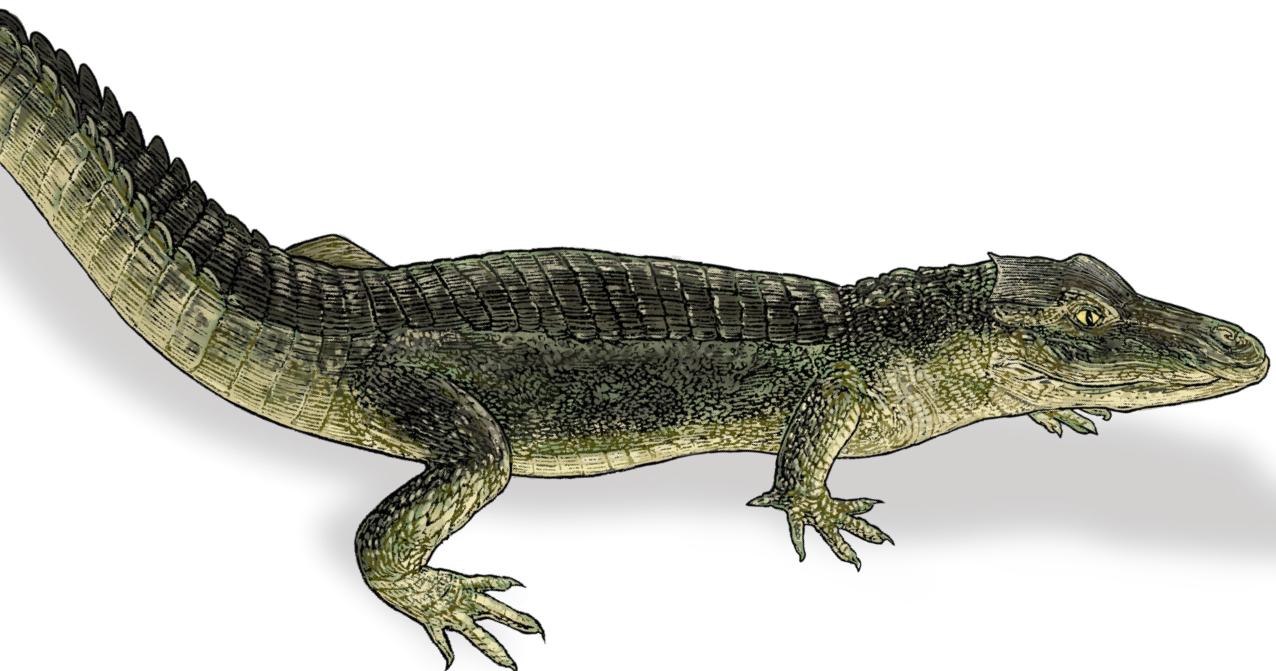


O'REILLY®

2nd Edition

Java Generics and Collections

Fundamentals and Recommended Practices



Maurice Naftalin
& Philip Wadler
with Stuart Marks

"This book really enriched my knowledge about the why and how regarding generics and the collections framework. A very valuable read for those who don't just consume but also want to effectively apply these powerful parts of the Java language."

Robert Scholte
Founder of Sourcegrounds, Java Champion

Java Generics and Collections

Java Generics and Collections has been the go-to guide to generics for more than a decade. This second edition covers Java 21, providing a clear guide to generics from their most common uses to the strangest corner cases, giving you everything you need to know to use and write generic APIs effectively. It covers the collections library thoroughly, so you'll always know how and when to use each collection for any given task. And it compares stream with collection processing, so you'll know which model to use and how they interoperate to get the best out of the platform library.

This indispensable guide covers:

- Fundamentals of generics: type parameters and generic methods
- Subtyping and wildcards
- Generics and reflection
- Tips and good practice for using generics
- Sets, queues, lists, maps, and their implementations
- Sequenced collections, introduced in Java 21
- Concurrent programming and thread safety with collections
- Best practices for using and extending the Java collections framework
- Design philosophy and comparison with other collections libraries

Maurice Naftalin is technical director at Morningside Light Ltd., a software consultancy in the United Kingdom. He has been using and teaching Java since JDK 1.0. He is a Java Champion and author of *Mastering Lambdas* (2015).

Philip Wadler is professor of theoretical computer science at the University of Edinburgh, where his research focuses on the design of programming languages. He was a codesigner of GJ, work that became the basis for Java generics.

Stuart Marks is the JDK core libraries project lead in the Java Platform Group at Oracle. He is currently the maintainer of the Java collections framework. As his alter ego "Dr Deprecator," he also works on Java's deprecation mechanism. He holds a master's degree in computer science from Stanford University.

JAVA

US \$65.99 CAN \$82.99

ISBN: 978-1-098-13672-7



5 6 5 9 9

9 781098136727

O'REILLY®

SECOND EDITION

Java Generics and Collections

Fundamentals and Recommended Practices

*Maurice Naftalin and Philip Wadler
with Stuart Marks*

O'REILLY®

Java Generics and Collections

by Maurice Naftalin and Philip Wadler

Copyright © 2025 Morningside Light. All rights reserved.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Brian Guerin

Indexer: nSight, Inc.

Development Editor: Sara Hunter

Cover Designer: Karen Montgomery

Production Editor: Aleeya Rahman

Cover Illustrator: José Marzan Jr.

Copyeditor: Rachel Head

Interior Designer: David Futato

Proofreader: Piper Content Partners

Interior Illustrator: Kate Dullea

June 2025: Second Edition

Revision History for the Second Edition

2025-06-17: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098136727> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Java Generics and Collections*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-13672-7

[LSI]

*We dedicate this book to Joyce Naftalin, Lionel Naftalin,
Adam Wadler, and Leora Wadler*

—Maurice Naftalin and Philip Wadler

Table of Contents

Preface.....	xiii
--------------	------

Part I. Generics

1. Introduction.....	3
Generic Types	3
Generics Versus Templates	6
Generic Methods and Varargs	7
Primitive and Reference Types	9
Conclusion	12
 2. Subtyping and Wildcards.....	 13
Subtyping and the Substitution Principle	13
Wildcards	16
Wildcards with extends	16
Wildcards with super	18
The Get and Put Principle	20
Arrays	24
Bounded or Unbounded?	27
Using an Unbounded Wildcard	27
Using a Bounded Wildcard	28
Wildcard Capture	29
Restrictions on Wildcards	30
Instance Creation	30

Generic Method Calls	32
Supertypes	32
Conclusion	33
3. Comparison and Bounds.....	35
Comparable	35
The Contract for Comparable	37
Consistent with equals	37
Comparing Integral Values	38
The Maximum of a Collection	39
A Fruity Example	41
Comparator	45
Comparator Methods	47
Enumerated Types	50
Multiple Bounds	53
Bridge Methods	55
Covariant Overriding	57
Conclusion	58
4. Declarations.....	59
Constructors	59
Static Members	60
Nested Classes	63
How Erasure Works	65
Conclusion	67
5. Reifiable and Nonreifiable Types.....	69
Reifiable Types	70
Instance Tests and Casts	71
Unchecked Casts	75
Exception Handling	76
Generics and Arrays	78
Generic Array Creation	79
The Principle of Truth in Advertising	81
How to Create Arrays	83
Array Begets Array	83
A Classy Alternative	84
How to Define ArrayList	86
The Principle of Indecent Exposure	88

Array Creation and Varargs	91
Where Reifiable Types Are Required	93
On the Design of Java Generics	94
Erasure	94
Unbounded Wildcards	95
Arrays	95
Arrays::copyOf	97
Conclusion	98
6. Reflection.....	99
Generics for Reflection	100
Reflected Types Are Reifiable Types	102
Reflection for Primitive Types	103
A Generic Reflection Library	104
Reflection for Generics	107
Reflecting Generic Types	108
Conclusion	111
7. Effective Generics.....	113
Eliminate Unchecked Warnings	113
Enforce Type Safety When Calling Untrusted Code	115
Specialize to Create Reifiable Types	118
Avoid Single-Use Type Variables	119
Use Generic Helper Methods to Capture a Wildcard	120
Cast Through Raw Types When Necessary	121
Use Generic Array Types with Care	122
Use Type Tokens for Run-Time Type Information	123
Conclusion	124

Part II. Collections

8. The Main Interfaces of the Java Collections Framework.....	127
Using the Different Collection Types	128
Set	128
List	129
Map	130
SequencedMap	131
Queue	131

Sequenced Collections	132
SequencedCollection	133
SequencedSet and NavigableSet	133
Deque	134
SequencedMap and NavigableMap	134
Conclusion	134
9. Preliminaries.....	135
Iterable and Iterators	135
Implementations	138
Views	141
Performance	142
Memory	143
Instruction Count and the O-notation	145
Immutability and Unmodifiability	147
Contracts	149
Content-Based Organization	151
Lambdas and Streams	152
Parallel Streams	153
Collections and Thread Safety	155
Synchronization and the Legacy Collections	157
Synchronized Collections and Fail-Fast Iterators	158
Concurrent Collections	159
Conclusion	161
10. The Collection Interface.....	163
The Methods of Collection	163
Adding Elements	163
Removing Elements	164
Querying the Contents of a Collection	164
Making a Collection's Contents Available for Further Processing	165
Using the Methods of Collection	168
Adding Elements	170
Removing Elements	171
Querying the Contents of a Collection	172
Making a Collection's Contents Available for Further Processing	172
Implementing Collection	175
Collection Constructors	176
Conclusion	176

11. The SequencedCollection Interface.....	177
The Methods of SequencedCollection	177
Adding Elements	178
Inspecting Elements	178
Removing Elements	179
Generating a Reversed View	179
Conclusion	180
12. Sets.....	181
Defining a Set: Equivalence Relations	182
Set Implementations	183
HashSet	183
CopyOnWriteArrayList	186
EnumSet	187
UnmodifiableSet	189
Set Views of Maps	191
SequencedSet	191
LinkedHashSet	192
NavigableSet	193
The Methods of NavigableSet	194
TreeSet	200
ConcurrentSkipListSet	203
Comparing Set Implementations	204
Conclusion	205
13. Queues.....	207
Queue Interface Methods	208
Adding an Element to a Queue	208
Retrieving an Element from a Queue	209
Using the Methods of Queue	209
Queue Implementations	211
PriorityQueue	211
ConcurrentLinkedQueue	214
BlockingQueue	215
The Methods of BlockingQueue	215
Using the Methods of BlockingQueue	217
BlockingQueue Implementations	219
TransferQueue	223
Deque	224

The Methods of Deque	225
Deque Implementations	226
BlockingDeque	229
Comparing Queue Implementations	230
Conclusion	232
14. Lists.....	233
List Interface Methods	233
Positional Access Methods	234
Search Methods	234
View-Generating Methods	234
List Iteration Methods	235
Methods Inherited from SequencedCollection	236
Factory Methods	236
Using the Methods of List	237
Using Range-View and Iterator Methods	240
List Implementations	241
ArrayList	241
LinkedList	243
CopyOnWriteArrayList	243
UnmodifiableList	244
Comparing List Implementations	245
Conclusion	246
15. Maps.....	247
Map Interface Methods	248
Iterable-like Operations	248
Collection-like Operations	248
Providing Collection Views of the Keys, Values, or Entries	249
Compound Operations	250
Factory Methods	253
The Interface Map.Entry	253
Using the Methods of Map	254
Map Implementations	256
HashMap	257
WeakHashMap	258
IdentityHashMap	259
EnumMap	261
UnmodifiableMap	262

SequencedMap	263
The Methods of SequencedMap	264
LinkedHashMap	265
NavigableMap	267
Retrieving the Comparator	268
Getting Range Views	268
Getting Closest Matches	269
Other Views	269
TreeMap	270
ConcurrentMap	270
ConcurrentHashMap	271
ConcurrentNavigableMap	272
ConcurrentSkipListMap	272
Comparing Map Implementations	272
Conclusion	273
16. The Collections Class.....	275
Generic Algorithms	275
Changing the Order of List Elements	276
Changing the Contents of a List	277
Finding Extreme Values in a Collection	278
Finding Specific Values in a List	278
Collection Factories	279
Wrappers	280
Synchronized Collections	281
Unmodifiable Collections	281
Checked Collections	282
Other Methods	283
Conclusion	286
17. Guidance for Using the Java Collections Framework.....	287
Avoid Anemic Domain Models	287
Respect the ‘Ownership’ of Collections	289
Prefer Immutable Objects as Set Elements or Map Keys	293
Balance Client and Library Interests in API Design	295
Exploit the Features of Records	297
Prefer Records to Parallel Lists	297
Use Records as Composite Keys	299
Manage the Mutability of Records	301

Avoid Legacy Implementations	302
Avoid Synchronized Wrapper Collections	302
Avoid LinkedList	304
Customize Collections Using the Abstract Classes	305
Conclusion	309
18. Design Retrospective.....	311
Fundamental Issues in the Collections Framework Design	311
Desirable Characteristics of a Collections Framework	312
Unmodifiability via Subtyping	313
Limitations of Subtyping	314
Complementing Static Typing	316
Lumping and Splitting	318
Summary	319
nulls	319
Inconsistent with equals	322
Object Versus E	324
Concurrent Modification	325
Afterword.....	329
Bibliography.....	331
Index.....	335

Preface

The second edition of this book is a comprehensive update for Java 21. Since a major impetus for writing it was the introduction of sequenced collections and the earlier impact of streams and lambdas, I expected before starting work that the major changes needed would be in [Part II](#), principally as a result of the introduction of sequenced collections. However, that was a serious underestimate of the changes that Java has undergone in the last 20 years: in fact, many revisions to [Part I](#) were also required. The language has changed; it now supports records, local variable type inference, pattern-matching expressions, and generic types in type comparison expressions. Also, most examples in [Part I](#) needed updating to take account of changes in the platform libraries, including the current deprecation of the `Integer` constructors, static methods on the `Comparator` interface, and the introduction of unmodifiable collections and of streams.

An important motivation for writing a second edition was to record some of the insights gained by the Java community in the use of both generics and collections since 2005. These are mainly contained in a retrospective section ([“On the Design of Java Generics” on page 94](#)) and two new collections chapters: [Chapter 17](#), which provides guidance for use of the Collections Framework, and [Chapter 18](#), which reviews the most crucial—and also the most controversial—decisions underlying the choices made in the design of the framework. In addition, the chapters on reification ([Chapter 5](#)) and effective generics ([Chapter 7](#)) have been substantially rewritten, the chapter on design patterns has been dropped, and the chapter on migration from pre-generic code has been made available online as an [Appendix](#) for technical and historical interest.

I have preserved the preface to the first edition, in an updated form, at the end of this section. I thought this was worthwhile to convey some of the sense of excitement that accompanied the finely coordinated collection of features with which Java 5 brought the Java language into the 21st century.

— Maurice Naftalin
Edinburgh, February 2025

Acknowledgments for the Second Edition

First, I want to thank all those people who, having read the first edition, nonetheless encouraged me to embark on a revision. Support from the Java community sustained this project throughout; I think myself very fortunate to be part of such a warm and supportive community.

At O'Reilly, Zan McQuade provided invaluable help in building the case for publishing the new edition. As my editor, Sara Hunter was always supportive and encouraging when it was most needed. My copy editor, Rachel Head, had a remarkable sixth sense for detecting technical problems. My production editor, Aleeya Rahman, worked valiantly to remove the many small but crucial bugs in the text.

One satisfying aspect of writing a later edition is to be able to correct the errors in an earlier one (while trying to avoid introducing too many new ones). Amongst the people who identified errata in the first edition were Valentin Baca, Thomas Costick, Svein Egil Nilsen, Emanuel Frua, Anders Granlund, Ravindra Ranwala, Randolph Rothfuss, Jan de Ruiter, and Christopher Sahnwaldt.

I'm grateful to the many people who gave their time to reviewing this book, providing corrections and helpful ideas, both small and large, in some cases giving extremely generously of their time. They include Mohamed Anees, Larry Cable, Ray Djajadinata, Brian Goetz, Mary Gouseti, Andrzej Grzesik, George Heineman, Cay Horstmann, Timmy Jose, Heinz Kabutz, Marc Loy, Chris Newland, Scott Oaks, Simon Park, José Paumard, Simon Ritter, Robert Scholte, Daniel Shaya, Ivan Šipka, Fred Smith, Tushar Srivastava, and Jens-Hagen Syrbe. Of course, all remaining errors and omissions are mine alone.

I'm very happy to repeat, 19 years later, the first edition's recommendation of Heinz Kabutz's *The Java Specialists' Newsletter*, still going strong after 325 editions!

My greatest pleasure is to acknowledge that many of the ideas in this revision were developed in collaboration with Stuart Marks. They were refined in many long discussions and joint presentations. His experience, knowledge, and encouragement made him the best technical editor any author could wish for.

Phil Wadler has been a good friend throughout. Love as ever to Ben, Daniel, Isaac, Joe, and Ruth.

Intended Audience

This book is intended for everyone who knows something about Java and would like to find out more. It's not a Java tutorial, so we assume knowledge of basic Java concepts like classes, static and instance methods, and so on. We use the syntax of Java 21, but most syntax features associated specifically with generics are explained in the text. If you are having difficulty understanding the syntax of a particular code sample, you may find that your integrated development environment (IDE) can offer helpful refactoring suggestions that will clarify its purpose.

We assume very basic knowledge of the Collections API. If you have written any programs using `ArrayList` or `HashSet`, you should have no difficulty. When discussing the behavior of particular collection classes and methods, we often refer to the [Javadoc](#). When in doubt, the Javadoc provides the definitive specification.

Obtaining the Example Programs

The sample code for this edition is available as a Maven project at [https://git
hub.com/MauriceNaftalin/JGC_2e_Book_Code](https://github.com/MauriceNaftalin/JGC_2e_Book_Code). Note that some of the code in the project fails to compile, for reasons given in the corresponding text.

If you have a technical question or a problem using the code examples, please send email to support@oreilly.com.

You may use the sample code in your programs and documentation. You do not need to contact O'Reilly Media for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Java Generics and Collections*, 2nd Edition, by Maurice Naftalin and Philip Wadler (O'Reilly). Copyright 2025 Morningside Light, 978-1-098-13672-7.”

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at permissions@oreilly.com.

Conventions Used in This Book

We use the following font and format conventions for code:

- Code is shown in constant width font, with boldface used for emphasis:

```
class InnocentClient {  
    public static void main(String[] args) {  
        List<Integer>[] intLists = DeceptiveLibrary.createIntLists(1);  
        List<? extends Number>[] numLists = intLists;  
        numLists[0] = List.of(1.01);  
        int i = intLists[0].get(0);           // class cast exception!  
    }  
}
```

- We often include code that corresponds to the body of an appropriate `main` method:

```
List<Integer>[] intLists = DeceptiveLibrary.createIntLists(1);  
List<? extends Number>[] numLists = intLists;  
numLists[0] = List.of(1.01);  
int i = intLists[0].get(0);           // class cast exception!
```

In the code repository, these fragments are assembled into `main` methods. Often, multiple fragments are assembled into the same class. Note that not all classes in the repository compile correctly; the explanation for the compile errors will be found in the corresponding text.

- Code fragments—including program elements such as variable or function names, databases, data types, environment variables, statements, and keywords—are printed in constant width font when they appear within a paragraph (as when we referred to a `main` method in the preceding item).
- We omit imports from the text. Necessary imports are included in the repository code.
- Sample interactive sessions, showing command-line input and corresponding output, are shown in constant width font, with user-supplied input preceded by a percent sign:

```
% javac g/Stack.java g/ArrayList.java g/Stacks.java l/Client.java  
Note: Client.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

Sometimes we show a snippet of a JShell session, with the prompt and user input followed by the JShell output:

```
jshell> List.of("Larry", "Curly", "Larry", "Moe")  
$2 ==> [Larry, Curly, Larry, Moe]
```

- When user-supplied input extends over two lines, the first line is ended with a backslash:

```
% javac -Xlint:unchecked g/Stack.java g/ArrayStack.java \
%   g/Stacks.java l/Client.java
l/Client.java:4: warning: [unchecked] unchecked call
  to push(E) as a member of the raw type Stack
      for (int i = 0; i<4; i++) stack.push(new Integer(i));
```

- New terms, URLs, email addresses, filenames, and file extensions are set in *italics*.
- *Constant width italic* shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a general note.

Assertions

We clarify our code by liberal use of the `assert` statement. Our expectation is that assertions will be enabled. They are not enabled by default, so to obtain the expected behavior you must enable them by invoking the Java virtual machine (JVM) with the `-ea` or `-enableassertions` flag. Each occurrence of `assert` is followed by a `boolean` expression that is expected to evaluate to `true`. If assertions are enabled and the expression evaluates to `false`, an `AssertionError` is thrown, including an indication of where the error occurred.

We only write assertions that we expect to evaluate to `true`. Since assertions may not be enabled, an assertion should never have side effects upon which any non-assertion code depends. When checking for a condition that might not hold (such as confirming that the arguments to a method call are valid), we use a conditional and throw an exception explicitly.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/java-generics-and-collections2e>.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>.

Watch us on YouTube: <https://youtube.com/oreillymedia>.

Preface to the First Edition

The introduction of *generics* to the Java language, accompanied at the same time by other improvements, has transformed the experience of programming in Java. The most immediate effect is in the use of the collection types, such as `List`. You could always use these classes to collect objects of different types—for example, `String` or `Integer`—but generics has added to this flexibility the capability to restrict, in a precise way, the type that a collection should contain, and to enforce this restriction.

Say you wish to process lists. Some may be lists of integers, others lists of strings, and yet others lists of lists of strings. In Java before generics, this was simple. You could represent all three by the same class, called `List`, which has elements of class `Object`:

```
list of integers      List
list of strings      List
list of lists of strings  List
```

In order to keep the language simple, you were forced to do some of the work yourself: you had to keep track of the fact that you had a list of integers (or strings or lists of strings), and when you extracted an element from the list you had to cast it from `Object` back to `Integer` (or `String` or `List`). The Collections Framework before generics made extensive use of this idiom.

As Einstein said, “Everything should be as simple as possible but no simpler.”¹ And some might say the approach above is too simple. In Java with generics, you may distinguish different types of lists:

list of integers	<code>List<Integer></code>
list of strings	<code>List<String></code>
list of lists of strings	<code>List<List<String>></code>

Now the compiler keeps track of whether you have a list of integers (or strings or lists of strings), and no explicit cast back to `Integer` (or `String` or `List<String>`) is required. In some ways, this is similar to *generics* in Ada or *templates* in C++, but the actual inspiration is *parametric polymorphism* as found in functional languages such as ML and Haskell.

Part I of this book provides a thorough introduction to generics. We discuss the interactions between generics and subtyping, and how to use wildcards and bounds; we describe techniques for evolving your code; we explain subtleties connected with casts and arrays; we treat advanced topics such as the interaction between generics and security, and how to maintain binary compatibility; and we update common design patterns to exploit generics.

The introduction of generics into Java has sparked some controversy. Certainly, the design of generics involved trade-offs: making code evolution easy requires that objects should not *reify* compile-time information about generic type parameters—that is, should not preserve this information for use at run time—but its absence at run time introduces corner cases into operations such as casting and array creation. We present a balanced treatment of generics, explaining how to exploit their strengths and work around their weaknesses.

Part II provides a comprehensive introduction to the Collections Framework. The English polymath Isaac Newton (1675) wrote, “If I have seen further, it is by standing on the shoulders of giants.” The best programmers live by this motto, building on existing frameworks and reusable code wherever appropriate. The Java Collections Framework provides reusable interfaces and implementations for a number of common collection types, including lists, sets, queues, and maps.²

1 This is the popular paraphrase: what Einstein actually said was that theory should be as simple as possible “without having to surrender the adequate representation of a single datum of experience” (Calaprice 2011, 384–5). The pre-generics code above clearly fails this test: it fails to represent an important aspect of reality, namely the type of the objects to be contained in the `List`.

2 Of course, not all programmers are so respectful of their predecessors’ work. Sometimes, as Richard Hamming said of computer scientists, “Instead of standing on each other’s shoulders, we stand on each other’s toes.”

Compared to pre-generic code, generic code using collections is easier to read and the compiler catches more type errors. Further, collections provide excellent illustrations of the use of generics. One might say that generics and collections were made for each other, and, indeed, ease of use of collections was one of the main reasons for introducing generics in the first place. In Part II, we describe the entire Collections Framework from first principles in order to help you use collections more effectively.

The design of generics for Java is influenced by a number of previous proposals—notably, GJ, by Bracha et al. (1998); the addition of wildcards to GJ, proposed by Igarashi and Viroli (2006); and further development of wildcards by Torgersen et al. (2004). Design of generics was carried out under the Java Community Process by a team led by Bracha and including Odersky, Thorup, and Wadler (as parts of JSR 14 and JSR 201). Odersky’s GJ compiler formed the basis of the original javac compiler for generics.

Acknowledgments for the First Edition

The folks at Oracle, and previously at Sun Microsystems, were fantastically good about answering our questions. They were always happy to explain a tricky point or mull over a design trade-off. Thanks to Joshua Bloch, Gilad Bracha, Martin Buchholz, Joseph D. Darcy, Neal M. Gafter, Mark Reinhold, David Stoutamire, Scott Violet, and Peter von der Ahé.

It has been a pleasure to work with the following researchers, who contributed to the design of generics for Java: Erik Ernst, Christian Plesner Hansen, Atsushi Igarashi, Martin Odersky, Mads Torgersen, and Mirko Viroli.

We received comments and help from a number of people. Thanks to Brian Goetz, David Holmes, Heinz M. Kabutz, Deepti Kalra, Angelika Langer, Stefan Liebeg, Doug Lea, Tim Munro, Steve Murphy, and C K Shabin.

We enjoyed reading Heinz M. Kabutz’s *The Java Specialists’ Newsletter* and Angelika Langer’s *Java Generics FAQ*, both available online.

Our editor, Michael Loukides, was always ready with good advice. Paul C. Anagnosopoulos of Windfall Software turned our LaTeX into camera-ready copy, and Jeremy Yallop produced the index.

Our families kept us sane (and insane). Love to Adam, Ben, Catherine, Daniel, Isaac, Joe, Leora, Lionel, and Ruth.

PART I

Generics

Generics are a powerful, and sometimes controversial, feature of the Java programming language. This part of the book describes generics, using the Collections Framework as a source of examples. A comprehensive introduction to the Collections Framework appears in the second part of the book.

The first four chapters focus on the fundamentals of generics. [Chapter 1](#) gives an overview of generic types and methods. [Chapter 2](#) reviews how subtyping works and explains how wildcards let you use subtyping in connection with generics. [Chapter 3](#) describes how generics work with the `Comparable` interface, which requires a notion of *bounds* on type variables. [Chapter 4](#) looks at how generics work with various declarations, including constructors, static members, and nested classes. Once you have these four chapters under your belt, you will be able to use generics effectively in most basic situations.

The next three chapters treat advanced topics. [Chapter 5](#) explains how the same design that leads to ease of evolution also necessarily leads to a few rough edges in the treatment of casts, exceptions, and arrays. The fit between generics and arrays is the worst rough corner of the language, and we formulate two principles to help you work around the problems. [Chapter 6](#) explains features that relate generics and reflection, including the type `Class<T>` and additions to the Java library that support reflection of generic types. [Chapter 7](#) contains advice on how to use generics effectively in practical coding. We consider a variety of techniques learned from the experience of using generics in practical development. This chapter also contains a section reviewing some of the design decisions that shaped the generics features of Java.

The [Appendix](#) explains how the features of generics allowed Java to evolve to become a language supporting generics while maintaining compatibility with legacy, nongeneric code.

CHAPTER 1

Introduction

The purpose of generics is to allow the same code to be reused for creating or handling objects of different types. For example, `List<String>` and `List<Integer>` are different types, but generics allows them to be implemented with the same code, with type safety preventing any possibility of confusion between them. Two kinds of Java code can be generic: types, such as the collection classes and interfaces; and methods, such as the static methods in the utility class `java.util.Collections`. Let's look at each of these in turn.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/
main/src/main/java/org/jgcbook/chapter01](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter01)

Generic Types

An interface or class may be declared to take one or more *type parameters*, which are written in angle brackets and must be supplied when you declare a variable belonging to the interface or class or when you create a new instance of a class.

Here is an example:¹

```
org/jcbook/chapter01/A_generic_types/Program_1  
List<String> words = new ArrayList<String>(); ①  
words.add("Hello ");  
words.add("world!");  
String s = words.get(0)+words.get(1); ②  
assert s.equals("Hello world!");
```

In the Collections Framework, the class `ArrayList<E>` implements the interface `List<E>`, where `E` is a *type variable*—a placeholder for a type that will be supplied whenever `List` or `ArrayList` are used in a program. This trivial code fragment declares the variable `words`, to be used to refer to a list containing strings. It then creates an instance of an `ArrayList`, assigns its reference to the variable `words`, adds two strings to the list, and gets them out again.

Don't deduce from this example that a variable of type `SomeClass<T>` necessarily refers to a `SomeClass` object *containing* objects of type `T`. If `SomeClass` is a collection type, then that is correct, but the general idea of generics is wider. For example, objects implementing the interface `Comparable<T>` don't contain `T` objects: rather, they have the capability of comparing themselves with objects of type `T` (see “[Comparable](#)” on page 35). For another example, the class `java.util.ServiceLoader<S>` locates providers for a service of type `S`, so a `ServiceLoader<CharsetProvider>` locates services to encode and decode character sets, a `ServiceLoader<FileSystemProvider>` locates services that manage filesystems, and so on. That is what we meant when we said earlier that the same code—that of the *generic class* `SomeClass`—can be reused for creating or handling objects of different types (that is, types that are different instantiations of the *type parameter* `T`). Since we'll mostly be discussing generics in the context of collections, you could be forgiven for thinking it means containment, but in fact the idea is more widely applicable.

Before generating bytecode, the compiler first processes source code to ensure that its use of type parameters is consistent throughout. In the previous example, it checks to see that a reference to an instance of `ArrayList<String>` can be assigned to a variable of type `List<String>`. If this check is successful, it will discard the type parameters before going on to generate the bytecode. So the source code for line ① that is actually compiled is:

```
List words = new ArrayList();
```

¹ If you view this code in your IDE you may be told, correctly, that the righthand side of line ① is equivalent to `new ArrayList<>()`. Writing that instead prompts the compiler to insert the type supplied in the variable declaration on the lefthand side.

Now, however, `words` is a list of `Object`. This means the return type of `words::get` is `Object`, and the compiler must insert casts to `String` to ensure that line ❷ compiles.

In Java before generics, the same code would have been written as follows:

```
org/jcbook/chapter01/A_generic_types/Program_2
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1));
assert s.equals("Hello world!");
```

This is, in effect, the same code that the generics compiler goes on to use for bytecode generation after the preprocessing just described, so the bytecode compiled from the two sources will be identical. We say that generics are implemented by type *erasure* because the types `List<Integer>` and `List<String>`—and, for that matter, `List<List<String>>`—are represented at run time by the same type, `List`. We also use *erasure* to describe the process that converts the first program to the second. There is a more detailed description of erasure in “[How Erasure Works](#)” on page 65, but for now you can think of it as follows:

Erasure: The compile-time process by which type annotations are removed prior to bytecode generation.

The term “erasure” is a slight misnomer, since the process not only erases type parameters, but also adds casts.

As this example shows, generics implicitly perform the same cast that is explicitly performed without generics. If such casts could fail, it might be hard to debug code written with generics. This is why it is reassuring that generics come with the following guarantee:

Cast-iron guarantee: The implicit casts added by the compilation of generics never fail.

There is some fine print on this guarantee: it applies only when no *unchecked warnings* have occurred. These are warnings generated by the compiler in situations in which it cannot ensure type safety. Later, we will discuss at some length what causes unchecked warnings to be issued and how to minimize their effect.

Implementing generics by erasure had a number of important effects. It kept things simple, in that generics did not require changes to the JVM or to bytecode. It kept things small, in that there is exactly one implementation of `List`, not one version for each type. And it eased evolution, since the same library can be accessed in both nongeneric and generic forms.

This last point is worth some elaboration. It meant that you could never get unpleasant problems due to maintaining two versions of the libraries: a *legacy* version that worked with pre-generic Java and a *generic* version that worked with generic Java. At the bytecode level, code that doesn't use generics looks just like code that does. This meant that there was never any need to switch to generics all at once—you could evolve your code by updating just one package, class, or method at a time to start using generics.² (Of course, the cast-iron guarantee we mentioned holds only if you add generic types that match the legacy code.)

Another consequence of implementing generics by erasure is that array types differ in important ways from parameterized types. Executing:

```
new String[size]
```

allocates an array and stores in that array an indication that its components are of type `String`. In contrast, executing:

```
new ArrayList<String>()
```

allocates a list but does not store in the list any indication of the type of its elements. We say that Java *reifies* array component types—that is, it preserves them for use at run time—but does not reify list element types (or other generic types). As explained previously, this design was crucial to easing evolution and therefore to the continued popularity of Java. Years later, on the other hand, it continues to complicate casts, instance tests, and array creation, as we will see in [Chapter 5](#). The last section of that chapter, “[On the Design of Java Generics](#)” on page 94, considers the arguments for and against the use of erasure and whether it is still the right choice for the design of Java today.

Generics Versus Templates

Generics in Java resemble templates in C++. There are just two important things to bear in mind about the relationship between Java generics and C++ templates: syntax and semantics. The syntax is deliberately similar and the semantics are deliberately different.

Syntactically, angle brackets were chosen because they are familiar to C++ users, and because square brackets would be hard to parse. Semantically, Java generics are defined by erasure, whereas C++ templates are defined by *expansion*. In C++ templates, each instance of a template at a new type is compiled separately. If you use a list of integers, a list of strings, and a list of lists of strings, there will be three versions of the code. If you use lists of a hundred different types, there will be a hundred versions of the code—a problem known as *code bloat*. In Java, no matter

² In the [Appendix](#), we even explain how you could declare generic types for legacy code.

how many types of lists you use, there is always one version of the code, so bloat does not occur.

Expansion may lead to more efficient implementation than erasure since it offers more opportunities for optimization, particularly for primitive types such as `int`. For code that is manipulating large amounts of data—for instance, large arrays in scientific computing—this difference may be significant. However, in practice, for most purposes the difference in efficiency is not important, whereas the problems caused by code bloat can be crucial.

C++ also offers the ability to manipulate values as well as types. This technique, known as *template metaprogramming*, makes it possible to use templates as a sort of “macroprocessor on steroids” that can perform arbitrarily complex computations at compile time. Java generics are deliberately restricted to types in order to avoid excessive complexity.

Generic Methods and Varargs

The preceding section described how interfaces and classes can accept a type argument. Individual methods can also be generic. Here is a method that accepts an array of any type and converts it to a list of the same type:

```
org/jcbook/chapter01/B_generic_methods_and_varargs/Lists_1
```

```
class Lists_1 {  
    public static <T> List<T> toList(T[] arr) {  
        List<T> list = new ArrayList<T>();  
        for (T elt : arr) list.add(elt);  
        return list;  
    }  
}
```

The static method `toList` accepts an array of type `T[]` and returns a list of type `List<T>`, and does so for *any* reference type `T`. This is indicated by writing `<T>` at the beginning of the method declaration, which declares `T` as a new type variable. The variable `T` can be any legal Java identifier, but by convention, type variable identifiers are usually single uppercase letters: `T` for Type, `R` for ReturnType, `U` for a second type parameter if required, and so on. A method that declares a type variable in this way is called a *generic method*. The scope of the type variable `T` is local to the method itself; it may appear in the method declaration, but not outside the method.

The method may be invoked as follows:

```
org/jcbook/chapter01/B_generic_methods_and_varargs/Lists_1
```

```
List<Integer> ints = Lists_1.toList(new Integer[] {1, 2, 3});  
List<String> words = Lists_1.toList(new String[] {"Hello", "world!"});
```

In the first line, boxing (see “Primitive and Reference Types” on page 9) converts the `int` values 1, 2, and 3 to `Integers`.

Packing the arguments into an array is cumbersome. *Variable arity parameters*, usually called `varargs`, permit a special, more convenient syntax for the case in which the last argument of a method is an array. To use this feature, we replace `T[]` with `T...` in the method declaration, giving a declaration very like that of `java.util.List::of`:

org/jcbook/chapter01/B_generic_methods_and_varargs/Lists_2

```
class Lists_2 {  
    public static <T> List<T> toList(T... arr) {  
        List<T> list = new ArrayList<T>();  
        for (T elt : arr) list.add(elt);  
        return list;  
    }  
}
```

Now the method can be invoked as follows:

org/jcbook/chapter01/B_generic_methods_and_varargs/Lists_2

```
List<Integer> ints1 = Lists_2.toList(1, 2, 3);  
List<String> words = Lists_2.toList("Hello", "world!");
```

This is just shorthand for what we wrote before. At run time, the arguments are packed into an array that is passed to the method, just as previously seen.

Any number of arguments may precede the final `varargs` argument. Here is a simplified version of the method `java.util.Collections::addAll`, which accepts a list and adds all the additional arguments to the end of the list:

org/jcbook/chapter01/B_generic_methods_and_varargs/Lists_3

```
public static <T> void addAll(List<T> list, T... arr) {  
    for (T elt : arr) list.add(elt);  
}
```

In calling a method with a `varargs` parameter, you can either pass a list of arguments to be implicitly packed into an array or explicitly pass the array directly. Thus, `addAll` may be invoked as follows:

org/jcbook/chapter01/B_generic_methods_and_varargs/Lists_3

```
List<Integer> ints = new ArrayList<Integer>();  
Lists_3.addAll(ints, 1, 2);  
Lists_3.addAll(ints, new Integer[] { 3, 4 });  
assert ints.equals(List.of(1, 2, 3, 4));
```

We will see later that when we attempt to create an array containing a generic type, we will always receive an unchecked warning. Since `varargs` always create an array, they should be used with care when the argument has a generic type (see “Array Creation and Varargs” on page 91).

The type parameter to the generic method is inferred in these examples, which illustrate the usual situation in which one or more arguments corresponding to a type parameter all have the same type. When there are no arguments, or the arguments are of different subtypes of the intended type, the type parameter may need to be supplied explicitly. For example:

```
org/jcbook/chapter01/B_generic_methods_and_varargs/Lists_2  
var ints = Lists_2.<Integer>toList();  
var objs = Lists_2.<Object>toList(1, "two");
```

In the first example, without the explicit type parameter the type inferred would be `Object`. In the second example, the type inferred would not only inherit from `Object` but would also implement all the interfaces that both `Integer` and `String` implement, including (but not only) `Serializable` and `Comparable`. This is an *intersection type*; we will explore intersection types in detail in “[Multiple Bounds](#)” on page 53.

When a type parameter is passed to a generic method invocation, it appears in angle brackets to the left, just as in the method declaration. Java grammar requires that type parameters appear only in method invocations that use a dotted form. Even if the method `toList` is defined in the same class that invokes the code or is imported as a static method, we cannot shorten it as follows:

```
List<Integer> ints = <Integer>toList(); // compile-time error
```

Primitive and Reference Types

The last topic to consider in this introductory chapter is that of primitive versus reference types. A *reference type* is any class, interface, or array type, whereas a *primitive type* is one of the eight listed in [Table 1-1](#). The distinction is fundamental to Java’s implementation of generics, in which only reference types can be used as type parameters; primitive types are disallowed. So for example, instead of writing `List<int>`, you must write `List<Integer>`.³ All reference types are subtypes of class `Object`, and any variable of reference type may be set to the value `null`. [Table 1-1](#) shows, for each of the eight primitive types, the corresponding library class of reference type, located in the package `java.lang`.

Table 1-1. Primitive and corresponding reference types

Primitive	Reference
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>

³ This difference may disappear in a future version of Java as part of [Project Valhalla](#), but much preparatory work is needed before this can happen.

Primitive	Reference
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Conversion of a primitive type to the corresponding reference type is called *boxing*. The compiler can often insert code to do this conversion automatically: this is called *autoboxing*. The reverse conversion, in which a reference value is unwrapped to produce a value of the corresponding primitive type, is called *unboxing*.

Boxing and unboxing conversions are applied automatically where appropriate. If an expression `e` of type `int` appears where a value of type `Integer` is expected, boxing converts it to `Integer.valueOf(e)`. If an expression `e` of type `Integer` appears where a value of type `int` is expected, unboxing converts it to the expression `e.intValue()`. For example, the sequence:

[org/jgcbook/chapter01/C_primitive_and_reference_types/Program_1](#)

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
int n = ints.getFirst();
```

produces bytecode equivalent to the sequence produced by:

[org/jgcbook/chapter01/C_primitive_and_reference_types/Program_2](#)

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

The call `Integer.valueOf(1)` returns an `Integer` instance representing the `int` value 1. The factory method `Integer::valueOf` is preferred to the constructor `Integer::new`, which was deprecated in Java 11 and marked for removal in Java 16 because it allows for the possibility of reusing cached `Integer` objects.⁴ In fact, the Java Language Specification ([Gosling et al. 2023](#), §5.1.7) mandates caching for certain values: it requires that any two boxing conversions of these values should return the same reference. The values concerned are `ints` and `shorts` between -128 and 127 inclusive, `chars` between '\u0000' and '\u007f', `bytes`, and `booleans`. So this assertion will always succeed:

⁴ Perhaps confusingly, this deprecation may eventually be reversed, also as part of [Project Valhalla](#).

```
assert Integer.valueOf(5) == Integer.valueOf(5);
```

whereas this one will usually succeed—but might not, depending on the caching policy of the JVM:

```
assert Integer.valueOf(500) != Integer.valueOf(500);
```

Here, again, is the code to find the sum of a list of integers, conveniently packaged as a static method:

```
public static int sum(List<Integer> ints) {  
    int s = 0;  
    for (int n : ints) { s += n; }  
    return s;  
}
```

Why does the argument have type `List<Integer>` and not `List<int>`? Because type parameters must always be bound to reference types, not primitive types. And why has the method been defined to return a value of type `int` and not `Integer`? Because result types may be either primitive or reference types, and it is more efficient to use the former than the latter. Unboxing occurs when each `Integer` in the list `ints` is bound to the variable `n` of type `int`. We could rewrite the method, replacing each occurrence of `int` with `Integer`:

```
public static Integer sumInteger(List<Integer> ints) {  
    Integer s = 0;  
    for (Integer n : ints) { s += n; }  
    return s;  
}
```

This code compiles and runs correctly but performs a lot of needless work. Each iteration of the loop unboxes the values in `s` and `n`, performs the addition, and boxes the result again. In an application containing code like this on a critical path, where it is very frequently executed, the work of boxing and unboxing can have a big impact on performance.

There is one `Integer` value that does not correspond to a valid `int`. This is `null`, which is a member of every type. Supplying either version of the `sum` method with a list containing a `null` value will result in a `NullPointerException`. Taking the second as an example:

```
jshell> chapter01.C_primitive_and_reference_types.SumInteger.sumInteger(  
...> Arrays.asList(1, 2, 3, null)  
| Exception java.lang.NullPointerException: Cannot invoke \  
"java.lang.Integer.intValue()" because "<local3>" is null
```

```
|      at SumInteger.sumInteger (SumInteger.java:8)
|      at (#3:1)
```

Conclusion

In this chapter, we saw the basic idea of generics in Java and learned their purpose: allowing reuse of the same code on objects of different types in a type-safe way. Generic type information is discarded after compilation; this process is called erasure, and we will need to study its consequences in depth.

In the next chapter, we'll go on to see how generics work with Java's object-oriented polymorphic subtyping system.

CHAPTER 2

Subtyping and Wildcards

Now that we've covered the basics, we can start to explore more advanced features of generics, such as subtyping and wildcards. In this chapter, we'll review how subtyping works and see how wildcards let us use subtyping in connection with generics. The Java Collections Framework will serve as the source of our examples; see [Part II](#) for details on specific features of the Collections API.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/
main/src/main/java/org/jgcbook/chapter02](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter02)

Subtyping and the Substitution Principle

Subtyping is a key feature of object-oriented languages such as Java. In Java, one type is a *subtype* of another if they are related by an `extends` or `implements` clause. Here are some examples:

<code>Integer</code>	is a subtype of	<code>Number</code>
<code>Double</code>	is a subtype of	<code>Number</code>
<code>ArrayList<E></code>	is a subtype of	<code>List<E></code>
<code>List<E></code>	is a subtype of	<code>Collection<E></code>
<code>Collection<E></code>	is a subtype of	<code>Iterable<E></code>

Subtyping is transitive, meaning that if one type is a subtype of a second, and the second is a subtype of a third, then the first is also a subtype of the third. So from the last two lines in the preceding list, it follows that `List<E>` is a subtype of `Iterable<E>`. If one type is a subtype of another, we also say that the second is a *supertype* of the first. Subtyping is reflexive, meaning that every type is a subtype—and therefore also a supertype—of itself. Every reference type is a subtype of `Object`, and `Object` is a supertype of every reference type.

The most important property of subtyping is summarized in the Substitution Principle:¹

Substitution Principle: Wherever a value of type `T` is expected, you can instead provide a value of a subtype of `T`.

This means a variable of a given type may be assigned a value of any subtype of that type; for instance, a variable of type `Number` may be assigned a value of type `Integer` or `Double`. Similarly, a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

Consider the interface `Collection<E>`. One of its methods is `add`, which takes a parameter of type `E`:

```
interface Collection<E> {  
    ...  
    boolean add(E elt);  
    ...  
}
```

According to the Substitution Principle, if we have a collection of numbers we may add an integer or a double to it, because `Integer` and `Double` are subtypes of `Number`:

org/jgcbook/chapter02/A_subtyping_and_the_substitution_principle/Program_1

```
List<Number> nums = new ArrayList<>();  
nums.add(2);  
nums.add(0.25);  
assert nums.equals(List.of(2, 0.25));
```

Here, subtyping is used in two ways for each method call. The first call is permitted because `nums` has the type `List<Number>`, which is a subtype of `Collection<Number>`, and `2` has the type `Integer` (thanks to autoboxing), which is a subtype of `Number`. The second call is similarly permitted: `0.25` is boxed to the type `Double`, also a subtype of `Number`. In both calls, the `E` in `List<E>` is taken to be `Number`.

¹ Liskov (1987) provides a definition of behavioral subtyping in terms of the ability to substitute values of one type for values of another without any change in program behavior. Applying this definition in reverse gives the design principle often called the *Liskov Substitution Principle*. We discuss its application to the Collections Framework in Chapter 18.

It may seem reasonable to expect that since `Integer` is a subtype of `Number`, it follows that `List<Integer>` is a subtype of `List<Number>`. But this is *not* the case, because if it were, the Substitution Principle would rapidly get us into trouble. To see why it is not safe to assign a value of type `List<Integer>` to a variable of type `List<Number>`, consider the following code:

org/jgcbook/chapter02/A_subtyping_and_the_substitution_principle/Program_2

```
List<Integer> ints = new ArrayList<>();
List<Number> nums = ints;           // can't be allowed
nums.add(3.14);
```

This code assigns the variable `ints` to point to a list of integers, and then assigns `nums` to point to the *same* list. If the Substitution Principle were to require that to succeed, the call in the following line—which is clearly legal—would add a `Double` to this list, meaning that the variable `ints`, typed as a `List<Integer>`, would be pointing at a list containing a `Double`, and Java’s type system would be broken. Obviously, this can’t be allowed; the solution is to outlaw such an assignment. If `List<Integer>` is prevented from being a subtype of `List<Number>`, then the Substitution Principle does not apply and the second assignment statement can be reported as a compile error.

Can we take `List<Number>` to be a subtype of `List<Integer>`? That doesn’t work either, as shown by the following code:

org/jgcbook/chapter02/A_subtyping_and_the_substitution_principle/Program_3

```
List<Number> nums = new ArrayList<>();
nums.add(3.14);
List<Integer> ints = nums;           // can't be allowed
```

where allowing the second assignment would lead to the same problem—once again, a variable typed as `List<Integer>` is pointing to a list containing a `Double`. So `List<Number>` is not a subtype of `List<Integer>`, and since we’ve already seen that `List<Integer>` is not a subtype of `List<Number>`, all we have is the trivial case where `List<Integer>` is a subtype of itself. (We also have the more intuitive relationship that `List<Integer>` is a subtype of `Collection<Integer>`.)

Arrays behave quite differently; with them, `Integer[]` is a subtype of `Number[]`. We will compare the treatment of lists and arrays later (see “[Arrays](#)” on page 24).

Sometimes we would like lists to behave more like arrays, in that we want to accept not only a list with elements of a given type, but also a list with elements of any subtype of a given type. For this purpose, we use *wildcards*.

Wildcards

Another `Collection` method is `containsAll`, which tests whether every member of a supplied collection also belongs to this one:

```
interface Collection<E> {  
    ...  
    public boolean containsAll(Collection<?> c);  
    ...  
}
```

The symbol `?` is the wildcard; it denotes some type that is *unknown but fixed*. In a call of `containsAll` you could supply a `Collection<String>`, a `Collection<Number>`, or a `Collection` of any other type (including `Collection<Object>`). If you can supply a `Collection<String>`, the Substitution Principle tells us that you can also supply a `List<String>` or a `Set<String>`. You may be wondering why the definition of `containsAll` allows you to test whether every member of a `Collection<String>` is also a member of a `Collection<Number>`. We explore this question later, in “[Object Versus E](#)” on page 324.

Given a `Collection<?>`, what can you say about the compile-time type of its elements? The answer is “very little”: the wildcard definition says that it is an unknown type, so all you can say about it is what you know about every Java type: it is a subtype of `Object`. That means, for example, that you can’t put an element of any type (except `null`) into a `Collection<?>`. To do that while maintaining its type consistency, you would have to know the element type, which is exactly what an *unbounded wildcard*—that is, `?`—cannot tell you. *Bounded wildcards* provide more information. There are two kinds: wildcards with `extends` and wildcards with `super`.

Wildcards with extends

Another method in the `Collection` interface is `addAll`, which adds all members of one collection to another:

```
interface Collection<E> {  
    ...  
    public boolean addAll(Collection<? extends E> c);  
    ...  
}
```

Clearly, given a collection of elements of type `E`, we should be able to add all members of another collection with elements of type `E`. Given a `Collection<Number>`, for example, we should be able to add all the elements of another collection of `Number`. It would also be safe to add all the elements of a collection of `Integer`, or of `Long`, because any values of these types are also `Numbers`. But we shouldn’t be able to add the elements of a `Collection<Object>` or a `Collection<String>`, because their values

may not (or certainly will not) be instances of `Number`. So when writing the type of the collection that will be the argument to `addAll`, what's needed is a way to express the constraint that its element type should be some subtype of `Number`. That is what the *bounded wildcard* `? extends E` means: like the unbounded wildcard `?`, it denotes a fixed type, but now, instead of it being totally unknown, we know that it must be a subtype of `E`. A wildcard like this is said to be *bounded above* (see [Figure 2-1](#)).

The method `addAll` is expecting a `Collection` of elements that are some subtype of `E`. What types can actually be supplied to it? For a concrete example, imagine a method declared with a parameter of type `Collection<? extends Number>`: it could accept a `Collection<Integer>`, say, or a `Collection<Double>`. So these are subtypes of `Collection<? extends Number>`. In general, if `F` is a subtype of `E`, then `Collection<F>` is a subtype of `Collection<? extends E>`.

Here is an example. We create an empty list of numbers, and add to it first a list of integers and then a list of doubles:

[org/jgcbook/chapter02/C_wildcards_with_extends/Program_1](#)

```
List<Number> nums = new ArrayList<>();
List<Integer> ints = List.of(1, 2);
List<Double> dbls = List.of(1.0, 0.5);
nums.addAll(ints);
nums.addAll(dbls);
assert nums.equals(List.of(1, 2, 1.0, 0.5));
```

The first call is permitted because `nums` has type `List<Number>`, which is a subtype of `Collection<Number>`, and `ints` has type `List<Integer>`, which is a subtype of `Collection<Integer>`, which is a subtype of `Collection<? extends Number>`. The second call is similarly permitted. In both calls, `E` is taken to be `Number`. If the parameter declaration for `addAll` had been written without the wildcard, then the calls to add lists of integers and doubles to a list of numbers would not have been permitted; you would only have been able to add a list that was explicitly declared to be a list of numbers.

We can also use wildcards when declaring variables. Here is a variant of the introductory example of the preceding section, changed by adding a wildcard to the second line:

[org/jgcbook/chapter02/C_wildcards_with_extends/Program_2](#)

```
List<Integer> ints = new ArrayList<>();
List<? extends Number> nums = ints; // now legal
nums.add(3.14); // can't be allowed
```

Previously, the second assignment caused a compile-time error (because `List<Integer>` is not a subtype of `List<Number>`), but the call to `add` was fine (because a double is a number, so you can add a double to a `List<Number>`). Now, the

second assignment is fine (because `List<Integer>` is a subtype of `List<? extends Number>`), but the call to `add` causes a compile error. You cannot add a double to a `List<? extends Number>`, since you don't know the type represented by the wildcard; all you know is that it is some subtype of `Number`. If instead this were allowed, as before, the last line would result in the variable `ints`, typed as a `List<Integer>`, pointing to a list containing a double.

Since every type in Java is a subtype of `Object`, you could say that “unbounded” is a misnomer: `Collection<?>` is really an abbreviation for `Collection<? extends Object>`. But this special case is so important and so frequently used that the term “unbounded” and its abbreviation are universally used.

In general, if a structure contains elements with a type of the form `? extends E`, we can get elements out of the structure, but we cannot put elements into the structure. To put elements into the structure we need another kind of wildcard, as explained in the next section.

Wildcards with super

Here is a method, from the convenience class `Collections`, that copies the elements of a source list into a destination list:

```
org/jcbook/chapter02/D\_wildcards\_with\_super/Program\_1
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    for (int i = 0; i < src.size(); i++) {
        dest.set(i, src.get(i));
    }
}
```

The phrase `? super T` means that the element type of the destination list may be any type that is a *supertype* of `T`, just as the source list may have elements of any type that is a *subtype* of `T`. A wildcard like this is said to be *bounded below* (see [Figure 2-1](#)).

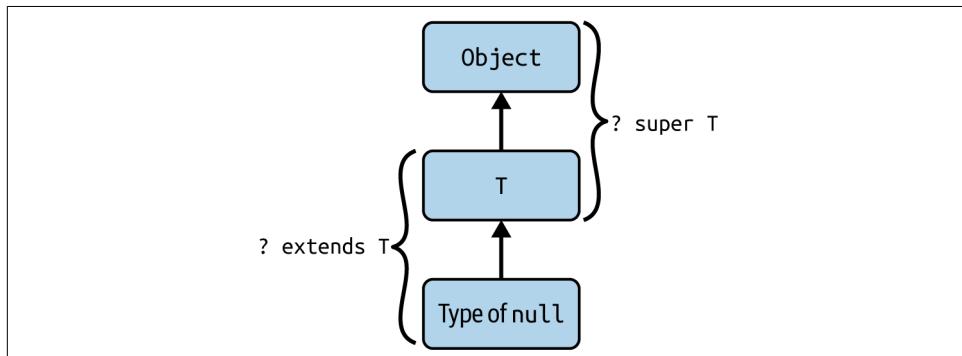


Figure 2-1. Bounded types: extends and super

Here is a sample call:

```
org/jcbook/chapter02/D_wildcards_with_super/Program_1
```

```
List<Object> objs = Stream.of(2, 3.14, "four")
    .collect(Collectors.toCollection(ArrayList::new));
List<Integer> ints = List.of(5, 6);
Collections.copy(objs, ints);
assert objs.equals(List.of(5, 6, "four"));
```

As with any generic method, the type parameter may be inferred or may be given explicitly. In this case, there are four possible choices, all of which type-check and all of which have the same effect:

```
org/jcbook/chapter02/D_wildcards_with_super/Program_1
```

```
Collections.copy(objs, ints);
Collections.<Object>copy(objs, ints);
Collections.<Number>copy(objs, ints);
Collections.<Integer>copy(objs, ints);
```

The first call leaves the type parameter implicit; it is taken to be `Integer`, since that is the most specific choice that works. In the third line, it is taken to be `Number`. The call is permitted because `objs` has type `List<Object>`, which is a subtype of `List<? super Number>` (since `Object` is a supertype of `Number`, as required by the `super` wildcard) and `ints` has type `List<Integer>`, which is a subtype of `List<? extends Number>` (since `Integer` is a subtype of `Number`, as required by the `extends` wildcard).

We could also declare the method with several possible *signatures* (a method signature is the combination of the method identifier, its type parameters, and the types and order of its parameters):²

```
public static <T> void copy(List<T> dst, List<T> src)
public static <T> void copy(List<T> dst, List<? extends T> src)
public static <T> void copy(List<? super T> dst, List<T> src)
public static <T> void copy(List<? super T> dst, List<? extends T> src)
```

The first of these is too restrictive, as it only permits calls when the destination and source have exactly the same type. The remaining three are equivalent for calls that use implicit type parameters, but differ for explicit type parameters. For the example calls shown earlier, the second signature works only when the type parameter is `Object`, the third signature works only when the type parameter is `Integer`, and the last signature works (as we have seen) for all three type parameters—`Object`, `Number`, and `Integer`. When writing a method signature, always use wildcards where you can, since this permits the widest range of calls.

² For type parameters to distinguish two method signatures, the number of parameters or their bounds must differ; simple renaming is not enough.

The Get and Put Principle

It may be good practice to insert wildcards whenever possible, but how do you decide *which* wildcard to use? Where should you use `extends`, where should you use `super`, and where is it inappropriate to use a wildcard at all?

Fortunately, a simple principle determines which is appropriate:

Get and Put Principle: Use an `extends` wildcard when you only *get* values out of a structure, use a `super` wildcard when you only *put* values into a structure, and don't use a wildcard when you *both* get and put.

In *Effective Java*, Joshua Bloch (2017, item 31) gives this principle the mnemonic PECS (producer-extends, consumer-super). We already saw it at work in the signature of the `copy` method:

```
public static <T> void copy(List<? super T> dst, List<? extends T> src)
```

The method gets values out of the source `src`, so it is declared with an `extends` wildcard, and it puts values into the destination `dst`, so it is declared with a `super` wildcard.

Whenever you use an iterator or a stream, you are getting values out of a structure, so you must use an `extends` wildcard. Here is the iterator version of a method that takes a collection of numbers, converts each to a double, and sums them up:

```
org/jcbook/chapter02/E\_the\_get\_and\_put\_principle/Program\_1
public static double sum(Collection<? extends Number> nums) {
    double s = 0.0;
    for (Number num : nums) s += num.doubleValue();
    return s;
}
```

The stream equivalent doesn't declare a `Number` variable explicitly, but the type constraint on the argument is exactly the same:

```
org/jcbook/chapter02/E\_the\_get\_and\_put\_principle/Program\_2
public static double sum(Collection<? extends Number> nums) {
    return nums.stream().mapToDouble(Number::doubleValue).sum();
}
```

Since these methods use `extends`, the following calls are all legal:

```
org/jcbook/chapter02/E\_the\_get\_and\_put\_principle/Program\_2
List<Integer> ints = List.of(1, 2, 3);
assert sum(ints) == 6.0;

List<Double> doubles = List.of(2.5, 3.5);
assert sum(doubles) == 6.0;
```

```
List<Number> nums = List.of(1, 2, 2.5, 3.5);
assert sum(nums) == 9.0;
```

The first two calls would not be legal if `extends` were not used.

Whenever you use a method like `add` that puts values into a structure, you should use a `super` wildcard. Here is a method that takes a collection of some supertype of `Integer` and an integer `n` and puts the first `n` integers, starting from zero, into the collection:

```
org/jcbook/chapter02/E_the_get_and_put_principle/Program_3
public static void storeIntegers(Collection<? super Integer> ints, int n) {
    for (int i = 0; i < n; i++) ints.add(i);
}
```

There is no simple stream-based equivalent to this method.³ Since the parameter to this method uses `super`, the following calls are all legal:

```
org/jcbook/chapter02/E_the_get_and_put_principle/Program_3
List<Integer> ints1 = new ArrayList<>();
storeIntegers(ints1, 5);
assert ints1.equals(List.of(0, 1, 2, 3, 4));

List<Number> nums1 = new ArrayList<>();
storeIntegers(nums1, 5);
nums1.add(5.0);
assert nums1.equals(List.of(0, 1, 2, 3, 4, 5.0));

List<Object> objs1 = new ArrayList<>();
storeIntegers(objs1, 5); objs1.add("five");
assert objs1.equals(List.of(0, 1, 2, 3, 4, "five"));
```

The last two calls would not be legal if `super` were not used.

Whenever you both put values into and get values out of the same structure, as in this example, you should not use a wildcard:

```
org/jcbook/chapter02/E_the_get_and_put_principle/Program_4
public static double sumValues(Collection<Number> nums, int n) {
    storeIntegers(nums, n);
    return sum(nums);
}
```

³ You may be tempted to think that you could write `IntStream.range(0,n).forEach(ints::add)`; as a simple equivalent—but don't give in to this temptation! Used like this, `Stream::forEach` will break under concurrent access. Even if you're certain that you have no concurrency issues now, your code should still be resilient to future refactoring that introduces concurrency, for example by someone iterating over the target collection. Using a collector future-proofs your code and is an investment in a better coding style. For the detailed reasons, see Naftalin, 2015.

The collection is passed to both `sum` and `storeIntegers`, so its element type must both extend `Number` (as required by `sum`) and be a supertype of `Integer` (as required by `storeIntegers`). The only two classes that satisfy both of these constraints are `Number` and `Integer`; we have picked the first of these.

Here is a sample call:

org/jgbook/chapter02/E_the_get_and_put_principle/Program_4

```
List<Number> nums2 = new ArrayList<>();
double sum = sumValues(nums2, 5);
assert sum == 10;
```

Since there is no wildcard, the argument must be a collection of `Number`.

If you don't like having to choose between `Number` and `Integer`, it might occur to you that if Java let you write a wildcard with both `extends` and `super`, you would not need to choose. For instance, you might think we could write the following:

```
double sumValues(Collection<? extends Number super Integer> coll, int n)
// not legal Java!
```

Then we could call `sumValues` on either a collection of numbers or a collection of integers. But Java *doesn't* permit this. The only reason for outlawing it is simplicity, and conceivably Java might support such notation in the future. However, for now, if you need to both get and put, then don't use wildcards.

