ORACLE PRESS

Also Covers **Java SE 11** Developer Exam

Java™

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

Pearson

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

Library of Congress Control Number: 2022951639

ISBN-13: 978-0-13-799364-2
ISBN-10: 0-13-799364-1

ScoutAutomatedPrintCode

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

Contents

This page intentionally left blank

Figures

Tables

Examples

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

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 *Java SE 17* and *Java SE 11 Developer Exams* 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 *not* 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
```

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`. 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

Streams 16

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 ❍ *Streams are covered in this chapter.* ❍ *For lambda expressions implementing functional interfaces, see Chapter 13, p. 673.*	*§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, \"%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.

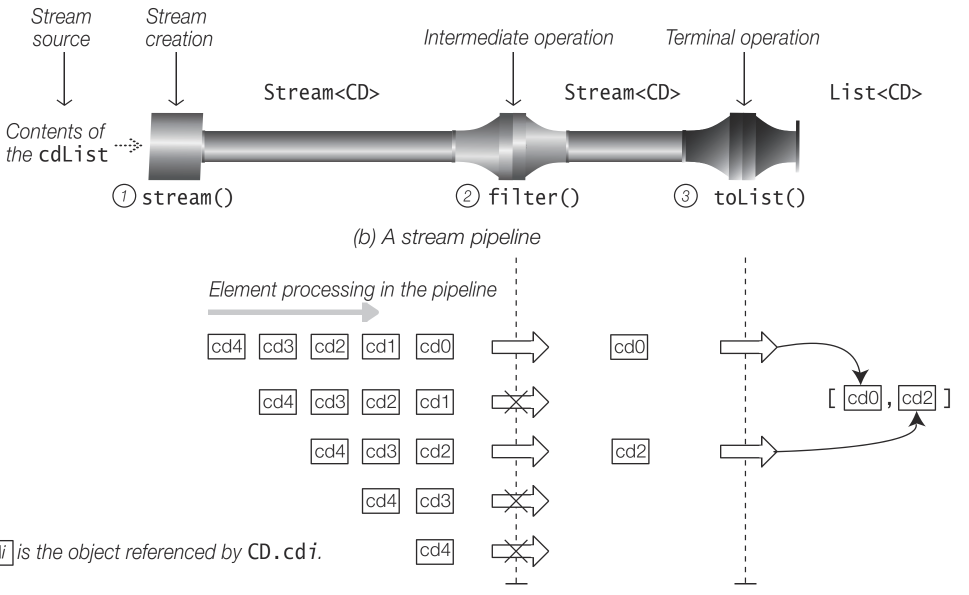
An example of an aggregate operation is filtering the elements in the stream according to some criteria, where all the elements of the stream must be examined in order to determine the result. We saw a simple example of filtering a list in the previous section. Figure 16.1a shows an example of a *stream-based* 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



(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.

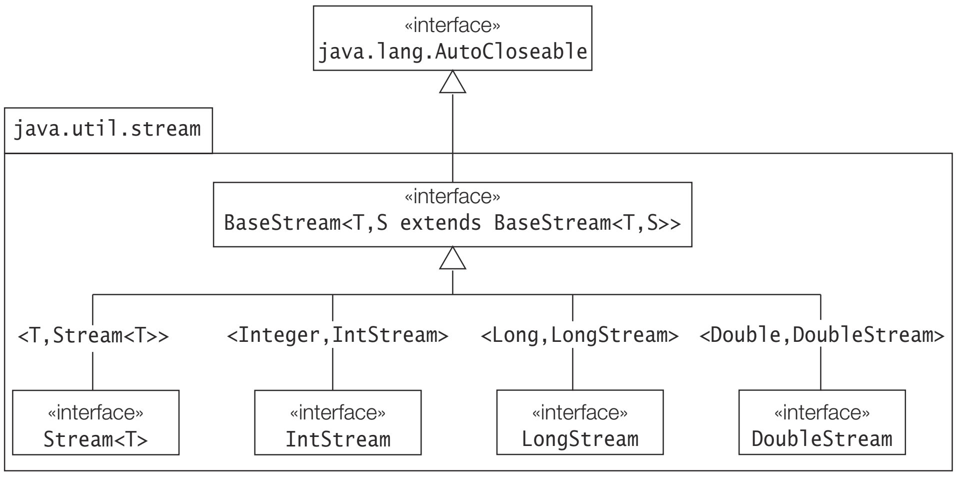
The Stream Interfaces

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 `DoubleSummary-Statistics` 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 `NullPointer-Exception`. 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 printout 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
```

Streams from Collections

The `default` methods `stream()` and `parallelStream()` of the `Collection` interface create streams with collections as the data source. Collections are the only data source that provide the `parallelStream()` method to create a parallel stream directly. Otherwise, the `parallel()` intermediate operation must be used in the stream pipeline.

The following `default` methods for building streams from collections are defined in the `java.util.Collection` interface:

```
default Stream<E> stream()
default Stream<E> parallelStream()
```

Return a *finite sequential* stream or a possibly *parallel* stream with this collection as its source, respectively. Whether it is ordered or not depends on the collection used as the data source.

We have already seen examples of creating streams from lists and sets, and several more examples can be found in the subsequent sections.

The code below illustrates two points about streams and their data sources. If the data source is modified before the terminal operation is initiated, the changes will be reflected in the stream. A stream is created at (2) with a list of CDs as the data source. Before a terminal operation is initiated on this stream at (4), an element is added to the underlying data source list at (3). Note that the list created at (1) is modifiable. The `count()` operation correctly reports the number of elements processed in the stream pipeline.

```
List<CD> listOfCDS = new ArrayList<>(List.of(CD.cd0, CD.cd1));       // (1)
Stream<CD> cdStream = listOfCDS.stream();                             // (2)
listOfCDS.add(CD.cd2);                                                // (3)
System.out.println(cdStream.count());                                 // (4) 3
// System.out.println(cdStream.count());           // (5) IllegalStateException
```

Trying to initiate an operation on a stream whose elements have already been consumed results in a `java.lang.IllegalStateException`. This case is illustrated at (5). The elements in the `cdStream` were consumed after the terminal operation at (4). A new stream must be created on the data source before any stream operations can be run.

