=10}
```

The `combine()` method incorporates the state of one `IntSummaryStatistics` object into another `IntSummaryStatistics` object. In the code below, the state of the `IntSummaryStatistics` object referenced by `stats2` is combined with the state of the `IntSummaryStatistics` object referenced by `stats1`. The resulting summary information is printed, showing that the new count is 5 ( $=3+2$ ), the new sum is 42 ( $=24+18$ ), and the new average is 8.4 ( $=42.0/5.0$ ). However, the min and max values were not affected.

```
stats1.combine(stats2); // Combine stats2 with stats1.
System.out.println(stats1);
//IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

Calling the `summaryStatistics()` method on an empty stream returns an instance of the `IntSummaryStatistics` class with a zero value set for all statistics, except for the min and max values, which are set to `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, respectively. The `IntSummaryStatistics` class provides a zero-argument constructor that also returns an empty instance.

```
IntSummaryStatistics emptyStats = IntStream.empty().summaryStatistics();
System.out.println(emptyStats);
//IntSummaryStatistics{count=0, sum=0, min=2147483647, average=0.000000,
//max=-2147483648}
```

The summary statistics classes are not exclusive for use with streams, as they provide a constructor and appropriate methods to incorporate numeric values in order to calculate common statistics, as we have seen here. We will return to calculating statistics when we discuss built-in collectors (p. 978).

## Summary of Terminal Stream Operations

The terminal operations of the `Stream<T>` class are summarized in Table 16.5. The type parameter declarations have been simplified, where any bound `<? super T>` or `<? extends T>` has been replaced by `<T>`, without impacting the intent of a method. A reference is provided to each method in the first column.

The last column in Table 16.5 indicates the function type of the corresponding parameter in the previous column. It is instructive to note how the functional interface parameters provide the parameterized behavior of an operation. For example, the method `allMatch()` returns a boolean value to indicate whether all elements of a stream satisfy a given predicate. This predicate is implemented as a functional interface `Predicate<T>` that is applied to each element in the stream.

The interfaces `IntStream`, `LongStream`, and `DoubleStream` define analogous methods to those shown for the `Stream<T>` interface in Table 16.5. Methods that are only defined by the numeric stream interfaces are shown in Table 16.6.

**Table 16.5** *Terminal Stream Operations*

Method name (ref.)	Any type parameter + return type	Functional interface parameters	Function type of parameters
<code>forEach</code> (p. 948)	<code>void</code>	<code>(Consumer&lt;T&gt; action)</code>	<code>T -&gt; void</code>
<code>forEachOrdered</code> (p. 948)	<code>void</code>	<code>(Consumer&lt;T&gt; action)</code>	<code>T -&gt; void</code>
<code>allMatch</code> (p. 949)	<code>boolean</code>	<code>(Predicate&lt;T&gt; predicate)</code>	<code>T -&gt; boolean</code>
<code>anyMatch</code> (p. 949)	<code>boolean</code>	<code>(Predicate&lt;T&gt; predicate)</code>	<code>T -&gt; boolean</code>
<code>noneMatch</code> (p. 949)	<code>boolean</code>	<code>(Predicate&lt;T&gt; predicate)</code>	<code>T -&gt; boolean</code>
<code>findAny</code> (p. 952)	<code>Optional&lt;T&gt;</code>	<code>()</code>	
<code>findFirst</code> (p. 952)	<code>Optional&lt;T&gt;</code>	<code>()</code>	
<code>count</code> (p. 953)	<code>long</code>	<code>()</code>	
<code>max</code> (p. 954)	<code>Optional&lt;T&gt;</code>	<code>(Comparator&lt;T&gt; cmp)</code>	<code>(T,T) -&gt; int</code>
<code>min</code> (p. 954)	<code>Optional&lt;T&gt;</code>	<code>(Comparator&lt;T&gt; cmp)</code>	<code>(T,T) -&gt; int</code>
<code>reduce</code> (p. 955)	<code>Optional&lt;T&gt;</code>	<code>(BinaryOperator&lt;T&gt; accumulator)</code>	<code>(T,T) -&gt; T</code>
<code>reduce</code> (p. 955)	<code>T</code>	<code>(T identity, BinaryOperator&lt;T&gt; accumulator)</code>	<code>T -&gt; T,</code> <code>(T,T) -&gt; T</code>
<code>reduce</code> (p. 955)	<code>&lt;U&gt; U</code>	<code>(U identity, BiFunction&lt;U,T,U&gt; accumulator, BinaryOperator&lt;U&gt; combiner)</code>	<code>U -&gt; U,</code> <code>(U,T) -&gt; U,</code> <code>(U,U) -&gt; U</code>
<code>collect</code> (p. 964)	<code>&lt;R,A&gt; R</code>	<code>(Collector&lt;T,A,R&gt; collector)</code>	Parameter is not a functional interface.
<code>collect</code> (p. 964)	<code>&lt;R&gt; R</code>	<code>(Supplier&lt;R&gt; supplier, BiConsumer&lt;R,T&gt; accumulator, BiConsumer&lt;R,R&gt; combiner)</code>	<code>() -&gt; R,</code> <code>(R,T) -&gt; void,</code> <code>(R,R) -&gt; void</code>
<code>toArray</code> (p. 971)	<code>Object[]</code>	<code>()</code>	
<code>toArray</code> (p. 971)	<code>&lt;A&gt; A[]</code>	<code>(IntFunction&lt;A[]&gt; generator)</code>	<code>int -&gt; A[]</code>
<code>toList</code> (p. 972)	<code>List&lt;T&gt;</code>	<code>()</code>	

Table 16.6 *Additional Terminal Operations in the Numeric Stream Interfaces*

Method name (ref.)	Return type
average (p. 949)	OptionalNumType, where NumType is Int, Long, or Double
sum (p. 949)	numtype, where numtype is int, long, or double
summaryStatistics (p. 974)	NumTypeSummaryStatistics, where NumType is Int, Long, or Double

## 16.8 Collectors

A *collector* encapsulates the functions required for performing reduction: the supplier, the accumulator, the combiner, and the finisher. It can provide these functions since it implements the Collector interface (in the `java.util.stream` package) that defines the methods to create these functions. It is passed as an argument to the `collect(Collector)` method in order to perform a reduction operation. In contrast, the `collect(Supplier, BiConsumer, BiConsumer)` method requires the functions supplier, accumulator, and combiner, respectively, to be passed as arguments in the method call.

Details of implementing a collector are not necessary for our purposes, as we will exclusively use the extensive set of predefined collectors provided by the static factory methods of the Collectors class in the `java.util.stream` package (Table 16.7, p. 1005). In most cases, it should be possible to find a predefined collector for the task at hand. The collectors use various kinds of containers for performing reduction—for example, accumulating to a map, or finding the minimum or maximum element. For example, the `Collectors.toList()` factory method creates a collector that performs mutable reduction using a list as a mutable container. It can be passed to the `collect(Collector)` terminal operation of a stream.

It is a common practice to import the static factory methods of the Collectors class in the code so that the methods can be called by their simple names.

```
import static java.util.stream.Collectors.*;
```

However, the practice adopted in this chapter is to assume that only the Collectors class is imported, enforcing the connection between the static methods and the class to be done explicitly in the code. Of course, static import of factory methods can be used once familiarity with the collectors is established.

```
import java.util.stream.Collectors;
```

The three-argument `collect()` method is primarily used to implement mutable reduction, whereas the Collectors class provides collectors for both functional and mutable reduction that can be either used in a stand-alone capacity or composed with other collectors.

One group of collectors is designed to collect to a *predetermined container*, which is evident from the name of the static factory method that creates it: `toCollection`, `toList`, `toSet`, and `toMap` (p. 979). The overloaded `toCollection()` and `toMap()` methods allow a specific implementation of a collection and a map to be used, respectively—for example, a `TreeSet` for a collection and a `TreeMap` for a map. In addition, there is the `joining()` method that creates a collector for concatenating the input elements to a `String`—however, internally it uses a mutable `StringBuilder` (p. 984).

Collectors can be composed with other collectors; that is, the partial results from one collector can be additionally processed by another collector (called the *downstream collector*) to produce the final result. Many collectors that can be used as a downstream collector perform functional reduction such as counting values, finding the minimum and maximum values, summing values, averaging values, and summarizing common statistics for values (p. 998).

Composition of collectors is utilized to perform *multilevel grouping* and *partitioning* on stream elements (p. 985). The `groupingBy()` and `partitionBy()` methods return composed collectors to create *classification maps*. In such a map, the keys are determined by a *classifier function*, and the values are the result of a downstream collector, called the *classification mapping*. For example, the CDs in a stream could be classified into a map where the key represents the number of tracks on a CD and the associated value of a key can be a list of CDs with the same number of tracks. The list of CDs with the same number of tracks is the result of an appropriate downstream collector.

## Collecting to a Collection

The method `toCollection(Supplier)` creates a collector that uses a mutable container of a specific `Collection` type to perform mutable reduction. A supplier to create the mutable container is specified as an argument to the method.

The following stream pipeline creates an `ArrayList<String>` instance with the titles of all CDs in the stream. The constructor reference `ArrayList::new` returns an empty `ArrayList<String>` instance, where the element type `String` is inferred from the context.

```
ArrayList<String> cdTitles1 = CD.cdList.stream() // Stream<CD>
    .map(CD::title)                             // Stream<String>
    .collect(Collectors.toCollection(ArrayList::new));
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

```
static <T,C extends Collection<T>> Collector<T,?,C>
    toCollection(Supplier<C> collectionFactory)
```

Returns a `Collector` that accumulates the input elements of type `T` into a new `Collection` of type `C`, in encounter order. A new empty `Collection` of type `C` is created by the `collectionFactory` supplier, thus the collection created can be of a specific `Collection` type.

```
static <T> Collector<T,?,List<T>> toList()
static <T> Collector<T,?,List<T>> toUnmodifiableList()
```

Return a Collector that accumulates the input elements of type T into a new List or an unmodifiable List of type T, respectively, in encounter order.

The `toList()` method gives no guarantees of any kind for the returned list.

The unmodifiable list returned does not allow null values.

See also the `Stream.toList()` terminal operation (p. 972).

```
static <T> Collector<T,?,Set<T>> toSet()
static <T> Collector<T,?,Set<T>> toUnmodifiableSet()
```

Return an unordered Collector that accumulates the input elements of type T into a new Set or an unmodifiable Set of type T, respectively.

## Collecting to a List

The method `toList()` creates a collector that uses a mutable container of type `List` to perform mutable reduction. This collector guarantees to preserve the encounter order of the input stream, if it has one. For more control over the type of the list, the `toCollection()` method can be used. This collector can be used as a *downstream collector*.

The following stream pipeline creates a list with the titles of all CDs in the stream using a collector returned by the `Collectors.toList()` method. Although the returned list is modified, this is implementation dependent and should not be relied upon.

```
List<String> cdTitles3 = CD.cdList.stream() // Stream<CD>
    .map(CD::title) // Stream<String>
    .collect(Collectors.toList());
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
titles.add("Java Jingles"); // OK
```

## Collecting to a Set

The method `toSet()` creates a collector that uses a mutable container of type `Set` to perform mutable reduction. The collector does not guarantee to preserve the encounter order of the input stream. For more control over the type of the set, the `toCollection()` method can be used.

The following stream pipeline creates a set with the titles of all CDs in the stream.

```
Set<String> cdTitles2 = CD.cdList.stream() // Stream<CD>
    .map(CD::title) // Stream<String>
    .collect(Collectors.toSet());
//[Hot Generics, Java Jive, Lambda Dancing, Keep on Erasing, Java Jam]
```

## Collecting to a Map

The method `toMap()` creates a collector that performs mutable reduction to a mutable container of type `Map`.

```
static <T,K,U> Collector<T,?,Map<K,U>> toMap(
    Function<? super T,? extends K> keyMapper,
    Function<? super T,? extends U> valueMapper)

static <T,K,U> Collector<T,?,Map<K,U>> toMap(
    Function<? super T,? extends K> keyMapper,
    Function<? super T,? extends U> valueMapper,
    BinaryOperator<U> mergeFunction)

static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(
    Function<? super T,? extends K> keyMapper,
    Function<? super T,? extends U> valueMapper,
    BinaryOperator<U> mergeFunction,
    Supplier<M> mapSupplier)
```

Return a `Collector` that accumulates elements of type `T` into a `Map` whose keys and values are the result of applying the provided key and value mapping functions to the input elements.

The `keyMapper` function produces keys of type `K`, and the `valueMapper` function produces values of type `U`.

In the first method, the mapped keys cannot have duplicates—an `IllegalStateException` will be thrown if that is the case.

In the second and third methods, the `mergeFunction` binary operator is used to resolve collisions between values associated with the same key, as supplied to `Map.merge(Object, Object, BiFunction)`.

In the third method, the provided `mapSupplier` function returns a new `Map` into which the results will be inserted.

The collector returned by the method `toMap()` uses either a default map or one that is supplied. To be able to create an entry in a `Map<K,U>` from stream elements of type `T`, the collector requires two functions:

- `keyMapper: T -> K`, which is a `Function` to extract a key of type `K` from a stream element of type `T`.
- `valueMapper: T -> U`, which is a `Function` to extract a value of type `U` for a given key of type `K` from a stream element of type `T`.

Additional functions as arguments allow various controls to be exercised on the map:

- `mergeFunction: (U,U) -> U`, which is a `BinaryOperator` to merge two values that are associated with the same key. The merge function must be specified if collision of values can occur during the mutable reduction, or a resounding exception will be thrown.

- `mapSupplier: () -> M` extends `Map<K,V>`, which is a `Supplier` that creates a map instance of a specific type to use for mutable reduction. The map created is a subtype of `Map<K,V>`. Without this function, the collector uses a default map.

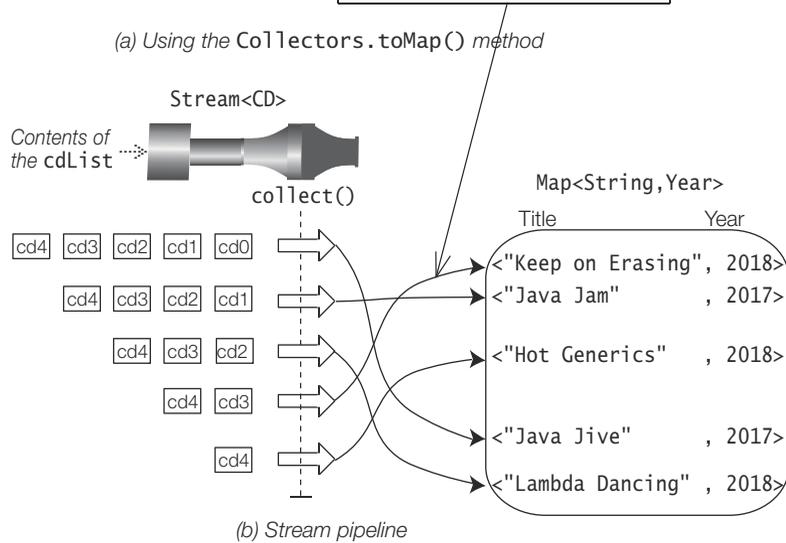
Figure 16.15 illustrates collecting to a map. The stream pipeline creates a map of CD titles and their release year—that is, a `Map<String, Year>`, where `K` is `String` and `V` is `Year`. The `keyMapper` `CD::title` and the `valueMapper` `CD::year` extract the title (`String`) and the year (`Year`) from each `CD` in the stream, respectively. The entries are accumulated in a default map (`Map<String, Year>`).

What if we wanted to create a map with CDs and their release year—that is, a `Map<CD, Year>`? In that case, the `keyMapper` should return the `CD` as the key—that is, map a `CD` to itself. That is exactly what the `keyMapper` `Function.identity()` does in the pipeline below.

```
Map<CD, Year> mapCDToYear = CD.cdList.stream()
    .collect(Collectors.toMap(Function.identity(), CD::year)); // Map<CD, Year>
```

**Figure 16.15** *Collecting to a Map*

```
//Query: Create a map of CD titles and their release year.
Map<String, Year> mapTitleToYear = CD.cdList.stream()
    .collect(Collectors.toMap(CD::title, CD::year));
```



As there were no duplicates of the key in the previous two examples, there was no collision of values in the map. In the list `dupList` below, there are duplicates of CDs (`CD.cd0`, `CD.cd1`). Executing the pipeline results in a runtime exception at (1).

```
List<CD> dupList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd0, CD.cd1);
Map<String, Year> mapTitleToYear1 = dupList.stream()
    .collect(Collectors.toMap(CD::title, CD::year)); // (1)
// IllegalStateException: Duplicate key 2017
```

The collision values can be resolved by specifying a merge function. In the pipeline below, the arguments of the merge function ( $y_1, y_2 \rightarrow y_1$ ) at (1) have the same value for the year if we assume that a CD can only be released once. Note that  $y_1$  and  $y_2$  denote the existing value in the map and the value to merge, respectively. The merge function can return any one of the values to resolve the collision.

```
Map<String, Year> mapTitleToYear2 = dupList.stream()
    .collect(Collectors.toMap(CD::title, CD::year, (y1, y2) -> y1));    // (1)
```

The stream pipeline below creates a map of CD titles released each year. As more than one CD can be released in a year, collision of titles can occur for a year. The merge function ( $(tt, t) \rightarrow tt + ":" + t$ ) concatenates the titles in each year separated by a colon, if necessary. Note that  $tt$  and  $t$  denote the existing value in the map and the value to merge, respectively.

```
Map<Year, String> mapTitleToYear3 = CD.cdList.stream()
    .collect(Collectors.toMap(CD::year, CD::title,
        (tt, t) -> tt + ":" + t));
//{2017=Java Jive:Java Jam, 2018=Lambda Dancing:Keep on Erasing:Hot Generics}
```

The stream pipeline below creates a map with the longest title released each year. For greater control over the type of the map in which to accumulate the entries, a supplier is specified. The supplier `TreeMap::new` returns an empty instance of a `TreeMap` in which the entries are accumulated. The keys in such a map are sorted in their natural order—the class `java.time.Year` implements the `Comparable<Year>` interface.

```
TreeMap<Year, String> mapYearToLongestTitle = CD.cdList.stream()
    .collect(Collectors.toMap(CD::year, CD::title,
        BinaryOperator.maxBy(Comparator.naturalOrder()),
        TreeMap::new));
//{2017=Java Jive, 2018=Lambda Dancing}
```

The merge function specified is equivalent to the following lambda expression, returning the greater of two strings:

```
(str1, str2) -> str1.compareTo(str2) > 0 ? str1 : str2
```

### Collecting to a ConcurrentMap

If the collector returned by the `Collectors.toMap()` method is used in a parallel stream, the multiple partial maps created during parallel execution are merged by the collector to create the final result map. Merging maps can be expensive if keys from one map are merged into another. To address the problem, the `Collectors` class provides the three overloaded methods `toConcurrentMap()`, analogous to the three `toMap()` methods, that return a *concurrent collector*—that is, a collector that uses a single *concurrent map* to perform the reduction. A concurrent map is *thread-safe* and *unordered*. A concurrent map implements the `java.util.concurrent.ConcurrentMap` interface, which is a subinterface of `java.util.Map` interface (§23.7, p. 1482).

Using a concurrent map avoids merging of maps during parallel execution, as a single map is created that is used concurrently to accumulate the results from the execution of each substream. However, the concurrent map is *unordered*—any

encounter order in the stream is ignored. Usage of the `toConcurrentMap()` method is illustrated by the following example of a parallel stream to create a concurrent map of CD titles released each year.

```
ConcurrentMap<Year, String> concMapYearToTitles = CD.cdList
    .parallelStream()
    .collect(Collectors.toConcurrentMap(CD::year, CD::title,
        (tt, t) -> tt + ":" + t));
//{2017=Java Jam:Java Jive, 2018=Lambda Dancing:Hot Generics:Keep on Erasing}
```

## Joining

The `joining()` method creates a collector for concatenating the input elements of type `CharSequence` to a single immutable `String`. However, internally it uses a mutable `StringBuilder`. Note that the collector returned by the `joining()` methods performs *functional* reduction, as its result is a single immutable string.

```
static Collector<CharSequence,?,String> joining()
static Collector<CharSequence,?,String> joining(CharSequence delimiter)
static Collector<CharSequence,?,String> joining(CharSequence delimiter,
        CharSequence prefix,
        CharSequence suffix)
```

Return a `Collector` that concatenates `CharSequence` elements into a `String`. The first method concatenates in encounter order. So does the second method, but this method separates the elements by the specified `delimiter`. The third method in addition applies the specified prefix and suffix to the result of the concatenation.

The wildcard `?` is a type parameter that is used internally by the collector.

The methods preserve the encounter order, if the stream has one.

Among the classes that implement the `CharSequence` interface are the `String`, `StringBuffer`, and `StringBuilder` classes.

The stream pipelines below concatenate CD titles to illustrate the three overloaded `joining()` methods. The `CharSequence` elements are `Strings`. The strings are concatenated in the stream encounter order, which is the positional order for lists. The zero-argument `joining()` method at (1) performs string concatenation of the CD titles using a `StringBuilder` internally, and returns the result as a string.

```
String concatTitles1 = CD.cdList.stream()           // Stream<CD>
    .map(CD::title)                               // Stream<String>
    .collect(Collectors.joining());               // (1)
//Java JiveJava JamLambda DancingKeep on ErasingHot Generics
```

The single-argument `joining()` method at (2) concatenates the titles using the specified delimiter.

```
String concatTitles2 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.joining(", "));           // (2) Delimiter
//Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics
```

The three-argument `joining()` method at (3) concatenates the titles using the specified delimiter, prefix, and suffix.

```
String concatTitles3 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.joining(", ", "[", "]")); // (3) Delimiter, Prefix, Suffix
//[[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

## Grouping

Classifying elements into groups based on some criteria is a very common operation. An example is classifying CDs into groups according to the number of tracks on them (this sounds esoteric, but it will illustrate the point). Such an operation can be accomplished by the collector returned by the `groupingBy()` method. The method is passed a *classifier function* that is used to classify the elements into different groups. The result of the operation is a *classification map* whose entries are the different groups into which the elements have been classified. The key in a map entry is the result of applying the classifier function on the element. The key is extracted from the element based on some property of the element—for example, the number of tracks on the CD. The value associated with a key in a map entry comprises those elements that belong to the same group. The operation is analogous to the *group-by* operation in databases.

There are three versions of the `groupingBy()` method that provide increasingly more control over the grouping operation.

```
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
    Function<? super T,? extends K> classifier)

static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(
    Function<? super T,? extends K> classifier,
    Collector<? super T,A,D> downstream)

static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(
    Function<? super T,? extends K> classifier,
    Supplier<M> mapSupplier,
    Collector<? super T,A,D> downstream)
```

The Collector returned by the `groupingBy()` methods implements a *group-by* operation on input elements to create a *classification map*.

The *classifier function* maps elements of type `T` to keys of some type `K`. These keys determine the groups in the classification map.

The collector returned by the single-argument method produces a classification map of type `Map<K, List<T>>`. The keys in this map are the results from applying the specified classifier function to the input elements. The input elements that map to the same key are accumulated into a `List` by the *default downstream collector* `Collector.toList()`.

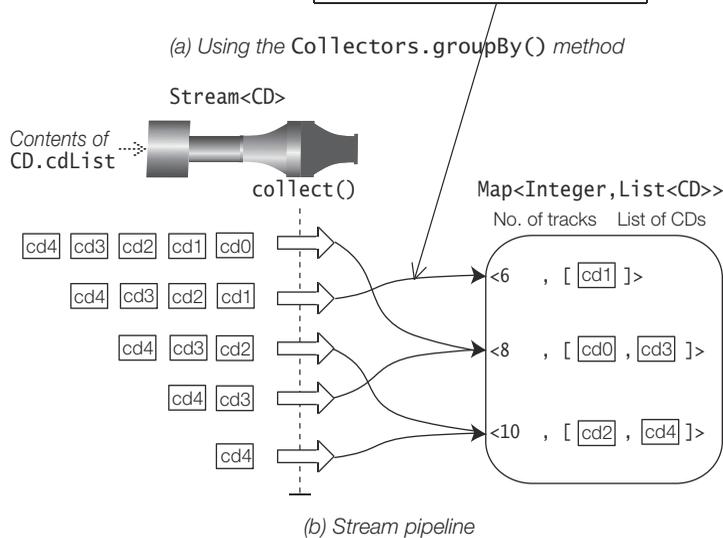
The two-argument method accepts a downstream collector, in addition to the classifier function. The collector returned by the method is composed with the specified downstream collector that performs a reduction operation on the input elements that map to the same key. It operates on elements of type T and produces a result of type D. The result of type D produced by the downstream collector is the value associated with the key of type K. The composed collector thus results in a classification map of type `Map<K, D>`.

The three-argument method accepts a map supplier as its second parameter. It creates an empty classification map of type M that is used by the composed collector. The result is a classification map of type M whose key and value types are K and D, respectively.

Figure 16.16 illustrates the `groupingBy()` operation by grouping CDs according to the number of tracks on them. The classifier function `CD::noOfTracks` extracts the number of tracks from a CD that acts as a key in the classification map (`Map<Integer, List<CD>>`). Since the call to the `groupingBy()` method in Figure 16.16 does not specify a downstream collector, the default downstream collector `Collector.toList()` is used to accumulate CDs that have the same number of tracks. The number of groups—that is, the number of distinct keys—is equal to the number of distinct values for the number of tracks on the CDs. Each distinct value for the number of tracks is associated with the list of CDs having that value as the number of tracks.

Figure 16.16 Grouping

```
// Query: Group by number of tracks.
Map<Integer, List<CD>> map11 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks)); // (1)
```



The three stream pipelines below result in a classification map that is equivalent to the one in Figure 16.16. The call to the `groupingBy()` method at (2) specifies the downstream collector explicitly, and is equivalent to the call in Figure 16.16.

```
Map<Integer, List<CD>> map22 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toList())); // (2)
```

The call to the `groupingBy()` method at (3) specifies the supplier `TreeMap::new` so that a `TreeMap<Integer, List<CD>>` is used as the classification map.

```
Map<Integer, List<CD>> map33 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,
                                   TreeMap::new,
                                   Collectors.toList())); // (3)
```

The call to the `groupingBy()` method at (4) specifies the downstream collector `Collector.toSet()` that uses a set to accumulate the CDs for a group.

```
Map<Integer, Set<CD>> map44 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toSet())); // (4)
```

The classification maps created by the pipelines above will contain the three entries shown below, but only the `groupingBy()` method call at (3) can guarantee that the entries will be sorted in a `TreeMap<Integer, List<CD>>` according to the natural order for the Integer keys.

```
{
  6=[<Jaav, "Java Jam", 6, 2017, JAZZ>],
  8=[<Jaav, "Java Jive", 8, 2017, POP>,
    <Genericos, "Keep on Erasing", 8, 2018, JAZZ>],
  10=[<Funkies, "Lambda Dancing", 10, 2018, POP>,
    <Genericos, "Hot Generics", 10, 2018, JAZZ>]
}
```

In general, any collector can be passed as a downstream collector to the `groupingBy()` method. In the stream pipeline below, the map value in the classification map is a count of the number of CDs having the same number of tracks. The collector `Collector.counting()` performs a functional reduction to count the CDs having the same number of tracks (p. 998).

```
Map<Integer, Long> map55 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.counting()));
//{6=1, 8=2, 10=2}
```

### Multilevel Grouping

The downstream collector in a `groupingBy()` operation can be created by another `groupingBy()` operation, resulting in a *multilevel grouping* operation—also known as a *multilevel classification* or *cascaded grouping* operation. We can extend the multilevel `groupingBy()` operation to any number of levels by making the downstream collector be a `groupingBy()` operation.

The stream pipeline below creates a classification map in which the CDs are first grouped by the number of tracks in a CD at (1), and then grouped by the musical genre of a CD at (2).

```
Map<Integer, Map<Genre, List<CD>>> twoLevelGrp = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,           // (1)
        Collectors.groupingBy(CD::genre)));                // (2)
```

Printing the contents of the resulting classification map would show the following three entries, not necessarily in this order:

```
{
6={JAZZ=[<Jaav, "Java Jam", 6, 2017, JAZZ>]},
8={JAZZ=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>],
  POP=[<Jaav, "Java Jive", 8, 2017, POP>]},
10={JAZZ=[<Genericos, "Hot Generics", 10, 2018, JAZZ>],
    POP=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
}
```

The entries of the resulting classification map can also be illustrated as a two-dimensional matrix, as shown in Figure 16.16, where the CDs are first grouped into rows by the number of tracks, and then grouped into columns by the musical genre. The value of an element in the matrix is a list of CDs which have the same number of tracks (row) and the same musical genre (column).