The Get and Put Principle also works the other way around. If an `extends` wildcard is present, pretty much all you will be able to do is get but not put values of that type; and if a `super` wildcard is present, pretty much all you will be able to do is put but not get values of that type. For example, consider the following code, which uses a list declared with an `extends` wildcard:

org/jgbook/chapter02/E_the_get_and_put_principle/Program_5

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
double dbl = sum(nums);      // ok
nums.add(3.14);             // compile-time error
```

The call to `sum` is fine, because it gets values from the list, but the call to `add` is not, because it puts a value into the list. This is just as well, since otherwise we could add a double to a list of integers!

Conversely, consider the following code, which uses a list declared with a `super` wildcard:

[org/jgcbook/chapter02/E_the_get_and_put_principle/Program_6](#)

```
List<Object> objs = new ArrayList<>();
objs.add(1);
objs.add("two");
List<? super Integer> ints = objs;
ints.add(3);           // ok
double dbl = sum(ints); // compile-time error
ints.add("three");    // compile-time error
```

Now the first call to `ints.add` is fine, because it puts a value into the list, but the call to `sum` is not, because it gets a value from the list. Again, this is just as well, because the sum of a list containing a string makes no sense! The last call to `add` is also a compile error because, although `ints` refers to an `ArrayList<Object>`, it is the compile-time type that matters, and the Get and Put Principle dictates that you can only add an `Integer` (strictly, a subtype of `Integer`) to a list variable with the compile-time type of `List<? super Integer>`.

You may find it helpful to think of `? extends T` as being an unknown type in an interval bounded by the type of `null` below and by `T` above (where the type of `null` is a subtype of every reference type). Similarly, you may think of `? super T` as being an unknown type in an interval bounded by `T` below and by `Object` above (see [Figure 2-1](#)).

When you are putting a value into either of these intervals, it must belong to the type at the bottom. So the one value that you can put into a type declared with an `extends` wildcard is `null`, the only value that belongs to every reference type:

[org/jgcbook/chapter02/E_the_get_and_put_principle/Program_7](#)

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(null); // ok
assert nums.equals(Arrays.asList(1, 2, null));
```

And the only values that you can get from either of these intervals must belong to the type at the top. This means that the only values that you can get from a type declared with a `super` wildcard are those of type `Object`, which is the supertype of every reference type:

[org/jgcbook/chapter02/E_the_get_and_put_principle/Program_8](#)

```
List<Object> objs = List.of(1, "two");
List<? super Integer> ints = objs;
String str = "";
for (Object obj : ints) str += obj.toString();
assert str.equals("1two");
```

It is tempting to think that an `extends` wildcard ensures immutability, or at least unmodifiability, but it does not. As we saw earlier, given a list of type `List<? extends Number>`, you can still add `null` values to the list. You can also remove list elements (using `remove`, `removeAll`, `retainAll`, or `clear`) or permute the list (using `swap`, `sort`, or `shuffle` in the convenience class `Collections`; see “[Changing the Order of List Elements](#)” on page 276). The problems and advantages of achieving unmodifiability or immutability are explored in “[Immutability and Unmodifiability](#)” on page 147 and further in [Chapter 18](#).

Because `String` is final and can have no subtypes, you might expect that `List<String>` is the same type as `List<? extends String>`. But in fact the former is a subtype of the latter, not the same type, as can be seen by an application of our principles. The Substitution Principle tells us it is a subtype, because it is fine to pass a value of the former type where the latter is expected. The Get and Put Principle tells us that it is not the same type, because we can add a string to a value of the former type but not the latter.

Arrays

It is instructive to compare the treatment of lists and arrays in Java, keeping in mind the Substitution Principle and the Get and Put Principle.

In Java, array subtyping is *covariant*, meaning that type `S[]` is a subtype of `T[]` whenever `S` is a subtype of `T`. Consider the following code fragment, which allocates an array of integers, assigns it to a variable typed as a `Number` array, and then attempts to store a double in the array:

[org/jgcbook/chapter02/F_arrays/Program_1](#)

```
Integer[] ints = {0};  
Number[] nums = ints;  
nums[0] = 3.14;      // array store exception (subtype of RuntimeException)
```

Something is wrong with this program since, if it were to succeed, the variable `ints`, typed as an `Integer[]`, would be pointing at an array containing a `Double`, and once again Java’s type system would be broken. Where is the problem? Since `Integer[]` is a subtype of `Number[]`, according to the Substitution Principle the assignment on the second line must be legal. Instead, the problem is caught at run time on the third line. When an array is allocated (as on the first line), it is tagged with its reified type (a run-time representation of its component type, in this case `Integer`), and every time a value is stored in the array (as on the third line), an array store exception is raised if the reified type is not compatible with the assigned value (in this case, a double cannot be stored into an array of integers).

In contrast, the subtyping relation for generics is *invariant*, meaning that type `List<S>` is *not* a subtype of `List<T>`, except in the trivial case where `S` and `T` are identical. Here is a code fragment analogous to the preceding one, with lists replacing arrays:

[org/jgcbook/chapter02/F_arrays/Program_2](#)

```
List<Integer> ints = Arrays.asList(0);
List<Number> nums = ints; // compile-time error
nums.set(0, 3.14);
```

Since `List<Integer>` is not a subtype of `List<Number>`, the problem is detected on the second line, not the third, and it is detected at compile time, not run time.

Wildcards reintroduce covariant subtyping for generics, in that type `List<S>` is a subtype of `List<? extends T>` when `S` is a subtype of `T`. Here is a third variant of the fragment:

[org/jgcbook/chapter02/F_arrays/Program_3](#)

```
List<Integer> ints = Arrays.asList(0);
List<? extends Number> nums = ints;
nums.set(0, 3.14); // compile-time error
```

As with arrays, the third line is in error, but here, the problem is detected at compile time, not run time. The assignment violates the Get and Put Principle, because you cannot put a value into a type declared with an `extends` wildcard.

Wildcards also introduce *contravariant* subtyping for generics, in that type `List<S>` is a *subtype* of `List<? super T>` when `S` is a *supertype* of `T` (as opposed to a subtype). Arrays do not support contravariant subtyping. For instance, recall the method `storeIntegers`:

[org/jgcbook/chapter02/F_arrays/Program_4](#)

```
public static void storeIntegers(Collection<? super Integer> ints, int n) {
    for (int i = 0; i < n; i++) ints.add(i);
}
```

This accepts a parameter of type `Collection<? super Integer>` and fills it with integers. There is no equivalent way to do this with an array, since Java does not permit you to write `(? super Integer)[]`.

Detecting problems at compile time rather than at run time brings two advantages, one minor and one major. The minor advantage is that it is more efficient. The system does not need to carry around a description of the element type at run time, and the system does not need to check against this description every time an assignment into an array is performed. The major advantage is that a common family of errors is detected by the compiler. This improves every aspect of the program's life cycle: coding, debugging, testing, and maintenance are all made easier, quicker,

and less expensive. Apart from the fact that typing errors are caught earlier, there are many other reasons to prefer collection classes to arrays:

- Collections are more flexible than arrays; the only operations supported on arrays are to get or set a component, and the number of elements they can contain is fixed.
- Collections support many additional operations, including testing for containment, adding and removing elements, and combining two collections.
- Collections may be lists (where order is significant, the same value may recur multiple times, and further operations are available), sets (where order is not significant and elements may not recur), or queues. For each of these data types, a number of representations are available, including arrays, linked lists, trees, and hash tables.
- Specialized collections are available for situations requiring efficient concurrent access.
- Finally, although this is not an inherent advantage of collections over arrays, the convenience class `Collections` offers operations to rotate or shuffle a list, to find the maximum of a collection, to make a collection unmodifiable or synchronized, and others—many more than are provided by the corresponding convenience class `Arrays`.

Nonetheless, there are situations in which arrays are preferable to collections. In particular, arrays of primitive types are much more efficient than either arrays or collections of reference types, since they don't involve boxing and they use less memory with much better spatial locality (see “[Performance](#)” on page 142). Further, assignments into a primitive array need not check for an array store exception, because arrays of primitive types do not have subtypes. These advantages may lead you to replace collections with arrays in performance-critical code sections. As always, you should measure comparative performance to justify such a design, bearing in mind that future compiler and library design improvements may change the relative performance balance. Finally, in some cases arrays may be preferable for reasons of compatibility.

To summarize, it is better to detect errors at compile time rather than run time, but Java arrays are forced to detect certain errors at run time by the decision to make array subtyping covariant. Was this a good decision? Before the advent of generics, it was absolutely necessary. For instance, look at the following methods, which are used to sort any array or to fill an array with a given value:

```
public static void sort(Object[] a);
public static void fill(Object[] a, Object val);
```

Thanks to covariance, these methods can be used to sort or fill arrays of any reference type. Without covariance and without generics, there would have been no way to declare methods that apply for all types. With generics, however, covariant arrays became unnecessary, since these methods could now have the following signatures, directly stating that they work for all types:

```
public static <T> void sort(T[] a);
public static <T> void fill(T[] a, T val);
```

In some sense, covariant arrays are an artifact of the lack of generics in earlier versions of Java. Now that we have generics, covariant arrays are probably the wrong design choice, and the only reason for retaining them is backward compatibility. The differing type systems of generics and arrays lead to a variety of inconvenient interactions, which we will explore in detail in [Chapter 5](#).

Bounded or Unbounded?

Earlier in this chapter, we saw the `Collection` method `containsAll`, which checks whether this collection contains every element of another one. It is a generalization of another `Collection` method, `contains`, which checks whether this collection contains a single given object. This section presents two alternative approaches to giving generic signatures for these methods. The first approach uses unbounded wildcards and is the one used in the Java Collections Framework. The second, more precise, approach uses wildcards bounded by the type parameter of the interface.

Using an Unbounded Wildcard

Here are the types that the `contains` methods have in Java with generics:

```
interface Collection<E> {
    ...
    public boolean contains(Object o);
    public boolean containsAll(Collection<?> c);
    ...
}
```

The parameter of the second method is consistent with the first, because the unbounded wildcard `?` actually stands for `? extends Object`.

These methods let us test for membership and containment:

[org/jgcbook/chapter02/G_wildcards_versus_type_parameters/Program_1](#)

```
Object obj = "one";
List<Object> objs = List.of("one", 2, 3.5, 4);
List<Integer> ints = List.of(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert ! ints.contains(obj);
assert ! ints.containsAll(objs);
```

The given list of objects contains both the string "one" and the given list of integers, but the given list of integers does not contain the string "one", nor does it contain the given list of objects.

The tests `ints.contains(obj)` and `ints.containsAll(objs)` might seem silly. Of course, a list of integers won't contain an arbitrary object, such as the string "one". But it is permitted because sometimes such tests might succeed:

org/jcbook/chapter02/G_wildcards_versus_type_parameters/Program_2

```
Object obj = 1;
List<Object> objs = List.of(1, 3);
List<Integer> ints = List.of(1, 2, 3, 4);
assert ints.contains(obj);
assert ints.containsAll(objs);
```

In this case, the object may be contained in the list of integers because it happens to be an integer, and the list of objects may be contained within the list of integers because every object in the list happens to be an integer.

In “[Object Versus E](#)” on page 324, we’ll explore in more detail the reasons why the generics designers chose these parameter types for `contains` and `containsAll`, as well as for other methods in `Collection`, `List`, and `Map` that test for, or remove, values from the collection. We will see that there are arguments both for and against this choice.

Using a Bounded Wildcard

You might reasonably choose an alternative design for collections—a design in which you can only test containment for subtypes of the element type:

```
interface MyCollection<E> { // alternative design
    ...
    public boolean contains(E o);
    public boolean containsAll(Collection<? extends E> c);
    ...
}
```

Say we have a class `MyList` that implements `MyCollection`. Now only one direction is legal for the tests:

```
Object obj = "one";
MyList<Object> objs = MyList.of("one", 2, 3.5, 4);
MyList<Integer> ints = MyList.of(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert ! ints.contains(obj);           // compile-time error
assert ! ints.containsAll(objs);      // compile-time error
```

The last two tests are illegal, because the type declarations now dictate that we can only test whether a list contains an element of a subtype of that list. So we can check whether a list of objects contains a list of integers, but not the other way around.

Wildcard Capture

One aspect of wildcards can seem confusing at first sight. Normally, it's safe to assume that two types written in identical form are the same type. That is true of type constants like `Integer`, and type variables like `E` in the previous section. But it is not true of wildcard types. Consider:

```
List<? extends Number> list1;
List<? extends Number> list2;
```

We know that each occurrence of `? extends Number` denotes some unknown subtype of `Number`. So for example, `list1` could be assigned to a `List<Integer>` and `list2` to a `List<Double>`. The two occurrences of `? extends Number` represent potentially *different* types, so the assignment:

```
list1.add(list2.get(0));
```

results in this message from JShell at Java 21:

```
incompatible types: java.lang.Number cannot be converted \
to capture#1 of ? extends java.lang.Number
```

It has given the expression `list2.get(0)` the type `Number`, because that is the most precise information available. But now it's unable to add a value of that type to `list1`, because the element type of `list1` is some unknown subtype of `Number`. That is what the error message is saying. But why is the unknown type called `capture#1` of `? extends java.lang.Number`?

The answer is that in the normal process of compilation, the compiler infers a type for every wildcard: this is called *wildcard capture*, and it is usually invisible to the programmer. But in cases like this one, the compiler tells you the internal name that it has given to this type so that you can see the type incompatibility problem in your code.

For a more substantial example, consider the method `reverse` in the `Collections` class, which accepts a list of any type and reverses it. It has the following declaration:

```
public static void reverse(List<?> list);
```

A naïve attempt at an implementation might attempt to copy the argument into a temporary list and then write from the copy back into the original in reverse order:

org/jgcbook/chapter02/H_wildcard_capture/Program_1

```
public static void reverse(List<?> list) {
    List<Object> tmp = new ArrayList<>(list);
```

```

for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1)); // compile-time error
}
}

```

This fails because it is not legal to write from the copy back into the original: we are trying to write from a list of objects into a list of unknown type. Compiled with OpenJDK 21, this version of `reverse` generates the following error message (a rather clearer version of the preceding JShell message):

```

Capture.java:8: error: incompatible types: Object cannot be converted to CAP#1
        list.set(i, tmp.get(list.size()-i-1)); // compile-time error
                                         ^
where CAP#1 is a fresh type-variable: CAP#1 extends Object from capture of ?

```

`CAP#1` (short for “`capture#1`”) is the name that the compiler has given to the type of the elements of `list`. If there is more than one distinct wildcard, each represents a different unknown type, so they will be assigned different names even if the type associated with each is the same—for example, `capture of ?` or, in the case of a bounded wildcard, `capture of ? extends Number`.

In this case the error is annoying rather than helpful, as the operation the code is attempting to perform is safe. For a way of circumventing the problem, see “[Use Generic Helper Methods to Capture a Wildcard](#)” on page 120.

Restrictions on Wildcards

Wildcards may not appear at the top level in class instance creation expressions (`new`), in explicit type parameters in generic method calls, or in supertypes (`extends` and `implements`). Let’s consider each of these cases in turn.

Instance Creation

In a class instance creation expression, if the type is a parameterized type, then none of the type parameters may be wildcards. For example, the following are illegal:

```

org/jgcbook/chapter02/l\_restrictions\_on\_wildcards/Program\_1
List<?> list = new ArrayList<?>(); // compile-time error
Map<String, ? extends Number> map =
    new HashMap<String, ? extends Number>(); // compile-time error

```

This is usually not a hardship. The Get and Put Principle tells us that if a structure contains a wildcard, we should only get values out of it (if it is an `extends` wildcard) or only put values into it (if it is a `super` wildcard). For a structure to be useful, we must do both. Therefore, we usually create a structure at a precise type, even if we use wildcard types to put values into or get values from the structure, as in the following example:

```
List<Number> nums = new ArrayList<Number>();
List<? super Number> sink = nums;
List<? extends Number> source = nums;
for (int i=0; i<5; i++) sink.add(i);
int sum = source.stream().mapToInt(Number::intValue).sum();
assert sum == 10;
```

Here, wildcards appear in the second and third lines, but not in the first line that creates the list.

Only top-level parameters in instance creation are prohibited from containing wildcards. Nested wildcards are permitted. Hence, the following is legal:

```
List<List<?>> lists = new ArrayList<List<?>>();
lists.add(List.of(1,2,3));
lists.add(List.of("four","five"));
assert lists.equals(List.of(List.of(1, 2, 3), List.of("four", "five")));
assert lists.getFirst().getFirst().toString().equals("1");
```

Even though the list of lists is created at a wildcard type, each individual list within it has a specific type: the first is a list of integers, and the second is a list of strings. The wildcard type prohibits us from extracting elements from the inner lists as any type other than `Object`; on the last line of the example, that is the type of the expression `lists.getFirst().getFirst()`.

One way to remember the restriction is that the relationship between wildcards and ordinary types is similar to the relationship between interfaces and classes—wildcards and interfaces are more general, ordinary types and classes are more specific, and instance creation requires the more specific information. Consider the following three statements:

```
List<?> list1 = new ArrayList<Object>(); // ok
List<?> list2 = new List<Object>(); // compile-time error
List<?> list3 = new ArrayList<?>(); // compile-time error
```

The first is legal; the second is illegal because an instance creation expression requires a class, not an interface; and the third is illegal because an instance creation expression requires an ordinary type, not a wildcard.

You might wonder why this restriction is necessary. The Java designers had in mind that every wildcard type is shorthand for some ordinary type, so they believed that ultimately every object should be created with an ordinary type. It is not clear whether this restriction is necessary, but it is unlikely to be a problem. (We tried hard to contrive a situation in which it was a problem, and we failed!)

Generic Method Calls

If a generic method call includes explicit type parameters, those type parameters must not be wildcards. For example, say we have the following generic method:

```
org/jgcbook/chapter02/1_restrictions_on_wildcards/Lists  
class Lists {  
    public static <T> List<T> factory() { return new ArrayList<T>(); }  
}
```

You may choose for the type parameters to be inferred, or you may pass an explicit type parameter. Both of the following are legal:

```
org/jgcbook/chapter02/1_restrictions_on_wildcards/Lists  
List<?> list1 = Lists.factory();  
List<?> list2 = Lists.<Object>factory();
```

If an explicit type parameter is passed, it must not be a wildcard:

```
org/jgcbook/chapter02/1_restrictions_on_wildcards/Lists  
List<?> list3 = Lists.<?>factory(); // compile-time error
```

As before, nested wildcards are permitted:

```
org/jgcbook/chapter02/1_restrictions_on_wildcards/Lists  
List<List<?>> list4 = Lists.<List<?>>factory(); // ok
```

The motivation for this restriction is similar to the previous one. Again, it is not clear whether it is necessary, but it is unlikely to be a problem.

Supertypes

When a class instance is created, it invokes the constructor for its supertype. Hence, any restriction that applies to instance creation must also apply to supertypes. In a class declaration, if the supertype or any superinterface has type parameters, these types must not be wildcards.

For example, this declaration is illegal:

```
class AnyList extends ArrayList<?> {...} // compile-time error
```

And so is this:

```
class AnotherList implements List<?> {...} // compile-time error
```

But, as before, nested wildcards are permitted:

```
class NestedList extends ArrayList<List<?>> {...} // ok
```

The motivation for this restriction is similar to the previous two. As before, it is not clear whether it is necessary, but it is unlikely to be a problem.

Conclusion

In this chapter, we saw the different subtyping rules for arrays and generics: arrays are covariant, generics are not. This means that the compiler can detect errors in generic programs that would only appear at run time using arrays. To replace covariant subtyping, generics use bounded wildcards to make use of the Substitution Principle. The Get and Put Principle captures the relationship between a structure's bounded type and its ability to accept or provide values in the range of its type bound.

In [Chapter 3](#), we'll learn how generic typing works with the interfaces `Comparable<T>` and `Comparator<T>`, which are central to practical Java programming.

Comparison and Bounds

Now that we have the basics, let's look at some more advanced uses of generics. This chapter describes the interfaces `Comparable<T>` and `Comparator<T>`, which are used to support comparison of objects by their elements. These interfaces are useful if, for instance, you want to find the maximum element of a collection or to sort a list. Along the way, we will introduce bounds on type variables, an important feature of generics that is particularly useful in combination with the `Comparable<T>` interface.



The examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/
main/src/main/java/org/jgcbook/chapter03](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter03)

Comparable

The interface `Comparable<T>` declares a single instance method for comparing one object with another:

```
interface Comparable<T> {  
    public int compareTo(T other);  
}
```

The `compareTo` method returns an integer value that is negative, zero, or positive depending upon whether the receiver—this object—is less than, equal to, or greater than the argument. When a class implements `Comparable`, the ordering specified by this interface is called the *natural ordering* for that class.

Typically, an object belonging to a class can only be compared with an object belonging to the same class. For instance, `Integer` implements `Comparable<Integer>`:

[org/jcbook/chapter03/A_comparable/Program_1](#)

```
Integer int0 = 0;
Integer int1 = 1;
assert int0.compareTo(int1) < 0;
```

This comparison returns a negative number, since 0 precedes 1 under numerical ordering. Similarly, `String` implements `Comparable<String>`:

[org/jcbook/chapter03/A_comparable/Program_2](#)

```
String str0 = "zero";
String str1 = "one";
assert str0.compareTo(str1) > 0;
```

This comparison returns a positive number, since "zero" follows "one" under alphabetic ordering.

The type parameter to the interface allows nonsensical comparisons to be caught at compile time:

[org/jcbook/chapter03/A_comparable/Program_3](#)

```
Integer i = 0;
String s = "one";
assert i.compareTo(s) < 0; // compile-time error
```

You can compare an integer with an integer or a string with a string, but attempting to compare an integer with a string is a compile-time error:

```
Snippet_3.java:8: error: incompatible types: String cannot be converted to Integer
        assert i.compareTo(s) < 0; // compile-time error
                           ^
```

Comparison is not supported between arbitrary numerical types:

[org/jcbook/chapter03/A_comparable/Program_4](#)

```
Number m = Integer.valueOf(2);
Number n = Double.valueOf(3.14);
assert m.compareTo(n) < 0; // compile-time error
```

Here, the comparison is illegal because the `Number` class does not implement the `Comparable` interface.

Comparison differs from equality in that it does not accept a `null` argument. If `x` is not `null`, `x.equals(null)` must return `false`, while `x.compareTo(null)` must throw a `NullPointerException`.

We adapt standard idioms for comparison, writing `x.compareTo(y) < 0` instead of `x < y` and writing `x.compareTo(y) <= 0` instead of `x <= y`.

The Contract for Comparable

The contract¹ for the Comparable<T> interface specifies three properties. The properties are defined using the sign function, which is defined such that `sgn(x)` returns -1, 0, or 1, depending on whether `x` is negative, zero, or positive.

First, comparison is *antisymmetric*. Reversing the order of the arguments reverses the result:

```
sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
```

This generalizes the property for numbers: $x < y$ if and only if $y > x$. It is also required that `x.compareTo(y)` raises an exception if and only if `y.compareTo(x)` raises an exception. Taking `x` and `y` to be the same gives us:

```
sgn(x.compareTo(x)) == -sgn(x.compareTo(x))
```

It follows that:

```
x.compareTo(x) == 0
```

so comparison is reflexive—that is, every value compares as the same as itself.

Second, comparison is *transitive*. If one value is smaller than a second, and the second is smaller than a third, then the first is smaller than the third:

```
if x.compareTo(y) < 0 and y.compareTo(z) < 0 then x.compareTo(z) < 0
```

This generalizes the property for numbers: if $x < y$ and $y < z$ then $x < z$.

Third, comparison is a *congruence*. This means that if two values compare as the same, then they compare the same way with any third value:

```
if x.compareTo(y) == 0 then sgn(x.compareTo(z)) == sgn(y.compareTo(z))
```

This generalizes the property for numbers: if $x == y$ then $x < z$ if and only if $y < z$. Presumably, it is also required that if `x.compareTo(y) == 0` then `x.compareTo(z)` raises an exception if and only if `y.compareTo(z)` raises an exception, although this is not explicitly stated.

Consistent with equals

The contract for Comparable strongly recommends that the `compareTo` method should be *consistent with equals*—that is, that two objects should satisfy the `equals` method if and only if they compare as the same:

```
x.equals(y) if and only if x.compareTo(y) == 0
```

¹ In software, a contract specifies the preconditions that a client (e.g., the caller of a method) must fulfill, and the action that a service must take (e.g., the values that the method returns). See “Contracts” on page 149.

Notice that this is a recommendation, not a mandatory part of the contract. But in fact, it is the normal case: most classes having a natural ordering do comply with this recommendation. If you are designing a class that implements Comparable, you should not ignore it without good reason. One such reason is that different representations of the same value may influence the results of computations. The best-known example of this in the platform library is `java.math.BigDecimal`: two instances of this class representing the same value but with different precisions, for example `4.0` and `4.00`, will compare as the same but not satisfy the `equals` method. One consequence of this design choice is that internally ordered collections like `NavigableSet` and `NavigableMap` behave differently from “typical” sets when they contain such values, as we will see in the introduction to [Chapter 12. “Inconsistent with equals” on page 322](#) contains a discussion of possible reasons for choosing inconsistency with `equals` and the wider consequences of this choice.

Comparing Integral Values

It’s worth pointing out a subtlety in the definition of *comparison*. Suppose that you have a series of `Event` objects, each described by its name and the number of milliseconds at which it occurred before or after the present moment. Here is one way to define the natural order for `Event` to be the same as the timing order:

[org/jgcbook/chapter03/A_comparable/Event_1](#)

```
record Event_1(String name, int millisecs) implements Comparable<Event_1> {  
    public int compareTo(Event_1 other) {  
        return this.millisecs < other.millisecs ? -1  
            : this.millisecs == other.millisecs ? 0  
            : 1;  
    }  
}
```

The conditional expression returns `-1`, `0`, or `1`, depending on whether the receiver is less than, equal to, or greater than the argument. At first sight, you might think that the following code would work just as well, since `compareTo` is permitted to return any negative integer if the receiver is less than the argument and any positive integer if it is greater:

[org/jgcbook/chapter03/A_comparable/Event_2](#)

```
record Event_2(String name, int millisecs) implements Comparable<Event_2> {  
    public int compareTo(Event_2 other) {  
        // bad implementation – don't do this  
        return this.millisecs - other.millisecs;  
    }  
}
```

But if the two values being compared have opposite signs, calculating the difference between them may result in overflow—that is, a value outside the range that can be

stored in an integer, between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. In fact, you don't have to choose between these bad alternatives: the numeric wrappers all expose a static method `compare`. So a better implementation of `Event::compareTo` than either of the two preceding versions is:

```
org/jcbook/chapter03/A\_comparable/Event\_3  
record Event_3(String name, int millisecs) implements Comparable<Event_3> {  
    public int compareTo(Event_3 other) {  
        return Integer.compare(this.millisecs, other.millisecs);  
    }  
}
```

In “[Comparator Methods](#)” on page 47, we will see how the factory methods of `Comparator` provide still more concise and flexible ways of comparing two objects.

The Maximum of a Collection

In this section, we show how to use the `Comparable` interface to find the maximum element in a collection. We begin with a simplified version; the signature of the version actually found in the Collections Framework is more complicated, and later we will see why.

Here is the code to find the maximum element in a nonempty collection, slightly simplified from the class `Collections`:

```
org/jcbook/chapter03/B\_maximum\_of\_a\_collection/Program\_1  
public static <T extends Comparable<T>> T max(Collection<T> coll) {  
    Iterator<? extends T> i = coll.iterator();  
    T candidate = i.next();  
    while (i.hasNext()) {  
        T next = i.next();  
        if (next.compareTo(candidate) > 0)  
            candidate = next;  
    }  
    return candidate;  
}
```

We first saw generic methods that declare new type variables in the signature in “[Generic Methods and Varargs](#)” on page 7. For instance, the method `asList` takes an array of type `E[]` and returns a result of type `List<E>`, and does so for *any* reference type `E`. Here we have a generic method that declares a *bound* on the type variable. The method `max` takes a collection of type `Collection<T>` and returns a `T`, and it does this for *any* type `T` that is a subtype of `Comparable<T>`.

The emphasized phrase in angle brackets declares the type variable `T`. As with wildcards, we say that `T` is bounded by `Comparable<T>`. Also as with wildcards, upper-bounded type variables are always indicated by the keyword `extends`, even when the

bound is an interface rather than a class, as is the case here. Unlike wildcards, type variables can only be bounded using `extends`, not `super`.

In this case, the bound is *recursive*, in that the bound on `T` itself depends upon `T`. It is even possible to have mutually recursive bounds, such as:

```
<T extends C<T,U>, U extends D<T,U>>
```

We explore recursive bounds in more detail in “[Enumerated Types](#)” on page 50.

The body of the method `max` first obtains an iterator over the collection, then calls the `next` method to select the first element as a candidate for the maximum; the specification of this method allows it to throw `NoSuchElementException` if it is supplied with an empty collection. It then compares the candidate with each element in the collection, setting the candidate to the element when the element is larger.

Stream Alternative

The code for `max` at the start of this section, simplified from the current (Java 21) version of the JDK, was written before Java 8 introduced streams and static methods on interfaces, which permit a more concise and probably more efficient version, with the same signature but returning a value of type `Optional<T>` to allow for the case of an empty collection:

```
org/jgcbook/chapter03/B\_maximum\_of\_a\_collection/Program\_2
public static <T extends Comparable<T>> Optional<T> max(Collection<T> coll) {
    return coll.stream().max(Comparator.naturalOrder());
}
```

We will explore the features of `Comparator` that make this possible in “[Comparator](#)” on page 45.

When calling the `Collections` method, `T` may be chosen to be `Integer` (since `Integer` implements `Comparable<Integer>`) or `String` (since `String` implements `Comparable<String>`):

```
org/jgcbook/chapter03/B\_maximum\_of\_a\_collection/Program\_3
List<Integer> ints = Arrays.asList(0, 1, 2);
assert Collections.max(ints) == 2;

List<String> strs = Arrays.asList("zero", "one", "two");
assert Collections.max(strs).equals("zero");
```

But T cannot be Number (since Number does not implement Comparable):

[org/jcbook/chapter03/B_maximum_of_a_collection/Program_4](#)

```
List<Number> nums = Arrays.asList(0, 1, 2, 3.14);
assert Collections.max(nums) == 3.14; // compile-time error
```

As expected, here the call to max is illegal.

Declarations for methods should be as general as possible to maximize utility. If you can replace a type parameter with a wildcard, then you should do so. We can improve the declaration of max by replacing:

```
<T extends Comparable<T>> T max(Collection<T> coll)
```

with:

```
<T extends Comparable<? super T>> T max(Collection<? extends T> coll)
```

Following the Get and Put Principle, we use extends with Collection because we *get* values of type T from the collection, and we use super with Comparable because we *put* values of type T into the compareTo method. In the next section, we'll see an example that would not type-check if the super clause added here were omitted.

If you look at the signature of this method in the Java library, you will see something that looks even worse than the preceding code:

```
<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

The emphasized bound was required for backward compatibility with pre-generic code, as we will explain at the end of “[Multiple Bounds](#)” on page 53.

A Fruity Example

The Comparable<T> interface gives fine control over what can and cannot be compared. Say that we have a Fruit class with subclasses Apple and Orange. Depending on how we set things up, we may *prohibit* comparison of apples with oranges or we may *permit* such comparison.

[Example 3-1](#) prohibits comparison of apples with oranges. Here are the three classes it declares:

```
class Fruit {...}
class Apple extends Fruit implements Comparable<Apple> {...}
class Orange extends Fruit implements Comparable<Orange> {...}
```

Each fruit has a name and a size, and two fruits are equal if they have the same name and the same size. Since we have overridden equals, we have also overridden hashCode, to ensure that equal objects have the same hash code (see “[Hash tables](#)” in the section “[Implementations](#)” on page 138 for an explanation of the importance of this). Apples are compared by comparing their sizes, and so are oranges. Since Apple

implements `Comparable<Apple>`, it is clear that we can compare apples with apples, but not with oranges. The test code builds three lists: one of apples, one of oranges, and one containing mixed fruits. We may find the maximum of the first two lists, but attempting to find the maximum of the mixed list signals an error at compile time:

```
% javac Example31.java
Example31.java:43: error: no suitable method found for max(List<Fruit>)
Collections.max(mixed); ❶ // compile-time error
^
method Collections.<T#1>max(Collection<? extends T#1>) is not applicable ❷
  (inference variable T#1 has incompatible bounds
    upper bounds: Object, Comparable<? super T#1>
    lower bounds: Fruit)
method Collections.<T#2>max(Collection<? extends T#2>, \
  Comparator<? super T#2>) is not applicable ❸
  (cannot infer type-variable(s) T#2
    (actual and formal argument lists differ in length))
where T#1,T#2 are type-variables:
T#1 extends Object,Comparable<? super T#1> declared in method \
<T#1>max(Collection<? extends T#1>)
T#2 extends Object declared in method \
<T#2>max(Collection<? extends T#2>,Comparator<? super T#2>)
1 error
```

This message may look intimidating, but it's worth working through it. The compiler is explaining why, following the declaration of `mixed` as a `List<Fruit>`, it failed to compile the last line ❶ of [Example 3-1](#). It made two attempts. First, it attempted to match the call against the `Collections` method:

```
<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

but failed, as reported in ❷, because no possible substitution for `T` is compatible with both `Fruit` (the type parameter of `mixed`) and `Comparable`. It then reports in ❸ that it failed to match the call against an overload of `Collections::max` with two parameters.

[Example 3-2](#) permits comparison of apples with oranges. Compare these three class declarations with those given previously (all differences between Examples [3-1](#) and [3-2](#) are emphasized, here and in the examples):

```
class Fruit implements Comparable<Fruit> {...}
class Apple extends Fruit {...}
class Orange extends Fruit {...}
```

As before, each fruit has a name and a size, and two fruits are equal if they have the same name and the same size. But now, since `Fruit` implements `Comparable<Fruit>`, any two fruits may be compared by comparing their sizes. So the test code can find the maximum of all three lists, including the one that mixes apples with oranges.

Recall that at the end of the previous section, we extended the type signature of `max` to use `super`:

```
<T extends Comparable<? super T>> T max(Collection<? extends T> coll)
```

Example 3-2 shows why this wildcard is needed. If we want to compare two oranges, we take `T` in the preceding code to be `Orange`:

```
Orange extends Comparable<? super Orange>
```

And this is true because both of the following hold:

```
Orange extends Comparable<Fruit> and Fruit super Orange
```

Without the `super` wildcard, finding the maximum of a `List<Orange>` would be illegal, even though finding the maximum of a `List<Fruit>` is permitted.

Also note that the natural ordering used here is not consistent with `equals` (see “[Comparable](#)” on page 35). Two fruits with different names but the same size compare as the same, but they are not equal.

Example 3-1. Prohibiting comparison of apples with oranges

org/jgcbook/chapter03/C_a_fruity_example/example_3_1/Example31

```
abstract class Fruit {  
    protected String name;  
    protected int size;  
    protected Fruit(String name, int size) {  
        this.name = Objects.requireNonNull(name);  
        this.size = size;  
    }  
    public boolean equals(Object o) {  
        if (o == null) return false;  
        if (o instanceof Fruit that) {  
            return this.name.equals(that.name) && this.size == that.size;  
        } else return false;  
    }  
    public int hashCode() {  
        return Objects.hash(name, size);  
    }  
    protected int compareTo(Fruit that) {  
        return Integer.compare(this.size, that.size);  
    }  
}  
class Apple extends Fruit implements Comparable<Apple> {  
    public Apple(int size) { super("Apple", size); }  
    public int compareTo(Apple a) { return super.compareTo(a); }  
}  
class Orange extends Fruit implements Comparable<Orange> {  
    public Orange(int size) { super("Orange", size); }  
    public int compareTo(Orange o) { return super.compareTo(o); }  
}  
class Test {  
    public static void main(String[] args) {
```

```

Apple a1 = new Apple(1); Apple a2 = new Apple(2);
Orange o3 = new Orange(3); Orange o4 = new Orange(4);

List<Apple> apples = Arrays.asList(a1, a2);
assert Collections.max(apples).equals(a2);

List<Orange> oranges = Arrays.asList(o3, o4);
assert Collections.max(oranges).equals(o4);

List<Fruit> mixed = List.of(a1, o3);
Collections.max(mixed);           // compile-time error
}
}

```

Example 3-2. Permitting comparison of apples with oranges

org/jgcbook/chapter03/C_a_fruity_example/example_3_2/Example32

```

abstract class Fruit implements Comparable<Fruit> {
    protected String name;
    protected int size;
    protected Fruit(String name, int size) {
        this.name = Objects.requireNonNull(name);
        this.size = size;
    }
    public boolean equals(Object o) {
        if (o == null) return false;
        if (o instanceof Fruit that) {
            return this.name.equals(that.name) && this.size == that.size;
        } else return false;
    }
    public int hashCode() {
        return Objects.hash(name, size);
    }
    public int compareTo(Fruit that) {
        return Integer.compare(this.size, that.size);
    }
}
class Apple extends Fruit {
    public Apple(int size) { super("Apple", size); }
}
class Orange extends Fruit {
    public Orange(int size) { super("Orange", size); }
}
class Test {
    public static void main(String[] args) {

        Apple a1 = new Apple(1); Apple a2 = new Apple(2);
        Orange o3 = new Orange(3); Orange o4 = new Orange(4);

        List<Apple> apples = Arrays.asList(a1, a2);
        assert Collections.max(apples).equals(a2);

        List<Orange> oranges = Arrays.asList(o3, o4);
        assert Collections.max(oranges).equals(o4);
    }
}

```

```
        List<Fruit> mixed = List.of(a1, o3);
        assert Collections.max(mixed).equals(o3); // ok
    }
}
```

Comparator

Sometimes we want to compare objects that do not implement the `Comparable` interface, or to compare objects using a different ordering from the one specified by that interface. The ordering provided by the `Comparable` interface is called the *natural ordering*, so the `Comparator` interface provides, so to speak, an unnatural ordering.

We specify additional orderings using the `Comparator` interface, which declares a single abstract method:²

```
interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

The `compare` method returns a value that is negative, zero, or positive depending upon whether the first object is less than, equal to, or greater than the second object—just as with `compareTo`.

Here is a comparator that considers the shorter of two strings to be smaller. Only if two strings have the same length are they compared using the natural (alphabetic) ordering:

org/jcbook/chapter03/D_comparator/Program_1

```
Comparator<String> sizeOrder = new Comparator<String>() {
    public int compare(String s1, String s2) {
        if (s1.length() < s2.length())
            return -1;
        if (s1.length() > s2.length())
            return 1;
        return s1.compareTo(s2);
    }
};
```

And here is an example of its use:

org/jcbook/chapter03/D_comparator/Program_1

```
assert "two".compareTo("three") > 0;
assert sizeOrder.compare("two", "three") < 0;
```

² If you check the source code for `Comparator`, you will notice that it also declares `equals` as an abstract method. `Comparator` still counts as a “single abstract method” (or *functional*) interface, however, as any concrete instance will either inherit the `Object` implementation or override it. The `equals` method is the one familiar from the class `Object`; it is included in the interface to remind implementors that equal comparators must have `compare` methods that impose the same ordering.

In the natural alphabetic ordering, "two" is greater than "three", whereas in the size ordering, it is smaller.

The Java libraries always provide a choice between `Comparable` and `Comparator`. For every generic method with a type variable bounded by `Comparable`, there is another generic method with an additional argument of type `Comparator`. For instance, corresponding to:

```
public static <T extends Comparable<? super T>
    T max(Collection<? extends T> coll)
```

we also have:

```
public static <T>
    T max(Collection<? extends T> coll, Comparator<? super T> comp)
```

There are similar methods to find the minimum. For example, here is how to find the maximum and minimum values of a list using the natural ordering and using the size ordering:

[org/jcbook/chapter03/D_comparator/Program_1](#)

```
Collection<String> strings = Arrays.asList("from", "aaa", "to", "zzz");
assert Collections.max(strings).equals("zzz");
assert Collections.min(strings).equals("aaa");
assert Collections.max(strings, sizeOrder).equals("from");
assert Collections.min(strings, sizeOrder).equals("to");
```

The string "from" is the maximum using the size ordering because it is longest, and "to" is the minimum because it is shortest.

Here, slightly simplified, is the code in the class `Collection` for the version of `max` that takes a `Comparator`:

[org/jcbook/chapter03/D_comparator/Program_2](#)

```
public static <T extends Comparable<T>>
    T max(Collection<T> coll, Comparator<? super T> comp) {
    Iterator<? extends T> i = coll.iterator();
    T candidate = i.next();
    while (i.hasNext()) {
        T next = i.next();
        if (comp.compare(next, candidate) > 0)
            candidate = next;
    }
    return candidate;
}
```

Compared to the previous version (at the start of ["The Maximum of a Collection" on page 39](#)), the only change is that where we wrote `next.compareTo(candidate)` previously, we now write `comp.compare(next, candidate)`.

It is easy to define a comparator that provides the natural ordering. This is a slightly simplified version of the code for the static method `naturalOrder` on the `Comparator` interface:

org/jcbook/chapter03/D_comparator/Program_3

```
public static <T extends Comparable<? super T>> Comparator<T> naturalOrder() {  
    return (o1, o2) -> o1.compareTo(o2);  
}
```

Using this, we can define the version of `max` that uses the natural ordering in terms of the version that uses a given comparator. This is the overload of `max` that was used in the stream version of the `Collections` method in “Comparable” on page 35:

org/jcbook/chapter03/D_comparator/Program_4

```
public static <T extends Comparable<? super T>> T max(Collection<? extends T> coll) {  
    return Collections.max(coll, Comparator.naturalOrder());  
}
```

Comparator Methods

In everyday practice, you will probably create `Comparator` instances using one of the static or default methods that the interface exposes, or a combination of them. Let’s explore how they work on a simple record type:

org/jcbook/chapter03/D_comparator/Program_6

```
record Person(String name, int age) {}
```

Probably the most commonly used methods are `comparing` and its variants, which accept a function that extracts a key to use in comparing two objects. For example, if we want to sort a list of `Person` objects by name, we can use this `Comparator`:

org/jcbook/chapter03/D_comparator/Program_6

```
Comparator<Person> compareByName = Comparator.comparing(Person::name);
```

The method `comparing` takes a function that extracts a sort key from the values to be sorted. That function is typically a getter, an implementation of the functional interface `java.util.function.Function`.³ It returns a comparator that, when it is applied to two objects of the type to be sorted, will return a value corresponding to the natural ordering on their respective sort keys. For example, if we define a list of `Person` objects:

³ See Naftalin (n.d. or 2014).

org/jcbook/chapter03/D_comparator/Program_6

```
Person a32 = new Person("Alice", 32);
Person b23 = new Person("Bob", 23);
List<Person> l = new ArrayList<>(List.of(a32, b23));
```

we can write:

org/jcbook/chapter03/D_comparator/Program_6

```
l.sort(compareByName);
assert l.equals(List.of(a32, b23));
```

Even somewhat simplified, the declaration of `Comparator.comparing` in the JDK looks very difficult:

org/jcbook/chapter03/D_comparator/Program_5

```
public static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> keyExtractor) {
    return (c1, c2) ->
        keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

In fact, we have met most of the difficulties already. `T` is the type of the class to be compared, and `U` is the type of the sort key. As we saw at the end of “[The Maximum of a Collection](#)” on page 39, the most generally useful bound on the type of a comparable is `U extends Comparable<? super U>`: in other words, if any supertype of `U` defines `compareTo`, `U` will inherit that definition. The type signature of the `Function` interface can be explained by the Substitution Principle: a function that can be applied to some supertype of `T` can certainly be applied to a `T` value; conversely, if it returns a subtype of `U`, that will be certainly be usable where a `U` is required.

The intimidating appearance of this declaration contrasts with the ease of using the method that it declares. This contrast is typical of Java generics: the API designer needs to think very carefully about how to make it as usable and general as possible. If the design is successful, client code developers will be highly appreciative—that is, if they stop to think about it, since the best-designed APIs are distinguished, paradoxically, by their unobtrusiveness.

The `Function` parameter of the method `comparing` can only be applied to an object of a reference type. So, for primitive types, we need specialized versions of `comparing`. For example:

org/jcbook/chapter03/D_comparator/Program_6

```
Comparator<Person> compareByAge = Comparator.comparingInt(Person::age);
l.sort(compareByAge);
assert l.equals(List.of(b23, a32));
```

In “Comparing Integral Values” on page 38, we saw two solutions to the problem of comparing two `Event` objects, one clumsy and one defective. We can use `comparingInt` to define a version of `compareTo` that is more readable and less error-prone than the first version, but without the overflow problem of the second:

[org/jgcbook/chapter03/D_comparator/Event](#)

```
record Event(String name, int millisecs) implements Comparable<Event> {
    private static final Comparator<Event> eventComparator =
        Comparator.comparingInt(Event::millisecs);
    public int compareTo(Event other) {
        return eventComparator.compare(this, other);
    }
}
```

The ordering of a comparator can be conveniently reversed using the instance method `reversed`:

[org/jgcbook/chapter03/D_comparator/Program_6](#)

```
l.sort(compareByAge.reversed());
assert l.equals(List.of(a32, b23));
```

Sorting often requires multiple levels. For example, when the result of sorting on one key produces groups of results, with the results in each group having the same value of the sort key, each group may need to be sorted by a secondary key. `Comparator` supports this with the instance method `thenComparing`, which creates a comparator that compares its arguments first using the receiver (this comparator) and then, if they are equal, using the comparator supplied as the argument. To show this in action, we’ll add a new `Person` object with the same name as one preexisting object and the same age as the other:

[org/jgcbook/chapter03/D_comparator/Program_6](#)

```
Person a23 = new Person("Alice", 23);
l.add(a23);
l.sort(compareByName.thenComparing(compareByAge));
assert l.equals(List.of(a23, a32, b23));
```

The `Comparator` API is designed so that these methods are composable. For example, suppose that we want to sort a list of `Person` objects in reverse order of age and then, in the case of tied results, by reverse order of name. That is easy to express:

[org/jgcbook/chapter03/D_comparator/Program_6](#)

```
l.sort(compareByAge.reversed().thenComparing(compareByName.reversed()));
assert l.equals(List.of(a32, b23, a23));
```

Unlike `Comparable::compareTo`, `Comparator::compare` can optionally permit comparison of `null` arguments. The static methods `nullsFirst` and `nullsLast` take a null-hostile comparator and return a null-friendly one. This new comparator treats non-null values in the same way as the supplied comparator, but handles `null` values

gracefully instead of throwing `NullPointerException`; it considers two `null` values to be equal, and a single `null` to be less (for `nullsFirst`) or greater (for `nullsLast`) than any non-`null` value:

org/jcbook/chapter03/D_comparator/Program_6

```
l.add(null);
l.sort(Comparator.nullsFirst(compareByAge));
assert l.equals(Arrays.asList(null, b23, a23, a32));
```

As a final example of comparators, here is a method that takes a comparator on elements and returns a comparator on lists of elements:

org/jcbook/chapter03/D_comparator/Program_7

```
public static <E>
    Comparator<List<E>> listComparator(final Comparator<? super E> comp) {
    return new Comparator<List<E>>() {
        public int compare(List<E> list1, List<E> list2) {
            int n1 = list1.size();
            int n2 = list2.size();
            for (int i = 0; i < Math.min(n1, n2); i++) {
                int k = comp.compare(list1.get(i), list2.get(i));
                if (k != 0) return k;
            }
            return Integer.compare(n1, n2);
        }
    };
}
```

The loop compares corresponding elements of the two lists and terminates when corresponding elements are found that are not equal (in which case, the list with the smaller element is considered smaller) or when the end of either list is reached (in which case, the shorter list is considered smaller). This is the usual ordering for lists; if we convert a string to a list of characters, it gives the usual (alphabetic) ordering on strings.

Enumerated Types

Java 5 included support for enumerated types, or *enums*—types whose values consist of a fixed set of constants. Here are two simple examples:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
```

Each enumerated type declaration can be expanded into a corresponding class in a stylized way. The corresponding class is designed so that it has exactly one instance for each of the enumerated constants, bound to a suitable static final variable. For example, the first of these enum declarations expands into a class called `Season`. Exactly four instances of this class exist, bound to four static final variables with

the names WINTER, SPRING, SUMMER, and FALL. There is no way to create any further instances of Season.

Foremost amongst the many useful features of Java enums is their type safety: for example, you can't assign a Weekday value to a variable of type Season, or compare two values of these different types. This type safety is enforced by making each enum type a subclass of `java.lang.Enum`, declared as in [Example 3-3](#). [Example 3-4](#) shows the declaration of one such subclass, `Season`.⁴

The class `Enum` provides properties that every enumerated type needs: its name and its ordinal—that is, its position in the enumeration sequence. Also, by implementing `Comparable`, it ensures that its derived classes will also be `Comparable`. But the requirement for type safety means that it is not enough that, for example, `Season` implements `Comparable`; on its own, that would still permit comparison of a `Season` with a `Weekday`. To prevent that, `Season` must implement `Comparable<Season>`.

This takes us some distance toward understanding the declaration of `Enum`:

```
class Enum<E extends Enum<E>> implements Comparable<E>
```

You may find this frightening at first sight—your authors certainly did! But don't panic. Matching things up shows how a class derived from `Enum` will have the property that we need. From what we already know, we have:

```
class Season extends ... implements Comparable<Season>
```

where `...` is some parameterization of `Enum`. If we take that parameterization to be `Enum<Season>`, we get:

```
class Season extends Enum<Season>
class Enum<Season> implements Comparable<Season>
```

and substituting the declaration of `Season` in the first line for the first occurrence of `Season` in the second line gives:

```
class Enum<Season extends Enum<Season>> implements Comparable<Season>
```

Of course, `Enum` needs to work on any enumerated type, not only `Season`, so its declaration in the class library is:

```
class Enum<E extends Enum<E>> implements Comparable<E>
```

⁴ The code for [Example 3-3](#) does not correspond on all points with the source in the Java library, and most of the code corresponding to [Example 3-4](#) is actually synthesized by the Java compiler—as it stands, it is not legal Java—but these examples are functionally equivalent to the code that is actually executed.

Example 3-3. Base class for enumerated types

[org/jcbook/chapter03/E_enumerated_types/Enum](#)

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E> {  
    private final String name;  
    private final int ordinal;  
    protected Enum(String name, int ordinal) {  
        this.name = name; this.ordinal = ordinal;  
    }  
    public final String name() { return name; }  
    public final int ordinal() { return ordinal; }  
    public String toString() { return name; }  
    public final int compareTo(E o) {  
        return ordinal - ((Enum<?>)o).ordinal;  
    }  
}
```

Example 3-4. Class corresponding to an enumerated type

[org/jcbook/chapter03/E_enumerated_types/Season](#)

```
/*  
 * Corresponds to enum Season { WINTER, SPRING, SUMMER, FALL }  
 */  
final class Season extends Enum<Season> {  
    private Season(String name, int ordinal) { super(name, ordinal); }  
    public static final Season WINTER = new Season("WINTER", 0);  
    public static final Season SPRING = new Season("SPRING", 1);  
    public static final Season SUMMER = new Season("SUMMER", 2);  
    public static final Season FALL = new Season("FALL", 3);  
    private static final Season[] VALUES = { WINTER, SPRING, SUMMER, FALL };  
    public static Season[] values() { return VALUES.clone(); }  
    public static Season valueOf(String name) {  
        for (Season e : VALUES) if (e.name().equals(name)) return e;  
        throw new IllegalArgumentException();  
    }  
}
```

Recursively bounded types like `T extends Comparable<T>` and `E extends Enum<E>` occur when we want to constrain a parameter to range only over the subtypes of a particular type. For example, in any subtype `T` of the base `Enum` class, the `compareTo` method can only be applied to an argument of type `T`, not to any other subtype of `Enum`; the definition provides a way of naming the type itself in the body of the class. (The term “self-type” is sometimes used for this in discussions of recursively bounded types.)

The rest of the definitions are straightforward. The base class `Enum` defines two fields, a string `name` and an integer `ordinal`, that are possessed by every instance of an enumerated type; the fields are final because once they are initialized, their value never changes. The constructor for the class is protected, to ensure that it is used only

within subclasses of this class. Each enumeration class makes the constructor private, to ensure that it is used only to create the enumerated constants. For instance, the `Season` class has a private constructor that is invoked exactly four times in order to initialize the final variables `WINTER`, `SPRING`, `SUMMER`, and `FALL`.

The base class defines accessor methods for the `name` and `ordinal` fields. The `toString` method returns the name, and the `compareTo` method just returns the difference of the ordinals for the two enumerated values. (Unlike the definition of `Event` in “[Comparable](#)” on page 35, this is safe; since the ordinals are always positive, there is no possibility of overflow.) Hence, constants have the same ordering as their ordinals—for example, `WINTER` precedes `SUMMER`.

Lastly, there are two static methods in every class that corresponds to an enumerated type. The `values` method returns a (shallow) clone of the internal array that holds references to all the constants of the type. Cloning is vital to ensure that the client cannot alter the internal array. Note that you don’t need a cast when calling the `clone` method, because cloning for arrays takes advantage of covariant return types (see “[Covariant Overriding](#)” on page 57). The `valueOf` method takes a string and returns the corresponding constant, found by lookup in a map initialized from the internal array. It returns an `IllegalArgumentException` if the string does not name a value of the enumeration.

Multiple Bounds

We have seen many examples where a type variable or wildcard is bounded by a single class or interface. In rare situations, it may be desirable to have multiple bounds, and we show how to do so here.

To demonstrate, we use three interfaces from the Java library. The `Readable` interface has a `read` method to read into a buffer from a character source, the `Appendable` interface has an `append` method to copy from a buffer into a target capable of receiving characters, and the `Closeable` interface has a `close` method to close a source or target. Possible sources and targets include character files, buffers, streams, and so on.

For maximum flexibility, we might want to write a `copy` method that accepts both a `Supplier` of any source that implements both `Readable` and `Closeable` and a `Supplier` of any target that implements both `Appendable` and `Closeable`. The method will copy the contents of the source to the target. Unfortunately, character readers and writers typically cannot be opened without potentially throwing `IOException`, so we need a specialized supplier interface:

[org/jcbook/chapter03/F_multiple_bounds/IoExceptionSupplier](#)

```
public interface IOExceptionSupplier<S> {  
    S get() throws IOException;  
}
```

The method `copy` can now be declared:

[org/jcbook/chapter03/F_multiple_bounds/Program_1](#)

```
public static <S extends Readable & Closeable, T extends Appendable & Closeable  
        void copy(IOExceptionSupplier<S> src, IOExceptionSupplier<T> tgt, int size)  
            throws IOException {  
    try (S s = src.get(); T t = tgt.get()) { ❶  
        CharBuffer buf = CharBuffer.allocate(size);  
        int i = s.read(buf);  
        while (i >= 0) {  
            buf.flip(); // prepare buffer for writing  
            t.append(buf);  
            buf.clear(); // prepare buffer for reading  
            i = s.read(buf);  
        }  
    }  
}
```

The declaration of this method specifies that `S` ranges over any type that implements both `Readable` and `Closeable`, and that `T` ranges over any type that implements `Appendable` and `Closeable`. When multiple bounds on a type variable appear, they are separated by ampersands. The types defined by multiple bounds are called *intersection types*.

The `try-with-resources` statement ❶ evaluates the supplied lambdas, which open and return a character source and a character target. The body of the statement then repeatedly reads from the source into a buffer and appends from the buffer into a target. When the source is empty, the `try` block is exited, closing both the source and the target. As an example, this code may be called with a `FileReader` and a `FileWriter` as the source and target:

[org/jcbook/chapter03/F_multiple_bounds/Program_1](#)

```
public static void main(String[] args) throws IOException {  
    int size = 32;  
    Files.writeString(Path.of("file.in"), "hello world");  
    copy(() -> new FileReader("file.in"),  
         () -> new FileWriter("file.out"), size);  
    assert Files.readString(Path.of("file.out")).equals("hello world");  
}
```

Other possible sources include `FilterReader`, `PipedReader`, and `StringReader`, and other possible targets include `FilterWriter`, `PipedWriter`, and `PrintStream`. But you could not use `StringBuffer` as a target, since it implements `Appendable` but not `Closeable`.

If you are familiar with the `java.io` library, you may have spotted that all classes that implement both `Readable` and `Closeable` are subclasses of `Reader`, and almost all classes that implement `Appendable` and `Closeable` are subclasses of `Writer`. So you might wonder why we don't simplify the method signature like this:

```
public static void copy(Reader src, Writer trg, int size)
```

This will indeed admit most of the same classes, but not all of them. For instance, `PrintStream` implements `Appendable` and `Closeable` but is not a subclass of `Writer`. Furthermore, you can't rule out the possibility that some programmer using your code might have his or her own custom class that, say, implements `Readable` and `Closeable` but is not a subclass of `Reader`.

When multiple bounds appear, the first bound is used for erasure. We saw this in “[The Maximum of a Collection](#)” on page 39:

```
public static <T extends Object & Comparable<? super T>>  
    T max(Collection<? extends T> coll)
```

Without the emphasized text, the erased type signature for `max` would have a return type of `Comparable`, which would be incompatible with the pre-generic return type of `Object`. Maintaining compatibility with legacy libraries is further discussed in the [Appendix](#).

Bridge Methods

As we explained in “[Generic Types](#)” on page 3, generics are implemented by erasure: the compiled representation of code written with generics is almost exactly the same as that written without. However, in the case of a parameterized supertype—for example, an interface such as `Comparable<T>`—erasure may require additional methods to be inserted by the compiler. These additional methods are called *bridge methods*, often shortened to *bridges*.

[Example 3-5](#) shows the `Comparable` interface and a simple `Point` record implementing that interface. The natural order on `Points` is defined by comparing their distances from the origin (it's actually more convenient to compare the squares of their distances from the origin, which amounts to the same thing). The declaration of `compareTo` in `Point` overrides that in `Comparable`, because the signatures match.

Example 3-5. Generic code implementing a parameterized interface

org/jgcbook/chapter03/G_bridges/example_3_5/Example35

```
interface Comparable<T> {  
    public int compareTo(T other);  
}  
record Point(double x, double y) implements Comparable<Point> {  
    public int compareTo(Point p) {
```

```

        return Double.compare(this.x * this.x + this.y * this.y,
                             p.x * p.x + p.y * p.y);
    }
}

```

Erasure changes the situation, however, as shown in [Example 3-6](#).

Example 3-6. Parameterized interface and implementation after erasure

[org/jgcbook/chapter03/G_bridges/example_3_6/Example36](#)

```

interface Comparable {
    public int compareTo(Object other);
}

record Point(double x, double y) implements Comparable { // fails compilation
    public int compareTo(Point p) {
        return Double.compare(this.x * this.x + this.y * this.y,
                             p.x * p.x + p.y * p.y);
    }
}

```

The signature of `compareTo` in the interface has changed, and to restore overriding the compiler must add an extra method to the implementation:

```

public int compareTo(Object other) {
    return compareTo((Point)other);
}

```

You can confirm the existence of this bridge using reflection:

[org/jgcbook/chapter03/G_bridges/example_3_5/Example35](#)

```

Map<Class<?>, Boolean> typeToBridge = (Arrays.stream(Point.class.getMethods())
    .filter(m -> m.getName().equals("compareTo"))
    .collect(Collectors.toMap(m -> m.getParameterTypes()[0], Method::isBridge)));
assert typeToBridge.size() == 2;
assert ! typeToBridge.get(Point.class); // compareTo(Point) is not a bridge method
assert typeToBridge.get(Object.class); // compareTo(Object) is a bridge method

```

And the entire declaration of the method can be seen by decompiling the class file, for example using `javap -verbose` or with the open source decompiler [FernFlower](#), which produces this output:

```

.....
// $FF: synthetic method
// $FF: bridge method
public int compareTo(Object var1) {
    return this.compareTo((Point)var1);
}
.....

```

Bridge methods are required whenever a class or interface implements or extends a parameterized supertype by instantiating its type parameter.⁵

Bridge methods played an important role when converting legacy code to use generics, as described in the article “[Maintain Binary Compatibility](#)”.

Covariant Overriding

At the same time that generics were introduced, Java started supporting covariant method overriding. This feature is not directly related to generics, but we discuss it here because it is worth knowing about, and because it is implemented using a bridging technique like that described in the previous section.

In early versions of Java, one method could override another only if the signatures and the return types matched exactly. With covariant overriding, the signatures must still match (after erasure), but the requirement to match the return types is relaxed so that now the return type of the overriding method need only be a subtype of the return type of the overridden method.

The `clone` method of class `Object` illustrates the advantages of covariant overriding:

```
class Object {  
    ...  
    protected Object clone() { ... }  
}
```

Without covariant overriding, any class that overrides `clone` has to give the overriding method exactly the same return type, namely `Object`. For example, here is a `Point` record that does this:

[org/jgcbook/chapter03/H_covariant_overriding/Point_1](#)

```
record Point_1(double x, double y) {  
    public Object clone() { return new Point_1(x, y); }  
}
```

Here, even though `clone` always returns a `Point_1`, without covariance the old rules required it to have the return type `Object`. This was annoying, since every invocation of `clone` had to cast its result:

[org/jgcbook/chapter03/H_covariant_overriding/Point_1](#)

```
Point_1 p = new Point_1(1, 2);  
Point_1 q = (Point_1)p.clone();
```

With covariant overriding, it is possible to give the `clone` method a return type that is more to the point, as it were:

⁵ Strictly speaking, this is true only if erasure changes the signature of any of the supertype's methods.

[org/jgcbook/chapter03/H_covariant_overriding/Point_2](#)

```
record Point_2(double x, double y) {  
    public Point_2 clone() { return new Point_2(x, y); }  
}
```

Now we may clone without a cast:

[org/jgcbook/chapter03/H_covariant_overriding/Point_2](#)

```
Point_2 p = new Point_2(1, 2);  
Point_2 q = p.clone();
```

Covariant overriding is implemented using the bridging technique described in the previous section. As before, you can see the bridge using decompilation or by applying reflection. Here is code that finds all methods with the name `clone` in the class `Point_2`, then maps the return type of each overload to the result of calling the `Method` method `isBridge`:

[org/jgcbook/chapter03/H_covariant_overriding/Point_2](#)

```
Map<Class<?>, Boolean> returnToBridge = (Arrays.stream(Point_2.class.getMethods())  
    .filter(m -> m.getName().equals("clone"))  
    .collect(Collectors.toMap(Method::getReturnType, Method::isBridge)));  
assert returnToBridge.size() == 2;  
assert returnToBridge.get(Object.class); // Object clone(Point_2) is a bridge method  
assert !returnToBridge.get(Point_2.class); // Point_2 clone(Point_2) is not
```

Here, the bridging technique exploits the fact that in a class file two methods of the same class may have the same argument signature, even though this is not permitted in Java source. The bridge method simply calls the first method.