To create a stream on the entries in a `Map`, a collection view can be used. In the code below, a `Map` is created at (1) and populated with some entries. An entry view on the map is obtained at (2) and used as a data source at (3) to create an unordered sequential stream. The terminal operation at (4) returns the number of entries in the map.

```
Map<Integer, String> dataMap = new HashMap<>();                       // (1)
dataMap.put(1, "en"); dataMap.put(2, "to");
dataMap.put(3, "tre"); dataMap.put(4, "fire");
long numOfEntries = dataMap
    .entrySet()                                                       // (2)
```

```
      .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* *`NumType`*`Stream` (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* *NumType*`Stream` 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

`Stream.empty()`
`NumType``Stream.empty()`
`(p. 893)`
	`Stream<T>`
	`NumType``Stream`
		Finite	*Sequential*	*Ordered*

`Stream.of()`
`Stream.ofNullable()`
`NumType``Stream.of()`
`(p. 893)`
	`Stream<T>`
	`Stream<T>`
	`NumType``Stream`
		Finite	*Sequential*	*Ordered*

`Stream.generate()`
`NumType``Stream.generate()`
`(p. 895)`
	`Stream<T>`
	`NumType``Stream`
		Infinite	*Sequential*	*Unordered*

`Stream.iterate()`
`NumType``Stream.iterate`
`(p. 895)`
	`Stream<T>`
	`NumType``Stream`
		Infinite	*Sequential*	*Ordered*

`Stream.concat()`
`NumType``Stream.concat()`
`(p. 895)`
	`Stream<T>`
	`NumType``Stream`
		Finite if both finite	*Parallel if either parallel*	*Ordered if both ordered*

`Collection.stream()`
`(p. 897)`
	`Stream<T>`	*Finite*	*Sequential*	*Ordered if collection ordered*

`Collection.parallelStream()`
`(p. 897)`
	`Stream<T>`	*Finite*	*Parallel*	*Ordered if collection ordered*

`Arrays.stream()`
`(p. 898)`
	`Stream<T>`
	`NumType``Stream`
		Finite	*Sequential*	*Ordered*

`IntStream.range()`
`IntStream.rangeClosed()`
`LongStream.range()`
`LongStream.rangeClosed()`
`(p. 898)`
	`IntStream`
	`IntStream`
	`LongStream`
	`LongStream`
		Finite	*Sequential*	*Ordered*

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`
		Finite or infinite, depending on parameters	*Sequential*	*Unordered*

`CharSequence.chars()`
`(p. 901)`
	`IntStream`	*Finite*	*Sequential*	*Ordered*

`String.lines()`
`(p. 902)`
	`Stream<String>`	*Finite*	*Sequential*	*Ordered*

`BufferedReader.lines()`
`(p. 902)`
	`Stream<String>`	*Finite*	*Sequential*	*Ordered*

`Files.lines()`
`(p. 903)`
	`Stream<String>`	*Finite*	*Sequential*	*Ordered*

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.

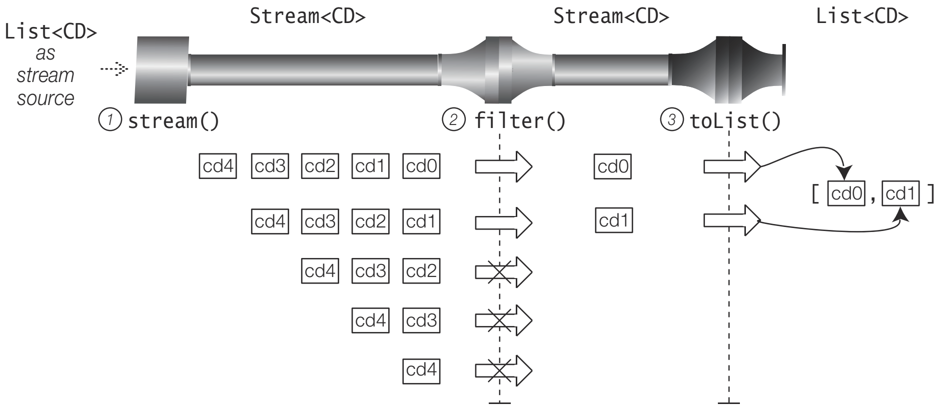
```
List<CD> popularCDs2 = new ArrayList<>(CD.cdList);
popularCDs2.removeIf(cd -> !(popularTitles.contains(cd.title())));
System.out.println("Query 1b: " + popularCDs2);
//Query 1b: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ>]
```

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.

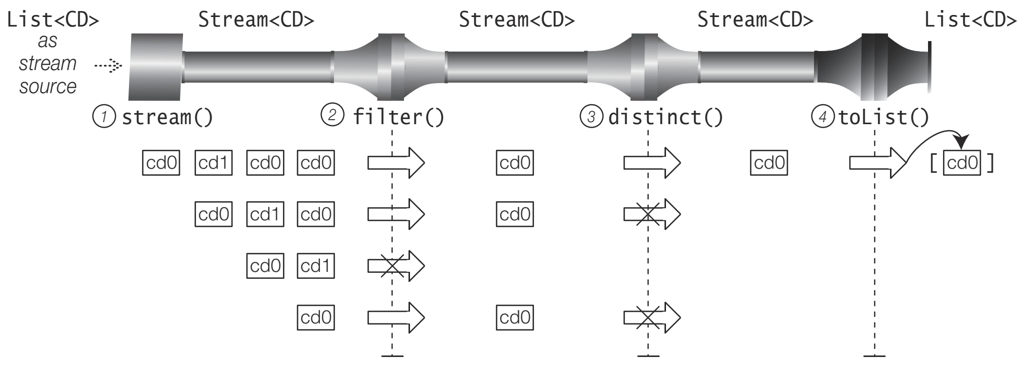
In Figure 16.5, Query 3a creates a list of jazz music CDs after skipping the first two CDs in the stream. The stream pipeline uses a `skip()` operation first to discard two CDs (one of them being a jazz music CD) and a `filter()` operation afterward to select any CDs with jazz music. The execution of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

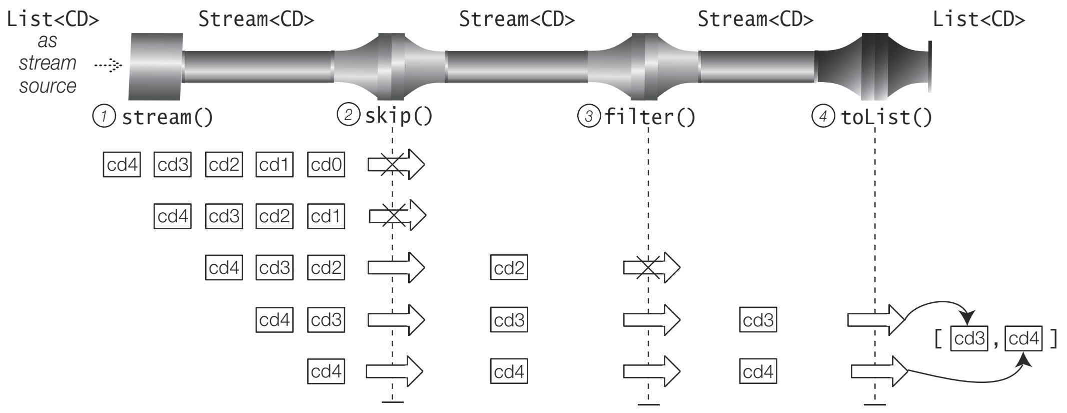


(b) Execution of stream pipeline

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



(b) Execution of stream pipeline

Truncating a Stream

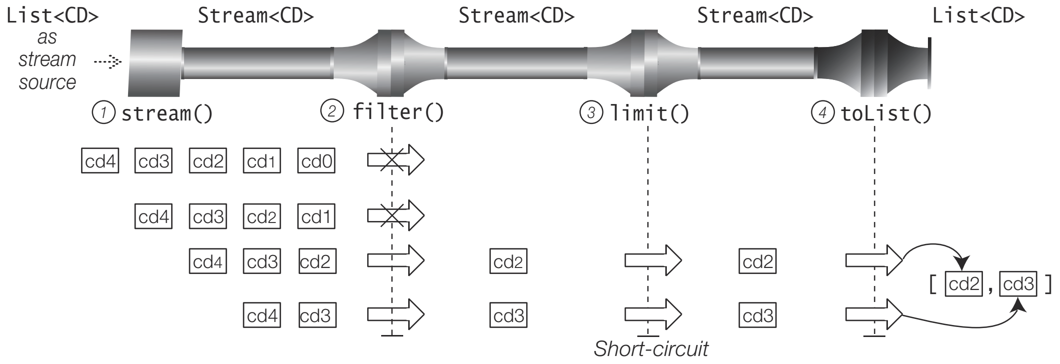