**Figure 16.17** *Multilevel Grouping as a Two-Dimensional Matrix*

		Genre	
		JAZZ	POP
No. of tracks	6	[ cd1 ]	
	8	[ cd3 ]	[ cd0 ]
	10	[ cd2 ]	[ cd4 ]

The number of groups in the classification map returned by the above pipeline is equal to the number of distinct values for the number of tracks, as in the single-level `groupingBy()` operation. However, each value associated with a key in the outer classification map is now an inner classification map that is managed by the second-level `groupingBy()` operation. The inner classification map has the type `Map<Genre, List<CD>>`; in other words, the key in the inner classification map is the musical genre of the CD and the value associated with this key is a `List` of CDs with this musical genre. It is the second-level `groupingBy()` operation that is responsible for grouping each CD in the inner classification map. Since no explicit downstream collector is specified for the second-level `groupingBy()` operation, it uses the default downstream collector `Collector.toList()`.

We can modify the multilevel `groupingBy()` operation to count the CDs that have the same musical genre and the same number of tracks by specifying an explicit downstream collector for the second-level `groupingBy()` operation, as shown at (3).

The collector `Collectors.counting()` at (3) performs a functional reduction by accumulating the count for CDs with the same number of tracks and the same musical genre in the inner classification map (p. 998).

```
Map<Integer, Map<Genre, Long>> twoLevelGrp2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,
        Collectors.groupingBy(CD::genre,
            Collectors.counting()))); // (3)
```

Printing the contents of the resulting classification map produced by this multi-level `groupingBy()` operation would show the following three entries, again not necessarily in this order:

```
{6={JAZZ=1}, 8={JAZZ=1, POP=1}, 10={JAZZ=1, POP=1}}
```

It is instructive to compare the entries in the resulting classification maps in the two examples illustrated here.

To truly appreciate the `groupingBy()` operation, the reader is highly encouraged to implement the multilevel grouping examples in an imperative style, without using the Stream API. Good luck!

### Grouping to a ConcurrentMap

If the collector returned by the `Collectors.groupingBy()` method is used in a parallel stream, the partial maps created during execution are merged to create the final map—as in the case of the `Collectors.toMap()` method (p. 983). Merging maps can carry a performance penalty. The `Collectors` class provides the three `groupingBy-Concurrent()` overloaded methods, analogous to the three `groupingBy()` methods, that return a *concurrent collector*—that is, a collector that uses a single *concurrent map* to perform the reduction. The entries in such a map are unordered. A concurrent map implements the `java.util.concurrent.ConcurrentMap` interface (§23.7, p. 1482).

Usage of the `groupingByConcurrent()` method is illustrated by the following example of a parallel stream to create a concurrent map of the number of CDs that have the same number of tracks.

```
ConcurrentMap<Integer, Long> map66 = CD.cdList
    .parallelStream()
    .collect(Collectors.groupingByConcurrent(CD::noOfTracks,
        Collectors.counting()));

//{6=1, 8=2, 10=2}
```

## Partitioning

Partitioning is a special case of grouping. The classifier function that was used for grouping is now a *partitioning predicate* in the `partitioningBy()` method. The predicate function returns the boolean value `true` or `false`. As the keys of the resulting map are determined by the classifier function, the keys are determined by the partitioning predicate in the case of partitioning. Thus the keys are always of type

Boolean, implying that the classification map can have, at most, two map entries. In other words, the `partitioningBy()` method can only create, at most, two partitions from the input elements. The map value associated with a key in the resulting map is managed by a downstream collector, as in the case of the `groupingBy()` method.

There are two versions of the `partitioningBy()` method:

```
static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
    Predicate<? super T> predicate)

static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
    Predicate<? super T> predicate,
    Collector<? super T,A,D> downstream)
```

The collector returned by the first method produces a classification map of type `Map<Boolean, List<T>>`. The keys in this map are the results from applying the partitioning predicate to the input elements. The input elements that map to the same Boolean key are accumulated into a `List` by the *default downstream collector* `Collector.toList()`.

The second method accepts a downstream collector, in addition to the partitioning predicate. The collector returned by the method is composed with the specified downstream collector that performs a reduction operation on the input elements that map to the same key. It operates on elements of type `T` and produces a result of type `D`. The result of type `D` produced by the downstream collector is the value associated with the key of type `Boolean`. The composed collector thus results in a resulting map of type `Map<Boolean, D>`.

Figure 16.18 illustrates the `partitioningBy()` operation by partitioning CDs according to the predicate `CD::isPop` that determines whether a CD is a pop music CD. The result of the partitioning predicate acts as the key in the resulting map of type `Map<Boolean, List<CD>>`. Since the call to the `partitioningBy()` method in Figure 16.18 does not specify a downstream collector, the default downstream collector `Collector.toList()` is used to accumulate CDs that map to the same key. The resulting map has two entries or partitions: one for CDs that are pop music CDs and one for CDs that are not. The two entries of the resulting map are also shown below:

```
{false=[<Jaav, "Java Jam", 6, 2017, JAZZ>,
        <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
        <Genericos, "Hot Genericos", 10, 2018, JAZZ>],
 true=[<Jaav, "Java Jive", 8, 2017, POP>,
       <Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

The values in a partition can be obtained by calling the `Map.get()` method:

```
List<CD> popCDs = map1.get(true);
List<CD> nonPopCDs = map1.get(false);
```

The stream pipeline at (2) is equivalent to the one in Figure 16.18, where the downstream collector is specified explicitly.

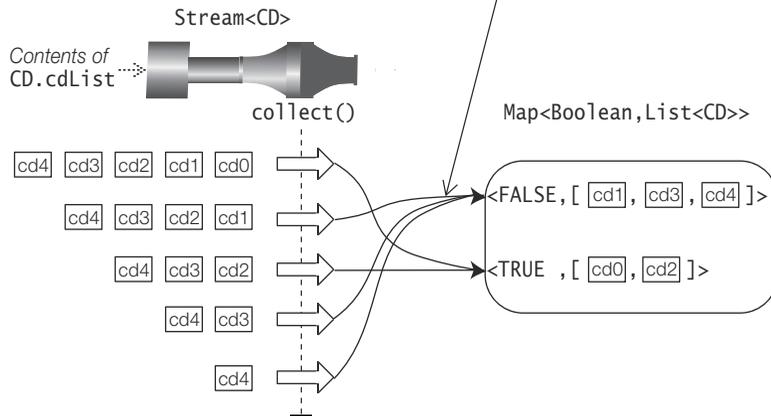
```
Map<Boolean, List<CD>> map2 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.toList())); // (2)
```

We could have composed a stream pipeline to filter the CDs that are pop music CDs and collected them into a list. We would have to compose a second pipeline to find the CDs that are *not* pop music CDs. However, the `partitioningBy()` method does both in a single operation.

**Figure 16.18** *Partitioning*

```
// Query: Partition by whether it is a pop music CD.
Map<Boolean, List<CD>> map1 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop)); // (1)
```

(a) Using the `Collectors.partitioningBy()` method



(b) Stream pipeline

Analogous to the `groupingBy()` method, any collector can be passed as a downstream collector to the `partitioningBy()` method. In the stream pipeline below, the downstream collector `Collector.counting()` performs a functional reduction to count the number of CDs associated with a key (p. 998).

```
Map<Boolean, Long> map3 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.counting()));
//{false=3, true=2}
```

### Multilevel Partitioning

Like the `groupingBy()` method, the `partitioningBy()` operation can be used in multilevel classification. The downstream collector in a `partitioningBy()` operation can be created by another `partitioningBy()` operation, resulting in a *multilevel partitioning* operation—also known as a *cascaded partitioning* operation. The downstream collector can also be a `groupingBy()` operation.

In the stream pipeline below, the CDs are partitioned at (1): one partition for CDs that are pop music CDs, and one for those that are not. The CDs that are associated with a key are grouped by the year in which they were released. Note that the CDs

that were released in a year are accumulated into a `List` by the default downstream collector `Collector.toList()` that is employed by the `groupingBy()` operation at (2).

```
Map<Boolean, Map<Year, List<CD>>> map1 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop,           // (1)
        Collectors.groupingBy(CD::year)));                // (2)
```

Printing the contents of the resulting map would show the following two entries, not necessarily in this order.

```
{false={2017=[<Jaav, "Java Jam", 6, 2017, JAZZ>],
        2018=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
              <Genericos, "Hot Genericos", 10, 2018, JAZZ>]},
 true={2017=[<Jaav, "Java Jive", 8, 2017, POP>],
       2018=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}}
```

## Filtering Adapter for Downstream Collectors

The `filtering()` method of the `Collectors` class encapsulates a *predicate* and a *downstream collector* to create an *adapter* for a *filtering* operation. (See also the `filter()` intermediate operation, p. 912.)

```
static <T,A,R> Collector<T,?,R> filtering(
    Predicate<? super T> predicate,
    Collector<? super T,A,R> downstream)
```

Returns a `Collector` that applies the predicate to input elements of type `T` to determine which elements should be passed to the downstream collector. This downstream collector accumulates them into results of type `R`, where the type parameter `A` is the intermediate accumulation type of the downstream collector.

The following code uses the `filtering()` operation at (2) to group pop music CDs according to the number of tracks on them. The `groupingBy()` operation at (1) creates the groups based on the number of tracks on the CDs, but the `filtering()` operation only allows pop music CDs to pass downstream to be accumulated.

```
// Filtering downstream from grouping.
Map<Integer, List<CD>> grpByTracksFilterByPopCD = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,           // (1)
        Collectors.filtering(CD::isPop, Collectors.toList()))); // (2)
```

Printing the contents of the resulting map would show the entries below, not necessarily in this order. Note that the output shows that there was one or more CDs with six tracks, but there were no pop music CDs. Hence the list of CDs associated with key 6 is empty.

```
{6=[],
 8=[<Jaav, "Java Jive", 8, 2017, POP>],
10=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

However, if we run the same query using the `filter()` intermediate stream operation at (1) prior to grouping, the contents of the result map are different, as shown below.

```
// Filtering before grouping.
Map<Integer, List<CD>> filterByPopCDGrpByTracks = CD.cdList.stream()
    .filter(CD::isPop) // (1)
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toList()));
```

Contents of the result map show that only entries that have a non-empty list as a value are contained in the map. This is not surprising, as any non-pop music CD is discarded before grouping, so only pop music CDs are grouped.

```
{8=[<Jaav, "Java Jive", 8, 2017, POP>],
 10=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

There are no surprises with partitioning, regardless of whether filtering is done before or after the partitioning, as partitioning always results in a map with two entries: one for the `Boolean.TRUE` key and one for the `Boolean.FALSE` key. The code below partitions CDs released in 2018 according to whether a CD is a pop music CD or not.

```
// Filtering downstream from partitioning.
Map<Boolean, List<CD>> partbyPopCDsFilterByYear = CD.cdList.stream() // (1)
    .collect(Collectors.partitioningBy(CD::isPop,
        Collectors.filtering(cd -> cd.year().equals(Year.of(2018)),
            Collectors.toList()))); // (2)

// Filtering before partitioning.
Map<Boolean, List<CD>> filterByYearPartbyPopCDs = CD.cdList.stream() // (2)
    .filter(cd -> cd.year().equals(Year.of(2018)))
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.toList()));
```

Both queries at (1) and (2) above will result in the same entries in the result map:

```
{false=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
        <Genericos, "Hot Genericos", 10, 2018, JAZZ>],
 true=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

## Mapping Adapter for Downstream Collectors

The `mapping()` method of the `Collectors` class encapsulates a *mapper function* and a *downstream collector* to create an *adapter* for a *mapping* operation. (See also the `map()` intermediate operation, p. 921.)

```
static <T,U,A,R> Collector<T,?,R> mapping(
    Function<? super T,? extends U> mapper,
    Collector<? super U,A,R> downstream)
```

Returns a `Collector` that applies the `mapper` function to input elements of type `T` and provides the mapped results of type `U` to the `downstream` collector that accumulates them into results of type `R`.

In other words, the method adapts a `downstream` collector accepting elements of type `U` to one accepting elements of type `T` by applying a `mapper` function to each input element before accumulation, where type parameter `A` is the intermediate accumulation type of the `downstream` collector.

The `mapping()` method at (1) creates an adapter that accumulates a set of CD titles in each year for a stream of CDs. The mapper function maps a CD to its title so that the downstream collector can accumulate the titles in a set.

```
Map<Year, Set<String>> titlesByYearInSet = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(                // (1)
            CD::title,                    // Mapper
            Collectors.toSet())));        // Downstream collector
System.out.println(titlesByYearInSet);
// {2017=[Java Jive, Java Jam],
// 2018=[Hot Generics, Lambda Dancing, Keep on Erasing]}
```

The `mapping()` method at (2) creates an adapter that joins CD titles in each year for a stream of CDs. The mapper function maps a CD to its title so that the downstream collector can join the titles.

```
Map<Year, String> joinTitlesByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(                // (2)
            CD::title,
            Collectors.joining(":"))));
System.out.println(joinTitlesByYear);
// {2017=Java Jive:Java Jam,
// 2018=Lambda Dancing:Keep on Erasing:Hot Generics}
```

The `mapping()` method at (3) creates an adapter that counts the number of CD tracks for each year for a stream of CDs. The mapper function maps a CD to its number of tracks so that the downstream collector can count the total number of tracks.

```
Map<Year, Long> TotalNumOfTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(                // (3)
            CD::noOfTracks,
            Collectors.counting())));
System.out.println(TotalNumOfTracksByYear);    // {2017=2, 2018=3}
```

## Flat Mapping Adapter for Downstream Collectors

The `flatMap()` method of the `Collectors` class encapsulates a *mapper function* and a *downstream collector* to create an *adapter* for a *flat mapping* operation. (See also the `flatMap()` intermediate operation, p. 924.)

```
static <T,U,A,R> Collector<T,?,R> flatMapping(
    Function<? super T,? extends Stream<? extends U>> mapper,
    Collector<? super U,A,R> downstream)
```

Returns a `Collector` that applies the specified mapper function to input elements of type `T` and provides the mapped results of type `U` to the downstream collector that accumulates them into results of type `R`.

That is, the method adapts a downstream collector accepting elements of type `U` to one accepting elements of type `T` by applying a flat mapping function to each input element before accumulation, where type parameter `A` is the intermediate accumulation type of the downstream collector.

The flat mapping function maps an input element to a *mapped stream* whose elements are *flattened* (p. 924) and passed downstream. Each mapped stream is closed after its elements have been flattened. An empty stream is substituted if the mapped stream is `null`.

Given the lists of CDs below, we wish to find all unique CD titles in the lists. A stream of CD lists is created at (1). Each CD list is used to create a stream of CDs whose elements are flattened into the output stream of CDs at (2). Each CD is then mapped to its title at (3), and unique CD titles are accumulated into a set at (4). (Compare this example with the one in Figure 16.9, p. 925, using the `flatMap()` stream operation.)

```
// Given lists of CDs:
List<CD> cdListA = List.of(CD.cd0, CD.cd1);
List<CD> cdListB = List.of(CD.cd0, CD.cd1, CD.cd1);

// Find unique CD titles in the given lists:
Set<String> set =
    Stream.of(cdListA, cdListB)                // (1) Stream<List<CD>>
        .collect(Collectors.flatMapping(List::stream, // (2) Flatten to Stream<CD>
            Collectors.mapping(CD::title, // (3) Stream<String>
                Collectors.toSet())));      // (4) Set<String>
```

Set of unique CD titles in the CD lists:

```
[Java Jive, Java Jam]
```

The collectors returned by the `flatMap()` method are designed to be used in multilevel grouping operations (p. 987), such as downstream from `groupingBy()` or `partitionBy()` operations. Example 16.13 illustrates such a use with the `groupingBy()` operation.

In Example 16.13, the class `RadioPlaylist` at (1) represents a radio station by its name and a list of CDs played by the radio station. Three CD lists are constructed at (2) and used to construct three radio playlists at (3). The radio playlists are stored in a common list of radio playlists at (4). A query is formulated at (5) to find the unique titles of CDs played by each radio station. Referring to the line numbers in Example 16.13:

- (6) A stream of type `Stream<RadioPlaylist>` is created from the list `radioPlaylists` of type `RadioPlaylist`.
- (7) The radio playlists are grouped according to the name of the radio station (`String`).
- (8) Each radio playlist of type `RadioPlaylist` is used as the source of a stream, which is then flattened into the output stream of type `Stream<CD>` by the `flatMap()` operation.
- (9) Each CD in the stream is mapped to its title.

(10) Each unique CD title is accumulated into the result set of each radio station (Set<String>).

The query at (5) uses four collectors. The result map has the type Map<String, List<String>>. The output shows the unique titles of CDs played by each radio station.

.....

**Example 16.13** *Flat mapping*

```
import java.util.List;

// Radio station with a playlist.
public class RadioPlaylist {                                     // (1)
    private String radioStationName;
    private List<CD> Playlist;

    public RadioPlaylist(String radioStationName, List<CD> cdList) {
        this.radioStationName = radioStationName;
        this.Playlist = cdList;
    }

    public String getRadioStationName() { return this.radioStationName; }
    public List<CD> getPlaylist() { return this.Playlist; }
}

.....

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

public class CollectorsFlatMapping {
    public static void main(String[] args) {
        // Some lists of CDs:                                     (2)
        List<CD> cdList1 = List.of(CD.cd0, CD.cd1, CD.cd1, CD.cd2);
        List<CD> cdList2 = List.of(CD.cd0, CD.cd0, CD.cd3);
        List<CD> cdList3 = List.of(CD.cd0, CD.cd4);

        // Some radio playlists:                                  (3)
        RadioPlaylist p11 = new RadioPlaylist("Radio JVM", cdList1);
        RadioPlaylist p12 = new RadioPlaylist("Radio JRE", cdList2);
        RadioPlaylist p13 = new RadioPlaylist("Radio JAR", cdList3);

        // List of radio playlists:                                (4)
        List<RadioPlaylist> radioPlaylists = List.of(p11, p12, p13);

        // Map of radio station names and set of CD titles they played: (5)
        Map<String, Set<String>> map = radioPlaylists.stream()      // (6)
            .collect(Collectors.groupingBy(RadioPlaylist::getRadioStationName, // (7)
                Collectors.flatMapping(rpl -> rpl.getPlaylist().stream(), // (8)
                    Collectors.mapping(CD::title, // (9)
                        Collectors.toSet())))); // (10)
        System.out.println(map);
    }
}
```

Output from the program (*edited to fit on the page*):

```
{Radio JAR=[Hot Generics, Java Jive],
 Radio JVM=[Java Jive, Lambda Dancing, Java Jam],
 Radio JRE=[Java Jive, Keep on Erasing]}
```

## Finishing Adapter for Downstream Collectors

The `collectingAndThen()` method encapsulates a *downstream collector* and a *finisher function* to allow the result of the collector to be adapted by the finisher function.

```
static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(
    Collector<T,A,R> downstream,
    Function<R,RR> finisher)
```

Returns a Collector that performs the operation of the downstream collector on input elements of type T, followed by applying the finisher function on the result of type R produced by the downstream collector. The final result is of type RR, the result of the finisher function. In other words, the method adapts a collector to perform an additional finishing transformation.

In the call to the `collectAndThen()` method at (1), the collector `Collectors.maxBy()` at (2) produces an `Optional<Integer>` result that is the maximum CD by number of tracks in each group. The finisher function at (3) extracts the value from the `Optional<Integer>` result, if there is one; otherwise, it returns 0. The `collectAndThen()` method adapts the `Optional<Integer>` result of its argument collector to an `Integer` value by applying the finisher function.

```
Map<Year, Integer> maxTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.collectingAndThen( // (1)
            Collectors.maxBy(Comparator.comparing(CD::noOfTracks)), // (2)
            optCD -> optCD.map(CD::noOfTracks).orElse(0)) // (3)
    ));
System.out.println(maxTracksByYear); // {2017=8, 2018=10}
```

In the call to the `collectAndThen()` method at (4), the collector `Collectors.averagingDouble()` at (5) produces a result of type `Double` that is the average number of tracks in each group. The finisher function at (6) maps the `Double` average value to a string with the specified number format.

```
Map<Genre, String> avgTracksByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.collectingAndThen( // (4)
            Collectors.averagingDouble(CD::noOfTracks), // (5)
            d -> String.format("%.1f", d)) // (6)
    ));
System.out.println(avgTracksByGenre); // {JAZZ=8.0, POP=9.0}
```

## Downstream Collectors for Functional Reduction

The collectors we have seen so far perform a *mutable reduction* to some *mutable container*, except for the functional reduction implemented by the `joining()` method (p. 984). The `Collectors` class also provides static factory methods that implement collectors which perform *functional reduction* to compute common statistics, such as summing, averaging, finding maximum and minimum values, and the like.

Like any other collector, the collectors that perform functional reduction can also be used in a standalone capacity as a parameter of the `collect()` method and as a downstream collector in a composed collector. However, these collectors are most useful when used as downstream collectors.

Collectors performing functional reduction have counterparts in terminal operations for streams that provide equivalent reduction operations (Table 16.8, p. 1008).

### Counting

The collector created by the `Collectors.counting()` method performs a functional reduction to count the input elements.

```
static <T> Collector<T,?,Long> counting()
```

The collector returned counts the number of input elements of type `T`. If there are no elements, the result is `Long.valueOf(0L)`. Note that the result is of type `Long`.

The wildcard `?` represents any type, and in the method declaration, it is the type parameter for the mutable type that is accumulated by the reduction operation.

In the stream pipeline at (1), the collector `Collectors.counting()` is used in a standalone capacity to count the number of jazz music CDs.

```
Long numOfJazzCds1 = CD.cdList.stream().filter(CD::isJazz)
    .collect(Collectors.counting());           // (1) Standalone collector
System.out.println(numOfJazzCds1);         // 3
```

In the stream pipeline at (2), the collector `Collectors.counting()` is used as a downstream collector in a grouping operation that groups the CDs by musical genre and uses the downstream collector to count the number of CDs in each group.

```
Map<Genre, Long> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.counting()));           // (2) Downstream collector
System.out.println(grpByGenre);           // {POP=2, JAZZ=3}
System.out.println(grpByGenre.get(Genre.JAZZ)); // 3
```

The collector `Collectors.counting()` performs effectively the same functional reduction as the `Stream.count()` terminal operation (p. 953) at (3).

```
Long numOfJazzCds2 = CD.cdList.stream().filter(CD::isJazz)
    .count();                               // (3) Stream.count()
System.out.println(numOfJazzCds2);         // 3
```

### Finding Min/Max

The collectors created by the `Collectors.maxBy()` and `Collectors.minBy()` methods perform a functional reduction to find the maximum and minimum elements in the input elements, respectively. As there might not be any input elements, an `Optional<T>` is returned as the result.

```
static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> cmp)
static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> cmp)
```

Return a collector that produces an `Optional<T>` with the maximum or minimum element of type `T` according to the specified `Comparator`, respectively.

The natural order comparator for CDs defined at (1) is used in the stream pipelines below to find the maximum CD. The collector `Collectors.maxBy()` is used as a standalone collector at (2), using the natural order comparator to find the maximum CD. The `Optional<CD>` result can be queried for the value.

```
Comparator<CD> natCmp = Comparator.naturalOrder(); // (1)

Optional<CD> maxCD = CD.cdList.stream()
    .collect(Collectors.maxBy(natCmp));           // (2) Standalone collector
System.out.println("Max CD: "
    + maxCD.map(CD::title).orElse("No CD.));     // Max CD: Java Jive
```

In the pipeline below, the CDs are grouped by musical genre, and the CDs in each group are reduced to the maximum CD by the downstream collector `Collectors.maxBy()` at (3). Again, the downstream collector uses the natural order comparator, and the `Optional<CD>` result in each group can be queried.

```
// Group CDs by musical genre, and max CD in each group.
Map<Genre, Optional<CD>> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.maxBy(natCmp)));           // (3) Downstream collector
System.out.println(grpByGenre);
//{JAZZ=Optional[<Jaav, "Java Jam", 6, 2017, JAZZ>],
// POP=Optional[<Jaav, "Java Jive", 8, 2017, POP>]}

System.out.println("Title of max Jazz CD: "
    + grpByGenre.get(Genre.JAZZ)
        .map(CD::title)
        .orElse("No CD.));                   // Title of max Jazz CD: Java Jam
```

The collectors created by the `Collectors.maxBy()` and `Collectors.minBy()` methods are effectively equivalent to the `max()` and `min()` terminal operations provided by the stream interfaces (p. 954), respectively. In the pipeline below, the `max()` terminal operation reduces the stream of CDs to the maximum CD at (4) using the natural order comparator, and the `Optional<CD>` result can be queried.

```
Optional<CD> maxCD1 = CD.cdList.stream()
    .max(natCmp);                             // (4) max() on Stream<CD>.
System.out.println("Title of max CD: "
    + maxCD1.map(CD::title)
        .orElse("No CD.));                   // Title of max CD: Java Jive
```

## Summing

The summing collectors perform a functional reduction to produce the sum of the numeric results from applying a numeric-valued function to the input elements.

```
static <T> Collector<T,?,NumType> summingNumType(
    ToNumTypeFunction<? super T> mapper)
```

Returns a collector that produces the sum of a *numtype*-valued function applied to the input elements. If there are no input elements, the result is zero. The result is of *NumType*.

*NumType* is `Int` (but it is `Integer` when used as a type name), `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`.

The collector returned by the `Collectors.summingInt()` method is used at (1) as a standalone collector to find the total number of tracks on the CDs. The mapper function `CD::noOfTracks` passed as an argument extracts the number of tracks from each CD on which the functional reduction is performed.

```
Integer sumTracks = CD.cdList.stream()
    .collect(Collectors.summingInt(CD::noOfTracks)); // (1) Standalone collector
System.out.println(sumTracks); // 42
```

In the pipeline below, the CDs are grouped by musical genre, and the number of tracks on CDs in each group summed by the downstream collector is returned by the `Collectors.summingInt()` method at (2).

```
Map<Genre, Integer> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.summingInt(CD::noOfTracks))); // (2) Downstream collector
System.out.println(grpByGenre); // {POP=18, JAZZ=24}
System.out.println(grpByGenre.get(Genre.JAZZ)); // 24
```

The collector `Collectors.summingInt()` performs effectively the same functional reduction at (3) as the `IntStream.sum()` terminal operation (p. 973).

```
int sumTracks2 = CD.cdList.stream() // (3) Stream<CD>
    .mapToInt(CD::noOfTracks) // IntStream
    .sum();
System.out.println(sumTracks2); // 42
```

## Averaging

The averaging collectors perform a functional reduction to produce the average of the numeric results from applying a numeric-valued function to the input elements.

```
static <T> Collector<T,?,Double> averagingNumType(
    ToNumTypeFunction<? super T> mapper)
```

Returns a collector that produces the arithmetic mean of a *numtype*-valued function applied to the input elements. If there are no input elements, the result is zero. The result is of type `Double`.

*NumType* is `Int`, `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`.

The collector returned by the `Collectors.averagingInt()` method is used at (1) as a standalone collector to find the average number of tracks on the CDs. The mapper function `CD::noOfTracks` passed as an argument extracts the number of tracks from each CD on which the functional reduction is performed.

```
Double avgNoOfTracks1 = CD.cdList.stream()
    .collect(Collectors
        .averagingInt(CD::noOfTracks));           // (1) Standalone collector
System.out.println(avgNoOfTracks1);           // 8.4
```

In the pipeline below, the CDs are grouped by musical genre, and the downstream collector `Collectors.averagingInt()` at (2) calculates the average number of tracks on the CDs in each group.

```
Map<Genre, Double> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.averagingInt(CD::noOfTracks) // (2) Downstream collector
    ));
System.out.println(grpByGenre);               // {POP=9.0, JAZZ=8.0}
System.out.println(grpByGenre.get(Genre.JAZZ)); // 8.0
```

The collector created by the `Collectors.averagingInt()` method performs effectively the same functional reduction as the `IntStream.average()` terminal operation (p. 974) at (3).

```
OptionalDouble avgNoOfTracks2 = CD.cdList.stream() // Stream<CD>
    .mapToInt(CD::noOfTracks)                       // IntStream
    .average();                                     // (3)
System.out.println(avgNoOfTracks2.orElse(0.0));     // 8.4
```

### Summarizing

The summarizing collector performs a functional reduction to produce summary statistics (count, sum, min, max, average) on the numeric results of applying a numeric-valued function to the input elements.

```
static <T> Collector<T,?,NumTypeSummaryStatistics> summarizingNumType(
    ToNumTypeFunction<? super T> mapper)
```

Returns a collector that applies a *numtype*-valued mapper function to the input elements, and returns the summary statistics for the resulting values.