Conclusion

In this chapter, we saw how the interfaces `Comparable<T>` and `Comparator<T>` work with their generic parameters, why `Comparable<T>` needs to have a recursive bound, and how to understand the more complex recursive bound for enumerated types.

From now on, our viewpoint will change from that of a client programmer who uses generic types, to that of a library writer who creates them. We'll begin in the next chapter by examining how to declare a generic class.

CHAPTER 4

Declarations

This chapter discusses how to declare a generic class. It describes constructors, static members, and nested classes, and it fills in some details of how erasure works.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter04](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter04)

Constructors

In a generic class, type parameters appear in the header that declares the class, but not in the constructor:

org/jgcbook/chapter04/A_constructors/Pair

```
class Pair<T,U> {  
    private final T first;  
    private final U second;  
    public Pair(T first, U second) {  
        this.first = first;  
        this.second = second;  
    }  
    public T getFirst() { return first; }  
    public U getSecond() { return second; }  
}
```

The type parameters `T` and `U` are declared at the beginning of the class, not in the constructor, but actual type arguments are passed to the constructor whenever it is invoked:

[org/jcbook/chapter04/A_constructors/Pair](#)

```
Pair<String, Integer> pair1 = new Pair<String, Integer>("one", 2);
assert pair1.getFirst().equals("one") && pair1.getSecond() == 2;
```

A common mistake is to forget the type parameters when invoking the constructor:

[org/jcbook/chapter04/A_constructors/Pair](#)

```
Pair pair2 = new Pair<String, Integer>("one", 2);
```

This mistake produces a warning when compiled with the option `-Xlint`, which enables detailed warnings:

```
% javac -Xlint Pair.java
Pair.java:18: warning: [rawtypes] found raw type: Pair
    Pair pair2 = new Pair<String, Integer>("one", 2);
                           ^
missing type arguments for generic class Pair<T,U>
where T,U are type-variables:
  T extends Object declared in class Pair
  U extends Object declared in class Pair
1 warning
```

But the compiler does not flag it as an error. It is taken to be legal because `Pair` is treated as a *raw type*—a type that contains no parametric type information but which can be converted to the corresponding parameterized type—generating only an unchecked warning. The effect is to turn off type checking for all subsequent uses of `pair2`: it will be possible to assign `Pair` values of any type to it without further warnings or errors. For further discussion, see the section in the Appendix “[Generic Library with Legacy Client](#)” for raw types and “[Unchecked Casts](#)” on page 75 for unchecked types.

Records can be parameterized in the same way as regular classes:

```
record Pair<T,U>(T first, U second) {}
```

Static Members

Compilation by erasure means that at run time parameterized types are replaced by the corresponding raw type: for example, the interfaces `List<Integer>`, `List<String>`, and `List<List<String>>` are all implemented by a single interface, namely `List`:

```
List<Integer> ints = Arrays.asList(1, 2, 3);
List<String> strings = Arrays.asList("one", "two");
assert ints.getClass() == strings.getClass();
```

We see here that the class object associated with a list of integers at run time is the same as the one associated with a list of strings.

One consequence is that static members of a generic class are shared across all instantiations of that class, including instantiations at different types. Static members of a class cannot refer to the type parameter of a generic class, and when accessing a static member the class name should not be parameterized.

For example, here is a class, `Cell<T>`, in which each cell has an integer identifier and a value of type `T`:

```
class Cell<T> {
    private final int id;
    private final T value;
    private final static AtomicInteger count = new AtomicInteger();
    private static int nextId() { return count.getAndIncrement(); }
    public Cell(T value) {
        this.value = value;
        id = nextId();
    }
    public T getValue() { return value; }
    public int getId() { return id; }
    public static int getCount() { return count.get(); }
}
```

A static field, `count`, is used to allocate a distinct identifier to each cell. It is declared as an `AtomicInteger` to ensure that unique identifiers are generated under concurrent access. The static `getCount` method returns the current count.

Here is some code that allocates a cell containing a string and a cell containing an integer, which are allocated the identifiers 0 and 1, respectively:

```
Cell<String> a = new Cell<String>("one");
Cell<Integer> b = new Cell<Integer>(2);
assert a.getId() == 0 && b.getId() == 1 && Cell.getCount() == 2;
```

Static members are shared across all instantiations of a class, so the same count is incremented when allocating either a string or an integer cell.

Because static members are independent of any type parameters, we are not permitted to follow the class name with type parameters when accessing a static member:

```
Cell.getCount();           // ok
Cell<Integer>.getCount(); // compile-time error
Cell<?>.getCount();      // compile-time error
```

The count is static, so it is a property of the class as a whole, not any particular instance.

For the same reason, you can't refer to a type parameter anywhere in the declaration of a static member. Here is a second version of `Cell`, which attempts to use a static variable to keep a list of all values stored in any cell:

[org/jgcbook/chapter04/B_static_members/Cell2](#)

```
class Cell2<T> {
    private final T value;
    private static List<T> values = new ArrayList<T>(); // illegal
    public Cell2(T value) {
        this.value=value;
        values.add(value);
    }
    public T getValue() { return value; }
    public static List<T> getValues() { return values; } // illegal
}
```

Since the class may be used with different type parameters at different places, it makes no sense to refer to `T` in the declaration of the static field `values` or the static method `getValues`, and these lines are reported as errors at compile time. If we want a list of all values kept in cells, then we need to use a list of objects, as in the following variant:

[org/jgcbook/chapter04/B_static_members/Cell3](#)

```
class Cell3<T> {
    private final T value;
    private static List<Object> values = new ArrayList<Object>(); // ok
    public Cell3(T value) {
        this.value=value;
        values.add(value);
    }
    public T getValue() { return value; }
    public static List<Object> getValues() { return values; } // ok
}
```

This code compiles and runs with no difficulty:

[org/jgcbook/chapter04/B_static_members/Cell3](#)

```
Cell3<String> a = new Cell3<String>("one");
Cell3<Integer> b = new Cell3<Integer>(2);
assert Cell3.getValues().equals(List.of("one", 2));
```

Nested Classes

Java permits nesting one class inside another. If the outer class has type parameters and the inner class is a member class—that is, not static—then type parameters of the outer class are visible within the inner class.

[Example 4-1](#) shows a class implementing collections as a singly linked list. This class extends `java.util.AbstractCollection`, so it only needs to define the methods `size`, `add`, and `iterator` (see [Chapter 17](#)). It contains an inner class, `Node`, for the list nodes, and an anonymous inner class implementing `Iterator<E>`.

Example 4-1. Type parameters are in scope for member classes

[org/jgcbook/chapter04/C_nested_classes/example_4_1/LinkedCollection](#)

```
class LinkedCollection<E> extends AbstractCollection<E> {
    private class Node {
        private final E element;
        private Node next = null;
        private Node(E elt) { element = elt; }
    }
    private Node first = new Node(null);
    private Node last = first;
    private int size = 0;
    public LinkedCollection() {}
    public LinkedCollection(Collection<? extends E> c) { addAll(c); }
    public int size() { return size; }
    public boolean add(E elt) {
        last.next = new Node(elt); last = last.next; size++;
        return true;
    }
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private Node current = first;
            public boolean hasNext() {
                return current.next != null;
            }
            public E next() {
                if (current.next != null) {
                    current = current.next;
                    return current.element;
                } else throw new NoSuchElementException();
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

The type parameter `E` is in scope in both these classes. This point is important: if you don't appreciate it, you may make the common error of thinking that you need to

propagate the scope of E into an inner class with a declaration like `class Node<E>`. Doing this will in fact declare a new type variable E, which will shadow the variable E in the outer class, with very confusing results. If your inner class really needs a different type parameter from the outer one, make it static, as described next.

Example 4-2 shows a similar implementation, but this time the inner `Node` class is static, so the type parameter E is *not* in scope for this class. Instead, the inner class is declared with its own type parameter, T. Where the previous version referred to `Node`, the new version refers to `Node<E>`. The anonymous iterator class in the preceding example has also been replaced by a static inner class, again with its own type parameter.

Example 4-2. Type parameters are not in scope for static inner classes

org/jgcbook/chapter04/C_nested_classes/example_4_2/LinkedCollection

```
class LinkedCollection<E> extends AbstractCollection<E> {
    private static class Node<T> {
        private final T element;
        private Node<T> next = null;
        private Node(T elt) { element = elt; }
    }
    private Node<E> first = new Node<E>(null);
    private Node<E> last = first;
    private int size = 0;
    public LinkedCollection() {}
    public LinkedCollection(Collection<? extends E> c) { addAll(c); }
    public int size() { return size; }
    public boolean add(E elt) {
        last.next = new Node<E>(elt); last = last.next; size++;
        return true;
    }
    private static class LinkedIterator<T> implements Iterator<T> {
        private Node<T> current;
        public LinkedIterator(Node<T> first) { current = first; }
        public boolean hasNext() {
            return current.next != null;
        }
        public T next() {
            if (current.next != null) {
                current = current.next;
                return current.element;
            } else throw new NoSuchElementException();
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
    public Iterator<E> iterator() {
        return new LinkedIterator<E>(first);
    }
}
```

If the node classes had been made public rather than private, you would refer to the node class in the first example as `LinkedCollection<E>.Node`, whereas you would refer to the node class in the second example as `LinkedCollection.Node<E>`.

Of the two alternatives described here, we recommend the second. Member classes are implemented by including a reference to the enclosing instance, since they may, in general, access components of that instance. Static inner classes are usually both simpler and more efficient.

How Erasure Works

To erase a type, the compiler does the following: if it is parameterized, drop all type parameters, and replace any type variable with the erasure of its bound, or with `Object` if it has no bound, or with the erasure of the leftmost bound if it has multiple bounds. Here are some examples:

- The erasure of `List<Integer>`, `List<String>`, and `List<List<String>>` is `List`.
- The erasure of `List<Integer>[]` is `List[]`.
- The erasure of `List` is itself (and the same for any raw type).
- The erasure of `int` is itself (and the same for any primitive type).
- The erasure of `Integer` is itself (and the same for any type without type parameters).
- The erasure of `T` in the definition of `toList` (see “[Generic Methods and Varargs](#)” on page 7) is `Object`, because `T` has no bound.
- The erasure of `T` in the definition of `max` (see “[The Maximum of a Collection](#)” on page 39) is `Comparable`, because `T` has bound `Comparable<? super T>`.
- The erasure of `T` in the final definition of `max` (see “[Multiple Bounds](#)” on page 53) is `Object`, because `T` has bound `Object & Comparable<T>` and we take the erasure of the leftmost bound.
- The erasures of `S` and `T` in the definition of `copy` (see “[Multiple Bounds](#)” on page 53) are `Readable` and `Appendable`, because `S` has bound `Readable & Closeable` and `T` has bound `Appendable & Closeable`.
- The erasure of `LinkedCollection<E>.Node` or `LinkedCollection.Node<E>` (see “[Nested Classes](#)” on page 63) is `LinkedCollection.Node`.

In Java, two methods of the same class cannot have the same signature—that is, the same name and parameter types. Since generics are implemented by erasure, it also follows that two distinct methods cannot have signatures with the same erasure. A class cannot overload two methods whose signatures have the same erasure, and a class cannot implement two interfaces that have the same erasure.

For example, here is a class with two convenience methods. One adds together every integer in a list of integers, and the other concatenates together every string in a list of strings:

```
org/jgcbook/chapter04/D_how_erasure_works/Overloaded
```

```
class Overloaded {
    // compile-time error, cannot implement two interfaces with same erasure
    public static int sum(List<Integer> ints) {
        int sum = 0;
        for (int i : ints) sum += i;
        return sum;
    }
    public static String sum(List<String> strings) {
        StringBuilder sum = new StringBuilder();
        for (String s : strings) sum.append(s);
        return sum.toString();
    }
}
```

Here are the erasures of the declarations of the two methods:

```
int sum(List)
String sum(List)
```

But it is the signatures alone, not the return types, that allow the Java compiler to distinguish different method overloads. In this case the erasures of the signatures of both methods are identical:

```
sum(List)
```

so a name clash is reported at compile time:

```
Overloaded.java:11: error: name clash: sum(List<String>) and sum(List<Integer>) \
have the same erasure
    public static String sum(List<String> strings) {
                           ^

```

Let's look at another example. Here is a bad version of the `Integer` class that tries to make it possible to compare an integer with either an integer or a long:

```
org/jgcbook/chapter04/D_how_erasure_works/Integer
```

```
class Integer implements Comparable<Integer>, Comparable<Long> {
    // compile-time error, cannot implement two interfaces with same erasure
    private int value;
    ...
    public int compareTo(Integer i) {
        return (value < i.value) ? -1 : (value == i.value) ? 0 : 1;
    }
}
```

```
    }
    public int compareTo(Long l) {
        return (value < l.intValue()) ? -1 : (value == l.intValue()) ? 0 : 1;
    }
    ...
}
```

If this were supported, it would, in general, require a complex and confusing definition of bridge methods (see “[Bridge Methods](#)” on page 55). The simplest and most understandable option by far is to ban this case.

Conclusion

In this chapter, we saw how to declare a generic class with constructors, static members, and nested classes, and we explored the mechanism of erasure in more detail than before.

In the next chapter, we’ll learn more about the consequences of erasure, in particular for the strained relationship between arrays and generic types.

Reifiable and Nonreifiable Types

By the time generics were introduced to Java in 2004, eight years after version 1.0 shipped, there were many millions of lines of code already in use in commercial applications. It is a tribute to the design of generics that the introduction was so successful, resulting in gradual generification of all major libraries and most client code within a few years, without major compatibility problems. This achievement is described in the [Appendix](#). The key to its success was erasure, without which Java as we know it today could not exist. Legacy code makes no distinction between `List<Integer>`, `List<String>`, and `List<List<String>>`, so erasing parameter types is essential to easing evolution and promoting compatibility between legacy code and new code. But everything comes at a cost, and in this chapter we must settle our debts.

The *Oxford English Dictionary* defines *reify* thus: “to convert mentally into a thing; to materialize.” A plainer word with the same meaning is *thingify*. In computing, *reification* has come to mean the explicit representation of a type—that is, run-time type information. In Java, a reified array retains information about its component type, whereas a reified generic type does not retain information about its type parameters.

Reification plays a critical role in certain aspects of Java, and its absence from generics, beneficial as that is for evolution, also necessarily leads to some rough edges. This chapter warns you of the limitations and describes some workarounds. It deals almost entirely with things you might wish you didn’t need to know—and, indeed, if you never use generic types in casts, instance tests, exceptions, or arrays, then you are unlikely to need the material covered here.

We begin with a precise definition of what it means for a type in Java to be reifiable. We then consider corner cases related to reification, including instance tests, casts, exceptions, and arrays. The fit between arrays and generics is the worst rough corner

in the language, so we identify two principles for avoiding the worst pitfalls: the Principle of Truth in Advertising and the Principle of Indecent Exposure.



The code examples for this chapter can be found at:

https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter05

Reifiable Types

In Java, the type of an array is reified *with* its component type, while a parameterized type is reified *without* its type parameters. For instance, an array of numbers will carry the reified type `Number[]`, while an `ArrayList<Number>` will carry the reified type `ArrayList`; the raw type, not the parameterized type, is reified. Of course, each element of the list will have a reified type attached to it—say, `Integer` or `Double`—but this is not the same as reifying the parameter type. If every element in the list were an integer, we would not be able to tell at run time whether we had an `ArrayList<Integer>`, `ArrayList<Number>`, or `ArrayList<Object>`; if the list were empty, we would not be able to tell what kind of empty list it was.

In Java, we say that a type is *reifiable* if the type is completely represented at run time—that is, if erasure does not remove any useful information. A type is reifiable if it is one of the following:

- A primitive type (such as `int`)
- A nonparameterized class or interface type (such as `Number`, `String`, or `Runnable`)
- A raw type (such as `List`, `ArrayList`, or `Map`)
- A parameterized type in which all type arguments are unbounded wildcards (such as `List<?>`, `ArrayList<?>`, or `Map<?, ?>`)
- An array whose component type is reifiable (such as `int[]`, `Number[]`, `List[]`, `List<?>[]`, or `int[][]`)

A type is *not* reifiable if it is one of the following:

- A type variable (such as `T`)
- A parameterized type with non-wildcard parameters (such as `List<Number>`, `ArrayList<String>`, or `Map<String, Integer>`)
- A parameterized type with a bounded wildcard parameter (such as `List<? extends Number>` or `Comparable<T super String>`)

So for instance, the type `List<? extends Object>` is *not* reifiable, even though it is equivalent to `List<?>`. Defining reifiable types in this way makes them easy to identify syntactically.

Instance Tests and Casts

Until Java 16, instance tests and casts depended entirely on examining types at run time and so could only be applied to reified types. For example, this is the model for an `equals` method for a user type `MyType`, as proposed by Bloch (2017, item 10):

```
public boolean equals(Object o) {  
    if (!(o instanceof MyType))  
        return false;  
    MyType mt = (MyType) o;  
    // now compare the field values of this object against mt  
}
```

This method takes an argument of type `Object`, checks whether the object is an instance of the reference type `MyType`, and, if so, casts it to that type and compares the values of the fields in the two objects. When the reference type is reifiable, all the information needed to check whether an object is an instance of it is available at run time. Used like this, `instanceof` is called the *type comparison operator*.

At Java 16, `instanceof` became an overloaded operator. It is now, depending on the context, either the type comparison operator or the *pattern match operator*. Here is the `equals` method, rewritten using the pattern match operator:

```
public boolean equals(Object o) {  
    return o instanceof MyType mt && ...  
    // now compare the field values of this object against mt  
}
```

Evaluating the pattern expression `o instanceof MyType mt` still returns the truth value of the type comparison expression `o instanceof MyType`. But in addition, if that value is `true`, it declares the variable `mt` with type `MyType` and binds it to the value of `(MyType)o`. Fortunately, the rules for types in such type pattern match expressions are so similar to those for type comparison expressions that we can restrict our discussion to the latter.

Up to Java 16, the second operand of the type comparison operator had to be reifiable. Java 16, however, lifted this requirement: now, some instance tests on non-reifiable types can be evaluated entirely at compile time. For example, the following code compiles (and therefore executes) without any problem:

org/jgcbook/chapter05/B_instance_tests_and_casts/Program_1

```
// compiles in Java 16 onward  
ArrayList<Integer> x = new ArrayList<>();  
assert x instanceof List<? extends Number>;
```

The `assert` statement is unnecessary here; the compiler alone can determine that, since `ArrayList` is a subtype of `List` and `Integer` is a subtype of `Number`, `ArrayList<Integer>` is a subtype of `List<? extends Number>`.

This feature was not introduced for such toy examples, however, but to be used in conjunction with run-time evaluation of `instanceof`. For example, the following method converts any collection to a list:

[org/jcbook/chapter05/B_instance_tests_and_casts/Program_2](#)

```
// compiles in Java 16 onward
public static <T> List<T> asList(Collection<T> cl) {
    return cl instanceof List<T> ? (List<T>)cl : cl.stream().toList();
}
```

Or, using a pattern-matching expression:

[org/jcbook/chapter05/B_instance_tests_and_casts/Program_3](#)

```
// compiles in Java 16 onward
public static <T> List<T> asList(Collection<T> cl) {
    return cl instanceof List<T> q ? q : cl.stream().toList();
}
```

Both the instance test and the cast are well typed, because the type parameter in the declaration of `cl` can be cast to—in this case, actually matches—that of `List<T>`. Then, at run time, if the reified type of `cl` is found to be a subtype of `List`, `cl` can be cast to `List<T>`. Otherwise, the collection is streamed into a new list.

Often, however, instance tests and casts involving nonreifiable types cannot be type checked in this way. When that is the case, attempting to compile an instance test results in an error, whereas compiling a cast results in a warning. For example, consider how one might define equality on lists, as in the class `AbstractList` in `java.util`. Here is a naïve—and incorrect—way to define this:

[org/jcbook/chapter05/B_instance_tests_and_casts/AbstractList_1](#)

```
abstract class AbstractList_1<E> extends AbstractCollection<E> implements List<E> {
    public boolean equals(Object o) {
        if (!(o instanceof List<E>)) { return false; }           // compile-time error
        ListIterator<E> it1 = listIterator();
        ListIterator<E> it2 = ((List<E>)o).listIterator(); // unchecked cast
        while (it1.hasNext() && it2.hasNext()) {
            E e1 = it1.next();
            E e2 = it2.next();
            if (!Objects.equals(e1, e2))
                return false;
        }
        return !(it1.hasNext() || it2.hasNext());
    }
    ...
}
```

Again, the `equals` method takes an argument of type `Object`, but this time it checks whether the object is an instance of a nonreifiable type, namely `List<E>`. If so, it casts it to `List<E>` and compares corresponding elements of the two lists. The instance test code does not compile, for reasons that differ between pre-16 versions and later ones. Up to Java 16, the compiler reported an instance test on *any* nonreifiable type as an error:

```
% javac AbstractList_1.java
AbstractList_1.java:8: illegal generic type for instanceof
    if (!(o instanceof List<E>)) { return false; }      // compile-time error
               ^
Note: AbstractList_1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
1 warning
```

From Java 16 onward, the issue becomes the type safety of the test:

```
% javac AbstractList_1.java
AbstractList_1.java:8: error: Object cannot be safely cast to List<E>
    if (!(o instanceof List<E>)) { return false; }      // compile-time error
               ^
where E is a type-variable:
  E extends Object declared in class AbstractList_1
Note: AbstractList_1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
1 warning
```

Now the instance check produces an error because there is no possible way, either at compile time or at run time, to test whether the given object belongs to the type `List<E>`.¹

The cast is a different matter. The compiler flags unchecked casts as warnings rather than errors—as with unchecked type comparison expressions—because it cannot usually determine their type safety. In this case, the compiler will perform the cast, but it cannot check whether the list elements are, in fact, of type `E`. Recompiling as suggested, using the `-Xlint` flag, produces this output, following the same compile error diagnostics earlier:

```
AbstractList_1.java:11: warning: [unchecked] unchecked cast
    ListIterator<E> it2 = ((List<E>)o).listIterator(); // unchecked cast
                           ^
required: List<E>
found:    Object
where E is a type-variable:
```

¹ Even if this code worked, there is a further problem. The contract for equality on lists doesn't mention types. A `List<Integer>` should be equal to a `List<Object>` if they contain the same values in the same order. For instance, `[1,2,3]` should be equal to itself, regardless of whether it is regarded as a list of integers or a list of objects.

```
E extends Object declared in class AbstractList_1
1 error
1 warning
```

Unchecked warnings are important: they may well be telling you about a type insecurity that will later produce a run time error (remember the caveat attached to [the cast-iron guarantee!](#)). We discuss how to deal with them in the next section and in “[Eliminate Unchecked Warnings](#)” on page 113.

In this case, both the unchecked warning and the compile error can be eliminated by replacing the nonreifiable type `List<E>` with the reifiable type `List<?>`. Here is a corrected definition (simplified from the actual source):

```
org/jcbook/chapter05/B_instance_tests_and_casts/AbstractList_2
abstract class AbstractList_2<E> extends AbstractCollection<E> implements List<E> {
    public boolean equals(Object o) {
        if (!(o instanceof List<?>)) { return false; }
        ListIterator<E> it1 = listIterator();
        ListIterator<?> it2 = ((List<?>)o).listIterator();
        while (it1.hasNext() && it2.hasNext()) {
            E e1 = it1.next();
            Object e2 = it2.next();
            if (! Objects.equals(e1, e2))
                return false;
        }
        return !(it1.hasNext() || it2.hasNext());
    }
    ...
}
```

In addition to changing the type of the instance test and the cast, the type of the second iterator changes from `Iterator<E>` to `Iterator<?>`, and the type of the second element changes from `E` to `Object`. This code type checks, because even though the element type of the second iterator is unknown, it is guaranteed that it must be a subtype of `Object`, and the nested call to `equals` requires only that its second argument be an object.²

Alternative fixes are possible. For instance, instead of the wildcard types `List<?>` and `Iterator<?>`, you could use the raw types `List` and `Iterator`, which are also reifiable. But we recommend using unbounded wildcard types in preference to raw types because they provide stronger static typing guarantees; many mistakes that are caught as an error when you use unbounded wildcards will only be flagged as a warning if you use raw types. For example, in “[Restrictions on Wildcards](#)” on page 30, we saw the declaration:

```
List<List<?>> lists = new ArrayList<List<?>>();
```

² In contrast to the earlier version, this code satisfies the contract for equality on lists. Now a `List<Integer>` will be equal to a `List<Object>` if they contain the same values in the same order.

The type of `lists` is a list of lists, each one having its own type. Attempting to copy an element from one inner list to another would produce a compilation error. If the type were `<List<List>>`, there would be no constraint on the types of the objects in the inner lists, and copying would produce only an unchecked warning.

Another alternative fix for `AbstractList_1` would be to change the declaration of the first iterator to `Iterator<?>` and of the first element to `Object` so that they match the second iterator and the code will still type check. But we recommend always using type declarations that are as specific as possible; this helps the compiler to catch more errors and to compile more efficient code.

Unchecked Casts

Only rarely will the compiler be able to determine, as in the preceding example, that a successful cast to a nonreifiable type must yield a value of that type. In all other cases, a cast to a nonreifiable type is flagged with an unchecked warning. This is in contrast to type comparisons involving nonreifiable types; for these, the compiler either determines that they are type safe or flags them with an error. We just saw examples of both situations.

In the preceding section, we saw that an unchecked warning means only that the compiler cannot confirm the safety of the code. But no type system is perfect: there will always be some facts that a programmer can deduce but that the type system cannot. Examples of unavoidable but safe unchecked warnings appear in “[How to Define ArrayList](#)” on page 86, “[Array Creation and Varargs](#)” on page 91, “[Cast Through Raw Types When Necessary](#)” on page 121, and “[Use Generic Array Types with Care](#)” on page 122, as well as in numerous places in [Chapter 6](#) and in the [Appendix](#).

We describe how to handle such situations in “[Eliminate Unchecked Warnings](#)” on page 113. In short, once you have documented an argument that your code is in fact type safe, you should use the annotation `@SuppressWarnings("unchecked")` to prevent the compiler from issuing spurious warnings that could have the effect of masking genuine ones.

Unchecked casts in Java are a workaround for a problem that you face, to a greater or lesser extent, in any statically typed language. For comparison, C does not check casts, and unchecked casts in its descendant C++ are much more dangerous than unchecked casts in Java. In either language, the result of an invalid pointer dereference is undefined. This means that the program can crash, silently return an incorrect result, or corrupt unrelated data. Unlike C, the Java runtime guarantees that none of these things can happen even in the presence of unchecked casts. The program may throw a `ClassCastException` and be unable to finish its task, but the JVM remains

in a well-defined state, and unrelated data corruption will not occur. Nonetheless, unchecked casts in Java are a workaround that should be used with caution.

Exception Handling

In a `try` statement, each `catch` clause checks whether the thrown exception (or exceptions) matches a given type. This is the same as the check performed by an instance test, so the same restriction applies: the type must be reifiable. Further, the type in a `catch` clause is required to be a subclass of `Throwable`. Since there is little point in creating a subclass of `Throwable` that cannot appear in a `catch` clause, the Java compiler complains if you attempt to create a parameterized subclass of `Throwable`.

For example, here is a permissible definition of a new exception, which contains an integer value:

```
org/jcbook/chapter05/C_exception_handling/IntegerExceptionTest
class IntegerException extends Exception {
    private final int value;
    public IntegerException(int value) { this.value = value; }
    public int getValue() { return value; }
}
```

And here is a simple example of how to use the exception:

```
org/jcbook/chapter05/C_exception_handling/IntegerExceptionTest
class IntegerExceptionTest {
    public static void main(String[] args) {
        try {
            throw new IntegerException(42);
        } catch (IntegerException e) {
            assert e.getValue() == 42;
        }
    }
}
```

The body of the `try` statement throws the exception with a given value, which is caught by the `catch` clause.

In contrast, the following definition of a new exception is prohibited, because it creates a parameterized type:

```
org/jcbook/chapter05/C_exception_handling/ParametricException
class ParametricException<T> extends Exception { // compile-time error
    private final T value;
    public ParametricException(T value) { this.value = value; }
    public T getValue() { return value; }
}
```

An attempt to compile this code reports an error:

```
% javac ParametricException.java
ParametricException.java:1: a generic class may not extend java.lang.Throwable
class ParametricException<T> extends Exception { // compile time error
^
1 error
```

This restriction is sensible because almost any attempt to catch such an exception must fail, since the type is not reifiable. Corresponding to this restriction on the declaration of exception types, a type in a `catch` clause also cannot have type parameters.

Although subclasses of `Throwable` cannot be parametric, it is possible to use a type variable in the `throws` clause of a method declaration, as in [Example 5-1](#).

Example 5-1. Using a type variable in a `throws` clause

[org/jgcbook/chapter05/C_exception_handling/TypeVariableInThrowsClause](#)

```
public class TypeVariableInThrowsClause {
    @FunctionalInterface
    interface Testable<X extends Throwable> {
        void run() throws X;
    }

    static <X extends Throwable> void checkThrows(Testable<X> test, Class<X> clazz) {
        try {
            test.run();
        } catch (Throwable t) {
            if (clazz.isInstance(t)) {
                return;
            } else {
                throw new AssertionError(t);
            }
        }
        throw new AssertionError("no exception was thrown");
    }

    public static void main(String[] args) {
        checkThrows(() -> List.of().iterator().next(), NoSuchElementException.class);
        checkThrows(() -> { throw new Exception(); }, Exception.class);
    }
}
```

The class `TypeVariableInThrowsClause` is the core of a hypothetical framework to test lambdas that are asserted to throw exceptions of some type. The `checkThrows` method accepts a lambda of the functional type `Testable<X>`, where `X` is a subtype of `Throwable`. It evaluates the lambda, then uses its second parameter, a class object `Class<X>`, to check reflectively that the exception actually thrown by the lambda is an instance of that class.

Generics and Arrays

As we saw in “[Arrays](#)” on page 24, the component type of arrays is reified, meaning that arrays carry run-time information about the type of their components, in contrast to generic data structures, whose type information is erased after compilation. As a result of this mismatch, the fit—or perhaps we should say misfit—between arrays and generics is the worst problem area in the language.

Accordingly, the rest of this chapter is devoted to understanding and working around the problems it causes. Our case study for the next few sections will be `ArrayList`, since studying the implementation of a generic collection backed by an array allows us to explore the major pain points in this troubled relationship and the workarounds for them. As you might expect, the crux of the problem is conversion between collections and arrays, and—because arrays require reified type information that generic collections do not possess—it is specifically conversion *from* generic collections *to* arrays that is the central issue.

The methods provided by the Collections Framework for this conversion are exposed by the interface `java.util.Collection`, the supertype of all collections other than maps:

```
interface Collection<E> {  
    ...  
    Object[] toArray(); ①  
    <T> T[] toArray(IntFunction<T[]> generator) ②  
    <T> T[] toArray(T[] a) ③  
    ...  
}
```

Taking `ArrayList` as the example, let’s look at the implementation of the first two of these methods:

- Method ①: Since `ArrayList` is backed by an `Object[]`, this method is simply implemented by copying that array.
- Method ②: This can be simply defined in terms of method ③. That is how the `Collection` interface defines it, as a default method:

```
default <T> T[] toArray(IntFunction<T[]> generator) {  
    return toArray(generator.apply(0));  
}
```

The generator is a function accepting an `int` and producing a `T[]` of that length. For example, this lambda is a generator for `String` arrays:

```
int n -> new String[n]
```

A typical invocation of method ② would use a method reference equivalent to that lambda:

```
Collection<String> cs = ...  
String[] sa = cs.toArray(String[]::new);
```

Method ❷ applies that function to create an array of the same size as the collection, then calls method ❸ with the result.

- Method ❸ will be the subject of the next three sections. It accepts an array of some reifiable type T. If that array is long enough, it copies the elements of the collection into it, partially or completely overwriting the contents, and returns it; otherwise, it creates a new array of the same type with length equal to the size of the collection, copies the collection elements into it, and returns it.

Before we go on to consider how to implement method ❸, we should acknowledge two questions that its declaration and specification raise:

- Why is the type parameter T unrelated to the collection type E, and what are the consequences of this? The short answer is that the declaration as it stands allows the possibility of giving the array a more specific component type than that of the collection. For a fuller discussion, see “[Making a Collection’s Contents Available for Further Processing](#)” on page 165.
- What should method calls supply for the T[] parameter? That parameter has two purposes. The first, always fulfilled, is to supply the reified type for the returned array, as we will explore in the following sections. The second, fulfilled only if the array is long enough, is to act as the recipient of the collection elements. When you write code calling this method, you have a choice between supplying a pre-allocated array of sufficient length or supplying a smaller (possibly zero-length) array and allowing the method to perform the allocation. This choice is discussed in “[Making a Collection’s Contents Available for Further Processing](#)” on page 165, which reaches the conclusion that for efficiency the best choice is usually ❷.

Generic Array Creation

Our aim here, which we will eventually achieve in “[How to Define ArrayList](#)” on page 86, is to provide a miniature implementation of ArrayList. For the moment, we will focus on method ❸ from the previous section, and to simplify things further, we will pretend for the moment that the specification always requires the creation of a new array, never allowing the supplied array to be reused. For our first attempt, let’s see whether it is possible to avoid using the supplied array altogether:

[org/jgcbook/chapter05/E_generic_array_creation/WrongMicroArrayList_1](#)

```
class WrongMicroArrayList_1<E> {  
    private int size;  
    private Object[] data;  
    ...  
    public <T> T[] toArray(T[] a) {
```

```

    a = new T[size]; // compile error
    System.arraycopy(data, 0, a, 0, size);
    return a;
}

```

The `Object` array `data` will hold the contents of the `ArrayList` and `size` will hold the size of the collection, which will often occupy only part of the array. This version of `toArray` accepts an array reference but simply attempts to assign to the argument variable a reference to a newly created array of the generic type `T`. If that were to succeed, the call to `System::arraycopy` would efficiently copy the contents of the backing array to the newly created array.

Attempted compilation of this program fails, because the emphasized code is creating a new array with a nonreifiable type. Accordingly, the compiler reports a *generic array creation* error:

```

% javac WrongMicroArrayList_1.java
WrongMicroArrayList_1.java:6: generic array creation
    a = new T[c.size()]; // compile-time error
           ^
1 error

```

Explicit generic array creation is forbidden because it would result in a mismatch between the declared type and the reified type of the array. We discuss this problem further, with an example, in the next section.

Generic array creation is one common pitfall: you might also encounter it when using a parameterized type to create an array. For example, consider the following (incorrect) code intended to return an array containing two lists:

org/jgcbook/chapter05/E_generic_array_creation/IndecentExposure

```

class IndecentExposure {
    public static List<Integer>[] twoLists() {
        List<Integer> a = List.of(1, 2, 3);
        List<Integer> b = List.of(4, 5, 6);
        return new List<Integer>[] {a, b}; // compile-time error
    }
}

```

This is an error, because a parameterized type is also not a reifiable type. Again, an attempt to compile this code reports a generic array creation error:

```

% javac IndecentExposure.java
IndecentExposure.java:6: generic array creation
    return new List<Integer>[] {a, b}; // compile-time error
           ^
1 error

```

We discuss this problem further in “[The Principle of Indecent Exposure](#)” on page 88.

Inability to create generic arrays is one of the most serious restrictions in Java. Because it is so annoying, it is worth reiterating the reason it occurs: generic arrays are problematic because generics are implemented via erasure, but erasure is necessary because it enabled evolution.

The best workaround is to use `ArrayList` or some other class from the Collections Framework in preference to an array. We discussed the trade-offs between collection classes and arrays in “[Arrays](#)” on page 24, and we noted that in many cases collections are preferable to arrays: they catch more errors at compile time, they provide more operations, and they offer more flexibility in representation. Often, the best solution to the problems offered by arrays is to “just say no” and use collections in preference to arrays.

Sometimes this won’t work, however, because you need an array for reasons of compatibility or efficiency. Examples of this occur in the Collections Framework: for compatibility, the method `toArray` converts a collection to an array, and for efficiency, the class `ArrayList` (along with many others) is implemented by storing the list elements in an array. We discuss both of these cases in detail in the following sections, together with the associated pitfalls and principles that help you avoid them: the Principle of Truth in Advertising and the Principle of Indecent Exposure. We also consider problems that arise with varargs and generic array creation.

The Principle of Truth in Advertising

We saw in the previous section that naïve creation of an array with a generic collection type will not work. The first fix we might try is to create an array of `Object` and cast its reference to `T[]`. This version of `toArray` compiles, so we can initialize the class with some trivial test data and add a `main` method to exercise `toArray`:

```
org/jgcbook/chapter05/F_the_principle_of_truth_in_advertising/WrongMicroArrayList_2
class WrongMicroArrayList_2<E> {
    private int size = 2;                      // test data
    private Object[] data = {"a", "b"};          // test data
    ...
    public <T> T[] toArray(T[] a) {
        a = (T[])new Object[size];           // unchecked cast
        System.arraycopy(data, 0, a, 0, size);
        return a;
    }
    public static void main(String[] args) {
        var wma = new WrongMicroArrayList_2<>();
        String[] a = wma.toArray(args); // class cast exception
    }
}
```

The compiler issues an unchecked cast warning:

```
% javac -Xlint WrongMicroArrayList_2.java
WrongMicroArrayList_2.java:4: warning: [unchecked] unchecked cast
    a = (T[])new Object[size]; // unchecked cast
          ^
required: T[]
found:   Object[]
where T is a type-variable:
    T extends Object declared in method <T>toArray(Collection<T>)
1 warning
```

The foolhardy developer who ignores this warning will get the following result on running the program:

```
% java WrongMicroArrayList_2
Exception in thread "main" java.lang.ClassCastException: class [Ljava.lang.Object; \
cannot be cast to class [Ljava.lang.String; ([Ljava.lang.Object; and \
[Ljava.lang.String; are in module java.base of loader 'bootstrap')
        at WrongMicroArrayList_2.main(WrongMicroArrayList_2.java:10)
```

The phrase `[Ljava.lang.Object` is how the Java virtual machine denotes the reified type of the array, where `[L` indicates that it is an array of reference type and `java.lang.Object` is the component type of the array. The class cast exception message refers to the line containing the call to `toArray`. This message may be confusing, since that line does not appear to contain a cast!

In order to see what went wrong with this program, let's look at how it is translated using erasure. Erasure drops the type parameter on the collection, replaces occurrences of the type variable `T` with `Object`, and inserts an appropriate cast on the call to `toArray`, yielding the following equivalent code:

[org/jgcbook/chapter05/F_the_principle_of_truth_in_advertising/WrongMicroArrayList_2_Erased](#)

```
class WrongMicroArrayList_2_Erased {
    private int size = 2; // test data
    private Object[] data = {"a", "b"}; // test data
    ...
    public Object[] toArray(Object[] a) {
        a = (Object[])new Object[size]; // unchecked cast
        System.arraycopy(data, 0, a, 0, size);
        return a;
    }
    public static void main(String[] args) {
        var wma = new WrongMicroArrayList_2_Erased();
        String[] arr = (String[])wma.toArray(args); // class cast exception
    }
}
```

When run, the first of these casts succeeds. But although the collection `data`, of whatever type, has been copied into the array returned from `toArray`, the reified component type of that array is `Object`, so the second cast fails.

In order to avoid this problem, you must stick to the following principle:

Principle of Truth in Advertising: The reified type of an array must be a subtype of the erasure of its static type.

This principle is obeyed within the body of `toArray` itself, where the erasure of `T` is `Object`, but not within the `main` method, where `T` has been bound to `String` but the reified type of the array is still `Object`.

Before we learn how to create arrays in accordance with this principle, there is one more point worth stressing. Recall the cast-iron guarantee that accompanies generics for Java: no cast inserted by erasure will fail, so long as there are no unchecked warnings. The preceding principle illustrates the converse: if there are unchecked warnings, then casts inserted by erasure may fail. Further, *the cast that fails may be in a different part of the source code than was responsible for the unchecked warning!* This is why code that generates unchecked warnings must be written with extreme care.

How to Create Arrays

Our previous attempts to create arrays have all involved the use of array creation expressions—that is, expressions of the form `new Type[length]` or `new Type[] {...}`. These attempts failed because array creation expressions require a reifiable type to be supplied at compile time, and the only type available from a generic collection at compile time is nonreifiable. Instead, we must supply the array type at run time. To do that, we turn to a different way of creating arrays: by reflection.

To create an array by reflection, all that is needed is its length and its type. These are provided, respectively, by an `int` and a *type token*—an object that represents a reified type. In the following two subsections, we explore the different possible ways of obtaining type tokens in order to create arrays.

Array Begets Array

“Tis money that begets money,” said Thomas Fuller (1732), observing that one way to get money is to already have money. Following Fuller’s dictum, we can create a new array from an existing one by copying the type token of the latter. Here is our next approach to `ArrayList`, using this technique:

org/jgcbook/chapter05/G_how_to_create_arrays/MicroArrayList_v1

```
class MicroArrayList_v1<E> {
    private int size = 2;                      // test data
    private Object[] data = {"a", "b"};          // test data
    ...
    @SuppressWarnings("unchecked")
    public <T> T[] toArray(T[] a) {
        a = (T[])Array.newInstance(a.getClass().getComponentType(), size);
    }
}
```

```

        System.arraycopy(data, 0, a, 0, size);
        return a;
    }
    public static void main(String[] args) {
        MicroArrayList_v1<String> mal = new MicroArrayList_v1<>();
        String[] arr = mal.toArray(args);
        assert Arrays.equals(arr, new String[]{"a", "b"});
    }
}

```

The emphasized expression, which we'll look at in detail shortly, returns a type token copied from the array supplied to `toArray`. This token is then provided, along with the size of the collection, to the method `java.lang.reflect.Array::newInstance`, which allocates a new array. The result type of the call to `newInstance` is `Object`,³ so an unchecked cast is required to the correct type, `T[]`. The test code calls `toArray`, supplying a reference to the `args` array; `toArray` ignores the contents of `args` and returns a string array containing the elements of the collection, which in this case were supplied when it was initialized.

A Classy Alternative

Some days it may seem that the only way to get money is to have money. The same is not quite true for arrays: an alternative is to use a type token from some other source, such as an instance of the class `java.lang.Class<T>`. The meaning of this parameterization is not intuitive: references to objects of this type, if they are obtained in certain ways, contain compile-time information about the run-time class that they represent. For example, the expression `MyType.class`, using the class literal syntax, has the *static* type `Class<MyType>`, so a reference to it contains type information useful at both compile time and run time. Just how much static type information a `Class` reference contains depends on how it has been obtained. Here are three different ways of obtaining class object references, with the static type information available from each one ranging from none at all up to a precise type:

- The class `java.lang.Class` declares the method `getComponentType`, which we saw earlier used in the example class `MicroArrayList_v1`:

```
public Class<?> getComponentType() {...}
```

If this method is called on an array class, it returns a reference corresponding to the component type of the array. (If called on any class other than an array class, it returns `null`.) The return type, `Class<?>`, tells us that this method cannot provide any compile-time information about the class object that it returns.

³ Why does `Array::newInstance` have a result type of `Object` rather than `Object[]`? Because `newInstance` may return an array of a primitive type such as `int[]`, which is a subtype of `Object` but not of `Object[]`. In fact, that won't happen here, because the type variable `T` must stand for a reference type.

- The class `MicroArrayList_v1` also used the method `Object::getClass`, which is declared as:

```
public <T> Class<?> getClass() {...}
```

but this method gets special treatment from the compiler: the Java Language Specification tells us that “the actual result type is `Class<? extends |X|>` where $|X|$ is the erasure of the static type of the expression on which `getClass` is called” ([Gosling et al. 2023, §4.3.2](#)).

We can understand this best by way of an example like the following one, provided in the Javadoc for `Object`. This code compiles and executes without a cast:

```
Number num = 0;
Class<? extends Number> clazz = num.getClass();
```

When the second line is compiled, the object that `num` refers to is known to belong to some subtype of `Number`. When, at run time, the call to `getClass` is executed, it will return a reference corresponding to the type of that object—i.e., some specific subtype of `Number`. So that information can be embedded in the static type, `Class<? extends Number>`, of the reference to the class object.

- An expression using the class literal syntax `MyType.class` has the precise static type of `Class<MyType>`. So this code also compiles and executes without a cast:

```
Class<Number> clazz = Number.class;
```

We can define a variant of `toArray` that accepts for its type token a reference of type `Class<T>` rather than an array of type `T[]`:

[org/jgcbook/chapter05/G_how_to_create_arrays/MicroArrayList_v2](#)

```
class MicroArrayList_v2<E> {
    private int size = 2;                      // test data
    private Object[] data = {"a", "b"};          // test data
    ...
    @SuppressWarnings("unchecked")
    public <T> T[] toArray(Class<T> clazz) {
        T[] a = (T[])Array.newInstance(clazz, size);
        System.arraycopy(data, 0, a, 0, size);
        return a;
    }
    public static void main(String[] args) {
        MicroArrayList_v2<String> mal = new MicroArrayList_v2<>();
        String[] arr = mal.toArray(String.class);
        assert Arrays.equals(arr, new String[]{"a", "b"});
    }
}
```

The method `toArray` is now passing a reference to `String.class` rather than to a `String[]` and uses it to create an array of components with that reified type, so

in principle `Array::newInstance` could return the precise type of the array. Unfortunately, the cast is still required because, as explained previously, the return type is `Object`, in order to accommodate primitive arrays.

How to Define ArrayList

We are now ready to understand how `ArrayList` works. [Example 5-2](#) presents, in 40 lines of code, a highly simplified version of the 1,800-line implementation of `ArrayList` in OpenJDK. This example should help you understand what to do in the circumstances where you need to use arrays for efficiency or compatibility reasons and to support interconversion with collections. Such implementations must be written with care, as they necessarily involve use of unchecked casts and the need to observe the Principle of Truth in Advertising.

At the end of this section, [Example 5-2](#) will show a miniature version of `ArrayList`, called `MicroArrayList`, that we will derive from `AbstractList`. That class is a “skeletal implementation” for lists (we will explore skeletal implementations in more detail in [Chapter 17](#)). `AbstractList` declares or inherits `get` and `size` as abstract methods and declares the methods `set`, `add`, and `remove` to throw `UnsupportedOperationException`, so all these methods must be overridden.

By contrast, the first and third inherited overloads of `toArray` that we met in “[Generics and Arrays](#)” on page 78 are supported, but they are overridden by `ArrayList` (and `MicroArrayList`) in the interests of efficiency. Further, any array-backed list, like `ArrayList`, should implement `RandomAccess`, a marker interface indicating to generic algorithms that processing a list using `get` is faster than using its iterator.

The class represents a list with elements of type `E` by two private fields: `size`, of type `int`, containing the length of the list, and `arr`, of type `Object[]`, containing the elements of the list. The array must be of length at least equal to `size`, but it may have additional unused elements at the end.

New instances of the backing array are allocated in two places: once in the constructor for the class and once in the `add` method, if the array capacity is insufficient and a new array needs to be created via the utility method `java.util.Arrays::copyOf`. There are two overloads of this method (slightly simplified here):

[org/jgcbook/chapter05/H_how_to_define_arraylist/Arrays](#)

```
public class Arrays {  
    ...  
    @SuppressWarnings("unchecked")  
    public static <T> T[] copyOf(T[] original, int newLength) {  
        return (T[]) copyOf(original, newLength, original.getClass());  
    }  
    public static <T,U> T[] copyOf(U[] original, int newLength,  
        ...  
    }  
}
```

```

    Class<? extends T[]> newType) {
    @SuppressWarnings("unchecked")
    T[] copy = (T[]) Array.newInstance(newType.getComponentType(), newLength);
    System.arraycopy(original, 0, copy, 0, Math.min(original.length, newLength));
    return copy;
}
...
}

```

The first of these methods, called by `MicroArrayList::add`, accepts an array, extracts the type token for the array (*not* the component), and supplies it in a call to the second method (also called by `toArray`). That method then extracts the token for the array component type (the emphasized code) and uses that token in a call to `Array::newInstance`, very like the call made in `MicroArrayList_v2`. These utility methods are good to know whenever you want to copy an array, not just in this case; using them removes the need for your code to use reflection directly and reduces concerns you might have about unchecked casts and the resulting compiler warnings.

The preceding sections have prepared us for the implementation of `toArray`. It is very like the version in `MicroArrayList_v1`, except that instead of calling the method `Array.newInstance` directly, it uses the utility method from `Arrays`. This version finally drops the simplifying pretense that the array will always be newly allocated; it will reuse the supplied array if that is long enough to accommodate the elements of the collection.

The motivation for this feature is to allow the same array to be repeatedly reused as the argument to `toArray`, in order to avoid allocating memory on each occasion. In this situation, when the size of the collection is less than the length of the array, code processing the array elements needs a way to detect when it has finished. That is the purpose of the `null` sentinel placed at the end of the collection data by the `toArray` method.

Example 5-2. ArrayList in miniature

org/jcbook/chapter05/H_how_to_define_arraylist/MicroArrayList

```

class MicroArrayList<E> extends AbstractList<E> implements RandomAccess {
    Object[] data;
    private int size;
    public MicroArrayList() { data = new Object[10]; }
    public int size() { return size; }
    public E get(int i) {
        Objects.checkIndex(i,size);
        return (E)data[i];           // unchecked cast
    }
    public E set(int i, E elt) {
        Objects.checkIndex(i,size);
        E old = (E)data[i];         // unchecked cast
        data[i] = elt;
        return old;
    }
}

```

```

}
public void add(int i, E elt) {
    Objects.checkIndex(i,size+1);
    if (size + 1 > data.length) {
        // ignore possibility of overflow
        data = Arrays.copyOf(data, data.length + (data.length << 1));
    }
    System.arraycopy(data, i, data, i+1, size-i);
    data[i] = elt;
    size++;
}
public E remove(int i) {
    Objects.checkIndex(i,size);
    E old = (E) data[i];      // unchecked cast
    size--;
    System.arraycopy(data, i+1, data, i, size-i);
    data[size] = null;        // release last element for potential garbage collection
    return old;
}
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[])Arrays.copyOf(data, size, a.getClass()); // unchecked cast
    System.arraycopy(data, 0, a, 0, size);
    if (a.length > size) a[size] = null; // sentinel
    return a;
}
public Object[] toArray() { return Arrays.copyOf(data, size); }
}

```

The Principle of Indecent Exposure

Although it is an error to *create* an array with a component type that is not reifiable, it is possible to *declare* an array with such a type or to perform an unchecked cast to such a type. These features must be used with extreme caution, and it is worthwhile to understand what can go wrong if they are not used properly. In particular, a library should *never* publicly expose an array with a nonreifiable type.

“*Arrays*” on page 24 presented an example of why reification is necessary:

org/jgcbook/chapter05/I_the_principle_of_indecent_exposure/Program_1

```

Integer[] ints = {0};
Number[] nums = ints;
nums[0] = 3.14; // array store exception
int n = ints[0];

```

This assigns an array of integers to an array of numbers, and then attempts to store a double into the array of numbers. The attempt raises an array store exception because of the check against the reified type. This is just as well, since otherwise the last line would attempt to store a double into an integer variable.

Here is a similar example, where arrays of numbers are replaced by arrays of *lists* of numbers:

```
org/jgcbook/chapter05/I_the_principle_of_indecency_exposure/Program_2
List[] intListArray = { List.of(0) };
List<Integer>[] intLists = (List<Integer>[])intListArray; // unchecked cast
List<? extends Number>[] numLists = intLists;
numLists[0] = List.of(3.14);
int n = intLists[0].get(0); // class cast exception!
```

This assigns an array of lists of integers to an array of lists of numbers, and then attempts to store a list of doubles into the array of lists of numbers. Although the attempted store should again fail, this time it does not, because the check against the reified type is inadequate: the reified information contains only the erasure of the type, indicating that it is an array of `List`, not an array of `List<Integer>`. Hence, the store succeeds, and the program fails unexpectedly elsewhere.

Example 5-3 presents a similar example, divided into two classes in order to demonstrate how a poorly designed library can create problems for an innocent client.

Example 5-3. Avoid arrays of nonreifiable type

```
org/jgcbook/chapter05/I_the_principle_of_indecency_exposure/DeceptiveLibrary
```

```
// DeceptiveLibrary.java:
public class DeceptiveLibrary {
    public static List<Integer>[] createIntLists(int size) {
        List<Integer>[] intLists =
            (List<Integer>[]) new List[size]; // unchecked cast
        for (int i = 0; i < size; i++)
            intLists[i] = List.of(i+1);
        return intLists;
    }
}
// InnocentClient.java:
class InnocentClient {
    public static void main(String[] args) {
        List<Integer>[] intLists = DeceptiveLibrary.createIntLists(1);
        List<? extends Number>[] numLists = intLists;
        numLists[0] = List.of(1.01);
        int i = intLists[0].get(0); // class cast exception!
    }
}
```

The first class, called `DeceptiveLibrary`, defines a static method that returns an array of lists of integers of a given size. Since generic array creation is not permitted, the array is created with components of the raw type `List`, and a cast is used to give the components the parameterized type `List<Integer>`. The emphasized code generates warnings from the use of the raw type and from the unchecked cast:

```
% javac -Xlint:unchecked DeceptiveLibrary.java
DeceptiveLibrary.java:9: warning: [rawtypes] found raw type: List
    (List<Integer>[]) new List[size]; // unchecked cast
          ^
missing type arguments for generic class List<E>
where E is a type-variable:
    E extends Object declared in interface List
DeceptiveLibrary.java:9: warning: [unchecked] unchecked cast
    (List<Integer>[]) new List[size]; // unchecked cast
          ^
required: List<Integer>[]
found:   List[]

2 warnings
```

Since the array really is an array of lists of integers, the cast appears reasonable, and you might think that this warning could be safely ignored. As we shall see, you do so at your peril!

The second class, called `InnocentClient`, has a `main` method similar to the one in the previous example. Because the unchecked cast appears in the library, no unchecked warning is issued when compiling this code. However, running the code overwrites a list of integers with a list of doubles. Attempting to extract an integer from the array of lists of integers causes the cast implicitly inserted by erasure to fail:

```
%java InnocentClient
Exception in thread "main" java.lang.ClassCastException: class java.lang.Double \
cannot be cast to class java.lang.Integer \
(java.lang.Double and java.lang.Integer are in module java.base of loader \
'bootstrap')
at InnocentClient.main(InnocentClient.java:8)
```

As in the previous section, this error message may be confusing, since that line does not appear to contain a cast!

In order to avoid this problem, you must stick to the following principle:

Principle of Indecent Exposure: Never publicly expose an array where the components do not have a reifiable type.

Again, this is a case where an unchecked cast in one part of the program may lead to a class cast error in a completely different part, where the cast does not appear in the source code but is instead introduced by erasure. Since such errors can be extremely confusing, unchecked casts must be used with extreme caution.

The Principle of Truth in Advertising and the Principle of Indecent Exposure are closely linked. The first requires that the run-time type of an array is properly reified, and the second requires that the compile-time type of an array be reifiable.

An interesting example of violating the Principle of Indecent Exposure is found in the reflection library:

```
TypeVariable<Class<T>>[] java.lang.Class.getTypeParameters()
TypeVariable<Method>[] java.lang.reflect.Method.getTypeParameters()
```

Following the preceding model, it is not hard to create your own version of `InnocentClient` that throws a class cast error at a point where there is no cast, where in this case the role of `DeceptiveLibrary` is played by the official Java library. We discuss the trade-offs in this design in “[Arrays](#)” on page 24.

Array Creation and Varargs

The convenient varargs notation allows methods to accept a variable number of arguments and packs them into an array, as discussed in “[Generic Methods and Varargs](#)” on page 7. However, this notation is not as convenient as you might like, because if the varargs parameter is generic, the result is to create a generic array, with the consequent problems that we have explored in this chapter.

For example, consider this code:

```
org/jgcbook/chapter05/J_array_creation_and_varargs/Varargs

class Varargs {
    static <T> T[] createGenericArray(T... args) {
        return args;
    }
}
```

As its name suggests, `createGenericArray` evades the prohibition, discussed in “[Generic Array Creation](#)” on page 79, on generic array creation. Calls to this method can go badly wrong. For example, it is easy to use it to reproduce the problem that introduced the Principle of Indecent Exposure, as discussed in the previous section:

```
org/jgcbook/chapter05/J_array_creation_and_varargs/Varargs

List<Integer>[] intLists = Varargs.createGenericArray(List.of(1));
List<? extends Number>[] numLists = intLists;
numLists[0] = List.of(3.14);           // heap pollution
int n = intLists[0].get(0);           // class cast exception
```

Heap pollution is the term used to describe the situation in which a variable of a parameterized type refers to an object of a different type. In this case, the variable `numLists[0]`, which is the same variable as `intLists[0]`, has been assigned a value of type `List<Double>` although it was declared of type `List<Integer>`. The invisible cast inserted by the compiler in the last line fails in this situation.

For this reason, compilation of `Varargs` produces two warnings, one for the generic array creation and this one for the heap pollution:

```
% javac -Xlint Varargs.java
Varargs.java:2: warning: [unchecked] Possible heap pollution from parameterized \
vararg type T
    static <T> T[] createGenericArray(T... args) {
                           ^
      where T is a type-variable:
        T extends Object declared in method <T>createGenericArray(T...)
```

Explicit generic array creation is an error, but this implicit creation is allowed, with a warning, because methods with varargs parameters are so useful in practice (`Arrays::asList` is one example; we will see others in [Chapter 16](#)). Like other unchecked warnings, a generic array creation warning invalidates the cast-iron guarantee that accompanies generics. It is not too difficult to take each of the previous examples where a mishap occurs as the result of an unchecked warning and create a similar example using varargs where a generic array creation warning is issued instead.

The compiler warning refers only to *possible* heap pollution, because methods with generic varargs parameters do not necessarily create this danger. For example, if the method `createGenericArray` simply printed out the supplied arguments without returning the created array, a call to it could not possibly result in heap pollution:

```
static <T> void createGenericArray(T... args) {
    for (T elem : args) System.out.println(elem);
}
```

Nonetheless, the compiler still issues a heap pollution warning at the method declaration as well as generic array creation warnings at all its call sites. If the method is in fact type safe—as, for example, `Arrays::asList` is—the warnings triggered by method calls can be very annoying. From Java 7, this problem was solved by the introduction of the method annotation `@SafeVarargs`, which suppresses the compiler warnings at both the method declaration and the call sites. But `@SafeVarargs` does nothing to ensure type safety; rather, it is an assertion that callers can rely on the type safety of the method. So how can we be sure when applying this annotation to a method with a generic varargs parameter that it cannot cause heap pollution? There are two possible dangers:

- We have seen one already: the first version of `createGenericArray` returned a reference to the generic array that the compiler created to accommodate the varargs parameters. This was a direct contravention of the Principle of Indecent Exposure, and it applies to any method that exposes the generic array to untrusted code. (Exposing it to trusted code—for example, passing it to another method annotated with `@SafeVarargs`—is safe.)
- An example of the second way in which a method with varargs parameters can be constructed is by including the code we used to call `createGenericArray` in the method itself:

```
static <T> void createGenericArray(List<Integer>... intLists) {
    List<? extends Number>[] numLists = intLists;
    numLists[0] = List.of(3.14);           // heap pollution
    int n = intLists[0].get(0);          // class cast exception
}
```

In this case, without even needing to expose the array of `List<Integer>` to untrusted code, the method itself has created the danger, by storing a value of type `List<Double>` into it. In general, storing values into generic varargs array parameters is not type safe.

Using `@SafeVarargs`, you can safely write methods with generic varargs parameters without inconvenience to your callers, provided that you avoid these two dangers. Notice that the annotation can only be applied to constructors and methods that cannot be overridden—in other words, those that are private, static, or final.

Instead of the implicit array parameter of varargs, another possibility is to use an explicit `List` parameter instead:

```
static <T> void useListInstead(List<? extends T> args) {
    for (T elem : args) System.out.println(elem);
}
```

As an alternative, this has become more attractive since the variadic `List::of` factory method was introduced in Java 9, allowing the caller to wrap the method arguments conveniently into the list to be supplied. For example:

```
useListInstead(List.of(1, 3.14, "a"));
```

For any number of arguments up to 10, `List::of` has an overload that handles that exact number of arguments. For more than 10 arguments, it has an overload with a varargs parameter. Notice that this idiom is convenient only because it was possible to annotate that overload with `@SafeVarargs`. In the first edition of this book, at Java 6, we recommended against ever using varargs with a nonreifiable type. The current situation is an example of the progressive improvements in the Java language: `SafeVarargs` was added in Java 7 and the `List::of` factory methods in Java 9.

Where Reifiable Types Are Required

Here is a checklist of places where reifiable types are required or recommended:

- A cast should usually be to a reifiable type. (A cast to a nonreifiable type usually results in an unchecked warning.)
- A class that extends `Throwable` must not be parameterized.
- Explicit array instance creation must be at a reifiable type.

- The reified type of an array must be a subtype of the erasure of its static type (see the Principle of Truth in Advertising), and a publicly exposed array should be of a reifiable type (see the Principle of Indecent Exposure).
- A method with a varargs parameter should neither expose the generic array to untrusted code nor store values into it. All such methods should be annotated with `@SafeVarargs`.

These restrictions arise from the fact that generics are implemented via erasure, and they should be regarded as the price one pays for the ease of evolution explored in the [Appendix](#).

For completeness, we'll add to this list a final restriction connected with reflection:

- Type tokens correspond to reifiable types, and the type parameter in `Class<T>` should be a reifiable type. These restrictions are discussed in “[Reflected Types Are Reifiable Types](#)” on page 102.

On the Design of Java Generics

The design space for generics was very large, so the designers' choices provided considerable scope for controversy—some continuing to the present day. Since much (though not all) of this revolves around the question of erasure, we conclude this chapter with discussion of some of the questions that continue to arise on generics design, and consideration of some possible alternatives.

Erasure

Erasure is certainly the most controversial feature of the generics design, from when generics was introduced in Java 5 right up to today. As this chapter has reminded us, its difficulties remain with us, although its original motivation of easing migration from nongeneric code (as described in the [Appendix](#)) is no longer compelling. So since changing the design toward reification is still technically an option, it's worth considering the costs and benefits of erasure at this point.

“[Generics Versus Templates](#)” on page 6 outlined the contrast between C++ template expansion and erasure in Java. The expansion strategy, which maps a parameterized class into a separate class for each instantiation, provides greater type safety (since each instantiation can be separately type checked after expansion) and specialized optimizations for each instantiation. But these advantages come at the cost of code bloat and an inability to abstract over families of parametric types as provided by, for example, Java wildcards. An early experiment comparing template expansion with erasure in a Java prototype compiler ([Odersky et al. 2000](#)) was influential in discouraging further exploration in this direction.

The other option for avoiding erasure would be reification. This would remove some of the difficulties discussed in this chapter, for example by allowing instance tests, heap pollution tests, and code optimizations such as layout specialization to be performed at run time. The question addressed in Goetz (2020) is whether, if a reification scheme could be introduced now, the benefits would outweigh the costs, which include these:

- Run-time costs, in the form of greater static and dynamic footprint, greater class loading costs, greater just-in-time (JIT) compilation costs, and code cache pressure
- Potential fracture of the JVM and language ecosystems, which depend on agreement between multiple JVM vendors and the implementers of multiple languages targeted on the JVM
- The need for extensive modification of the JVM (although there would be limited impact on source code) to handle both reified and nonreified code and to avoid the reappearance of binary compatibility issues

Goetz (2020) considers these issues in greater detail and makes the case for erasure as the best engineering compromise at the time of the introduction of generics—and also now.

Unbounded Wildcards

Arrays must be created with components of reifiable type, so the designers attempted to make the notion of reifiable type as general as possible in order to minimize this restriction. If they had been willing to restrict the notion of reified type, they could have simplified it by excluding types with unbounded wildcards, such as `List<?>`. Had they done so, reifiable types would have become synonymous with unparameterized types (that is, primitive types, raw types, and types declared without a type parameter). This would be consistent with the syntax of class literals, since `List.class` is permitted but `List<?>.class` is illegal.

Arrays

One reason for some of the complexities in the design of generics was to provide good support for the use of arrays. An alternative design would have been simpler, although it would have made arrays somewhat less convenient to use. Let's see what this might have looked like:

- Although array creation is restricted to arrays of reifiable type, it is permitted to declare an array of nonreifiable type or to cast to an array type that is not reifiable, at the cost of an unchecked warning somewhere in the code. As we have

seen, such warnings violate the cast-iron guarantee that accompanies generics and may lead to class cast errors even when the source code contains no casts.

A simpler and arguably safer design would outlaw references to an array of nonreifiable type. This design would mean that you could never declare an array of type `E[]`, where `E` is a type variable.

However, this change would also mean that you could not assign a generic type to the `toArray` method for collections (or similar methods). Instead of:

```
public <T> T[] toArray(T[] arr)
```

we would have:

```
public Object[] toArray(Object[] arr)
```

and many uses of this method would require an explicit cast of the result, making life more awkward for users.

- This alternative design would also require a different implementation of varargs, using lists rather than arrays. Before the introduction of `@SafeVarargs` (see “[Array Creation and Varargs](#)” on page 91), the array implementation of varargs meant that methods with nonreifiable varargs parameters could only be used with a nonreifiable type at the unpleasant cost of separately suppressing unchecked warnings at every call site. However, now that such methods can be called much more conveniently, the advantages of an array implementation become clearer. For example, the compiler can infer the least general type of a series of parameters and make that type available at run time, as in this code:

```
class Varargs {  
    @SafeVarargs  
    static <T> Class<T> getVarargsType(T... args) {  
        @SuppressWarnings("unchecked")  
        Class<T> componentType = (Class<T>) args.getClass().getComponentType();  
        return componentType;  
    }  
    public static void main(String[] args) {  
        assert getVarargsType(3, 4, 5.5).equals(java.lang.Number.class);  
        assert getVarargsType(3, 4, 5).equals(java.lang.Integer.class);  
    }  
}
```

This would not be possible in a varargs implementation based on generic lists.

- The same change would also remove the need for the Principle of Indecent Exposure. We saw at the end of “[The Principle of Indecent Exposure](#)” on page 88 that two methods in the reflection library violate this principle, introducing type insecurity into client code:

```
TypeVariable<Class<T>>[] java.lang.Class.getTypeParameters()  
TypeVariable<Method>[] java.lang.reflect.Method.getTypeParameters()
```

There are arguments both for and against this design choice. The argument against is naturally that we would prefer to avoid type insecurity. The argument in favor is that the reflection library is entirely array-based, for historic reasons, and making these methods collections-based would have created a confusing special case in the API.

Another way of looking at this question parallels the discussion of the collections type system in “[Complementing Static Typing](#)” on page 316. In that section, we consider the type system as only one of the tools available to the designer for communicating and enforcing API semantics. Applying the same argument here, we could observe that the purpose of the API is to provide generic type information at run time. The returned arrays offer this information to a reader. There is no likely use case in which the client will write to the arrays, so the heap pollution that could potentially result is unlikely to be a problem in practice.

No type system is perfect, as we will see for the Collections Framework in [Part II](#) (and will discuss at length in “[Fundamental Issues in the Collections Framework Design](#)” on page 311). This argument leads to viewing type system soundness as just one of the forces, albeit one of the most important, that shape API design.

Arrays::copyOf

The issues involved in choosing constraints for the type parameters of this method connect it with the pragmatic style of analysis of the preceding section. There, the balance between compile-time security and convenience of usage led the designers to choose an unsafe API. Here is another example. The declaration of `Arrays::copyOf` is:

```
static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType)
```

A question that often arises is why the type variables `U` and `T`, representing the source and destination component types, are independent: that is, neither is bounded by the other. The result is that array types can be mismatched, resulting in `ArrayStoreExceptions`.

The obvious fix would be to declare `U` with a bound of `U extends T`. This would certainly prevent some errors, but it would also reduce the flexibility of the method. If such a bound were added, this method could be used only to copy array elements into an array whose component type was a supertype of the original. For example, you could copy all the elements of a `Number[]` into a new `Object[]`.

This would indeed improve type safety, but at the cost of excluding another use case: copying elements from an array into another whose component type is a *sub-type* of the original. (Of course, this can only succeed if the caller has previously ascertained that every element of the array conforms to the destination type.) When

you encounter this use case, you will be happy to have an array-copying utility that supports it. Here is a trivial example to demonstrate this use case:

```
Number[] orig = { 1, 2, 3 };
assert Arrays.stream(array).allMatch(e -> e instanceof Integer);
Integer[] copy = Arrays.copyOf(orig, 3, Integer[].class);
```

This is analogous to safe downcasting of an individual object after type checking using `instanceof`, and in the same way, its safety comes from run-time type checking, not static type checking. The argument of the previous section applies again here: static type safety does not need at all times to be the overriding consideration in API design.

Similar considerations explain the signature of the overloads of `Arrays::sort` that use natural ordering. The documentation of these methods specifies that all elements must implement the `Comparable` interface, but the method declarations accept `Object[]` as their argument:

```
static void sort(Object[] arr)
static void sort(Object[] arr, int fromIndex, int toIndex)
```

The array `arr` might contain only `Comparable` elements, despite having a component type of `Object`. In this case, as with `Arrays::copyOf`, the more liberal type scheme that was adopted allows the array to be sorted directly without the overhead of first copying it into a new `Comparable[]`.

Conclusion

In this chapter, we saw some of the less pleasant consequences of the strategy of erasure, through the complexity of array creation and the Principles of Truth in Advertising and Indecent Exposure, and we evaluated the most important decisions that shaped the design of Java generics.

In the next chapter, we'll study reflection, what it means for the class type `Class<T>` to be generic, and how, although generics is a compile-time feature, generic type information preserved at run time can be accessed through reflection.

CHAPTER 6

Reflection

Reflection is the term for a set of features that allow a program to examine its own definition. Reflection in Java plays a role in class browsers, object inspectors, debuggers, interpreters, services such as JavaBeans™ and object serialization, and any tool that creates, inspects, or manipulates arbitrary Java objects on the fly.

Reflection was present in Java from the beginning, but the introduction of generics changed it in two important ways, introducing both generics for reflection and reflection for generics.

By *generics for reflection*, we mean that some of the types used for reflection became generic types. In particular, the class `Class` became the generic class `Class<T>`. We previously met these parameterized `Class` types in “[How to Create Arrays](#)” on page 83, where we saw the special techniques that class literals and the `Object` method `getClass` use to return more precise type information. In this chapter, we will see how generics are used to especially good effect in annotations and explore the consequences of the fact that the type parameter `T` in `Class<T>` can only be bound to a reifiable type. Additionally, we’ll present a short library that can help you avoid many common cases of unchecked casts.

By *reflection for generics*, we mean that reflection returns information about generic types. There are interfaces to represent generic types, including type variables, parameterized types, and wildcard types, and there are methods that get the generic types of fields, constructors, and methods.

In the following sections, we explain each of these points in turn.



The code examples for this chapter can be found at:

https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter06

Generics for Reflection

Central to the Java reflection API is the class `Class`, which represents information about the type of an object at run time. In “[How to Create Arrays](#)” on page 83, we met different ways of obtaining a class object. Two are of interest here:

- You can call the method `Object::getClass`, which returns a reference of type `Class<? extends |X|>`, where `|X|` is the erasure of the static type of the receiver of `getClass`.
- You can write a class literal of the form `MyType.class`, which returns a reference of type `Class<MyType>`.

A reference of type `Class<T>` contains both run-time and static type information: the class object itself represents the run-time type, and `T` represents the static type.

Here is an example:

[org/jgcbook/chapter06/A_generics_for_reflection/Program_1](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter06/A_generics_for_reflection/Program_1)

```
Class<Integer> ki = Integer.class;
Number n = Integer.valueOf(42);
Class<? extends Number> kn = n.getClass();
assert ki == kn;
```

For a given class loader, the same type is always represented by the same class object. To emphasize this point, here we compare class objects using identity (the `==` operator). This is in fact the test for `Class` equality: `Class` does not override the `equals` method inherited from `Object`, which is implemented as identity comparison. Not overriding `Object::equals` makes sense when there will be only one instance of the class (and in some other circumstances; see [Bloch 2017, item 10](#)).

This example shows how the compiler treats class literals and the `getClass` method: if `T` is a type without type parameters, then `T.class` has type `Class<T>`, and if `e` is an expression of type `T`, then `e.getClass()` has type `Class<? extends T>`. (We’ll see what happens when `T` does have type parameters in the next section.) The wildcard is needed because the type of the object referred to by the variable may be a subtype of the type of the variable—as in this case, where a variable of type `Number` contains an object of type `Integer`.

Often, when you're using reflection, you don't know the static type of the class object. In those cases, you must write `Class<?>` for the type, using an unbounded wildcard. When it is available, however, the type information provided by the type parameter can be valuable, as in the method `toArray` of `MicroArrayList_v2` (see “[A Classy Alternative](#)” on page 84):

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(Class<T> clazz) {
    T[] a = (T[])Array.newInstance(clazz, size);      // unchecked cast
    System.arraycopy(data, 0, a, 0, size);
    return a;
}
```

Here, the type parameter guarantees that the unchecked cast is safe.

The class `Class<T>` contains some methods that use the type parameter in an interesting way:

```
class Class<T> {
    public T newInstance(); ①
    public boolean isInstance(Object obj); ②
    public T cast(Object o); ③
    public Class<? super T> getSuperclass(); ④
    public boolean isAssignableFrom(Class<?> cls); ⑤
    public <U> Class<? extends U> asSubclass(Class<U> k); ⑥
    public <A extends Annotation> A getAnnotation(Class<A> annotationClass); ⑦
    public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass); ⑧
    ...
}
```

Method ① returns a new instance of the class, which will, of course, have type `T`. Method ③ casts an arbitrary object to the class of the receiver, and so it either throws a class cast exception or returns a result of type `T`. To avoid the exception, you can use method ② to test whether the object is an instance of this class. Method ④ returns the superclass, which must have the specified type. Method ⑥ checks that the receiver class is a subclass of the argument class, and either throws a class cast exception or returns the receiver with its type suitably changed. Again, you can avoid the exception by first calling method ⑤ to ensure that this class is indeed a subclass of the argument.

Methods ⑦ and ⑧ are part of the annotation facility. These methods are interesting because they show how the type parameter for classes can be used to good effect. For example, `Retention` is a subclass of `Annotation`, so you can extract the `Retention` meta-annotation on an annotation class `ac` as follows:

```
Retention r = ac.getAnnotation(Retention.class);
```

Here, the generic type provides two advantages. First, it means that no cast is required on the result of the call, because the generic type system can assign it precisely the correct type. Second, it means that if you accidentally call the method

supplying a class object for a class that is not a subclass of `Annotation`, this is detected at compile time rather than at run time.

In another interesting use of class objects, the convenience class `Collections` contains a method that builds a wrapper that checks whether every element added to or extracted from the given list belongs to the given class. (There are similar methods for other collection classes, such as sets and maps.) It has the following signature:

```
public static <T> List<T> checkedList(List<T> l, Class<T> k)
```

The wrapper supplements static checking at compile time with dynamic checking at run time, which can be useful for improving security or interfacing with legacy code (see “[Checked Collections](#)” on page 282). The implementation calls the method `Class::cast`, described earlier, where the receiver is the class token passed into the method, and the cast is applied to any element read from or written into the list using `get`, `set`, or `add`. Once again, the type parameter on `Class<T>` means that the code of `checkedList` requires no additional casts beyond the calls to `Class::cast`, and that the compiler can check that the method is called with an appropriate class object.

Reflected Types Are Reifiable Types

Reflection makes only reified type information available to the program. Necessarily, therefore, every class token corresponds to a reifiable type. If you try to reflect a parameterized type, you get the reified information for the corresponding raw type:

```
org/jgcbook/chapter06/B\_reflected\_types\_are\_reifiable\_types/Program\_1
List<Integer> ints = new ArrayList<Integer>();
List<String> strs = new ArrayList<String>();
assert ints.getClass() == strs.getClass();
assert ints.getClass() == ArrayList.class;
```

Here, the type list of integers and the type list of strings are both represented by the same class object, the class literal for which is written `ArrayList.class`.

Because the class always represents a reifiable type, there is no point in supplying a nonreifiable type parameter to the class `Class`. Hence, the two main methods for producing a class with a type parameter, namely the `getClass` method and class literals, are both designed to yield a reifiable type for the type parameter in all cases.

Recall that the `getClass` method is treated specially by the compiler. If expression `e` has type `T`, then the expression `e.getClass()` has type `Class<? extends |T|>`, where `|T|` is the erasure of the type `T`. Here’s an example:

```
org/jgcbook/chapter06/B\_reflected\_types\_are\_reifiable\_types/Program\_2
List<Integer> ints = new ArrayList<Integer>();
Class<? extends List> k = ints.getClass();
assert k == ArrayList.class;
```

Here the expression `ints` has type `List<Integer>`, so the expression `int.getClass()` has type `Class<? extends List>`; this is the case because erasing `List<Integer>` yields the raw type `List`. The actual value of `k` is `ArrayList.class`, which has type `Class<ArrayList>`, which is indeed a subtype of `Class<? extends List>`.

Class literals are also restricted; it is not even syntactically valid to supply a type parameter to the type in a class literal. Thus, the following fragment is illegal:

org/jgcbook/chapter06/B_reflected_types_are_reifiable_types/ClassLiteral

```
class ClassLiteral {  
    public Class<?> k = List<Integer>.class; // syntax error  
}
```

Indeed, Java's grammar makes a phrase such as the preceding one difficult to parse, and it results in a series of syntax error messages:

```
% javac ClassLiteral.java  
ClassLiteral.java:6: error: <identifier> expected  
    public Class<?> k = List<Integer>.class; // syntax error  
                           ^  
ClassLiteral.java:6: error: <identifier> expected  
    public Class<?> k = List<Integer>.class; // syntax error  
                           ^  
2 errors
```

This syntax problem leads to an irregularity. Everywhere else that a reifiable type is required, you may supply either a raw type (such as `List`) or a parameterized type with unbounded wildcards (such as `List<?>`). However, for class literals, you must supply a raw type. No wildcards, not even unbounded ones, may appear; replacing `List<Integer>` with `List<?>` in the preceding code leads to a similar error cascade.

Note that class literals represent class objects, which are created by the JVM at run time. So in a type of the form `Class<T>`, `T` can only represent a type known at run time—that is, a reifiable type. This is in line with the syntax restriction that allows only raw types to be used to create class literals. Although a variable or method parameter can be declared with a bounded parametric type like `Class<? extends T>`, the object it refers to will always have some specific reifiable type, in this case a subtype of `T`. The same is true for array types: as we saw in [Chapter 5](#), it is possible to declare a variable typed to an array of a nonreifiable type, such as `List<? extends Number>[]`, but the array that it refers to must always be created with a reifiable type.

Reflection for Primitive Types

Every type in Java, including primitive types and array types, has a class literal and a corresponding class object.

For instance, `int.class` denotes a class object for the primitive type for integers (this token is also the value of the static field `Integer.TYPE`). The type of this token cannot be `Class<int>`, since `int` is not a reference type, so it is taken to be `Class<Integer>`. Arguably, this is an odd choice, since according to this type you might expect the calls `int.class.cast(o)` and `int.class.newInstance()` to return values of type `Integer`, but in fact these calls raise an exception. Similarly, you might expect the call:

```
java.lang.reflect.Array.newInstance(int.class, size)
```

to return a value of type `Integer[]`, but in fact the call returns a value of type `Object` (as we saw in “[How to Create Arrays](#)” on page 83), which can only then be cast to `int[]`. These examples suggest that it might have made more sense to give the literal `int.class` the type `Class<?>`.

On the other hand, `int[].class` denotes a class object for arrays with components of the primitive type `int`, and its type is `Class<int[]>`, which is permitted since `int[]` is a reference type.

A Generic Reflection Library

As we’ve seen, careless use of unchecked casts can lead to problems, such as violating the [Principle of Truth in Advertising](#) or the [Principle of Indecent Exposure](#). One technique to minimize the use of unchecked casts is to encapsulate these within a library. The library can be carefully scrutinized to ensure that its use of unchecked casts is safe, while code that calls the library can be free of unchecked casts.

[Example 6-1](#) provides a library of generic functions that use reflection in a type-safe way. It defines a convenience class, `GenericReflection`, containing the following methods:

```
public static <T> T newInstance(T object)
public static <T> Class<? extends T> getComponentType(T[] a)
public static <T> T[] newArray(Class<? extends T> k, int size)
public static <T> T[] newArray(T[] a, int size)
```

The first takes an object, finds the class of that object, and returns a new instance of the class; this must have the same type as the original object. The second takes an array and returns a class token for its component type, as carried in its run-time type information. Conversely, the third allocates a new array with its component type specified by a given class token and a specified size. The fourth takes an array and a size and allocates a new array with the same component type as the given array and the given size; it simply composes calls to the previous two methods. The code for each of the first three methods consists of a call to one or two corresponding methods in the Java reflection library and an unchecked cast to the appropriate return type.

Unchecked casts are required because the methods in the Java reflection library cannot return sufficiently accurate types, for various reasons. The `getComponentType`

method is in the class `Class<T>`, and Java provides no way to restrict the receiver type to be `Class<T[]>` in the signature of the method (though the call raises an exception if the receiver is not a class token for an array type). The method `newInstance` in `java.lang.reflect.Array` must have the return type `Object` rather than the return type `T[]`, because it may return an array of a primitive type. The method `getClass`, when called on a receiver of type `T`, returns a token not of type `Class<? extends T>` but of type `Class<?>`, because of the erasure that is required to ensure that class tokens always have a reifiable type. However, in each case the unchecked cast is safe, and users can call on the four library routines defined here without violating the cast-iron guarantee.

Example 6-1. A type-safe library for generic reflection

org/jgcbook/chapter06/D_a_generic_reflection_library/GenericReflection

```
class GenericReflection {
    public static <T> T newInstance(T obj) throws ReflectiveOperationException {
        Object newobj = obj.getClass().getConstructor().newInstance();
        return (T)newobj; // unchecked cast
    }
    public static <T> Class<? extends T> getComponentType(T[] a) {
        Class<?> k = a.getClass().getComponentType();
        return (Class<? extends T>)k; // unchecked cast
    }
    public static <T> T[] newArray(Class<? extends T> k, int size) {
        if (k.isPrimitive()) {
            throw new IllegalArgumentException ("Argument cannot be primitive: "+k);
        }
        Object a = java.lang.reflect.Array.newInstance(k, size);
        return (T[])a; // unchecked cast
    }
    public static <T> T[] newArray(T[] a, int size) {
        return newArray(getComponentType(a), size);
    }
}
```

The first method uses `Constructor::newInstance` (in `java.lang.reflect`) in order to avoid the use of `Class.newInstance`, which has long been deprecated because its propagation of exceptions thrown by the nullary constructor bypasses compile-time exception checking. `Constructor::newInstance` avoids this problem by wrapping exceptions thrown by the constructor in a (checked) `InvocationTargetException`. However, the calls of `getConstructor` and `newInstance` can throw other exceptions; the common superclass is `ReflectiveOperationException`.

The second method is guaranteed to be well typed in any program that obeys the **Principle of Indecent Exposure** and the **Principle of Truth in Advertising**. The first principle guarantees that the component type at compile time will be a reifiable type,

and the second principle guarantees that the reified component type returned at run time is a subtype of the reifiable component type declared at compile time.

The third method raises an `IllegalArgumentException` if its class argument is a primitive type. This catches the following tricky case: if the first argument is, say, `int.class`, then its type is `Class<Integer>`, but the new array will have type `int[]`, which is not a subtype of `Integer[]`. This problem would not have arisen if `int.class` had the type `Class<?>` rather than `Class<Integer>`, as discussed in the preceding section.

As an example of the use of the first method, here is a method that copies a collection into a fresh collection of the same kind, preserving the type of the argument:

```
org/jcbook/chapter06/D_a_generic_reflection_library/Program_1
```

```
public static <T, C extends Collection<T>> C copy(C coll) {
    try {
        C copy = GenericReflection.newInstance(coll);
        copy.addAll(coll);
        return copy;
    } catch (ReflectiveOperationException e) {
        throw new RuntimeException(e);
    }
}
```

Calling `copy` on an `ArrayList<Integer>` returns a new `ArrayList<Integer>`:

```
org/jcbook/chapter06/D_a_generic_reflection_library/Program_1
```

```
List<Integer> coll = new ArrayList<>(List.of(1, 2, 3));
assert copy(coll).equals(coll);
assert copy(coll).getClass().equals(coll.getClass());
```

Similarly, calling `copy` on a `HashSet<String>` returns a new `HashSet<String>`, and so on.

As an example of the use of the last method, here is the `toArray` method from “[The Principle of Truth in Advertising](#)” on page 81, rewritten to replace its unchecked casts with a call to the generic reflection library:

```
org/jcbook/chapter06/D_a_generic_reflection_library/Program_2
```

```
public static <T> T[] toArray(Collection<T> c, T[] a) {
    if (a.length < c.size()) a = GenericReflection.newArray(a, c.size());
    int i=0; for (T x : c) a[i++] = x;
    if (i < a.length) a[i] = null;
    return a;
}
```

In general, we recommend that if you need to use unchecked casts you should encapsulate them into a small number of library methods, as we’ve done here. Don’t let unchecked code proliferate through your program!

Reflection for Generics

Generics change the reflection library in two ways. We have discussed generics for reflection, where Java added a type parameter to the class `Class<T>`. In this section, we'll discuss reflection for generics, where Java adds methods and classes that support access to generic types.

Example 6-2 shows a simple demonstration of the use of reflection for generics. It uses reflection to find the class associated with a given name, and it prints out the class together with its fields, constructors, and methods using the reflection library classes `Field`, `Constructor`, and `Method`. The two methods available for converting a class, field, constructor, or method to a string for printing are `toString` and `toGenericString`. The less informative pre-generics implementation of `toString` is maintained mainly for backward compatibility. A small sample class is shown in **Example 6-3**, and a sample run with this class is shown in **Example 6-4**.

Example 6-2. Using reflection for generics

org/jcbook/chapter06/E_reflection_for_generics/ReflectionForGenerics

```
class ReflectionForGenerics {  
    public static void toString(Class<?> k) {  
        System.out.println(k + " (toString)");  
        System.out.println(k);  
        for (Field f : k.getDeclaredFields())  
            System.out.println(f.toString());  
        for (Constructor<?> c : k.getDeclaredConstructors())  
            System.out.println(c.toString());  
        for (Method m : k.getDeclaredMethods())  
            System.out.println(m.toString());  
        System.out.println();  
    }  
    public static void toGenericString(Class<?> k) {  
        System.out.println(k + " (toGenericString)");  
        System.out.println(k.toGenericString());  
        for (Field f : k.getDeclaredFields())  
            System.out.println(f.toGenericString());  
        for (Constructor<?> c : k.getDeclaredConstructors())  
            System.out.println(c.toGenericString());  
        for (Method m : k.getDeclaredMethods())  
            System.out.println(m.toGenericString());  
        System.out.println();  
    }  
    public static void main (String[] args) throws ClassNotFoundException {  
        for (String name : args) {  
            Class<?> k = Class.forName(name);  
            toString(k);  
            toGenericString(k);  
        }  
    }  
}
```

Example 6-3. A sample class

org/jcbook/chapter06/E_reflection_for_generics/Cell

```
class Cell<E> {  
    private E value;  
    public Cell(E value) { this.value=value; }  
    public E getValue() { return value; }  
    public void setValue(E value) { this.value=value; }  
    public static <T> Cell<T> copy(Cell<T> cell) {  
        return new Cell<T>(cell.getValue());  
    }  
}
```

Example 6-4. A sample run

```
% java ReflectionForGenerics Cell  
class Cell (toString)  
class Cell  
private java.lang.Object Cell.value  
public Cell(java.lang.Object)  
public java.lang.Object Cell.getValue()  
public void Cell.setValue(java.lang.Object)  
public static Cell Cell.copy(Cell)  
  
class Cell (toGenericString)  
class Cell<E>  
private E Cell.value  
public Cell(E)  
public E Cell.getValue()  
public void Cell.setValue(E)  
public static <T> Cell<T> Cell.copy(Cell<T>)
```

The sample run shows that although the reified type information for objects and class tokens contains no information about generic types, the actual bytecode of the class does encode information about generic types as well as erased types. The information about generic types is essentially a comment. It is ignored when running the code, and it is preserved only for use in reflection.

Reflecting Generic Types

The reflection library provides a `Type` interface to describe a generic type. There is one class that implements this interface and four other interfaces that extend it, corresponding to the five different kinds of types:

- The class `Class`, representing a primitive type or raw type
- The interface `ParameterizedType`, representing an application of a generic class or interface to parameter types, from which you can extract an array of the parameter types

- The interface `TypeVariable`, representing a type variable, from which you can extract the bounds on the type variable
- The interface `GenericArrayType`, representing an array, from which you can extract the array component type
- The interface `WildcardType`, representing a wildcard, from which you can extract the lower and upper bounds on the wildcard

By performing a series of instance tests on each of these interfaces, you may determine which kind of type you have. You can then print or process the type; we will see an example of this shortly.

Methods are available to return the superclass and superinterfaces of a class as types, and to access the generic type of a field, the argument types of a constructor, and the argument and result types of a method.

You can also extract the type variables that stand for the formal parameters of a class or interface declaration, or of a generic method or constructor. The type for type variables takes a parameter and is written `TypeVariable<D>`, where `D` represents the type of object that declared the type variable. Thus, the type variables of a class have type `TypeVariable<Class<?>>`, while the type variables of a generic method have type `TypeVariable<Method>`. Arguably, this type parameter is a questionable design decision: it is not particularly helpful, and its inclusion is responsible for the problem described at the end of “[The Principle of Indecent Exposure](#)” on page 88.

[Example 6-5](#) uses these methods to print out all of the header information associated with a class. Here are two examples of its use:

```
% java ReflectionDemo java.util.AbstractList
class java.util.AbstractList<E>
extends java.util.AbstractCollection<E>
implements java.util.List<E>

% java ReflectionDemo java.lang.Enum
class java.lang.Enum<E extends java.lang.Enum<E>>
implements java.lang.constant.Constable,java.lang.Comparable<E>,java.io.Serializable
```

The code in [Example 6-5](#) contains methods to print the declaration of a class, its superclass, and the interfaces that it implements. The core of the code is the method `printType`, which uses a pattern-matching `switch` statement to classify a type according to the five cases in the previous list.

Example 6-5. Manipulating the type Type

org/jgbook/chapter06/F_reflecting_generic_types/ReflectionDemo

```
class ReflectionDemo {  
    public static String printSuperclass(Type sup) {  
        if (!sup.equals(Object.class)) {  
            return "extends " + printType(sup) + "\n";  
        }  
        return "";  
    }  
    public static String printInterfaces(Type[] implem) {  
        if (implem.length > 0) {  
            return "implements " + Arrays.stream(implem)  
                .map(ReflectionDemo::printType)  
                .collect(Collectors.joining(",")) + "\n";  
        }  
        return "";  
    }  
    public static String printTypeParameters(TypeVariable<?>[] vars) {  
        if (vars.length > 0) {  
            return Arrays.stream(vars)  
                .map(b -> b.getName() + printBounds(b.getBounds()))  
                .collect(Collectors.joining(",","<",">\\n"));  
        }  
        return "";  
    }  
    public static String printBounds(Type[] bounds) {  
        if (bounds.length > 0 && ! Arrays.equals(bounds,new Type[]{ Object.class })) {  
            return " extends " +  
                Arrays.stream(bounds)  
                    .map(ReflectionDemo::printType)  
                    .collect(Collectors.joining(" & "));  
        }  
        return "";  
    }  
    public static String printType(Type type) {  
        return switch(type) {  
            case Class<?> cls -> cls.getName();  
            case ParameterizedType p -> printParameterizedType(p);  
            case TypeVariable<?> tv -> tv.getName();  
            case GenericArrayType gat -> gat.getGenericComponentType() + "[]";  
            case WildcardType wt -> printWildcard(wt);  
            default -> throw new IllegalStateException("Unexpected value: " + type);  
        };  
    }  
    private static String printParameterizedType(ParameterizedType p) {  
        Class<?> c = (Class<?>)p.getRawType();  
        Type o = p.getOwnerType();  
        String ptString = o != null ? printType(o) + "." : "";  
        ptString = ptString + c.getName();  
        return ptString + Arrays.stream(p.getActualTypeArguments())  
            .map(ReflectionDemo::printType)  
            .collect(Collectors.joining(",","<",">"));  
    }  
    private static String printWildcard(WildcardType wt) {
```

```

Type[] upper = wt.getUpperBounds();
Type[] lower = wt.getLowerBounds();
if (lower.length == 1) {
    return "? super " + printType(lower[0]);
} else if (upper.length == 1 && ! upper[0].equals(Object.class)) {
    return "? extends" + printType(upper[0]);
}
return "";
}
public static String printClass(Class<?> c) {
    return "class " +
        printType(c) +
        printTypeParameters(c.getTypeParameters()) +
        printSuperclass(c.getGenericSuperclass()) +
        printInterfaces(c.getGenericInterfaces());
}
public static void main(String[] args) throws ClassNotFoundException {
    for (String name : args) {
        Class<?> c = Class.forName(name);
        System.out.println(printClass(c));
    }
}
}

```

Conclusion

In this chapter, we explored the relationship between reflection and generics in the form of the class `Class<T>`, which combines compile-time and reified type information, and the reflection techniques that allow access to the generic type information preserved at run time.

The next chapter will conclude **Part I** of the book with a number of ideas about how to use generics in practice and to work around problems that sometimes arise using generics in the course of practical programming life.

Effective Generics

This chapter contains advice on how to use generics effectively in practical coding. These items vary in their applicability, from good practice that you should always try to follow, through to techniques that are only to be used as a last resort. Each item is introduced with an indication of where it lies on this scale.

The title of this chapter is an homage to Joshua Bloch's book, *Effective Java (2017)*.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter07](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter07)

Eliminate Unchecked Warnings

Unchecked warnings are rarely unavoidable. This item describes the unusual situations in which you cannot avoid an unchecked warning, and shows how to handle it in these cases.

We mentioned in “[Unchecked Casts](#)” on page 75 some of the situations in which the compiler is unable to verify that an unchecked warning is unnecessary. Here is another example. The `promote` method in this code promotes a list of objects to a list of strings, if the list of objects contains only strings, and throws a class cast exception otherwise. The `main` method is a simplistic test of `promote` that uses `assert` statements to show the expected execution path of the program.

Run with assertions enabled, it will not throw an assertion error:

```
org/jgcbook/chapter07/A\_eliminate\_unchecked\_warnings/Promote\_1
class Promote_1 {
    public static List<String> promote(List<Object> objs) {
        for (Object o : objs)
            if (!(o instanceof String))
                throw new ClassCastException();
        return (List<String>)(List<?>)objs; // unchecked cast
    }
    public static void main(String[] args) {
        List<Object> objs1 = List.of("one", "two");
        List<Object> objs2 = List.of(1, "two");
        List<String> strs1 = promote(objs1);
        assert strs1 == (List<?>)objs1;
        try {
            List<String> strs2 = promote(objs2);
            assert false;
        } catch (ClassCastException e) {
            // should always get here
        }
    }
}
```

The method `promote` loops over the list of objects and throws a class cast exception if any object is not a string. Hence, when the last line of the method is reached, it is safe to cast the list of objects to a list of strings.

But the compiler cannot deduce this, so the programmer must use an unchecked cast. It is illegal to cast a list of objects to a list of strings, so the cast must take place in two steps. First, we cast the list of objects into a list of wildcard type; this cast is safe. Second, we cast the list of wildcard type into a list of strings; this cast is permitted but generates an unchecked warning:

```
% javac -Xlint:unchecked Promote.java
Promote.java:7: warning: [unchecked] unchecked cast
    return (List<String>)(List<?>)objs; // unchecked cast
                                         ^
required: List<String>
found:   List<CAP#1>
where CAP#1 is a fresh type-variable:
  CAP#1 extends Object from capture of ?
1 warning
```

Since we know that this cast is indeed type safe, it is legitimate to suppress the warning. In fact, we recommend that you do so, as otherwise the result of many such false alarms appearing among the compiler diagnostics may lead you to ignore them all—very possibly including genuine ones. This is the purpose of the annotation `@SuppressWarnings`. It follows from the justification for suppressing warnings that you should:

- Restrict the scope of the annotation as much as possible, to avoid the possibility of masking other, genuine, warnings in its scope.
- Provide the justification for suppressing the warning, for the benefit of anyone subsequently maintaining the code.

Implementing these recommendations (which follow [Bloch 2017, item 27](#)), the previous example becomes:

```
org/jgcbook/chapter07/A\_eliminate\_unchecked\_warnings/Promote\_2
class Promote_2 {
    public static List<String> promote(List<Object> objs) {
        for (Object o : objs)
            if (!(o instanceof String))
                throw new ClassCastException();
        @SuppressWarnings("unchecked") // this cast is type safe because the code can
                                // be reached only if all the list elements are
                                // strings
        List<String> strings = (List<String>) (List<?>) objs;
        return strings;
    }
}
```

Notice that the new local declaration avoids the necessity of annotating the entire method and instead allows the `@SuppressWarnings` annotation to control only the single following line.

As always with recommendations of good practice, this advice should be followed pragmatically. If you have many unchecked warnings within a single method, following it to the letter may lead to the code becoming cluttered and difficult to read. In such situations, widening the scope of the annotation to the method or even class level may be justifiable. For similar reasons, we have sometimes deviated from it in this book, especially where unchecked warnings in example code are fully explained in the narrative.

Finally, let us repeat that before suppressing unchecked warnings, you need to be very sure that you understand the reason for the warning and that it is harmless. Otherwise, to suppress a warning now is to exchange it for an exception later!

Enforce Type Safety When Calling Untrusted Code

Many systems require your code to interoperate with other, untrusted, code. This item explains how to reduce the risks involved.

It is important to be aware that the guarantees offered by generic types apply only if there are no unchecked warnings. This means that generic types are much less useful for ensuring safety or security in code written by others, since you have no way of knowing whether that code raised unchecked warnings when it was compiled.

Say we have a class that defines an order, with a subclass that defines an authenticated order:

```
org/jcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/Order
class Order { ... }
class AuthenticatedOrder extends Order { ... }
```

Interfaces specify suppliers and processors of orders. Here, the supplier is required to provide only authenticated orders, while the processor handles all kinds of orders:

```
org/jcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/Order
interface OrderSupplier {
    public void addOrders(List<AuthenticatedOrder> orders);
    public List<AuthenticatedOrder> getOrders();
}
interface OrderProcessor {
    public void processOrders(List<? extends Order> orders);
}
```

From the types involved, you might think that the following broker guarantees that only authenticated orders can pass from the supplier to the processor:

```
org/jcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/NaiveBroker
class NaiveBroker {
    public void connect(OrderSupplier supplier, OrderProcessor processor) {
        List<AuthenticatedOrder> orders = new ArrayList<>();
        supplier.addOrders(orders);
        processor.processOrders(orders);
    }
}
```

But a devious supplier may, in fact, supply unauthenticated orders in two ways. Here is the first:

```
org/jcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/DeviousSupplier_1
class DeviousSupplier_1 implements OrderSupplier {
    public void addOrders(List<AuthenticatedOrder> orders) {
        List raw = orders;
        Order order = new Order(); // not authenticated
        raw.add(order);           // unchecked call
    }
    public List<AuthenticatedOrder> getOrders() {
        ...
    }
}
```

Compiling the devious supplier will issue an unchecked warning, but the broker has no way of knowing this.

Incompetence can cause just as many problems as deviousness. Any code that issues unchecked warnings when compiled could cause similar problems, perhaps simply

because the author made a mistake. And pre-generic legacy code may present the same problem without being either devious or incompetent, but simply by working with raw types.

The correct solution is for the broker to pass a checked list to the supplier:

```
org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/WaryBroker_1
class WaryBroker_1 {
    public void connect(OrderSupplier supplier, OrderProcessor processor) {
        List<AuthenticatedOrder> orders = new ArrayList<AuthenticatedOrder>();
        supplier.addOrders(Collections.checkedList(orders, AuthenticatedOrder.class));
        processor.processOrders(orders);
    }
}
```

Now a class cast exception will be raised if the supplier attempts to add anything to the list that is not an authenticated order.

Checked collections cannot be used in this way when the supplier returns a collection that it has itself created. This is the second way in which `DeviousSupplier` can supply unauthenticated orders:

```
org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/DeviousSupplier_2
class DeviousSupplier_2 implements OrderSupplier {
    public void addOrders(List<AuthenticatedOrder> orders) {
        ...
    }
    public List<AuthenticatedOrder> getOrders() {
        List<Order> orders = new ArrayList<Order>();
        orders.add(new Order());
        return (List<AuthenticatedOrder>)(List)orders; // unchecked cast
    }
}
```

In this case, it is up to the client code to check that the returned collection does not contain objects of an incorrect type. To avoid a class cast exception occurring much later, making diagnosis of the problem difficult, a wary client will check the returned collection immediately in order to detect type violations as near as possible to their cause. The emphasized code in this example will throw an exception if the returned list contains an object of the wrong type:

```
org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/WaryBroker_2
class WaryBroker_2 {
    public void connect(OrderSupplier supplier, OrderProcessor processor) {
        List<AuthenticatedOrder> orders = supplier.getOrders();
        for (AuthenticatedOrder order : orders) {}
    }
}
```

A third technique to avoid type violations is specialization to create a special type of list that can contain only authorized orders. We discuss this next.

Specialize to Create Reifiable Types

This item describes a technique that is useful and sound, but not commonly required.

Parameterized types are not reifiable, so operations such as instance tests, casting, and array creation, which apply only to reifiable types, cannot be used with them. In such cases, one workaround is to create a specialized version of a parameterized type. We will use the technique of inheritance from an abstract collection, delegating to a concrete one (see “[Customize Collections Using the Abstract Classes](#)” on page 305 for a fuller explanation). [Example 7-1](#) shows how to specialize lists to strings; specializing to other types is similar.

Example 7-1. Specializing to create a reifiable type

[org/jgcbook/chapter07/C_specialize_to_create_reifiable_types/ListString](#)

```
class ListString extends AbstractList<String> implements List<String> {  
    private final List<String> list;  
    public ListString() { this.list = new ArrayList<>(); }  
    public ListString(Collection<? extends String> c) { this.list = new ArrayList<>(c); }  
    public ListString(int capacity) { this.list = new ArrayList<>(capacity); }  
    public int size() { return list.size(); }  
    public String get(int i) { return list.get(i); }  
    public String set(int i, String s) { return list.set(i,s); }  
    public String remove(int i) { return list.remove(i); }  
    public void add(int i, String s) { list.add(i,s); }  
}
```

This declares `ListString` (an unparameterized type, hence reifiable) to be a subtype of `List<String>` (a parameterized type, hence not reifiable). Thus, every value of the first type also belongs to the second, but not vice versa.

Here is an example of its use:

[org/jgcbook/chapter07/C_specialize_to_create_reifiable_types/Program_1](#)

```
ListString listString1 = new ListString(List.of("one", "two"));  
ListString listString2 = new ListString(List.of("seven", "eight"));  
List<? extends List<?>> lists =  
    Arrays.asList(listString1, List.of(3, 4), List.of("five", "six"), listString2);  
ListString[] listStringArray = lists.stream()  
    .filter(list -> list instanceof ListString)  
    .toArray(ListString[]::new);  
assert Arrays.toString(listStringArray).equals("[[one, two], [seven, eight]]");
```

This creates a list of lists, then scans it for those lists that implement `ListString` and places them into an array. Array creation, instance tests, and casts now pose no problems, as they act on the reifiable type `ListString` rather than the nonreifiable type `List<String>`. Notice that a `List<String>` that has not been wrapped will not be recognized as an instance of `ListString`; that is why the third list in the list of lists is not copied into the array.

Implementing the `ListString` class is a straightforward application of the technique described in “[Customize Collections Using the Abstract Classes](#)” on page 305. A `ListString` instance is a view of the underlying `ArrayList` that will raise a compile error if any attempt is made to insert an element other than a string.

This code has been kept simple in the interests of a concise explanation. Other variations are possible: a more efficient version might skip the use of `AbstractList` and instead directly delegate all 25 methods of the `List` interface together with the `toString` method. You could choose an implementation that, instead of creating an internal list, wraps a supplied list so that type safety is guaranteed at run time by compiler-inserted casts. You also might want to provide additional methods in the `ListString` interface, such as an `unwrap` method that returns the underlying `List<String>`, or a version of `subList` that returns a `ListString` rather than a `List<String>` by recursively applying `wrap` to the delegated call.

Avoid Single-Use Type Variables

The advice in this item is universally applicable.

The type parameters to generic methods are normally present to express some constraint on the type or types of the method arguments and/or the return value. For example, a method to sum the values in a collection could constrain its single parameter to be a collection of `Number` instances:

```
public static <T extends Number> T sum(Collection<T> c);
```

Alternatively, any type may be acceptable, but the type parameter is used to express some relationship between different method parameters or between a parameter and the return type. The declaration of `java.util.Collections::synchronizedList`, for instance, allows it to accept any list type but constrains the return value to be of the same type:

```
public static <T> List<T> synchronizedList(List<T> list);
```

Sometimes, however, a generic method has no such constraint. For example, the `Collections` method `shuffle`, which randomly permutes the elements of a list, can accept a list of any type. It might be tempting to declare it in the same way as the preceding examples:

```
public static <T> void shuffle(List<T> list);
```

and in fact this declaration is perfectly legal. But it is an obscure and confusing way to convey the meaning that the list type is unconstrained; in this situation, a wildcard expresses the same meaning in a more concise and explicit way:

```
public static void shuffle(List<?> list);
```

The key advantage to using a wildcard is the reduction of cognitive load on developers using the method API. A reader seeing a type parameter naturally looks for the constraints that it represents. By contrast, a wildcard immediately conveys the message that there are no constraints on this type.

Use Generic Helper Methods to Capture a Wildcard

The technique described in this item is useful when you need to define a method with wildcard parameters.

In “[Wildcard Capture](#)” on page 29, we saw that a naïve implementation of the method `Collections::reverse` fails because the compiler will not permit references of the type `Object` to be assigned to elements of a `List<?>`:

```
org/jgcbook/chapter07/E\_use\_generic\_helper\_methods\_to\_capture\_wildcard/Program\_1
public static void reverse(List<?> list) {
    List<Object> tmp = new ArrayList<>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1)); // compile-time error
    }
}
```

This fails because it is not legal to write from the copy back into the original; we are trying to write from a list of objects into a list of unknown type:

```
Capture.java:8: error: incompatible types: Object cannot be converted to CAP#1
    list.set(i, tmp.get(list.size()-i-1)); // compile-time error
                                         ^
where CAP#1 is a fresh type-variable: CAP#1 extends Object from capture of ?
```

If, however, the temporary list is given the same type as the method parameter, the method compiles without error:

```
org/jgcbook/chapter07/E\_use\_generic\_helper\_methods\_to\_capture\_wildcard/Program\_2
public static <T> void reverse(List<T> list) {
    List <T> tmp = new ArrayList<>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1));
    }
}
```

Now the problem is the one described in the previous section: the public API of `reverse` defines a type variable without any semantics.

The solution is to combine the two implementations, exposing the wildcard declaration as the public API and implementing it by calling the generic method as a private helper method:

[org/jgcbook/chapter07/E_use_generic_helper_methods_to_capture_wildcard/Program_3](#)

```
public static void reverse(List<?> list) { rev(list); }

private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1));
    }
}
```

This technique is often useful when defining methods whose declarations contain wildcards.

Cast Through Raw Types When Necessary

We present this technique with a strong warning: avoid using it if possible!

Casting through a raw type is a way of evading the constraints of the generic type system. Those constraints exist for a reason, so of course this technique comes with a price: casting back from a raw type to a generic type generates an unchecked warning, and the programmer bears the full responsibility of preventing run-time class cast exceptions. But it can be useful when you need to work around the limitations of Java's generic type system.

As an example, consider how you could modify the technique of wildcard capture. The helper method used in the previous section carries small costs in efficiency, in the form of increased class file and code cache sizes and the performance overhead of an extra method call. These are not normally significant, but it may be necessary for critical code in intensively used libraries to make every possible optimization. So `Collections::reverse` uses a raw type, in code equivalent to this:

[org/jgcbook/chapter07/F_cast_through_raw_types/Program_1](#)

```
@SuppressWarnings({"rawtypes", "unchecked"})
public static void reverse(List<?> list) {
    List tmp = new ArrayList(list);          // Unchecked call to ArrayList::new
    for (int i = 0; i < list.size(); i++) {
        ((List) list).set(i, tmp.get(list.size()-i-1)); // Unchecked call to set
    }
}
```

Using raw types in this way avoids creating and calling an extra method just to get the generic types to work, at the cost of generating unchecked warnings, which have to be suppressed and the suppression justified: in this case type safety is guaranteed, because the elements we put into the list were all obtained from it in the first place. It is unusual for this to be worthwhile, but it is a useful technique to know for cases in which performance and code size are critical.

Use Generic Array Types with Care

This item shows the usefulness of generic array types, while reinforcing the caution advised by the Principle of Indecent Exposure.

In “[The Principle of Indecent Exposure](#)” on page 88, we saw that a generic array type, like `List<Integer>[]` in [Example 5-3](#), is a kind of convenient fiction: because the parametric type is nonreifiable, the array component type is a raw `List`, so it is possible to assign a `List` of some other nonreifiable type into the array without causing a compile-time error or even a run-time `ArrayStoreException`. The Principle of Indecent Exposure warns against exposing such a generic array type outside the context in which the array was created, because the type system provides no safety in this case.

Handled with care, however, generic array types can be useful. For example, consider constructing a nonbinary tree, in which each node can have an arbitrary number of child nodes. Usually, you would model a node with a list of children:

```
org/jgcbook/chapter07/G\_use\_generic\_array\_types\_with\_care/ListNode
class ListNode<T> {
    private T data;
    private final List<ListNode<T>> children;
    public ListNode(T data) {
        this.data = data;
        children = new ArrayList<>();
    }
    public void addChild(ListNode<T> child) { this.children.add(child); }
    public void removeChild(ListNode<T> child) { this.children.remove(child); }
    public T getData() { return data; }
    public List<ListNode<T>> getChildren() { return children; }
}
```

But in a performance-critical application, one in which the number of child nodes could be very large, you might prefer to use an array to contain the child node references. Now adding and removing children becomes impossible, but data values and entire child nodes can be replaced:

```
org/jgcbook/chapter07/G\_use\_generic\_array\_types\_with\_care/TreeNode
class TreeNode<T> {
    private T data;
    private final TreeNode<T>[] children;
    public TreeNode(T data, int childCount) {
        this.data = data;
        this.children =
            (TreeNode<T>[] ) new TreeNode[childCount]; // unchecked cast
    }
    public void replaceData(T newData) { this.data = newData; }
    public void replaceChild(TreeNode<T> child, int index) {
        this.children[index] = child;
    }
}
```

```
public ArrayTreeNode<T>[] getChildren() { return children; }
public T getData() { return data; }
}
```

This will all work perfectly well, provided references to the generic array are contained within the class, where the dangers of using a covariant subclass of the array are clearly visible. If, however, the API contravenes the Principle of Indecent Exposure, as the method `getChildren` does, client code runs the risk of unexpected class cast exceptions:

[org/jgobook/chapter07/G_use_generic_array_types_with_care/Program_1](#)

```
ArrayTreeNode<Integer> integerNode = new ArrayTreeNode<>(3, 2);
ArrayTreeNode<? extends Number>[] numberTreeNodes = integerNode.getChildren();
numberTreeNodes[0] = new ArrayTreeNode<>(1.0, 2);
Integer data = integerNode.getChildren()[0].getData(); // class cast exception
```

Use Type Tokens for Run-Time Type Information

This item shows that type tokens can be useful in wider situations than those in which we met them, in [Chapter 5](#).

We saw in [“A Classy Alternative” on page 84](#) that a type token of the form `Class<T>` can be used to create an array of component type `T`. This is an example of a general technique useful in implementing generic methods and classes that need to access or instantiate types specified at run time. For example, a generic factory method specified for classes having a no-argument constructor could look like this:

[org/jgobook/chapter07/H_use_type_tokens_for_run_time_typing/Program_1](#)

```
public static <T> T createInstanceParameterized(Class<T> clazz) {
    try {
        Constructor<T> constructor = clazz.getDeclaredConstructor();
        constructor.setAccessible(true);
        return constructor.newInstance();
    } catch (ReflectiveOperationException e) {
        throw new RuntimeException(e);
    }
}
```

A nonparameterized version of `createInstance` would work equally well at run time:

[org/jgobook/chapter07/H_use_type_tokens_for_run_time_typing/Program_1](#)

```
public static Object createInstanceWildcard(Class<?> clazz) {
    try {
        Constructor<?> constructor = clazz.getDeclaredConstructor();
        constructor.setAccessible(true);
        return constructor.newInstance();
    } catch (ReflectiveOperationException e) {
        throw new RuntimeException(e);
    }
}
```

But the parameterized version is type safe, so no cast is necessary when using it:

```
org/jgcbook/chapter07/H_use_type_tokens_for_run_time_typing/Program_1  
class Foo {};  
Foo f1 = createInstanceParameterized(Foo.class);  
Foo f2 = (Foo) createInstanceWildcard(Foo.class);
```

Besides object creation, type tokens can also be used for type-safe access to objects whose type is unknown at compile time. For example, consider implementing a registry in which each type token is mapped to a list of objects of that type:

```
org/jgcbook/chapter07/H_use_type_tokens_for_run_time_typing/TypeSafeRegistry  
public class TypeSafeRegistry {  
    private Map<Class<?>,List<?>> typeSafeRegistry = new HashMap<>();  
  
    public <T> void addObject(Class<T> type, T object) {  
        getObjects(type).add(object);  
    }  
  
    @SuppressWarnings("unchecked")  
    public <T> List<T> getObjects(Class<T> type) {  
        List<?> untypedList =  
            typeSafeRegistry.computeIfAbsent(type, k -> new ArrayList<>());  
        return (List<T>) untypedList; // unchecked cast  
    }  
}
```

Although the cast in the method `getObjects` is unchecked, we can be confident that it is safe because the method `addObject` will only accept objects whose type matches the type of the token being used as a key.

Conclusion

This chapter concludes Part I of the book. Now that we have an appreciation of the fundamental benefits of generics, as well as how to get around the difficulties that we sometimes encounter in using them, we are ready to move on to their most important area of application: the Java Collections Framework.

PART II

Collections

The Java Collections Framework is a set of interfaces and classes in the packages `java.util` and `java.util.concurrent`. They provide client programs with various models of how to organize their objects, and various implementations of each model. These models are sometimes called *abstract data types*, and we need them because different programs need different ways of organizing their objects. In one situation, you might want to organize your program's objects in a sequential list because their ordering is important and there are duplicates. In another, a set might be the right data type because now ordering is unimportant and you want to discard the duplicates. These two data types (and others) are represented by different interfaces in the Collections Framework, and we will look at examples of their use in this chapter. But that's not all! None of these data types has a single "best" implementation—that is, one implementation that is better than all the others for all the operations. For example, a linked list may be better than an array implementation of lists for inserting and removing elements from the middle, but much worse for random access. So choosing the right implementation for your program involves knowing how it will be used as well as what is available.

This part of the book is structured as follows:

Chapter 8, “The Main Interfaces of the Java Collections Framework”

Introduces the Java Collections Framework interfaces, with an example illustrating the primary purpose of each and an overview of how they fit together to form a (mostly) coherent whole

Chapter 9, “Preliminaries”

Explores the many background concepts and design influences that help to understand how the Collections Framework works

Chapter 10, “The Collection Interface”

Explores the `Collection` interface, the original root of the Collections Framework, and introduces the example that will be developed in the remaining chapters

Chapter 11, “The SequencedCollection Interface”

Provides a brief overview of `SequencedCollection`, the first new interface in the Collections Framework since Java 6 and the first to bring together existing features rather than introducing new ones

Chapter 12, “Sets”

Begins the detailed examination of each of the major interfaces with a treatment of the operations and implementations of `Set`

Chapter 13, “Queues”

Studies the features of `Queue`, the interface whose implementations support producer/consumer interaction, central to many large systems

Chapter 14, “Lists”

Explores the methods and implementations of the universally popular `List` interface

Chapter 15, “Maps”

Examines the last of the five main Collections Framework interfaces: `Map`, the most important one for maintaining system state in a concurrently updatable form

Chapter 16, “The Collections Class”

Describes the special-purpose and generic collection algorithms in the utility class `Collections`

Chapter 17, “Guidance for Using the Java Collections Framework”

Provides guidelines for using the Java Collections Framework in the design and implementation of large—and not-so-large—systems

Chapter 18, “Design Retrospective”

Reviews some of the major design decisions that formed the framework, seen in the perspective of the community’s long experience of its use

The Main Interfaces of the Java Collections Framework

A *collection* is an object that provides access to a group of objects, allowing them to be processed in a uniform way. A *collections framework* provides a uniform view of a set of collection types specifying and implementing common data structures, following consistent design rules so that they can work together. Figure 8-1 shows the main interfaces of the Java Collections Framework, all in the package `java.util`, together with one other—`java.lang.Iterable`—which is outside the framework. `Iterable` (see “[Iterable and Iterators](#)” on page 135) defines the contract (see “[Contracts](#)” on page 149) that a class—any class, not only one of those in the framework—must implement in order to be used as the target for an “enhanced for statement,” usually called a *foreach* statement.

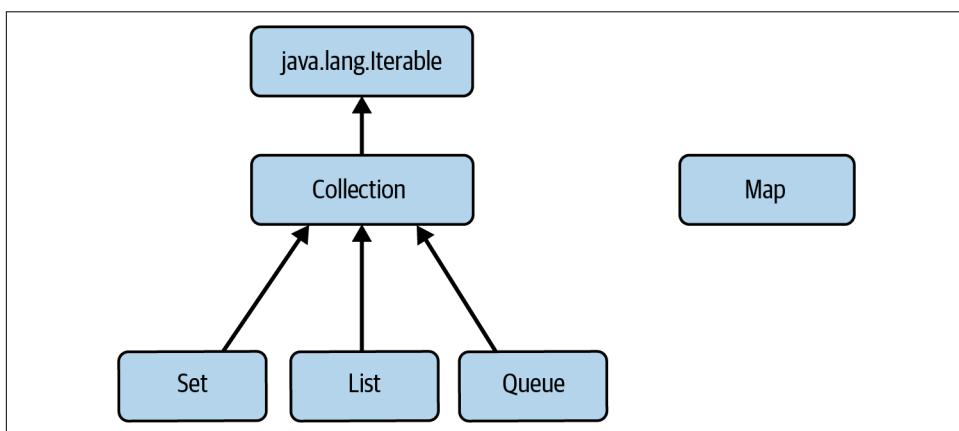


Figure 8-1. The main interfaces of the Java Collections Framework

The interfaces in [Figure 8-1](#)—together with the subinterfaces that we will meet in this chapter—define the structure and functionality of the Collections Framework. You should choose the collections to use in a program according to the functionality of the interfaces as described here, and as far as possible your variables should have interface rather than implementation types. The choice of implementations comes later in program design.

The central interface in [Figure 8-1](#) is `Collection`, which exposes the core functionality required of any collection other than a `Map`. Its methods support managing elements by adding or removing single or multiple elements, checking membership of a single or multiple values, and inspecting and exporting elements. It has no direct concrete implementations; the concrete collection classes all implement one of its subinterfaces as well.

Using the Different Collection Types

We can use the simple example of a word cloud to explore the ways in which the different collection types in [Figure 8-1](#) can be useful in implementing a solution.

Word cloud generators take a list of words for their input, normally ignoring the order, so any of the word clouds in [Figure 8-2](#) could be generated from this list of names of [the Three Stooges](#):

`"Larry", "Curly", "Moe"`

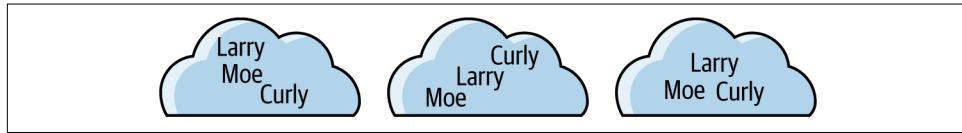


Figure 8-2. Simple word cloud

Let's consider the different ways in which the input to a word cloud generator could be represented. If you already know something about the Collections Framework, you might take a moment to consider the options before reading on.

Set

The simplest way to represent the names would be as a `Set`, a collection in which order is not significant and there can be no duplicates. It adds no operations to `Collection`, but the contracts for its element-adding operations specify that they will not create duplicates.

A Set containing the names of the Stooges can be created by this JShell code:

```
jshell> Set<String> stoogesSet = Set.of("Larry", "Curly", "Moe")
stoogesSet ==> [Curly, Larry, Moe]
```

To emphasize the unimportance of ordering in a set, you will find that if you run this code yourself more than once in different instances of JShell, the order of the names in the output will vary from one run to another.

Let's now make the problem a little more realistic. The whole point of a word cloud is that the font size of a word should depend on the number of its repetitions in the input to the generator. So, since Sets cannot store duplicate elements, they're not actually a suitable data structure for representing the contents of a word cloud. For a collection type that does accommodate duplicates, we can turn to List.

List

A List is a collection in which order is significant and which accommodates duplicate elements. It adds operations supporting indexed access.

The word cloud generated in this JShell snippet:

```
jshell> List<String> stoogesList = List.of("Larry", "Curly", "Larry", "Moe")
stoogesList ==> [Larry, Curly, Larry, Moe]
```

could look like one of those in [Figure 8-3](#), reflecting the fact that the string "Larry" occurs twice as often as the other two. In this case, although the order of the words in each word cloud is still random, that is not because the order has been lost from the collection; on the contrary, lists maintain their order. But that order would be ignored by a word cloud generator.

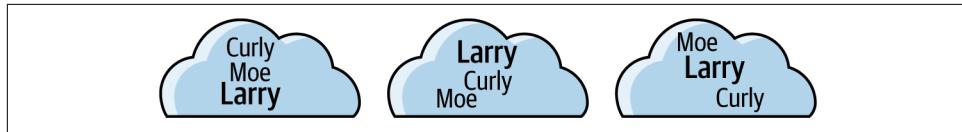


Figure 8-3. Word clouds generated from List data

Since any Set can be represented by a List, why have a separate data type at all? The reason is that the operations of Set make it much easier to guarantee element uniqueness, when that is what you actually want, and being able to disregard order can allow Set operations to be much more efficient.

In this case, though, it's important to preserve the number of occurrences of each element, so that disqualifies Set. List is a better choice, but is it the best? It has two significant disadvantages: it preserves an arbitrary (and in this example, meaningless) order, and more importantly, to count how many occurrences there are of a single

element, you have to examine every element of the list. This may not matter in our toy example, but real-life data structures often contain tens of millions of elements.

So a still better representation of the input list would be a `Map`, a collection that can create an association of each word with the frequency of its occurrence and that allows these associations (or *entries*) to be efficiently found.

Map

A `Map` is a collection that uses key-value entries to store and retrieve elements. The keys of a `Map` form a `Set`, so they cannot be duplicated and ordering is not significant. The operations of `Map` allow it to be maintained by addition and removal of single or multiple key-value entries and to have these associations queried. `Maps` are used in situations in which it is important that read and write operations on key-value associations should be easy and, especially, fast.

Here is the JShell code to generate a `Map` recording the frequency of each of the names in our word cloud example:

```
jshell> Map<String, Integer> stoogesMap = Map.of("Curly", 1, "Larry", 2, "Moe", 1)
stoogesMap ==> {Curly=1, Moe=1, "Larry", 2}
```

You can think of a `Map` as a table, like those shown in [Figure 8-4](#).

Key	Value
"Larry"	2
"Curly"	1
"Moe"	1

Key	Value
"Curly"	1
"Larry"	2
"Moe"	1

Key	Value
"Moe"	1
"Larry"	2
"Curly"	1

Figure 8-4. The word cloud data represented as a Map

The three tables in [Figure 8-4](#) represent the same `Map`, because the table rows—that is, the key-value pairs that make up the `Map`—form a `Set`, so their order is not significant; again, running the script in different JShell instances will produce different orders for the key-value pairs. Further, the keys also form a `Set`, so there can't be two rows with the same key (say, "Larry"). So `Map` suffers neither of the two disadvantages of `List`: there is no need to maintain the order of its key-value pairs, and discovering the frequency of occurrence of one of its elements only requires a table lookup, which, as we will see, can be very fast.

This may still not be enough, however. To achieve the greatest density in the cloud, real-life word cloud applications place the “largest” words (those with the greatest frequency and therefore font size) first, and then fit progressively smaller ones around

them.¹ So it turns out that in this case we do in fact need the table entries to be ordered, in a specific way. The Collections Framework provides for that requirement with the `SequencedMap` interface, introduced in Java 21 to unify the preexisting capabilities of different Map implementations.