The `limit()` operation returns an output stream whose maximum size is equal to the max size specified as an argument to the method. The input stream is only truncated if its size is greater than the specified max size.

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 Generics", 10, 2018, JAZZ>]
Query 3b: [<Genericos, "Hot Generics", 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.

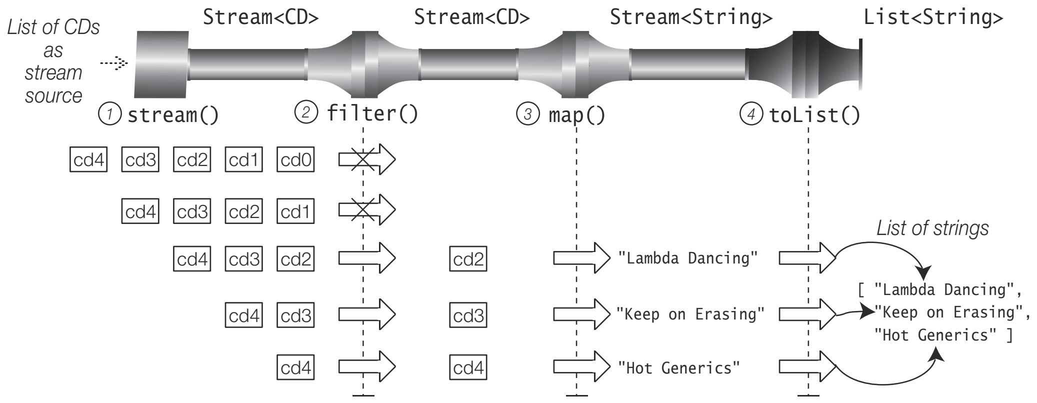
The query below illustrates transforming an object stream to a numeric stream. When executed, the stream pipeline prints the years in which the CDs were released. Note the transformation of the initial stream, Stream<CD>. The map() operation first transforms it to a Stream<Year> and the distinct() operation 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* (*`NumType`*`Stream`). The method returns an output stream (*`NumType`*`Stream`), 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`.

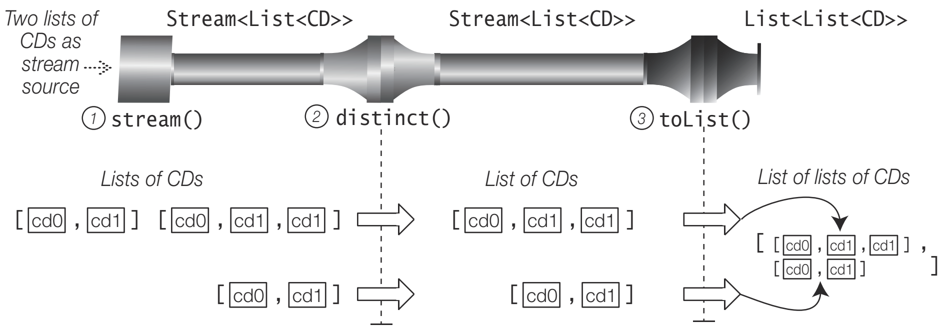
To motivate using the `flatMap()` operation, we look at how to express the query for creating a list of unique CDs from two given lists of CDs. Figure 16.8 shows an attempt to express this query by creating a stream of lists of CDs, `Stream<List<CD>>`, and selecting the unique CDs using the `distinct()` 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*

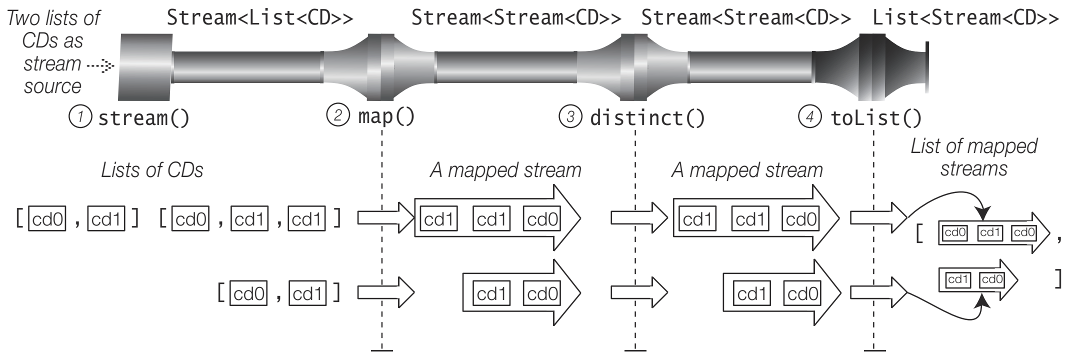


(b) Stream pipeline

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*



(b) Stream pipeline

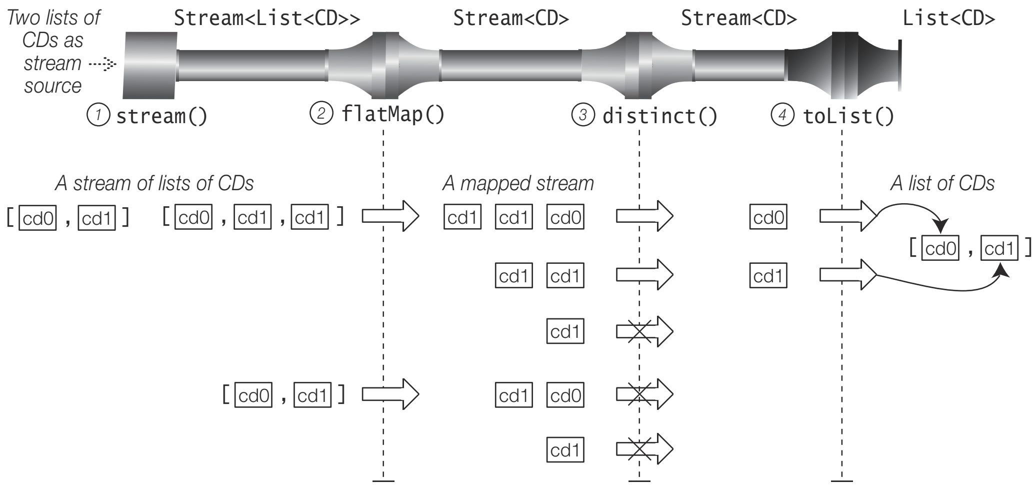
function of the `map()` operation maps each list of CDs to *a mapped stream* that is a stream of CDs, `Stream<CD>`. The resulting stream from the `map()` operation is a stream of streams of CDs, `Stream<Stream<CD>>`. The `distinct()` method distinguishes between elements that are mapped streams of CDs. 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 `int`s (`IntStream`) by applying the `Array.stream()` method at (2) to each row. This would result in a stream of mapped streams of `int`s (`Stream<IntStream>>`), but it is flattened by the `flatMapToInt()` operation to a final stream of `int`s (`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 Generics", 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`

From `Stream<T>`	`map(Function)`
	`flatMap(Function)`
		`mapToInt(ToIntFunction)`
		`flatMapToInt(Function)`
			`mapToLong(ToLongFunction)`
			`flatMapToLong(Function)`
				`mapToDouble(ToDoubleStream)`
				`flatMapToDouble(Function)`

From `IntStream`	`mapToObj(IntFunction)`
	`Stream<Integer> boxed()`
		`map(IntUnary-Operator)`
		`flatMap(IntFunction)`
			`mapToLong(IntToLong-Function)`
			`asLongStream()`
				`mapToDouble(IntToDouble-Function)`
				`asDoubleStream()`

Table 16.2 *Interoperability between Stream Types (Continued)*

Stream types	To `Stream<R>`	To `IntStream`	To `LongStream`	To `DoubleStream`

From `LongStream`	`mapToObj( LongFunction)`
	
	`Stream<Long> boxed()`
		`mapToInt( LongToInt- Function)`	`map( LongUnary- Operator)`
			
			`flatMap( LongFunction)`
				`mapToDouble( LongToDouble- Function)`
				
				`asDoubleStream()`

From `DoubleStream`	`mapToObj( DoubleFunction)`
	
	`Stream<Double> boxed()`
		`mapToInt( DoubleToInt- Function)`	`mapToLong( DoubleToLong- Function)`	`map( DoubleUnary- Operator)`
				
				`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 `mapTo`*`NumType`*`()` 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 `flatMapTo`*`NumType`*`()` 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 transforming the initial `Stream<List<CD>>` into an `IntStream`. The terminal operation `sum()` sums the values in this `IntStream` (p. 973).