*NumType* is `Int` (but it is `Integer` when used as a type name), `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`.

The collector `Collectors.summarizingInt()` is used at (1) as a standalone collector to summarize the statistics for the number of tracks on the CDs. The mapper function `CD::noOfTracks` passed as an argument extracts the number of tracks from each CD on which the functional reduction is performed.

```
IntSummaryStatistics stats1 = CD.cdList.stream()
    .collect(
        Collectors.summarizingInt(CD::noOfTracks) // (1) Standalone collector
    );
```

```
System.out.println(stats1);
// IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

The `IntSummaryStatistics` class provides `get` methods to access the individual results (p. 974).

In the pipeline below, the CDs are grouped by musical genre, and the downstream collector created by the `Collectors.summarizingInt()` method at (2) summarizes the statistics for the number of tracks on the CDs in each group.

```
Map<Genre, IntSummaryStatistics> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.summarizingInt(CD::noOfTracks))); // (2) Downstream collector
System.out.println(grpByGenre);
// {POP=IntSummaryStatistics{count=2, sum=18, min=8, average=9.000000, max=10},
//   JAZZ=IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}}
```

```
System.out.println(grpByGenre.get(Genre.JAZZ)); // Summary stats for Jazz CDs.
// IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}
```

The collector returned by the `Collectors.summarizingInt()` method performs effectively the same functional reduction as the `IntStream.summaryStatistics()` terminal operation (p. 974) at (3).

```
IntSummaryStatistics stats2 = CD.cdList.stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics(); // (3)
System.out.println(stats2);
// IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

## Reducing

Collectors that perform common statistical operations, such as counting, averaging, and so on, are special cases of functional reduction that can be implemented using the `Collectors.reducing()` method.

```
static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> bop)
```

Returns a collector that performs functional reduction, producing an `Optional` with the *cumulative* result of applying the binary operator `bop` on the input elements:  $e_1 \text{ bop } e_2 \text{ bop } e_3 \dots$ , where each  $e_i$  is an input element. If there are no input elements, an empty `Optional<T>` is returned.

Note that the collector reduces input elements of type `T` to a result that is an `Optional` of type `T`.

```
static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> bop)
```

Returns a collector that performs functional reduction, producing the *cumulative* result of applying the binary operator `bop` on the input elements: `identity bop  $e_1$  bop  $e_2$  ..., where each  $e_i$  is an input element. The identity value is the initial value to accumulate. If there are no input elements, the identity value is returned.`

Note that the collector reduces input elements of type `T` to a result of type `T`.

```
static <T,U> Collector<T,?,U> reducing(
    U identity,
    Function<? super T,? extends U> mapper,
    BinaryOperator<U> bop)
```

Returns a collector that performs a *map-reduce* operation. It maps each input element of type *T* to a mapped value of type *U* by applying the mapper function, and performs functional reduction on the mapped values of type *U* by applying the binary operator *bop*. The *identity* value of type *U* is used as the initial value to accumulate. If the stream is empty, the *identity* value is returned.

Note that the collector reduces input elements of type *T* to a result of type *U*.

Collectors returned by the `Collectors.reducing()` methods effectively perform equivalent functional reductions as the `reduce()` methods of the stream interfaces. However, the three-argument method `Collectors.reducing(identity, mapper, bop)` performs a map-reduce operation using a mapper function and a binary operator *bop*, whereas the `Stream.reduce(identity, accumulator, combiner)` performs a reduction using an accumulator and a combiner (p. 955). The accumulator is a `BiFunction<U,T,U>` that accumulates a partially accumulated result of type *U* with an element of type *T*, whereas the *bop* is a `BinaryOperator<U>` that accumulates a partially accumulated result of type *U* with an element of type *U*.

The following comparators are used in the examples below:

```
// Comparator to compare CDs by title.
Comparator<CD> cmpByTitle = Comparator.comparing(CD::title); // (1)
// Comparator to compare strings by their length.
Comparator<String> byLength = Comparator.comparing(String::length); // (2)
```

The collector returned by the `Collectors.reducing()` method is used as a standalone collector at (3) to find the CD with the longest title. The result of the operation is an `Optional<String>` as there might not be any input elements. This operation is equivalent to using the `Stream.reduce()` terminal operation at (4).

```
Optional<String> longestTitle1 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.reducing(
        BinaryOperator.maxBy(byLength))); // (3) Standalone collector
System.out.println(longestTitle1.orElse("No title")); // Keep on Erasing

Optional<String> longestTitle2 = CD.cdList.stream() // Stream<CD>
    .map(CD::title) // Stream<String>
    .reduce(BinaryOperator.maxBy(byLength)); // (4) Stream.reduce(bop)
```

The collector returned by the one-argument `Collectors.reducing()` method is used as a downstream collector at (5) to find the CD with the longest title in each group classified by the year a CD was released. The collector at (5) is equivalent to the collector returned by the `Collectors.maxBy(cmpByTitle)` method.

```
Map<Year, Optional<CD>> cdWithMaxTitleByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
```

```

        Collectors.reducing(
            BinaryOperator.maxBy(cmpByTitle))
    ));
System.out.println(cdWithMaxTitleByYear);
// {2017=Optional[<Jaav, "Java Jive", 8, 2017, POP>],
// 2018=Optional[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
System.out.println(cdWithMaxTitleByYear.get(Year.of(2018))
    .map(CD::title).orElse("No title")); // Lambda Dancing

```

The collector returned by the three-argument `Collectors.reducing()` method is used as a downstream collector at (6) to find the longest title in each group classified by the year a CD was released. Note that the collector maps a CD to its title. The longest title is associated with the map value for each group classified by the year a CD was released. The collector will return an empty string (i.e., the identity value `""`) if there are no CDs in the stream. In comparison, the collector `Collectors.mapping()` at (7) also maps a CD to its title, and uses the downstream collector `Collectors.maxBy(byLength)` at (7) to find the longest title (p. 993). The result in this case is an `Optional<String>`, as there might not be any input elements.

```

Map<Year, String> longestTitleByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.reducing("", CD::title,
            BinaryOperator.maxBy(byLength))
    ));
System.out.println(longestTitleByYear); // {2017=Java Jive, 2018=Keep on Erasing}
System.out.println(longestTitleByYear.get(Year.of(2018))); // Keep on Erasing

Map<Year, Optional<String>> longestTitleByYear2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(CD::title,
            Collectors.maxBy(byLength))
    ));
System.out.println(longestTitleByYear2);
// {2017=Optional[Java Jive], 2018=Optional[Keep on Erasing]}
System.out.println(longestTitleByYear2.get(Year.of(2018))
    .orElse("No title.")); // Keep on Erasing

```

The pipeline below groups CDs according to the year they were released. For each group, the collector returned by the three-argument `Collectors.reducing()` method performs a map-reduce operation at (8) to map each CD to its number of tracks and accumulate the tracks in each group. This map-reduce operation is equivalent to the collector returned by the `Collectors.summingInt()` method at (9).

```

Map<Year, Integer> noOfTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.reducing(
            0, CD::noOfTracks, Integer::sum));
    ));
System.out.println(noOfTracksByYear); // {2017=14, 2018=28}
System.out.println(noOfTracksByYear.get(Year.of(2018))); // 28

```

```
Map<Year, Integer> noOfTracksByYear2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.summingInt(CD::noOfTracks))); // (9) Special case collector
```

## Summary of Static Factory Methods in the Collectors Class

The static factory methods of the `Collectors` class that create collectors are summarized in Table 16.7. All methods are static generic methods, except for the overloaded `joining()` methods that are not generic. The keyword `static` is omitted, as are the type parameters of a generic method, since these type parameters are evident from the declaration of the formal parameters to the method. The type parameter declarations have also been simplified, where any bound `<? super T>` or `<? extends T>` has been replaced by `<T>`, without impacting the intent of a method. A reference is also provided for each method in the first column.

The last column in Table 16.7 indicates the function type of the corresponding parameter in the previous column. It is instructive to note how the functional interface parameters provide the parameterized behavior to build the collector returned by a method. For example, the method `averagingDouble()` returns a collector that computes the average of the stream elements. The parameter function mapper with the functional interface type `ToDoubleFunction<T>` converts an element of type `T` to a `double` when the collector computes the average for the stream elements.

**Table 16.7** *Static Methods in the Collectors Class*

Method name (ref.)	Return type	Functional interface parameters	Function type of parameters
<code>averagingDouble</code> (p. 1000)	<code>Collector&lt;T,?,Double&gt;</code>	<code>(ToDoubleFunction&lt;T&gt; mapper)</code>	<code>T -&gt; double</code>
<code>averagingInt</code> (p. 1000)	<code>Collector&lt;T,?,Double&gt;</code>	<code>(ToIntFunction&lt;T&gt; mapper)</code>	<code>T -&gt; int</code>
<code>averagingLong</code> (p. 1000)	<code>Collector&lt;T,?,Double&gt;</code>	<code>(ToLongFunction&lt;T&gt; mapper)</code>	<code>T -&gt; long</code>
<code>collectingAndThen</code> (p. 997)	<code>Collector&lt;T,A,RR&gt;</code>	<code>(Collector&lt;T,A,R&gt; downstream, Function&lt;R,RR&gt; finisher)</code>	<code>(T,A) -&gt; R,</code> <code>R -&gt; RR</code>
<code>counting</code> (p. 998)	<code>Collector&lt;T,?,Long&gt;</code>	<code>()</code>	
<code>filtering</code> (p. 992)	<code>Collector&lt;T,?,R&gt;</code>	<code>(Predicate&lt;T&gt; predicate, Collector&lt;T,A,R&gt; downstream)</code>	<code>T -&gt; boolean,</code> <code>(T,A) -&gt; R</code>

Table 16.7 *Static Methods in the Collectors Class (Continued)*

Method name (ref.)	Return type	Functional interface parameters	Function type of parameters
flatMap (p. 994)	Collector<T,?,R>	(Function<T, Stream<U>> mapper, Collector<U,A,R> downstream)	T->Stream<U>,  (U,A) -> R
groupingBy (p. 985)	Collector<T,?, Map<K,List<T>>>	(Function<T,K> classifier)	T -> K
groupingBy (p. 985)	Collector<T,?,Map<K,D>>	(Function<T,K> classifier, Collector<T,A,D> downstream)	T -> K,  (T,A) -> D
groupingBy (p. 985)	Collector<T,?,Map<K,D>>	(Function<T,K> classifier, Supplier<Map<K,D>> mapSupplier, Collector<T,A,D> downstream)	T -> K,  ()->Map<K,D>,  (T,A) -> D
joining (p. 984)	Collector <CharSequence,?,String>	()	
joining (p. 984)	Collector <CharSequence,?,String>	(CharSequence delimiter)	
joining (p. 984)	Collector <CharSequence,?,String>	(CharSequence delimiter, CharSequence prefix, CharSequence suffix)	
mapping (p. 993)	Collector<T,?,R>	(Function<T,U> mapper, Collector<U,A,R> downstream)	T -> U,  (U,A) -> R
maxBy (p. 999)	Collector<T,?,Optional<T>>	(Comparator<T> comparator)	(T,T) -> T
minBy (p. 999)	Collector<T,?,Optional<T>>	(Comparator<T> comparator)	(T,T) -> T
partitioningBy (p. 989)	Collector<T,?, Map<Boolean,List<T>>>	(Predicate<T> predicate)	T -> boolean
partitioningBy (p. 989)	Collector<T,?, Map<Boolean,D>>	(Predicate<T> predicate, Collector<T,A,D> downstream)	T -> boolean,  (T,A) -> D

Table 16.7 *Static Methods in the Collectors Class (Continued)*

Method name (ref.)	Return type	Functional interface parameters	Function type of parameters
reducing (p. 1002)	Collector<T,?,Optional<T>>	(BinaryOperator<T> op)	(T,T) -> T
reducing (p. 1002)	Collector<T,?,T>	(T identity, BinaryOperator<T> op)	T -> T, (T,T) -> T
reducing (p. 1002)	Collector<T,?,U>	(U identity, Function<T,U> mapper, BinaryOperator<U> op)	U -> U, T -> U, (U,U) -> U
summarizingDouble (p. 1001)	Collector<T,?, DoubleSummaryStatistics>	(ToDoubleFunction<T> mapper)	T -> double
summarizingInt (p. 1001)	Collector<T,?, IntSummaryStatistics>	(ToIntFunction<T> mapper)	T -> int
summarizingLong (p. 1001)	Collector<T,?, LongSummaryStatistics>	(ToLongFunction<T> mapper)	T -> long
summingDouble (p. 978)	Collector<T,?,Double>	(ToDoubleFunction<T> mapper)	T -> double
summingInt (p. 978)	Collector<T,?,Integer>	(ToIntFunction<T> mapper)	T -> int
summingLong (p. 978)	Collector<T,?,Long>	(ToLongFunction<T> mapper)	T -> long
toCollection (p. 979)	Collector<T,?,C>	(Supplier<C> collFactory)	() -> C
toList toUnmodifiableList (p. 980)	Collector<T,?,List<T>>	()	
toMap (p. 981)	Collector<T,?,Map<K,U>>	(Function<T,K> keyMapper, Function<T,U> valueMapper)	T -> K, T -> U
toMap (p. 981)	Collector<T,?,Map<K,U>>	(Function<T,K> keyMapper, Function<T,U> valueMapper, BinaryOperator<U> mergeFunction)	T -> K, T -> U, (U,U) -> U

**Table 16.7** *Static Methods in the Collectors Class (Continued)*

Method name (ref.)	Return type	Functional interface parameters	Function type of parameters
toMap (p. 981)	Collector<T,?,Map<K,U>>	(Function<T,K> keyMapper, Function<T,U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<Map<K,U>> mapSupplier)	T -> K, T -> U, (U,U) -> U, ()-> Map<K,U>
toSet toUnmodifiableSet (p. 980)	Collector<T,?,Set<T>>	()	

Table 16.8 shows a comparison of methods in the stream interfaces that perform reduction operations and static factory methods in the Collectors class that implement collectors with equivalent functionality.

**Table 16.8** *Method Comparison: The Stream Interfaces and the Collectors Class*

Method names in the stream interfaces	Static factory method names in the Collectors class
collect (p. 964)	collectingAndThen (p. 997)
count (p. 953)	counting (p. 998)
filter (p. 912)	filtering (p. 992)
flatMap (p. 924)	flatMaping (p. 994)
map (p. 921)	mapping (p. 993)
max (p. 954)	maxBy (p. 999)
min (p. 954)	minBy (p. 999)
reduce (p. 955)	reducing (p. 1002)
toList (p. 972)	toList (p. 980)
average (p. 972)	averagingInt, averagingLong, averagingDouble (p. 1001)
sum (p. 972)	summingInt, summingLong, summingDouble (p. 978)
summaryStatistics (p. 972)	summarizingInt, summarizingLong, summarizingDouble (p. 1001)

## 16.9 Parallel Streams

---

The Stream API makes it possible to execute a sequential stream in parallel without rewriting the code. The primary reason for using parallel streams is to improve performance, but at the same time ensuring that the results obtained are the same, or at least compatible, regardless of the mode of execution. Although the API goes a long way to achieve its aim, it is important to understand the pitfalls to avoid when executing stream pipelines in parallel.

### Building Parallel Streams

The execution mode of an existing stream can be set to parallel by calling the `parallel()` method on the stream (p. 933). The `parallelStream()` method of the `Collection` interface can be used to create a parallel stream with a collection as the data source (p. 897). No other code is necessary for parallel execution, as the data partitioning and thread management for a parallel stream are handled by the API and the JVM. As with any stream, the stream is not executed until a terminal operation is invoked on it.

The `isParallel()` method of the stream interfaces can be used to determine whether the execution mode of a stream is parallel (p. 933).

### Parallel Stream Execution

The Stream API allows a stream to be executed either sequentially or in parallel—meaning that all stream operations can execute either sequentially or in parallel. A sequential stream is executed in a single thread running on one CPU core. The elements in the stream are processed sequentially in a single pass by the stream operations that are executed in the same thread (p. 891).

A parallel stream is executed by different threads, running on multiple CPU cores in a computer. The stream elements are split into substreams that are processed by multiple instances of the stream pipeline being executed in multiple threads. The partial results from processing of each substream are merged (or combined) into a final result (p. 891).

Parallel streams utilize the Fork/Join Framework (§23.3, p. 1447) under the hood for executing parallel tasks. This framework provides support for the thread management necessary to execute the substreams in parallel. The number of threads employed during parallel stream execution is dependent on the CPU cores in the computer.

Figure 16.12, p. 963, illustrates parallel functional reduction using the three-argument `reduce(identity, accumulator, combiner)` terminal operation (p. 962).

Figure 16.14, p. 967, illustrates parallel mutable reduction using the three-argument `collect(supplier, accumulator, combiner)` terminal operation (p. 966).

## Factors Affecting Performance

There are no guarantees that executing a stream in parallel will improve the performance. In this subsection we look at some factors that can affect performance.

### *Benchmarking*

In general, increasing the number of CPU cores and thereby the number of threads that can execute in parallel only scales performance up to a threshold for a given size of data, as some threads might become idle if there is no data left for them to process. The number of CPU cores boosts performance to a certain extent, but it is not the only factor that should be considered when deciding to execute a stream in parallel.

Inherent in the total cost of parallel processing is the start-up cost of setting up the parallel execution. At the onset, if this cost is already comparable to the cost of sequential execution, not much can be gained by resorting to parallel execution.

A combination of the following three factors can be crucial in deciding whether a stream should be executed in parallel:

- *Sufficiently large data size*

The size of the stream must be large enough to warrant parallel processing; otherwise, sequential processing is preferable. The start-up cost can be too prohibitive for parallel execution if the stream size is too small.

- *Computation-intensive stream operations*

If the stream operations are small computations, then the stream size should be proportionately large as to warrant parallel execution. If the stream operations are computation-intensive, the stream size is less significant, and parallel execution can boost performance.

- *Easily splittable stream*

If the cost of splitting the stream into substreams is higher than processing the substreams, employing parallel execution can be futile. Collections like `Array-Lists`, `HashMaps`, and simple arrays are efficiently splittable, whereas `LinkedLists` and IO-based data sources are less efficient in this regard.

Benchmarking—that is, measuring performance—is strongly recommended to decide whether parallel execution will be beneficial. Example 16.14 illustrates a simple scheme where reading the system clock before and after a stream is executed can be used to get a sense of how well a stream performs.

The class `StreamBenchmarks` in Example 16.14 defines five methods, at (1) through (5), that compute the sum of values from 1 to  $n$ . These methods compute the sum in various ways. Each method is executed with four different values of  $n$ ; that is, the stream size is the number of values for summation. The program prints the benchmarks for each method for the different values of  $n$ , which of course can vary, as many factors can influence the results—the most significant one being the number of CPU cores on the computer.

- The methods `seqSumRangeClosed()` at (1) and `parSumRangeClosed()` at (2) perform the computation on a sequential and a parallel stream, respectively, that are created with the `closeRange()` method.

```
return LongStream.rangeClosed(1L, n).sum();           // Sequential stream
...
return LongStream.rangeClosed(1L, n).parallel().sum(); // Parallel stream
```

Benchmarks from Example 16.14:

Size	Sequential	Parallel
1000	0.05681	0.11031
10000	0.06698	0.13979
100000	0.71274	0.52627
1000000	7.02237	4.37249

The terminal operation `sum()` is not computation-intensive. The parallel stream starts to show better performance when the number of values approaches 100000. The stream size is then significantly large for the parallel stream to show better performance. Note that the range of values defined by the arguments of the `rangeClosed()` method can be efficiently split into substreams, as its start and end values are provided.

- The methods `seqSumIterate()` at (3) and `parSumIterate()` at (4) return a sequential and a parallel sequential stream, respectively, that is created with the `iterate()` method.

```
return LongStream.iterate(1L, i -> i + 1).limit(n).sum(); // Sequential
...
return LongStream.iterate(1L, i -> i + 1).limit(n).parallel().sum(); // Parallel
```

Benchmarks from Example 16.14:

Size	Sequential	Parallel
1000	0.08645	0.34696
10000	0.35687	1.27861
100000	3.24083	11.38709
1000000	29.92285	117.87909

The method `iterate()` creates an infinite stream, and the `limit()` intermediate operation truncates the stream according to the value of `n`. The performance of both streams degrades fast when the number of values increases. However, the parallel stream performs worse than the sequential stream in all cases. The values generated by the `iterate()` method are not known before the stream is executed, and the `limit()` operation is also stateful, making the process of splitting the values into substreams inefficient in the case of the parallel stream.

- The method `iterSumLoop()` at (5) uses a `for(;;)` loop to compute the sum.

Benchmarks from Example 16.14:

Size	Iterative
1000	0.00586
10000	0.02330
100000	0.22352
1000000	2.49677

Using a `for(;;)` loop to calculate the sum performs best for all values of `n` compared to the streams, showing that significant overhead is involved in using streams for summing a sequence of numerical values.

In Example 16.14, the methods `measurePerf()` at (6) and `xqtFunctions()` at (13) create the benchmarks for functions passed as parameters. In the `measurePerf()` method, the system clock is read at (8) and the function parameter `func` is applied at (9). The system clock is read again at (10) after the function application at (9) has completed. The execution time calculated at (10) reflects the time for executing the function. Applying the function `func` evaluates the lambda expression or the method reference implementing the `LongFunction` interface. In Example 16.14, the function parameter `func` is implemented by method references that call methods, at (1) through (5), in the `StreamBenchmarks` class whose execution time we want to measure.

```
public static <R> double measurePerf(LongFunction<R> func, long n) { // (6)
    // ...
    double start = System.nanoTime(); // (8)
    result = func.apply(n); // (9)
    double duration = (System.nanoTime() - start)/1_000_000; // (10) ms.
    // ...
}
```

---

#### Example 16.14 *Benchmarking*

```
import java.util.function.LongFunction;
import java.util.stream.LongStream;
/*
 * Benchmark the execution time to sum numbers from 1 to n values
 * using streams.
 */
public final class StreamBenchmarks {

    public static long seqSumRangeClosed(long n) { // (1)
        return LongStream.rangeClosed(1L, n).sum();
    }

    public static long paraSumRangeClosed(long n) { // (2)
        return LongStream.rangeClosed(1L, n).parallel().sum();
    }

    public static long seqSumIterate(long n) { // (3)
        return LongStream.iterate(1L, i -> i + 1).limit(n).sum();
    }

    public static long paraSumIterate(long n) { // (5)
        return LongStream.iterate(1L, i -> i + 1).limit(n).parallel().sum();
    }

    public static long iterSumLoop(long n) { // (5)
        long result = 0;
        for (long i = 1L; i <= n; i++) {
            result += i;
        }
        return result;
    }
}
```

```

/*
 * Applies the function parameter func, passing n as parameter.
 * Returns the average time (ms.) to execute the function 100 times.
 */
public static <R> double measurePerf(LongFunction<R> func, long n) { // (6)
    int numOfExecutions = 100;
    double totTime = 0.0;
    R result = null;
    for (int i = 0; i < numOfExecutions; i++) { // (7)
        double start = System.nanoTime(); // (8)
        result = func.apply(n); // (9)
        double duration = (System.nanoTime() - start)/1_000_000; // (10)
        totTime += duration; // (11)
    }
    double avgTime = totTime/numOfExecutions; // (12)
    return avgTime;
}

/*
 * Executes the functions in the varargs parameter funcs
 * for different stream sizes.
 */
public static <R> void xqtFunctions(LongFunction<R>... funcs) { // (13)
    long[] sizes = {1_000L, 10_000L, 100_000L, 1_000_000L}; // (14)

    // For each stream size ...
    for (int i = 0; i < sizes.length; ++i) { // (15)
        System.out.printf("%7d", sizes[i]);
        // ... execute the functions passed in the varargs parameter funcs.
        for (int j = 0; j < funcs.length; ++j) { // (16)
            System.out.printf("%10.5f", measurePerf(funcs[j], sizes[i]));
        }
        System.out.println();
    }
}

public static void main(String[] args) { // (17)

    System.out.println("Streams created with the rangeClosed() method:");// (18)
    System.out.println(" Size Sequential Parallel");
    xqtFunctions(StreamBenchmarks::seqSumRangeClosed,
        StreamBenchmarks::paraSumRangeClosed);

    System.out.println("Streams created with the iterate() method:"); // (19)
    System.out.println(" Size Sequential Parallel");
    xqtFunctions(StreamBenchmarks::seqSumIterate,
        StreamBenchmarks::paraSumIterate);

    System.out.println("Iterative solution with an explicit loop:"); // (20)
    System.out.println(" Size Iterative");
    xqtFunctions(StreamBenchmarks::iterSumLoop);
}
}

```

Possible output from the program:

Streams created with the `rangeClosed()` method:

Size	Sequential	Parallel
1000	0.05681	0.11031
10000	0.06698	0.13979
100000	0.71274	0.52627
1000000	7.02237	4.37249

Streams created with the `iterate()` method:

Size	Sequential	Parallel
1000	0.08645	0.34696
10000	0.35687	1.27861
100000	3.24083	11.38709
1000000	29.92285	117.87909

Iterative solution with an explicit loop:

Size	Iterative
1000	0.00586
10000	0.02330
100000	0.22352
1000000	2.49677

### Side Effects

Efficient execution of parallel streams that produces the desired results requires the stream operations (and their behavioral parameters) to avoid certain side effects.

- *Non-interfering behaviors*

The behavioral parameters of stream operations should be non-interfering (p. 909)—both for sequential and parallel streams. Unless the stream data source is concurrent, the stream operations should not modify it during the execution of the stream. See building streams from collections (p. 897).

- *Stateless behaviors*

The behavioral parameters of stream operations should be stateless (p. 909)—both for sequential and parallel streams. A behavioral parameter implemented as a lambda expression should not depend on any state that might change during the execution of the stream pipeline. The results from a stateful behavioral parameter can be nondeterministic or even incorrect. For a stateless behavioral parameter, the results are always the same.

*Shared state* that is accessed by the behavior parameters of stream operations in a pipeline is not a good idea. Executing the pipeline in parallel can lead to *race conditions* in accessing the global state, and using synchronization code to provide thread-safety may defeat the purpose of parallelization. Using the three-argument `reduce()` or `collect()` method can be a better solution to encapsulate shared state.

The intermediate operations `distinct()`, `skip()`, `limit()`, and `sorted()` are stateful (p. 915, p. 915, p. 917, p. 929). See also Table 16.3, p. 938. They can carry extra

performance overhead when executed in a parallel stream, as such an operation can entail multiple passes over the data and may require significant data buffering.

### *Ordering*

An ordered stream (p. 891) processed by operations that preserve the encounter order will produce the same results, regardless of whether it is executed sequentially or in parallel. However, repeated execution of an unordered stream—sequential or parallel—can produce different results.

Preserving the encounter order of elements in an ordered parallel stream can incur a performance penalty. The performance of an ordered parallel stream can be improved if the ordering constraint is removed by calling the `unordered()` intermediate operation on the stream (p. 932).

The three stateful intermediate operations `distinct()`, `skip()`, and `limit()` can improve performance in a parallel stream that is unordered, as compared to one that is ordered (p. 915, p. 915, p. 917). The `distinct()` operation need only buffer *any* occurrence of a duplicate value in the case of an unordered parallel stream, rather than the *first* occurrence. The `skip()` operation can skip *any* *n* elements in the case of an unordered parallel stream, not necessarily the *first* *n* elements. The `limit()` operation can truncate the stream after *any* *n* elements in the case of an unordered parallel stream, and not necessarily after the *first* *n* elements.