SequencedMap

A `SequencedMap` is a Map that maintains its entries in a defined order. Some implementations of `SequencedMap` sort their entries automatically according to a key ordering. The following code uses one such implementation (`TreeMap`) to create the map pictured in Figure 8-5; the keys are word frequencies, listed in descending order, each one mapped to a set of words with that frequency. If you aren't familiar with Java streams, don't worry about understanding this rather complex code in detail; you just need to see that the variable `stoogesSequencedMap` presents the data in the right form for a word cloud algorithm. If the algorithm simply steps through the entries one by one, it will encounter the entries in descending order of font size. In each entry, the value references a set of words for the cloud, to be rendered in a font size determined by the entry's key:

```
jshell> import static java.util.stream.Collectors.*  
jshell> SequencedMap<Integer, Set<String>> stoogesSequencedMap =  
...> Map.of("Curly", 1, "Larry", 2, "Moe", 1)  
...> .entrySet().stream()  
...> .collect(groupingBy(Map.Entry::getValue,  
...> () -> new TreeMap<Integer, Set<String>>(Comparator.reverseOrder()),  
...> mapping(Map.Entry::getKey, toSet())));  
  
stoogesSequencedMap ==> {2=[Larry], 1=[Moe, Curly]}
```

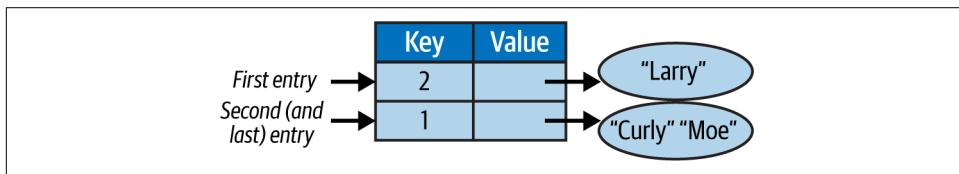


Figure 8-5. Word cloud data represented as a `SequencedMap`

Queue

Queue is different in kind from the collections we have met so far. Sets, lists, and maps are typically “owned” by another object and form part of its state (see “[Respect the ‘Ownership’ of Collections](#)” on page 289 for a more detailed explanation of this idea and its implications). By contrast, queues are not usually owned by a single object, but are used for transmission of values from *producers* to *consumers*. A queue can

¹ The whole process is quite complex; you can read about it in detail in Steele and Iliinsky (2010, Chapter 3).

have multiple producers and multiple consumers; these can be objects, or threads, or processes.

Although `Queue` inherits the operations of `Collection`, these operations are used in a different way from the other collection types: producers use methods that add to the *tail* of the queue, and consumers use methods that remove from its *head*. In this way, information flows from producers to consumers. Like other collections, a queue has storage, but this storage is temporary, and elements are stored only after being *enqueued* by a producer and while waiting to be *dequeued* by a consumer. So you should think of queues as channels or conduits of information between objects, rather than being part of the state of another object.

Sequenced Collections

`SequencedMap` is only one of a number of collection types that preserve order—and, sometimes, impose it. These *sequenced collections* differ from `Collection`, `Set`, or `Map` in that they have a defined order, called in the documentation an *encounter order* (see [Chapter 11](#)). They differ from `Queue`, which also has a defined order, in that they can be iterated in either direction. The ordering of sequenced collections can be derived in two different ways: for some, like `List`, elements retain the order in which they were added, whereas for others, like `NavigableSet` (see “[Sequenced-Set and NavigableSet](#)” on page 133), the ordering is dictated by the values of the elements. These are sometimes called *externally ordered* and *internally ordered* types, respectively. These terms reflect the difference between an order that is arbitrarily imposed on the elements, for example by the order in which they are added, and an order that is an inherent property of the elements themselves, such as alphabetic ordering on strings.

A new interface, `SequencedCollection`, was introduced Java 21 to unify the sequenced collections, of both kinds. [Figure 8-6](#) adds the sequenced interfaces of the framework to those of [Figure 8-1](#).

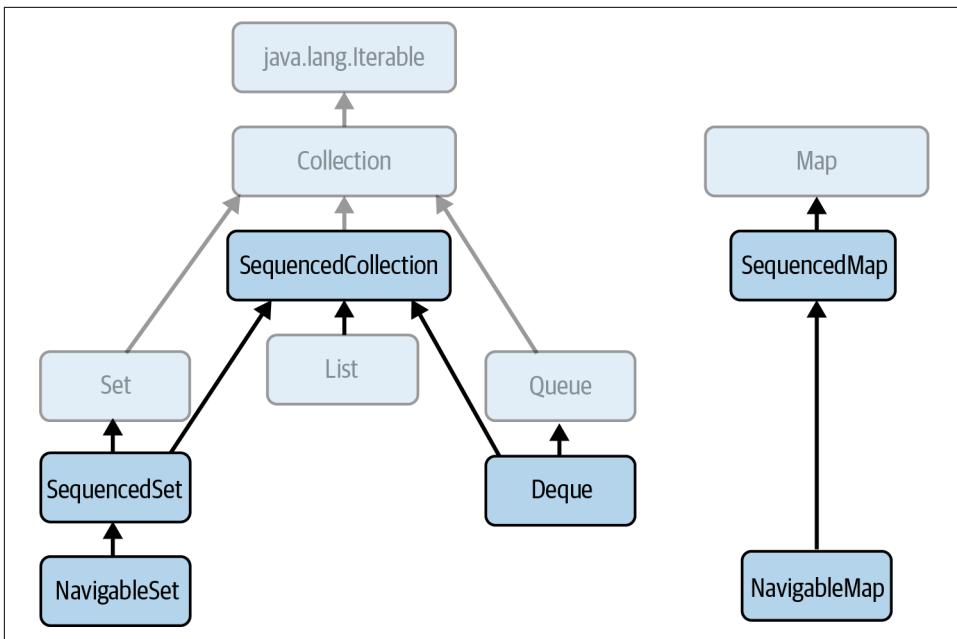


Figure 8-6. The sequenced interfaces of the Java Collections Framework

SequencedCollection

Collections generally support operations for addition (`add`), removal (`remove`), and, in some cases, access (`get`) of elements. The `SequencedCollection` interface provides versions of these that can be applied to the first and last element of the collection: `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst`, and `getLast`. In addition, `SequencedCollection` provides a reversed view—that is, a way of working with the collection as though the ordering has been reversed. This simplifies many programming problems and often provides more efficient implementations. We explore sequenced collections and views in more detail, in [Chapter 11](#) and “[Views](#)” on page [141](#), respectively.

SequencedSet and NavigableSet

A `SequencedSet` is an externally or internally ordered `Set` that also exposes the methods of `SequencedCollection`. A `NavigableSet` is an internally ordered `SequencedSet` that therefore also automatically sorts its elements, and provides additional methods to find elements adjacent to a target value.

Deque

A **Deque** (pronounced “deck”) is a double-ended queue that can both accept and yield up elements at either end. A **Deque**, like a **Queue**, can be used as a conduit of information between producers and consumers. The ability to remove elements from the tail facilitates *work stealing*, a load-balancing technique in which idle threads “steal” tasks from busier threads to maximize parallel efficiency. **Deques** can also be used to store the state of an object, if updates to the state require operations at either end.

SequencedMap and NavigableMap

A **SequencedMap** is a **Map** whose keys form a **SequencedSet**. A **NavigableMap** is a **SequencedMap** whose keys form a **NavigableSet** so that its entries are automatically sorted by the key ordering and its methods can find keys and key-value pairs adjacent to a target key value.

Conclusion

The brief outline in this chapter had the purpose of quickly familiarizing you with the main features of the Collections Framework. Chapters 10 through 15 concentrate on each of the Collections Framework interfaces in turn. First, though, [Chapter 9](#) covers some fundamental background ideas that inform the design of the Collections Framework and will help you to use collections more effectively.

CHAPTER 9

Preliminaries

To understand any framework, it's as important to have a grasp of the ideas underlying its design as it is to be familiar with the details of its implementation. This chapter explores the concepts behind the Java Collections Framework—concepts that provide essential motivation for the design of the individual collection classes.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter09](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter09)

Iterable and Iterators

For the very common requirement of processing every element of a collection in the same way, we must *iterate* over it. This ability is provided, in three different ways, by the interface `java.lang.Iterable<E>`. As this is a superinterface of `Collection`, its methods are common to every nonmap collection. It exposes three methods:

<code>void forEach(Consumer<? super T> action)</code>	Performs action for each element of the Iterable
<code>Iterator<T> iterator()</code>	Returns an iterator over elements of type T
<code>Spliterator<T> spliterator()</code>	Creates a Spliterator over the elements described by this Iterable

We defer discussion of the purpose of `Spliterators` to “Parallel Streams” on page 153. The other two methods, however are relevant here. The first provides the simplest means of iterating over a collection: for example, to print every element of a collection `coll` to the terminal, we can simply write:

```
coll.forEach(System.out::println);
```

If, however, the body of the iteration is more complex than can be conveniently expressed in a single `Consumer`, the `Iterator` returned by the second method can be used in two ways. `Iterator` exposes three methods:

boolean <code>hasNext()</code>	Return <code>true</code> if the iteration has further elements
<code>E next()</code>	Return the next element in the iteration
<code>void remove()</code>	Remove the last element returned by the iterator

Using the standard idiom for explicit use of an iterator, the previous example would be written (for a collection of `String`, say):

```
for (Iterator<String> itr = coll.iterator() ; itr.hasNext() ; ) {  
    System.out.println(itr.next());  
}
```

This is awkward compared to the `foreach` statement, which we consider next, and so is less commonly used. It is necessary when you want to make a *structural* change to a collection—broadly speaking, adding or removing elements—in the course of iteration. (As we just saw, `Iterator` exposes only a method for removal of collection elements, but its subinterface `ListIterator`, available to `List` implementations, also provides methods to add and replace elements.) In other use cases, it may be necessary to control the iteration depending on the values of the elements encountered; [Example 10-2](#) provides an example.

Outside of these use cases, the `foreach` statement, which makes use of an `Iterator` internally, is more convenient. The `foreach` code corresponding to the current example is:

```
for (String s : coll) {  
    System.out.println(s);  
}
```

The target of a `foreach` statement can be an array or any class that implements the interface `Iterable`. Since the `Collection` interface extends `Iterable`, any set, list, or queue can be the target of `foreach`. If you write your own implementation of `Iterable`, that too can be used with `foreach`. For example, in the following code, a `Counter` object is initialized with a count of integers, and its iterator returns these in ascending order, starting from 1, in response to calls of `next`:

org/jgcbook/chapter09/A_iterable_and_iterators/Counter

```
class Counter implements Iterable<Integer> {  
    private final int count;  
    public Counter(int count) { this.count = count; }  
    public Iterator<Integer> iterator() {  
        return new Iterator<Integer>() {
```

```

        private int i = 0;
        public boolean hasNext() { return i < count; }
        public Integer next() { i++; return i; }
        public void remove() { throw new UnsupportedOperationException(); }
    };
}
}

```

Now, Counter objects can be the target of a *foreach* statement:

[org/jgcbook/chapter09/A_iterable_and_iterators/Counter](#)

```

int total = 0;
for (int i : new Counter(3)) {
    total += i;
}
assert total == 6;

```

In practice, it is unusual to implement `Iterable` directly in this way, as *foreach* is most commonly used with arrays and the standard collection classes. Direct use of iterators, rather than a *foreach* statement, is necessary mainly when you want to make a *structural* change to a collection—broadly speaking, adding, changing, or removing elements—in the course of iteration. As we just saw, the `Iterator` interface exposes only a method for removal of collection elements, but its subinterface `ListIterator`, available to `List` implementations, also provides methods to add and replace elements.

At one time or another, most people have made the mistake of trying, during iteration, to make a structural change directly—that is, rather than using one of the iterator methods. If the collection that you were iterating over was one of the general-purpose `Collection` implementations—`ArrayList`, `HashMap`, and so on—you may have been puzzled by seeing `ConcurrentModificationException` thrown from single-threaded code. The iterators of these collections throw this exception whenever they detect that the collection from which they were derived has been structurally modified—except by the iterator itself, of course. For example, the following code throws `ConcurrentModificationException`:

[org/jgcbook/chapter09/A_iterable_and_iterators/Program_1](#)

```

List<String> strings = new ArrayList<>(List.of("alpha", "bravo", "charlie"));
for (String s : strings) {
    if (!s.contains("r")) {
        strings.remove(s);      // throws ConcurrentModificationException
    }
}

```

The motivation for this behavior is that a structural change made during iteration is possible evidence that another thread is accessing the collection; structural changes that you wish the iterating thread to make should be made using the iterator. Allowing another thread to access a non-thread-safe collection will probably result

in a failure some time later, when it will be difficult to diagnose. To avoid this problem, the general-purpose Collections Framework iterators are *fail-fast*: their methods check for any structural modifications made since the last iterator method call, throwing `ConcurrentModificationException` if they detect one. The reasoning behind this policy is explored in “[Concurrent Modification](#)” on page 325.

A corrected version of the preceding code uses the iterator itself to remove the elements:

```
org/jcbook/chapter09/A\_iterable\_and\_iterators/Program\_2
```

```
List<String> strings = new ArrayList<>(List.of("alpha", "bravo", "charlie"));
for (Iterator<String> itr = strings.iterator() ; itr.hasNext() ; ) {
    String s = itr.next();
    if (! s.contains("r")) {
        itr.remove();
    }
}
assert strings.equals(List.of("bravo", "charlie"));
```

This strategy works only if the iterator exposes a method for making the appropriate structural modification. All iterators expose a `remove` method, but only list iterators, obtained using the method `listIterator`, can add or insert elements.

Stream Alternative

If memory constraints don’t prevent you from making a new copy of the list, streams offer a neater solution to this problem:

```
org/jcbook/chapter09/A\_iterable\_and\_iterators/Program\_3
```

```
List<String> strings = new ArrayList<>(List.of("alpha", "bravo", "charlie"));
List<String> modifiedStrings = strings.stream()
    .filter(s -> s.contains("r"))
    .toList();
assert modifiedStrings.equals(List.of("bravo", "charlie"));
```

The concurrent collections have other strategies for handling concurrent modification, such as weakly consistent iterators. We discuss them in more detail in “[Collections and Thread Safety](#)” on page 155.

Implementations

We have looked briefly at the interfaces of the Collections Framework, which define the behavior that we can expect of each collection. But there are several ways of implementing each of these interfaces. Why doesn’t the framework just use the best implementation for each interface? That would certainly make life simpler—too simple, in fact, to be anything like real life. If an implementation is a greyhound for

some operations, Murphy's law tells us that it will be a tortoise for others. Because there is no "best" implementation of any of the interfaces for every situation, you have to make a trade-off, judging which operations are used most frequently in your application and choosing the implementation that optimizes those operations.

The three main kinds of operations that most collection interfaces require are insertion and removal of elements by position, retrieval of elements by content, and iteration over the collection elements. The various implementations provide many variations on these operations, but the main differences among them can be discussed in terms of how they carry out these three. In this section, we'll briefly survey the four main structures used as the basis of the implementations; later, as we need them, we will look at each in more detail. The four structures are:

Arrays

These are the structures familiar from the Java language—and just about every other programming language since Fortran. Because arrays are implemented directly in hardware, they have the properties of random-access memory: very fast for accessing elements by position and for iterating over them, but slower for inserting and removing elements at arbitrary positions (because that may require adjusting the position of other elements). Arrays are used in the Collections Framework as the backing structure for `ArrayList`, `CopyOnWriteArrayList`, `EnumSet`, and `EnumMap`, and for many of the `Queue` and `Deque` implementations. They also form an important part of the mechanism for implementing hash tables (discussed shortly).

Linear linked lists

As the name implies, these consist of chains of linked cells. Each cell contains a reference to data and a reference to the next cell in the list (and, in some implementations, the previous cell). Linked lists perform quite differently from arrays: accessing elements by position is slow, because you have to follow the reference chain from the start of the list, but insertion and removal operations can be performed in constant time by rearranging the cell references. Linked lists are the primary backing structure used for the classes `ConcurrentLinkedQueue`, `LinkedBlockingQueue`, and `LinkedList`.

Other linked data structures

Linked structures are particularly suitable for representing nonlinear types like trees and skip lists (see "["ConcurrentSkipListSet" on page 203](#)"), especially if they need to be rearranged as new elements are added. Such structures provide an inexpensive way of maintaining sorted order in their data, allowing fast searching by content. Trees are the backing structures for `TreeSet` and `TreeMap`. Skip lists are used in `ConcurrentSkipListSet` and `ConcurrentSkipListMap`.

Hash tables

These provide a way of storing elements indexed on their content rather than on an integer-valued index, as with lists. In contrast to arrays and linked lists, hash tables provide no support for accessing elements by position, but access by content is usually very fast, as are insertion and removal. Hash tables are the backing structure for many `Set` and `Map` implementations, including `HashSet` and `LinkedHashSet` together with the corresponding maps `HashMap` and `LinkedHashMap`, as well as `WeakHashMap`, `IdentityHashMap`, and `ConcurrentHashMap`.

The content-based indexing of hashed collections depends on two `Object` methods, `hashCode` and `equals`, which are applied in succession to position an element for insertion or to locate it for retrieval. The requirements on the relationship between these methods are defined in the contract for `hashCode`. The crucial requirement, sometimes overlooked—with disastrous consequences—by Java programmers, is that if two objects are equal according to the `equals` method, then the result of calling `hashCode` on each must be the same. If `hashCode` depends on an instance field—or any other data about an object—that is not used by the `equals` method, then the first stage of a retrieval operation will very likely be misdirected. One way this can occur is if you do not override `Object::hashCode` at all; the value that it returns in this case will be implementation-dependent (in OpenJDK, it is usually randomly generated) but is in any case highly unlikely to be the same for two different instances.

The toy class `Person` is a case in point:

[org/jgcbook/chapter09/B_implementations/Person](#)

```
class Person {  
    private final String name;  
    public Person(String name) {  
        this.name = name;  
    }  
    public boolean equals(Object o) {  
        return o instanceof Person p && name.equals(p.name);  
    }  
}
```

This class should override `hashCode` so that it depends only on the same field that `equals` depends on (i.e., `name`). But it does not; as a result, hashed collections will not work correctly with it:

[org/jgcbook/chapter09/B_implementations/Person](#)

```
Set<Person> people = new HashSet<>();  
people.add(new Person("Alice"));  
assert ! people.contains(new Person("Alice"));
```

Views

In the Collections Framework, a *view* of a data structure provides a way of working with it as though it had been transformed in some way—into either a differently organized structure of the same type, or a structure of another type completely. The most obvious examples of this, in a general sense, are the backing structures discussed in the previous section. But even though replacing an element in an `ArrayList`, for instance, is implemented by replacing an element in the underlying array, we don't normally refer to an `ArrayList` as a view of an array. That's because `ArrayList` is more than just a view; it's capable of insulating its user from the limitations of arrays. For example, you will frequently want to add elements to a list. Arrays can't support that, but `ArrayList` hides that limitation, if necessary by transferring all its elements to a new and larger backing array. So an `ArrayList` is much more than a simple view of a single array, and that term isn't usually used to describe it.

Sometimes, however, a simple view is what is needed. For example, suppose we want to test for the presence of a particular object in an array. One obvious way to do this is to iterate over the array elements, testing each for equality with the search target. An alternative would be to use the `contains` method of the `List` interface to do that work instead. An array is not a `List`, though, so how can a `List` method be useful in handling it? We might well want to avoid the overhead of creating a new `ArrayList` object, physically copying all the elements of the array into a new collection. In this situation, a better answer is to get a `List` *view* of the array—an object that “looks like” a `List`, but implements all its operations directly on the underlying array. The method `asList` of the utility class `Arrays` provides such a view. The simple view that it returns supports some `List` operations, such as `contains`, and methods like `get` and `set` that access or replace the array elements, but it won't allow you to make *structural modifications*, like adding or removing elements, which aren't supported by the underlying array.

The data “of” the view actually resides in the underlying structure, so changes made to that structure are immediately visible in the view, and vice versa. For example, the following code compiles and runs without errors:

org/jgcbook/chapter09/C_views/Program_1

```
Integer[] arr = {1, 2, 3};
var list = Arrays.asList(arr);
list.set(0, 3);                                // change the list view...
assert arr[0] == 3;                            // and the underlying array changes
arr[2] = 0;                                    // now change the underlying array...
assert list.get(2) == 0;                          // and the list view changes
```

The Collections API exposes many methods returning views. For example, the keys of a Map can be viewed as a Set, as can its entries; collections can be viewed as unmodifiable, and so on. Each of these views has different rules dictating which modifications they will accept and reflect into the backing collection. In descending order of permissiveness, they may allow:

- All changes
- Some structural and all nonstructural modifications
- Only nonstructural modifications
- No modifications at all (fully unmodifiable)

So the interfaces that these views implement have some of their operations labeled *optional*. This is perhaps the most controversial aspect of the Java Collections Framework design. [Chapter 18](#) and the section “[Contracts](#)” on page 149 discuss this issue in detail; the views themselves will be discussed in subsequent chapters, each one in the context of its backing collection.

Views can generally be composed for read operations and are often commutative—that is, can be applied in any order. For example:

```
List<String> names = List.of("alpha", "bravo", "charlie", "delta");
List<String> reverseThenSublist = names.reversed().subList(1, 3);
List<String> subListThenReverse = names.subList(1, 3).reversed();
assert reverseThenSublist.equals(subListThenReverse);
```

Performance

The usefulness of a collections framework depends on the impact of the performance of its collections as part of a working system. Unfortunately, this is very difficult both to predict in theory and to assess in practice. Many factors contribute to it, including:

- How often any of the collection’s operations are executed
- Which operations are executed most frequently
- The time cost of each of the operations that are executed
- How many orphaned objects each produces, and what overhead is incurred in collecting them
- The locality properties (discussed in the following subsection) of the collection
- How much parallelism is involved, both at instruction and thread level

The study of how these factors combine to affect the speed of a real-life system belongs to the subject of performance tuning, the most important rule of which is often quoted in the form provided by Donald Knuth ([1974](#)): “Premature optimization is the root of all evil.” The explanation of this remark is twofold. First, for many

programs, performance is just not a critical issue. If a program is rarely executed or already uses few resources, optimization is a waste of effort, and indeed may well be harmful. Second, even for performance-critical programs, assessing *which* part is critical normally requires accurate measurement; in the same paper, Knuth added, “It is often a mistake to make *a priori* judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.”

This carries an important implication about comparing the performance of different collections. For example, `CopyOnWriteArrayList` provides highly efficient concurrent read operations at the cost of very expensive writes. So to use it in a system that requires highly performant concurrent access to a `List`, you have to have confidence, gained by measurement if necessary, that read operations greatly outnumber writes.

Memory

Traditionally, an algorithm was assessed on the basis of its use of two resources: time and space. The algorithms of the Collections Framework have generally good space performance, and this, together with the 50-year exponential decrease in the cost of memory from the 1970s onward, meant that for a long time space complexity got less attention than time complexity (see “[Instruction Count and the O-notation](#)” on page [145](#)). But modern trends in both software and hardware systems architecture have complicated the picture by reintroducing memory concerns as a major focus. The software concern is relatively straightforward: garbage collection can be an expensive overhead cost to the operation of allocation-intensive programs.

The effects of modern trends in hardware design require more explanation. For four decades, from the 1970s onward, processor speeds rose exponentially, far outstripping the speed increase of every other component—including memory and the memory bus, the components that together are responsible for keeping processors supplied with data and instructions to work on. The different grades of memory available conform to the same rule that governs storage technologies of all kinds, including on-board memory, flash memory, disk drives, and even tape (still in use for archival storage): the cost of storage is inversely correlated with the speed of the medium. The faster the medium can store and retrieve data, the greater the cost per byte of capacity. This leads designers to create *memory hierarchies*, with large amounts of cheap storage, like disk and even tape, at the bottom, and small amounts of expensive static RAM located on-chip, physically close to the processor, at the top. [Figure 9-1](#) shows a highly simplified view of the top part—the on-board and on-chip components—of the hierarchy. On-chip memory is organized in caches called Level 1, Level 2, and (sometimes) Level 3, each one slower and larger (and less power-hungry) than its predecessor.

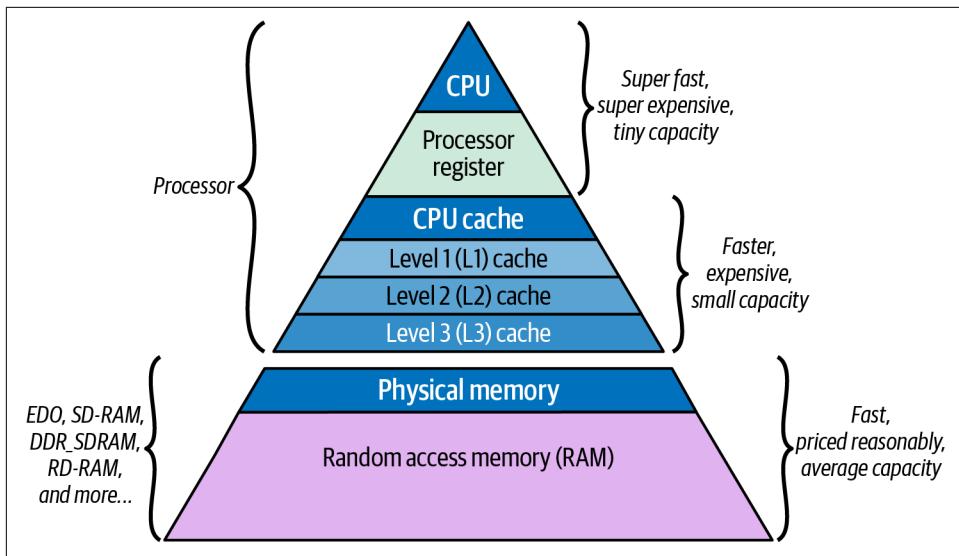


Figure 9-1. The memory hierarchy

If a data item is not available at one level, lower levels are searched until it is found. If the level is a cache, this is called a *cache miss*. Data in caches is organized in units called *cache lines*, commonly 64 bytes. If a failure to find one byte of data causes a cache miss, an entire cache line must be discarded in order to load that byte from a lower level in the hierarchy, along with the 63 bytes adjacent to it in memory. Such expensive cache misses will occur less often for programs that conform to the principle of *spatial locality*, which says that programs tend to reuse data and instructions near to those that they have used recently. Two programs, each of which execute the same number of instructions, may have very different execution times if one exhibits spatial locality and the other does not.

Object-oriented programs often exhibit poor spatial locality because objects can be located anywhere in memory, but some are worse than others. For example, a program iterating over an array of primitives will show good spatial locality and a predictable pattern of data access, enabling predictive retrieval of data leading to reduction of cache miss overheads. Iterating over an array of reference types will be worse, because although access to the array itself is predictable and localized, access to the referenced objects is not. A linked structure is worse still; memory access is in general not predictable or localized at all, so cache misses will typically be frequent and expensive. Every further level of indirection increases the likelihood of cache misses and, even more expensively, of page faults.

Instruction Count and the O-notation

Although, as we observed in the previous section, memory usage can no longer be separated cleanly from execution time, there is still merit in trying to make a separate estimate of the latter. Traditionally, execution time was assumed to be proportional to the count of CPU operations needed to complete a program. This assumption is also subject to some qualifications, which we'll discuss at the end of this section. First, though, how is the execution count estimated? Detailed analysis can be complex. A relatively simple example is provided in Donald Knuth's (1998) classic book *Sorting and Searching*, where the worst-case execution count for a multiple list insertion sort program on Knuth's notional MIX machine is derived as:

$$3.5N^2 + 24.5N + 4M + 2$$

where N is the number of elements being sorted and M is the number of lists.

As a shorthand way of describing algorithm efficiency, this is hardly convenient. Clearly, we need a broader brush for general use. The one most commonly used is the *O-notation* (pronounced, and often written, as “big-O notation”). The O-notation is a way of describing the performance of an algorithm in an abstract way, without the detail required to predict the precise performance of a particular program running on a particular machine. The main reason for using it is that it gives us a way of describing how the execution time for an algorithm depends on the size of its data set, provided the data set is large enough. For example, in the previous expression, the first two terms are comparable for low values of N ; in fact, for $N < 7$, the second term is larger. But as N grows, the first term increasingly dominates the expression, and, by the time it reaches 100, the first term is 15 times as large as the second one. Using a very broad brush, we say that the worst case for this algorithm takes time $O(N^2)$. We don't care too much about the coefficient because that doesn't make any difference to the single most important question we want to ask about any algorithm: What happens to the execution count when the data size increases—say, when it doubles? For the worst-case insertion sort mentioned previously, the answer is that the execution count goes up fourfold. That makes $O(N^2)$ pretty bad—worse than anything we will meet in practical use in this book.

Table 9-1 shows some commonly found execution counts, together with examples of algorithms to which they apply. Many other execution counts are possible, including some that are much worse than those in **Table 9-1**. A large class of important problems can be solved only by algorithms that take $O(2^N)$ steps—for these, when N doubles, the execution count is squared! For all but the smallest data sets, such algorithms are infeasibly slow, although quantum computers will change that situation for some important problems.

Table 9-1. Some common big-O classes

Time	Common name	Effect on the execution count if N is doubled	Example algorithms
$O(1)$	Constant	Unchanged	Insertion into a hash table (see “ Set Implementations ” on page 183)
$O(\log N)$	Logarithmic	Increased by a constant amount	Insertion into a tree (see “ TreeSet ” on page 200)
$O(N)$	Linear	Doubled	Linear search
$O(N \log N)$		Doubled plus an amount proportional to N	Merge sort (see “ Changing the Order of List Elements ” on page 276)
$O(N^2)$	Quadratic	Increased fourfold	Insertion sort worst case (input in reverse order)

Sometimes we have to think about situations in which the cost of an operation varies with the state of the data structure. For example, adding an element to the end of an `ArrayList` can normally be done in constant time, unless the backing array has reached its capacity. In that case, a new and larger array must be allocated, and the contents of the old array must be transferred into it. The cost of this operation is linear in the number of elements in the array, but it happens relatively rarely. In situations like this, we calculate the *amortized cost* of the operation—that is, the total cost of performing it N times divided by N , taken to the limit as N becomes arbitrarily large. In the case of adding an element to an `ArrayList`, the total cost for N elements is $O(N)$, so the amortized cost is $O(1)$.

Big-O analysis of execution counts is often very useful for broad-brush comparisons of the running times of programs and, for that reason, each interface chapter of this book includes a big-O analysis of common operations for each implementation. All the same, there are reasons for caution in treating O-numbers as a proxy for running times. They include:

- The assumption in determining the dominant term is that the data size will always be sufficiently large enough to outweigh constant factors and multipliers; for example, $O(N + c)$ is taken to be $O(N)$ for constant c , on the assumption that in situations where it matters, $N \gg c$ (that is, N will be much bigger than c). This may not be true for very large factors.
- Machine instructions do not all have the same execution time. For example, inserting an element at the first position of an `ArrayList` has complexity $O(N)$, because every element must be moved to a position one place higher in the array. But if it is possible for that operation to be executed with a single bulk data memory-to-memory operation implemented in hardware, then the actual execution time may compare favorably with the same operation on a `LinkedList`, even though there it has complexity $O(1)$.

- Parallelism complicates the picture further. With instruction-level parallelism, instructions are processed in stages, with the different stages of successive instructions being executed simultaneously. So the elapsed execution time of a program is normally much less than the summed execution times of its individual instructions. The same applies, on a larger scale, to thread-level parallelism, in which different threads can process their tasks simultaneously except when they are in contention for some resource.
- The time complexity of single operations is not important in itself; what matters is the time required to execute an entire use case. Consider, for example, iterating an entire data structure and replacing all of its elements (if that is the usage scenario for your application) rather than simply advancing an iterator by one element. Calculating the time required for such a use case requires you to take into account factors such as the cost of cache misses and the amortized cost of garbage collection, which depend on characteristics of a data structure that are not captured by the time complexity of single operations. This point is explored further in [Table 14-2](#).

In conclusion, it is worth repeating the value of theoretical performance analysis, including O -numbers: it can provide valuable guidance at the design stage. You would never choose an $O(N^2)$ algorithm over an $O(N)$ one for a large data set, for example. But if you have a running system with a performance problem, there is no substitute for accurate measurement to compare different candidate implementations.

Immutability and Unmodifiability

Functional programming is an attractive style; at their best, functional programs are elegant and demonstrably sound—much more so than object-oriented programs. For this reason, some of the features of functional languages that provide these advantages have been gradually adopted into Java, starting with generics, continuing with streams and lambdas (see “[Lambdas and Streams](#)” on page 152), and culminating with pattern matching for `switch`, `instanceof`, and records, at the time of this writing still being delivered in [Project Amber](#). One important feature of the functional style is that its data structures are *immutable*—that is, their state cannot be modified after their creation. Immutability confers a number of advantages on a program:

- Immutable objects are thread-safe.
- Immutable objects are perfectly encapsulated, thus removing the need for *defensive copying* (the technique by which objects guarantee the integrity of their data by only ever exposing copies of it; see “[Respect the ‘Ownership’ of Collections](#)” on page 289).

- Immutability guarantees stable lookup in keyed and ordered collections (see “Contracts” on page 149).
- Immutability reduces the number of states a program can be in, making it simpler, clearer, and easier to understand and reason about.

Realizing these advantages in a Java program is difficult, however. For an object to be truly immutable, its entire object graph—that is, the object itself and everything it refers to, directly or indirectly—must not observably change after construction. Obviously, immutable graph components like wrapper objects and strings present no problem, but for mutable components the often difficult requirement is that the graph must have guaranteed exclusive access to them.

This has led to alternative conflicting terminologies for describing immutability of collections:

- Frameworks like [Guava](#) and [Eclipse Collections](#) refer to immutability of an entire object graph as *deep immutability*. They refer to a collection that refuses modification at the first level—that is, an attempt to add, remove, or replace an element—as *shallow immutability*, or often just *immutability*.
- The Java Collections Framework documentation uses *immutability* to mean immutability of the entire object graph. What the other frameworks refer to as “shallow immutability,” the Java documentation—and this book—calls unmodifiability.

Unmodifiability is a kind of partial immutability that doesn’t fully confer any of the advantages of immutability just discussed, so you might well question its usefulness. But it does confer real advantages:

- Collections of immutable objects, including wrapper objects and strings, are not uncommon, and unmodifiable collections of these provide the full benefits of immutability.
- Even partial immutability reduces the number of program states that you have to consider when understanding a program and reasoning about its correctness.
- Because unmodifiable sets and maps can be backed by arrays instead of hashed structures, they can provide very significant space savings.

A further advantage of unmodifiability is that in principle (and in other frameworks, though not the Java Collections Framework), it can be reflected in the type system. By contrast, immutability of an object graph requires it to have exclusive access to every mutable component. This property cannot be enforced in general at compile time, not even by static analysis, let alone by typing constraints. True, an object graph consisting entirely of immutable objects evades this objection, but in general collections are backed by arrays, which in Java are always mutable.

[“Fundamental Issues in the Collections Framework Design” on page 311](#) discusses this topic further, and [Chapter 17](#) explores the practical situations in which you would choose to use unmodifiable collections.

Contracts

In reading about software design, you are likely to come across the term *contract*, often without any accompanying explanation. In fact, software engineering gives this term a meaning that is very close to what people usually understand a contract to be. In everyday usage, a contract defines what two parties can expect of each other: their obligations to each other in some transaction. Obviously, a contract must specify the obligations of a service supplier to a client. But the client, too, may have obligations—including, but not only, the obligation to pay—and failing to meet them will automatically release the supplier from their obligations as well. For example, airlines’ conditions of carriage—for the class of tickets that your authors can afford, anyway—release them from the obligation to carry passengers who have failed to turn up on time. This allows the airlines to plan their service on the assumption that all the passengers they are carrying are punctual; they do not have to incur extra work to accommodate clients who have not fulfilled their side of the contract.

Contracts work the same way in software. If the contract for a method lays down preconditions on its arguments (i.e., the obligations that a client must fulfill), the method is required to return its contracted results only when those preconditions are fulfilled. For example, binary search (see [“Finding Specific Values in a List” on page 278](#)) is a fast algorithm to find an element, but only within an ordered list, and it is entitled to fail if you apply it to an unordered list. So the contract for `Collections::binarySearch` can say “if the list is unsorted, the results are unspecified,” and the implementer of binary search is free to write code that, given an unordered list, returns any result at all. Where possible, preconditions are checked and an exception thrown if they are violated; this is a good approach for general libraries such as the Collections Framework, which are going to be heavily used in widely varying situations by programmers of widely varying ability. APIs of less general libraries usually avoid it, because it restricts the flexibility of the library unnecessarily. And it is not always feasible: checking the precondition for binary search would be $O(N)$, more expensive than the search itself. In such cases, being prepared to return the wrong result is the best alternative that falls within the terms of the contract.

In summary, all that a client should need to know is how to uphold their side of the contract; if they fail to do that, the library is released from all guarantees of correctness.

It's good practice in Java to code to an interface rather than to a particular implementation, to provide maximum flexibility in choosing implementations. This idea is explored in detail in “[Balance Client and Library Interests in API Design](#)” on page [295](#). For that to work, what does it imply about the behavior of implementations? If your client code uses methods of the `List` interface, for example, and at run time the object doing the work is actually an `ArrayList`, you would like to know that the assumptions you have made about how `Lists` behave are true for `ArrayLists` also. So a class implementing an interface usually has to fulfill all the obligations laid down by the terms of the interface contract. A weaker form of these obligations is already imposed by the compiler: a class claiming to implement an interface must provide concrete method definitions matching the declarations in the interface, but contracts take this further by specifying the behavior of these methods as well.

The obligations imposed by the Collections Framework interfaces are unusual in some respects, however. A guiding principle in the design of the framework was that it should be simple—a vital characteristic in a library that every Java programmer must learn. Frameworks that separate modifiable from unmodifiable interfaces have many more types than the Java Collections Framework, so the framework designers decided to avoid this separation. But since many collection views (and, since Java 9, unmodifiable collections also) do not support all write operations, the interface contracts label these operations *optional*. For example, the `Set` view of a `Map`'s keys can have elements removed but not added (see [Chapter 15](#)), while other view collections can have elements neither added nor removed (e.g., the list view returned by `Arrays::asList`), or support no modification operations at all, like collections that have been wrapped in an unmodifiable wrapper (see “[Unmodifiable Collections](#)” on page [281](#)). The contracts of the Collection Framework interfaces provide for implementations to throw `UnsupportedOperationException` when optional methods are called. We explore this design decision in detail in “[Fundamental Issues in the Collections Framework Design](#)” on page [311](#).

Although until now we have only discussed functional requirements, contracts can also require performance guarantees. To fully understand the performance characteristics of a class, however, you often need to know about the algorithms of the implementation in detail. In the following chapters, when we discuss the different JDK implementations of the framework interfaces, we also provide some detail about their algorithms where that might be useful. This can help in deciding between implementations, but remember that it is not stable; while contracts are binding, one of the main advantages of using them is that they allow implementations to change as better algorithms are discovered or as hardware improvements change their relative

merits.¹ And of course, if you are using another implementation, algorithm details not governed by the contract may be entirely different.

Content-Based Organization

“Sequenced Collections” on page 132 introduced the idea of internally ordered sequences, in which the elements are automatically positioned according to the order of their contents as defined by some ordering relation (see Chapter 3). These are not the only collection classes to organize their elements automatically according to their contents: hashed collections, priority queues, and delay queues have the same property. In the cases of the internally ordered sequences and queues, the content-based organization is observable via the operations of the collection; not so in the case of the hashed collections, but their organization is used when elements need to be located. What all these collections have in common is an implicit invariant: for them to operate correctly, the value of the fields used to position the elements must not change once they have been inserted into the collection. For example, consider a different version of the class `Person`, this time defined with a `hashCode` method, but one that depends on the value of the mutable field `name`:

[org/jgcbook/chapter09/F_contracts/Person](#)

```
class Person {  
    private String name;  
    public Person(String name) {  
        this.name = name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int hashCode() {  
        return name.hashCode();  
    }  
    public boolean equals(Object o) {  
        return o instanceof Person p && name.equals(p.name);  
    }  
}
```

Now an object of type `Person` can be reliably retrieved, but only if the value of `name` is unchanged. If that field is modified, the collection can no longer match the object with either the old or the new value:

[org/jgcbook/chapter09/F_contracts/Person](#)

```
Set<Person> people = new HashSet<>();  
Person alice = new Person("Alice");
```

¹ A startling example of this occurred as this book was going to press, when it was announced that an undergraduate student of computing science had discovered a way to radically speed up certain hash table operations, overturning decades of accepted wisdom (Farach-Colton et al. 2025).

```
people.add(alice);
alice.setName("Bob");
assert ! people.contains(new Person("Alice"));
assert ! people.contains(new Person("Bob"));
```

To avoid problems like this, the best guideline is this: whenever you are storing objects in a `Set`, a `Map`, or an internally ordered `Queue`, ensure that the fields used by the collection to organize its contents are immutable.

Lambdas and Streams

In “[Immutability and Unmodifiability](#)” on page 147, we saw that the advantages of the functional programming style have led object-oriented languages toward adoption of many functional language features. Following the introduction of generics in Java 5, the next big innovation came in Java 8, with the introduction of lambdas and streams. Lambdas introduced a way in which functions could be represented in Java programs. For example, the statement:

```
Function<Integer, String> intToString = i -> i.toString();
```

declares a function `intToString`, which takes an argument of type `Integer` and returns its string representation.

The importance of lambdas for collections and collection processing comes from their use in stream operations. Streams are a mechanism for transporting a sequence of values from a source to a destination through a series of operations, typically implemented as lambdas, each of which can transform, drop, or insert values on the way. This provides an alternative model to the traditional use of collections for aggregate data processing. A simple, if somewhat contrived, example (from *Mastering Lambdas* [Naftalin 2014]) illustrates the change in thinking. The record `Point` represents a point defined by its `x` and `y` coordinates, with a single method `distanceFrom` that calculates its distance from another point:

[org/jcbook/chapter09/G_lambdas_and_streams/Point](#)

```
record Point(int x, int y) {
    public double distanceFrom(Point p) { ... }
}
```

Without streams, a common model of bulk data processing is to process collections in a series of stages: a collection is iteratively processed to produce a new collection, which in turn is iteratively processed, and so on. The following code starts with a collection of `Integer` instances, applies an arbitrary transformation to produce a set of `Point` instances, and finally finds the maximum among the distances of each `Point` from the origin:

```
Point origin = new Point(0, 0);
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
List<Point> pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i % 3, i / 3));
}
double maxDistance = 0;
for (Point p : pointList) {
    maxDistance = Math.max(p.distanceFrom(origin), maxDistance);
}
```

The real code for which this is a model has several disadvantages: it is very verbose; the intermediate collection `pointList` is an overhead on the operation of the program, resulting in increased garbage collection costs or even in heap space exhaustion; there is an implicit assumption, difficult to spot, that the minimum value of an empty list is 0; and, perhaps worst of all, the intent of the program is hard to discern, because the crucial operations are interspersed with the code for collection handling.

Here is the equivalent code, expressed as a pipeline processing the original collection `intList` through a series of transformations:

```
Point origin = new Point(0, 0);
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
OptionalDouble maxDistance = intList.stream()
    .map(i -> new Point(i % 3, i / 3))
    .mapToDouble(p -> p.distanceFrom(origin))
    .max();
```

This example shows in miniature the advantages of stream code: it is more concise and readable, often uses less intermediate storage, handles an empty source gracefully, and can never attempt to mutate the source collection.

From here on in this book, where example collection code can benefit from partial conversion to the use of streams, we will show stream code as an alternative in a sidebar, in the hope that there will be benefit in seeing the two idioms compared.

Parallel Streams

A major design aim of the Stream API was to enable suitable workloads to be distributed over multiple cores without great effort on the part of application programmers. This is achieved by use of *recursive decomposition*, in which a data set is broken down into parts for separate processing, with the results of the constituent processes being combined on completion.

For example, given a four-core processor and an array-based list of N elements, a program might define a `solve` algorithm to break the task down for parallel execution in the following way, using a highly simplified pseudocode description:

```
if the task list contains more than N/4 elements {
    leftTask = task.getLeftHalf()
    rightTask = task.getRightHalf()
    doInParallel {
        leftResult = leftTask.solve()
        rightResult = rightTask.solve()
    }
} else {
    result = combine(leftResult, rightResult)
    result = task.solveSequentially()
}
```

Implementing recursive decomposition requires knowing how to split tasks in this way and how eventually to execute sufficiently small ones without further splitting. The strategy for these operations depends on the source of the data: accordingly, every implementation of `Iterable`—in other words, every `Collection` subtype—has an associated `Spliterator` object, which contains the appropriate strategies for that collection.

In this case, splitting an array-based list has an obvious implementation: divide the array in two, divide each of the resulting parts in two, and so on until further division would result in more parts than there are cores on which to process them, or until the overhead of splitting would outweigh the gain from parallelizing the processing.

The stream framework hides the complexity of programming recursive decomposition in the convenient abstraction of *parallel streams*. If a stream is created parallel or made parallel by a call of the `parallel` method, the framework applies recursive decomposition to the data set being streamed and assigns each of the resulting parts to its own thread. For our current example, the only change necessary to the client code is emphasized here:

```
OptionalDouble maxDistance =
    intList.stream()
        .parallelStream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p -> p.distance(0, 0))
        .max();
```

Each thread is, ideally, assigned a core on which it can execute without interruption. In the best case—that of a large data set requiring compute-intensive processing with very little I/O—this can result in near-perfect parallelization.

For more detail on parallel streams, see [Naftalin \(2014\)](#).

Collections and Thread Safety

When a Java program is running, it is executing one or more execution sequences, or *threads*. A thread is like a lightweight process, so a program simultaneously executing several threads can be thought of as a computer running several programs simultaneously, but with one important difference: different threads can simultaneously access the same memory locations and other system resources. On machines with multiple processors, truly concurrent thread execution can be achieved by assigning a processor to each thread. If, however, there are more threads than processors—the usual case—multithreading is implemented by *time slicing*, in which a processor executes some instructions from each thread in turn before switching to the next one.

There are two good reasons for using multithreaded programs. An obvious one, in the case of multicore and multiprocessor machines, is to share the work and get it done quicker. (This reason is becoming ever more compelling as hardware designers turn increasingly to parallelism as the way of improving overall performance.) A second one is that two operations may take varying, perhaps unknown, amounts of time, and you do not want the response to one operation to await the completion of the other. This is particularly true for a graphical user interface (GUI), where the response to the user clicking a button should be immediate and should not be delayed if, say, the program happens to be running a compute-intensive part of the application at the time.

Although concurrency may be essential to achieving good performance, it comes at a price. Different threads simultaneously accessing the same memory location can produce unexpected results, unless you take care to constrain their access. Consider [Example 9-1](#), in which the class `ArrayStack` uses an array and an index to implement the interface `Stack`, which models a stack of `int`. For `ArrayStack` to work correctly, the variable `index` should always point one position beyond the top element of the stack, no matter how many elements are added to or removed from the stack. This is an *invariant* of the class. Now think about what can happen if two threads simultaneously attempt to push an element onto the stack. As part of the `push` method, each will execute the emphasized lines, which are correct in a single-threaded environment but in a multithreaded environment may break the invariant.

Example 9-1. A non-thread-safe stack implementation

org/jgcbook/chapter09/H_collections_and_thread_safety/ArrayStack

```
interface Stack {  
    public void push(int elt);  
    public int pop();  
    public boolean isEmpty();  
}  
  
public class ArrayStack implements Stack {
```

```

private final int MAX_ELEMENTS = 10;
private int[] stack;
private int index;
public ArrayStack() {
    stack = new int[MAX_ELEMENTS];
    index = 0;
}
public void push(int elt) {
    if (index != stack.length) {
        stack[index] = elt;
        index++;
    } else {
        throw new IllegalStateException("stack overflow");
    }
}
public int pop() {
    if (index != 0) {
        return stack[--index];
    } else {
        throw new IllegalStateException("stack underflow");
    }
}
public boolean isEmpty() { return index == 0; }
}

```

For one possible way (out of many) in which this code could fail, consider this scenario, illustrated in [Figure 9-2](#): thread A executes the first emphasized line, thread B executes the first emphasized line and then the second, and finally thread A executes the second emphasized line. Now only the value added by thread B will be on the stack, because that will have overwritten the value added by thread A. The variable `index`, on the other hand, may² have been incremented by two, in which case it is no longer pointing one position beyond the top element of the stack as it should do. A program execution like this is said to be in a *race condition*, because its result depends on the relative speed of multiple threads. If, as in this case, the execution breaks the invariant, it will leave the program in an inconsistent state, very likely to fail because other program parts will depend on the invariant being true.

Thread A		stack[0] = 1						index++	
Thread B				stack[0] = 2		index++			
Value of index	0		0		0		1		2

Time →

Figure 9-2. A race condition

² The effect on `index` of this unsynchronized code is also not deterministic.

The increasing importance of concurrent programming during the lifetime of Java has led to a corresponding emphasis in the Collections Framework on flexible and efficient concurrency policies. As a user of the Java platform collections, you need a basic understanding of the concurrency policies of the different collections in order to know how to choose between them and how to use them appropriately. In this section, we'll briefly outline the different ways in which the collections handle concurrency, and the implications for the programmer. For a full treatment of the general theory of concurrent programming, see *Concurrent Programming in Java* (Lea 1999), and for more detail about concurrency in Java and the collections implementations, see *Java Concurrency in Practice* (Goetz et al. 2006).

Synchronization and the Legacy Collections

Code like that in `ArrayStack` is not thread-safe—it works when executed by a single thread, but may break in a multithreaded environment. Since the incorrect behavior we observed in the previous example involved two threads simultaneously executing the `push` method, we could change the program to make that impossible. Using `synchronized` to modify the declaration of the `push` method will guarantee that once a thread has started to execute it, all other threads are prevented from executing that method until the first thread is done:

```
public synchronized void push(int elt) { ... }
```

This is called *synchronizing* on a *critical section* of code—in this case, the whole of the `push` method. Before a thread can execute synchronized code, it has to get the *lock* on some object, called the *monitor*, which for methods is by default (as in this case) the current object. While a lock is held by one thread, another thread that tries to enter any critical section synchronized on that lock will *block*—that is, will be suspended—until it can obtain the lock. This synchronized version of `push` is thread-safe; in a multithreaded environment, each thread behaves consistently with its behavior in a single-threaded environment.

To safeguard the invariant and make `ArrayStack` as a whole thread-safe, the methods `pop` and `isEmpty` must also be synchronized on the same object. The method `isEmpty` doesn't write to shared data, so synchronizing it isn't required to prevent a race condition, but for a different reason. Each thread may use a separate memory cache, which means that writes by one thread may not immediately be seen by another one. This behavior is allowed by the Java Memory Model (JMM) in order to maximize parallelism. The JMM, which forms part of the Java Language Specification (Gosling et al. 2023, §17.4), specifies how to ensure data coherence between threads: either by use of the `volatile` keyword, or by ensuring that reads take place only after both reader and writer have synchronized on the same monitor.

Full method synchronization was, in fact, the policy of the collection classes provided in JDK 1.0: for `Vector`, `Hashtable`, and their subclasses, all methods that access

their instance data are synchronized. These are now regarded as legacy classes to be avoided because of the high price this policy imposes on all clients of these classes, whether they require thread safety or not.³ Synchronization can be very expensive: forcing threads to queue up to enter the critical section one at a time slows down the overall execution of the program, and the overhead of administering locks can be very high if they are often contended. The JDK 1.0 concurrency model also suffers from the same problem as synchronized wrappers, discussed briefly in the next section and at more length in “[Avoid Synchronized Wrapper Collections](#)” on page 302: they require the synchronization of multiple method calls by the caller in order to obtain consistent behavior.

Synchronized Collections and Fail-Fast Iterators

The performance cost of internal synchronization in the JDK 1.0 collections led the designers to avoid it when the Collections Framework was first introduced in JDK 1.2. Instead, the platform implementations of the interfaces `List`, `Set`, and `Map` widened the programmer’s choice of concurrency policies. To provide maximum performance for single-threaded execution, the new collections provided no concurrency control at all. (Similar design changes were made for other synchronized classes—for example, the synchronized class `StringBuffer` was complemented in Java 5 by its unsynchronized equivalent, `StringBuilder`.)

Along with this change came a new concurrency policy for collection iterators. In multithreaded environments, a thread that has obtained an iterator will usually continue to use it while other threads modify the original collection. So iterator behavior has to be considered as an integral part of a collection’s concurrency policy. The policy of the iterators for Java 2 collections is to *fail fast*, as described in “[Iterable and Iterators](#)” on page 135: every time they access the backing collection, they check it for structural modification (which, in general, means that elements have been added to or removed from the collection) by comparing the collection’s modification counter with their own internal copy. A discrepancy implies that the collection has been modified, but not by one of the iterator’s own methods; in that case, the iterator fails immediately, throwing `ConcurrentModificationException` rather than continuing to attempt to iterate over the modified collection with unpredictable results. Note that this fail-fast behavior is provided to help find and diagnose bugs; it is not guaranteed as part of the collection contract.

The appearance of Java collections without compulsory synchronization was a welcome development. However, thread-safe collections were still required in many situations, so the framework provided an option to use the new collections with the old

³ An exception is `java.util.Properties`, which, although a subclass of the legacy class `Hashtable`, is used widely enough to have been worth modifying to use a `ConcurrentHashMap` internally.

concurrency policy, by means of synchronized wrappers (see [Chapter 16](#)). These are created by calling one of the factory methods in the `Collections` class, supplying an unsynchronized collection that it will encapsulate. We'll explore them in greater detail in "[Avoid Synchronized Wrapper Collections](#)" on page 302, but a short summary of that section may be useful here. It concludes that their usefulness is severely limited: since the thread safety they offer is confined to individual operations, clients need to implement their own locking policy, synchronizing multiple method calls, in order to obtain consistent behavior. This limits their usefulness as thread-safe collections and degrades their performance, since client locks must be held over multiple method calls.

Concurrent Collections

Java 5 introduced thread-safe concurrent collections as part of a much larger set of concurrency utilities, including primitives—atomic variables and locks—which give the Java programmer access to relatively recent hardware innovations for managing concurrent threads (notably compare-and-set operations, explained in the following subsection). Atomic variables get their name from the *atomic operations* they expose—operations that appear, from the perspective of all other threads, to be executed in a single step. For example, the `AtomicInteger` method `incrementAndGet` increments the object's value and returns the new value without any possibility of interference from other threads, as described in "[Collections and Thread Safety](#)" on page 155.

The concurrent collections remove the necessity for client-side locking as described in the previous section—in fact, external synchronization is not even possible with these collections, as there is no one object that when locked will block all methods. Where operations need to be atomic—for example, inserting an element into a `Map` only if it is currently absent—the concurrent collections provide a method specified to perform atomically; in this case, `ConcurrentMap.putIfAbsent`.

If you need thread safety, the concurrent collections generally provide much better performance than synchronized collections. This is primarily because their throughput is not reduced by the need to serialize access, as is the case with the synchronized collections. Synchronized collections also suffer the overhead of managing locks, which can be high if there is much contention. These differences can lead to efficiency differences of two orders of magnitude for concurrent access by more than a few threads.

Mechanisms of concurrent collections

The concurrent collections achieve thread safety by several different mechanisms. The first of these—the only one that does not use the new primitives—is *copy-on-write*. Classes that use copy-on-write store their values in an internal array, which is effectively immutable; any change to the value of the collection results in a new array

being created to represent the new values. Synchronization is used by these classes, though only briefly, during the creation of a new array. Because read operations do not need to be synchronized, copy-on-write collections perform well in the situations for which they are designed—those in which reads greatly predominate over writes. The collection classes `CopyOnWriteArrayList` and `CopyOnWriteArraySet` use this mechanism.

A second group of thread-safe collections relies on hardware operations generically known as *compare-and-set* (CAS), which provide a fundamental improvement on traditional synchronization. To see how CAS works, consider a computation in which the value of a single variable is used as input to a long-running calculation whose eventual result is used to update the variable. Traditional synchronization makes the whole computation atomic, excluding any other thread from concurrently accessing the variable. This reduces opportunities for parallel execution and therefore throughput. An algorithm based on CAS behaves differently: a thread makes a local copy of the variable and performs the calculation without getting exclusive access. Only when it is ready to update the variable does it call CAS, which in one atomic operation compares the variable's value with its value at the start and, if they are the same, updates it with the new value. If they are not the same, the variable must have been modified by another thread; in this situation, the CAS thread can try the whole computation again using the new value, or give up, or—in some algorithms—continue, because the interference will have actually done its work for it! This style of locking is called *optimistic* concurrency control, in contrast to the traditional pessimistic style, which blocks all other threads from access to the variable while this thread's operation is in progress. Collections using CAS include `ConcurrentLinkedQueue` and `ConcurrentSkipListMap`.

The third group uses implementations of `java.util.concurrent.locks.Lock`, an interface introduced in Java 5 as a more flexible alternative to classical synchronization. A `Lock` has the same basic behavior as classical synchronization, but a thread can also acquire it under special conditions: only if the lock is not currently held, or if it becomes free only within a set timeout period, or if the thread is not interrupted. Unlike synchronized code, in which an object lock is held while a code block or a method is executed, a `Lock` is held until its `unlock` method is called (making it possible for a `Lock` to be released in a different block or method from that in which it was acquired). Some of the collection classes in this group make use of these features to divide the collection into parts that can be separately locked, giving improved concurrency. For example, `LinkedBlockingQueue` has separate locks for the head and tail ends of the queue so that elements can be added and removed in parallel. Other collections using these locks include most of the implementations of `BlockingQueue`.

Iterators

The mechanisms just described lead to iterator policies more suitable for concurrent use than fail-fast, which implicitly regards concurrent modification as a problem to be eliminated. Copy-on-write collections have *snapshot iterators*. These collections are backed by arrays that, once created, are never changed; if a value in the collection needs to be changed, a new array is created. This means an iterator can read the values in one of these arrays (but never modify them) without danger of them being changed by another thread. Snapshot iterators do not throw `ConcurrentModificationException`.

Collections that rely on CAS have *weakly consistent* iterators, which reflect some but not necessarily all of the changes that have been made to their backing collections since they were created. For example, if elements in the collection have been modified or removed before the iterator reaches them, it definitely will reflect these changes, but no such guarantee is made for insertions. The third group described in the previous subsection also have weakly consistent iterators. Weakly consistent iterators do not throw `ConcurrentModificationException`.

Conclusion

This chapter was preparation for an in-depth exploration of the Collections Framework. To get the best out of the framework, it is very helpful to understand the software engineering ideas that shaped its design, as well as the changes in hardware that continue to influence its evolution.

We begin in [Chapter 10](#) with a survey of the most important interface of the Collections Framework: `Collection` itself, the supertype of all collections other than maps.

The Collection Interface

The interface `Collection<E>` is at the root of the type hierarchy of the Collections Framework. It exposes the core functionality that we expect of any collection other than a map.

The Methods of Collection

The methods defined by `Collection` can be divided into four groups, for adding elements, removing them, querying them, and making them available for further processing.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter10](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter10)

Adding Elements

```
boolean add(E e)                                add the element e  
boolean addAll(Collection<? extends E> c)    add the contents of c
```

The contracts for these methods specify that after their execution, the collection must contain the element or elements supplied in the argument. The return value indicates whether the method call has changed the contents of the collection. So for example, a call attempting to add an element to a set that already contains it will return normally, with the value `false`. If the collection did not contain the element or elements, and the method was unable to add them, it must throw an exception. This can happen

because the collection may not support these operations, or because the method is attempting to add an element that is unacceptable to the collection—for example, it may be attempting to add `null` to a `null`-hostile collection.

To maintain the type uniformity of the collection, these methods only allow you to add elements, or element collections, of the parametric type. This is in contrast to the methods of the next group.

Removing Elements

<code>void clear()</code>	removes all elements
<code>boolean remove(Object o)</code>	remove an element <code>o</code>
<code>boolean removeAll(Collection<?> c)</code>	remove all occurrences of the elements in <code>c</code>
<code>boolean retainAll(Collection<?> c)</code>	remove the elements <i>not</i> in <code>c</code>
<code>boolean removeIf(Predicate<? super E> p)</code>	remove the elements for which <code>p</code> is true

Apart from `clear`, which empties the collection, the methods in this group, as in the first one, return a `boolean` result indicating whether the collection was changed by their action.

If the argument `o` to `remove` is `null`, the method will remove a single `null` from the collection if one is present. Otherwise, if any elements `e` are present for which `o.equals(e)`, it removes one; if not, it leaves the collection unchanged and returns `false`. The behavior of removing a single element is in contrast to `removeAll`, which removes all occurrences in this collection of every matching element in the supplied collection.

In contrast to the methods for adding elements, these methods—and those of the next group—will accept as arguments elements or element collections of any type. “Object Versus E” on page 324 discusses this design choice in some detail. However, they are like the methods for adding elements in that it is optional for collection classes to implement them. This issue is discussed in detail in “Fundamental Issues in the Collections Framework Design” on page 311.

Querying the Contents of a Collection

<code>boolean contains(Object o)</code>	returns <code>true</code> if <code>o</code> is present
<code>boolean containsAll(Collection<?> c)</code>	returns <code>true</code> if all elements of <code>c</code> are present in the collection
<code>boolean isEmpty()</code>	returns <code>true</code> if no elements are present
<code>int size()</code>	returns the element count (or <code>Integer.MAX_VALUE</code> if that is less)

The behavior of these methods requires little explanation, besides the return value of `size` for extremely large collections. The value `Integer.MAX_VALUE` was probably chosen on the assumption that collections this large—with more than two billion elements—will rarely occur. Even so, an alternative design, throwing an exception instead of returning an arbitrary value, would have the advantage of ensuring that the contract for `size` could clearly state that if it does succeed in returning a value, that value will be correct.

Making a Collection's Contents Available for Further Processing

<code>Iterator<E> iterator()</code>	return an <code>Iterator</code> over the elements
<code>Spliterator<E> spliterator()</code>	return a <code>Spliterator</code> over the elements
<code>Stream<E> stream()</code>	return a sequential <code>Stream</code> with this as its source
<code>Stream<E> parallelStream()</code>	return a parallel <code>Stream</code> with this as its source
<code>Object[] toArray()</code>	copy contents to an <code>Object[]</code>
<code><T> T[] toArray(T[] t)</code>	copy contents to a <code>T[]</code> (for any <code>T</code>)
<code><T> T[] toArray(IntFunction<T[]> generator)</code>	copy contents to a <code>T[]</code> , created by the generator

The first two methods in this group create objects that make the collection's elements available for processing sequentially or in parallel, as described in “[Parallel Streams](#)” on page 153. As that section explains, the most common use of `Spliterators` is internally in one of the next two methods, `stream` and `parallelStream`, which support the functional-style processing of `stream` elements.