```
List<CD> cdList1 = List.of(CD.cd0, CD.cd1);
List<CD> cdList2 = List.of(CD.cd2, CD.cd3, CD.cd4);
int totalNumOfTracks =
    Stream.of(cdList1, cdList2)                      // Stream<List<CD>>
          .flatMapToInt(                             // (1)
              lst -> lst.stream()                    // Stream<CD>
                        .mapToInt(CD::noOfTracks))   // IntStream
                                                     // Stream<IntStream>,
                                                     //   flattened to IntStream.
          .sum();                                    // 42
```

Mapping a Numeric Stream to an Object Stream

The `mapToObj()` method defined by the numeric stream interfaces transforms a numeric stream to an object stream of type `R`, and the `boxed()` method transforms a numeric stream to an object stream of its wrapper class.

The query below prints the squares of numbers in a given closed range, where the number and its square are stored as a pair in a list of size 2. The `mapToObj()` intermediate operation at (2) transforms an `IntStream` created at (1) to a `Stream<List<Integer>>`. Each list in the result stream is then printed by the `forEach()` terminal operation.

```
IntStream.rangeClosed(1, 3)                         // (1) IntStream
         .mapToObj(n -> List.of(n, n*n))            // (2) Stream<List<Integer>>
         .forEach(p -> System.out.print(p + " "));  // [1, 1] [2, 4] [3, 9]
```

The query above can also be expressed as shown below. The `boxed()` intermediate operation transforms the `IntStream` at (3) into a `Stream<Integer>` at (4); in other words, each `int` value is boxed into an `Integer` which is then mapped by the `map()` operation at (5) to a `List<Integer>`, resulting in a `Stream<List<Integer>>` as before. The compiler will issue an error if the `boxed()` operation is omitted at (4), as the `map()` operation at (5) will be invoked on an `IntStream`, expecting an `IntUnaryFunction`, which is not the case.