The terminal operation `findAny()` is intentionally nondeterministic, and can return *any* element in the stream (p. 952). It is specially suited for parallel streams.

The `forEach()` terminal operation ignores the encounter order, but the `forEachOrdered()` terminal operation preserves the order (p. 948). The `sorted()` stateful intermediate operation, on the other hand, enforces a specific encounter order, regardless of whether it executed in a parallel pipeline (p. 929).

### *Autoboxing and Unboxing of Numeric Values*

As the Stream API allows both object and numeric streams, and provides support for conversion between them (p. 934), choosing a numeric stream when possible can offset the overhead of autoboxing and unboxing in object streams.

As we have seen, in order to take full advantage of parallel execution, composition of a stream pipeline must follow certain rules to facilitate parallelization. In summary, the benefits of using parallel streams are best achieved when:

- The stream data source is of a sufficient size and the stream is easily splittable into substreams.
- The stream operations have no adverse side effects and are computation-intensive enough to warrant parallelization.



## Review Questions

**16.1** Given the following code:

```
import java.util.*;

public class RQ1 {
    public static void main(String[] args) {
        List<String> values = Arrays.asList("X", "XXX", "XX", "XXXX");
        int value = values.stream()
            .mapToInt(v -> v.length())
            .filter(v -> v != 4)
            .reduce(1, (x, y) -> x * y);
        System.out.println(value);
    }
}
```

What is the result?

Select the one correct answer.

- (a) 4
  - (b) 6
  - (c) 24
  - (d) The program will throw an exception at runtime.
- 16.2** Which statement is true about the Stream methods?
- (a) The `filter()` method discards elements from the stream that match the given Predicate.
  - (b) The `findFirst()` method always returns the first element in the stream.
  - (c) The `reduce()` method removes elements from the stream that match the given Predicate.
  - (d) The `sorted()` method sorts the elements in a stream according to their natural order, or according to a given Comparator.

**16.3** Given the following code:

```
import java.util.stream.*;

public class RQ3 {
    public static void main(String[] args) {
        IntStream values = IntStream.range(0, 5);
        // (1) INSERT CODE HERE
        System.out.println(sum);
    }
}
```

Which of the following statements when inserted independently at (1) will result in a compile-time error?

Select the two correct answers.

- (a) `int sum = values.reduce(0, (x, y) -> x + y);`
- (b) `int sum = values.parallel().reduce(0, (x, y) -> x + y);`

- (c) `int sum = values.reduce((x, y) -> x + y).orElse(0);`
- (d) `int sum = values.reduce(0, (x, y) -> x + y).orElse(0);`
- (e) `int sum = values.parallel().reduce((x, y) -> x + sum).orElse(0);`
- (f) `int sum = values.sum();`

**16.4** Given the following code:

```
import java.util.stream.*;

public class RQ4 {
    public static void main(String[] args) {
        IntStream values = IntStream.range(0, 5);
        // (1) INSERT CODE HERE
        System.out.println(value);
    }
}
```

Which of the following statements, when inserted independently at (1), will result in the value 4 being printed?

Select the two correct answers.

- (a) `int value = values.reduce(0, (x, y) -> x + 1);`
- (b) `int value = values.reduce((x, y) -> x + 1).orElse(0);`
- (c) `int value = values.reduce(0, (x, y) -> y + 1);`
- (d) `int value = values.reduce(0, (x, y) -> y);`
- (e) `int value = values.reduce(1, (x, y) -> y + 1);`
- (f) `long value = values.count();`

**16.5** Given the following code:

```
import java.util.*;
import java.util.stream.*;

public class RQ5 {
    public static void main(String[] args) {
        List<String> values = List.of("AA", "BBB", "C", "DD", "EEE");
        Map<Integer, List<String>> map = null;
        // (1) INSERT CODE HERE
        map.forEach((i, s) -> System.out.println(i + " " + s));
    }
}
```

Which statement when inserted independently at (1) will result in the output 1 [C]?

Select the one correct answer.

- (a) `map = values.stream()
 .collect(Collectors.groupingBy(s -> s.length(),
 Collectors.filtering(s -> !s.contains("C"),
 Collectors.toList())));`
- (b) `map = values.stream()
 .collect(Collectors.groupingBy(s -> s.length(),
 Collectors.filtering(s -> s.contains("C"),
 Collectors.toList())));`

- (c) `map = values.stream()  
     .filter(s -> !s.contains("C"))  
     .collect(Collectors.groupingBy(s -> s.length(),  
                                   Collectors.toList()));`
- (d) `map = values.stream()  
     .filter(s -> s.contains("C"))  
     .collect(Collectors.groupingBy(s -> s.length(),  
                                   Collectors.toList()));`

**16.6** Given the following code:

```
import java.util.stream.*;

public class RQ7 {
    public static void main(String[] args) {
        Stream<String> values = Stream.generate(() -> "A");
        boolean value = values.peek(v -> System.out.print("B"))
                               .takeWhile(v -> !v.equals("A"))
                               .peek(v -> System.out.print("C"))
                               .anyMatch(v -> v.equals("A"));

        System.out.println(value);
    }
}
```

What is the result?

Select the one correct answer.

- (a) Btrue
- (b) Ctrue
- (c) BCtrue
- (d) Bfalse
- (e) Cfalse
- (f) BCfalse

**16.7** Given the following code:

```
import java.util.stream.*;

public class RQ9 {
    public static void main(String[] args) {
        IntStream.range('a', 'e')
                 .mapToObj(i -> String.valueOf((char) i).toUpperCase())
                 .filter(s -> "AEIOU".contains(s))
                 .forEach(s -> System.out.print(s));
    }
}
```

What is the result?

Select the one correct answer.

- (a) A
- (b) AE
- (c) BCD
- (d) The program will fail to compile.

**16.8** Given the following code:

```
import java.util.stream.*;

public class RQ10 {
    public static void main(String[] args) {
        IntStream.range(0, 5)
            .filter(i -> i % 2 != 0)
            .forEach(i -> System.out.println(i));
    }
}
```

Which of the following statements will produce the same result as the program?  
Select the two correct answers.

- (a) `IntStream.rangeClosed(0, 5)`  
`.filter(i -> i % 2 != 0)`  
`.forEach(i -> System.out.println(i));`
- (b) `IntStream.range(0, 10)`  
`.takeWhile(i -> i < 5)`  
`.filter(i -> i % 2 != 0)`  
`.forEach(i -> System.out.println(i));`
- (c) `IntStream.range(0, 10)`  
`.limit(5)`  
`.filter(i -> i % 2 != 0)`  
`.forEach(i -> System.out.println(i));`
- (d) `IntStream.generate(() -> {int x = 0; return x++;})`  
`.takeWhile(i -> i < 4)`  
`.filter(i -> i % 2 != 0)`  
`.forEach(i -> System.out.println(i));`
- (e) `var x = 0;`  
`IntStream.generate(() -> return x++)`  
`.limit(5)`  
`.filter(i -> i % 2 != 0)`  
`.forEach(i -> System.out.println(i));`

**16.9** Given the following code:

```
import java.util.function.*;
import java.util.stream.*;

public class RQ11 {
    public static void main(String[] args) {
        Stream<String> abc = Stream.of("A", "B", "C");
        Stream<String> xyz = Stream.of("X", "Y", "Z");
        String value = Stream.concat(xyz, abc).reduce((a, b) -> b + a).get();
        System.out.println(value);
    }
}
```

What is the result?

Select the one correct answer.

- (a) ABCXYZ
- (b) XYZABC

- (c) ZYXCBA
- (d) CBAZYX

**16.10** Which statement produces a different result from the other statements? Select the one correct answer.

- (a) `Stream.of("A", "B", "C", "D", "E")`  
`.filter(s -> s.compareTo("B") < 0)`  
`.collect(Collectors.groupingBy(s -> "AEIOU".contains(s)))`  
`.forEach((x, y) -> System.out.println(x + " " + y));`
- (b) `Stream.of("A", "B", "C", "D", "E")`  
`.filter(s -> s.compareTo("B") < 0)`  
`.collect(Collectors.partitioningBy(s -> "AEIOU".contains(s)))`  
`.forEach((x, y) -> System.out.println(x + " " + y));`
- (c) `Stream.of("A", "B", "C", "D", "E")`  
`.collect(Collectors.groupingBy(s -> "AEIOU".contains(s),`  
`Collectors.filtering(s -> s.compareTo("B") < 0,`  
`Collectors.toList()))`  
`.forEach((x, y) -> System.out.println(x + " " + y));`
- (d) `Stream.of("A", "B", "C", "D", "E")`  
`.collect(Collectors.partitioningBy(s -> "AEIOU".contains(s),`  
`Collectors.filtering(s -> s.compareTo("B") < 0,`  
`Collectors.toList()))`  
`.forEach((x, y) -> System.out.println(x + " " + y));`

**16.11** Given the following code:

```
import java.util.stream.*;

public class RQ13 {
    public static void main(String[] args) {
        Stream<String> strings = Stream.of("i", "am", "ok").parallel();
        IntStream chars = strings.flatMapToInt(line -> line.chars()).sorted();
        chars.forEach(c -> System.out.print((char)c));
    }
}
```

What is the result?

Select the one correct answer.

- (a) iamok
  - (b) aikmo
  - (c) amiok
  - (d) The result from running the program is unpredictable.
  - (e) The program will throw an exception at runtime.
- 16.12** Which of the following statements are true about the Stream methods? Select the two correct answers.
- (a) The `filter()` method accepts a Function.
  - (b) The `peek()` method accepts a Function.
  - (c) The `peek()` method accepts a Consumer.

- (d) The `forEach()` method accepts a `Consumer`.
- (e) The `map()` method accepts a `Predicate`.
- (f) The `max()` method accepts a `Predicate`.
- (g) The `findAny()` method accepts a `Predicate`.

**16.13** Which `Stream` methods are terminal operations?  
Select the two correct answers.

- (a) `peek()`
- (b) `forEach()`
- (c) `map()`
- (d) `filter()`
- (e) `sorted()`
- (f) `min()`

**16.14** Which `Stream` methods have short-circuit execution?  
Select the two correct answers.

- (a) `collect()`
- (b) `limit()`
- (c) `flatMap()`
- (d) `anyMatch()`
- (e) `reduce()`
- (f) `sum()`

**16.15** Given the following code:

```
import java.util.stream.*;

public class RQ17 {
    public static void main(String[] args) {
        Stream<String> values = Stream.of("is", "this", "", null, "ok", "?");
        // (1) INSERT CODE HERE
        System.out.println(c);
    }
}
```

Which statement inserted independently at (1) produces the output 6?  
Select the one correct answer.

- (a) `long c = values.count();`
- (b) `long c = values.collect(Collectors.counting());`
- (c) `int c = values.mapToInt(v -> 1).reduce(0, (x, y) -> x + 1);`
- (d) `long c = values.collect(Collectors.reducing(0L, v -> 1L, Long::sum));`
- (e) `int c = values.mapToInt(v -> 1).sum();`
- (f) Insert any of the above.

**16.16** Which code produces identical results?  
Select the two correct answers.

- (a) `Set<String> set1 = Stream.of("XX", "XXXX", "", null, "XX", "X")
 .filter(v -> v != null)
 .collect(Collectors.toSet());`

```
set1.stream()
    .mapToInt(v -> v.length())
    .sorted()
    .forEach(v -> System.out.print(v));
```

(b) `Set<Integer> set2 = Stream.of("XX", "XXXX", "", null, "XX", "X")`  
`.map(v -> (v == null) ? 0 : v.length())`  
`.filter(v -> v != 0)`  
`.collect(Collectors.toSet());`

```
set2.stream()
    .sorted()
    .forEach(v -> System.out.print(v));
```

(c) `List<Integer> list1 = Stream.of("XX", "XXXX", "", null, "XX", "X")`  
`.map(v -> (v == null) ? 0 : v.length())`  
`.filter(v -> v != 0)`  
`.toList();`

```
list1.stream()
    .sorted()
    .forEach(v -> System.out.print(v));
```

(d) `List<Integer> list2 = Stream.of("XX", "XXXX", "", null, "XX", "X")`  
`.map(v -> (v == null) ? 0 : v.length())`  
`.filter(v -> v != 0)`  
`.distinct()`  
`.toList();`

```
list2.stream()
    .sorted()
    .forEach(v -> System.out.print(v));
```

*This page intentionally left blank*

# Index



## Symbols

- 63  
-- 69  
^ 78, 82  
^= 79  
\_ 36, 1103  
, 1125  
; 101  
: 157, 340, 1100, 1287  
:: 724  
! 78, 1101  
!= 75, 426  
? 580, 582, 588  
? extends T 581, 589  
? super T 582, 589  
?: 90  
. 9, 127, 128, 326, 338, 1125, 1288  
.. 1289  
... 136, 141  
' 37, 38  
" 39  
"" 458  
[] 92, 118, 120  
{ } 101, 119, 354  
@ 1558, 1563  
@Deprecated 1580  
@Documented 1574  
@FunctionalInterface 1579  
@Inherited 1574  
@Override 198, 224, 241, 618, 1578  
@param 100, 112  
@Repeatable 1575  
@Retention 1567  
@return 184  
@SafeVarargs 632, 650, 659, 863, 1585  
@SuppressWarnings 1582  
@SuppressWarnings("unchecked") 627, 632  
@Target 1569  
@throws 390  
\* 58, 61, 330  
\*= 66  
/ 61, 62, 1287  
/\* and \*/ 40  
/\*\* and \*/ 41  
// 40  
/= 66  
\ 38, 1101, 1287  
& 78, 82  
&& 80  
&= 79  
# 1101, 1125, 1146  
% 61, 62  
%= 66  
+ 63, 67  
+ concatenation 448  
++ 69  
+= 66  
< 74, 1146  
<< 86, 87  
<= 90  
<= 74  
<> 567, 570, 646  
-= 66  
= 54, 1100  
== 75, 426, 433  
-> 92, 160, 679  
> 74  
>= 64  
>> 86, 88  
>>= 90  
>>> 86, 89  
>>>= 90  
| 78, 82, 399, 1146

|= 79  
 || 80  
 ~ 82

## Numerics

0 1125

## A

ability interfaces  
   *see* marker interfaces  
 abrupt method completion 367  
 absolute adjusters 1035  
 absolute path 1288  
 absolute values 478  
 abstract  
   interfaces 239  
   methods 240, 675  
 abstract 239, 240, 675  
 abstract classes 218  
   abstract methods 224  
   considered incomplete 222  
   declaring 219  
   extending 220  
   incomplete 219  
   polymorphic references 220  
   UML class diagram 219  
 abstract data type 565  
 abstract keyword 219, 224  
 abstract method declarations 675  
   in enum types 294  
   in interfaces 240  
 abstract methods 224  
   declaration 224  
   implementing 224  
   method header 224  
   overloading 225  
   overriding 224  
 abstraction 5, 13  
 access modifiers 99, 100, 109, 345  
 access order 841, 842  
 accessibility 10, 19, 352, 1173  
   access modifiers 345  
   anonymous classes 525  
   local classes 512, 513  
   members 347, 352  
   modifiers 493  
   non-static member classes 504  
   static member classes 498  
   top-level types 345  
   UML notation 348  
 accessibility for class members  
   package 347, 351  
   private 351  
   protected 350  
   public 350  
 accessibility for top-level types  
   package 345  
   public 345  
 accessibility rules 1174  
 accessor methods *see* get methods  
 accounting currency formatting 1118  
 acquisition order 1461  
 activation frame 534  
   *see* method execution 365  
 actual parameter 128  
 actual parameter list 127, 265  
 actual type parameters 569  
 adapter 992, 993, 994  
 adding to class 191  
 additive operators 63  
 aggregate module 1168  
 aggregate of values 300  
 aggregate operations 881  
 aggregation 13, 16, 194  
   hierarchy 195  
 algorithms 783  
 aliases 55, 76, 130  
   *see also* symbolic link  
 all-or-none model 1474  
 ambiguous call 202, 266  
 analyzing program code 1620  
*and* operator 83  
 AND-ing 86  
 AnnotatedElement 1588  
 Annotation 1557  
 annotation 1557  
   @ 1563  
   applying annotation type 1563  
   declaration annotation 1571  
   default values 1565  
   marker annotation 1563  
   metadata 1557  
   normal annotation 1563  
   processing 1587  
   single-element annotation 1563  
   type annotations 1572  
 annotation type 1557, 1558  
   @ 1558  
   access modifier 1559  
   annotation type element 1560  
   containing annotation type 1576  
   declaring 1558

- marker annotation type 1560
- member declarations 1559
- multi-element annotation type 1561
- repeatable annotation type 1576
- simple name 1559
- single-element annotation type 1560
- syntax 1558
- annotation type declaration
  - see* annotation type
- annotation type element 1560
  - access modifier 1561
  - attributes 1560
  - default values 1562
  - element type 1561
  - method declaration 1560
  - optional annotation element 1562
  - required annotation element 1562
  - value() element 1560, 1561
- annotations 239
  - @Deprecated 1580
  - @FunctionalInterface 675, 1579
  - @Override 198, 618
  - @SafeVarargs 632
  - @SuppressWarnings("unchecked") 627
- anonymous arrays 122, 126
  - [] 122
- anonymous classes 494, 521, 571, 690
  - accessing members in enclosing context 525
  - compared with lambda expressions 692
  - declaration 521
  - event listeners 521
  - extending an existing class 521
  - hiding members in enclosing context 525
  - implementing an interface 523
  - in non-static context 521
  - in static context 521
  - instantiation 524
  - objects on the fly 521
  - referencing instances 524
  - shadowing local variables 525, 692
  - using qualified this 525, 693
- anti-symmetric 762
- API (application programming interface) 2
- apostrophe 38
- application 19
- architecture neutral 4
- argument
  - see* actual parameter
- arguments to main method 141
- arithmetic compound assignment
  - operators 66
- ArithmeticException 372
- arity 136, 695
- array constructor references 725, 731
- array creation expression 119, 122, 143
- array element initializer 1564
- array initializer 119, 120, 122, 126, 146
- array list 802
- array mismatch 871
- array operations 798
- array store check 260, 627
- array store check report
  - false positive 631
- array types 579
  - see* arrays
- ArrayBlockingQueue 1496
- ArrayDeque 823
- ArrayIndexOutOfBoundsException 121, 372, 397
- ArrayList 450, 802
  - add collection 651
  - add element 647, 651
  - autoboxing 654
  - capacity 646
  - clear list 652
  - comparison with arrays 662
  - constructing 646
  - constructors 648
  - converting to array 658
  - element search 656
  - filtering 688
  - import 646
  - inheritance hierarchy 645
  - initial capacity 647
  - insertion order 644
  - list of lists 648
  - membership test 655
  - modifying 651
  - nested lists 648
  - object value equality 656
  - open range-view operations 645
  - ordered 644
  - position-based access 645
  - positional index 655
  - positional order 644
  - positional retrieve 655
  - querying 655
  - references 646
  - remove collection 652
  - remove element 652
  - replace element 652
  - size 646, 655
  - subtype relationship 648

- text representation 647
- trim to size 653
- type-safety 646, 648
- unchecked conversion warning 646
- zero-based index 644
- Arrays 868
  - creating streams 898
- arrays 117, 425
  - [] 118, 119, 120
  - { } 119
  - anonymous 122, 126
  - array creation expression 119
  - array initialize list 119, 122
  - array initializer 120, 126
  - array name 118
  - array size 119
  - array store check 260, 627
  - ArrayIndexOutOfBoundsException 121
  - binary search 866
  - bounds 121
  - collection operations 798
  - common prefix 870, 871
  - comparators 867
  - comparing 868
  - construction 119
  - converting to list 659
  - declarations 118
  - default initialization 118, 119
  - dynamic 645, 802
  - element access expression 120
  - element default value 260
  - element type 118
  - elements 117, 120
  - equality 868
  - filling 873
  - index 117
  - index expression 120
  - initialization 119, 124
  - initializing 873
  - iterating over 176
  - iterating over an array 121
  - length 118
  - lexicographic comparison 869
  - mismatch 871
  - multidimensional 122, 124
  - objects 425
  - proper prefix 871, 872
  - ragged 124
  - reference 118, 121, 260
  - searching 866
  - sequential streams 898
  - sorting 865
  - subtype covariance 259
  - text representation 873
  - using 120
- arrays of arrays 118, 124
- multidimensional 124
- ArrayStoreException 260, 372, 648, 658, 798, 799
- array-valued single-element annotation 1566
- arrow notation (->) 160
- arrow operator (->) 92
- ascending-order key set 847
- ASCII 37, 42
- AssertionError 374
- assessor methods
  - see* get methods
- assignable 47, 264
- assignment
  - widening reference 584
- assignment compatible 47, 264, 646
- assignment conversions 47
- assignment operator 8
- assignments
  - arithmetic compound operators 66
  - bitwise 85
  - cascading 55
  - compound operators 79, 85
  - expression statement 55
  - extended operator 90
  - implicit narrowing 56
  - multiple 55
  - numeric conversions 56
  - operator 50, 54
  - primitive values 54
  - references 55
  - widening reference conversions 196
- association 16
  - aggregation 194
  - composition 195
  - realization 244
- associativity 51, 52
- asynchronous 1370
- atomic actions 1455, 1456, 1469
  - CAS: Compare and Set 1457
- atomic classes 1457
- atomic move 1308
- atomic operation 1305, 1339
- atomic variables 1456
  - atomic actions 1456
- AtomicInteger 1457, 1458

- atomicity 1455
- AtomicMoveNotSupportedException 1313
- at-sign 1558
- attributes 1512
- attributes *see* properties
- AttributeView interface 1334
- autoboxing 430
  - for(·) statement 177
  - see* boxing conversion
- AutoCloseable 409, 412, 536, 1234, 1244, 1519
  - streams 889
- automatic garbage collection 533
- automatic memory management 533
- automatic module 1206, 1207
  - naming 1207
- Automatic Resource Management (ARM) 411
- automatic variables *see* local variables
- auxiliary methods 252
- available time zones 1073

## B

- backing collection 856
- backslash 38
- backspace 38
- backtrace 405
- backward compatibility 565
- bags 804
- balanced tree 788, 789, 812
- barrier
  - see* cyclic barrier
- barrier action 1471
- barrier point 1471
- base 33, 431, 435
- base case 1448
- base class 191
- base name 1102
- basic collection operations 790
- basic for statement 174
- basic map operations 831
- basic statement 1523
- BasicFileAttributes interface 1330
- BasicFileAttributeView interface 1334
- Before Current Era (BCE) 1029
- behavior 675
- behavior parameterization 675
- behavioral parameters 909
- BiConsumer functional interface 709
  - primitive type specializations 711
- BiFunction interface 717, 718
  - composing two-arity functions 718
  - primitive type specializations 719
- big numbers 484, 485
- BigDecimal
  - customizing formatting 1125
  - formatting 1117
  - parsing 1119
  - rounding 1122
- BigDecimal class 485
- BigInteger 485
- binary
  - numeric promotion 50
  - operators 50
  - shift operators 86
- binary files 1238, 1239, 1240
- binary operator 52
- binary representation
  - see* binary values
- binary search
  - arrays 866
  - lists 861
- binary values 1238, 1239, 1240
- BinaryOperator interface 721, 722
  - composing binary operators 721
  - primitive type specializations 722
- bind parameter 1526
- bind variable 1526
- binding of method call 205, 227
- BiPredicate functional interface 703
- bit mask 84
- bit masking 86
- bit patterns 54
- bit shifting 86
- bitwise
  - and* operator 83
  - assignment 85
  - complement 83
  - compound assignment 85
  - operators 82
  - or* operator 83
  - shifting 86
  - xor* 83
- bitwise AND
  - & 82
- bitwise complement
  - ~ 82
- bitwise exclusive OR
  - ^ 82
- bitwise OR
  - | 82

- bitwise XOR
  - ^ 82
- blank final local variables 232
- blank final variables 135, 229, 231
- block 161, 166, 169
- block scope 169, 355, 684
- BLOCKED state 1382
- blocking collection 1483
- blocking operations 1234, 1405
- blocking queues 1495
- BlockingDeque 1496, 1498
- BlockingQueue 1496, 1497
- blocks 101, 354
  - scope 354
  - try 377
- boilerplate code 409, 691
- Boolean
  - condition 152
- Boolean wrapper class 437
- booleans 41, 43
  - casting 49
  - expressions 74
  - literals 37
- BooleanSupplier functional interface 699
- bounded collection 1483
- bounded instance method references 727
- bounded type parameters
  - constraint 591
- bounded wildcard 581
- bounds
  - constraint 591
- boxing conversions 45
- break statement 157, 180
- bridge method 615, 764
- BS *see* backspace
- buckets 753
- buffered readers 1251
  - creating streams 1252
  - end of stream 1252
  - reading text lines 1252
- buffered writers 1250
- BufferedReader class 1242, 1315
  - streams 902
- BufferedWriter class 1243, 1316
- buffering streams 1238
- building abstractions 13
- built-in functional interfaces
  - java.util.function package 695
  - BiConsumer 709
  - BiFunction 717, 718
  - BinaryOperator 721, 722
  - BiPredicate 703
  - BooleanSupplier 699
  - Consumer 709
  - DoubleBinaryOperator 722
  - DoubleConsumer 709
  - DoubleFunction 712
  - DoublePredicate 703
  - DoubleSupplier 699
  - DoubleToIntFunction 713
  - DoubleToLongFunction 713
  - DoubleUnaryOperator 720
  - Function 712
  - IntBinaryOperator 722
  - IntConsumer 709
  - IntFunction 712
  - IntPredicate 703
  - IntSupplier 699
  - IntToDoubleFunction 713
  - IntToLongFunction 712
  - IntUnaryOperator 720
  - LongBinaryOperator 722
  - LongConsumer 709
  - LongFunction 712
  - LongPredicate 703
  - LongSupplier 699
  - LongToDoubleFunction 713
  - LongToIntFunction 713
  - LongUnaryOperator 720
  - ObjDoubleConsumer 709
  - ObjIntConsumer 709
  - ObjLongConsumer 709
  - Predicate 703
  - Supplier 699
  - ToDoubleBiFunction 718
  - ToDoubleFunction 712
  - ToIntBiFunction 718
  - ToIntFunction 712
  - ToLongBiFunction 718
  - ToLongFunction 712
  - UnaryOperator 720
- built-in synchronization
  - see* intrinsic locking
- bulk map operations 839
- bulk operations on collections 790
- bulk task execution 1439
- bundling resources 1102
- byte 32, 42
- byte streams 1233
  - reading bytes 1234
  - writing bytes 1234
- bytecode 4, 21
- bytecode verification 1611

**C**

- C++ 4
- cached thread pool 1426
- caching 358
- call by reference 131
- call by value 131
- call chaining 128
- call signature 266
- call stack
  - see* JVM stack 365
- callable statement 1530
  - stored functions 1530
  - stored procedures 1530
- callee 127
- caller 127, 184
- candidate bundle name 1108
- capacity 646
- capture conversion 606
- capture of 605
- carriage return 38, 39
- CAS: Compare and Set 1457
- cascaded grouping 987
- cascaded partitioning 991
- cascading assignments 55
- cascading if-else statements 155
- case constants 156, 157, 160
- case labels 156, 157, 160
- case sensitivity 30
- case-insensitive ordering 771
- cast 631
  - references 281
- cast operator 45, 48, 50, 57, 66, 67, 76, 143, 269
- casting 47, 48
  - see also* conversions
- catch clause 377
  - multi- 397
  - rethrowing exception 401
  - uni- 376
- catching exceptions 365
- catch-or-declare rule 388
- catch-or-specify requirement 388
- cause 405
- ceiling entry 846
- ceiling key 846
- chaining 555
  - constructors 213, 555
  - new operator 507
- chaining exceptions 405
- changing list contents 863
- char 42
- character case 448
- character encoding 1241
  - see also* character sets
- character sequences
  - see* strings *and* string builders
- character set
  - ASCII 37, 42, 1241
  - ISO Latin-1 37, 42
  - Unicode 37, 42, 1241
- character streams 1233, 1241
  - character encoding 1241
  - end of stream 1243
  - reading characters 1243
  - writing characters 1244
- Character wrapper class 437
- character-based device
  - console 1256
- characters 42
  - literals 37
  - searching for 451, 466
- CharSequence interface 444, 450, 901
- charsets
  - see also* character sets
- checked exceptions 374
- child class 191
- child threads 1373, 1378, 1454
- choice format 1146, 1148
  - limits 1146
  - subformat pattern 1146
  - syntax 1146
- ChoiceFormat 1115, 1148
- choosing between String and StringBuilder class 464
- ChronoField enum type 1036, 1046, 1079
  - list of enum constants 1046
- ChronoUnit enum type 1040, 1041, 1044, 1081, 1082
  - list of enum constants 1044
- Class 1588
- Class class 426
- class design 1611
- class file 21
- class file name 497
- class hierarchy
  - see* inheritance hierarchy
- class initialization 230
- class literals 636
- class lock 1380, 1391, 1392
- class method 13
- class path 337, 1206
  - absolute pathnames 340
  - entries order 340
  - entry-separator character 340
  - fully qualified package name 340