The last three methods, different overloads of `toArray`, all return an array containing the elements of this collection. The first one creates a new `Object[]`, while the second and third take a `T[]` or a function producing a `T[]` of a given size, respectively, and return a `T[]` containing the elements of the collection. These methods are important because many APIs—principally older ones and those for which performance is especially important—expose methods that accept or return arrays.

As discussed in “[How to Create Arrays](#)” on page 83, the arguments of the last two methods are required in order to provide the virtual machine with the reifiable type of the array. The new array will be created during the method execution, although if the array supplied as the argument to the second overload of `toArray` is long enough, it is used to receive the elements of the collection, overwriting its existing elements. Taking advantage of this is actually less efficient than supplying an array of zero length, like so:

```
Collection<String> cs = ...
String[] sa = cs.toArray(new String[0]);
```

although this equivalent call to the third overload of `toArray` expresses the meaning most clearly:

```
Collection<String> cs = ...  
String[] sa = cs.toArray(String[]::new);
```

Why is *any* type allowed for T in the declaration of the last two methods? The type variable T is unrelated to the collection type parameter E, permitting errors at run time that it seems should have been caught at compile time. For example, this code:

```
List.of(1, 2, 3).toArray(new String[0])      // array store exception
```

compiles successfully but throws `ArrayStoreException` at run time.

One plausible way of relating the two type variables in the declaration of the second overload of `toArray` might have been something like this:

```
<T super E> T[] toArray(T[] a)
```

allowing an array to be created at any supertype of E. But lower-bounded type variables aren't allowed in the language, possibly because of limitations of the type inference algorithm employed when generics were introduced in Java 5 ([Smith and Cartwright 2008](#)). (See also Langer, <https://oreil.ly/nZQmH>.)

Even if lower-bounded type parameters were allowed, they wouldn't provide complete type safety—it would still be possible for code to be compiled without warnings but to throw `ArrayStoreException` at run time. For example:

```
List<Integer> list = List.of(1, 2, 3);  
Object[] array = new String[3];  
list.toArray(array);      // throws ArrayStoreException
```

Of course, no one would write this toy code, but once you decide to allow T and E to represent different types, a situation like this could realistically occur. The exception is thrown because the run-time component type of the array is `String` (so it cannot accept `Integer` values), but the static component type of the variable `array` is `Object`. As usual, the covariance of arrays circumvents static type checking.

It is often possible to come up with a workaround for the lack of lower-bounded type parameters by introducing static helper methods (see Langer, <https://oreil.ly/nZQmH>). A method like this:

```
static <O,I extends O> O[] toArray(Collection<I> c, O[] a) {  
    return c.toArray(a);  
}
```

would catch some errors. However, it could still be subverted by array covariance as described previously.

The question remains: Why not restrict the array component type to exactly E, the parametric type of the collection? The principal reason is to allow the possibility of

giving the array a more specific component type than that of the collection, when the elements of the collection all happen to belong to the same subtype:

```
List<Object> l = List.of("zero", "one");
String[] a = l.toArray(new String[0]);
```

Here, a list of objects happens to contain only strings, so it can be converted into a `String[]`, in an operation analogous to the `promote` method described in “[Instance Tests and Casts](#)” on page 71. Of course, if the list contains an object that is not a string, the error is caught at run time rather than compile time (see “[Arrays](#)” on page 24):

```
List<Object> l = List.of("zero", "one", 2);
String[] a = l.toArray(new String[0]);      // throws ArrayStoreException
```

In general, you may want to copy a collection of a given type into an array of a more specific type (for instance, copying a list of objects into an array of strings, as just shown) or of a more general type (for instance, copying a list of strings into an array of objects).

On occasion, you might want to map a collection of a given type into an array of a completely unrelated type (for instance, mapping a list of integers into an array of strings), but that requires more than simply copying and isn’t possible in this way. You can try to do it, because the unrelated type `T` argument of the second and third overloads of `toArray` allows you to write the code, but the result will be an `ArrayStoreException` at run time.

One drawback of this design is that, applied to collections of wrapper types, it doesn’t accommodate automatic unboxing into the corresponding array of primitives:

```
List<Integer> l = List.of(0, 1, 2);
int[] a = l.toArray(new int[0]); // compile-time error
```

This is illegal because the parameter `T` in the method call must—as for any type parameter—be a reference type. The call would work if we replaced both occurrences of `int` with `Integer`, but often this isn’t an option because, for performance or compatibility reasons, we require an array of primitive type. In such cases, we have to resort to copying the array explicitly:

```
jshell> List<Integer> integers = List.of(0, 1, 2);
integers ==> [0, 1, 2]
jshell> int[] ints = new int[integers.size()];
ints ==> int[3] { 0, 0, 0 }
jshell> for (int i=0; i<integers.size(); i++) { ints[i] = integers.get(i); }
jshell> ints
ints ==> int[3] { 0, 1, 2 }
```

In this situation, the Stream API provides a neater and clearer alternative:

```
jshell> int[] ints = integers.stream()
...>     .mapToInt(Integer::intValue)
```

```
...>     .toArray();
ints ==> int[3] { 0, 1, 2 }
```

Using the Methods of Collection

Let's construct a small example to illustrate the use of the collection classes. Your authors are forever trying to get organized; our latest effort involves writing our own (very basic) to-do manager. In this example, you'll see how to write code in the classical Collections Framework idiom, and also, in sidebars, the more modern functional style of the Stream API.

We begin by defining an interface to represent tasks, and records to represent two different kinds of tasks: writing code and making phone calls.

This is the interface that defines a task:

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/Task
public interface Task extends Comparable<Task> {
    default int compareTo(Task t) {
        return toString().compareTo(t.toString());
    }
}
```

and the two concrete Task types are:

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/CodingTask
public record CodingTask(String spec) implements Task {}
```

and:

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/PhoneTask
public record PhoneTask(String name, String number) implements Task {}
```

Tasks will need to be comparable to be used in ordered collections (such as SequencedSet and SequencedMap). The other methods they will require—equals, hashCode, and toString—are automatically supplied by the record implementation. The natural ordering on tasks (see “Comparable” on page 35) corresponds to the ordering on their string representations, and as records, equality of two tasks is defined by the equality of their string-valued fields. Since the natural ordering on strings is consistent with equality—that is, compareTo returns 0 exactly when equals returns true—it follows that for tasks also, the natural order is consistent with equality (see “Consistent with equals” on page 37 and “Inconsistent with equals” on page 322).

A coding task is specified by its name, and a phone task is specified by the name and number of the person to be called. As records whose string components are all immutable, objects of these types are themselves immutable (see “Immutability and Unmodifiability” on page 147).

The default `toString` method of a record always returns a string that begins with the name of the record type. Since "CodingTask" precedes "PhoneTask" in the alphabetic ordering on strings, and since tasks are ordered according to the results returned by `toString`, it follows that coding tasks come before phone tasks in the natural ordering—highly appropriate for people who much prefer coding to talking on the phone!

We also define an empty task:

```
org/jcbook/chapter10/A_using_the_methods_of_collection/EmptyTask
public record EmptyTask() implements Task {
    public String toString() {
        return "";
    }
}
```

Since the empty string precedes all others in the natural ordering on strings, the empty task comes before all others in the natural ordering on tasks. This task will be useful when we construct range views of sorted sets (see "[Getting range views](#)" on [page 196](#)).

[Example 10-1](#) shows how to define a series of tasks to be carried out. (In a real system, of course, tasks would be most likely held in a database and retrieved from there.)

Example 10-1. Sample data for the task manager

```
org/jcbook/chapter10/A_using_the_methods_of_collection/Example101
```

```
PhoneTask mikePhone = new PhoneTask("Mike", "987 6543");
PhoneTask paulPhone = new PhoneTask("Paul", "123 4567");
CodingTask databaseCode = new CodingTask("db");
CodingTask guiCode = new CodingTask("gui");
CodingTask logicCode = new CodingTask("logic");

Collection<PhoneTask> phoneTasks = new HashSet<>();
Collection<CodingTask> codingTasks = new HashSet<>();
Collection<Task> mondayTasks = new HashSet<>();
Collection<Task> tuesdayTasks = new HashSet<>();

Collections.addAll(phoneTasks, mikePhone, paulPhone);
Collections.addAll(codingTasks, databaseCode, guiCode, logicCode);
Collections.addAll(mondayTasks, logicCode, mikePhone);
Collections.addAll(tuesdayTasks, databaseCode, guiCode, paulPhone);

assert phoneTasks.equals(Set.of(mikePhone, paulPhone));
assert codingTasks.equals(Set.of(databaseCode, guiCode, logicCode));
assert mondayTasks.equals(Set.of(logicCode, mikePhone));
assert tuesdayTasks.equals(Set.of(databaseCode, guiCode, paulPhone));
```

As part of the retrieval process, the tasks have been organized into various categories—phone tasks, coding tasks, and so on—represented by sets, using the method `Collections.addAll` introduced in “[Generic Methods and Varargs](#)” on page 7. Set is unique in the Collections Framework in having exactly the same methods as the top-level `Collection` interface. (Fortunately, it is also appropriate for this example because there is never a need to store duplicate tasks.) So we can now use the methods of `Collection` to work with these categories. The examples that follow use the methods in the order in which they were presented earlier.

Adding Elements

We can add new tasks to the schedule:

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_1](#)

```
PhoneTask ruthPhone = new PhoneTask("Ruth", "567 1234");
mondayTasks.add(ruthPhone);
assert mondayTasks.equals(Set.of(logicCode, mikePhone, ruthPhone));
```

or we can combine schedules together using `addAll`, which implements set union:

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_2](#)

```
Collection<Task> allTasks_1 = new HashSet<>(mondayTasks);
allTasks_1.addAll(tuesdayTasks);
assert allTasks_1.equals(Set.of(logicCode, mikePhone, ruthPhone,
    databaseCode, guiCode, paulPhone));
```

Stream Alternative

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_3](#)

```
Collection<Task> allTasks_2 = Stream.of(mondayTasks, tuesdayTasks)
    .flatMap(Collection::stream)
    .collect(Collectors.toSet());
assert allTasks_2.equals(Set.of(logicCode, mikePhone, ruthPhone,
    databaseCode, guiCode, paulPhone));
```

The collection `allTasks_2`, resulting from the stream alternative, may not be the same as `allTasks_1`. The contract for `Collectors::toSet` provides no guarantees on the type of the set returned. At Java 21, the OpenJDK returns a `HashSet`, so the stream alternative does in fact produce the same result. But it is possible, if unlikely, that future implementations could return, for example, an unmodifiable set. If you want to specify the type of the returned set precisely, use `Collectors::toCollection`, supplying a collection constructor. For example, to specify a `HashSet` for `allTasks_2`, replace the emphasized line with:

```
.collect(Collectors.toCollection(HashSet::new))
```

This comment applies to many of the subsequent stream alternative examples.

Removing Elements

When a task is completed, we can remove it from a schedule:

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_4
boolean wasPresent = mondayTasks.remove(mikePhone);
assert wasPresent;
assert mondayTasks.equals(Set.of(logicCode, ruthPhone));
```

and we can clear a schedule out altogether because all of its tasks have been done (a method that, in reality, we don't get to use very often):

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_5
mondayTasks.clear();
assert mondayTasks.equals(Collections.EMPTY_SET);
```

The removal methods also allow us to combine entire collections in various ways. For example, to see which tasks other than phone calls are scheduled for Tuesday, we can write:

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_6
Collection<Task> tuesdayNonPhoneTasks = new HashSet<>(tuesdayTasks);
tuesdayNonPhoneTasks.removeAll(phoneTasks);
assert tuesdayNonPhoneTasks.equals(Set.of(databaseCode, guiCode));
```

Stream Alternative

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_7
Collection<Task> tuesdayNonPhoneTasks = tuesdayTasks.stream()
    .filter(t -> ! phoneTasks.contains(t))
    .collect(Collectors.toSet());
```

And we can see which phone calls are scheduled for that day using `retainAll`, which implements set intersection:

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_8
Collection<Task> phoneTuesdayTasks = new HashSet<>(tuesdayTasks);
phoneTuesdayTasks.retainAll(phoneTasks);
assert phoneTuesdayTasks.equals(Set.of(paulPhone));
```

This last example can be approached differently to achieve the same result:

```
org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_9
Collection<PhoneTask> tuesdayPhoneTasks = new HashSet<>(phoneTasks);
tuesdayPhoneTasks.retainAll(tuesdayTasks);
assert tuesdayPhoneTasks.equals(Set.of(paulPhone));
```

Stream Alternative

[org/jcbook/chapter10/A_using_the_methods_of_collection/Program_10](#)

```
Collection<PhoneTask> tuesdayPhoneTasks = phoneTasks.stream()
    .filter(tuesdayTasks::contains)
    .collect(Collectors.toSet());
```

Note that `phoneTuesdayTasks` has the type `Collection<Task>`, while `tuesdayPhoneTasks` has the more precise type `Collection<PhoneTask>`.

This last example provides an explanation of the signatures of methods in this group and the next. We have already discussed (in “[Bounded or Unbounded?](#)” on page 27) why they take arguments of type `Object` or `Collection<?>` whereas the methods for adding to the collection restrict their arguments to its parametric type. Taking the example of `retainAll`, its contract requires the removal of the elements of this collection that do not occur in the argument collection. That doesn’t justify any restriction on what the argument collection may contain; in the example just given it can contain instances of any kind of `Task`, not just `PhoneTask`. There is no reason even to restrict the argument in this way to collections of supertypes of the parametric type; we actually want the least restrictive type possible, which is `Collection<?>`. Similar reasoning applies to `remove`, `removeAll`, `contains`, and `containsAll`. “[Object Versus E](#)” on page 324 explores this topic in greater detail.

Querying the Contents of a Collection

These methods allow us to check, for example, that the preceding operations have worked correctly. We’ll use `assert` here to make the system check our belief that we have programmed those operations correctly. For example, the first of these statements will fail, throwing an `AssertionError`, if `tuesdayPhoneTasks` does not contain `paulPhone`:

[org/jcbook/chapter10/A_using_the_methods_of_collection/Program_11](#)

```
assert tuesdayPhoneTasks.contains(paulPhone);
assert tuesdayTasks.containsAll(tuesdayPhoneTasks);
assert mondayTasks.isEmpty();
assert mondayTasks.size() == 0;
```

Making a Collection’s Contents Available for Further Processing

The methods in this group provide an iterator over the collection, deliver its contents into a stream, or convert it to an array.

“[Iterable and Iterators](#)” on page 135 showed how most uses of iterators can be replaced by the simpler `foreach` statement, which uses them implicitly. But there are

uses of iteration with which *foreach* can't help; for instance, you have to use an explicit iterator if you want to change the structure of a collection without encountering a `ConcurrentModificationException`, or if you want to process two lists in parallel. To give an example, suppose we decide that we don't have time for phone tasks on Tuesday. It may be tempting to use *foreach* to filter them from our task list, but that won't work for the reasons described in “[Iterable and Iterators](#)” on page 135:

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_12](#)

```
// throws ConcurrentModificationException
for (Task t : tuesdayTasks) {
    if (t instanceof PhoneTask) {
        tuesdayTasks.remove(t);
    }
}
```

Using an iterator explicitly is no improvement if you still use `Collection` methods to modify the structure:

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_13](#)

```
// throws ConcurrentModificationException
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
    Task t = it.next();
    if (t instanceof PhoneTask) {
        tuesdayTasks.remove(t);
    }
}
```

But using the *iterator*'s structure-changing method gives the result we want:

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_14](#)

```
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
    Task t = it.next();
    if (t instanceof PhoneTask) {
        it.remove();
    }
}
assert tuesdayTasks.equals(Set.of(databaseCode, guiCode));
```

Stream Alternative

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_15](#)

```
Set<Task> tuesdayCodeTasks = tuesdayTasks.stream()
    .filter(t -> !(t instanceof PhoneTask))
    .collect(Collectors.toSet());
assert tuesdayCodeTasks.equals(Set.of(databaseCode, guiCode));
```

The stream version has the advantages of clarity, brevity, and the possibility of efficiently parallelizing this operation, simply by changing the call of `stream` to `parallel`

Stream. The performance cost of these advantages is the extra memory required to create the new collection.

For another example, suppose we are the kind of fastidious people who like to keep all our lists of tasks in ascending order, and we want to merge two lists of tasks into a single list, while maintaining the order. [Example 10-2](#) shows how we can merge two collections into a third, provided that the iterators of each return their elements ascending in natural order. This method relies on the fact that the collections to be merged contain no null elements; if one is encountered, the method throws a `NullPointerException`.

Example 10-2. Merging collections using natural ordering

org/jcbook/chapter10/A_using_the_methods_of_collection/MergeCollections

```
public class MergeCollections {  
    static <T extends Comparable<? super T>> List<T> merge  
        (Collection<? extends T> c1, Collection<? extends T> c2) {  
        List<T> mergedList = new ArrayList<T>();  
        Iterator<? extends T> itr1 = c1.iterator();  
        Iterator<? extends T> itr2 = c2.iterator();  
        T c1Element = getNextElement(itr1);  
        T c2Element = getNextElement(itr2);  
        // each iteration will take a task from one of the iterators;  
        // continue until neither iterator has any further tasks  
        while (c1Element != null || c2Element != null) {  
            // use the current c1 element if either the current c2  
            // element is null or both are non-null and the c1 element  
            // precedes the c2 element in the natural order  
            boolean useC1Element = c2Element == null ||  
                c1Element != null && c1Element.compareTo(c2Element) < 0;  
            if (useC1Element) {  
                mergedList.add(c1Element);  
                c1Element = getNextElement(itr1);  
            } else {  
                mergedList.add(c2Element);  
                c2Element = getNextElement(itr2);  
            }  
        }  
        return mergedList;  
    }  
    static <E> E getNextElement(Iterator<E> itr) {  
        if (itr.hasNext()) {  
            E nextElement = itr.next();  
            if (nextElement == null) throw new NullPointerException();  
            return nextElement;  
        } else {  
            return null;  
        }  
    }  
}
```

We cannot use the collections `mondayTasks` and `tuesdayTasks` in [Example 10-1](#) directly to demonstrate this example, since they are `HashSets`, whose iterators are not specified to return their elements in any particular order. But we can instead use `TreeSet`, another implementation of the `Set` interface, whose iterators by default return the set elements in natural order (see “[NavigableSet](#)” on page 193 for details):

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_16](#)

```
Collection<Task> sortedMondayTasks = new TreeSet<>(mondayTasks);
Collection<Task> sortedTuesdayTasks = new TreeSet<>(tuesdayTasks);
Collection<Task> mergedTasks =
    MergeCollections.merge(sortedMondayTasks, sortedTuesdayTasks);
assert mergedTasks.equals(
    List.of(databaseCode, guiCode, logicCode, mikePhone, paulPhone));
```

Stream Alternative

[org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_17](#)

```
<T> List<T> merge(Collection<? extends T> c1, Collection<? extends T> c2) {
    return Stream.of(c1, c2)
        .flatMap(Collection::stream)
        .sorted()
        .collect(Collectors.toCollection(ArrayList::new));
}
```

This method is certainly more concise, and arguably easier to understand, than [Example 10-2](#). The algorithm that it uses for the sort, called Timsort, can make use of ascending runs and so is space-efficient in the same way as `MergeCollections`. However, it is limited to merging using the comparison method; if the two lists to be merged are ordered in some other way—say, preserving encounter order, as when two lists are zipped together—then `MergeCollections` has a role.

Implementing Collection

There are no concrete implementations of `Collection`. The `AbstractCollection` class, which partially implements it, is one of five skeletal implementations—the others are `AbstractSet`, `AbstractList`, `AbstractQueue`, and `AbstractMap`—which provide functionality common to the different concrete implementations of each interface. These skeletal implementations exist to help the designer of new implementations of the framework interfaces. For example, `Collection` could serve as the interface for bags (unordered lists), and a programmer implementing bags could extend `AbstractCollection` and find most of the implementation work already done. For a worked example using `AbstractList`, see “[Customize Collections Using the Abstract Classes](#)” on page 305.

Collection Constructors

Before going on to look at the four main kinds of collections (sets, queues, lists, and maps), we should first explain two common forms of constructor that are shared by most of their implementations. Taking `HashSet` as an example of `Collection` implementations, these are:

```
public HashSet()
public HashSet(Collection<? extends E> c)
```

The first of these creates an empty set, and the second a set that will contain the elements of any collection of the parametric type—or one of its subtypes, of course. Using this constructor has the same effect as creating an empty set with the default constructor, then adding the contents of a collection using `addAll`. This is sometimes called a “copy constructor,” but that term should really be reserved for constructors that make a copy of an object of the same *class*, whereas constructors of the second form can take any object that implements the *interface* `Collection<? extends E>`. Bloch (2017, item 13) proposes the term “conversion constructor.”

Implementations of the `Map` interface follow an analogous pattern. For example, the constructors of `HashMap` include these two variants:

```
public HashMap()
public HashMap(Map<? extends K, ? extends V> m)
```

Not all collection classes have constructors of both forms—`ArrayBlockingQueue`, for example, cannot be created without fixing its capacity, and `SynchronousQueue` cannot hold any elements at all, so no constructor of the second form is appropriate. In addition, many collection classes have other constructors besides these two, but which ones they have depends not on the interface they implement but on the underlying implementation; these additional constructors are used to configure the implementation.

Conclusion

In this chapter, we explored the `Collection` interface, the original root of the Collections Framework. Understanding its operations is fundamental to effective use of the framework.

By contrast, `SequencedCollection` is a recent addition. It brings together disparate classes that all have something in common: they represent a sequence of values with operations that work equally on their beginning and their end and can be traversed in either direction. That is the subject of the next chapter.

The SequencedCollection Interface

The interface `SequencedCollection` occupies a unique place in the design space of the Collections Framework. Rather than specifying the contract for a particular abstract data structure, it unifies behaviors exposed by a variety of preexisting types—`List`, `NavigableSet`, `Deque`, and `LinkedHashSet`—that are spread across the type hierarchy and mostly not otherwise related (see [Figure 11-1](#)).

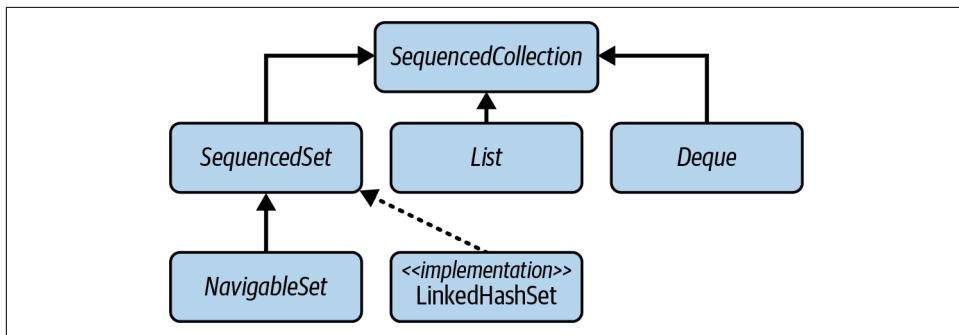


Figure 11-1. Type hierarchy for `SequencedCollection`

What these collections have in common is that they all represent a sequence of elements whose order is semantically significant (they obey a total order, called in the JDK documentation the *encounter order*), whether the order is internally or externally imposed (see “[Sequenced Collections](#)” on page 132).

The Methods of `SequencedCollection`

It was `Deque` (see “[Deque](#)” on page 224) that introduced the set of methods that has now been generalized to all collections with an encounter order, providing the ability to add, inspect, and remove the first and last elements of the sequence. Some of

these capabilities were previously available, unevenly and under different names, in these collections: for example, `NavigableSet` supported the removal of the first and last elements but not their inspection, while `List` directly supported only addition of a last element, all other operations requiring a sequence position to be specified. **Table 12-1** shows the correspondence between `NavigableSet` method calls and their `SequencedCollection` equivalents, with the correspondence for `List` shown in **Table 14-1**. No such table is needed for `Deque` because, apart from `reversed`, the operations of `SequencedCollection` are all the result of promotion from `Deque`.

Of course, it would be impossible to unify the behavior of so many disparate collections without some special cases. For example, it makes no sense to add a first or last element to a `NavigableSet`, given that the position of all the elements in these collections is determined by the internal sorting algorithm. For another example, unmodifiable lists cannot support structural modifications such as addition and removal of elements. These special cases will be dealt with in the chapters covering the different collection types.

The single new method provided by `SequencedCollection` is `reversed`, which provides a reverse-ordered view (see “[Views](#)” on page 141) of the original collection. This makes the ability to process an ordered collection in reverse (previously only available to clients of `NavigableSet` via its `descendingSet` method) available to users of any collection with encounter order.

Adding Elements

```
void addFirst(E e)    add e as the first element of this collection  
void addLast(E e)   add e as the last element of this collection
```

These methods add elements to either end of a `SequencedCollection`. The contracts for these methods specify that after their successful execution, the collection must contain the supplied element as the first or, respectively, last element in the encounter order.

Inspecting Elements

```
E getFirst()  return the first element of this collection  
E getLast()   return the last element of this collection
```

These methods inspect the elements at either end of a `SequencedCollection`. Calling an inspection method on an empty collection will result in a `NoSuchElementException`.

Removing Elements

```
E removeFirst()    remove and return the first element of this collection  
E removeLast()   remove and return the last element of this collection
```

These methods remove elements from either end of a `SequencedCollection`. Calling a removal method on an empty collection will result in a `NoSuchElementException`.

Generating a Reversed View

```
SequencedCollection<E> reversed()  return a reversed view of this collection
```

This method returns a reverse-ordered view of a collection. Of course, you would prefer that calling `reversed` on, for example, a `NavigableSet` (an interface that inherits from `SequencedCollection`) would return a `NavigableSet`, rather than a generic `SequencedCollection` as the declaration implies. For this reason, the four interfaces that inherit from `SequencedCollection`—`List`, `SequencedSet`, `NavigableSet`, and `Deque`—all define covariant overrides (see “[Covariant Overriding](#)” on page 57) for `reversed`. For example, `SequencedSet`’s declaration of `reversed` is:

```
SequencedSet<E> reversed()
```

Similar overrides are declared by the other three interfaces.

The reverse ordering affects iteration and other order-sensitive operations. It affects the order returned by `Iterable::foreach`, `iterator`, `listIterator`, `spliterator`, `stream`, and `parallelStream`. It swaps the senses of `first` and `last` in the `add/get/remove` family of methods, as well as `List`’s `indexOf/lastIndexOf`. The order of elements deposited into the array by the `toArray` family of methods also reflects the `reversed` encounter order. These effects extend to views of the returned view.

The contract states that any successful modifications to this view must write through to the underlying collection, but that the inverse—visibility in this view of changes to the underlying collection—is implementation-dependent. The most commonly used implementation, `ArrayList`, does provide this feature: modifications to the underlying collection are visible in the `reversed` view.

Conclusion

In this chapter, we were able to review the features of `SequencedCollection` quickly, because it was introduced not to provide radical new capabilities but to unify capabilities of existing ordered implementations like `NavigableSet` and `Deque` and to make them available elsewhere (for example, to `LinkedHashSet` and `LinkedHashMap`).

The interfaces of the last two chapters, `Collection` and `SequencedCollection`, provide essential background for the subject of the next chapter: the interface `Set` and its implementations.

CHAPTER 12

Sets

A *set* is a collection of items that cannot contain duplicates; adding an item that is already present in the set has no effect. The interface `Set<E>` has the same methods as those of `Collection` but it is defined separately in order to allow the contract of `add` (and `addAll`, which is defined in terms of `add`) to reflect this. The `equals` method is overridden: the `Set` contract states that a `Set` can only ever be equal to another `Set`, and then only if they are the same size and contain equal elements. The `hashCode` method is also overridden, as should always be the case when `equals` is overridden (see “Hash tables” in “[Implementations](#)” on page 138). The hash code of a `Set` is the sum of the hash codes of its elements.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter12](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter12)

Let’s see an example of set operations in action. Returning to the to-do manager example introduced in “[Using the Methods of Collection](#)” on page 168, suppose that on Monday you have free time to carry out your telephone tasks. You can make the new collection of tasks for Monday by adding all your telephone tasks to the existing Monday tasks. Let `mondayTasks` and `phoneTasks` be as declared in [Example 10-1](#).

Using a set, you can write:

[org/jgcbook/chapter12/Program_1](#)

```
Set<Task> phoneAndMondayTasks = new HashSet<>(mondayTasks);
phoneAndMondayTasks.addAll(phoneTasks);
assert phoneAndMondayTasks.equals(Set.of(logicCode, mikePhone, paulPhone));
```

This works because of the way that duplicate elements are handled. The task `mikePhone`, although it is in both `mondayTasks` and `phoneTasks`, appears only once in `phoneAndMondayTasks`—appropriately for this application, as you definitely don't want to have to start doing tasks twice over!

Defining a Set: Equivalence Relations

This seems straightforward, but to understand exactly how different `Set` implementations work, we need to look more closely at what defines a duplicate—that is, the *equivalence relation* for that set. The previous example uses `HashSet`, for which the equivalence relation is the `equals` method: in other words, for a `HashSet`, two objects are duplicates if and only if the `equals` method, called on one with the other as its argument, returns `true`. This might seem an obvious choice, but it's not the only one (although the Javadoc at JDK 21 for `Set` and its implementations often implies that it is). Some other implementations also use `equals`, like `CopyOnWriteArrayList` and the family of classes that we're calling *UnmodifiableSet* (see “[UnmodifiableSet](#)” on [page 189](#)). But another choice, if less common, is the identity relation; a set using that relation will contain a reference to every unique object that has been added to it. One such example is `EnumSet` (see “[EnumSet](#)” on [page 187](#))—although, since enums are singletons, the result of the `equals` method matches the result of the identity relation for all comparisons. Another example is the set view of the keys of an `IdentityHashMap`, or any set created from an `IdentityHashMap` (see “[IdentityHashMap](#)” on [page 259](#)) using the `Collections` method `newSetFromMap`.

A third alternative for an equivalence relation is specified by the contract for `NavigableSet`. A `NavigableSet` (see “[NavigableSet](#)” on [page 193](#)) maintains its elements in sorted order using an ordering relation provided by either its natural order or a `Comparator` (see [Chapter 3](#)). `NavigableSet` also uses that ordering relation in another way: it defines two objects as equivalent if, using it, they compare as equal—that is, if the comparison method returns `0`—regardless of whether they satisfy the equality relation.

The fact that different sets use different equivalence relations creates no difficulties so long as the different relations are compatible (see [Chapter 3](#))—that is, they give the same result applied to every value pair. Confusion arises, however, with incompatible relations; for example, two objects may not satisfy the equality relation even though the set's ordering relation compares them as equal. This confusion is unfortunately compounded by the Javadoc for `Set` and its implementations, which in places fails to recognize that some implementations use equivalence relations other than `equals`. (Hopefully, future Javadoc versions may correct that problem.)

You can avoid the confusion by understanding the consequences of using different equivalence relations for different sets. One consequence, obvious once you understand the problem, is that sets may contain duplicate elements that satisfy `equals` or, conversely, that they may elide occurrences of ones that don't. A less obvious consequence concerns equality between sets: to determine whether two sets `A` and `B` are equal, `A` must test each member of `B` to discover whether it is equivalent to a member of `A`. If the roles are reversed, and if `A` and `B` are using different equivalence relations, the results may be different, so set equality loses symmetry.

There is a longer discussion of this problem, with examples, in “[Inconsistent with equals](#)” on page 322.

Set Implementations

When we used the methods of `Collection` in the examples in [Chapter 10](#), we emphasized that they would work with any implementation of `Collection`, not only `Set`. But in practice, of course, you have to decide on a concrete implementation. This chapter surveys the different `Set` implementations provided by the Collections Framework, which differ both in how fast they perform the basic operations of `add`, `contains`, and iteration and in the order in which their iterators return their elements. At the end of this chapter, we'll summarize the comparative performance of the different implementations.

Of the `Set` implementations in the Collections Framework, four directly implement `Set` itself; the others inherit from the subinterface `SequencedSet` (see “[SequencedSet](#)” on page 191). In this section, we will look at these four: `HashSet`, `CopyOnWriteArraySet`, `EnumSet`, and the family of implementations that we're calling `UnmodifiableSet` (see “[UnmodifiableSet](#)” on page 189).

HashSet

This class is the most commonly used implementation of `Set`. As the name implies, it is implemented by a *hash table*, an array in which elements are stored at a position derived from their contents. Since hash tables store and retrieve elements by their content, they are well suited to implementing the operations of `Set` (the Collections Framework also uses them for various implementations of `Map`). For example, to implement `contains(Object o)`, you would look for the element `o` and return `true` if it were found.

An element's position in a hash table is calculated by a *hash function* of its contents. Hash functions are designed to give, as far as possible, an even spread of results (*hash codes*) from the element values that might be stored. For an example, the `String` class uses code like this to calculate a hash code:

```

int hash = 0;
for (char ch : str.toCharArray()) {
    hash = hash * 31 + ch;
}

```

Traditionally, hash tables obtain a table index from the hash code by taking the remainder after division by the table length. Division is a slow operation in practice, so the Collections Framework classes use bit masking instead. Since that means it is the pattern of bits at the low end of the hash code that is significant, prime numbers (such as 31, here) are used in calculating the hash code, because multiplying by primes will not tend to shift information away from the low end, as would multiplying by a power of two, for example.

Clearly, unless your table has more locations than there are values that might be stored in it, sometimes two distinct values will hash to the same location in the hash table (this is called a *collision*). For instance, no `int`-indexed table can be large enough to store all string values without collisions. We can minimize the problem with a good hash function—one that spreads the elements out equally in the table—but, when collisions do occur, we need to have a way of keeping the colliding elements at the same table location, or *bucket*. This is often done by storing them in a linked structure—a list or a tree—as shown in [Figure 12-1](#). We will look at linked structures in more detail later, but for now it's enough to see that different elements stored in the same bucket can still be accessed, at the cost of following a chain of cell references.

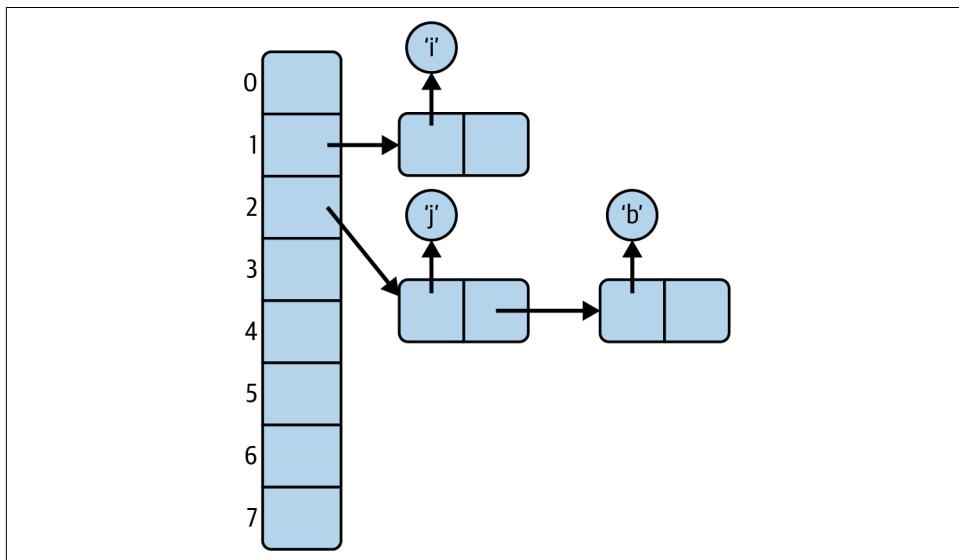


Figure 12-1. A hash table with chained overflow

[Figure 12-1](#) shows the situation resulting from running this code on the OpenJDK implementation of Java 21:

```
Set<Character> s1 = new HashSet<Character>(8);
s1.add('j');
s1.add('i');
s1.add('b');
```

The index values of the table elements have been calculated by using the bottom three bits (for a table of length 8) of the hash code of each element. In this implementation, a `Character`'s hash code is just the Unicode value of the character it contains. In practice, a hash table would be much bigger than this. Also, note that this diagram is a simplified representation: `HashSet` is actually implemented by a specialized `HashMap` (see [“HashMap” on page 257](#)), so each of the cells in the chain contains not one but two references, to a key and a value. Therefore, the picture is more like that of [Figure 15-2](#), only with the same value for every key. [Figure 12-1](#) shows only the key because, when a hash table is being used to represent a set, only the presence of the key is significant.

As long as there are no collisions, the cost of inserting or retrieving an element is constant. As the hash table fills, collisions become more likely; assuming a good hash function, the probability of a collision in a lightly loaded table is proportional to its *load*, defined as the number of elements in the table divided by its capacity (the number of buckets). If a collision does take place, an overflow structure—a linked list or tree—has to be created and subsequently traversed, adding an extra cost to insertion.

The overflow structure used is normally a linear list, but if for a given bucket the list grows in length past a certain threshold, it is converted to a red-black binary tree (see [“TreeSet” on page 200](#)). The ordering relationship used for the nodes of this tree is the `compareTo` method of the keys, if they implement `Comparable`. If they do not, the node ordering fallback is class name comparison or, failing that, identity hash code comparison. These fallback orderings should be avoided in performance-critical situations. The same caution applies to `HashMap` and `ConcurrentHashMap`. If the size of the hash table is fixed, performance will worsen as more elements are added and the load increases. To prevent this from happening, the table size is increased by *rehashing*—copying all elements to a newly allocated, larger table—when the load reaches a specified threshold (its *load factor*).

Iterating over a hash table requires each bucket to be examined to see whether it is occupied and therefore requires an execution count proportional to the capacity of the hash table plus the number of elements it contains. Since the iterator examines each bucket in turn, the order in which elements are returned depends on their hash codes, so there is no guarantee as to the order in which the elements will be returned. In the OpenJDK implementation of Java 21, the hash table shown in [Figure 12-1](#)

yields its elements in order of ascending table index and forward traversal of the linked lists. Printing it produces the following output:

```
[i, j, b]
```

In a later section, we will look at `LinkedHashSet`, a variant of this implementation with an iterator that instead returns elements in their insertion order.

The chief attraction of a hash table implementation for sets is the constant-time performance (for lightly loaded tables and with a good hash function) of the basic operations of `add`, `remove`, `contains`, and `size`. Its main performance disadvantages are the poor performance of heavily loaded tables and the iteration performance: iterating through the table involves examining every bucket, so the cost includes a factor attributable to the table length, regardless of the size of the set it contains.

HashSet constructors

`HashSet` has the standard constructors that we introduced in “[Collection Constructors](#)” on page 176, together with two additional constructors:

```
HashSet(int initialCapacity)
HashSet(int initialCapacity, float loadFactor)
```

Both of these constructors create an empty set but allow some control over the size of the underlying table, creating one at least as large as the supplied capacity (in the current implementation it has a length of the next largest power of two) and, optionally, with the desired load factor. Most of the hash-based collections have similar constructors. You can use these constructors to create a table large enough to store all the elements you expect it to hold without requiring expensive resizing operations. In practice, however, they have proved confusing and difficult to use—confusing because their `int` parameter, often misunderstood to be the expected number of entries, is in fact used to compute the table size, and difficult to use because computing the argument correctly from the expected maximum number of entries is implementation-dependent and error-prone. So Java 19 added static factory methods, which take only a parameter signifying the expected maximum number of entries. These methods are recommended as being easier to use and less subject to implementation changes. The factory method for `HashSet` is `newHashSet`:

```
static <T> HashSet<T> newHashSet(int numElements)
```

`HashSet` is unsynchronized and not thread-safe; its iterators are fail-fast.

CopyOnWriteArrayList

The functional specification of `CopyOnWriteArrayList` is again straightforwardly derived from the `List` contract, but with quite different performance characteristics from `ArrayList`. This class is implemented as a thin wrapper around an instance of

`CopyOnWriteArrayList`, which in turn is backed by an array. The array is treated as immutable; any modification of the set results in the creation of an entirely new array. So `add` has complexity $O(N)$, as does `contains`, which has to be implemented by a linear search. Clearly, you wouldn't use `CopyOnWriteArrayList` in a context where you were expecting many searches or insertions. But the array implementation means that iteration costs $O(1)$ per element—faster than `HashSet`—and it has one advantage that is really compelling in some applications: it provides thread safety (see “[Collections and Thread Safety](#)” on page 155) without adding to the cost of read operations. This is in contrast to those collections that use locking to achieve thread safety for all operations (for example, the synchronized collections discussed in “[Synchronized Collections](#)” on page 281). Locking operations are always a potential bottleneck in multithreaded applications. By contrast, read operations on copy-on-write collections are implemented on the backing array, and thanks to its immutability they can be used by any thread without danger of interference from a concurrent write operation.

When would you want to use a set with these characteristics? One common situation is managing shared configuration in a multithreaded environment. For example, a server application might maintain a global configuration set of allowed IP addresses that multiple threads frequently read but that is only rarely updated. The process of updating can't be allowed to interfere with read operations; with a locking set implementation, read and write operations share the overhead necessary to ensure this, whereas with `CopyOnWriteArrayList` the overhead is carried entirely by write operations. This makes sense in a scenario in which read operations occur much more frequently than changes to the server configuration.

Since there are no configuration parameters for `CopyOnWriteArrayList`, the constructors are just the standard ones discussed in “[Collection Constructors](#)” on page 176.

EnumSet

An `EnumSet` is a convenience set that can contain a subset of enum members, from none to all of them. This class exists to take advantage of the efficient implementations that are possible when the maximum number of possible elements is fixed and a unique index can be assigned to each. These two conditions hold for a set of elements of the same `Enum` class; the number of keys is fixed by the constants of the enumerated type, and the `ordinal` method returns values that are guaranteed to be unique to each constant. In addition, the values that `ordinal` returns form a compact range, starting from zero—ideal, in fact, for use as array indices or, in the standard implementation, indices of a bit vector. So `add`, `remove`, and `contains` are implemented as bit manipulations, with constant-time performance. Bit manipulation on a single word is extremely fast, and a long value can be used to represent `EnumSets` over enum types with up to 64 values. For example, if the enum `Day` is declared as:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

then the set of weekend days, defined by `EnumSet.of(Day.SATURDAY, DAY.SUNDAY)`, is internally represented by `long` value of 65—that is, $2^0 + 2^6$. Larger enums can be treated in a similar way, with some overhead, using more than one word for the representation.

`EnumSet` is a sealed abstract class that implements these different representations by means of two different package-private subclasses, one for those enums that can be represented by bit positions in a `long` value—that is, those with 64 or fewer elements—and one for all others. It hides these concrete implementations from the programmer, instead exposing factory methods that call the constructor for the appropriate subclass. The following group of static factory methods provides ways of creating `EnumSets` with different initial contents—empty, specified elements only, or all elements of the enum:

<code><E extends Enum<E>> EnumSet<E> of(E first, E... rest)</code>	create a set initially containing the specified elements
<code><E extends Enum<E>> EnumSet<E> range(E from, E to)</code>	create a set initially containing all of the elements in the range defined by the two specified endpoints
<code><E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)</code>	create a set initially containing all elements in <code>elementType</code>
<code><E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)</code>	create a set of <code>elementType</code> , initially empty

An `EnumSet` contains the reified type of its elements, which is used at run time for checking the validity of new entries. This type is supplied by the factory methods in two different ways. The methods `of` and `range` receive at least one enum argument, which can be queried for its declaring class (that is, the `Enum` class that it belongs to). For `allOf` and `noneOf`, which have no enum arguments, a class object is supplied instead (see “[A Classy Alternative](#)” on page 84).

Common cases for `EnumSet` creation are optimized by a second group of methods, which allow you to efficiently create sets with one, two, three, four, or five elements of an enumerated type:

```
<E extends Enum<E>> EnumSet<E> of(E e)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
```

These methods were introduced long before the unmodifiable collections, but those are so much more frequently used now that it’s sometimes useful to be reminded that

not all of factory methods result in unmodifiable collections. This is a case in point: the `EnumSets` created here are modifiable.

A third set of methods allows the creation of an `EnumSet` from an existing collection:

<code><E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)</code>	create an <code>EnumSet</code> with the same element type as <code>s</code> , and with the same elements
<code><E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)</code>	create an <code>EnumSet</code> from the elements of <code>c</code> , which must contain at least one element
<code><E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)</code>	create an <code>EnumSet</code> with the same element type as <code>s</code> , containing the elements not in <code>s</code>

The collection supplied as the argument to the second version of `copyOf` must be nonempty so that the element type can be determined.

In use, `EnumSet` obeys the contract for `Set`, with the added requirement that its iterators will return their elements in their natural order (the order in which their enum constants are declared). It is not thread-safe, but unlike the unsynchronized general-purpose collections, its iterators are not fail-fast. Although the Javadoc (as of Java 21) states that they are weakly consistent (see “[Collections and Thread Safety](#)” on [page 155](#)), this is only partially true: the more commonly used concrete subclass of `EnumSet`—the one for enums of 64 or fewer elements—has in fact a snapshot iterator.

UnmodifiableSet

You won’t find any reference to the name `UnmodifiableSet<E>` in the Javadoc or in the code of the Collections Framework. It’s a name invented in this book for a family of package-private classes that client programmers can never access by name, but that are important because they provide the implementation of the unmodifiable sets obtained from the various overloads of the factory methods `Set.of` and `Set.copyOf`. The properties of the members of this family are described in the Javadoc for `Set`:

- They are unmodifiable: elements cannot be added or removed. Calling any mutator method will always cause `UnsupportedOperationException` to be thrown.
- They are null-hostile. Attempts to create them with null elements result in `NullPointerException`.
- They reject duplicate elements at creation time. Duplicate elements passed to a factory method result in an `IllegalArgumentException`.

The factory methods (which are static, like all factory methods—the keyword is omitted in the following tables for brevity) can be divided into three groups analogous to

those of `EnumSet`, with the obvious difference that the sets created are unmodifiable, whereas `EnumSets` are not. The first contains just an overload of the method `of` that takes no arguments and returns an empty unmodifiable set:

```
<E> Set<E> of()  returns an unmodifiable set containing zero elements
```

The second group of methods allows you to create an unmodifiable set from up to 10 specified elements:

```
<E> Set<E> of(E e1)          return an unmodifiable set containing one element
```

```
<E> Set<E> of(E e1, E e2)    return an unmodifiable set containing two elements
```

```
<E> Set<E> of(E e1, E e2, E e3)  return an unmodifiable set containing three elements
```

further overloads, taking from four to nine arguments

```
<E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)  return an unmodifiable set containing 10 elements
```

The third group allows you to create an unmodifiable set from a collection, an array, or a varargs-supplied list of arguments:

```
<E> Set<E> copyOf(Collection<? extends E> coll)  return an unmodifiable set containing an arbitrary number of elements
```

```
<E> Set<E> of(E... elements)  return an unmodifiable set containing an arbitrary number of elements
```

The classes that make up `UnmodifiableSet` take advantage of their unmodifiability to use fixed-length arrays as the backing structures. Without the overhead of empty table buckets or linked overflow structures, these implementations require much less space than a hashed structure. Iteration is also correspondingly more efficient, with the added benefit of improved spatial locality (see “[Memory](#)” on page 143). The trade-off for faster iteration is that containment can only be determined by a linear search, $O(N)$ in complexity.

In the OpenJDK implementation, the iterators for `UnmodifiableSet` have an unusual characteristic: the order of the iteration is randomly determined for each virtual machine instance. The reason for this design is to avoid replicating an odd sort of compatibility problem with `HashSet`. In the past, developers have based tests and even production code on the apparently fixed iteration order of `HashSet`. But since the contract for `HashSet` doesn’t specify the iteration order, implementers have changed it between versions, resulting in code breaking on upgrades. The iterators for unmodifiable sets have been designed to prevent developers from being lulled into this error.

Set Views of Maps

In the Collections Framework, many sets are implemented as wrappers around a corresponding map, although the maps are encapsulated and invisible to the client. However, the converse does not hold: some maps do not have a corresponding set, including three that we will see in [Chapter 15](#): `WeakHashMap`, `IdentityHashMap`, and `ConcurrentHashMap`. To obtain a set for one of these with the same ordering, concurrency, and performance characteristics as the backing map, you can call the method `Collections::newSetFromMap` on an empty map with a type `Map<E, Boolean>`, where `E` is the element type of the set that you want to create. For example, to create a concurrent set of `Integer`, you could write:

```
Set<Integer> concurrentIntegerSet =  
    Collections.newSetFromMap(new ConcurrentHashMap<Integer, Boolean>())
```

This idiom guarantees that no direct access to the backing map can take place after the set view is created, as required by the specification of `newSetFromMap`.

SequencedSet

A `SequencedSet` is an externally or internally ordered Set that also exposes the methods of `SequencedCollection`. It combines the methods of these two interfaces, adding to them in only one respect: it provides a covariant override of the method `reversed` of `SequencedCollection` in order to return a value of type `SequencedSet` (see “[Generating a Reversed View](#)” on page 179). It has a single direct implementation, `LinkedHashSet`, and is extended by the interface `NavigableSet` (actually, by its parent interface, `SortedSet`, but see the introductory remarks in “[NavigableSet](#)” on page 193). Figure 12-2 illustrates the relationships between these types.

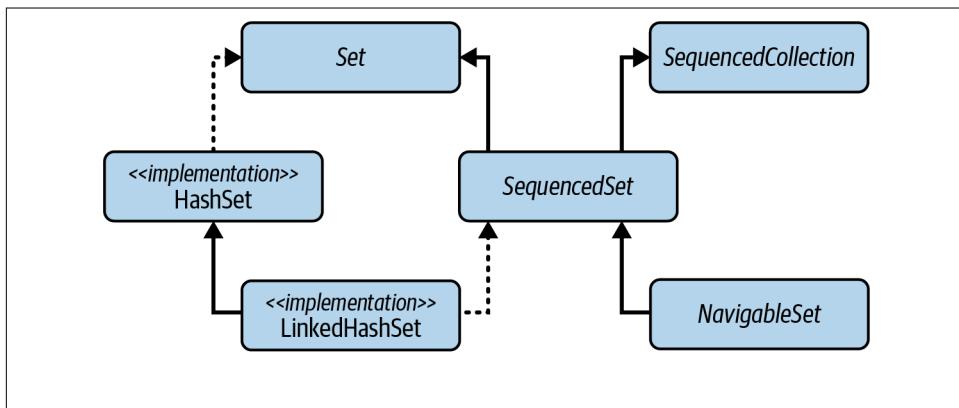


Figure 12-2. *SequencedSet* and related types

LinkedHashSet

This class inherits from `HashSet`, and implements `SequencedSet` by maintaining a linked list of its elements, as shown by the curved arrows in [Figure 12-3](#).

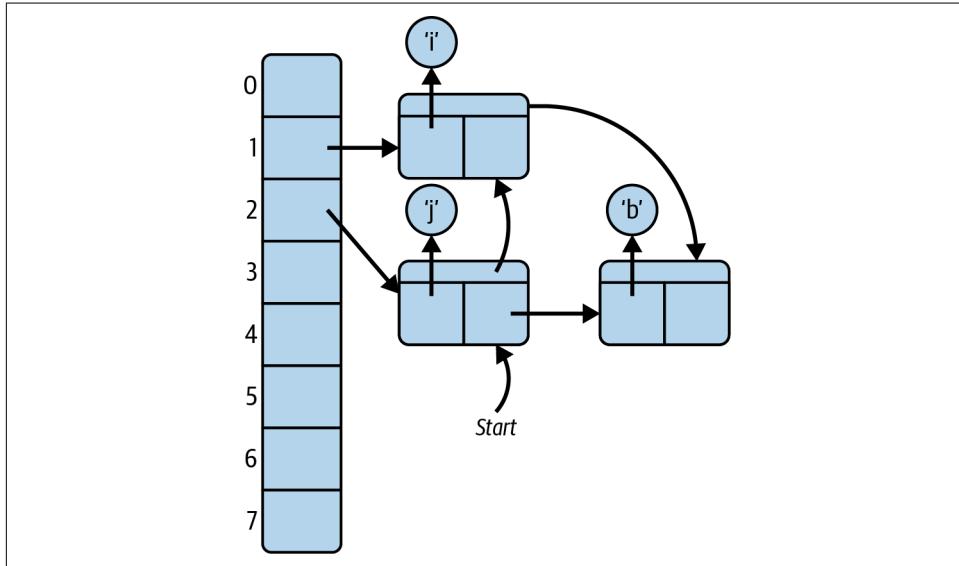


Figure 12-3. A linked hash table

The situation in [Figure 12-3](#) would result from this code:

```
jshell> LinkedHashSet.<Character>newLinkedHashSet(3)
$1 ==> []
jshell> Collections.addAll($1, 'j', 'i', 'b');
$2 ==> true
```

The iterators of a `LinkedHashSet` return their elements in insertion order:

```
jshell> $1
$1 ==> [j, i, b]
```

The methods of `SequencedSet` are all available:

```
jshell> $1.getLast()
$3 ==> 'b'
jshell> $1.reversed()
$4 ==> [b, i, j]
jshell> $1.addLast('k')
jshell> $1
$1 ==> [j, i, b, k]
```

But notice that calling `addFirst` or `addLast` on an element that is already present will reposition it as the first or last element:

```
jshell> $1.addFirst('k')
jshell> $1
$1 ==> [k, j, i, b]
```

The linked structure also has a useful consequence in terms of improved performance for iteration: `next` executes in constant time, as the linked list can be used to visit each element in turn. This is in contrast to `HashSet`, for which every bucket in the hash table must be visited, whether it is occupied or not. In the case of a large table that is sparsely occupied that would be a significant inefficiency, but in general the overhead involved in maintaining the linked list means that you would choose `LinkedHashSet` in preference to `HashSet` only if that situation were likely to apply in your application.

The constructors for `LinkedHashSet` provide the same facilities as those of `HashSet` for configuring the underlying hash table. And similarly to `HashSet`, it has a modern factory method, `newLinkedHashSet`, that accepts as its argument the expected number of elements and sizes the table optimally.

This class is unsynchronized and not thread-safe; its iterators are fail-fast.

NavigableSet

The interface `NavigableSet` adds to the `SequencedSet` contract a guarantee that its iterator will traverse the set in ascending element order, and adds further methods to find the elements adjacent to a target value. Unlike `LinkedHashSet`, its elements are ordered internally by the comparison method of its natural order or of its comparator. So `addFirst` and `addLast` cannot fulfill their contract, and accordingly throw `UnsupportedOperationException`. Prior to the introduction of `NavigableSet`, the only subinterface of `Set` was an interface called `SortedSet`, which guarantees iteration order but does not expose the closest-match methods. `SortedSet` is still in the JDK—it extends `SequencedSet` and is in turn extended by `NavigableSet`—but it is no longer of any great interest, since it has no direct implementations in the platform.

We can use `NavigableSet` to add some useful functionality to the to-do manager. Until now, the methods of `Collection` and `Set` have provided no help in ordering our tasks—surely one of the central requirements of a task manager. [Example 12-1](#) defines a record, `PriorityTask`, that attaches a priority to a task. There are three priorities, `HIGH`, `MEDIUM`, and `LOW`, declared so that `HIGH` priority comes first (i.e., is lowest) in the natural ordering, to correspond with the intuition that “priority one” tasks should get our attention before “priority two” tasks:

[org/jgcbook/chapter12/C_navigable_set/Priority](#)

```
public enum Priority { HIGH, MEDIUM, LOW }
```

To compare two `PriorityTasks`, the comparator `priorityTaskCmpr` first compares their priorities: if the priorities are unequal, the higher-priority task comes first. If the priorities are equal, it goes on to use the natural ordering on the underlying tasks. So two objects can only compare as equal if they actually are equal—that is, the natural ordering is consistent with `equals`, and the problems discussed at the beginning of this chapter don't arise. Conversely, an example of how they could arise would be if we defined the comparator such that all tasks of equal priority compared as equal. With that definition, a `NavigableSet` could only contain at most one task of each priority.

Example 12-1. The record PriorityTask

org/jcbook/chapter12/C_navigable_set/PriorityTask

```
public record PriorityTask(Task task, Priority priority)
    implements Comparable<PriorityTask> {

    static Comparator<PriorityTask> priorityTaskCmpr =
        Comparator.comparing(PriorityTask::priority)
            .thenComparing(PriorityTask::task);

    public int compareTo(PriorityTask pt) {
        return priorityTaskCmpr.compare(this, pt);
    }
}
```

The following code shows `NavigableSet` working with a set of `PriorityTasks`:

org/jcbook/chapter12/C_navigable_set/Program_1

```
NavigableSet<PriorityTask> priorityTasks = new TreeSet<PriorityTask>();

priorityTasks.add(new PriorityTask(mikePhone, Priority.MEDIUM));
priorityTasks.add(new PriorityTask(paulPhone, Priority.HIGH));
priorityTasks.add(new PriorityTask(databaseCode, Priority.MEDIUM));
priorityTasks.add(new PriorityTask(guiCode, Priority.LOW));

assert priorityTasks.toString().equals("""
    [PriorityTask[task=PhoneTask[name=Paul, number=123 4567], priority=HIGH],
     PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
     PriorityTask[task=PhoneTask[name=Mike, number=987 6543], priority=MEDIUM],
     PriorityTask[task=CodingTask[spec=gui], priority=LOW]]""");
```

The Methods of `NavigableSet`

The methods defined by the `NavigableSet` interface fall into six groups.

Retrieving the comparator

```
Comparator<? super E> comparator()  return the Comparator, if the NavigableSet has one; otherwise  
                                         return null
```

This is the method to retrieve the set's comparator, if it has been given one at construction time. If the set uses the natural ordering of its elements, this method returns `null`. The reason that the bounded type can be used for the `Comparator` parameter is because a `NavigableSet` parameterized on `E` can rely for ordering on a `Comparator` defined on any supertype of `E`. For example, recalling “[A Fruity Example](#)” on page 41, a `Comparator<Fruit>` could be used for a `NavigableSet<Apple>`.

Inspecting the first and last elements

```
E first()   return the first element in the set  
E last()    return the last element in the set
```

These are the methods for inspecting the elements at either end of a `SequencedSet` (see [Table 12-1 in “NavigableSet now extends SequencedSet” on page 199](#) for the correspondence with `SequencedSet` methods). If the set is empty, these operations throw `NoSuchElementException`.

Removing the first and last elements

```
E pollFirst()  retrieve and remove the first (lowest) element, or return null if this set is empty  
E pollLast()   retrieve and remove the last (highest) element, or return null if this set is empty
```

These are the methods for removing elements from either end of a `SequencedSet` (see [Table 12-1](#) for the correspondence with `SequencedSet` methods). They are analogous to the methods of the same names in `Deque` (see “[Deque](#)” on page 224) and help to support the use of `NavigableSet` in applications that require queue functionality. For example, in the version of the to-do manager in this section, we could get the highest-priority task off the list, ready to be carried out, by means of this:

[org/jcbook/chapter12/C_navigable_set/Program_2](#)

```
PriorityTask nextTask = priorityTasks.pollFirst();  
assert nextTask.toString().equals(  
    "PriorityTask[task=PhoneTask[name=Paul, number=123 4567], priority=HIGH]");
```

Getting range views

When you have to work with an ordered set of values, a useful way to view them is as a *range*. For example, given a set of timestamped events, you might like to inspect all those that happened within a certain time period. In the case of `PriorityTasks`, we might want to process all those that fall within a range of priorities—high and medium, say. You can always extract a range by iterating over the set and collecting the elements that fall within it, but a data type like `NavigableSet` can make this much more convenient by providing methods that can extract ranges in a single operation.

An interval such as a range view can be *open*, *half-open*, or *closed*, depending on how many of its limit points it contains. For example, the range of numbers x for which $0 \leq x \leq 1$ is closed, because it contains both limit points 0 and 1. The ranges $0 \leq x < 1$ and $0 < x \leq 1$ are half-open because they contain only one of the limit points, and the range $0 < x < 1$ is open because it contains neither.

This preliminary explanation is necessary because each of the methods in this group appears in two overloads, one inherited from `SortedSet` that returns a half-open `SortedSet` view, and one defined in `NavigableSet` returning a `NavigableSet` view that can be open, half-open, or closed according to the user's choice.

<code>SortedSet<E> subSet(E fromValue, E toValue)</code>	return a view of the portion of this set ranging from <code>fromValue</code> , inclusive, to <code>toValue</code> , exclusive
<code>SortedSet<E> headSet(E toValue)</code>	return a view of the portion of this set up to but excluding <code>toValue</code>
<code>SortedSet<E> tailSet(E fromValue)</code>	return a view of the portion of this set whose elements are greater than or equal to <code>fromValue</code>

The `SortedSet` methods return a view of the set elements starting from `fromValue`—including it if it is present—and up to but excluding `toValue`. Notice that although the Javadoc for these methods calls the arguments to these operations “elements”—`fromElement` and `toElement`—this is misleading; in fact, they do not themselves have to be members of the set.