```
IntStream.rangeClosed(1, 3)                         // (3) IntStream
         .boxed()                                   // (4) Stream<Integer>
         .map(n -> List.of(n, n*n))                 // (5) Stream<List<Integer>>
         .forEach(p -> System.out.print(p + " "));  // [1, 1] [2, 4] [3, 9]
```

The examples above show that the `IntStream.mapToObj()` method is equivalent to the `IntStream.boxed()` method followed by the `Stream.map()` method.

The `mapToObj()` method, in conjunction with a range of `int` values, can be used to create sublists and subarrays. The query below creates a sublist of `CD` titles based on a closed range whose values are used as an index in the `CD` list.

```
List<String> subListTitles = IntStream
    .rangeClosed(2, 3)                         // IntStream
    .mapToObj(i -> CD.cdList.get(i).title())   // Stream<String>
    .toList();                                 // [Lambda Dancing, Keep on Erasing]
```

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 `mapTo`*`NumType`*`()` 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 `Integer`s. 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)	*Stateful*	*Yes*	*No*	*Unchanged*
`dropWhile` (p. 913)	*Stateful*	*Yes*	*No*	*Unchanged*
`filter` (p. 910)	*Stateless*	*Yes*	*No*	*Unchanged*
`flatMap` (p. 921)	*Stateless*	*Yes*	*Yes*	*Not guaranteed*
`limit` (p. 917)	*Stateful, short-circuited*	*Yes*	*No*	*Unchanged*
`map` (p. 921)	*Stateless*	*No*	*Yes*	*Not guaranteed*
`mapMulti` (p. 927)	*Stateless*	*Yes*	*Yes*	*Not guaranteed*
`parallel` (p. 933)	*Stateless*	*No*	*No*	*Unchanged*
`peek` (p. 920)	*Stateless*	*No*	*No*	*Unchanged*
`sequential` (p. 933)	*Stateless*	*No*	*No*	*Unchanged*
`skip` (p. 915)	*Stateful*	*Yes*	*No*	*Unchanged*
`sorted` (p. 929)	*Stateful*	*No*	*No*	*Ordered*
`takeWhile` (p. 913)	*Stateful, short-circuited*	*Yes*	*No*	*Unchanged*
`unordered` (p. 932)	*Stateless*	*No*	*No*	*Not guaranteed*

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 `flatMapTo`*`NumType()`* 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
`distinct (p. 915)`	`Stream<T>`	`()`	
`dropWhile (p. 913)`	`Stream<T>`	`(Predicate<T> predicate)`	`T -> boolean`
`filter (p. 910)`	`Stream<T>`	`(Predicate<T> predicate)`	`T -> boolean`
`flatMap (p. 921)`	`<R> Stream<R>`	`(Function<T,Stream<R>> mapper)`	`T -> Stream<R>`
`flatMapToDouble (p. 921)`	`DoubleStream`	`(Function<T,DoubleStream> mapper)`	`T -> DoubleStream`
`flatMapToInt (p. 921)`	`IntStream`	`(Function<T,IntStream> mapper)`	`T -> IntStream`
`flatMapToLong (p. 921)`	`LongStream`	`(Function<T,LongStream> mapper)`	`T -> LongStream`
`limit (p. 917)`	`Stream<T>`	`(long maxSize)`	
`map (p. 921)`	`<R> Stream<R>`	`(Function<T,R> mapper)`	`T -> R`
`mapMulti (p. 927)`	`<R> Stream<R>`	`(BiConsumer<T,Consumer<R>> mapper)`	`(T, Consumer<R>) -> void`
`mapToDouble (p. 921)`	`DoubleStream`	`(ToDoubleFunction<T> mapper)`	`T -> double`
`mapToInt (p. 921)`	`IntStream`	`(ToIntFunction<T> mapper)`	`T -> int`
`mapToLong (p. 921)`	`LongStream`	`(ToLongFunction<T> mapper)`	`T -> long`