- path-separator character 340
- relative pathnames 340
- search in a named package 339
- searching for classes 337
- whitespace 340
- class search path
  - see* class path
- class variable 13
- class versioning 1273
- ClassCastException 271, 372, 1105
- classes
  - abstract 218
  - accessibility 345
  - adding to 191
  - anonymous 492, 494, 521
  - base 191
  - body 99
  - child 191
  - class file name 497
  - concrete 218, 220, 226
  - constructors 109
  - contract 218, 242
  - declaration context 493
  - declarations 99, 325
  - definitions 6, 8
  - derived 191
  - diagram 11, 12
  - encapsulation 324
  - extending 191, 225
  - final 225, 358
  - final vs. abstract 226
  - fully qualified name 337
  - fully qualified package name 328
  - generalized 194
  - grouping 326
  - header 99
  - immutable 356
  - implementation 218
  - implementing interfaces 240
  - incomplete 219
  - initialization 230, 557
  - inner *see* inner classes
  - instances 8
  - local 492, 494, 512
  - members 9
  - methods 115
  - name 326
  - nested *see* nested classes
  - non-static member 492, 494, 501
  - Object 425
  - on-the-fly 492
  - parent 191
  - partial implementation 218
  - records 299
  - runtime 426
  - scope 352
  - sealed 312
  - searching for 337
  - specialized 194
  - static fields 113
  - static members 113, 494, 495
  - static methods 115
  - static variables 113
  - subclass 13, 191
  - superclass 13, 191
  - top-level 491, 494
  - utility 115
  - wrappers 425, 429
- class-level lock
  - see* class lock
- ClassLoader class 425
- ClassNotFoundException 371
- CLASSPATH environment variable
  - see* class path
- classpath option
  - see* class path
- clauses
  - catch 377
  - extends 191
  - finally 377, 382
  - implements 240
  - permits 312
  - throws 388
- cleaning up 536
- clean-up code 382, 1378
- client 10
- Cloneable interface 427
- CloneNotSupportedException 427
- cloning objects 427
- Closeable 412
- closing an I/O stream 1234, 1239, 1240
- closing delimiter 458
- code corruption 1606
- code injection 1605
- code migration
  - bottom-up strategy 1209
  - top-down strategy 1210
- code optimizations 229
- Code Ranch ([www.coderanch.com](http://www.coderanch.com)) 1616
- code reuse 3, 191, 195
- codebase 1608
- Collection 644
- Collection interface 784
- collections 586, 644, 783, 784

- adding elements 863
- array operations 798
- ArrayList 802
- as single entity 644
- basic operations 790, 831
- bulk operations 790, 839
- Cloneable 787
- compare with streams 888
- data structures 788
- duplicate elements 801, 804
- elements 644, 783
- filtering 796
- first element 810, 811, 1486
- HashSet 806
- implementations 783, 785, 788
- interfaces 783
- iteration 791, 792
- iteration order 795
- iterators 793
- last element 810, 811, 1486
- LinkedHashSet 806
- LinkedList 802
- List 802
- lists 801
- map views 839
- maximum 862
- minimum 862
- NavigableSet 810
- optional operations 790
- ordered 644, 789, 801
- parallel streams 897
- range-views 801, 810, 811, 1486
- retaining elements 791
- sequential streams 897
- Serializable 787
- Set 804
- shuffle elements 860
- sorted 789
- SortedSet 810, 811
- sorting 644
- subset 810, 811, 1486
- text representation 795
- thread-safety 802, 840
- TreeSet 812, 847
- unmodifiable view 856
- unordered 789
- unsupported operations 831
- utility methods 783
- Vector 802
- views 656, 810, 811, 839, 1486
- collections framework 783
- Collections utility class 858, 861
- Collector interface 890, 978
- Collectors 890, 978
  - importing 978
- collectors 948, 964, 978
  - averaging 1000
  - classification mapping 979
  - classification maps 979
  - classifier function 979
  - collecting to a Collection 979
  - collecting to a ConcurrentMap 983
  - collecting to a List 980
  - collecting to a Map 981
    - key mapper 981
    - map supplier 982
    - merge function 981
    - value mapper 981
  - collecting to a Set 980
  - comparison with terminal operations 1008
  - composing 979
  - concurrent 983, 989
  - counting 998
  - downstream 979, 985, 986, 992, 993, 994, 997, 998
  - filtering 992
    - downstream collector 992
    - predicate 992
  - finding min/max 999
  - finishing 997
    - downstream collector 997
    - finisher function 997
  - flat mapping 994
    - downstream collector 994
    - mapper function 994
  - functional reduction 998, 1002
  - grouping 979, 985
    - classification map 985
    - classifier function 985
    - default downstream collector 985
    - downstream collector 986
    - map supplier 986
    - multilevel 987
    - to a ConcurrentMap 989
  - joining to a string 984
  - mapping 993
    - downstream collector 993
    - mapper function 993
  - partitioning 979, 989
    - classification map 990
    - default downstream collector 990
    - downstream collector 990
    - multilevel 991
    - partitioning predicate 989

- reducing 1002
  - binary operator 1002, 1003
  - identity value 1002, 1003
  - mapper function 1003
- summarizing 1001
- summary of 1005
- summing 1000
- colon notation (:) 156
- columns 1512
- command line 21, 141
- command prompt 21
- commands
  - java 21
  - javac 20
- comments 40
- common memory space 1367
- common prefix 870, 871
- communication 127
- compact canonical record constructor 304, 306
- compact form styles 1120
- compact number formatting 1120
- compact number parsing 1121
- compact strings 440
- CompactNumberFormat 1115
- Comparable interface 432, 447, 761
  - compareTo 761
- comparable objects 761
- Comparator interface
  - compare 769
  - functional interface 769
- comparators 761
  - conditional 765, 771, 773
  - reverse order 859
- comparing objects 426, 744, 769
  - see also* equality
- comparing strings 447
- comparison 74
- compilation unit 328
  - deriving permitted direct types 317
- compiler
  - Xlint:unchecked option 577
- compiler optimizations 1453
- compiling and running a service 1203
- compiling Java source code 20
- complement
  - ~ 83
- completes abruptly
  - see* exception propagation 367
- component fields 302
- component list 300
- composite object 13
  - composition 16, 195
    - see* aggregation
  - compound declaration 146
  - compound messages
    - conditional formatting 1145
    - formatting 1143
    - handling plurals 1146
  - compound mutually exclusive operations 1480
  - compound state 1382
  - compound statement 101
  - concatenation of strings 448
  - concatenation operator 67
  - concise try-with-resources 411
  - concrete class 218
  - concrete method 226
  - concrete parameterized types 580
  - concrete type 573, 580, 586
  - concurrency 1368, 1540
  - concurrent activities 1367
  - concurrent collections 1482, 1483
    - blocking 1483
    - bounded 1483
    - duplicate elements 1483
    - happens-before relationship 1483
    - non-blocking 1483
    - ordering 1484
    - unbounded 1483
  - concurrent collector 983, 989
  - concurrent maps 983, 989, 1490
    - segments 1491
  - ConcurrentHashMap 1491
  - ConcurrentLinkedDeque 1486, 1487
  - ConcurrentLinkedQueue 1486, 1487
  - ConcurrentMap 983, 989, 1491
  - ConcurrentModificationException 658, 795, 796
  - ConcurrentNavigableMap 1492
  - ConcurrentSkipListMap 1491
  - ConcurrentSkipListSet 1486
  - condition
    - Boolean 152
    - expressions 152
  - conditional 74
    - operators 90
    - statements 152
  - conditional *and* operator 80
    - pattern variable 277
  - conditional comparator 765, 771, 773
  - conditional expressions 90
    - associativity 92
    - nested 92

- precedence 91
- short-circuit evaluation 91
- side effects 91
- conditional formatting 1115, 1145
- conditional operators 80
- conditional *or* operator 80
  - pattern variable 277
- conditions 74
- configuration 5
- configuration data 1100
- congruent 762
- connecting punctuation character 30
- connection 1519
- Console 1153
- console 1256
  - formatted prompting 1257
  - printing formatted values 1257
  - reading password 1256
- Console class 1256
- const 32
- constant 546
- constant expression 47, 56, 69, 157, 229, 546, 1562
  - labels 157
- constant field values
  - case labels 162
- constant string expressions 162
- constant values 32, 229
- constant variable 56, 229, 546
- constants 254
- constant-specific class bodies
  - using anonymous classes 294
- constituent objects 13
- constituents 16
- constraints 1512
- constructing array 119
- constructor chaining 210, 213, 555
- constructor references 725, 730
- constructors 7, 109
  - access modifier 109
  - accessibility 352
  - body 109
  - canonical 302
  - chaining 210, 213
  - class name 109
  - declaration 99
  - default 110
  - header 109
  - implicit default 110
  - local declarations 109
  - no-argument 110, 210, 213
  - non-zero argument 111, 214
  - overloading 112
  - superclass constructor 110
- constructs 30
  - high-level 30
  - loops *see* iteration statements
  - synchronized statements 1391, 1392
- Consumer functional interface 709
  - composing consumers 710
  - primitive type specializations 711
- contained annotation type 1562
- containers 586
  - see also* collections
- containing annotation type 1562, 1576
- contains characters 452
- context switching 1368
- contextual keywords 31
  - non-sealed 312
  - permits 312
  - sealed 312, 315
  - yield 164
- continue statement 182
- contract 6, 195, 218, 240, 324
- control flow
  - break 157, 180
  - continue 182
  - do-while 173
  - for(;;) 174
  - for(:) 176
  - if 152
  - if-else 153
  - iteration *see* iteration statements
  - loops *see* iteration statements
  - return 184
  - statements 101
  - switch expression 164
  - switch statement 155
  - throw 386
  - transfer statements 179
  - while 172
  - yield 164
- control transfer 179
- conversion categories 47
- conversion contexts 47
- conversions 261
  - assignment 47
  - contexts 46
  - explicit 43
  - identity 67
  - implicit 43
  - implicit narrowing 67
  - method invocation 48
  - narrowing reference 270, 585, 605

- numeric promotions 49
  - parameters 129
  - reference casting 269
  - string concatenation 68
  - to strings 454
  - truncation 57
  - type-safe 264
  - unsafe casts 271
  - widening reference 196, 270, 584
  - converting values 431, 432, 434, 435, 437, 454
  - copy-on-write collections 1501
  - CopyOnWriteArrayList 1502
  - CopyOnWriteArraySet 1502, 1503
  - CopyOption interface 1301
  - Core Collections Interfaces 786, 1485, 1495
  - core interfaces 783
  - Core Map Interfaces 787
  - core stream interfaces 890
  - count-down latch 1474
  - CountDownLatch 1474
  - counter-controlled loops 174
  - country codes 1097
  - covariant return 197, 201, 619
  - cp option
    - see* class path
  - CPU core 1009
  - CR *see* carriage return
  - crab 176
  - CREATE statement 1514
  - creating
    - objects 92
    - threads 1370
  - creating a runtime image 1204
  - creating modular JAR 1189
  - creating records 303
  - critical section 1388
  - CRUD operations 1514
  - currency 1096, 1116
    - currency symbol 1126
    - formatting 1116, 1119
  - currency formatter 1117, 1119
  - currency symbol 30, 1126
  - current directory 1289
    - . 338, 1288
    - system property 1292
  - Current Era (CE) 1029
  - current object 106
  - currying functions 723
    - partial application 723
  - cursor 801, 1534
  - cyclic barrier 1471
    - all-or-none model 1474
    - barrier action 1471
    - barrier point 1471
  - CyclicBarrier 1471
- ## D
- d option 335
  - daemon 1377, 1402
  - dangling references 533
  - Data Definition Language (DDL) 1514
  - data encryption 1604
  - data integrity 1606
  - Data Manipulation Language (DML) 1514
  - Data Query Language (DQL) 1514
  - data source 884
  - data streams 1236, 1238
  - data structures 644, 788
  - data types *see* types
  - database capabilities 1543
  - database connection 1519
  - database cursor 1534, 1540
  - Database Management System (DBMS) 1512
  - database protocol 1520
  - database transactions 1545
  - DataFormat 1115
  - DataInputStream class 1235
  - DataOutputStream class 1236
  - Date 1024, 1078, 1141
  - date 761, 1096
    - see* temporal objects
  - date fields 1024
  - date units 1040
  - DateFormat 1141
  - date-time
    - see* temporal objects
  - date-time formatters
    - format styles 1132
    - immutability 1128
    - ISO-based default 1127, 1129
    - ISO-based predefined 1127, 1129
    - letter pattern 1134
    - parsing 1128
    - pattern letters 1134, 1135
    - pattern-based 1134
    - style-based 1127, 1131
    - thread-safety 1128
  - DateTimeException 373, 1025, 1033, 1075
  - DateTimeFormatter 1115, 1127
  - DateTimeParseException 374, 1051, 1058, 1066, 1131, 1134
  - daylight saving time (DST) 1073, 1082
    - time gap 1082
    - time overlap 1082

- time zone rules 1082, 1084
- vs. standard time 1082
- DayOfWeek enum type 1033
- DBMS 1512
- DDL 1514
- DDL operations 1514
- deadlocks 1408
- decimal separator 1125
- DecimalFormat 1115, 1125
- declaration 99
- declaration annotations 1571
- declaration context 493, 1571
- declaration statement 8, 65, 70, 81, 102, 103, 176
- declaration-before-reading rule 547
- declarations
  - arrays 92, 118
  - classes 99, 325
  - interfaces 325
  - local 101
  - main method 141
  - methods 100
  - multidimensional arrays 122
  - packages 325, 328
  - statements 101
  - variable arity method 136
- declare-before-use rule 572
- declared type 203, 264
  - see* static type
- declared-type parameters 680
- declaring *see* declarations
- decoupling 283
- decrement operator 69
- deep copying 426
- deep reflection 1191
- default
  - constructor 110
  - exception handler 367
  - values 103, 541, 555
- default 246
  - label 156, 161, 166
  - method 246
- default accessibility for class members
  - see* package accessibility for class members
- default accessibility for top-level types
  - see* package accessibility for top-level types
- default constructor 110
- default downstream collectors 985
- default file system 1289
- default locale 1098, 1099, 1104, 1107, 1109, 1119
- default method 246, 675, 676
  - conflict resolution 248
  - multiple inheritance of implementation 248
  - overriding 247
- default ordering
  - see* natural order
- default package 328
- default resource bundle 1102, 1103, 1109, 1110
- default values 1562, 1565
- defensive copying 359
- deferred execution 679, 684
- definite assignment analysis 105, 232
- definitions
  - interfaces 238
  - polymorphism 278
- DelayQueue 1496
- DELETE statement 1516
- delimiters 1120
- Denial of Service (DoS) 1602
- dependency cycles 1172
- deprecated 429
- deprecated code
  - @Deprecated 1580
  - deprecation warning 1581
  - ordinarily deprecated 1580
  - terminally deprecated 1580
- deprecation warning 1581, 1583
- Deque 821
- deque 785, 788
  - ArrayDeque 823
  - Deque 821
  - FIFO queue 821
  - head 821
  - LinkedList 823
  - tail 821
- Derby database 1520
  - connection 1520
- derived class 191
- deserialization 1611
- deserialization of objects 1261
- destination directory 335, 1187, 1219
- destination object 1152
- destination of data 1233
- destination stream 25
- destructive collection operations 791
- diagrams
  - class 6

- object 9
  - see also* UML
- diamond operator ( $\diamond$ ) 570
- dictionary 785
- dictionary order 447
- difference 804
- direct sealed superclass 313
- direct superclass 211
- directory
  - copy entire directory 1352
    - see also* directory entry
- directory entry 1288
  - accessing individual file attributes 1321
  - atomic move 1313
  - copy entire directory 1352
  - copy options 1308
  - copying 1308
  - copying using I/O streams 1311
  - creating 1339
  - creating regular directories 1342, 1343
  - creating regular files 1339, 1340
  - creating streams 1345
  - creating symbolic links 1342
  - creating temporary directories 1342, 1344
  - creating temporary files 1339, 1341
  - creation time 1324
  - deleting 1307
  - denotes same as another entry 1306
  - determining existence 1305
  - determining file accessibility 1324
  - determining kind of entry 1323
  - determining last modification time 1324
  - determining owner 1325
  - determining size 1323
  - handling POSIX file permissions 1325
  - hidden 1324
  - last access time 1324
  - last modified time 1324
  - managing file attributes 1321
  - moving and renaming 1312
  - reading and writing bytes 1318
  - reading and writing character data 1315
  - stream to read text lines 1346
- directory hierarchy
  - breadth-first traversal 1348
  - cyclic path dependency 1352
  - depth 1347
  - depth-first traversal 1348
  - finding/searching 1354
  - limit traversal by depth 1351
  - root 1347
  - siblings 1347
  - traversal strategies 1348
  - walking/traversing 1350
- directory-level separator 1189
- DirectoryNotEmptyException 1307
- disjunct collections 808
- dispensing canned advice application 1179
- distance of shift 86
- distributed 4
- divide-and-conquer
  - base case 1448
  - forking 1448
  - joining 1448
  - parallel processing 1448
- divide-and-conquer algorithm 868
- divide-and-conquer paradigm 1447
- dividend 62
- division
  - floating-point 62
  - integer 62
- division operator
  - / 62
- divisor 62
- DML 1514
- documentation 40
- documentation comment 40
  - tags 40
- documenting *see* documentation
- Domain Name System (DNS) 327, 1171
- DosFileAttributes interface 1330, 1333
- DosFileAttributeView interface 1334, 1338
- dot 326
- double 36, 42
- double quote 38
- DoubleBinaryOperator interface 722
- DoubleConsumer functional interface 709
- double-ended queues
  - deque 821
- DoubleFunction interface 712
- DoublePredicate functional interface 703
- DoubleStream interface 889
- DoubleSummaryStatistics 890
- DoubleSupplier functional interface 699
- DoubleToIntFunction interface 713
- DoubleToLongFunction interface 713
- DoubleUnaryOperator interface 720
  - composing unary operators 721
- doubly linked list 788
- do-while statement 173
- downcasting 45
- downstream collectors 979, 986, 1003
  - default 985
  - filtering adapter 992
  - finishing adapter 997

- flat mapping adapter 994
- mapping adapter 993
- DQL 1514
- dropping stream elements 910, 913
- duplicate elements 801, 804
- duplicating objects 427
- duration
  - accessing units 1067
  - comparable 1068
  - compared with period 1072
  - creating 1065
  - equality 1068
  - get methods 1067
  - immutable 1057, 1064
  - ISO format 1067
  - parsing 1066
  - plus/minus methods 1069
  - supported ChronoUnits 1066, 1068, 1070
  - temporal arithmetic 1069
  - thread-safe 1057, 1064
  - time-based 1057, 1064
  - unit conversion 1068
  - with methods 1069
- Duration class 1064
  - see* duration
- dynamic 4
- dynamic arrays 645
- dynamic binding
  - see* dynamic method lookup
- dynamic method lookup 206, 278, 283
- dynamic type 196, 203
- dynamically resizable arrays 802

## E

- eager execution 887
- effectively final 514, 687
- element type 118
- element type values for @Target 1569
- elements 117, 644, 783
- elements of collections 783
- eligible for garbage collection 534
- ellipsis 136
- else clause matching 155
- embedded applications 2
- empty set 804
- empty statement 101
- empty string 441
- encapsulation 3, 324, 326, 501, 1163, 1606
  - nested types 493
- encapsulation of implementation 195
- enclosing class 492, 500, 502
- enclosing context 493, 500, 505
- enclosing instance 493, 498, 524
- enclosing object reference 503
- encoding of characters 1241
- encounter order 891
- end of file
  - see* end of I/O stream
- end of file (EOF) 371
- end of I/O stream 1234, 1240, 1243, 1252, 1253, 1261
- ends with characters 452
- enhanced for loop 172
- enterprise applications 2
- entries 785, 830
- entry point 19, 22, 141, 1189, 1190, 1191, 1212
- entry set 831, 1380, 1388
- enum class
  - see* enum types
- enum constants 287
  - symbolic names 287
  - values 287
- enum types 163, 256, 287, 332, 635
  - abstract method declarations 294
  - accessibility 345
  - canonical form 288
  - constant-specific class bodies 294
  - declaring 288
  - enum constants 287
  - enum constructors 290
  - equality 293
  - final methods 293
  - implementing interfaces 298
  - implicit static methods 292
  - in switch expression 297
  - inherited methods 293
  - initializer blocks 291
  - local 494, 519
  - members 290
  - named constants 287
  - natural order 293
  - nested 492
  - ordinal value 163, 293
  - permitted direct subtypes 316
  - selector expression 156
  - static member 494, 495
  - top-level 491, 494
  - using 289
- enumerated types
  - see* enum types
- EOFException 371
- epoch 1024, 1049

- epoch-second 1049
- equality 75, 426, 744
  - equals method 77, 426, 744
  - object value 77
  - objects 77
  - primitive values 75
  - reference values 76
- equals method 77, 426, 744
- equivalence relation 746
- equivalent collections 808
- erasure 613, 616, 619
  - of the supertype method 621
  - same 620
- Error 374
- escape character 1101
- escape sequences 38, 459
- essential whitespace 462
- evaluation order 52, 80
  - arithmetic expressions 59
- evaluation short-circuits 80
- event listeners
  - anonymous classes 521
- exact integer arithmetic functions 481
- exam 1615
  - multiple-choice 1621
  - Oracle exam attempt 1617
  - program 1618
  - questions 1619
  - registration 1616
  - result 1619
- exam objectives
  - OCP Java SE 11 1629
  - OCP Java SE 17 1623
- exam question assumptions
  - OCP Java SE 17 1619
- Exception 370, 372
  - exception handler 365
    - see also* exceptions
- exception handling
  - advantages 416
- exception type 388
- exceptions 365, 376
  - ArrayStoreException 798
  - backtrace 405
  - cause 405
  - chaining 405
  - checked 371
  - ClassNotFoundException 1262
  - ConcurrentModificationException 795
  - customized 375
  - default handler 367
  - detail message 370
  - EOFException 1238
  - FileNotFoundException 1236
  - handler 365
  - IllegalThreadStateException 1378
  - IOException 1234, 1243
    - propagation 365
    - rethrowing 401
    - situations 370
    - suppressed 415
    - throw 386
    - throwing *see* throwing exceptions
    - thrown by JVM 370
    - thrown by method 100
    - thrown programmatically 370
    - throws 388
    - types 368
    - uncaught 367
    - unchecked 374
      - UnsupportedOperationException 790
  - exchanging information 127
  - execution 1370
  - execution paths 1367
  - execution stack
    - see* JVM stack
  - Executor 1424
  - executor 1423
  - Executor Framework 1423
  - executor service 1427
    - bulk task execution 1439
    - creating 1429, 1431
    - executor lifecycle 1427
    - shutdown 1429, 1431, 1433
    - submitting tasks 1429, 1431
    - task cancellation 1444
    - task execution policy 1426
    - tasks 1423
      - termination 1429, 1431, 1433
      - thread pool 1425
  - Executors 1424, 1425
  - ExecutorService 1427
  - explicit garbage collection 537
  - explicit module 1206, 1208
  - explicit referencing 515
  - exploded module 1181
  - exploded module directory 1181
  - exponent 36
  - exponential functions 480
  - exports directive 1169, 1171, 1178
    - accessibility 1173
  - exports-to directive 1178
  - expression statements 55, 70, 101, 160, 175, 176, 682

- expressions 166
    - actual parameters 128
    - boolean 74
    - conditional 90
    - deterministic evaluation 50
    - return 184
    - statements 101
    - switch expression 164
    - throw 386
    - yield 164
  - extending 521
    - inner class 508
    - interfaces 244
    - Thread 1375
  - extends 581, 589, 591
    - bounded type parameters 591
    - constraint 591
  - extends clause 13, 191
  - extensibility 1607
  - extensions
    - .class 21
    - .java 21, 22
  - external iteration 889
  - extracting substrings 453, 466
- F**
- fairness 1408
  - fairness policy 1461, 1470
    - acquisition order 1461
  - fallback 1110
  - fall-through 156, 157, 165
  - false literal 32, 37
  - fetch size 1536
  - FF *see* form feed
  - field declarations 99
  - field hiding 203, 231
  - field initialization 555
  - fields 6
    - final 229
    - hiding from superclass 203
    - shadowing by local variables 114
  - FIFO queue 821
  - file 761
    - see also* directory entry
  - file attribute views 1329, 1333, 1334
    - view name 1328
  - file attributes 1321
    - accessing by attribute name 1327
    - accessing by bulk operations 1328
    - accessing individually 1321
    - file attribute name format 1328
    - file attribute views 1329
    - file attributes interfaces 1330
    - file attributes interfaces 1328, 1330
    - File class 1290, 1292
    - file extension 1288
      - .properties 1103
    - file name 325
    - file path 335
      - separator character 335
    - file streams 1236
    - file system 1233, 1287
      - accessing 1610
      - default 1289
    - file system resources 1345
    - file visit option 1349
    - FileAlreadyExistsException 1309, 1339
    - FileAttribute interface 1339
    - FileAttributeView interface 1334
    - FileInputStream class 1235
    - FileNotFoundException 371
    - FileOutputStream class 1236
    - FileOwnerAttributeView interface 1334
    - FileReader class 1243
    - files
      - binary 1238
      - low-level I/O streams 1236
      - not found 1236
    - Files class 1287
      - method characteristics 1304
      - streams 903
    - FileSystem class 1289
    - FileSystems class 1289
    - FileTime class 1324
    - FileVisitOption enum type 1349
    - FileWriter class 1244
    - filtering 688
    - filtering adapter 992
    - filtering streams 910
      - using a Predicate 912
    - FilterInputStream class 1235
    - FilterOutputStream class 1236
    - filters 910
      - see also* I/O filter streams
    - final
      - effectively 525, 692
      - variables 525, 692
    - final fields 229, 230
      - hiding 231
    - final instance fields
      - initialization 231
    - final keyword 225
    - final local variables 229, 231

- final members 225
- final methods 226
  - binding of method call 227
- final parameters 135, 232
- final static fields
  - initialization 231
  - manifest constant 231
- final variables 229
  - blank 229
  - constant variables 229
  - definite assignment analysis 232
  - definitely assigned 232
  - definitely unassigned 233
  - fields 229
  - local variables 229
  - object state 229
- finalize method 427
- finally clause 377, 382
- finishing adapter 997
- finite streams 892
  - extracting lines from a String 902
  - from a BufferedReader 902
  - from a CharSequence 901
  - from arrays 898
  - from collections 897
  - from factory methods in the Files class 903
  - from specified values 893
  - from the Random class 900
- first element 810, 811, 1486
- first map entry 846
- fixed arity method 136
- fixed arity method call 140
- Fixed Thread Pool 1426
- fixed-size list view 659
- flat mapping adapter 994
- float 36, 42
- floating-point 41
  - double 42
  - float 42
  - literals 36
- floating-point arithmetic 60
- floating-point data types 36
- floating-point division 62
- floating-point remainder 63
- floor entry 846
- floor key 846
- flow control *see* control flow
- flushing I/O streams 1234
- for(;) statement 174
  - backward 175
  - collections 794
  - forward 175
  - iterating over array 121
  - iterator 794
- for(;) statement 176, 795
  - iterating over array 121
- for-each loop 172
- Fork/Join Framework 1009, 1447
- forking 1448
- fork-join pool 1449
- ForkJoinPool 1448
- ForkJoinTask 1448
- form feed 38, 39
- formal parameters 100, 109, 128, 265, 354
  - modifier 100
  - name 100
  - type 100
- formal type parameter 567
- Format class 1115
- format elements 1140
  - format style 1140
  - format type 1140
- format specifications 25, 457
- format specifiers 25, 1247
- format style 1127, 1132, 1140
- format type 1140
- FormatStyle enum type 1127, 1131
- formatted output 24
  - format specifiers 25
- formatted prompting 1257
- formatted string 457
- formatted values
  - Console 1257
  - PrintStream 1247
  - PrintWriter 1247
- Formatter 1152
- formatter
  - destination object 1152
- formatting 39, 1115
- formatting date and time values 1127
- formatting messages 1139
  - conditional formatting 1145
- formatting numbers/currency values 1116
- forward reference 541, 542, 545, 551, 555
- fractional signed numbers 41
- frame stack
  - see* JVM stack
- framework for collections 783
- full permissions 1343
- fully 1191
- fully qualified class name 337, 1189, 1190, 1191
- fully qualified name 314, 345, 497