`parallel (p. 933)`	`Stream<T>`	`()`	
`peek (p. 920)`	`Stream<T>`	`(Consumer<T> action)`	`T -> void`
`sequential (p. 933)`	`Stream<T>`	`()`	
`skip (p. 915)`	`Stream<T>`	`(long n)`	
`sorted (p. 929)`	`Stream<T>`	`()`	
`sorted (p. 929)`	`Stream<T>`	`(Comparator<T> cmp)`	`(T,T) -> int`

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 runtime, 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 `Optional`*`NumType`* instance and an `Optional`*`NumType`* 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 `Optional`*`NumType`*, 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 `Optional`*`NumType`*, 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 optNOC0 = 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: $e_1 \oplus e_2 \oplus e_3 \ldots$, 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 subexpressions 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: identity ⊕ e_1 ⊕ e_2 ⊕ e_3 ..., where e_i is an element of this stream, and ⊕ 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, identity ⊕ 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 © (identity ⊕ t) == u ⊕ t

where © and ⊕ 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) -> U, and the combiner function has the function type (U, U) -> 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.noOfTracks();      // (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:

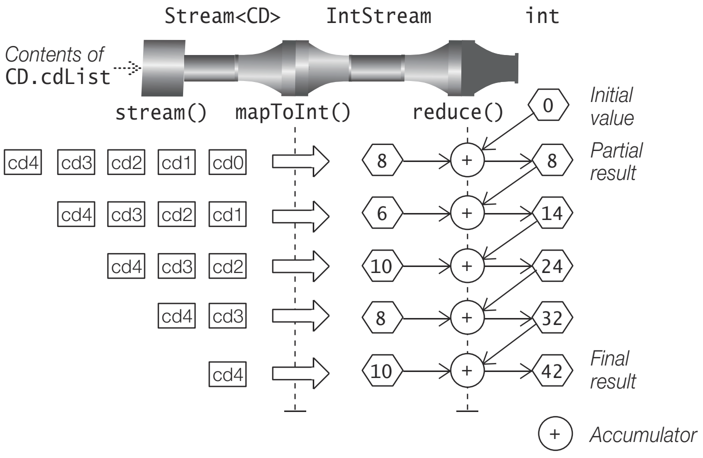
- Initialization of the variable `sum` at (1)
- The accumulative operation at (4) that is applied repeatedly to compute a new partial result in the variable `sum`, based on its previous value and the number of tracks in the current CD

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 `Integer`s 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 `Integer`s 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.

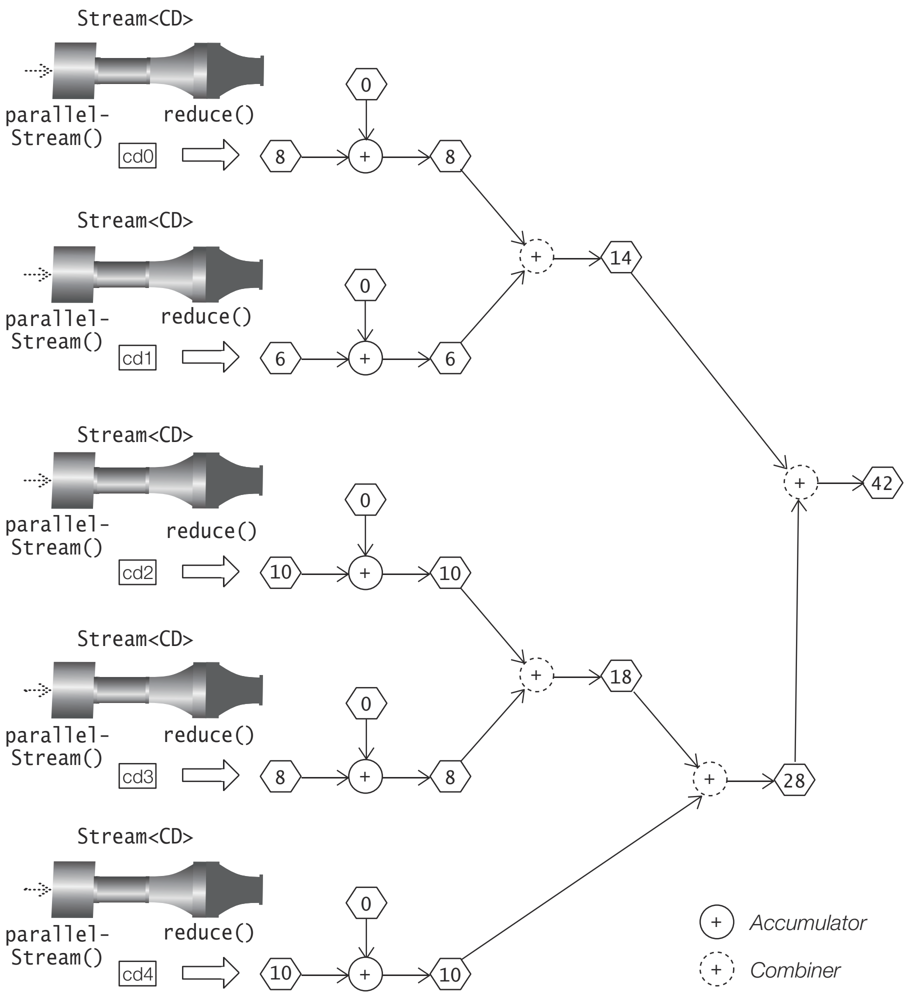
Figure 16.12 also illustrates why the initial value must be an identity value. Say we had specified the initial value to be 3. Then the value 3 would be added multiple times to the sum during parallel execution. We also see why both the accumulator and the combiner are associative, as this allows for 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 `CD`s is mapped to a stream of `Integer`s 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.

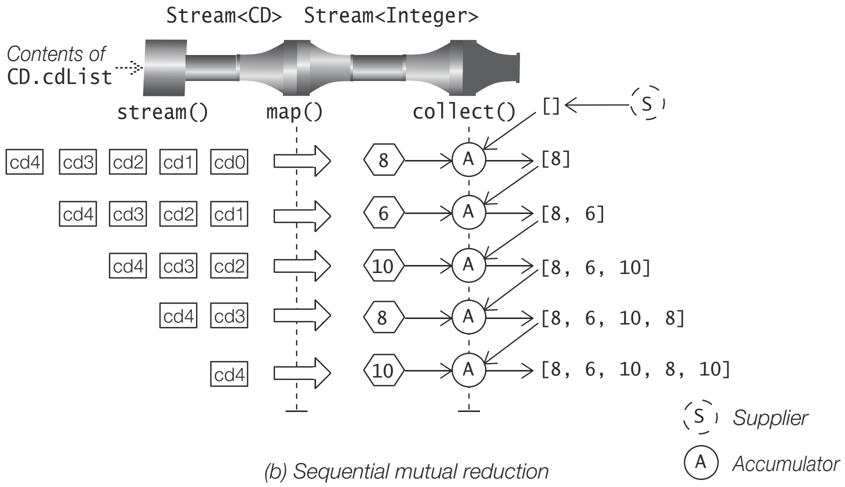
- *Supplier*: The supplier is a `Supplier<R>` that is used to create new instances of a mutable result container of type `R`. Such a container holds the results computed by the accumulator and the combiner. In Figure 16.13, the supplier at (4) returns an *empty* `ArrayList<Integer>` every time it is 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*



(b) Sequential mutual reduction

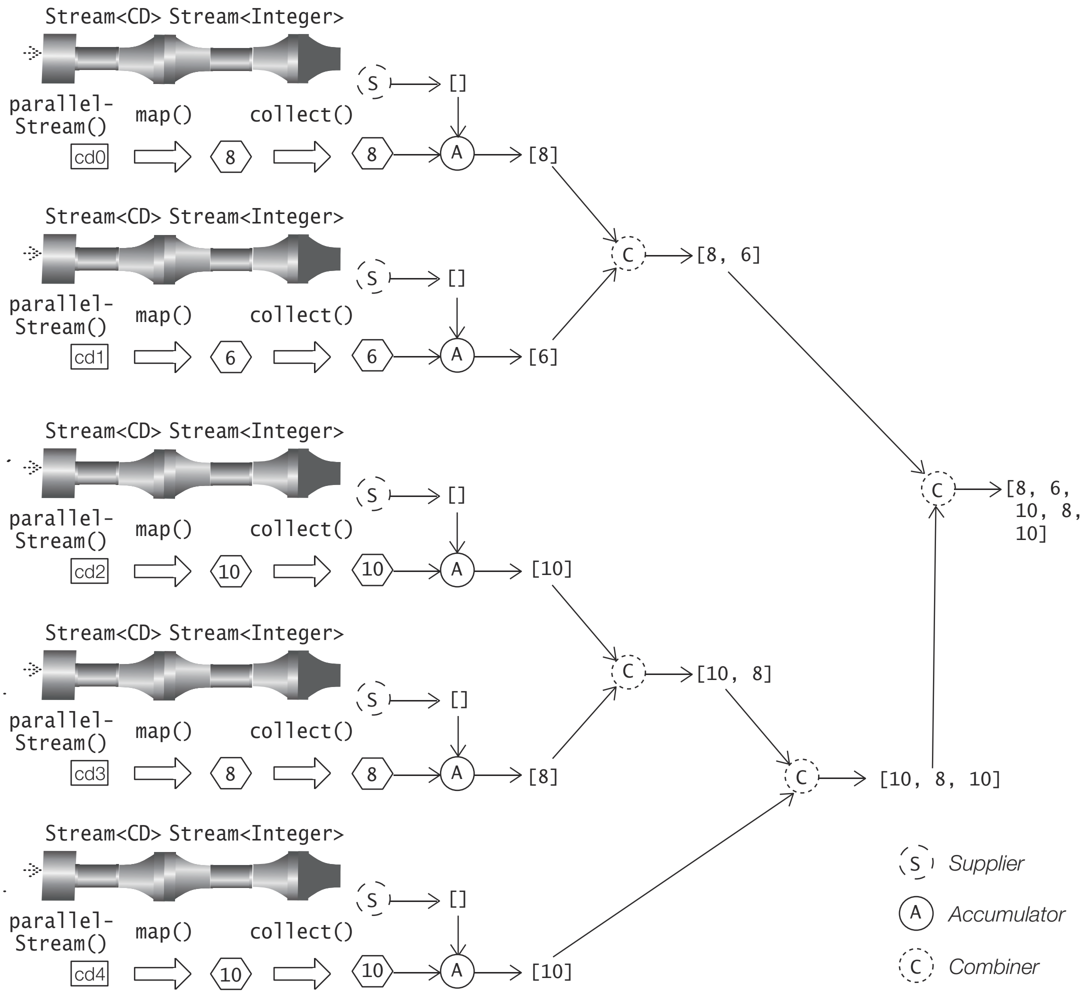
Parallel Mutable Reduction

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 in 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: babanabananabanananabanananana
```

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 `String`s. 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.
>
> *`NumType`*`SummaryStatistics summaryStatistics()`
>
> This terminal operation returns a *`NumType`*`SummaryStatistics` 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 *`NumType`*`SummaryStatistics`, 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`:

`NumType``SummaryStatistics()`

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 `IntSummary-Statistics` object referenced by `stats2` is combined with the state of the `IntSummary-Statistics` 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
`forEach (p. 948)`	`void`	`(Consumer<T> action)`	`T -> void`
`forEachOrdered (p. 948)`	`void`	`(Consumer<T> action)`	`T -> void`
`allMatch (p. 949)`	`boolean`	`(Predicate<T> predicate)`	`T -> boolean`
`anyMatch (p. 949)`	`boolean`	`(Predicate<T> predicate)`	`T -> boolean`
`noneMatch (p. 949)`	`boolean`	`(Predicate<T> predicate)`	`T -> boolean`
`findAny (p. 952)`	`Optional<T>`	`()`	
`findFirst (p. 952)`	`Optional<T>`	`()`	
`count (p. 953)`	`long`	`()`	
`max (p. 954)`	`Optional<T>`	`(Comparator<T> cmp)`	`(T,T) -> int`
`min (p. 954)`	`Optional<T>`	`(Comparator<T> cmp)`	`(T,T) -> int`
`reduce (p. 955)`	`Optional<T>`	`(BinaryOperator<T> accumulator)`	`(T,T) -> T`
`reduce (p. 955)`	`T`	`(T identity, BinaryOperator<T> accumulator)`	`T -> T, (T,T) -> T`
`reduce (p. 955)`	`<U> U`	`(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)`	`U -> U, (U,T) -> U, (U,U) -> U`
`collect (p. 964)`	`<R,A> R`	`(Collector<T,A,R> collector)`	Parameter is not a functional interface.
`collect (p. 964)`	`<R> R`	`(Supplier<R> supplier, BiConsumer<R,T> accumulator, BiConsumer<R,R> combiner)`	`() -> R, (R,T) -> void, (R,R) -> void`
`toArray (p. 971)`	`Object[]`	`()`	
`toArray (p. 971)`	`<A> A[]`	`(IntFunction<A[]> generator)`	`int -> A[]`
`toList (p. 972)`	`List<T>`	`()`	

Table 16.6 *Additional Terminal Operations in the Numeric Stream Interfaces*

Method name (ref.)	Return type
`average (p. 949)`	`Optional`*`NumType`*, where *`NumType`* is `Int`, `Long`, or `Double`
`sum (p. 949)`	*`numtype`*, where *`numtype`* is `int`, `long`, or `double`
`summaryStatistics (p. 974)`	*`NumType`*`SummaryStatistics`, 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>`).

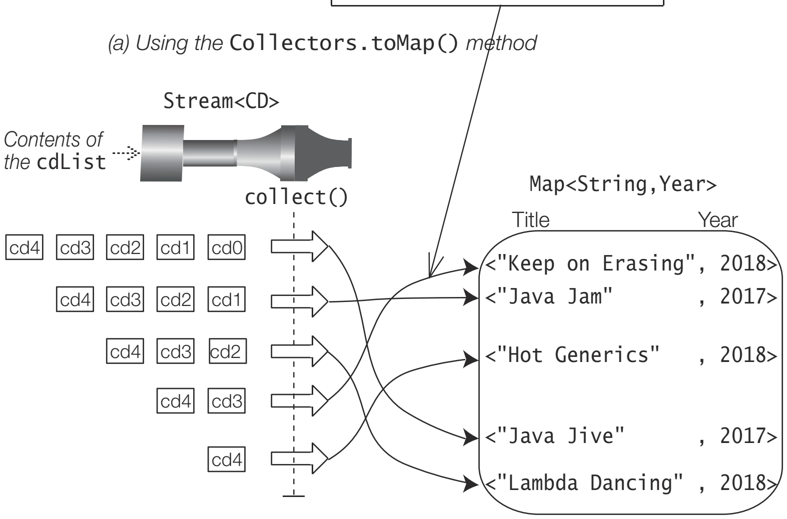
What if we wanted to create a map with CDs and their release year—that is, a `Map<CD, Year>`? In that case, the `keyMapper` should return the CD as the key—that is, map a CD to itself. That is exactly what the `keyMapper` `Function.identity()` does in the pipeline below.

```
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));
```