- fully qualified package name 326, 328, 329
- fully qualified type name 326, 331
- function 684
- function arrow -> 679
- Function interface 712
  - composing functions 715
- function type 683
- functional interface
  - @FunctionalInterface 675, 1579
  - abstract method 675
  - built-in functional interfaces 695
  - function type 683
  - functional method 675
  - primitive values 696
  - target type 682
  - see also* lambda expressions
- functional method 675
- functional programming 5
- functional reduction 955, 956, 972, 984, 998
- functional requirements (FRs) 1600
- FunctionalInterface annotation 675
- functionality 675
- functional-style programming 675
- functional-style programming paradigm 3
- functions
  - currying functions 723

## G

- garbage collection 536, 537
  - automatic 533
  - facilitate 537
- general abstractions 194
- general contract
  - compareTo() 762
  - equals() 746
  - hashCode() 756
- general loops 174
- generalization 13
- generalized classes 194
- generic class 567, 568
- generic instance method 594
- generic interface 572
- generic method 593, 610, 657
  - calling 594
- generic method call
  - inference 594
- generic static method 594
- generic subtype method 621
- generic type 567, 645
  - extending 573
  - implementing 598

- generics 147
  - arrays 627
  - casting 625
  - class literals 636
  - collections 578
  - constructors 593
  - enums 635
  - erasure 613, 620
  - exception handling 632
  - inherited methods 620
  - instanceof operator 624
  - JVM 565
  - method signature 615
  - methods 593
  - nested classes 633
  - non-reifiable types 624
  - overloading 615, 616
  - overriding 615, 617
  - reifiable types 623, 624
  - subsignature 616
  - type erasure 576, 613
  - variable arity parameter 630
  - wildcards 579
- GESS 590
- get methods 300, 302, 358, 590, 975, 1032, 1099, 1531, 1534, 1536, 1537
- get operation 586
- getter methods *see* get methods
- GMT (Greenwich Mean Time) 1073
- goto 32
- grammar rules 30
- grant 1608
- graph
  - nodes 1166
  - unidirectional edges 1166
- grouping 326
- grouping separator 1125
- groups of objects 783
- GUI applications 1378

## H

- half-open interval 865, 867, 1145
- handle-or-declare rule 388
- handles *see* references
- happens-before relationship 1414, 1454
  - transitivity 1414
- hard link
  - see* symbolic link
- has-a* relationship 194
- hash code 108, 426, 753, 754
- hash function 753
- hash sign 1101

- hash table 754, 788, 789
    - see* map
  - hash value 426
    - see* hash code
  - hashing 753
    - collision 753
    - hash code 753
  - HashSet 806
  - Haskell Curry 724
  - head map view 845, 846
  - head set view 810, 811, 1486
  - heap 533, 1370
  - heap pollution 1585
  - heap pollution warning 630, 650, 659, 805, 833, 863
  - helper methods 252
  - heuristics 756
  - hidden members
    - accessing 505, 506
  - hiding internals 324
  - hierarchical file system 1233
  - higher entry 846
  - higher key 846
  - high-level I/O streams 1233
  - high-performance 5
  - holdability 1540
  - horizontal tab 38
  - hotspots 5
  - HT *see* horizontal tab
  - human time 1049
- I**
- I/O buffering 1238
  - I/O filter streams 1235, 1236, 1238
  - I/O package 1233
  - I/O streams 1233
    - buffering 1238
    - chaining 1238
    - closing 1234, 1239, 1240
    - end of stream 1240
    - files 1236
    - filter streams 1238
    - flushing 1234
    - high-level 1233
    - input filter streams 1235
    - low-level 1233
    - of binary values 1236, 1238
    - of characters 1241
    - of objects 1236
    - output filter streams 1236
    - skipping bytes 1238
  - IANA Time Zone Database 1073
  - IDE (integrated development environment) 1616
  - idempotent method 408, 413
  - identifiers 30
    - variable 102
  - identity container 965
  - identity conversion 46, 67
  - identity of object 8
  - identity() function 982
  - IEEE 754-1985 42
  - if block 152
  - if-else statement 153
    - scope of pattern variable 276
  - illegal thread state 1378
  - IllegalArgumentException 372, 810, 1134
  - IllegalStateException 373, 794
  - immediately enclosing instance 492
  - immutability 1453, 1607
  - immutable 302, 796, 1024, 1075
  - immutable classes
    - in Java SE platform documentation 356
  - immutable object pattern
    - see* immutable objects
  - immutable objects 429, 440
    - advantages 360
    - caching 358
    - consistency of state 358
    - defensive copying 359
    - guidelines 358
    - state 356
    - static factory method 359
    - thread-safety 356
  - immutable primitive values 356
  - immutable strings 440
  - implementation inheritance 245
  - implementations 6, 193, 324
    - collections 783
    - inheritance hierarchy 226
  - implementations of collections 785
  - implementing
    - Comparable 761
    - Comparator 769
    - equals 744
    - hashCode 753
    - interfaces 240
  - implements clause 240
  - implicit canonical record constructor 302
  - implicit default constructor 110
  - implicit dependency 1167, 1172, 1216
  - implicit inheritance 191
  - implicit narrowing conversions 67

- implied dependency 1175
- import 345
  - declaration 329
  - see also* static import
  - single-type-import declaration 329
  - statement 325
  - type-import-on-demand declaration 330
- importing
  - enum constants 332
  - nested types 498
  - reference types 329
  - static members 331
- in lockstep 797
- IN parameters 1530
- incidental whitespace 462
- increment operator 69
- indenting lines 456
- index 117
- index expression 120
- IndexOutOfBoundsException 400, 444, 445, 454, 465, 467, 468, 651, 652, 655
- individual array elements 120
- individual module compilation 1186
- inequality 75
  - see also* equality
- inferred-type lambda parameters 147
- inferred-type parameters 680
- infinite loop 176
- infinite streams 892
  - building with generator functions 894
  - from the Random class 900
- infinity 60, 432
  - negative 60
  - positive 60
- information hiding 324
- inheritance 13, 194, 238
  - hierarchy 193, 507
  - serialization 1270
  - subtype-supertype relationship 195
- inheritance of implementation 238
- inheritance of type 238
- initial capacity 647
- initial state of object 555
- initialization
  - arrays 119, 124
  - code 120
  - default values 103
  - for statement 174
  - objects 8
  - references 102
  - variables 102
- initialization expression 102, 142
- initializer 540
  - declaration-before-reading rule 542
  - static 541, 542, 553, 554, 557
- initializer blocks
  - instance 551
  - static 545
- initializer expression 103, 229, 230, 540
  - static 543
- initializers
  - declaration order 545
  - exception handling 549
  - forward references 545
  - non-static block 99
  - non-static field 99
  - static block 99
  - static field 99
  - using class name 547
  - using simple name 545
- initializing *see* initialization
- inlining 227
- inner classes 491, 492
  - anonymous 492, 494, 521
  - local 492, 494, 512
  - non-static member 492, 494
  - qualified superclass constructor
    - invocation 509
    - synchronization 1392
- INOUT parameters 1532
- input 1233
- input stream 1233
- input validation 1605
- InputStream abstract class 1234
- InputStreamReader class 1243
- INSERT statement 1515
- insertion order 644, 789, 807, 842
- insertion point 861, 867
- instance
  - members 13
  - methods 13, 101, 106
  - variable initialization 103
  - variables 13, 106
  - see also* object
- instance initializer blocks 551
  - local variable type inference 552
- instance methods 9
- instance variables 8, 555
- instanceof operators 92, 269, 274, 624
- instanceof pattern match operator 274
  - pattern variable 275
  - type pattern 275
- instanceof type comparison operator 270, 281

- instant
  - accessing fields 1052
  - comparing 1053
  - convert to temporal objects 1056
  - creating 1049
  - creating from date-time 1051
  - current instant 1050
  - epoch 1049
  - epoch-second 1049
  - immutable 1049
  - ISO format 1050
  - max value 1049
  - min value 1049
  - nano-of-second 1049
  - parsing 1051
  - plus/minus methods 1054
  - representation 1049
  - supported ChronoFields 1054
  - supported ChronoUnits 1056
  - temporal arithmetic 1054
  - text representation 1051
  - thread-safe 1049
  - timestamps 1049
  - with methods 1054
  - zero UTC offset 1049
- Instant class
  - see* instant
- instant timeline 1074
- instantiation 8
  - anonymous classes 524
  - local classes 517
  - non-static member classes 502
  - parameterized type 569
  - static member classes 495
- int 32, 42
- IntBinaryOperator interface 722
- IntConsumer functional interface 709
- integer arithmetic 59
- integer bitwise operators 82
- integer constant expressions 48
- integer data types 32
- integer division 62
- integer remainder operation 62
- integers 41
  - and* operator 83
  - byte 42
  - complement 83
  - data types 41
  - int 42
  - literals 32
  - long 42
  - or* operator 83
  - representation 54
  - short 42
  - types 41
  - xor* 83
- integral types 41, 42, 44
- interface constant antipattern 332
- interface-based 787
- interfaces
  - abstract 239
  - abstract methods 240
  - accessibility 345
  - body 238
  - collection core 783
  - constants 254
  - contract 240
  - declarations 325
  - default methods 246
  - evolution 250
  - extending 244
  - functional interfaces 675
  - header 238
  - implementing 240, 298, 523
  - inheriting from multiple interfaces 621
  - initialization 557
  - local 494, 519
  - markers 239
  - member declarations 238, 239
  - multiple inheritance of implementation 248
  - nested 491, 492
  - overriding default methods 247
  - partial implementation 242
  - private methods 252
  - public methods from Object class 245
  - realization 244
  - references 246
  - static member 494, 495
  - static methods 251
  - subinterfaces 244
  - superinterfaces 244
  - top-level 491, 494
  - UML 244
  - UML stereotype 244
  - variables 254
  - with single abstract method 678
- intermediate stream operations 885, 905
  - examining elements 920
  - filtering streams 912
  - flattening streams 924
  - input stream 885
  - lazy execution 887
  - mapping streams 921
  - non-interfering behavioral parameters 910

- order of 908
  - output stream 885
  - replacing each element with multiple elements 927
  - selecting distinct elements 915
  - setting execution mode 933
  - setting stream as unordered 932
  - short-circuit execution 908, 917, 918
  - skipping elements 915
  - sorting streams 929
  - stateful operations 908
  - stateless behavioral parameters 910
  - stateless operations 908
  - truncating a stream 917
  - internal iteration 889
  - internal packages 1171
  - internationalization 1096
  - interned strings 440, 442
  - interned values 433
  - interpackage accessibility 324
  - interrupt handling 1393
    - interrupt status 1393
      - InterruptedException 1393
      - spurious wakeup 1399
    - InterruptedException 1393
  - interruptible locking 1462, 1467
  - intersection 804
  - IntFunction interface 712
  - IntPredicate functional interface 703
  - intrinsic lock
    - see* object lock
  - intrinsic locking 1388, 1460
  - IntStream interface 889
  - IntSummaryStatistics 890
  - IntSupplier functional interface 699
  - IntToDoubleFunction interface 713
  - IntToLongFunction interface 712
  - IntUnaryOperator interface 720
    - composing unary operators 721
  - invocation
    - parameterized type 569
  - invocation stack
    - see* JVM stack
  - invoker 184
  - invoking garbage collection 537
  - IOException 371
  - is-a* relationship 194, 195
  - ISO 8859-1 encoding scheme 439
  - ISO Latin-1 37, 42
  - ISO standard 1127, 1129
  - Iterable interface 450, 657, 792, 795
  - iteration 174, 792
  - iteration order 795
  - iteration statements 172
    - next iteration 182
    - termination 172, 181
  - iterators 645, 657, 793, 797
    - fail-fast 794, 1485
    - ListIterator 801
    - ordered collection 796
    - snapshot-style 1484
    - sorted collection 796
    - unordered collection 796
    - weakly consistent 1484
- ## J
- Jakarta EE (Enterprise Edition) 2
  - JAR 1163, 1189
  - jar tool 1189
  - java 21
  - Java Application Launcher 21
    - see* JDK tool java
  - Java Archive
    - see* JAR
  - Java Archive Tool
    - see* JDK tool jar
  - Java bytecode 21
  - Java Card 2
  - Java Class Dependency Analyzer Tool
    - see* JDK tool jdeps
  - Java Collections Framework 644
  - Java compilation unit 1168
  - Java Database Connectivity
    - see* JDBC
  - Java Development Kit (JDK) 2
  - Java ecosystem 2
  - Java EE (Enterprise Edition) 2
  - Java Language Compiler 20
    - see* JDK tool javac
  - Java Linker
    - see* JDK tool jlink
  - Java ME (Micro Edition) 2
  - Java Platform Module System 1163
  - Java Platforms 2
  - Java Runtime Environment (JRE) 3
  - Java SE (Standard Edition) 2
  - Java Shell Tool (jshell) 23
  - Java Virtual Machine (JVM) 2
  - java.base module 1167, 1214
  - java.io package 1233, 1290
  - java.lang.annotation package 1557
  - java.lang.Record class 304
  - java.lang.reflect package 1587
  - java.math package 485
  - java.net package 1290

- java.nio package 1287
- java.nio.file package 1287
- java.nio.file.attribute package 1324
- java.se module 1168
- java.sql package 1517
- java.text package 1115
- java.time package 1024, 1115
- java.time.format package 1115
- java.util package 644
- java.util.concurrent package 1424
- java.util.function package 695
- javac 20
- Javadoc comment 40
  - @param tag 100, 112
  - @return tag 184
  - @throws tag 390
- javadoc tool 40
- JDBC 1517
- JDBC API 1517
- JDBC drivers 1517
  - legacy drivers 1522
- JDBC interfaces 1519
- JDBC resources 1518
- JDBC URL 1519
- JDK 1616
  - licencing 3
- JDK tools
  - jar 1189, 1211, 1212, 1213, 1221
  - java 21, 1189, 1212, 1213, 1220
  - javac 20, 1186, 1219
  - javadoc 40
  - jdeps 1167, 1214, 1222
  - jlink 1204, 1222
  - jmod 1166
  - jsell 23
  - summary of operations 1218
- JMOD archive 1166
- JMOD files 1213
- join
  - starvation 1412
- joining 1448
- joining threads 1383
- JPMS 1163
- just-in-time (JIT) 4
- JVM 534, 537
  - heap 1370
  - JVM stack 1370
  - method area 1370
  - program counter (PC) 1370
  - runtime constant pool 1370
  - runtime data areas 1369
- JVM stack 365, 534, 1370

## K

- key 866
- key insertion order 841
- key objects 785
- key sets 831, 847
- key sorting 845
- key-based lookup 1104
- keys 830
- keystore 1608
- key-value entries 754
- key-value pairs 1102
- keywords 31
  - abstract 219, 240, 675
  - boolean 43
  - break statement 180
  - byte 42
  - case 156, 160, 164
  - catch 377
  - char 42
  - class 99, 238
  - contextual words 31
  - continue 182
  - default 156, 161, 166, 246
  - do 173
  - double 42
  - else 153
  - extends 191, 581, 589, 591
  - final 135, 225, 232, 312, 358, 400, 410, 1456
  - finally 382
  - float 42
  - for 174, 176
  - if 152
  - implements 240
  - import 329, 345
  - instanceof 92, 270, 274, 281, 624
  - int 42
  - interface 238
  - long 42
  - new *see* new operator
  - null 49, 77, 269
  - package 328, 345
  - private 351
  - protected 350
  - public 350
  - reserved words 31
  - return 184
  - short 42
  - static 19, 113, 251, 331, 495, 545
  - super 110, 201, 206, 211, 248, 507, 582
  - switch 155, 164

- synchronized 1388, 1456
  - this 106, 504
  - throw 386
  - throws 388
  - try 377
  - unused words 31
  - void 19, 429
  - volatile 1454
  - while 172, 173
  - keywords (contextual)
    - exports 31, 1169, 1177
    - module 31, 1177
    - non-sealed 31
    - open 31, 1177
    - opens 31, 1177
    - permits 31
    - provides 31, 1177
    - record 31
    - requires 31, 1170, 1175, 1177
    - sealed 31
    - to 31, 1177
    - transitive 31, 1177
    - uses 31, 1177
    - var 31
    - with 31, 1177
    - yield 31
  - keywords (reserved)
    - const 32
    - goto 32
- L**
- labeled break statement 181
  - labels 179, 181
    - break 181
    - case constants 156, 160
    - constant expressions 157
    - default 156, 161, 166
    - labeled statement 179
    - switch statement 156, 160
  - lambda body 679, 681
  - lambda expressions 679, 691
    - access class members 685
    - block scope 684, 686
    - compared with anonymous classes 692
    - contexts providing target types 733
    - declared-type parameters 680
    - deferred execution 679, 684
    - expression 681
    - expression statement 682
    - function 684
    - function arrow -> 679
    - implicitly typed 680
    - inferred-type parameters 680
    - lambda body 679, 681
    - lambda parameters 680
    - non-void return 681
    - parameter list 679
    - shadowing members 685
    - single expression 681
    - statement block 682
    - target type 682
    - target typing 684
    - type checking 682
    - using this and super 685
    - variable capture 687
    - var-type inferred parameters 680
    - void return 681
  - lambda parameters 680
  - language codes 1097
  - language tag 1118
  - last element 810, 811, 1486
  - last map entry 846
  - latch
    - see* count-down latch
  - late binding
    - see* dynamic method lookup
  - LATIN-1 encoding scheme 439, 440
  - lazy execution 887, 907
  - least significant bit 34
  - left associativity 51
  - left shift 86
  - legacy class 566
  - legacy code 565
  - legacy supertypes 619
  - legal assignments 264
  - length method 445, 902
  - letter pattern 1134
  - lexical nesting 496
  - lexical scope
    - see* block scope
  - lexical tokens 30
  - lexicographic array comparison 869
  - lexicographic comparison 869
  - lexicographical ordering 447
  - LF *see* linefeed
  - lifetime 535
    - see* scope 106
  - limits 1146
  - line continuation 38
  - line separator 25, 1246
  - line terminator 39, 445
  - linear implementation inheritance 193
  - linear inheritance 237
  - line-continuation character 1186
  - linefeed 38

- lines of text 1252
- LinkageError 374
- linked data structures 566
- linked list 788, 802
- LinkedBlockingDeque 1496
- LinkedBlockingQueue 1496
- LinkedHashSet 806
- LinkedList 647, 802, 815, 823
- LinkedTransferQueue 1496
- LinkOption enum type 1301
- List 645, 801
- list views 656, 659, 660
- listing observable modules 1212
- ListIterator 801
- lists 784, 801, 802
  - ArrayList 802
  - binary search 861
  - changing contents 863
  - copying 863
  - LinkedList 802
  - List 801
  - ListIterator 801
  - positional 785
  - replacing elements 862, 863
  - reverse order 859
  - sorted 861
  - sorting 858, 859
  - sublist view 656
  - sublists 862
  - unmodifiable 649, 801
  - unmodifiable views 856
  - Vector 802
  - views 659
  - see also* ArrayList
- literals 32
  - boolean 37
  - character 37
  - default type 32, 36
  - double 36
  - escape sequences 38
  - false 37
  - float 36
  - floating-point 36
  - integer 32
  - null 32
  - predefined 31
  - prefix 33
  - quoting 37
  - scientific notation 36
  - string 39
  - suffix 32, 36
  - true 37
- live threads 1383
- livelock 1410
- liveness 1408
- local 102, 104, 519
  - block 492
    - chaining of constructors 210, 555
    - classes 494, 512
    - enum types 492, 494
    - interfaces 492, 494
    - record classes 492, 494
    - variables 106, 354
- local classes 492, 494, 512
  - accessing local declarations 513
  - accessing members in enclosing context 515
  - declaration 512
  - hiding members from enclosing context 515
  - in non-static context 512
  - in static context 512
  - instantiation 517
  - scope 512
  - shadowing local variables 514
  - using qualified this 515
- local date
  - conversion to Date 1088, 1141
- local date-time
  - conversion to Date 1088, 1141
  - converting to instant 1051
- local declarations 101
- local enum types 519
- local interfaces 519
- local record class 519
- local scope 169
- local time 1073
  - conversion to Date 1088, 1141
- local timeline 1074
- local variable type inference 142, 410
  - arrays 143
  - instance initializer blocks 552
  - parameterized types 571
  - static initializer blocks 545, 547
- local variables 109, 142
  - block scope 684
- LocalDate
  - see* temporal objects
- LocalDateTime class
  - see* temporal objects
- Locale 1096
- locale 448, 1096
  - country codes 1097
  - country name 1098

- default 1098
- language codes 1097
- language name 1098
- predefined 1097
- locale qualifiers 1098
- locale-sensitive 1115
- locale-sensitive operations 1096
- localization 1096
- localizing an application 1105
- localizing information 324
- LocalTime class
  - see* temporal objects
- locations
  - see* class path
- lock 1461
- lock 1380
  - acquire 1380, 1388
  - class 1391
  - entry set 1380, 1388
  - release 1380
- lock acquisition
  - starvation 1412
- logarithm 480
- logical 277
- logical AND
  - & 78
- logical complement
  - ! 78
- logical complement (!) operator
  - pattern variable 277
- logical exclusive OR
  - ^ 78
- logical inclusive OR
  - | 78
- logical XOR
  - ^ 78
- long 32, 42
  - suffix 32
- LongBinaryOperator interface 722
- LongConsumer functional interface 709
- LongFunction interface 712
- LongPredicate functional interface 703
- LongStream interface 889
- LongSummaryStatistics 890
- LongSupplier functional interface 699
- LongToDoubleFunction interface 713
- LongToIntFunction interface 713
- LongUnaryOperator interface 720
  - composing unary operators 721
- loop body 172, 174
- loop condition 172, 174
- loop variable 174

- loops *see* iteration statements
- loss of precision 44
- lower bound 581
- lower bounded wildcard
  - ? super T 582
- lower bounded wildcard references 589
- lower entry 846
- lower key 846
- low-level I/O streams 1233
- LTS (long-term support) release 3

## M

- magnitude 44
- main memory 1453
- main method 21, 141
  - arguments 141
  - modifiers 141
- main thread 1373, 1378
- Main-Class attribute 1212
- MANIFEST.MF file 1212
- many-to-one relation 830
- Map 831
- map
  - key-value entries 754
- map views 839
- Map.Entry 833, 846
- mapping adapter 993
- mappings 785
  - see* entries 785
- map-reduce operation 1003
- map-reduce transformation 958
- maps 785, 830
  - access order 841
  - entries 785
  - iteration 839
  - key insertion order 841
  - key-based operations 831, 835
  - Map 831, 840
  - map views 839
  - mappings 785
  - multimap 597
  - NavigableMap 845
  - ordering mode 841
  - replacing values 839
  - SortedMap 845
  - submap view 846
  - text representation 841
  - TreeMap 847
  - unmodifiable 832
  - unmodifiable views 857
  - views 839

- marker annotation 1563, 1565
- marker annotation type 1560
- marker interfaces 239, 1262
- marker parameter 1526, 1530
- mask
  - see* bit mask
- Math class 108, 478
- math constants 478
- mathematical set 804
- MAX\_VALUE constant 434
- maximum value 479
- member declarations 99, 307
- member type declarations 99
  - non-static member class 492
  - static member class 492
  - static member enum type 492
  - static member interface 492
  - static record class 492
- members 6, 352
  - access 106
  - accessibility 347
  - default values 103
  - inheritance 191
  - modified 191
  - of objects 9
  - scope 352
  - shorthand 107
  - static 10
  - terminology 13
  - variables *see* fields
- memory consistency errors 1414
- memory management 533
- memory organization 533
- MessageFormat 1115
- meta-annotation 1559, 1567
  - @Documented 1574
  - @Inherited 1574
  - @Repeatable 1575
  - @Retention 1567
  - @Target 1569
- metadata 239, 1557
- META-INF directory 1211, 1212
- method area 1370
- method call 9, 100, 127
  - chaining 467, 469
  - fixed arity 140
  - variable arity 139
- method chaining 906, 1036, 1042, 1060, 1069
- method declaration 99
  - variable arity parameter 630
- method header 224
- method invocation conversions 48, 265
- method modifiers 100
- method naming conventions 1025
- method overloading 108, 202, 225
- method overriding 202, 224, 226, 556
  - covariant return 201
- method references 724, 725
  - bounded instance method references 727
  - static method references 726
  - unbounded instance method references 729
- method signature 101, 196
- method type 683
- methods 6
  - @Override 198
  - abstract 224, 240, 675
  - abstract method declarations 240
  - advisory 1385, 1387
  - ambiguous call 202, 266
  - automatic variables *see* local variables
  - behavior 675
  - binding of method call 205
  - blocks 101
  - body 100
  - bridge 615
  - call chaining 128
  - call *see* method call
  - calling variable arity method 137
  - chained 449
  - clone 426
  - concrete 226
  - declaration 100, 128
  - default 246
  - dynamic lookup 283
  - equals 77, 426, 744
  - exceptions 100
  - final 226
  - finalize 427
  - fixed arity 136
  - functional 675
  - generic 593
  - getClass 426
  - header 100
  - hiding 226
  - hiding static methods 203
  - implementation 224
  - invocation *see* method call
  - local declarations 101
  - local variables
  - locale-sensitive 1099
  - main *see* main method
  - method invocation conversions 265

- method type 683
- modifiers 100
- most specific 266, 655
- mutually exclusive 1391
- name 127
- objects 106
- overloaded resolution 265
- overloading *see* method overloading
- overriding *see* method overriding
- overriding vs. overloading 202
- parameters 100
- return 184
- return value 100
- signature 101, 108, 202, 616
- static 115, 251
- subsignature 617
- synchronized 1388
- termination 184
- throws clause 388
- toString 426
- variable arity 136
- migration to modules 1209
- MIN\_VALUE constant 434
- minimizing overhead 536
- minimum value 478
- MissingResourceException 374, 1104
- mixing raw types and generic types 575
- mobile applications 2
- modifiers
  - abstract 240
  - access 345
  - default 246
  - final 225
  - static 251
- modular JAR 1189, 1206
  - creating 1189
  - describe a JAR 1213
  - extract contents 1212
  - list contents 1211
  - listing observable modules 1212
  - Main-Class attribute 1212
  - MANIFEST.MF file 1212
  - META-INF directory 1211, 1212
  - running from 1191
- modular JDK 3, 1164
  - goals 1165
  - java.base module 1167, 1214
  - java.se module 1168
- modular JDK installation 1165
- module application directory structure 1180
- module declaration 1168
  - module as contextual keyword 1169
- module dependency 1170
- module descriptor 1169, 1187
- module directives 1169
  - exports directive 1169, 1171, 1178
  - exports-to directive 1178
  - opens directive 1178, 1191, 1192
  - opens-to directive 1178, 1195
  - overview of directives 1177
  - provides-with directive 1178, 1200
  - qualified export 1178
  - requires directive 1169, 1170, 1177
  - requires static directive 1177
  - requires transitive directive 1175, 1177
  - unqualified export 1169, 1178
  - uses directive 1178, 1202
- module graph 1166, 1172, 1180, 1217
  - acyclic 1172
  - dependency cycles 1172
  - implicit dependency 1167, 1172
  - implied dependency 1175
  - module dependency 1170
  - sink nodes 1172
- module name 1171
- module path 1206
- module resolution 1172, 1186
- module root directory 1181
- module-info.java file 1181
- modules 1168
  - accessibility 1173
  - accessibility rules 1170, 1174
  - aggregate module 1168
  - automatic 1206, 1207
  - benefits 1163
  - comparison 1208
  - creating modular JAR 1189
  - dependencies 1166
  - explicit 1206, 1208
  - exploded module 1181
  - exploded module directory 1181
  - exploring modules 1211
  - implicit dependency 1216
  - individual module compilation 1186
  - internal packages 1171
  - migration to 1209
  - modular JAR 1189
  - module application directory structure 1180
  - module declaration 1168
  - module descriptor 1169
  - module graph 1166
  - module name 1171
  - module resolution 1186
  - module root directory 1181
  - module-info.java file 1181