(a) Using the `Collectors.toMap()` *method*



Stream<CD>
Contents of the cdList
collect()
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd4 cd3
cd4
Map<String,Year>
Title
Year
<"Keep on Erasing", 2018>
<"Java Jam"   , 2017>
<"Hot Generics"   , 2018>
<"Java Jive"   , 2017>
<"Lambda Dancing" , 2018>

(b) Stream pipeline

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 `(y1, y2) -> y1` at (1) have the same value for the year if we assume that a CD can only be released once. Note that `y1` and `y2` 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) -> 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 `String`s. 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>`.

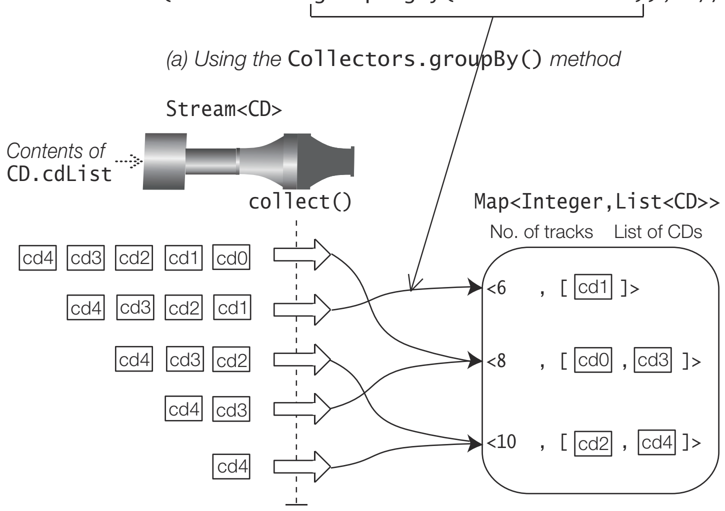
The three-argument method accepts a map supplier as its second parameter. It creates an empty classification map of type `M` that is used by the composed collector. The result is a classification map of type `M` whose key and value types are `K` and `D`, respectively.

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.to-List()` 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)
```

(a) Using the `Collectors.groupBy()` *method*



(b) Stream pipeline

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,                        // (3)
                                   TreeMap::new,
                                   Collectors.toList()));
```

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).

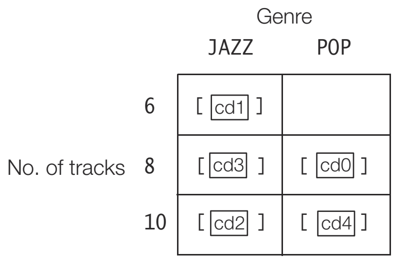
```
Map<Integer, Map<Genre, List<CD>>> twoLevelGrp = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,           // (1)
                Collectors.groupingBy(CD::genre)));          // (2)
```

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*



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 multilevel `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 Generics", 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.

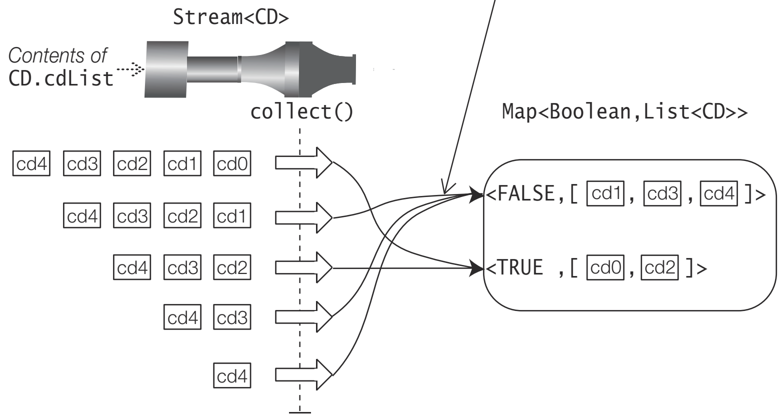
```
Map<Boolean, List<CD>> map2 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.toList())); // (2)
```

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.partitionBy()` *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 Generics", 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 Generics", 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 `flatMapping()` 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 `flatMapping()` 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 `flatMapping()` 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 pl1 = new RadioPlaylist("Radio JVM", cdList1);
    RadioPlaylist pl2 = new RadioPlaylist("Radio JRE", cdList2);
    RadioPlaylist pl3 = new RadioPlaylist("Radio JAR", cdList3);

    // List of radio playlists:                                             (4)
    List<RadioPlaylist> radioPlaylists = List.of(pl1, pl2, pl3);

    // 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 `CD`s 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`$_1$ `bop e`$_2$ `bop e`$_3$ `...`, 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(                      // (5) Downstream collector
             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,        // (6) Downstream collector
                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,             // (7) Downstream collector
                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(                      // (8) Downstream collector
                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
`averagingDouble` (p. 1000)	`Collector<T,?,Double>`	`(ToDoubleFunction<T> mapper)`	`T -> double`
`averagingInt` (p. 1000)	`Collector<T,?,Double>`	`(ToIntFunction<T> mapper)`	`T -> int`
`averagingLong` (p. 1000)	`Collector<T,?,Double>`	`(ToLongFunction<T> mapper)`	`T -> long`
`collectingAndThen` (p. 997)	`Collector<T,A,RR>`	`(Collector<T,A,R> downstream, Function<R,RR> finisher)`	`(T,A) -> R, R -> RR`
`counting` (p. 998)	`Collector<T,?,Long>`	`()`	
`filtering` (p. 992)	`Collector<T,?,R>`	`(Predicate<T> predicate, Collector<T,A,R> downstream)`	`T -> boolean, (T,A) -> R`

Table 16.7 *Static Methods in the* `Collectors` *Class (Continued)*

Method name (ref.)	Return type	Functional interface parameters	Function type of parameters
`flatMapping` (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)`	`flatMapping (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));
```

Index