- multi-module compilation 1187
  - named 1208
  - non-standard modules 1166
  - open modifier 1178
  - platform modules 1166
  - principle exported package 1171
  - public API 1163, 1170, 1174
  - readability 1173
  - readable 1170
  - reliable configuration 1163
  - running from a modular JAR 1191
  - services 1196
  - standard modules 1166
  - stronger encapsulation 1163
  - system modules 1212
  - unnamed 1206
  - viewing dependencies 1214
  - monitor 1380
    - entry set 1380
    - lock 1380
  - monitor lock
    - see* object lock
  - Month enum type 1028
  - most specific method 266, 655
  - multi-catch clause 397
    - final parameter 400
  - multicore architectures 5
  - multidimensional arrays 122, 124
  - multi-element annotation type 1561
  - multilevel classification 987
  - multilevel grouping 987
  - multilevel partitioning 991
  - multimap 597
  - multi-module compilation 1187
  - multi-paradigm programming 3
  - multiple assignments 55
  - multiple bounds 592
  - multiple catch clauses 376, 397
  - multiple inheritance of implementation
    - 193, 238, 248
  - multiple inheritance of type 238
  - multiple-line comment 40
  - multiplication operator
    - \* 61
  - multiplicative operators 61
  - multisets 804
  - multitasking 1367
  - multithreaded 5
  - multithreaded programming 1368
  - mutability 661
  - mutable character sequences 464
  - mutable containers 947, 964
  - mutable objects 359
  - mutable reduction 971, 972
    - mutable containers 964
    - see also* collectors
  - mutator method *see* set methods
  - mutex
    - see* lock
  - mutual exclusion 1455
  - mutually comparable 858, 861, 867
  - mutually exclusive
    - actions 154
    - locks 1380
  - MVC design pattern 1179
- ## N
- name 30
  - name clash 620, 621, 622
  - name elements 1288
  - name separator in path 1288, 1291
  - named constants 231
  - named modules 314, 1208
  - named package 314
  - namespaces 109, 1171
  - naming conventions of paths 1288
  - NaN (Not-a-Number) 61, 432, 1606
  - nano-of-second 1049
  - narrower range 44
  - narrowing conversions
    - primitive 44
    - reference 45
  - narrowing reference conversions 196, 270,
    - 585, 605
  - natural logarithms 478, 480
  - natural order 761, 858, 865
  - natural ordering 865
  - navigable map 785
    - first entry 846
    - last entry 846
    - range-views 846
    - views 846
  - navigable set 784, 810
    - ceiling 812, 1487
    - floor 812, 1487
    - higher 812, 1487
    - iterator 812, 1487
    - lower 812, 1487
  - NavigableMap 845
  - NavigableSet 810
  - navigation methods 845
    - closest match 811
  - nbsp
    - see* non-breaking space
  - negative zero 61

- nested classes 492
  - anonymous 492, 494, 521
  - class file name 497
  - fully qualified name 497
  - generic 633
  - local 492, 494, 512
  - non-static member 492, 494, 501
  - qualified name 496
  - static member 491, 494, 495
  - synchronization 1392
- nested enum types
  - local 494
  - static member 491, 494
- nested interfaces 491
  - static member 494
- nested lists 648
- nested loops 126
- nested parameterization 575
- nested parameterized types 608
- nested record classes 492
  - local 494
  - static member 494
- nested type declaration 491
- nested types 491
  - accessibility 493
  - enclosing context 493
  - enclosing instance 493
  - importing 498
- nested wildcards 584
- new operator 8, 92, 109, 119, 507, 521, 555
  - qualified class instance creation
    - expression 502
- NEW state 1382, 1386
- newline *see* linefeed
- NIO.2 1287
- NL *see* newline
- no-argument constructor 110, 210, 213
- NoClassDefFoundError 374
- nodes 566, 1166
- non-access method modifier 99, 100
- non-associativity 50
- non-atomic operation 1455
- non-blocking collection 1483
- non-blocking lock-free synchronization 1457
- non-breaking space (nbsp) 1117, 1120
- non-functional requirements (NFRs) 1600
- non-interfering behavioral parameters 910
- non-modular JARS
  - see* plain JARS
- non-observable substate 1382
- non-reifiable types 623, 631
- non-runnable states 1383
- non-standard modules 1166
- non-static code 100
  - see* non-static context 99
- non-static context 99, 100
- non-static field 13
- non-static field initializers 99
- non-static initializer block 99
- non-static member classes 494
  - access hidden members 505
  - access in enclosing context 504
  - declaration 501
  - fully qualified name 497
  - hiding in enclosing context 507
  - inheritance 507
  - instantiation 502
  - qualified class instance creation
    - expression 502, 503
  - qualified name 496
  - qualified superclass constructor
    - invocation 509
  - qualified this 504
- non-static nested classes
  - see* inner classes
- non-static type 568
- non-varargs call
  - see* fixed arity call
- non-void return 681
- non-zero argument constructor 111, 214
- normal annotation 1563
  - array element initializer 1564
  - element-value pair 1563
  - single-element array-valued element-value pair 1564
- normal canonical record constructor 304, 305
- normal execution 366
- normal non-canonical record constructor 305, 307
- normal threads 1377
- NoSuchElementException 793, 811, 815
- NoSuchFileException 1307
- Not-a-Number
  - see* NAN
- notifying threads 427, 1396
- NotSerializableException 371
- null literal 32
  - casting 49
  - equality comparison 77
- null reference 32
  - casting 269
- nulling references 536

- NullPointerException 373
  - null-safe 743
  - Number class 434
  - number format pattern symbols 1125
  - number formatter 1117, 1119
    - accounting currency 1118
    - compact form 1120
    - currency 1116, 1117
    - numbers 1116
    - percentage 1116, 1117
  - number formatting
    - accounting currency 1118
    - compact form 1120
    - compact form styles 1120
    - pattern symbols 1125
    - static factory methods 1116
    - string pattern 1125
  - number systems
    - base 33
    - decimal 33
    - hexadecimal 33
    - octal 33
    - radix 33
  - NumberFormat 1115, 1116
  - NumberFormatException 373, 397, 431
  - numbers 1096, 1116
    - formatting 1116
  - numeric optional classes 944
  - numeric promotions 49
    - assignment 56
    - binary 50
    - unary 49
  - numeric streams 885, 892
    - averaging 974
    - from a range 898
    - from arrays 898
    - from the Random class 900
    - functional reduction 972
    - summary statistics 972, 974
    - summation 973
  - numeric types 41
  - numeric wrapper classes 434
  - numerical literals
    - using underscore 36
- O**
- ObjDoubleConsumer functional interface 709
  - Object class 193, 425
  - object creation 1270
  - object graph 1266
  - object hierarchy 195
  - object lock 1380, 1388, 1391, 1397, 1398, 1399, 1408
  - Object lock rule 1414
  - object reference equality 745
  - object references 8, 102
  - object serialization
    - see* serialization
  - object state 8, 109, 131, 555
    - immutable 356, 1453
  - object streams 885, 892
  - object type *see* dynamic type
  - object value equality 745
  - ObjectInputStream class 1235
  - object-level lock
    - see* object lock
  - object-oriented programming 6
  - object-oriented programming paradigm 3
  - ObjectOutputStream class 1236, 1261
  - objects 8
    - aggregate 16
    - alive 534
    - arrays 118
    - behaviors 6
    - callee 127
    - caller 127
    - Class class 426
    - cleaning up 536
    - cloning 427
    - communication 127
    - comparing 426, 744, 769
    - compound 534
    - constituent 16, 534
    - constructing 555
    - contract 324
    - creation 1270
    - decoupling 283
    - deserialization 1261
    - eligible 536
    - equality 77, 426, 744
    - exchanging information 127
    - garbage collection 533
    - I/O streams 1236
    - identity 8
    - immutable 429
    - implementation 324
    - initial state 555
    - initialization 8, 109
    - initializer block 551
    - internals 324
    - lifetime 535
    - members 9
    - methods 106

- Object class 425
- persistence 1261
- properties 6
- reachable 533, 534
- reading 1261
- serialization 1261
- services 324
- state 229
- state *see* object state
- value equality 77
- wait set for threads 1396
- writing 1261
- Objects class
  - comparing objects 743
  - computing hash code 743
  - null-safe methods 743
- ObjIntConsumer functional interface 709
- ObjLongConsumer functional interface 709
- observable modules 1213
- on/off latch 1475
- one's complement 34
- one-arity 695
- one-dimensional arrays 118
- one-shot tasks 1440
- one-to-many mapping 924, 927
- one-to-one mapping 921
- one-to-one transformation 927
- online proctored exam 1618
- on-the-fly classes 492
- open modifier 1178, 1196
- open module 1196
- open package 1192
- open range-view operations 801
- opening delimiter 458
- OpenOption interface 1301
- opens directive 1178, 1191, 1192
- opens-to directive 1178, 1195
- operands 48
  - evaluation order 52
- operations 6
  - put 586
  - read 586
  - write 586
- operators 50
  - 58, 63
  - 69
  - ^ 78, 82
  - ^= 79, 85
  - ! 78
  - != 75, 76
  - ? : 90
  - . 9, 326
  - [] 92, 120
  - \* 58, 61
  - \*= 66
  - / 58, 61, 62
  - /= 66
  - & 78, 82
  - && 80
  - &= 79, 85
  - % 58, 61, 62
  - %= 66
  - + 58, 63, 67
  - ++ 69
  - += 66
  - < 74
  - << 86, 87
  - <<= 90
  - <= 74
  - = 66
  - = 54
  - == 75, 76
  - > 92
  - > 74
  - >= 74
  - >> 86, 88
  - >>= 90
  - >>> 86, 89
  - >>>= 90
  - | 78, 82
  - |= 79, 85
  - || 80
  - ~ 82
- arithmetic compound assignment 66
- assignment 50, 54
- associativity 50
- binary 50
- bitwise 82, 86
- boolean 74, 75, 78
- cast 50
- comparisons 74
- compound assignment 79, 85
- conditional 80, 90
- decrement 69
- dot 9
- equality 75
- execution order 52
- extended assignments 90
- floating-point 60
- floating-point division 62
- floating-point remainder 63
- increment 69
- instanceof 92
- instanceof pattern match operator 274

- instanceof type comparison operator 270
  - integer 82
  - integer arithmetic 59
  - integer division 62
  - integer remainder 62
  - logical 78
  - multiplicative 61
  - new *see* new operator
  - overflow 60
  - overloaded 59, 62
  - postfix 50
  - precedence 50
  - relational 74
  - shift 86
  - short-circuited 80
  - string concatenation 67
  - ternary 50
  - unary 50, 61
  - unary - 61
  - unary + 61
  - optimizations 5
  - optional annotation element 1562, 1566
  - Optional class 890, 940
    - absence of a value 940
    - creating 941
    - declaring and returning 940
    - querying 943
    - see also* numeric optional classes 944
  - optional dependency 1177
  - optional operations 790
  - OptionalDouble 890, 944
  - OptionalInt 890, 944
  - OptionalLong 890, 944
  - optionally bounded collections 1483
  - or operator 83
  - Oracle CertView 1617
  - Oracle exam attempt 1617
  - Oracle JDK 3
  - Oracle MyLearn 1617
  - ordered collections 784, 801
  - ordered streams 891
    - extracting lines from a String 902
    - from a BufferedReader 902
    - from a CharSequence 901
    - from factory methods in the Files class 903
  - ordering mode 841
  - ordinal value 163, 293
  - OUT parameters 1530
  - outer instance 508, 509
  - OutOfMemoryError 374, 538
  - output 24, 1233
  - output stream 1233
  - OutputStream abstract class 1234
  - OutputStreamWriter class 1244
  - overflow 34, 60
  - overflow errors 481
  - overloaded 59
  - overloaded method resolution 265
  - overloaded methods
    - ambiguous call 202
  - overloading 616
    - constructors 112
    - method resolution 265
    - methods 108, 202
  - overloading vs. overriding 202
  - override criteria 617
  - override-equivalent 616, 621
  - overriding 390, 621
    - @Override 618, 1578
    - covariant return
    - equals 77, 744
    - hashCode 753
    - methods 202
    - override-equivalent 616
    - toString 68
  - overriding vs. overloading 202
- ## P
- package accessibility for class members
    - 347, 351
  - package accessibility for top-level types 345
  - package directory 335
  - package statement 325, 328, 345
  - package-private accessibility for class
    - members
    - see* package accessibility for class members
  - package-private accessibility for top-level
    - types
    - see* package accessibility for top-level types
  - packages 326
    - classpath 339
    - declaration 325
    - definition 328
    - destination directory 335
    - hierarchy 326
    - java.io 1233
    - java.lang 425
    - java.util 783
    - location 339
    - members 326
    - package directory 335
    - root 339

- running code from 337
  - shorthand 329
  - statement *see* package statement
  - unnamed 328
  - using 329
- palindromes 688
- parallel processing 1448
- parallel streams 891, 1009
  - autoboxing 1015
  - behavioral parameters 910
  - benchmarking 1010
  - building parallel streams 934, 1009
  - collecting to a `ConcurrentMap` 983
  - computation-intensive operations 1010
  - concatenating 896
  - concurrent collectors 983, 989
  - data size 1010
  - determine if parallel 934
  - encounter order 953
  - execution 962, 966, 1009
  - factors affecting performance 1010
  - from collections 897
  - grouping to a `ConcurrentMap` 989
  - ordered 892
  - ordering 1015
  - parallel functional reduction 962
  - parallel mutable reduction 966
  - setting parallel execution mode 933, 934
  - shared state 1014
  - side effects 910, 948
    - non-interfering behaviors 1014
    - stateless behaviors 1014
  - splittable 1010
  - stateful operations 911, 912, 1014, 1015
  - unordered 892
- parallelism 1368
- parameter list *see* formal parameters
- parameter passing
  - by value 128
  - variable arity 136
- parameterized types 569, 607
  - as formal parameters 609
- parameters 100
  - actual 128
  - array elements 134
  - effectively `final` 400
  - `final` 135, 232, 400
  - fixed arity 136
  - formal *see* formal parameters
  - implicit 106
  - main method 141
  - passing 127
  - primitives 129
  - program 141
  - references 130
  - `this` 106
  - variable arity 136
- parent class 191
- parent directory (`..`) 1289
- parent resource bundle 1108
- parentheses 50
- `ParseException` 371
- parsing 1115
  - compact number form 1121
  - currency values 1119
  - date and time 1127
  - numbers 1119
  - percentage values 1120
  - using patterns 1150
- parsing numeric values 434
- partial application 723
- partial implementation 218, 242
- partly synchronizing a method 1391
- pass by value 128
- passing
  - parameters 127
  - references 130
  - variable arity parameter 136
- passwords
  - reading from console 1256
- path 1288
  - comparing with another path 1303
  - constructing relative path to another path 1300
  - converting 1297
  - converting to a real path 1302
  - converting to an absolute path 1298
  - creating 1290, 1291
  - interoperability with `File` class 1292
  - interoperability with `URI` class 1293
  - name elements 1288
  - name separator 1288, 1291
  - naming conventions 1288
  - normalizing 1299
  - querying 1294
  - resolving with another path 1299
- `Path` interface 1287
  - see also* path
- paths
  - see* class path
- paths of execution 1367
- path-separator character 340
- pattern letters 1127, 1134, 1135
- pattern match operator 274

- pattern matching 274
- pattern symbols 1125
- pattern variable 275
  - cannot shadow local variable 276
  - declare final 276
  - scope 275, 276
  - shadow a field 276
- patterns
  - format elements 1140
- percentage 1116, 1117
  - formatting 1116, 1119
- percentage formatter 1117, 1120
- performance 5
- period 1057
  - compared with duration 1072
  - creating 1057
  - date-based 1057, 1064
  - equality 1060
  - get methods 1059
  - immutable 1057
  - normalization 1061
  - parsing 1058
  - period-based loop 1062
  - plus/minus methods 1061
  - supported ChronoUnits 1060
  - temporal arithmetic 1061
  - text representation 1058, 1067
  - thread-safe 1057
  - with methods 1060
- Period class 1057
  - see* period
- periodic tasks 1441
- permits clause 312
- permitted direct subclasses 312
  - accessibility 314
  - contract 313
  - extending direct sealed superclass 313
  - final 313
  - locality 314
  - non-sealed 313
  - sealed 314
- permitted direct subtypes
  - contract 316
  - enum types 316
  - record classes 316
- permute elements 860
- persistence
  - see* serialization
- plain data 299
- plain data class 299
- plain data objects 299
- plain JAR 1189, 1206
- Plain Old Java Objects (POJOs) 299
- platform modules 1166
- plurals 1146
- poison value 1499
- POJOs 299
- polled locking 1462, 1467
- polymorphic method 593
- polymorphism 260, 278, 282, 518
- polynomial 759
- portability 4
- Portable Operating System Interface (POSIX) 1325
- positional order 644
- position-based access 801
- positive zero 61
- POSIX file permissions
  - enum type 1326
  - full permissions 1343
  - group 1326
  - others 1326
  - owner 1326
  - read (r), write (w), execute (x) 1325
  - string format 1326
- PosixFileAttributes interface 1330, 1332
- PosixFileAttributeView interface 1334, 1336
- PosixFilePermission enum type 1325
- PosixFilePermissions class 1326
- postfix operators 50
- power 480
- precedence rules 51
- precision 56
- predefined literals 31
- predicate 689
- Predicate functional interface 703
  - composing predicates 704
  - primitive type specializations 706
- preempted threads 1386
- preemptive scheduling 1386
- prefix 33
  - 0 33
  - 0x 33
- prepared statement 1526
  - substitutional parameterization 1526
- primitive data types 41
  - see* primitive types
- primitive types 44
  - autoboxing 430
  - unboxing 432
  - see also* primitive values
- primitive values
  - assignment 54
  - binary representation 1238

- equality 75
  - passing 129
  - principle exported package 1171
  - principle of least privilege (PoLP) 1611
  - print writers 1245
    - formatted values 1247
    - text representation of values 1246
  - printing values 24
  - PrintWriter class 1244
  - priorities 1385
  - priority
    - priority queue 816
    - priority heap 788, 816
    - priority of thread 1385
      - starvation 1412
    - priority queue 787, 788, 815
  - PriorityBlockingQueue 1496
  - PriorityQueue 815
  - private members 351
  - privileged code 1609
  - privileges 1609
  - PRNG
    - see* pseudorandom number generators 483
  - procedural programming paradigm 3
  - process of elimination 1619
  - process-based multitasking 1367
  - processes 1367
  - program
    - application 19
    - arguments 141
    - command line 141
    - compiling 20
    - correctness 1385
    - formatting 39
    - performance 1385
    - running 21
    - single-file source-code 21
  - program arguments 141
  - program counter (PC) 1370
  - program output 24
  - programmatically locking 1460
    - interruptible locking 1462, 1467
    - lock acquisition 1460
    - lock disciplines 1460
    - polled locking 1462, 1467
    - read-write locking 1466
    - reentrant 1465
    - reentrant locking 1461
    - timed locking 1462, 1467
    - unconditional locking 1467
  - programming to interfaces 647, 1196
  - proleptic year 1029
  - promotion 49
  - proper prefix 871, 872
  - properties 6
    - see also* class members
  - properties file
    - comment 1101
    - escaping whitespace 1101
    - key-value pair syntax 1100
    - key-value pairs 1100
    - properties 1100
    - property list 1100
    - text file 1101
    - see also* Properties table
    - see also* property resource file
  - Properties table 1102
  - property
    - key-value pair 1100
    - see also* properties file
    - see also* Properties table
    - see also* property resource file
  - property list 1100
    - see also* properties file
    - see also* property resource file
  - property resource file 1103
    - .properties file extension 1103
    - naming scheme 1103
    - properties 1103
    - see also* properties file
    - see also* resource bundle
  - PropertyResourceBundle 1105, 1108
  - protected members 350
  - provides-with directive 1178, 1200
  - pseudorandom generator 1422
  - pseudorandom number generators 900
    - see* Math class
    - see* Random class
  - pseudorandom numbers 482, 483
  - public 19
  - public API 1163, 1170, 1174
  - public members 350
  - punctuators 32
- ## Q
- qualified class instance creation expression 502, 503
  - qualified export 1176
    - see* exports-to directive
  - qualified name 251, 492, 496, 497, 500, 501
  - qualified opens directive
    - see* opens-to directive

- qualified superclass constructor invocation 509
- qualified this 504, 515
- query results 1533
- Queue 814
- queue 785, 788
  - capacity-restricted 815
  - FIFO 821
  - head 814
  - LinkedList 815
  - PriorityQueue 815
  - Queue 814
  - unbounded 815
- quotation mark 38, 39
- quotient 62
  
- R**
- race conditions 1014, 1389
- radix
  - prefix 33
  - see* base 431
- ragged arrays 124
- raised to power 480
- random access by index 802
- Random class 482, 1422
- random numbers 482
- random order 860
- range
  - character values 42
  - floating-point values 42
  - integer values 42
- range of values for date fields 1029
- range of values for time fields 1028
- range-view 656, 801, 810, 811, 846, 1486
- ranking criteria 644
- raw types 571, 575, 860
  - unchecked warnings 575
- read lock 1466
- read operation 541
- readability 1173
- readable 1170
- readers 1241
- Read-Evaluate-Print Loop (REPL) 23
- reading binary values 1238
- reading bytes 1234
- reading characters 1249, 1250
- reading data 1233
- ReadLock 1466
- read-write lock
  - fairness policy 1470
  - read lock 1466
- reentrant 1470
  - write lock 1466
- read-write locking 1466
- realization 244
- reclaiming memory 533
- record classes 168, 299, 300, 882
  - augmenting 304
  - basic syntax 300
  - compact canonical record constructor 304, 306
  - component fields 302
  - component list 300
  - generating methods of the Object class 303
  - generic 311
  - get methods 300, 302
  - immutable 302
  - implementing interfaces 310
  - implicit canonical record constructor 302
  - local 494, 519
  - member declarations 307
  - nested 311, 492
  - normal canonical record constructor 304, 305
  - normal non-canonical record constructor 305, 307
  - permitted direct subtypes 316
  - permitted in sealed interfaces 311
  - record construction 303
  - record elements 300
  - record header 300
  - record name 300
  - record serialization 310
  - restrictions 304
  - serialization 1263
  - shallowly immutable 302
  - static member 494
  - top-level 494
- record components 300
- record header 300
- record name 300
- recursive type bound 591, 613
- RecursiveAction 1448
- RecursiveTask 1448
- reduced graph 1167
- reducing complexity 324
- reduction operations 947
- reentrant 1465
- reentrant locking 1461
  - fairness policy 1461
- reentrant synchronization 1390
- ReentrantLock 1461

- reference types 99, 102, 195, 491
  - classes 99
  - enum types 287, 288
  - interface types 237, 246
- reference values 8
- reference variables 102
- references 8, 11, 102, 113, 127
  - abstract types 221
  - aliases 76, 130
  - array 118, 121, 260
  - assignment 55
  - casting 49, 269
  - dangling 533
  - downcasting 45
  - dynamic type 196
  - equality 76
  - field 534
  - local 534
  - narrowing conversions 45
  - null *see* null reference
  - parameterized types 569
  - passing 130
  - raw types 588
  - reachable 533, 534
  - static type 196
  - super 206, 515
  - this 106, 515
  - upcasting 45
  - widening conversions 45
- reflection 629, 1165, 1178
- Reflection API 1587
- reflective access 1191
- reifiable types 623
- relational databases 1512
- relational model 1512
- relational operators 74
- relational tables
  - columns 1512
  - constraints 1512
  - rows 1512
- relative adjusters 1040
- relative path 1288
- reliability 5
- reliable configuration 1163
- remainder 62
- remainder operator
  - % 62
- remove whitespace 453
- repeatable annotation type 1576
- replacing characters 452
- required annotation element 1562
- requires directive 1169, 1170, 1177
  - readability 1173
  - requires static directive 1177
  - requires transitive directive 1175, 1177
- reserved keywords 31
- reserved literals
  - false 32, 37
  - null *see* null reference
  - true 32, 37
- resizable arrays 787, 788, 802
- resource
  - see* resource bundle
- resource bundle 1102
  - base name 1102
  - candidate bundle name 1108
  - default resource bundle 1102, 1103, 1109, 1110
  - fallback 1110
  - key-based lookup 1104
  - key-value pairs 1102
  - locating resources 1107
  - naming convention 1102
  - parent resource bundle 1108
  - property resource file 1103
  - resource 1102
  - resource bundle class 1103
  - result bundle 1108
- resource bundle class 1103
- resource bundle family 1102
  - see also* resource bundle
- resource declaration statement 410
- resource leakage 407
- resource variable
  - final 410
- ResourceBundle 1102, 1103
- resources 407, 536
- restricted type name var 100
- result bundle 1108
- result set 1533
  - concurrency 1539, 1540, 1541
  - concurrency mode 1541
  - cursor 1534
  - customizing 1539
  - extracting column values 1536
  - fetch size 1536
  - holdability 1539, 1540, 1541
  - holdable 1540
  - metadata 1543
  - navigational direction 1539, 1540
  - scrollable 1539
  - type 1539, 1540
  - updatability 1539, 1540
- retaining elements 791
- retention policy values for @Retention 1568
- rethrowing exceptions 401

- return statement 184
    - @return tag 184
  - return type
    - covariant
  - return value 10
  - reuse of code 191, 195
  - reverse DNS 327, 1171
  - reverse-order key set 847
  - reverse-order view 847
  - rhyming order 771
  - right associativity 52
  - rightmost bit 34
  - ripple effect 195
  - robustness 5, 416, 1607
  - role relationship 195
  - root
    - see* inheritance hierarchy
  - root component 1287
  - root node 1287
  - round method 479
  - rounding functions 478
  - rounding mode 1122
  - rounding numbers 479
  - round-robin scheduling 1386
  - rows 1512
  - Runnable functional interface 1370, 1371, 1376, 1424
  - RUNNABLE state 1382
  - Running 1203
  - running a Java application 21
  - running from a modular JAR 1191
  - running from a runtime image 1205
  - runtime
    - bounds checking 121
    - runtime checks 48, 648
    - Runtime class 425, 537
    - runtime class 426
    - runtime constant pool 1370
    - runtime data areas 1369
    - runtime environment 533, 1100
    - runtime image 3, 1204
    - runtime stack
      - see* JVM stack
    - runtime type
      - see* dynamic type
  - RuntimeException 372, 397
- S**
- scheduled executor service
    - scheduling one-shot tasks 1440
    - scheduling periodic tasks 1441
  - scheduled thread pool 1427
  - ScheduledExecutorService 1440
  - schedulers 1386
  - schema 1293
  - scientific notation 36
  - scope 352
    - block 354
      - catch clause 381, 410
      - class 352
      - disjoint 355
    - scope of local variables 324
  - sealed classes 312
    - direct sealed superclass 313
    - in compilation unit 317
      - permits clause 312
      - permitted direct subclasses 312, 313
  - sealed interfaces
    - direct sealed superinterface 315
    - in compilation unit 317
      - permits clause 315
      - permitted direct subtypes 315
  - searching 783, 856
    - arrays 866
    - collections 861
  - searching in string 451
  - searching in string builder 466
  - secure 5
  - secure coding 1600
  - security guidelines 1610
  - security policies 1608
  - security threats 1602
  - SecurityManager class 425
  - seed 484
  - SELECT statement 1515
  - selection statements 152
  - selector expression 160
  - self-reference 146
  - self-referential data structure 566
  - semantic definition 30
  - semicolon 101
  - sensitive information leakage 1603
  - separators 32, 51
  - sequence 784
  - sequential I/O 1233
  - sequential streams 891
    - execution 958
    - extracting lines from a String 902
    - from a BufferedReader 902
    - from a CharSequence 901
    - from arrays 898
    - from collections 897
    - from factory methods in the Files class 903
    - ordered 891
    - unordered 891

- serial access 1478
- serialization 1261
  - customizing 1268
  - inheritance 1270
  - serialVersionUID 1275
  - transient fields 1267
  - versioning 1273
- serialVersionUID 1275
- service 1178, 1196
- service consumer 1178, 1197, 1202
- service interface 1196, 1197
- service loader 1197, 1200
- service locator 1178, 1197, 1200
- service provider 1178, 1197, 1199
- service provider interface
  - see* service interface
- ServiceLoader class 1197
- ServiceProvider.Provider interface 1200
- services 324
- Set 804
- set 754, 784
  - capacity 807
  - HashSet 806
  - LinkedHashSet 806
  - load factor 807
  - NavigableSet 810
  - Set 804
  - SortedSet 810
  - TreeSet 812
  - unmodifiable 804, 805
  - unmodifiable views 856
- set methods 358
- set operation 586
- setter methods *see* set methods
- SHA-256 digest algorithm 1603
- shadowing fields by local variables 114
- shadowing local variables 514
- shadowing members 685
- shallow copying 426
- shallowly immutable 302
- shared field 1453
- shared memory 1369
- shift
  - compound operators 90
  - distance 86
  - left 86
  - operators 86
- shifting 86
- short 32, 42
- short cut
  - see* symbolic link
- short-circuit 80
  - evaluation 80
- short-circuit execution 908, 947, 952
- shuffle elements 860
- sign bit 86
- sign fill 86
- signature 108, 202, 616
- signed shift 86
- simple
  - assignment operator 54
  - if 152
  - statement 101
- simple name 345
- simple type name 326
- SimpleFormat 1115
- simplicity 4
- single implementation inheritance 193
- single inheritance of implementation 193
- single quote (') 37, 38
- single source file 22
- single static import 331
- single thread executor 1426
- single thread scheduled executor 1427
- single-element annotation 1563, 1565
  - array-valued single-element annotation 1566
  - optional annotation element 1566
  - single-element array-valued single-element annotation 1566
  - value() element 1565
- single-element annotation type 1560
- single-file source-code program 21
- single-line comment 7, 40
- singleton 1256
- single-valued maps 830
- sink nodes 1172
- skeletal source file 325
- skip list 1492
- sleeping 1395
- software security 1600
- sorted map 785
  - comparator 845
- sorted set 784, 810, 811
  - comparator 811
  - half-open interval 810
- SortedMap 845
- SortedSet 810
- sorting 783, 856, 929
- sorting arrays 865
- sorting lists 858
- source
  - file 20, 328
  - file name 325
  - file structure 325
- source directory 1219

- source of data 1233
- SP *see* space
- space 38
- spaces 39
- spawning threads 1378
- special character values 38
- specialization 13
- specialized classes 194
- spurious wakeup 1399
- SQL 1514
- SQL injection 1525, 1605
- SQL statement API
  - basic statement 1523
  - callable statements 1530
  - prepared statement 1526
- SQL statements 1522
- SQLException 371, 1518
- square root 480
- stability guarantee 911, 930
- stack
  - generic 598
  - LIFO deque 821
- stack frame
  - see* method execution
- stack trace 367, 370, 415
  - see* method execution
- StackOverflowError 374
- stacks 815
- standard error stream 370, 1255
- standard input stream 1255
- standard modules 1166
  - JMOD archive 1166
- standard out 24
- standard output stream 1255
- standard time 1082
- StandardCopyOption enum type 1308
- starting threads 1371
- starvation 1412
- state *see* object state
- stateful operations 908
- stateless behavioral parameters 910
- stateless operations 908
- statement block 682
- statement terminator 1514
- statements 101
  - break 180
  - compound 101
  - conditional 152
  - continue 182
  - control flow 101
  - control transfer 179
  - declaration 65, 70, 81, 101
  - do-while 173
  - empty 101
  - expression 70, 101
  - for(;;) 174
  - for(:) 176
  - if 152
  - if-else 153
  - iteration 172
  - labeled 179
  - resource declaration 410
  - return 184
  - selection 152
  - simple 101
  - simple if 152
  - switch statement 155, 160
  - synchronized 1391, 1392
  - throw 386
  - transfer 179
  - try 377
  - try-with-resources 407
  - while 172
  - yield 164
- static
  - members *see* static members
  - methods 11, 13, 101
  - variable initialization 103
  - variables *see* static variables
- static 10, 251, 331
- static code
  - see* static context 99, 100
- static context 99, 100
- static factory method 359, 1116
- static field initializers 99
- static fields 13, 113
- static import 331
  - conflicts 334
  - on demand 331
  - shadow static members 333
  - single static import 331
- static initializer blocks 99, 545
  - declaration order 545
  - exception handling 549
  - local variable type inference 545, 547
- static keyword 113, 545
  - static fields 113
  - static methods 115
- static local enum types 492, 494
- static local interfaces 492, 494
- static local nested types
  - see* static local types 491
- static local record classes 492, 494
- static local types 491
  - enum types 492, 494
  - interfaces 492, 494

- record classes 492, 494
  - see* local enum types
  - see* local interfaces
  - see* local record class
- static member classes 494, 495
  - class file name 497
  - declaration 495
  - direct access 498
  - fully qualified name 497
  - instantiation 497
  - qualified name 496
  - UML class diagram 498
- static member enum types 494
- static member interfaces 494
- static member record classes 494
- static member types 491
  - static member class 491
  - static member enum type 492
  - static member interface 491
  - static member record class 492
- static members 10, 11, 13
- static method references 726
- static methods 115
  - hiding from superclass 203
- static nested classes
  - see* static member types
- static nested enum types
  - see* static member enum types
- static nested interfaces
  - see* static member interfaces
- static nested record class
  - see* static member record class
- static nested types
  - see* static member types
- static type 196
- static variables 11, 13, 106
- stored functions 1530
  - IN parameter 1530
  - INOUT parameter 1532
  - OUT parameter 1530
- stored procedures 1530
  - IN parameter 1530
  - INOUT parameter 1532
  - OUT parameter 1530
- stream 798
  - transformations 905
- Stream API 889
  - BaseStream interface 889
  - building streams 890
  - Collector interface 890
  - Collectors 890
  - DoubleStream interface 889
  - IntStream interface 889
  - LongStream interface 889
  - Stream interface 889
- Stream interface 889
- stream mapping 906
- stream operations 881
  - non-interfering behavioral parameters 910
  - stateless behavioral parameters 910
- stream pipelines
  - composing 886
  - executing 887
- streams 881
  - see also* I/O streams
  - aggregate operations 881
  - building from a `BufferedReader` 902
  - building from a `CharSequence` 901
  - building from a range 898
  - building from arrays 898
  - building from collections 897
  - building from directory entries 1345
  - building from specified values 893
  - building from the `Random` class 900
  - building using factory methods in the `Files` class 903
  - collecting into a list 972
  - collecting into an array 971
  - compare with collections 888
  - concatenating 895
  - conversion 934
  - counting elements 953
  - data source 884
  - dropping elements 910, 913
  - empty 893
  - encounter order 891
  - examining elements 920
  - execution mode 933
  - extracting lines from a `String` 902
  - filtering 910
  - finding first/any 952
  - finding min/max 954
  - finite 892
  - flattening 924
  - functional reduction 953, 954, 955, 964, 972
  - infinite 892
  - intermediate operations 885, 905, 908
  - lazy execution 907
  - mapping 905, 921
  - map-reduce transformations 905
  - matching operations 949
  - method chaining 906

- mutable reduction 947, 964
  - see also* collectors
- numeric 885, 892
- numeric summary statistics 972
- object 885, 892
- operations with side effects 946, 948
- ordered 891
- parallel streams 891, 962, 966, 1009
- reduction operations 947, 964
- replacing each element with multiple elements 927
- searching operations 947
- selecting distinct elements 915
- sequential 891
- sequential execution 958
- setting as unordered 932
- short-circuit execution 908, 947, 952
- skipping elements 915
- sorting 929
- stability guarantee 911, 930
- stateful operations 908
- stateless operations 908
- Stream API 889
- stream mapping 906
- stream operations 881
- stream pipelines 886
- taking elements 910, 913
- terminal stream operations 885, 946
- truncating 917
- unordered 891
- strictfp 58
- string builders 69
  - appending 467
  - capacity 464, 470
  - compareTo 467
  - construction 465
  - copying characters 465
  - deleting 468
  - extracting substrings 466
  - individual characters 465
  - inserting 468
  - joining 450
  - length 465
  - mutable 464
  - reading characters 465
  - replacing 468
  - reversing 469
  - searching 466
  - setting length 470
  - substrings 466
  - trimming 470
- String class
  - see* strings
- string conversion 46, 68, 455
- string literal pool 440
  - interned 440
- string literals 440
  - case labels 162
  - hash value 162
  - interned 440
- string pattern 1125
- StringBuffer legacy class
  - thread-safe 464
- StringBuilder class
  - see* string builders
- strings
  - capacity 470
  - changing case 448
  - compact 440
  - compareTo 447
  - comparing 444, 447, 761
  - concatenation 67, 448
  - concatenation operator + 69
  - contains 452
  - conversions 455
  - convert to character array 445
  - copying characters 445
  - creating 440
  - empty 441, 444
  - ends with 452
  - equals 447
  - extracting lines 902
  - extracting substrings 453
  - finding index 451
  - formatted 457
  - ignoring case in comparison 447
  - immutable 440
  - indenting lines 456
  - individual characters 445, 465
  - initializing 440
  - integer stream 444
  - internal representation 439
  - interned 440, 442
  - joining 450, 843, 984
  - length 444, 445
  - lexicographical ordering 447
  - literals 39, 440
  - natural order 761
  - read character at index 444, 445
  - repeating 449
  - replacing 452
  - searching 451
  - selector expression 156
  - starts with 452
  - streams 901, 902
  - string literal pool 440

- stripping 453
- substrings 453
- text blocks 458
- thread-safe 440
- transforming 455
- trimming 453
- strongly typed language 48
- Structured Query Language 1514
- style-based formatter for dates and times 1131
- subclass 13, 15, 191
- subdirectories 1287
- subformat pattern 1146
- subinterface 244
- sublist 656
- submap view 845, 846
- sub-maps 1491
- subsequence 444
- subset 804, 808, 810, 811, 1486
- subsignature 617
- substitutional parameterization 1526
- substring searching 451, 466
- substrings 451, 453, 466
- subtags 1098
- subtype bivariance 581
- subtype contravariance 581
- subtype covariance 259, 260, 580, 581
- subtype invariance 581
- subtype notation <: 584
- subtype relationship 648
- subtypes 242, 278, 581, 592
- subtype-supertype relationship 45, 195
- suffix
  - D 36
  - F 36
  - L 32
- super 248
  - construct 211
  - keyword 201, 206
  - reference 206
- superclass 13, 191
- superclass constructor 110
- superinterfaces 244
- superset 808
- supertypes 242, 278, 582
- Supplier functional interface 699
  - primitive type specializations 700
- suppressed exceptions 370, 415
- suppressed warnings
  - deprecation warnings 1583
  - unchecked warnings 1583
- switch block
  - local variable scope 169
- switch rules 160
- switch expression 164
  - arrow notation 166
  - case labels 164
  - colon notation 164
  - comparison with switch statement 169
  - default clause 166
  - exhaustive 165, 166, 167
  - fall-through 165
  - local variable scope 169
  - return multiple values 168
  - yield statement 164, 166
- switch rules 160
  - mutually exclusive 161
- switch statement 155
  - arrow notation 160
  - break 157
  - case labels 156
  - colon notation 155, 157
  - comparison with switch expression 169
  - default clause 156, 161
  - enum types 163
  - execution 156
  - exhaustive 163
  - fall-through 156
  - local variable scope 169
  - selector expression 160
  - using strings 162
- switching between threads 1368
- symbolic link 1289, 1304
- synchronization 1387
- synchronized
  - keyword 1388
  - methods 1388
  - starvation 1412
  - statements 1391, 1392
  - vs. volatile 1456
- synchronized block 1391
- synchronized collections 1477
  - compound mutually exclusive operations 1480
  - serial access 1478
  - underlying collection 1477
- synchronized method 1388
- synchronized views of collections
  - see synchronized collections
- SynchronousQueue 1496
- syntactically legal 30
- System 425
  - out 24
  - reassigning standard streams 1255
  - standard streams 1255
- system clock 1030

system default zone ID 1074  
 system modules 1212

## T

TAB *see* horizontal tab

table definition 1513

tabs 39

tabulators 39

tagging interface

*see* marker interfaces

tags 40

tail map view 845, 846

tail set view 810, 811, 1486

taking stream elements 910, 913

target directory 1187

target parallelism level 1426

target reference 724

target type 570, 682

target typing 684

task 1367, 1423

task cancellation 1444

task execution policy 1426

TCL 1514

telephone directory order 447

temporal arithmetic 1040, 1054, 1061, 1069

temporal fields 1027

temporal objects

access specific temporal values 1032, 1077

before/after methods 1034

combining date and time 1030

common methods 1026

comparing 1034

conversion to Date 1088

creating 1027, 1075

date 1024

date fields 1024

date units 1040

date-time 1024

fields 1024, 1025

formatting 1127

get methods 1032, 1077

immutable 1024, 1075

method naming convention 1026

parsing 1127

plus/minus methods 1040, 1081

range of values for date fields 1029

range of values for time fields 1028

temporal arithmetic 1040, 1061, 1062,

1071, 1080

temporal fields 1027, 1036

thread-safe 1024, 1075

time 1024

time fields 1024

time units 1040

with methods 1035, 1078

zoned date-time 1024, 1074

TemporalAmount interface 1061

*see also* Duration

*see also* Period

TemporalField interface

*see also* ChronoField

TemporalUnit interface

*see also* ChronoUnit

terminal stream operations 885, 946

averaging 974

collecting into a list 972

collecting into an array 971

counting elements 953

eager execution 887

finding first/any 947, 952

finding min/max 954

functional reduction 947, 953, 954, 955,  
 972

accumulator 956, 957, 962

combiner 956, 962

identity value 956, 957, 962

parallel execution 962

sequential execution 958

matching operations 947, 949

mutable reduction 947, 964, 971, 972

accumulator 965

combiner 965

parallel stream execution 966

sequential stream execution 965

supplier 965

*see also* collectors

operations with side effects 946, 948

reduction operations 947, 964

*see also* collectors

searching operations 947

short-circuit operations 947, 952

summary statistics 972, 974

summation 973

TERMINATED state 1382, 1405

terminating loops 180

ternary conditional expressions

*see also* conditional expressions 90

ternary conditional operator 50, 90

text block 458

closing delimiter 458

content 458

escape sequences 459

essential whitespace 462

- incidental whitespace 462
- indentation 462
- opening delimiter 458
- text files
  - reading from 1249
  - stream to read lines 1346
  - writing to 1247
- text representation 426
- this
  - reference 106
- this keyword
  - qualified form 504
  - with new 503
- this() constructor call 209, 555
- thousands separator 1125
- Thread class 1370
- thread coordination 1396
- thread execution
  - runtime organization 1369
- thread interference 1373, 1389
- thread interruption rule 1415
- thread lifecycle 1380
- thread of execution 1367
- thread pool 1423, 1425
- thread safety 1368
- Thread start rule 1414
- Thread termination rule 1414
- thread-based multitasking 1367
- thread-local variable 1453
- ThreadLocalRandom 1422
- threads 5, 425, 534, 1402
  - acquire lock 1388
  - alive 1377, 1383
  - BLOCKED state 1382
  - blocking operation 1405
  - child 1373, 1378
  - class 1370
  - code executed 1371
  - controlling termination 1405
  - creation 1370
  - current thread 1372
  - daemon 1377
  - deadlocks 1408
  - death 367
  - deprecated methods 1408
  - determining state 1381
  - entry set for 1380
  - exception propagation 367
  - extending Thread 1375
  - fairness 1408
  - happens-before relationship 1414
  - IllegalThreadStateException 1378
  - interrupt status 1393
  - interrupted 1399
  - InterruptedException 1393
  - joining 1383, 1402
  - JVM stack 534
  - live 534
  - livelock 1410
  - liveness 1408
  - main thread 1373, 1378
  - names 1372
  - NEW state 1382, 1386
  - non-observable substate 1382
  - non-runnable state 1383
  - normal 1377
  - notification 1396, 1398
  - notifying 427, 1396
  - Object lock rule 1414
  - priorities 1385
  - priority 1383
  - run() method 1372
  - Runnable 1370, 1371, 1376
  - RUNNABLE state 1382
  - running 1375
  - running thread 1383
  - scheduler 1385
  - shared field 1453
  - shared memory 1369
  - sleeping 1383, 1395
  - spawning 1378
  - spurious wakeup 1399
  - start() method 1372
  - starting 1371, 1375, 1386
  - starvation 1412
  - states 1381
  - switching 1368
  - synchronization 1387
  - synchronized 1388
  - synchronized method 1388
  - tasks 1423
  - TERMINATED state 1382, 1405
  - termination 1405
  - Thread class 1370
  - thread coordination 1396
  - Thread interruption rule 1415
  - thread lifecycle 1380
  - Thread start rule 1414
  - thread status 1378
  - Thread termination rule 1414
  - thread-local 1453
  - TIMED\_WAITING state 1382
  - transitions 1381
  - Volatile field rule 1414, 1454
  - wait set for 1397
  - waiting 427, 1396

- WAITING state 1382
- yielding 1383, 1386
- thread-safe 464, 645, 789, 1024, 1075
  - immutable classes 356
- thread-safe code 1387, 1451
- thread-safe collection 1477
- thread-safety 356
  - atomic variables 1456
  - immutability 1453
  - intrinsic locking 1460
  - programmatic locking 1460
  - synchronized collections 1477
  - unmodifiable collections 1477
  - volatile fields 1453
- throw statement 386
- Throwable 368, 405, 425
- throw-and-catch paradigm 365
- throwing exceptions 365
- throws clause 388
  - and rethrowing 402
- time fields 1024
- time gap 1082
- time overlap 1082
- time standard 1073
- time units 1040, 1421
- time zone 1024
  - local time 1073
  - see* zone ID
- time zone offset 1073
- time zone rules 1074, 1082, 1084
- timed locking 1462, 1467
- TIMED\_WAITING state 1382
- timeline 1024
  - instant 1074
  - local 1074
- time-sliced scheduling 1386
- timestamps 1049
- TimeUnit 1421
- ToDoubleBiFunction interface 718
- ToDoubleFunction interface 712
- ToIntBiFunction interface 718
- ToIntFunction interface 712
- tokens 30
- ToLongBiFunction interface 718
- ToLongFunction interface 712
- top-level classes 491, 494
- top-level enum types 491, 494
- top-level interfaces 491, 494
- top-level record classes 491, 494
- top-level types 491
- toString method 426, 431, 432
- total order 761
- total ordering 865
- Transaction Control Language (TCL) 1514
- transactions 1545
  - auto-commit mode 1545
  - commit 1545
  - rollback 1545
  - savepoint 1545
- transfer statements 179
- TransferQueue 1496, 1498
- transforming strings 455
- transient fields 1267
- transitive relation 194
- transitive relationship 1414
- tree 1287
  - traversal 1347
- tree map 847
- tree set 812, 847
- TreeMap 847
- trees 788
- TreeSet 812
- trim method 453
- true literal 32, 37
- truth-values 37, 43
- try block 377
- try-catch-finally construct 375
- try-with-resources 147, 407, 1519
  - basic 410
  - concise syntax 411
  - extended 410
  - header 410
  - var in header 410
- tuples 1512
- two's complement 33, 34, 86
- two-arity 695
- type annotation 1572
- type cast 48
- type cast expression 269
- type checking 682
- type comparison operator 270
- type contexts 1572
- type declarations 325
  - nested types 491
  - top-level types 491, 494
- type erasure 576, 613, 622
- type hierarchy 45, 195
- type import
  - see* import
- type inference 142
- type information 565
- type instance 569
- type parameter constraints 591
- type parameters 633
  - actual 569
  - bounds 591

- formal 567
- recursive bounds 611
- restrictions 568
- type pattern 275
- type variables 567
- type witness 594, 595, 928
- types
  - arrays 579
  - boolean 41, 43
  - byte 32, 42
  - casting 48
  - char 42
  - classes *see* classes
  - comparison 270
  - compatibility 48
  - double 42
  - exceptions 368
  - float 42
  - floating-point 41, 42
  - int 32, 42
  - integers 41
  - integral types 41
  - long 32, 42
  - numeric types 41
  - parsing 434
  - short 32, 42
  - unrelated 570
  - wrappers 429
  - see also* classes
- type-safe 264
- type-safety 646, 648
- `typeValue` method 432, 434

## U

- UML 6
  - accessibility 348
  - aggregation 16
  - classes 6
  - inheritance 14
  - see also* diagrams
- UML class diagram
  - non-static member classes 508
  - static member classes 498
- UML notation for members 348
- UN M.49 three-digit area code 1096
- unary arithmetic operators 61
- unary numeric promotion 49
- unary operators 50
- `UnaryOperator` interface 720
  - composing unary operators 720
  - primitive type specializations 721
- unbounded collections 1483
- unbounded instance method references 729
- unbounded type parameter 581, 583
- unbounded type references 590
- unbounded wildcard
  - ? 581, 582
- unbounded wildcard references 588
- unboxing 432
  - do-while statement 173
  - for(;;) statement 174
  - for(:) statement 177
  - if statement 152
  - if-else statement 153
  - switch statement 156
  - while statement 173
- unboxing conversions 45
- uncaught exceptions 367
- unchecked 627, 630
- unchecked array creation warning 630
- unchecked call warning 576, 588
- unchecked cast warning 625, 627
- unchecked conversion warning 571, 576, 585, 627, 646
- unchecked conversions 46
- unchecked exceptions 374
- unchecked generic array creation warning 630, 650, 659, 805, 833, 863
- unchecked warnings 45, 575, 576, 1583
  - `@SuppressWarnings("unchecked")` 627
  - call warning 576, 588
  - cast warning 625
  - conversion warning 576
  - generic array creation warning 630
  - heap pollution warning 630, 1585
  - unchecked generic array warning 1585
- unconditional locking 1467
- underflow 34, 61
- underlying collection 1477
- uni-catch clause 376, 397
  - effectively final parameter 400
- Unicode 37, 42, 437, 447, 1241
- Unicode character encoding 1101
- Unicode locale/language extension 1118
- unidirectional edges 1166
- Unified Modeling Language *see* UML
- Uniform Resource Identifier (URI) 1290
- union 804
- union of alternatives 399
- unmodifiable 660
- unmodifiable collections 649, 805, 857, 1477
- unmodifiable lists 649, 660, 801
- unmodifiable maps 832

- unmodifiable sets 805
  - unmodifiable view of a collection 856, 857
  - unmodifiable view of a list 856
  - unmodifiable view of a map 857
  - unmodifiable view of a navigable map 857
  - unmodifiable view of a navigable set 856
  - unmodifiable view of a set 856
  - unmodifiable view of a sorted map 857
  - unmodifiable view of a sorted set 856
  - unmodifiable views 856, 857
  - unmodifiable views of collections 1477
  - unnamed module 314, 1206
  - unnamed package 314
  - unordered streams 891
    - from the `Random` class 900
  - unqualified export 1178
    - see* `exports` directive
  - unreachable code 381, 401
  - unsafe casts 271
  - unsigned integer 435
  - unsupported operations 790, 831
  - `UnsupportedOperationException` 373, 794, 797
  - `UnsupportedTemporalTypeException` 1033, 1062, 1068, 1070, 1075, 1131
  - unused keywords 31
  - upcasting 45
  - updatability 1539
  - update expression 174
  - `UPDATE` statement 1515
  - upper bound 581, 591
  - upper bounded wildcard
    - ? extends `T` 581
  - upper bounded wildcard references 589
  - `URI` class 1290, 1293
    - schema 1293
  - user threads
    - see* normal thread
  - `uses` directive 1178, 1202
  - using arrays 120
  - using packages 329
  - using variables 102
  - UTC (Coordinated Universal Time) 1073
  - UTC/Greenwich 1073
    - Z 1073
  - UTF-16 encoding scheme 439, 440
  - utility class 115, 251, 360
- V**
- value collection 831
  - value obfuscation 1603
  - value objects 785
  - `value()` element 1560, 1561, 1565
  - `valueOf` method 430, 431, 455
  - values 34, 830
    - constants 32
    - overflow 34
    - underflow 34
    - wrap-around 34
    - see also* variables
  - `var` 571
  - var declaration
    - see* local variable type inference
  - varargs
    - variable arity 136
  - varargs method
    - see* variable arity method
  - varargs parameter
    - see* variable arity parameter
  - variable arity argument 659, 863
  - variable arity call 139
  - variable arity method 136
  - variable arity parameter 136, 630, 863, 1304
    - generic array creation 630
    - heap pollution 630
  - variable capture 687
  - variable declarations 102, 354
  - variable initialization 11, 102, 104
  - variables 8, 102
    - blank final 135
    - blank final local 231, 232
    - constant values 229
    - default values 103
    - effectively final 687
    - final 229
    - identifiers 102
    - in interfaces 254
    - initialization *see* variable initialization
    - lifetime 106
    - local 142, 354
    - parameters 100, 128
    - reference variable 102
    - references 102
    - static 11
  - var-type inferred parameters 680
  - `Vector` 802
  - vector 802
  - Venn diagrams 791
  - versioning 1273
  - view of a list 656, 660
  - viewing dependencies 1214
    - class-level dependencies 1216
    - module-level dependencies 1216
    - package-level dependencies 1214

- views 810, 839
- virtual method invocation
  - see* dynamic method lookup
- Virtual Private Network (VPN) 1601
- VirtualMachineError 374
- visibility 1454
- visibility modifiers
  - see* access modifiers
- void 19, 184, 429
- void return 681
- Void wrapper class 429
- volatile
  - vs.* synchronized 1456
- Volatile field rule 1414, 1454
- volatile fields 1453
  - visibility 1454
- volume name 1287

## W

- wait set 1396, 1397
  - starvation 1412
- waiting 1396
  - timeout 1399
- WAITING state 1382
- waiting threads 427
- WHERE clause 1515, 1516
- while statement 172
- whitespace 39, 453, 454
- whole-part relationship 194
- widening conversions
  - primitive 44
  - references 45
- widening reference conversions 196, 270, 584
- wider range 44
- wildcard parameterized types 584
  - type hierarchy 581, 582, 583
- wildcard types 580
  - see* wildcards
- wildcards 579, 580
  - ? 588
  - ? extends T 589
  - ? super T 589
  - assignment 584
  - bounded 581
  - capture 604
  - capture conversion 606
  - nested 607
  - parameterized types 584
  - restrictions 584
  - subtype relationships 579
  - type hierarchy 581, 582, 583

- withers 1035
- work stealing pool 1426
  - target parallelism level 1426
- worker threads 1423
- wrapper 1233
- wrapper classes 41, 425, 426, 429, 744, 754, 761
  - interned values 433
- wrapper type 45, 46, 59, 70
- write lock 1466
- write operation 542
- WriteLock 1466
- writers 1241
- writing binary values 1238
- writing bytes 1234
- writing characters 1238, 1249, 1253
- writing data 1233

## X

- xor 83

## Y

- yield statement 164, 166
- yielding 1386

## Z

- z 1073
- zero
  - negative 61
  - positive 61
- zero fill shift 86
- zero-arity 695
- zero-based index 644, 801
- zone ID 1073
  - available zone IDs 1073
  - system default 1074
- zoned date-time
  - accessing time zone 1077
  - accessing time zone offset 1077
  - conversion to Date 1088
  - converting to different time zone 1079
  - converting to instant 1078
  - see also* daylight saving time (DST)
  - see* temporal objects
- ZonedDateTime class 1072, 1074
  - see* temporal objects
  - see* zoned date-time
- ZoneId class 1072
- ZoneOffset class 1072