<code>NavigableSet<E> subSet(E fromValue, boolean fromInclusive, E toValue, boolean toInclusive)</code>	return a view of the portion of this set ranging from <code>fromValue</code> to <code>toValue</code>
<code>NavigableSet<E> headSet(E toValue, boolean inclusive)</code>	return a view of the portion of this set up to <code>toValue</code>
<code>NavigableSet<E> tailSet(E fromValue, boolean inclusive)</code>	return a view of the portion of this set from <code>fromValue</code>

The `NavigableSet` methods are similar but allow you to specify for each bound whether it should be included or excluded. In our example, these methods could be

useful in providing different views of the elements in `priorityTasks`. For instance, we can use `headSet` to obtain a view of the high- and medium-priority tasks. To do this, we need a special task that comes before all others in the task ordering; fortunately, we defined an `EmptyTask` class for just this purpose in “[Using the Methods of Collection](#)” on page 168. Using this, it’s easy to extract all tasks that come before any low-priority task:

[org/jcbook/chapter12/C_navigable_set/Program_3](#)

```
PriorityTask firstLowPriorityTask = new PriorityTask(new EmptyTask(), Priority.LOW);

NavigableSet<PriorityTask> highAndMediumPriorityTasks =
    priorityTasks.headSet(firstLowPriorityTask, false);

assert(highAndMediumPriorityTasks.toString()).equals("""
    [PriorityTask[task=PhoneTask[name=Paul, number=123 4567], priority=HIGH],
     PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
     PriorityTask[task=PhoneTask[name=Mike, number=987 6543], priority=MEDIUM]]""");
```

In fact, because we know that tasks with empty details will never normally occur, we can also use one as the first endpoint in a half-open interval:

[org/jcbook/chapter12/C_navigable_set/Program_4](#)

```
PriorityTask firstMediumPriorityTask =
    new PriorityTask(new EmptyTask(), Priority.MEDIUM);

NavigableSet<PriorityTask> mediumPriorityTasks =
    priorityTasks.subSet(firstMediumPriorityTask, true, firstLowPriorityTask, false);

assert (mediumPriorityTasks.toString()).equals("""
    [PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
     PriorityTask[task=PhoneTask[name=Mike, number=987 6543], priority=MEDIUM]]""");
```

All changes in the underlying set are reflected in the view:

[org/jcbook/chapter12/C_navigable_set/Program_5](#)

```
PriorityTask logicCodeMedium = new PriorityTask(logicCode, Priority.MEDIUM);
priorityTasks.add(logicCodeMedium);
assert mediumPriorityTasks.toString().equals("""
    [PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
     PriorityTask[task=CodingTask[spec=logic], priority=MEDIUM],
     PriorityTask[task=PhoneTask[name=Mike, number=987 6543], priority=MEDIUM]]""");
```

To understand how this works, think of all the possible values in an ordering as lying on a line, like the number line used in arithmetic. A range is defined as a fixed segment of that line, regardless of which values are actually in the original set. So a subset defined on a `NavigableSet` and a range will allow you to work with whichever elements of the `NavigableSet` currently lie within the range.

The converse also applies; changes in the view—including structural changes—are reflected in the underlying set:

```
assert priorityTasks.size() == 5;
mediumPriorityTasks.remove(logicCodeMedium);
assert priorityTasks.size() == 4;
```

Getting closest matches

- E `ceiling(E e)` return the least element x in this set such that $x \geq e$, or `null` if there is no such element
- E `floor(E e)` return the greatest element x in this set such that $x \leq e$, or `null` if there is no such element
- E `higher(E e)` return the least element x in this set such that $x > e$, or `null` if there is no such element
- E `lower(E e)` return the greatest element x in this set such that $x < e$, or `null` if there is no such element

These methods are useful for short-distance navigation. For example, suppose that we want to find, in a sorted set of strings, the last three strings in the subset that is bounded above by "x-ray", including that string itself if it is present in the set. `NavigableSet` methods make this easy:

```
NavigableSet<String> stringSet = new TreeSet<>();
Collections.addAll(stringSet, "abc", "cde", "x-ray", "zed");
Optional<String> last = Optional.ofNullable(stringSet.floor("x-ray"));
assert last.equals(Optional.of("x-ray"));
Optional<String> secondToLast = last.map(stringSet::lower);
assert secondToLast.equals(Optional.of("cde"));
Optional<String> thirdToLast = secondToLast.map(stringSet::lower);
assert thirdToLast.equals(Optional.of("abc"));
```

Notice that, in line with a general trend in the design of the Collections Framework, `NavigableSet` returns `null` values to signify the absence of elements where, for example, the `first` and `last` methods of `SortedSet` would throw `NoSuchElementException`. For this reason, you should avoid `null` elements in `NavigableSets`, and in fact the newer implementation, `ConcurrentSkipListSet`, does not permit them (though `TreeSet` must continue to do so, for backward compatibility).

Navigating the set in reverse order

- `NavigableSet<E> descendingSet()` return a reverse-order view of the elements in this set
- `Iterator<E> descendingIterator()` return a reverse-order iterator

Methods of this group make traversing a `NavigableSet` equally easy in the descending (that is, reverse) ordering. As a simple illustration, let's generalize the previous example using the nearest-match methods. Suppose that, instead of finding just the

last three strings in the sorted set bounded above by "x-ray", we want to iterate over all the strings in that set, in descending order:

[org/jcbook/chapter12/C_navigable_set/Program_8](#)

```
NavigableSet<String> headSet = stringSet.headSet(last.get(), true);
NavigableSet<String> reverseHeadSet = headSet.descendingSet();
assert reverseHeadSet.toString().equals("[x-ray, cde, abc]");
String conc = " ";
for (String s : reverseHeadSet) {
    conc += s + " ";
}
assert conc.equals(" x-ray cde abc ");
```

If the iterative processing involves structural changes to the set, and the implementation being used is TreeSet (which has fail-fast iterators), we will have to use an explicit iterator to avoid a ConcurrentModificationException:

[org/jcbook/chapter12/C_navigable_set/Program_9](#)

```
for (Iterator<String> itr = headSet.descendingIterator(); itr.hasNext(); ) {
    itr.next(); itr.remove();
}
assert headSet.isEmpty();
```

NavigableSet now extends SequencedSet

Although NavigableSet has been retrofitted to extend SequencedSet, none of the new methods provide any different functionality; they are just renamed versions of existing methods. [Table 12-1](#) shows the correspondence between the new and existing methods.

Table 12-1. SequencedSet methods compared with their NavigableSet equivalents

SequencedSet	NavigableSet
getFirst	first (inherited from SortedSet)
getLast	last (inherited from SortedSet)
removeFirst	pollFirst
removeLast	pollLast
addFirst	Unsupported method for internally ordered collections
addLast	Unsupported method for internally ordered collections
reversed	descendingSet

The reason for the duplication between methods of SequencedSet and NavigableSet is that the first six SequencedSet methods in [Table 12-1](#) were copied from Deque. Prior to the introduction of SequencedSet in Java 21, SortedSet and NavigableSet had several methods that were similar to the Deque methods, but with different names and somewhat different behavior:

- `NavigableSet::first` and `NavigableSet::last`, inherited from `SortedSet`, are the same as `SequencedSet::getFirst` and `SequencedSet::getLast`, throwing `NoSuchElementException` if the collection is empty.
- `NavigableSet::pollFirst` and `NavigableSet::pollLast` remove and return the respective element. However, they differ from `SequencedSet::removeFirst` and `SequencedSet::removeLast` in that the poll methods return `null` on an empty collection instead of throwing `NoSuchElementException`.

TreeSet

Let's take a little time now to consider how trees perform in comparison to the other implementation types used by the Collections Framework.

Trees are the data structure you would choose for an application that needs fast insertion and retrieval of individual elements, but which also requires that elements can be retrieved in sorted order.

For example, suppose you want to match all the words from a set against a given prefix—a common requirement in visual applications where a drop-down should ideally show all the possible elements that match against the prefix that the user has typed. A hash table can't return its elements in sorted order and a list can't retrieve its elements quickly by their content, but a tree can do both.

In computer science, a tree is a branching structure that represents hierarchy. Computing trees borrow a lot of their terminology from genealogical trees, though there are some differences; the most important is that in computing trees, each node has only one parent (except the root, which has none). An important class of tree often used in computing is a *binary tree*—one in which each node can have at most two children. [Figure 12-4](#) shows an example of a binary tree containing the words of this sentence in alphabetical order.

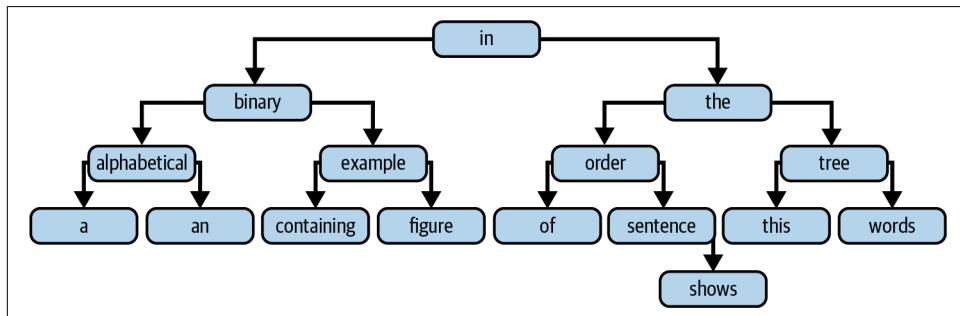


Figure 12-4. An ordered, balanced binary tree

The most important property of this tree can be seen if you look at any nonleaf node—say, the one containing the word *the*: all the nodes below that on the left contain words that precede *the* alphabetically, and all those on the right contain words that follow it. To locate a word, you would start at the root and descend level by level, doing an alphabetic comparison at each level, so the cost of retrieving or inserting an element is proportional to the depth of the tree.

How deep, then, is a tree that contains N elements? The complete binary tree with two levels has three elements (that's $2^2 - 1$), and the one with three levels has seven elements ($2^3 - 1$). In general, a binary tree with N complete levels will have $2^N - 1$ elements, and the depth of a tree with N elements will be bounded by $\log N$ (since $2^{\log N} = N$). Just as N grows much more slowly than 2^N , $\log N$ grows much more slowly than N , so `contains` on a large tree is much faster than on a list containing the same elements. It's still not as fast as a hash table—whose operations can ideally work in constant time—but a tree has the big advantage over a hash table in that its iterator can return its elements in sorted order.

Not all binary trees will have this nice performance, though. [Figure 12-4](#) shows a *balanced* binary tree—one in which each node has an equal number of descendants (or as near as possible) on each side. An unbalanced tree can give much worse performance—in the worst case, as bad as a linked list (see [Figure 12-5](#)). `TreeSet` uses a data type called a *red-black tree* (see, for example, [Sedgewick and Wayne 2011](#)), which has the advantage that if it becomes unbalanced through insertion or removal of an element, it can always be rebalanced in $O(\log N)$ time.

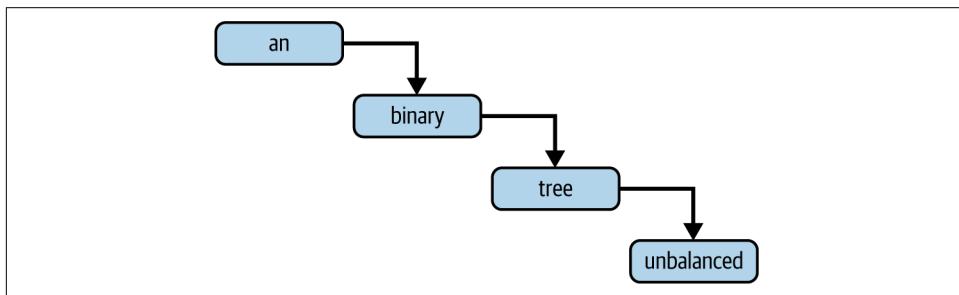


Figure 12-5. An unbalanced binary tree

The constructors for `TreeSet` include, besides the standard ones, one that allows you to supply a `Comparator` (see “[Comparator](#)” on page 45) and one that allows you to create one from another `SortedSet`:

<code>TreeSet(Comparator<? super E> c)</code>	construct an empty set that will be sorted using the specified comparator
<code>TreeSet(SortedSet<E> s)</code>	construct a new set containing the elements of the supplied set, sorted according to the same ordering

The second of these is rather too close in its declaration to the standard “conversion constructor” (see “[Collection Constructors](#)” on page 176):

`TreeSet(Collection<? extends E> c)` construct a new set containing the elements of the supplied set, sorted according to the natural ordering

As Joshua Bloch explains in *Effective Java* (2017, item 52), calling one of two constructor or method overloads that take parameters of related type can give confusing results. This is because, in Java, calls to overloaded constructors and methods are resolved at compile time on the basis of the static type of the argument, so applying a cast to an argument can make a big difference to the result of the call, as the following code shows:

[org/jcbook/chapter12/C_navigable_set/Program_10](#)

```
// construct and populate a NavigableSet whose iterator returns its
// elements in the reverse of natural order:
NavigableSet<String> base = new TreeSet<>((Comparator.reverseOrder()));
Collections.addAll(base, "b", "a", "c");

// call the two different constructors for TreeSet, supplying the
// set just constructed, but with different static types:
NavigableSet<String> sortedSet1 = new TreeSet<>((Set<String>)base);
NavigableSet<String> sortedSet2 = new TreeSet<>(base);
// and the two sets have different iteration orders:
List<String> forward = new ArrayList<>(sortedSet1);
List<String> backward = new ArrayList<>(sortedSet2);
assert !forward.equals(backward);
assert forward.reversed().equals(backward);
```

This problem afflicts the constructors for all the sorted collections in the framework (`TreeSet`, `TreeMap`, `ConcurrentSkipListSet`, and `ConcurrentSkipListMap`). To avoid it in your own class designs, choose parameter types for different overloads such that an argument of a type appropriate to one overload cannot be cast to the type appropriate to a different one. If that is not possible, the two overloads should be designed to behave identically with the same argument, regardless of its static type. For example, a `PriorityQueue` (see “[PriorityQueue](#)” on page 211) constructed from a collection uses the ordering of the original, whether the static type with which the constructor is supplied is one of the `Comparator`-containing types `PriorityQueue` or `SortedSet`, or just a plain `Collection`. To achieve this, the conversion constructor uses an `instanceof` test to determine the type of the supplied collection; if the type is `SortedSet` or `PriorityQueue`, it extracts the `Comparator`, falling back on natural ordering if there isn’t one.

`TreeSet` is unsynchronized and not thread-safe; its iterators are fail-fast.

ConcurrentSkipListSet

ConcurrentSkipListSet was the first concurrent set implementation. It is backed by a *skip list*, a modern alternative to the binary trees of the previous section. A skip list for a set is a series of *linked lists*, each of which is a chain of cells consisting of two fields: one to hold a value, and one to hold a reference to the next cell. Elements are inserted into and removed from a linked list in constant time by pointer rearrangement, as shown in Figure 12-6, parts (a) and (b), respectively.

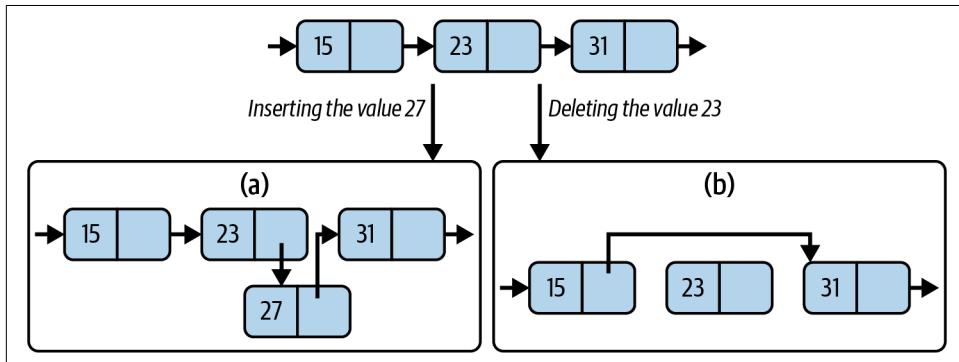


Figure 12-6. Modifying a linked list

Figure 12-7 shows a skip list consisting of three linked lists, labeled levels 0, 1, and 2. The first linked list of the collection (level 0 in the figure) contains the elements of the set, sorted according to their natural order or by the comparator of the set. Each list above level 0 contains a subset of the list below it, chosen randomly according to a fixed probability. For this example, let's suppose that the probability is 0.5; on average, each list will contain half the elements of the list below it. Navigating between links takes a fixed time, so the quickest way to find an element is to start at the beginning (the lefthand end) of the top list and to go as far as possible in each list before dropping to the one below it.

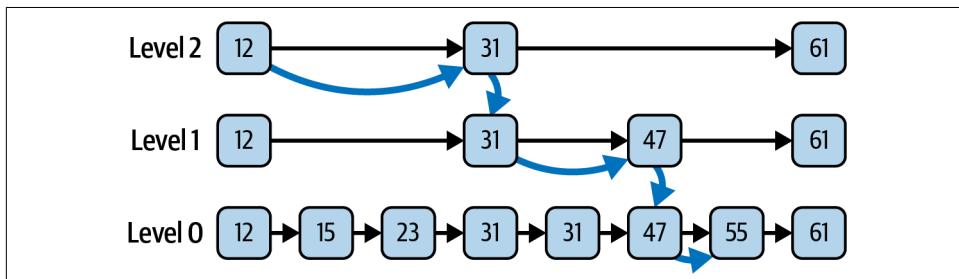


Figure 12-7. Searching a skip list

The curved arrows in [Figure 12-7](#) show the progress of a search for the element 55. The search starts with the element 12 at the top left of level 2, steps to the element 31 on that level, then finds that the next element is 61, higher than the search value. So it drops one level, and then repeats the process; element 47 is still smaller than 55, but 61 is again too large, so it once more drops a level and finds the search value in one further step.

Inserting an element into a skip list always involves at least inserting it at level 0. When that has been done, should it also be inserted at level 1? If level 1 contains, on average, half of the elements at level 0, then we should toss a coin (that is, randomly choose with probability 0.5) to decide whether it should be inserted at level 1 as well. If the coin toss does result in it being inserted at level 1, then the process is repeated for level 2, and so on. When we remove an element from a skip list, it is removed from each level in which it occurs.

If the coin tossing goes badly, we could end up with every list above level 0 empty—or full, which would be just as bad. These outcomes have very low probability, however, and analysis shows that, in fact, the probability is very high that skip lists will give performance comparable to binary trees: search, insertion, and removal all have complexity of $O(\log N)$. Their compelling advantage for concurrent use is that they have efficient lock-free insertion and deletion algorithms, whereas there are none known for binary trees.

The iterators of `ConcurrentSkipListSet` are weakly consistent.

Comparing Set Implementations

[Table 12-2](#) shows the comparative performance of the different `Set` implementations. When you are choosing an implementation, efficiency is only one of the factors you should take into account. Some of these implementations are specialized for specific situations; for example, `EnumSet` should always (and only) be used to represent sets of enum values. Similarly, `CopyOnWriteArraySet` should only be used where set size will remain relatively small, read operations greatly outnumber writes, thread safety is required, and read-only iterators are acceptable.

Table 12-2. Comparative performance of different Set implementations

	<code>add</code>	<code>contains</code>	<code>next</code>	Notes
<code>HashSet</code>	$O(1)$	$O(1)$	$O(h/M)$	h is the table capacity
<code>LinkedHashSet</code>	$O(1)$	$O(1)$	$O(1)$	
<code>CopyOnWriteArraySet</code>	$O(N)$	$O(N)$	$O(1)$	
<code>EnumSet</code>	$O(1)$	$O(1)$	$O(1)^a$	
<code>TreeSet</code>	$O(\log N)$	$O(\log N)$	$O(\log N)$	

	add	contains	next	Notes
ConcurrentSkipListSet	$O(\log N)$	$O(\log N)$	$O(1)$	

^a In the `EnumSet` implementation for enum types with more than 64 values, `next` has worst-case complexity of $O(\log m)$, where m is the number of elements in the enumeration.

That leaves the general-purpose implementations: `HashSet`, `LinkedHashSet`, `TreeSet`, `ConcurrentSkipListSet`, and the set view of `ConcurrentHashMap` obtained by `newSetFromMap`.

The first three are mainly for use in single-threaded applications. They are not thread-safe, so they can only be used in multithreaded code either in conjunction with client-side locking or wrapped in `Collection.synchronizedSet` (see “[Synchronized Collections](#)” on page 281). When there is no requirement for the set to be sorted, your choice is between `HashSet` and `LinkedHashSet`. If your application will be frequently iterating over the set, `LinkedHashSet` is the implementation of choice. If the set needs to support the methods of `NavigableSet`, use `TreeSet`.

In a multithreaded environment, the choice is between the set view provided by `ConcurrentHashMap::newSetFromMap`, and `ConcurrentSkipListSet`. The first of these is the default choice, on efficiency grounds, but the second supports the methods of `NavigableSet`.

Conclusion

In this chapter, we saw a variety of different ways in which the Collections Framework implements the deceptively simple abstraction of an unordered collection without duplicates. These implementations reflect the differing requirements of the various scenarios in which sets are useful in practical programming.

The subject of the next chapter, `Queue`, is useful in a different way from `Set` and the other major collection classes. Queues are conduits used by objects to communicate rather than to store data. They do not form part of the state of other objects in the way that the other collections usually do; we will see how this shapes the API for `Queue`.

Queues

A *queue* is a collection designed to hold elements for processing, yielding them up in the order in which they are to be processed. The corresponding Collections Framework interface `Queue<E>` has a number of different implementations embodying different rules about what this order should be. Many of the implementations use the rule that tasks are to be processed in the order in which they were submitted (*first in, first out*, or *FIFO*), but other rules are possible—for example, the Collections Framework includes queue classes whose processing order is based on task priority. The `Queue` interface was a later addition to the framework, motivated in part by the need for queues in the concurrency utilities included at the same time. A glance at the hierarchy of implementations shown in [Figure 13-1](#) shows that, in fact, nearly all the `Queue` implementations in the Collections Framework are in the package `java.util.concurrent`.

One classic requirement for queues in concurrent systems arises when a number of tasks have to be executed by a number of threads working in parallel. An everyday example of this situation is that of a single queue of airline passengers being handled by a line of check-in operators. Each operator works on processing a single passenger (or a group of passengers) while the remaining passengers wait in the queue. As they arrive, passengers join the *tail* of the queue and wait until they reach its *head*, when they are assigned to the next operator who becomes free. A good deal of fine detail is involved in implementing a queue such as this; operators have to be prevented from simultaneously attempting to process the same passenger, empty queues have to be handled correctly, and in computer systems there has to be a way of defining queues with a maximum size, or *bound*. (This last requirement may not often be imposed in airline terminals, but it can be very useful in systems in which there is a maximum waiting time for a task to be executed.) The `Queue` implementations in `java.util.concurrent` look after these implementation details for you.

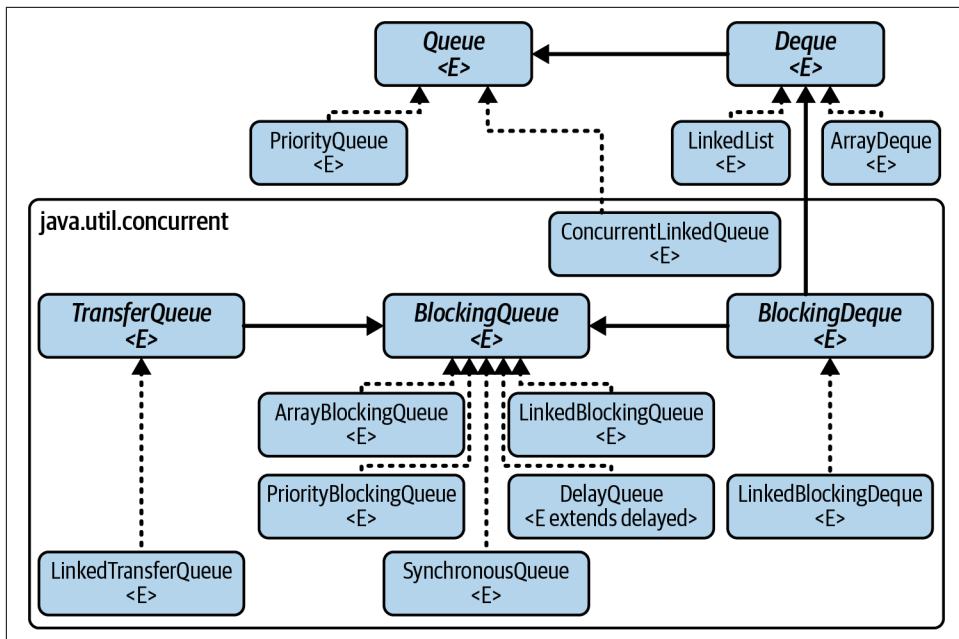


Figure 13-1. Implementations of Queue in the Collections Framework



The code examples for this chapter can be found at:

https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter13

Queue Interface Methods

In addition to the operations inherited from `Collection`, the `Queue` interface includes operations to add an element to the tail of the queue, to inspect the element at its head, and to remove the element at its head. Each of these operations comes in two varieties: one that returns a `null` value to indicate failure and one that throws an exception.

Adding an Element to a Queue

`boolean offer(E e)` insert the given element if possible

The exception-throwing variant of this operation is the `add` method inherited from `Collection`. Although `add` does return a `boolean` signifying its success in inserting an element, that value can't be used to report that a bounded queue is full. The contract

for `add` specifies that it may return `false` only if the reason for refusing the element is that it was already present—otherwise, it must throw an (unchecked) exception. For a bounded queue, the value-returning variant `offer` is usually a better option.

The value returned by `offer` indicates whether the element was successfully inserted. Note that `offer` does throw an exception if the element is illegal in some way (for example, if the value `null` is submitted to a queue that doesn't permit `nulls`). Normally, if `offer` returns `false`, it has been called on a bounded queue that has reached capacity.

Retrieving an Element from a Queue

The methods in this group are `peek` and `element` for inspecting the head element, and `poll` and `remove` for removing it from the queue and returning its value.

The methods that throw an exception for an empty queue are:

- E `element()` retrieve but do not remove the head element
- E `remove()` retrieve and remove the head element

Notice that this is a different method from the `Collection` method `remove(Object)`. The methods that return `null` for an empty queue are:

- E `peek()` retrieve but do not remove the head element
- E `poll()` retrieve and remove the head element

Because these methods return `null` to signify that the queue is empty, you should avoid using `null` as a queue element. In general, the use of `null` as a queue element is discouraged by the `Queue` interface; in the JDK, the only implementation that allows it is the legacy class `LinkedList`.

Using the Methods of Queue

Let's look at some examples of the use of these methods. Queues should provide a good way of implementing a task manager, since their main purpose is to yield up elements, such as tasks, for processing. For the moment we'll use `ArrayDeque`, as this is the fastest and most straightforward implementation of `Queue` (and `Deque`). As before, we'll confine ourselves to the methods of the interface—but remember that in choosing a `Queue` implementation, you're also choosing the ordering of task processing. With `ArrayDeque`, you get FIFO ordering; this is a good choice for the

task manager, which, in its initial version, doesn't support assigning priorities to tasks. (A later version, in [Example 13-1](#), will use a priority queue.)

`ArrayDeque` is unbounded, so we could use either `add` or `offer` to set up the queue with new tasks:

```
Queue<Task> taskQueue = new ArrayDeque<>();
taskQueue.offer(mikePhone);
taskQueue.add(paulPhone);
```

Anytime we feel ready to do a task, we can take the one that has reached the head of the queue:

```
Task nextTask = taskQueue.poll();
if (nextTask != null) {
    // process nextTask
}
```

The choice between using `poll` and `remove` depends on whether we want to regard queue emptiness as an exceptional condition. Realistically—given the nature of the application—that might be a sensible assumption, so this is an alternative:

```
try {
    Task nextTask = taskQueue.remove();
    // process nextTask
} catch (NoSuchElementException e) {
    // but we *never* run out of tasks!
}
```

This scheme needs some refinement to allow for the nature of different kinds of tasks. Phone tasks fit into relatively short time slots, whereas we don't like to start coding unless there is reasonably substantial time available to get into the task. So if time is limited—say, until the next meeting—we might like to check that the next task is of the right kind before we take it off the queue:

```
Task nextTask = taskQueue.peek();
if (nextTask instanceof PhoneTask) {
    taskQueue.remove();
    // process nextTask
}
```

These inspection and removal methods are a major benefit of the `Queue` interface; `Collection` has nothing like them (though `NavigableSet` does). The price we pay for this benefit is that the methods of `Queue` are useful to us only if the head element is actually one that we want. True, the class `PriorityQueue` allows us to provide a comparator that will order the queue elements so that the one we want is at the head, but that may not be a particularly good way of expressing the algorithm for choosing the next task. For example, it might help to know something about *all* the outstanding tasks before you choose the next one. Otherwise, in a situation of limited time with an entirely queue-based to-do manager, you might end up going for coffee until the meeting starts. As an alternative, you could consider using the `List`

interface, which provides more flexible means of accessing its elements but has the drawback that its implementations provide much less support for multithread use.

This might sound overly pessimistic; after all, `Queue` is a subinterface of `Collection`, so it inherits methods that support traversal, like `iterator`. In fact, although these methods are implemented, their use is not recommended in normal situations. In the design of the queue classes, efficiency in traversal has been traded against fast implementation of the methods of `Queue`; in addition, queue iterators do not guarantee to return their elements in proper sequence and, for some concurrent queues, will actually fail in normal conditions (see “[BlockingQueue Implementations](#)” on page 219).

Queue Implementations

In this section, we will look at the two direct JDK implementations of `Queue`, `PriorityQueue`, and `ConcurrentLinkedQueue`. Then, in “[BlockingQueue](#)” on page 215, we’ll look at `BlockingQueue` and its implementations. These classes differ widely in their behavior. Most of them are thread-safe, and most provide *blocking* facilities (that is, operations that wait for conditions to be right for them to execute). Some support priority ordering, while one—`DelayQueue`—holds elements until their delay has expired and another—`SynchronousQueue`—is purely a synchronization facility. In choosing between `Queue` implementations, you’ll typically be influenced more by these functional differences than by their performance.

PriorityQueue

`PriorityQueue` is one of the two nonlegacy `Queue` implementations (that is, other than `LinkedList`) not designed primarily for concurrent use (the other one is `ArrayDeque`). It is not thread-safe, nor does it provide blocking behavior. It gives up its elements for processing according to an ordering like that used by `NavigableSet`—either the natural order of its elements if they implement `Comparable`, or the ordering imposed by a `Comparator` supplied when the `PriorityQueue` is constructed. It would therefore be an alternative design choice (obviously, given its name) for the priority-based to-do manager that we outlined in “[NavigableSet](#)” on page 193 using `NavigableSet`. Your application will dictate which alternative to choose: if it needs to examine and manipulate the set of waiting tasks, use `NavigableSet`; if its main requirement is efficient access to the next task to be performed, use `PriorityQueue`.

Choosing `PriorityQueue` allows us to reconsider the ordering: since it accommodates duplicates, it does not share the requirement of `NavigableSet` for an ordering consistent with `equals`. To emphasize this point, we will define a new ordering for our to-do manager that depends only on priorities. Contrary to what you might expect, `PriorityQueue` gives no guarantee of how it presents multiple elements with

the same value. So if, in our example, several tasks are tied for the highest priority in the queue, it will choose one of them arbitrarily as the head element.

The constructors for `PriorityQueue` are:

<code>PriorityQueue()</code>	natural ordering, default initial capacity
<code>PriorityQueue(Collection<? extends E> c)</code>	if <code>c</code> is a <code>PriorityQueue</code> or <code>SortedSet</code> , order according to <code>c</code> 's ordering; otherwise, use the natural ordering of <code>c</code> 's elements
<code>PriorityQueue(int initialCapacity)</code>	natural ordering, specified initial capacity
<code>PriorityQueue(int initialCapacity, Comparator<? super E> comparator)</code>	Comparator ordering, specified initial capacity
<code>PriorityQueue(PriorityQueue<? extends E> c)</code>	ordering and elements copied from <code>c</code>
<code>PriorityQueue(SortedSet<? extends E> c)</code>	ordering and elements copied from <code>c</code>

Notice how the second of these constructors avoids the problem of the overloaded `TreeSet` constructor that we discussed in “[TreeSet](#) on page 200”. Using the ordering of a supplied `PriorityQueue` or `SortedSet` allows this constructor to take advantage of the binary heap implementation to perform a bulk copy of the contents of the supplied collection, avoiding the need to build the heap element by element. In the case of the constructor taking a `PriorityQueue`, this works because the source already contains a properly arranged heap. In the case of the constructor taking a `SortedSet`, it first calls `SortedSet::toArray`, which produces an array that is already a properly arranged heap.

We can use `PriorityQueue` for a simple implementation of the to-do manager with the `PriorityTask` class defined in “[NavigableSet](#) on page 193” and a new `Comparator` depending only on the task's priority:

[org/jgcbook/chapter13/B_implementing_queue/Program_1](#)

```
final int INITIAL_CAPACITY = 10;
Comparator<PriorityTask> priorityComp = Comparator.comparing(PriorityTask::priority);
Queue<PriorityTask> priorityQueue =
    new PriorityQueue<>(INITIAL_CAPACITY, priorityComp);
priorityQueue.add(new PriorityTask(mikePhone, Priority.MEDIUM));
priorityQueue.add(new PriorityTask(paulPhone, Priority.HIGH));
PriorityTask nextTask = priorityQueue.poll();
System.out.println(nextTask);
...
nextTask = priorityQueue.poll();
```

Priority queues are usually efficiently implemented by *priority heaps*. A priority heap is a binary tree somewhat like those we saw implementing `TreeSet` in “[TreeSet](#) on page 200”, but with two differences. First, the only ordering constraint is that each

node in the tree should be ordered with respect to its children: either smaller, in the case of a *min heap* (which is Java's default for naturally ordered elements), or larger, in the case of a *max heap*. Second, the tree should be complete at every level except possibly the lowest; if the lowest level is incomplete, the nodes it contains must be grouped together at the left. [Figure 13-2\(a\)](#) shows a small min heap, with each node shown only by the field containing its priority. To add a new element to a priority heap, it is first attached at the leftmost vacant position, as shown by the darker node in [Figure 13-2\(b\)](#). Then it is repeatedly exchanged with its parent until it reaches a parent that has higher priority—that is, has a smaller value. In the figure, this required only a single exchange of the new element with its parent, giving [Figure 13-2\(c\)](#). (Nodes shown darker in Figures 13-2 and 13-3 have just changed position.)

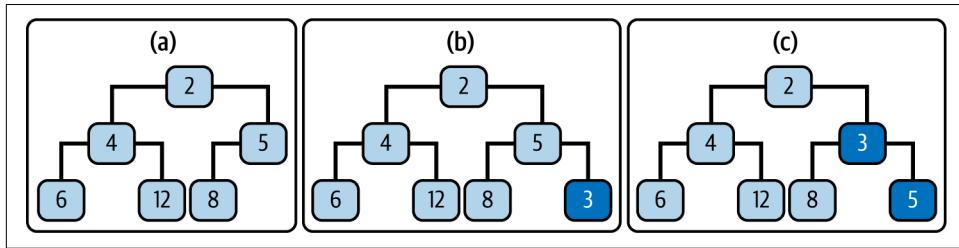


Figure 13-2. Adding an element to a PriorityQueue

Getting the highest-priority element from a priority heap is trivial: it is the root of the tree. But when that has been removed, the two separate trees that result must be reorganized into a priority heap again. This is done by first placing the rightmost element from the bottom row into the root position. Then—in the reverse of the procedure for adding an element—it is repeatedly exchanged with the smaller of its children until it has a higher priority than either, or until it has become a leaf. [Figure 13-3](#) shows the process—again requiring only a single exchange—starting from the heap in [Figure 13-2\(c\)](#) after the head has been removed.

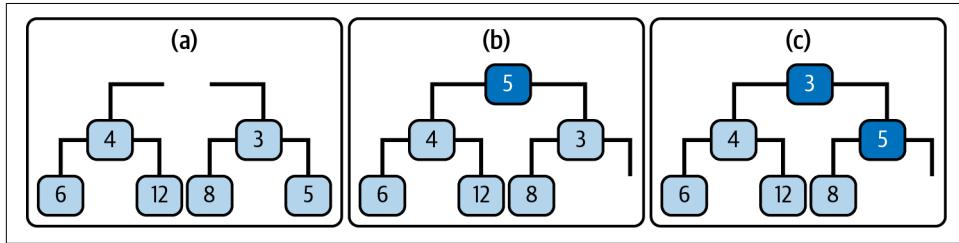


Figure 13-3. Removing the head of a PriorityQueue

Apart from constant overheads, both addition and removal of elements require a number of operations proportional to the height of the tree. So `PriorityQueue` provides $O(\log N)$ performance for `offer`, `poll`, parameterless `remove`, and `add`. The methods `remove(Object)` and `contains` may require the entire tree to be traversed, so they require $O(N)$ time. The methods `peek` and `element`, which just retrieve the root of the tree without removing it, take constant time, as does `size`, which uses an object field that is continually updated.

`PriorityQueue` is not intended for concurrent use. The framework offers a thread-safe version instead: `PriorityBlockingQueue` (see “[PriorityBlockingQueue](#)” on page 221). `PriorityQueue`’s iterators are fail-fast. The results of iterating over it may be surprising until you understand the implementation: the elements are stored in an array, row by row, left to right across each row. Iterators simply traverse the array, so, as you can see from the diagrams, the order in which elements are presented is unrelated to the order imposed by the comparison method. This priority order can only be revealed by removing the elements one at a time.

ConcurrentLinkedQueue

As [Figure 13-1](#) shows, most queue implementations are blocking queues, which we explore in the next section. The exceptions are `PriorityQueue`, discussed in the previous section, and `ConcurrentLinkedQueue`. This is an unbounded, thread-safe, FIFO-ordered queue. It uses a linked structure, similar to those we saw in “[ConcurrentSkipListSet](#)” on page 203 as the basis for skip lists and in “[HashSet](#)” on page 183 for hash table overflow chaining. We noted there that one of the main attractions of linked structures is that the insertion and removal operations implemented by pointer rearrangements are performed in constant time. This makes them especially useful as FIFO queue implementations, where these operations are always required on nodes at the ends of the structure—that is, nodes that do not need to be located using the slow sequential search of linked structures.

`ConcurrentLinkedQueue` uses a CAS-based (see “[Concurrent Collections](#)” on page 159) *wait-free* algorithm—that is, one that guarantees that every thread will make progress over time, regardless of the state of other threads accessing the queue. It executes queue insertion and removal operations in constant time, but requires linear time to execute `size`. This is because the algorithm, which relies on cooperation between threads for insertion and removal, does not keep track of the queue size and has to iterate over the queue to calculate it when it is required.

`ConcurrentLinkedQueue` has the two standard constructors discussed in “[Collection Constructors](#)” on page 176. Its iterators are weakly consistent.

BlockingQueue

The classes in the Collections Framework intended to support concurrent applications are mostly implementations of the Queue subinterface `BlockingQueue<E>`, designed primarily for use in producer/consumer scenarios.

One common example of the use of producer/consumer queues is in systems that perform print spooling: client processes add print jobs to the spool queue to be processed by one or more print service processes, each of which repeatedly consumes the task at the head of the queue. The key facilities that `BlockingQueue` provides to such systems are, as its name implies, enqueueing and dequeuing methods that do not return until they have executed successfully. So in this example, a print server does not need to constantly poll the queue to discover whether any print jobs are waiting; it need only call the `poll` method, supplying a timeout, and the system will suspend it until either a queue element becomes available or the timeout expires.

The Methods of BlockingQueue

`BlockingQueue` defines seven new methods, in three groups.

Adding an element

`boolean offer(E e, long timeout, TimeUnit unit)` insert e, waiting up to the timeout

`void put(E e)` add e, waiting as long as necessary

The methods inherited from `Queue`—`add` and `offer`—fail immediately if called on a bounded queue that has reached capacity: `add` by throwing an exception, `offer` by returning `false`. These blocking methods are more patient: the `offer` overload waits for a time specified using `java.util.concurrent.TimeUnit` (an enum that allows timeouts to be defined in units such as milliseconds or seconds), and `put` will block indefinitely.

Removing an element

`E poll(long timeout, TimeUnit unit)` retrieve and remove the head, waiting up to the timeout

`E take()` retrieve and remove the head, waiting as long as necessary

Taking these methods together with those inherited from `Queue`, the methods for removing an element have the same four possible behaviors in the case of failure as those for adding an element. In this case, failure will typically be caused by the queue currently containing no elements. The inherited `poll` method returns `null` in this

situation, whereas blocking `poll` returns `null` but only after waiting until its timeout; the inherited `remove` method throws an exception immediately, and `take` blocks until it succeeds.

Although the behavior of the various `BlockingQueue` methods varies systematically, their naming can be difficult to understand. The clearest picture is provided by the Javadoc ([Table 13-1](#)).

Table 13-1. BlockingQueue methods

	Throws exception	Special value	Blocks	Times out
Insert	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
Remove	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
Examine	<code>element()</code>	<code>peek()</code>	Not applicable	Not applicable

Retrieving or querying the contents of the queue

```
int drainTo(Collection<? super E> c)    clear the queue into c  
int drainTo(Collection<? super E> c, int maxElements)    clear at most the specified number of elements into c  
int remainingCapacity()    return the number of elements that would be accepted without  
                           blocking, or Integer.MAX_VALUE if unbounded
```

The `drainTo` methods perform atomically and efficiently, so the second overload is useful in situations in which you know that you have processing capability available immediately for a certain number of elements, and the first is useful, for example, when all producer threads have stopped working. Their return value is the number of elements transferred. `remainingCapacity` reports the spare capacity of the queue, although as with any such value in multithreaded contexts, the result of a call should not be used as part of a test-then-act sequence: between the test (the call of `remainingCapacity`) and the action (adding an element to the queue) of one thread, another thread might have intervened to add or remove elements.

`BlockingQueue` guarantees that the queue operations of its implementations will be thread-safe and atomic. But this guarantee doesn't extend to the bulk operations inherited from `Collection`—`addAll`, `containsAll`, `retainAll`, and `removeAll`—unless the individual implementation provides it. So it is possible, for example, for `addAll` to fail, throwing an exception, after adding only some of the elements in a collection.

Using the Methods of BlockingQueue

A to-do manager that works for just one person at a time is very limited; we really need a cooperative solution—one that will allow us to share both the production and the processing of tasks. [Example 13-1](#) shows `StoppableTaskQueue`, a simple version of a concurrent task manager based on `PriorityBlockingQueue`, that will allow its users (us) to independently add tasks to the task queue as we discover the need for them, and to take them off for processing as we find the time. The class `StoppableTaskQueue` has methods `addTasks`, `getFirstTask`, and `shutDown`, together with a convenience method `addTask`. A `StoppableTaskQueue` is either working or stopped. The method `addTasks` returns a boolean value indicating whether it successfully added its tasks; this value will be `true` unless the `StoppableTaskQueue` is stopped. The method `getFirstTask` returns the head task from the queue. If no task is available, it does not block but returns `null`. The method `shutDown` stops the `StoppableTaskQueue`, waits until all pending `addTasks` operations are completed, then drains the `StoppableTaskQueue` and returns its contents.

Example 13-1. A concurrent queue-based task manager

[org/jgcbook/chapter13/C_blocking_queue/StoppableTaskQueue](#)

```
public class StoppableTaskQueue {  
    private final int MAXIMUM_PENDING_OFFERS = Integer.MAX_VALUE;  
    private final BlockingQueue<PriorityTask> taskQueue = new PriorityBlockingQueue<>();  
    private final Semaphore semaphore = new Semaphore(MAXIMUM_PENDING_OFFERS);  
    private volatile boolean isStopping;  
  
    // return true if the task was successfully placed on the queue, false  
    // if the queue is being shut down  
    public boolean addTask(PriorityTask task) {  
        return addTasks(List.of(task));  
    }  
  
    // return true if the tasks in the supplied collection were successfully  
    // placed on the queue, false if the queue is being shut down  
    public boolean addTasks(Collection<PriorityTask> tasks) {  
        if (isStopping) return false;  
        if (! semaphore.tryAcquire()) {  
            return false;  
        } else {  
            taskQueue.addAll(tasks);  
            semaphore.release();  
            return true;  
        }  
    }  
  
    // return the head task from the queue, or null if no task is available  
    public PriorityTask getFirstTask() {  
        return taskQueue.poll();  
    }  
}
```

```

// stop the queue, wait for producers to finish, then return the contents
public Collection<PriorityTask> shutDown() {
    isStopping = true;
    // blocks until all outstanding addTasks() calls have completed
    semaphore.acquireUninterruptibly(MAXIMUM_PENDING_OFFERS);
    List<PriorityTask> returnCollection = new ArrayList<>();
    taskQueue.drainTo(returnCollection);
    return returnCollection;
}
}

```

In this example, as in most uses of the `java.util.concurrent` collections, the collection itself takes care of the problems arising from the interaction of different threads in adding items to or removing them from the queue. Most of the code of [Example 13-1](#) is instead solving the problem of providing an orderly shutdown mechanism. The reason for this emphasis is that when we go on to use the class `StoppableTaskQueue` as a component in a larger system, we will need to be able to stop daily task queues without losing task information. Achieving graceful shutdown can often be a problem in concurrent systems; for more detail, see Chapter 7 of [Goetz et al. \(2006\)](#).

The larger system will model each day's scheduled tasks over the next year, allowing consumers to process tasks from each day's queue. An implicit assumption of the example in this section is that if there are no remaining tasks scheduled for this day, a consumer will not wait for one to become available, but will immediately go on to look for a task in the next day's queue. (In the real world, we would go home at this point, or more likely go out to celebrate.) This assumption simplifies the example, as we don't need to invoke any of the blocking methods of `PriorityBlockingQueue`, though we do use one method, `drainTo`, from the `BlockingQueue` interface.

There are a number of ways of shutting down a producer/consumer queue such as this; in the one we've chosen for this example, the manager exposes a `shutdown` method that can be called by a “supervisor” thread in order to stop producers writing to the queue, then drain it, and return the outstanding tasks. The difficulty is in ensuring that `addtasks` performs atomically—that is, it adds all of its tasks or none of them—and that once the `shutdown` method has stopped the queue and is draining or has drained it, no other threads can subsequently gain access to it to add further tasks.

[Example 13-1](#) achieves this using a *semaphore*—a thread-safe object that maintains a fixed number of *permits*. Semaphores are usually used to regulate access to a finite set of resources—a pool of database connections, for example. The permits the semaphore has available at any time represent the resources not currently in use. A thread requiring a resource acquires a permit from the semaphore and releases it when it releases the resource. If all the resources are in use, the semaphore will have

no permits available; at that point, a thread attempting to acquire a permit will block until some other thread returns one.

The semaphore in this example is used differently. We don't want to restrict producer threads from writing to the queue—it's a concurrent queue, after all, quite capable of handling multithread access without help from us. We just want the `shutdown` method to be able to tell if there are any writes currently in progress. So we create the semaphore with the largest possible number of permits, which in practice will never all be required. The producer method `addTasks` first checks the volatile `boolean` flag `isStopping` to ensure that a shutdown has not been initiated, then calls the semaphore method `tryAcquire`, which returns `false` immediately if no permits are available (indicating that the shutdown is in process or has completed). If `tryAcquire` succeeds, `addTasks` adds its tasks to the queue, then releases the permit.

The `shutdown` method first sets the `isStopping` flag so that no new `addTasks` method executions can begin. Then it has to wait until all the permits previously acquired have been returned. To do that, it calls `acquireUninterruptibly`, specifying that it needs *all* the permits; that call will block until they are all released by the producer threads and `shutdown` can be completed by returning the unprocessed tasks.

BlockingQueue Implementations

The Collections Framework provides five implementations of `BlockingQueue`.

LinkedBlockingQueue

This implementation is a thread-safe, FIFO-ordered queue, based on a linked node structure. The two standard collection constructors create a thread-safe blocking queue with a capacity of `Integer.MAX_VALUE`. You can specify a lower capacity using a third constructor:

```
LinkedBlockingQueue(int capacity)
```

The ordering imposed by `LinkedBlockingQueue` is FIFO. Queue insertion and removal are executed in constant time; operations such as `contains` that require traversal of the array have linear complexity. The iterators are weakly consistent.

ArrayBlockingQueue

This implementation is based on a *circular array*—a linear structure in which the first and last elements are logically adjacent. [Figure 13-4](#) shows the idea. The position labeled “head” indicates the head of the queue; each time the head element is removed from the queue, the head index is advanced. Similarly, each new element is added at the tail position, resulting in that index being advanced. When either index needs to be advanced past the last element of the array, it gets the value 0. If the two

indices have the same value, the queue is either full or empty, so an implementation must separately keep track of the count of elements in the queue.

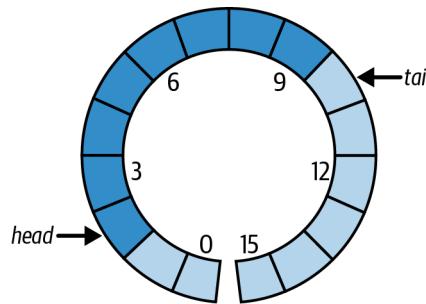


Figure 13-4. A circular array

Constructors for array-backed collection classes generally have a single configuration parameter, the initial length of the array. For fixed-size classes like `ArrayBlockingQueue`, this parameter is necessary in order to define the capacity of the collection. (For variable-size classes like `ArrayList`, a default initial capacity can be used, so constructors are provided that don't require the capacity.) For `ArrayBlockingQueue`, the three constructors are:

```
ArrayBlockingQueue(int capacity)
ArrayBlockingQueue(int capacity, boolean fair)
ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c)
```

The `Collection` parameter to the last of these allows an `ArrayBlockingQueue` to be initialized with the contents of the specified collection, added in the traversal order of the collection's iterator. For this constructor, the specified capacity must be at least as great as the size of the supplied collection, or at least 1 if the supplied collection is empty. Besides configuring the backing array, the last two constructors also require a `boolean` argument to control how the queue will handle multiple blocked requests. These will occur when multiple threads attempt to remove items from an empty queue or add items to a full one. When the queue becomes able to service one of these requests, which one should it choose? The alternatives are to provide a guarantee that the queue will choose the request that has been waiting longest—that is, to implement a *fair* scheduling policy—or to allow the implementation to choose one. Fair scheduling sounds like the better alternative, since it avoids the possibility that an unlucky thread might be delayed indefinitely, but in practice, the benefits it provides are rarely important enough to justify incurring the large overhead that it imposes on a queue's operation. If fair scheduling is not specified, `ArrayBlockingQueue` will normally approximate fair operation, but with no guarantees.

The ordering imposed by `ArrayBlockingQueue` is FIFO. Queue insertion and removal are executed in constant time; operations such as `contains` that require traversal of the array have linear complexity. The iterators are weakly consistent.

PriorityBlockingQueue

This implementation is a thread-safe, blocking version of `PriorityQueue` (see “[PriorityQueue](#)” on page 211), with similar ordering and performance characteristics. Its iterators are fail-fast, so they will throw `ConcurrentModificationException` under multithread access; only if the queue is quiescent will they succeed. To iterate safely over a `PriorityBlockingQueue`, transfer the elements to an array and iterate over that instead.

DelayQueue

This is a specialized priority queue, in which the ordering is based on the *delay time* for each element—the time remaining before the element will be ready to be taken from the queue. If all elements have a positive delay time—that is, none of their associated delay times has expired—an attempt to poll the queue will return `null`. If one or more elements has an expired delay time, the one with the longest-expired delay time will be at the head of the queue. The elements of a `DelayQueue` belong to a class that implements `java.util.concurrent.Delayed`:

```
interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

The `getDelay` method of a `Delayed` object returns the remaining delay associated with that object. (This interface precedes the introduction of the `java.time` API, but with that now available, an obvious improvement would be the addition of a default `getDelay` method returning a `Duration`.) The `compareTo` method of `Comparable` (see “[Comparable](#)” on page 35) must be defined to give results that are consistent with the delays of the objects being compared. This means that it will rarely be compatible with `equals`, so `Delayed` objects are not suitable for use with implementations of `SortedSet` and `SortedMap`.

For example, in our to-do manager we are likely to need reminder tasks, to ensure that we follow up on email and phone messages that have gone unanswered. We could define a new `DelayedTask` class, as shown in [Example 13-2](#), and use it to implement a reminder queue:

[org/jgcbook/chapter13/C_blocking_queue/Program_1](#)

```
BlockingQueue<DelayedTask> reminderQueue = new DelayQueue<DelayedTask>();
reminderQueue.offer(new DelayedTask(databaseCode, 1));
reminderQueue.offer(new DelayedTask(guiCode, 2));
...
```

```

// now get the first reminder task that is ready to be processed
DelayedTask t1 = reminderQueue.poll();
if (t1 == null) {
    // no reminders ready yet
} else {
    // process t1
}

```

Example 13-2. The class DelayedTask

org/jcbook/chapter13/C_blocking_queue/DelayedTask

```

public class DelayedTask implements Delayed {

    private final LocalDateTime endTime;
    private final Task task;
    private static Comparator<Delayed> dtComparator =
        Comparator.comparing(dt -> dt.getDelay(TimeUnit.MILLISECONDS));

    public DelayedTask(Task t, int daysDelay) {
        task = t;
        endTime = LocalDateTime.now().plusDays(daysDelay);
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return unit.convert(Duration.between(LocalDateTime.now(), endTime));
    }

    @Override
    public int compareTo(Delayed d) {
        return dtComparator.compare(this, d);
    }

    public Task getTask() {
        return task;
    }
}

```

Most queue operations respect delay values and will treat a queue with no expired elements as if it were empty. The exceptions are `peek` and `remove`, which, perhaps surprisingly, will allow you to examine the head element of a `DelayQueue` whether or not it is expired. Like them, and unlike the other methods of `Queue`, collection operations on a `DelayQueue` do not respect delay values. For example, here are two ways of copying the elements of `reminderQueue` into a set:

```

Set<DelayedTask> delayedTaskSet1 = new HashSet<DelayedTask>();
delayedTaskSet1.addAll(reminderQueue);

Set<DelayedTask> delayedTaskSet2 = new HashSet<DelayedTask>();
reminderQueue.drainTo(delayedTaskSet2);

```

The set `delayedTaskSet1` will contain all the reminders in the queue, whereas the set `delayedTaskSet2` will contain only those ready to be used.

`DelayQueue` shares the performance characteristics of `PriorityQueue`, on which it is based, and, like it, has fail-fast iterators. The comments on `PriorityBlockingQueue` iterators (see “[PriorityBlockingQueue](#)” on page 221) apply to these too.

SynchronousQueue

At first sight, you might think there is little point to a queue with no internal capacity, which is a short description of `SynchronousQueue`. But, in fact, it can be very useful; a thread that wants to add an element to a `SynchronousQueue` must wait until another thread is ready to simultaneously take it off, and the same is true—in reverse—for a thread that wants to take an element off the queue. So `SynchronousQueue` has the function that its name suggests: that of a *rendezvous*—a mechanism for synchronizing two threads. (Don’t confuse the concept of synchronizing threads in this way—allowing them to cooperate by exchanging data—with Java’s keyword `synchronized`, which prevents simultaneous execution of code by different threads.) There are two constructors for `SynchronousQueue`:

```
SynchronousQueue()
SynchronousQueue(boolean fair)
```

A common application for `SynchronousQueue` is in work-sharing systems in which the design ensures that there are enough consumer threads to guarantee that producer threads can hand tasks over without having to wait. In this situation, it allows safe transfer of task data between threads without incurring the `BlockingQueue` overhead of enqueueing, then dequeuing, each task being transferred.

As far as the `Collection` methods are concerned, a `SynchronousQueue` behaves like an empty `Collection`; `Queue` and `BlockingQueue` methods behave as you would expect for a queue with zero capacity, which is therefore always empty. The `iterator` method returns an empty iterator whose `hasNext` method always returns `false`.

TransferQueue

The interface `TransferQueue<E>` provides producers with a way of choosing between enqueueing data synchronously and asynchronously—useful, for example, in messaging systems that allow both synchronous and asynchronous messages. As an extension of `BlockingQueue`, it provides a system with the ability to throttle production by blocking producers from adding indefinitely to a bounded queue. (This feature is not actually used in the only platform implementation, `LinkedTransferQueue`, which is always unbounded; so, for this class, `put` always implements asynchronous queueing.) In addition, however, it exposes a new method, `transfer`, which a producer can call if it wishes to block until the enqueued element has been taken by a consumer—a

synchronous handshake like that provided by `SynchronousQueue`. `transfer` is provided in three versions:

```
void transfer(E e)           transfer the element to a consumer, waiting as long as necessary
boolean tryTransfer(E e);    transfer the element to a consumer if possible
boolean tryTransfer(E e,
long timeout, TimeUnit unit)
```

It also exposes two helper methods that provide a rough metric of the waiting consumer count:

```
boolean hasWaitingConsumer();  return true if there is at least one waiting consumer
int getWaitingConsumerCount(); return an estimate of the number of waiting consumers
```

Like all methods summarizing the state of a concurrent collection, these methods will at best give a snapshot value, which may have changed by the time the results can be processed.

The JDK offers one implementation of `TransferQueue`, `LinkedTransferQueue`. This is an unbounded FIFO queue, with some interesting properties: it is lock-free, like `ConcurrentLinkedQueue` but with the blocking methods that that class lacks; it supports the `transfer` methods of its interface via a “dual queue” whose nodes can represent either enqueued data or outstanding dequeue requests; and, unusually among concurrent classes, it provides fairness without degrading performance. In fact, it outperforms `SynchronousQueue` even in the latter’s “nonfair” mode.

Enqueueing and dequeuing are both $O(1)$. The iterator is weakly consistent.

Deque

A *deque* is a double-ended queue. Unlike a queue, in which elements can be inserted only at the tail and inspected or removed only at the head, a deque can accept elements for insertion and present them for inspection or removal at either end. Also unlike `Queue`, the contract for the Collections Framework interface `Deque<E>` specifies the ordering it will use in presenting its elements: it is a linear structure in which elements added at the tail are yielded up in the same order at the head. Used as a queue, then, a deque is always a FIFO structure; the contract does not allow for, say, priority deques. If elements are removed from the same end (either head or tail) at which they were added, a deque acts as a stack or *last in, first out* (LIFO) structure.

The fast `Deque` implementation `ArrayDeque` uses a circular array (see “[BlockingQueue Implementations](#)” on page 219) and is the implementation of choice for stacks and

queues. Concurrent deques have a special role to play in parallelization, discussed in “[BlockingDeque](#)” on page 229.

The Methods of Deque

The `Deque` interface extends `Queue` with methods symmetric with respect to the head and tail. For clarity of naming, the `Queue` methods that implicitly refer to one end of the queue acquire a synonym that makes their behavior explicit for `Deque`. For example, the methods `peek` and `offer`, inherited from `Queue`, are equivalent to `peekFirst` and `offerLast`. (“First” and “last” refer to the head and tail of the deque; the Javadoc for `Deque` also uses “front” and “end.”)

Collection-like methods

<code>void addFirst(E e)</code>	insert e at the head if there is enough space
<code>void addLast(E e)</code>	insert e at the tail if there is enough space
<code>void push(E e)</code>	insert e at the head if there is enough space
<code>boolean removeFirstOccurrence(Object o);</code>	remove the first occurrence of o
<code>boolean removeLastOccurrence(Object o);</code>	remove the last occurrence of o
<code>Iterator<E> descendingIterator()</code>	get an iterator, returning deque elements in reverse order

The contracts for the methods `addFirst` and `addLast` are similar to the contract for the `add` method of `Collection`, but specify in addition where the element to be added should be placed and that the exception to be thrown if it cannot be added is `IllegalStateException`. As with bounded queues, users of bounded deques should avoid these methods in favor of `offerFirst` and `offerLast`, which can report “normal” failure by means of a returned `boolean` value.

The method name `push` is a synonym of `addFirst`, provided for the use of `Deque` as a stack. The methods `removeFirstOccurrence` and `removeLastOccurrence` are analogues of `Collection::remove`, but specify in addition exactly which occurrence of the element should be removed. The return value signifies whether an element was removed as a result of the call.

Queue-like methods

<code>boolean offerFirst(E e)</code>	insert e at the head if the deque has space
<code>boolean offerLast(E e)</code>	insert e at the tail if the deque has space

The method `offerLast` is a synonym for the equivalent method `offer` on the `Queue` interface.

The methods that return `null` for an empty deque are:

- E `peekFirst()` retrieve but do not remove the first element
- E `peekLast()` retrieve but do not remove the last element
- E `pollFirst()` retrieve and remove the first element
- E `pollLast()` retrieve and remove the last element

The methods `peekFirst` and `pollFirst` are synonyms for the equivalent methods `peek` and `poll` on the `Queue` interface.

The methods that throw an exception for an empty deque are:

- E `getFirst()` retrieve but do not remove the first element
- E `getLast()` retrieve but do not remove the last element
- E `removeFirst()` retrieve and remove the first element
- E `removeLast()` retrieve and remove the last element
- E `pop()` retrieve and remove the first element

The methods `getFirst` and `removeFirst` are synonyms for the equivalent methods `element` and `remove` on the `Queue` interface. The method name `pop` is a synonym for `removeFirst`, again provided for stack use.

Methods inherited from `SequencedCollection`

Of the seven methods of `SequencedCollection`, six are in fact promoted from `Deque`. The only new one—which is also the only one providing a view of a `Deque`—is `reversed`, which is a covariant override of the `SequencedCollection` method, returning a `Deque`:

```
Deque<E> reversed() return a reverse-ordered view of this Deque
```

Deque Implementations

A `Deque` is a double-ended queue that can both accept and yield up elements at either end. A `Deque`, like a `Queue`, can be used as a conduit of information between producers and consumer threads. Its particular advantage in this context is that it provides the ability to implement work stealing, a load-balancing technique in which

idle threads steal tasks from the tail of the work queues of busier threads, to maximize parallel efficiency. “[BlockingDeque](#)” on page 229 explains work stealing in more detail.

ArrayDeque

Along with the introduction of the interface `Deque` came a very efficient implementation, `ArrayDeque`, based on a circular array like that of `ArrayBlockingQueue` (see “[BlockingQueue Implementations](#)” on page 219). It fills a gap among `Queue` classes; previously, if you wanted a FIFO queue to use in a single-threaded environment, you would have had to use the class `LinkedList` (which we cover next, but which should be avoided as a general-purpose `Queue` implementation), or else pay an unnecessary overhead for thread safety with one of the concurrent classes `ArrayBlockingQueue` or `LinkedBlockingQueue`. `ArrayDeque` is instead the general-purpose implementation of choice, for both deques and FIFO queues. It has the performance characteristics of a circular array: adding or removing elements at the head or tail takes constant time. The iterators are fail-fast.

A circular array in which the head and tail can be continuously advanced in this way is better as a deque implementation than a noncircular one (e.g., the standard implementation of `ArrayList`, which we cover in “[List Implementations](#)” on page 241) in which removing the head element requires changing the position of all the remaining elements so that the new head is at position 0. Notice, though, that only the elements at the ends of the deque can be inserted and removed in constant time. If an element is to be removed from near the middle, which can be done for deques via the method `Iterator::remove`, then all the elements from one end must be moved along to maintain a compact representation. As a result, insertion and removal of elements in the middle of the deque has complexity $O(N)$. [Figure 13-5](#) shows the element at index 6 being removed from the deque.

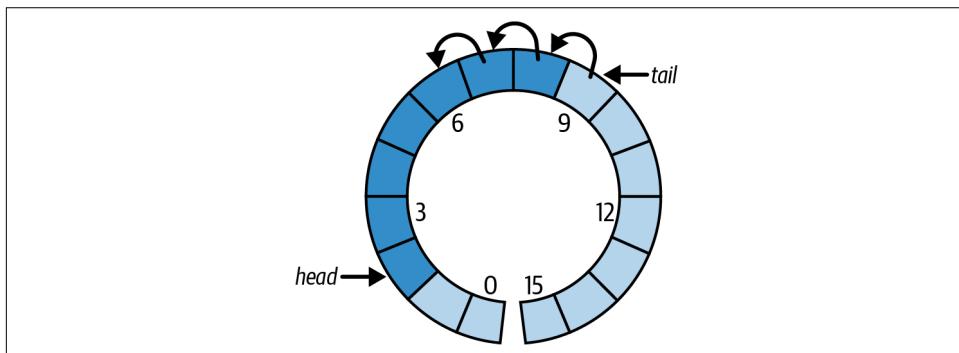


Figure 13-5. Removing an element from a circular array

LinkedList

Among Deque implementations, `LinkedList` is an oddity; for example, it is alone in permitting `null` elements, which are discouraged by the `Queue` interface because of the common use of `null` as a special value. It has been in the Collections Framework from the start, originally as one of the standard implementations of `List` (see “[List Implementations](#)” on page 241), and was retrofitted with the methods of `Queue` when that was introduced, and then later those of `Deque`. It is based on a linked list structure similar to those we saw in “[ConcurrentSkipListSet](#)” on page 203 as the basis for skip lists, but with an extra field in each node, pointing to the previous entry (see [Figure 13-6](#)). These pointers allow the list to be traversed backward—for example, for reverse iteration, or to remove an element from the end of the list.

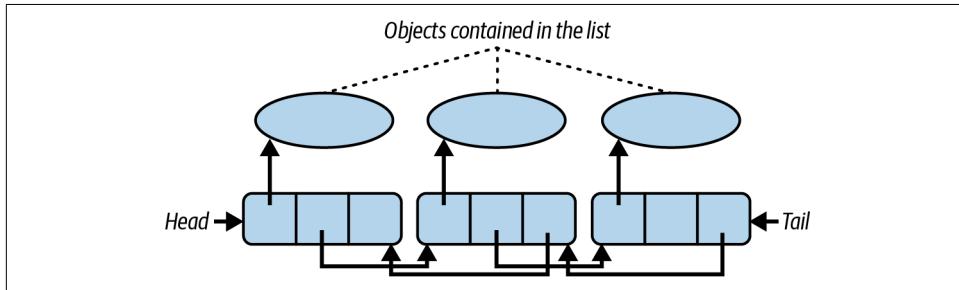


Figure 13-6. A doubly linked list

As an implementation of `Deque`, `LinkedList` is not popular. Its main advantage, that of constant-time enqueueing and dequeueing, is rivaled for queues and deques by the otherwise superior `ArrayDeque`. The only reason for using `LinkedList` as a queue or deque implementation would be that, besides the usual head and tail operations, you also need to add or remove elements from the middle of the list—an unusual requirement. With `LinkedList`, even that comes at a high price; the position of such elements has to be reached by linear traversal, with a time complexity of $O(N)$. “[Avoid LinkedList](#)” on page 304 explains why in general you should avoid using this class.

The only method exposed by `LinkedList` that is not inherited from its two interfaces, `Deque` and `List`, is a covariant override of `reversed`:

```
LinkedList<E> reversed()  return a reverse-ordered view of this LinkedList
```

The constructors for `LinkedList` are just the standard ones described in “[Collection Constructors](#)” on page 176. Its iterators are fail-fast.

BlockingDeque

In “[BlockingQueue](#)” on page 215, we saw that `BlockingQueue` adds four methods to the `Queue` interface, enabling enqueueing or dequeuing an element either indefinitely or until a fixed timeout has elapsed. `BlockingDeque` provides two new methods for each of those four, to allow for the operation to be applied either to the head or the tail of the `Deque`. So for example, to the `BlockingQueue` method:

```
void put(E e)    add e to the Queue, waiting as long as necessary
```

`BlockingDeque` adds:

```
void putFirst(E e)  add e to the head of the Deque, waiting as long as necessary
```

```
void putLast(E e)   add e to the tail of the Deque, waiting as long as necessary
```

Clearly, `put` and `putLast` are synonymous, as are `take` and `takeFirst`, and so on; the name duplication is provided for clarity of use. Similar new methods are defined for each of the other three `BlockingQueue` methods: `offer`, `poll`, and `take`.

Good load-balancing algorithms have become increasingly important as multicore and multiprocessor architectures have become standard. Concurrent deques are the basis of one of the best load-balancing methods, *work stealing*. To understand work stealing, imagine a load-balancing algorithm that distributes tasks in some way—round-robin, say—to a series of queues, each of which has a dedicated consumer thread that repeatedly takes a task from the head of its queue, processes it, and returns for another. Although this scheme does provide speedup through parallelism, it has a major drawback: we can imagine two adjacent queues, one with a backlog of long tasks and a consumer thread struggling to keep up with them, and next to it an empty queue with an idle consumer waiting for work. It would clearly improve throughput if we allowed the idle thread to take a task from the head of another queue.

Work stealing improves still further on this idea; observing that for the idle thread to steal work from the head of another queue risks contention for the head element, it changes the queues for deques and instructs idle threads to take a task from the *tail* of another thread’s deque. This turns out to be a highly efficient mechanism, and it is becoming widely used.

Implementing BlockingDeque

BlockingDeque has only one implementation in the JDK: `LinkedBlockingDeque`. `LinkedBlockingDeque` uses a doubly linked list structure like that of `LinkedList`. It can optionally be bounded, so, besides the two standard constructors, it provides a third that can be used to specify its capacity:

```
LinkedBlockingDeque(int capacity)
```

It has similar performance characteristics to `LinkedBlockingQueue`—queue insertion and removal take constant time, and operations such as `contains`, which require traversal of the queue, take linear time. The iterators are weakly consistent.

Comparing Queue Implementations

Table 13-2 shows the sequential performance, disregarding locking and CAS overheads, for some sample operations of the Deque and Queue implementations we have discussed. These results should be interesting to you in terms of understanding the behavior of your chosen implementation, but as we mentioned at the start of the chapter, they are unlikely to be the deciding factor. Your choice is more likely to be dictated by the functional and concurrency requirements of your application.

Table 13-2. Comparative performance of different Queue and Deque implementations

	offer	peek	poll	size
PriorityQueue	$O(\log N)$	$O(1)$	$O(\log N)$	$O(1)$
ConcurrentLinkedQueue	$O(1)$	$O(1)$	$O(1)$	$O(N)$
ArrayBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
LinkedBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
PriorityBlockingQueue	$O(\log N)$	$O(1)$	$O(\log N)$	$O(1)$
DelayQueue	$O(\log N)$	$O(1)$	$O(\log N)$	$O(1)$
LinkedTransferQueue	$O(1)$	$O(1)$	$O(1)$	$O(n)$
LinkedList	$O(1)$	$O(1)$	$O(1)$	$O(1)$
ArrayDeque	$O(1)$	$O(1)$	$O(1)$	$O(1)$
LinkedBlockingDeque	$O(1)$	$O(1)$	$O(1)$	$O(1)$

In choosing a Queue, the first question to ask is whether the implementation you choose needs to support concurrent access. If not, your choice is straightforward: for FIFO ordering, choose `ArrayDeque`; for priority ordering, choose `PriorityQueue`.

If your application does demand thread safety, you next need to consider ordering. If you need priority or delay ordering, your choice will be `PriorityBlockingQueue` or `DelayQueue`, respectively. If, on the other hand, FIFO ordering is acceptable,

the third question is whether you need blocking methods, as you usually will for producer/consumer problems (either because the consumers must handle an empty queue by waiting, or because you want to constrain demand on them by bounding the queue, and then producers must sometimes wait). If you don't need blocking methods or a bound on the queue size, choose the efficient and wait-free `ConcurrentLinkedQueue`.

If you do need a blocking queue, because your application requires support for producer/consumer cooperation, pause to consider whether you really need to buffer data, or whether all you need is a safe hand-off of data between the threads. If you can do without buffering (usually because you are confident that there will be enough consumers to prevent data from piling up), then `SynchronousQueue` is an efficient alternative to the remaining FIFO-blocking implementations, `LinkedBlockingQueue` and `ArrayBlockingQueue`.

Otherwise, you are finally left with the choice between these two. If you cannot fix a realistic upper bound for the queue size, then you must choose `LinkedBlockingQueue`, as `ArrayBlockingQueue` is always bounded. For bounded use, you will choose between the two on the basis of performance. Their performance characteristics in [Table 13-2](#) are the same, but these are only the formulas for sequential access; how they perform in concurrent use is a different question. A number of factors combine to influence their relative performance:

- Having separate locks on the head and the tail means that producer and consumer threads do not need to contend with each other for `LinkedBlockingQueue`. `ArrayBlockingQueue` uses a single lock.
- An upside of the bounded nature of `ArrayBlockingQueue` is that its use of memory is predictable: it never allocates, unlike `LinkedBlockingQueue`. On the other hand, preallocation means that it may be using more memory than it needs, unlike `LinkedBlockingQueue`, whose allocation will more or less match the queue size.
- Conversely, an `ArrayBlockingQueue` does not have to allocate new objects with each insertion, unlike a `LinkedBlockingQueue`.
- Linked data structures generally have much worse cache behavior than array-based ones. As we saw in [“Memory” on page 143](#), cache misses can be the dominant factor in an algorithm's performance.

In deciding whether to use a bounded or unbounded queue, you may want to consider other factors less directly linked to performance. An unbounded queue can temporarily grow very large in order to buffer spikes in supply by producers that consumers may struggle to satisfy. This can be useful, but potential drawbacks include

increased latency, the inability to block producers and so propagate back-pressure, and even the possibility of the queue using up all available memory.

These factors combine in ways that differ in every application. If queue performance is critical to the success of your application, you should measure both implementations with the benchmark that means the most to you: your application itself.

Conclusion

As this chapter has shown, queues have a different role within systems from the other collection types. Because they are used primarily for transmitting data rather than forming part of the state of an object, the emphasis of their APIs is on the features to be used for interaction: in nearly all cases, blocking methods.

In the next chapter, we return to the more storage-oriented collection types, exploring the one that probably sees more use than any other: `List`.

Lists are probably the most widely used Java collections in practice. A *list* is a collection that (unlike a set) can contain duplicates, and that (unlike a queue) gives the user full visibility of and control over the ordering of its elements. The corresponding Collections Framework interface is `List<E>`.

The `List` contract overrides the `equals` method; it states that a `List` can only ever be equal to another `List`, and then only if they contain the same elements, in the same order. The `hashCode` method is also overridden, as should always be the case when `equals` is overridden (see “Hash tables” in the section [“Implementations” on page 138](#)). The `List` interface documentation specifies the algorithm to be used to calculate the `hashCode` of a `List` from the hash codes of its elements.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter14](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter14)

List Interface Methods

The `List` interface exposes methods for positional access, for searching for a given value, for generating views, and for creating `ListIterators`—a subtype of `Iterator` with additional features that take advantage of a `List`’s sequential nature. In addition, the methods inherited from `SequencedCollection` provide convenient shorter versions of common positional access calls. Finally, static factory methods are available to create unmodifiable lists of different lengths.

Positional Access Methods

void add(int index, E e)	add element e at given index
boolean addAll(int index, Collection<? extends E> c)	add contents of c at given index
E get(int index)	return element at given index
E remove(int index)	remove element at given index
E set(int index, E e)	replace element at given index by e

These methods access elements based on their numerical position in the list. The declaration of `remove` has an unfortunate conflict with the inherited method `Collection::remove` in the case of a `List<Integer>`. Although overload resolution rules make it clear that a call like `list.remove(0)` refers to the element at index 0, this is a common source of confusion to programmers, and hence a common source of errors as well.

Search Methods

int indexOf(Object o)	return index of first occurrence of o
int lastIndexOf(Object o)	return index of last occurrence of o

These methods search for a specified object in the list and return its numerical position, or -1 if the object is not present.

View-Generating Methods

List<E> subList(int fromIndex, int toIndex)	return a view of a portion of the list
List<E> reversed()	provide a reverse-ordered view of the original collection

These methods provide different views (see “[Views](#)” on page 141) of a `List`. The method `subList` works in a similar way to the `subSet` operations on `SortedSet` (see “[NavigableSet](#)” on page 193), but it uses the positions of elements in the list rather than their values: the returned list contains the list elements starting at `fromIndex`, up to but not including `toIndex`. The returned list has no separate existence; it is just a view of part of the list from which it was obtained, and all changes in it are reflected in the original list. There is an important difference from `subSet` here, though: changes made to the sublist write through to the backing list, but the reverse is true only for nonstructural changes. If any structural changes are made to the

backing list by insertion or deletion other than via the sublist, subsequent attempts to use the sublist will result in a `ConcurrentModificationException`.

The view returned by `reversed` allows any modifications that are permitted by the original list.

List Iteration Methods

<code>ListIterator<E> listIterator()</code>	return a <code>ListIterator</code> for this list, initially positioned at index 0
<code>ListIterator<E> listIterator(int idx)</code>	return a <code>ListIterator</code> for this list, initially positioned at index <code>idx</code>

These methods return a `ListIterator`, which is an `Iterator` with extended semantics that take advantage of the list's sequential nature. The methods added by `ListIterator` support traversing a list in reverse order, changing list elements or adding new ones, and getting the current position of the iterator. The current position of a `ListIterator` always lies between two elements, so in a list of length N , there are $N + 1$ valid list iterator positions, from 0 (before the first element) to N (after the last one). A call of the first overload returns a `listIterator` set at position 0. The second overload uses the supplied value to set the initial position of the returned `listIterator`.

In addition to the `Iterator` methods `hasNext`, `next`, and `remove`, `ListIterator` exposes the following methods:

<code>void add(E e);</code>	insert the specified element into the list
<code>boolean hasPrevious();</code>	return <code>true</code> if this list iterator has further elements in the reverse direction
<code>int nextIndex();</code>	return the index of the element that would be returned by a subsequent call to <code>next</code>
<code>E previous();</code>	return the previous element in the list
<code>int previousIndex();</code>	return the index of the element that would be returned by a subsequent call to <code>previous</code>
<code>void set(E e);</code>	replace the last element returned by <code>next</code> or <code>previous</code> with the specified element

Figure 14-1 shows a list of three elements. Consider an iterator at position 2, either moved there from elsewhere or created there by a call to `listIterator(2)`. The effects of most of the operations of this iterator are intuitive: `add` inserts an element at the current iterator position (between the elements at indices 1 and 2); `hasPrevious` and `hasNext` return `true`; `previous` and `next` return the elements at indices 1 and 2, respectively; and `previousIndex` and `nextIndex` return those indices themselves. At the extreme positions of the list—0 and 3 in the figure—`previousIndex` and

`nextIndex` would return `-1` and `3` (the size of the list), respectively, and `previous` or `next` would throw `NoSuchElementException`.

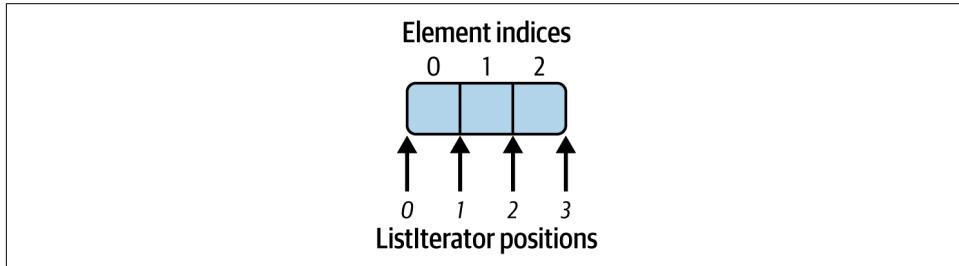


Figure 14-1. *ListIterator* positions

The operations `set` and `remove` work differently. Their effect depends not on the current position of the iterator, but on its “current” element, which is the one last traversed over using `next` or `previous`: `set` replaces the current element, and `remove` removes it. If there is no current element, either because the iterator has just been created or because the current element has been removed, these methods will throw `IllegalStateException`.

Methods Inherited from SequencedCollection

Apart from the method `reversed` (see “View-Generating Methods” on page 234), the methods inherited from `SequencedCollection` are convenience versions of positional methods already present in `List`. Table 14-1 shows the correspondence between calls of `SequencedCollection` and those of the positional access methods of `List`.

Table 14-1. *SequencedCollection* method calls, with their *List* equivalents

<code>SequencedCollection</code> call	<code>List</code> positional access call
<code>addFirst(el)</code>	<code>add(0, el)</code>
<code>addLast(el)</code>	<code>add(el)</code>
<code>getFirst()</code>	<code>get(0)</code>
<code>getLast()</code>	<code>get(size() - 1)</code>
<code>removeFirst()</code>	<code>remove(0)</code>
<code>removeLast()</code>	<code>remove(size() - 1)</code>

Factory Methods

These methods, overloaded versions of `List::of`, create unmodifiable `List` objects. We'll discuss them in “[UnmodifiableList](#)” on page 244.

Using the Methods of List

Let's see how we might use some of these methods in the to-do manager. In the last chapter, we considered representing the organization of a single day's tasks in a queue-based class with shutdown capabilities. One useful way of enlarging the scope of the application is to have a number of objects of this type, each one representing the tasks that are scheduled for a day in the future. We can store references to these objects in a `List`, which (to keep things simple and to avoid getting involved with the details of `java.time`) will be indexed on the number of days in the future that it represents. So the queue of tasks scheduled for today will be stored at index 0 of the list, the queue scheduled for tomorrow at index 1, and so on. [Example 14-1](#) shows the scheduler.

Example 14-1. A list-based task scheduler

[org/jgcbook/chapter14/B_using_the_methods_of_list/DailyTaskScheduler](#)

```
public class DailyTaskScheduler {  
    private final List<StoppableTaskQueue> schedule;  
    private final int FORWARD_PLANNING_DAYS = 365;  
  
    public DailyTaskScheduler() {  
        schedule = Stream.generate(StoppableTaskQueue::new)  
            .limit(FORWARD_PLANNING_DAYS)  
            .collect(Collectors.toCollection(CopyOnWriteArrayList::new));  
    }  
  
    public Optional<PriorityTask> getTopTask() {  
        return schedule.stream()  
            .map(StoppableTaskQueue::getFirstTask)  
            .filter(Objects::nonNull)  
            .findFirst();  
    }  
  
    // This method will be called at midnight to create a new day's  
    // queue at the planning horizon. It removes and shuts down the  
    // queue for day 0 and assigns its tasks to the new day 0.  
    public synchronized void rollOver() {  
        schedule.add(new StoppableTaskQueue());  
        StoppableTaskQueue oldDay = schedule.removeFirst();  
        schedule.getFirst().addTasks(oldDay.shutDown());  
    }  
  
    public void addTask(PriorityTask task, int day) {  
        addTasks(List.of(task), day);  
    }  
  
    public void addTasks(Collection<PriorityTask> tasks, int day) {  
        if (day < 0 || day >= FORWARD_PLANNING_DAYS)  
            throw new IllegalArgumentException("day out of range");  
  
        while (! schedule.get(day).addTasks(tasks)) {
```

```

    // If addTasks() fails, shutdown has been initiated on this
    // queue, so it must be the day 0 queue that has already been
    // removed from the schedule. The correct action, then, is to
    // call addTasks() on the new day 0 queue.
}
}
}

```

Although the example aims primarily to show the use of `List` interface methods rather than to explore any particular implementation, we can't set it up without choosing one. Since a major factor in the choice will be the concurrency requirements of the application, we need to consider them now. They are quite straightforward: clients consuming or producing tasks only ever read the `List` representing the schedule, so once it has been constructed, it is only ever written at the end of each day. At those points, the current day's queue is removed from the schedule, and a new one is added at the end (at the "planning horizon," which we have set to a year in the example). We don't need to exclude clients from using the current day's queue before that happens, because the `StoppableTaskQueue` design of [Example 13-1](#) ensures that they will be able to complete in an orderly way once the queue is stopped. So the only exclusion required is to ensure that clients don't try to read the schedule itself while the rollover procedure is changing its values.

If you recall the discussion of `CopyOnWriteArrayList` in ["Concurrent Collections" on page 159](#), you'll see that it fills these requirements very nicely. It optimizes read access, in line with one of our requirements. And in the event of a write operation, it synchronizes just long enough to create a new copy of its internal backing array, thus fulfilling our other requirement of preventing interference between read and write operations.

With the implementation chosen, you can understand the constructor of [Example 14-1](#); writing to the list is expensive, so it is sensible to use a conversion constructor to set it up with a year's worth of task queues in one operation.

The `getTask` method is straightforward; we simply iterate over the task queues, starting with today's queue, looking for a scheduled task. If the method finds no outstanding tasks, it returns `null`—and if finding a task-free day was noteworthy, how should we celebrate a task-free year?

At midnight each day, the system will call the method `rollOver`, which implements the sad ritual of shutting down the old day's task queue and transferring the remaining tasks in it to the new day. The sequence of events here is important. `rollOver` first removes the queue from the list, at which time producers and consumers may still be about to insert or remove elements. It then calls `StoppableTaskQueue::shutDown`, which, as you saw in [Example 13-1](#), returns the remaining tasks in the queue and guarantees that no more will be added. Depending on how far they have progressed,

calls of `addTask` will either complete or return `false`, indicating that they failed because the queue was shut down.

This motivates the logic of `addTask`. The only situation in which the `addTask` method of `StoppableTaskQueue` can return `false` is that in which the queue being called is already stopped. Since this only happens as the result of a rollover, a return value of `false` from `addTask` must result from a producer thread getting a reference to the day 0 queue just as a midnight rollover is about to happen. In that case, the current value of element 0 of the list is by now the new day 0, and it is safe to try again. If the second attempt fails, the thread must have been suspended for exactly 24 hours! That may be unlikely in this particular case, but any program written for such a scenario should take account of the possibility. The correct action is to continue retrying the transfer.

We could demonstrate a rollover working by adding a task and then, after the rollover, simply showing that it is still in the system. But a more convincing demonstration would show the rollover merging the old day 0 tasks into the new day 0, yielding the combined tasks for the new day 0 in the correct priority order. This can be done, at the cost of temporarily subverting the design of `StoppableTaskQueue` and `DailyTaskScheduler` by writing diagnostic methods to expose their state. For `StoppableTaskQueue`, we can add a method to copy the (transitory) contents of the queue into a list. Although iterators for priority queues don't return the elements in any particular order, the `BlockingQueue` method `drainTo` does respect the queue's ordering while emptying it into a collection. Since `drainTo` is destructive, we must apply it to a copy of the queue. Fortunately, `PriorityBlockingQueue`'s copy constructor also respects the ordering of its argument:

```
org/jgcbook/chapter13/C_blocking_queue/StoppableTaskQueueWithAccessor
public List<PriorityTask> getTaskList() {
    BlockingQueue<PriorityTask> copyQueue = new PriorityBlockingQueue<>(taskQueue);
    List<PriorityTask> taskList = new ArrayList<>();
    copyQueue.drainTo(taskList);
    return taskList;
}
```

And for `DailyTaskScheduler` we can add a method to call `getTasks`, returning the contents of a given day of the schedule:

```
org/jgcbook/chapter14/B_using_the_methods_of_list/DailyTaskSchedulerWithAccessor
public List<PriorityTask> tasksForDay(int day) {
    return schedule.get(day).getTaskList();
}
```

Now it is possible to demonstrate a rollover in action:

```
org/jgcbook/chapter14/B\_using\_the\_methods\_of\_List/DailyTaskSchedulerWithAccessor
PriorityTask guiCoding = new PriorityTask(new CodingTask("gui"), Priority.MEDIUM);
PriorityTask logicCoding = new PriorityTask(new CodingTask("logic"), Priority.HIGH);

DailyTaskScheduler scheduler = new DailyTaskScheduler();
scheduler.addTask(guiCoding, 0);
assert scheduler.tasksForDay(0).equals(List.of(guiCoding));
scheduler.addTask(logicCoding, 1);
scheduler.rollover();
assert scheduler.tasksForDay(0).equals(List.of(logicCoding, guiCoding));
```

Notice that the `rollover` method is quite expensive; it writes to the schedule twice, and since the schedule is represented by a `CopyOnWriteArrayList` (see “[CopyOn-WriteArrayList](#)” on page 243), each write causes the entire backing array to be copied. The argument in favor of this implementation choice is that `rollover` is very rarely invoked compared to the number of calls made to `getTask`, which iterates over the schedule. The alternative to using `CopyOnWriteArrayList` would be to use a `BlockingQueue` implementation, but the improvement that would provide in the rarely used `rollover` method would come at the cost of slowing down the frequently used `getTask` method, since queue iterators are not intended to be used in performance-critical situations.

Using Range-View and Iterator Methods

[Example 14-1](#) makes use of the methods of one of the four groups described earlier—positional access—in several places. To see how range-view and iterator methods could also be useful, consider how the `DailyTaskScheduler` could export its schedule, or a part of it, for a client to modify. You would want the client to be able to view this subschedule and perhaps to insert or remove tasks, but you would definitely want to forbid the insertion or removal of elements of the list itself, since these represent the sequence of days for which tasks are being scheduled. The standard way to achieve this would be by means of an unmodifiable list, as provided by the `Collections` class (see “[Unmodifiable Collections](#)” on page 281). An alternative in this case would be to return a list iterator, as the snapshot iterators for copy-on-write collections do not support modification of the backing collection. So we could define a method to provide clients with a “planning window”:

```
ListIterator<StoppableTaskQueue> getSubSchedule(int startDay, int endDay) {
    return schedule.subList(startDay, endDay).listIterator();
```

This view will be fine for today, but we have to remember to discard it at midnight, when the structural changes of removing and adding entries will invalidate it.

List Implementations

There are four concrete implementations of `List` in the Collections Framework (see [Figure 14-2](#)), differing in how fast they perform the various operations defined by the interface and how they behave in the face of concurrent modification. Unlike `Set` and `Queue`, however, `List` has no subinterfaces to specify differences in functional behavior. The following sections look at each implementation in turn and provide a performance comparison.

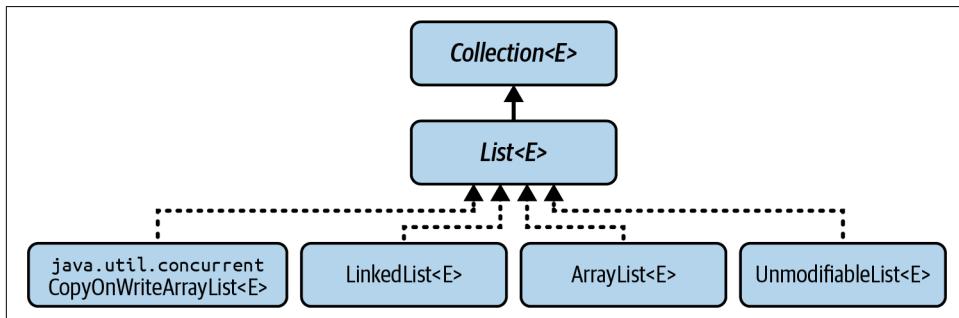


Figure 14-2. Implementations of the `List` interface

ArrayList

Arrays are provided as part of the Java language and have a very convenient syntax, but their key disadvantage—that, once created, they cannot be resized—makes them increasingly less popular than `List` implementations, which (if resizable at all) are indefinitely extensible. The most commonly used implementation of `List` is, in fact, `ArrayList`—that is, a `List` backed by an array.

The standard implementation of `ArrayList` stores the `List` elements in contiguous array locations, with the first element always stored at index 0 in the array. It requires an array at least large enough (with sufficient *capacity*) to contain the elements, together with a way of keeping track of the number of “occupied” locations (the size of the `List`). If an `ArrayList` has grown to the point where its size is equal to its capacity, attempting to add another element will require it to replace the backing array with a larger one capable of holding the old contents and the new element, and with a margin for further expansion (the standard implementation actually uses a new array that is 1.5 times the length of the old one). As we explained in [“Instruction Count and the O-notation” on page 145](#), this leads to an amortized cost of $O(1)$.

The performance of `ArrayList` reflects array performance for “random-access” operations: `set` and `get` take constant time. The downside of an array implementation is in inserting or removing elements at arbitrary positions, because that may require adjusting the position of other elements. We have already met this problem with the

`remove` method of the iterators of array-based queues (e.g., `ArrayBlockingQueue`—see “[BlockingQueue Implementations](#)” on page 219). But the performance of positional `add` and `remove` methods is much more important for lists than that of `Iterator::remove` for queues.

For example, [Figure 14-3\(a\)](#) shows a new `ArrayList` after three elements have been added by means of the following statements:

```
List<Character> charList = new ArrayList<>();  
Collections.addAll(charList, 'a', 'b', 'c');
```

If we now want to remove the element at index 1 of the array, the implementation must preserve the order of the remaining elements and ensure that the occupied region of the array still starts at index 0. So the element at index 2 must be moved to index 1, that at index 3 to index 2, and so on. [Figure 14-3\(b\)](#) shows the sample `ArrayList` after this operation has been carried out. Since every element must be moved in turn, the time complexity of this operation is proportional to the size of the list (even though, because this operation can usually be implemented in hardware, the constant factor is low).

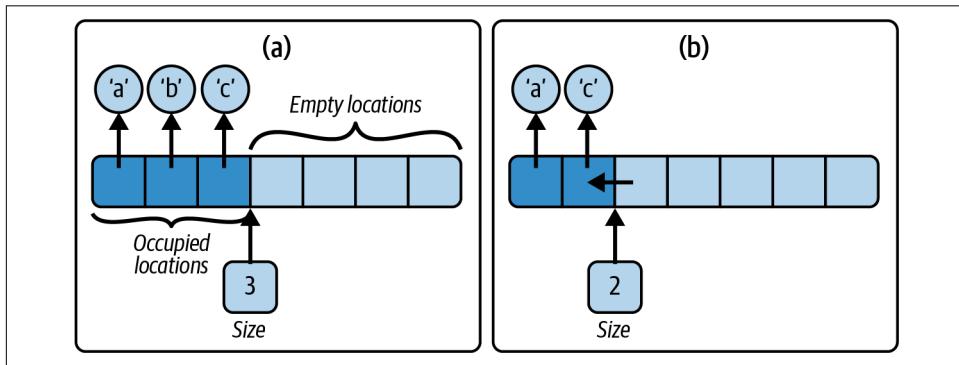


Figure 14-3. Removing an element from an `ArrayList`

Even so, you may recall the discussion of the circular arrays used to implement `ArrayBlockingQueue` and `ArrayDeque` (see “[Deque Implementations](#)” on page 226) and wonder why a circular array was not also chosen for the implementation of `ArrayList`. It is true that the `add` and `remove` methods of a circular array show much better performance only when they are called with an index argument of 0, but this is such a common case, and the overhead of using a circular array is so small, that the question remains.

Indeed, [Heinz Kabutz \(2001\)](#) presented an outline implementation of a circular array list in *The Java Specialists’ Newsletter*. In principle it is still possible that `ArrayList` may be reimplemented in this way, possibly leading to significant performance gains in many existing Java applications. A possible alternative, though less likely, is that

the circular `ArrayDeque` could be retrofitted to implement the methods of `List`. In the meantime, if your application is using a `List` in which the performance of element insertion and removal at the beginning of the list is more important than that of random-access operations, consider writing to the `Deque` interface and taking advantage of its very efficient `ArrayDeque` implementation.

As mentioned in the discussion of `ArrayBlockingQueue` in the previous chapter, variable-size array-backed collection classes can have one configuration parameter: the initial length of the array. So besides the standard Collections Framework constructors, `ArrayList` has one that allows you to choose the value of the initial capacity to be large enough to accommodate the elements of the collection without frequent create/copy operations. In Java 21, the initial capacity of an `ArrayList` created by the default constructor is 10, and that of one initialized with the elements of another collection is equal to the size of that collection.

The iterators of `ArrayList` are fail-fast.

LinkedList

We discussed `LinkedList` as a `Deque` implementation in “[Deque Implementations](#)” on page 226. There is rarely a good reason to choose it as a `List` implementation either: since the list must iterate internally to reach the required position, positional `add` and `remove` have linear time complexity, on average. Compared to a circular array implementation like `ArrayDeque`, it is superior only in addition of the first or last element, since it never has to copy or resize as an array-backed implementation must do when its capacity is reached. In principle, adding and removing elements via an iterator takes constant time if the list is not traversed, against the linear time required for noncircular array implementations. However, the other drawbacks of `LinkedList`, explored in “[Avoid LinkedList](#)” on page 304, mean that even in this situation, it is unlikely to provide better performance than `ArrayList`.

CopyOnWriteArrayList

“[Set Implementations](#)” on page 183 included a discussion of `CopyOnWriteArraySet`, a set implementation designed to provide thread safety together with very fast read access. `CopyOnWriteArrayList` is a `List` implementation with the same design aims. This combination of thread safety with fast read access is useful in some concurrent programs, especially when a collection of observer objects needs to receive frequent event notifications. The cost is that the array that backs the collection has to be treated as immutable, so a new copy is created whenever any changes are made to the collection. This cost may be acceptable if changes to the set of observers occur only rarely.

The class `CopyOnWriteArrayList` in fact delegates all of its operations to an instance of `CopyOnWriteArrayList`, taking advantage of the atomic operations `addIfAbsent` and `addAllAbsent` provided by the latter to enable the `Set` methods `add` and `addAll` to avoid introducing duplicates to the set. In addition to the two standard constructors (see “[Collection Constructors](#)” on page 176), `CopyOnWriteArrayList` has an extra one that allows it to be created using the elements of a supplied array as its initial contents. Its iterators are snapshot iterators, reflecting the state of the list at the time of their creation.

UnmodifiableList

Like `UnmodifiableSet` (see “[UnmodifiableSet](#)” on page 189), you won’t find any reference to the implementation(s) `UnmodifiableList<E>` of the `List` interface in the Javadoc or in the code of the Collections Framework: it’s another name invented in this book for a family of package-private classes that client programmers can never access by name, but that are important because they provide the implementation of the unmodifiable lists obtained from the various overloads of the factory methods `List::of` and `List::copyOf`. The properties of the members of this family are described in the Javadoc for `List`:

- They are unmodifiable—elements cannot be added or removed. Calling any mutator method will always cause an `UnsupportedOperationException` to be thrown.
- They are null-hostile. Attempts to create them with `null` elements will result in `NullPointerException`.
- The lists and their `subList` views implement the `RandomAccess` interface.

The static factory methods—the keyword is omitted in the following tables for brevity—can be divided into the same three groups as those of `UnmodifiableSet`. The first contains just an overload of the method `of` that takes no arguments and returns an empty unmodifiable list:

<E> `List<E> of()` return an unmodifiable list containing zero elements

The second group of methods allows you to create an unmodifiable list from up to 10 specified elements:

<E> <code>List<E> of(E e1)</code>	return an unmodifiable list containing one element
<E> <code>List<E> of(E e1, E e2)</code>	return an unmodifiable list containing two elements
<E> <code>List<E> of(E e1, E e2, E e3)</code>	return an unmodifiable list containing three elements

further overloads, taking from four to nine arguments

<E> List<E> of(E e1, E e2, E e3, E e4, return an unmodifiable list containing 10 elements
E e5, E e6, E e7, E e8, E e9, E e10)

The third group allows you to create an unmodifiable list from a collection, an array, or a varargs-supplied list of arguments:

<E> List<E> copyOf(Collection<? extends E> coll)	return an unmodifiable list containing an arbitrary number of elements
<E> List<E> of(E... elements)	return an unmodifiable list containing an arbitrary number of elements

The performance characteristics of the *UnmodifiableList* classes are like those of *ArrayList* for read operations.

Comparing List Implementations

Table 14-2 shows the comparative performance for some sample operations on *List* classes. Even though the choice here is much narrower than with queues or even sets, the same process of elimination can be used. As with queues, the first question to ask is whether your application requires thread safety. If so, you should use *CopyOnWriteArrayList*, if you can—that is, if writes to the list will be relatively infrequent. If not, you will have to use a synchronized wrapper (see “[Synchronized Collections](#)” on page 281) around *ArrayList* or *LinkedList*.

Table 14-2. Comparative performance of different *List* implementations

	get	add	add(int,e)	contains	iterator .next	remove(0)	iterator .remove
ArrayList	O(1)	O(1)	O(N)	O(N)	O(1)	O(N)	O(N)
LinkedList	O(N)	O(1)	O(1) ^a	O(N)	O(1)	O(1)	O(1) ^a
CopyOnWriteArrayList	O(1)	O(N)	O(N)	O(N)	O(1)	O(N)	n/a ^b

^a The complexity measures of O(1) for the operations `add(int, e)` and `iterator.remove` for *LinkedList* should be understood in the context of the O(N) complexity of the operation needed to locate the site of the addition or removal.

^b Throws `UnsupportedOperationException`.

For most list applications the choice is between *ArrayList* and *LinkedList*, synchronized or not. Reading Table 14-2, you might think that there will be situations—if, for example, your application needs to frequently insert and remove elements near the start of the list—in which *LinkedList* is the better choice. In practice, however, this is rarely the case; we explore the reasons for this in “[Avoid LinkedList](#)” on page 304. If the insertions and removals are actually *at* the start of the list, an alternative better worth considering for single-threaded code is to write to the *Deque*

interface, taking advantage of its very efficient `ArrayDeque` implementation. For relatively infrequent random access, use an iterator, or copy the `ArrayDeque` elements into an array using `toArray`. As always, if in doubt, measure your application's performance with each implementation.

Conclusion

This chapter on `List` concludes our exploration of the `Collection` subtypes. Despite its popularity, `List` is the only one of the major interfaces of the Collections Framework not to have a mainstream concurrent implementation. This contrasts with the subject of the next chapter, `Map`, concurrent implementations of which are mission-critical for many enterprise Java systems.

CHAPTER 15

Maps

The `Map` interface is the last of the major Collections Framework interfaces, and the only one that does not inherit from `Collection`. A `Map` stores key-to-value associations, or *entries*, in which the keys are unique. The importance of this interface lies in the fact that its implementations provide very fast—ideally, constant-time—operations to look up the value corresponding to a given key.

The `Map` interface overrides the `equals` and `hashCode` methods and provides definitions for them (`hashCode` should always be overridden when `equals` is overridden, as we saw in the discussion of hash tables in “[Implementations](#)” on page 138). For the purposes of these methods, a `Map` is considered as a `Set` of map entries (key-value pairs). As with the `Set` contract, a `Map` can only ever be equal to another `Map`, and then only if they are the same size and contain equal entries. The hash code of a `Map` is the sum of the hash codes of its entries. A map entry is defined by the `Map.Entry` interface. Two map entries are equal if both their keys and values are equal, and the hash code of a map entry is defined to be the result of applying an exclusive OR operation on the hash codes of the key and the value.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter15](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter15)

Map Interface Methods

The methods of `Map` can be divided according to the two ways in which a map can be seen: as a set of entries or as a lookup mechanism. From the first viewpoint, the operations to consider correspond broadly to the operations of `Iterable` and `Collection`. From the second viewpoint, the important groups are view-creating operations, compound operations, and factory methods.

Where `Map` operations accept lambdas as parameters, the types of the parameters to these lambdas are often bounded, above or below, by the key or value type. Similarly, `Map.Entry` is usually parameterized on bounded types. In this section, however, the listings provide the precise types instead, in the interests of brevity.

Iterable-like Operations

```
void forEach(BiConsumer<K,V> action)  perform action on each entry in the map, in the iteration order of  
the entry set if that is specified
```

In the same way that `Iterable::forEach` (see “[Iterable and Iterators](#)” on page 135) allows a collection to supply each of its elements in turn to a `Consumer`, this method allows a map to supply each of its key-value entries to a `BiConsumer`.

Collection-like Operations

The methods in this group can be divided into three subgroups that are broadly parallel to the first three operation groups of `Collection`, for adding or modifying entries, removing entries, and querying map contents.

Adding or replacing associations

<code>V put(K key, V value)</code>	add or replace a key-value association; return the old value if the key was present, otherwise <code>null</code>
<code>void putAll(Map<K,V> m)</code>	copy all the mappings in <code>m</code> into the receiver
<code>void replaceAll(BiFunction<K,V,V> remapper)</code>	replace each value with the result of invoking <code>remapper</code>

Removing associations

<code>void clear()</code>	remove all associations from this map
<code>V remove(Object key)</code>	remove the association, if any, with the given key; return the value with which it was associated if any, and otherwise <code>null</code>

Querying the contents of a map

<code>V get(Object k)</code>	return the value corresponding to <code>k</code> , or <code>null</code> if <code>k</code> is not present as a key
<code>boolean containsKey(Object k)</code>	return <code>true</code> if <code>k</code> is present as a key
<code>boolean containsValue(Object v)</code>	return <code>true</code> if <code>v</code> is present as a value
<code>int size()</code>	return the number of associations
<code>boolean isEmpty()</code>	return <code>true</code> if there are no associations

The contracts for `put`, `remove`, and `get` present a problem with `null`-tolerant maps: a returned `null` for a map key can signify either that a `null` value had been associated with that key, or that the key was not present. The method `containsKey` can be used to distinguish between these situations. This problem normally arises only with non-concurrent maps, as concurrent maps usually do not accept `nulls`. If you want to write to a `null`-tolerant map conditionally on a key being absent or present, one of the compound methods (see “[Compound Operations](#)” on page 250) may be useful. However, with non-concurrent maps they cannot be relied on to act atomically, so they may appear to behave inconsistently if used on such a map when it is under concurrent access. This is the TOCTOU problem, discussed in “[Using synchronized collections safely](#)” on page 303. The design decision to allow `null` values in maps is evaluated in “[nulls](#)” on page 319.

Providing Collection Views of the Keys, Values, or Entries

<code>Set<Map.Entry<K,V>> entrySet()</code>	return a <code>Set</code> view of the associations
<code>Set<K> keySet()</code>	return a <code>Set</code> view of the keys
<code>Collection<V> values()</code>	return a <code>Collection</code> view of the values

The view collections returned by these methods are backed by the map, so they reflect changes to the map. The connection in the opposite direction is more limited: you can remove elements from the view collections, but attempting to add elements will result in an `UnsupportedOperationException`. Removing a key from the `keySet` removes the single corresponding key-value association. Removing a value from the collection returned by `values`, on the other hand, removes only one of the associations mapping to it; the value may still be present as part of an association with a different key. An iterator over the view will become undefined if the backing map is concurrently modified.

Compound Operations

Because maps are very often used in multithreaded environments, the interface exposes a variety of compound actions. These fuse a conditional test—whether a key is absent or present, possibly with a specific value—either with a supplied value or with an action, represented by a lambda, to compute the value lazily. The primary intended use of these methods is to perform conditional transactions on the state of a concurrent map. Some methods of this group originally formed the `ConcurrentMap` interface when that was introduced in the `java.util.concurrent` package. Later, the introduction of default methods in Java 8 allowed the `Map` interface to be extended to include them, and new ones were added to both `Map` and `ConcurrentMap`.

They fall into two groups, corresponding to the two different styles of locking, pessimistic and optimistic (see “[Mechanisms of concurrent collections](#)” on page 159).

Pessimistic-style atomic operations

These are essential in multithreaded environments to avoid unsafe test-then-act operation sequences, as seen in “[Synchronized Collections and Fail-Fast Iterators](#)” on page 158. They are also very useful as convenience methods for non-thread-safe maps.

`V getOrDefault(Object k, V defaultValue)` return the value to which k is mapped, or defaultValue if this map contains no mapping for the key

If a map has values only for certain keys, the `getOrDefault` method allows you to use a default value for all other keys without having to store that value in the map against them all.

`V putIfAbsent(K key, V value)` create a key-value mapping if the key is absent or mapped to null; in these cases, return null, and otherwise return the current value

`putIfAbsent` is useful if you want to write something the first time you see it but not thereafter. For example, to record the timestamp corresponding to the first occurrence of a particular kind of event, you could write:

```
Map<EventKind,Long> firstOccurrenceMap = ... ;  
...  
firstOccurrenceMap.putIfAbsent(event.getKind(), System.currentTimeMillis());
```

In this example, the overhead of boxing the timestamp into a `Long` will be incurred for every event, not only the first. You will usually want to avoid this performance cost by using another compound method, `computeIfAbsent`, which computes the new value lazily:

```
firstOccurrenceMap.computeIfAbsent(event.getKind(),  
    key -> System.currentTimeMillis());
```

This is the declaration of `computeIfAbsent`:

```
V computeIfAbsent(K k, Function<K,V> mapper) apply the mapper function to k and use the result to  
create a key-value pair, unless either k is currently mapped  
to a non-null value or the result is null; return the value  
now associated with k, or null if k is not now present in  
the map
```

The Javadoc for `computeIfAbsent` offers an example of its most common use, creating a map from a key to a list (or other accumulation) of multiple values:

```
map.computeIfAbsent(key, k -> new ArrayList<V>()).add(newValue);
```

In a different scenario, you might want to accumulate the value in the map by means of an operator, like addition or concatenation, that combines the old and newly supplied values to create a new one to store against the key. The most useful method in that scenario is `merge`:

```
V merge(K k, V initialValue, BiFunction<V,V,V> remapper) if k is not present or is mapped to null,  
associate it with newValue; otherwise,  
apply remapper to the existing value  
and newValue and associate k with  
the result if it is non-null, or otherwise  
remove k
```

The Javadoc for `merge` gives this example, to either initialize an entry value to a given string `msg` or append a new value to an existing one:

```
map.merge(key, msg, String::concat);
```

The next method, `compute`, is similar to `merge`, but with the difference that instead of an initial value, it allows the contents of the key to be used in a lazy computation of the new value:

```
V compute(K k, BiFunction<K,V,V> remapper) use the result computed by remapper from k and the  
existing value (or null if k is not present) to create or modify  
a key-value pair, and return the result; if the computed result  
is null, remove any existing entry and return null
```

For example, if we want to map each word in a text to the total character count that all its occurrences contribute to the text length, we can use code like the following:

```
map.compute(word, (s,i) -> s.length() + (i == null ? 0 : i));
```

The last member of the `compute*` family is `computeIfPresent`:

```
V computeIfPresent(K k, BiFunction<K,V,V> remapper) if k is currently mapped to null, return null;  
otherwise, use the result computed by remap  
per from k and the existing value to replace  
that value, and return the result; if the computed  
result is null, remove the existing entry and  
return null
```

In the same way that `computeIfAbsent` is most useful where it may be necessary to add a new key, `computeIfPresent` can be used to remove an existing one. In an earlier example, we saw how to use `putIfAbsent` to produce a map from each kind of event to the timestamp of its first occurrence. Now suppose that we later want to process a different phase of the event stream so that the first event—and only the first—of each kind in this later phase triggers the writing of a log entry with the previously recorded timestamp. We can do that as follows:

```
firstOccurrenceMap.computeIfPresent(event.getKind(),(kind, timestamp) -> {  
    log.info("first occurrence of event " + kind + " was at " + timestamp);  
    return null;  
});
```

Returning `null` from the lambda ensures that the key will be removed from the map so that the log message can only be triggered once for each kind of event.

Optimistic-style atomic operations

<code>V replace(K k, V newValue)</code>	replace existing value for k, provided k is currently in the map; return the old value, or <code>null</code> if k was not present
<code>boolean replace(K k, V oldValue, V newValue)</code>	replace existing value for k, provided it is currently mapped to <code>oldValue</code> ; return <code>true</code> if <code>oldValue</code> was replaced
<code>boolean remove(Object key, Object value)</code>	remove the entry for this key if it is mapped to this value, returning <code>true</code> if the operation succeeded

These methods support the optimistic style of concurrent locking (see “[Mechanisms of concurrent collections](#)” on page 159). The uses of these conditional update methods are probably fairly narrow in non-concurrent systems, but they may be convenient for some applications. For example, a caller may fetch a key’s value, value A, and work on a task based on that value. When the task is complete, the caller will want to update the value to value B. However, a second concurrent caller might also fetch value A and start working, and eventually it may want to update the value to value C; thus, the callers’ updates are racing.

In order to avoid conflicting updates, the callers can use the `replace(K,V,V)` method. This performs an update and returns `true` only when the current value in the map

matches the first argument. Otherwise, the method does nothing and returns `false`. Callers are expected to pass the previous expected value as the first argument. The caller that “wins” the race will see the original value, as expected, and its update will be performed. The caller that “loses” will see a different value from its original value, no update will be performed, and the method will return `false`. At that point the “losing” caller is expected to retry its operation, possibly discarding work that it might have performed based on the original value.

Used together, these methods allow multiple concurrent callers to advance the value of a key-value pair through a state machine, computing optimistically while possibly discarding redundant results. The method `replace(K,V)` performs an unconditional state change, modifying an existing mapping without the risk of creating a new one. (Its complement is `putIfAbsent`, which will not modify an existing mapping but only create a new one.) The `remove(K,V)` method effects a conditional state change to a terminal state in which the mapping is removed.

Factory Methods

These methods create unmodifiable `Map` objects. “[UnmodifiableMap](#)” on page 262 discusses them in detail.

The Interface `Map.Entry`

The members of the set returned by `entrySet` implement the interface `Map.Entry`, representing a key-value association. This interface exposes factory methods for creating `Comparators` by key and value and instance methods for accessing the components of the entry. An optional `setValue` method can be used to change the value in an entry if the backing map is modifiable, and if so will write changes through. According to the Javadoc, a `Map.Entry` object obtained by iterating over a set returned by `entrySet` retains its connection only for the duration of the iteration, but in fact this is only a guaranteed minimum: implementation behaviors vary, some preserving the connection indefinitely. However, modern idioms for map processing depend less than before on durable `Map.Entry` objects: where in the past you might have used a `Map.Entry` set iterator to remove some entries, you would now be more likely to call `entrySet.removeIf`. Alternatively, and also for use cases in which you want to change association values, the instance methods of `Map`, listed in the previous section, provide a variety of ways to remove mappings or alter their values.

You can create `Map.Entry` objects using the `Map.entry` factory method; this is most commonly useful for creating unmodifiable `Maps`, discussed in “[UnmodifiableMap](#)” on page 262.

Using the Methods of Map

One problem with basing the to-do manager on priority queues, as we have done in the last two chapters, is that priority queues are unable to preserve the order in which mappings are added to them (unless that can be incorporated in the priority ordering, for example as a timestamp or sequence number). To avoid this, we could use as an alternative model a series of FIFO queues, each one assigned to a single priority. A Map would be suitable for holding the association between priorities and task queues; for example, we might use `EnumMap`, a highly efficient Map implementation specialized for use with keys that are members of an enum.

This model will rely on a Queue implementation that maintains FIFO ordering. To focus on the use of the Map methods, let's assume a single-threaded client and use a series of `ArrayDeques` as the implementation:

[org/jgcbook/chapter15/C_using_the_methods_of_map/Program_1](#)

```
var taskMap = new EnumMap<Priority,Queue<Task>>(Priority.class);
for (Priority p : Priority.values()) {
    taskMap.put(p, new ArrayDeque<Task>());
}
// populate the queues, for example:
taskMap.get(Priority.MEDIUM).add(new PhoneTask("Mike", "987 6543"));
taskMap.get(Priority.HIGH).add(new CodingTask("db"));
```

Now, to get to one of the task queues—say, the one with the highest-priority tasks—we can write:

[org/jgcbook/chapter15/C_using_the_methods_of_map/Program_1](#)

```
Queue<Task> highPriorityTaskList = taskMap.get(Priority.HIGH);
```

Polling this queue will now give us the high-priority to-dos, in the order in which they were entered into the system.

To see the use of some other methods of Map, let's extend the example a little to allow for the possibility that some of these tasks might actually earn us some money by being billable. One way of representing this would be by defining a class `Client`:

```
class Client {...}
Client acme = new Client("Acme Corp.", ...);
```

and creating a mapping from tasks to clients:

```
Map<Task,Client> billingMap = new HashMap<>();
billingMap.put(interfaceCode, acme);
```

Only billable tasks will appear in the domain of `billingMap`. Now, as part of the code for processing a task `t`, we can write:

```
Task t = ...
Client client = billingMap.get(t);
if (client != null) {
    client.bill(t);
}
```

When we have finally finished all the work we were contracted to do by our client Acme Corp., there are various options for removing the map entries that associate tasks with Acme. Perhaps the clearest one is to iterate over the entries of the map, testing each for its value element:

```
billingMap.entrySet().removeIf(e -> e.getValue().equals("acme"))
```

The `values` view of `billingMap` can contain duplicate clients, if there are multiple tasks associated with the same client. If we want to find the set of clients that need to be billed, we can copy and deduplicate the clients from the `values` view using a `Set` constructor:

```
Set<Client> billedClients = new HashSet<>(billingMap.values());
```

Extending the example further, suppose you have created a set of tasks that can only be accomplished when you're on-site at the client:

```
Set<Task> onsiteTasks = ...
```

Now you want to determine which clients have on-site, billable tasks so that you can schedule visits to them. You could of course build the set of on-site clients by iterating the entry set obtained from `billingMap`, including the client extracted from each entry if the corresponding task was contained in `onsiteTasks`. However, it's more concise, and a more modern idiom, to copy `billingMap`, use a destructive bulk operation, `retainAll`, on its key set, and then deduplicate the clients:

```
Map<Task,Client> onsiteMap = new HashMap<>(billingMap);
onsiteMap.keySet().retainAll(onsiteTasks);
Set<Client> clientsToVisit = new HashSet<>(onsiteMap.values());
```

Stream Alternative

Streaming the entry set is also a common idiom. For example, you could create a map associating each client with the list of their on-site tasks:

```
Map<Client,List<Task>> tasksByClient = billingMap.entrySet().stream()
    .filter(entry -> onsiteTasks.contains(entry.getKey()))
    .collect(Collectors.groupingBy(Map.Entry::getKey));
```

Map Implementations

The implementations, nine in all, that the Collections Framework provides for Map are shown in [Figure 15-1](#). We shall discuss HashMap, WeakHashMap, IdentityHashMap, EnumMap, and *UnmodifiableMap* here; the class LinkedHashMap and the interfaces NavigableMap, SequencedMap, ConcurrentMap, and ConcurrentNavigableMap are discussed, along with their implementations, in the following sections.

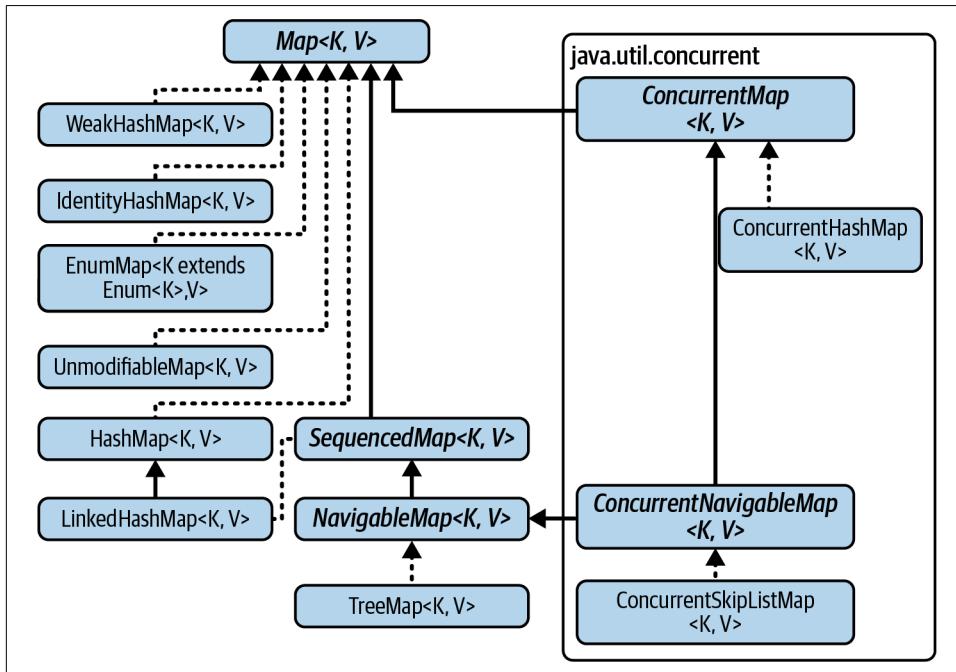


Figure 15-1. The structure of Map implementations in the Collections Framework

For constructors, the general rule for Map implementations is like that for Collection implementations (see “[Collection Constructors](#)” on page 176). Every implementation except for *EnumMap* has at least two constructors; taking *HashMap* as an example, they are:

```
public HashMap()
public HashMap(Map<? extends K, ? extends V> m)
```

The first of these creates an empty map, and the second a map that will contain the key-value mappings contained in the supplied map *m*. The keys and values of map *m* must have types that are the same as (or are subtypes of) the keys and values, respectively, of the map being created. Using this second constructor has the same effect as creating an empty map with the default constructor and then adding the contents of map *m* using *putAll*. In addition to these two constructors, *HashMap*, *LinkedHashMap*,

and `WeakHashMap` have other constructors for configuration purposes, but these have been replaced in practice by factory methods, for the reasons described in “[HashSet constructors](#)” on page 186).

HashMap

In “[HashSet](#)” on page 183, we mentioned that `HashSet` is implemented by a specialized `HashMap`; this specialization consists simply of storing all keys against the same constant value. So [Figure 15-2\(a\)](#) is similar to [Figure 12-1](#), but without the simplification that removed the value elements from the map. The discussion of hash tables and their performance in “[Set Implementations](#)” on page 183 applies equally to `HashMap`. In particular, `HashMap` provides constant-time performance for `put` and `get`. Although in principle constant-time performance is only attainable with no collisions, it can be closely approached by the use of rehashing to control the load and thereby to minimize the number of collisions.

Iteration over a collection of keys or values requires time proportional to the capacity of the map plus the number of key-value mappings that it contains. The iterators are fail-fast.

The factory method to use when you can predict the maximum number of entries is `newHashMap`:

```
static <K,V> HashMap<K,V> newHashMap(int numElements)
```

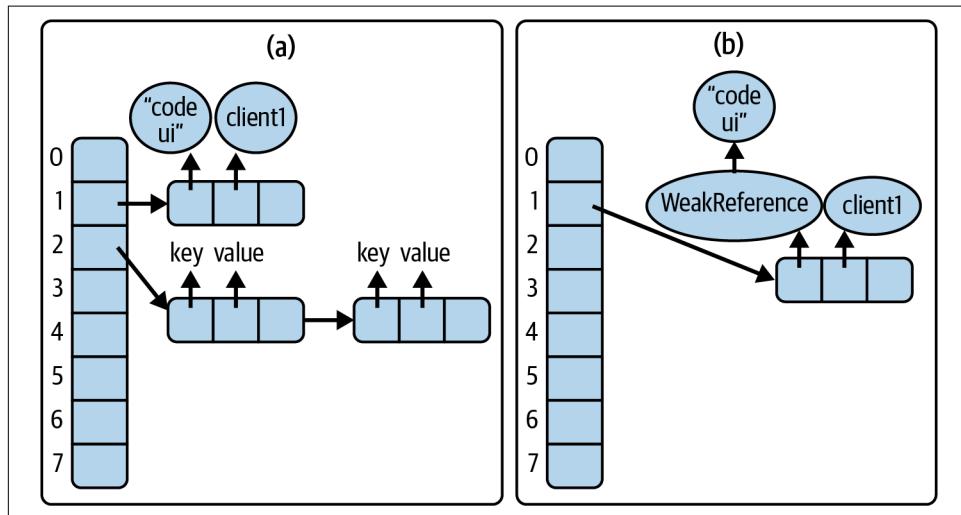


Figure 15-2. (a) `HashMap` and (b) `WeakHashMap`

WeakHashMap

Most maps keep ordinary (“strong”) references to all the objects they contain. That means that even when a key has become unreachable by any means other than through the map itself, its mapping cannot be garbage collected. In the example at the beginning of this chapter, that is the situation of task-client mappings: they reference both task and client objects, both of which occupy memory, so preserving them unnecessarily has the potential to degrade garbage collection performance and create memory leaks. The idea behind `WeakHashMap` is to avoid this situation by allowing a mapping and its referenced objects to be reclaimed once the key is no longer reachable in the application.

Internally, `WeakHashMap` holds references to its key objects through objects of the class `java.lang.ref.WeakReference`. A `WeakReference` introduces an extra level of indirection in reaching an object. [Figure 15-2\(b\)](#) shows the situation. A weak reference does not protect an object from garbage collection; in this case, if the task object “code ui” is not reachable with a normal reference from anywhere else in the application, then it is eligible for garbage collection. The map detects this and removes the entry, with the effect that the entire map entry will seem to have spontaneously disappeared.

What is a `WeakHashMap` good for? Imagine you have a program that allocates some transient system resource—a buffer, for example—on request from a client. Besides passing a reference to the resource back to the client, your program might also need to store information about it locally—for example, associating the buffer with the client that requested it. That could be implemented by means of a map from resource to client objects. But with a strong reference, then even after the client has disposed of the resource, the map will still hold a reference that will prevent the resource object from being garbage collected. Memory will gradually be used up by resources that are no longer in use. On the other hand, if the reference is weak, held by a `WeakHashMap`, the garbage collector will be able to reclaim the objects once they are no longer strongly referenced, so the memory leak is prevented.

A more general use is in those applications—for example, caches—where you don’t mind information disappearing if memory is low. `WeakHashMap` isn’t perfect for this purpose. One of its drawbacks is that it weakly references the map’s keys rather than its values, which usually occupy much more memory; so even after the garbage collector has reclaimed a key, the real benefit in terms of available memory will not be experienced until the map has removed the stale entry. A second drawback is that weak references are *too* weak: the garbage collector is liable to reclaim a weakly reachable object at any time, and the programmer cannot influence this in any way. (A sister class of `WeakReference`, `java.lang.ref.SoftReference`, is treated differently: the garbage collector postpones reclaiming these until it is under severe

memory pressure. [Heinz Kabutz \(2004\)](#) has written a `SoftReference`-based map that will work better as a cache.

`WeakHashMap` performs similarly to `HashMap`, though more slowly because of the overheads of the extra level of indirection for keys. The cost of clearing out unwanted key-value associations before each operation is proportional to the number of associations that need to be removed because the garbage collector has reclaimed the key. The iterators over collections of keys and values returned by `WeakHashMap` are fail-fast.

The factory method to use when you can predict the maximum number of entries is `newWeakHashMap`:

```
static <K,V> WeakHashMap<K,V> newWeakHashMap(int numMappings)
```

IdentityHashMap

The distinguishing characteristic of `IdentityHashMap` is discussed in [“Defining a Set: Equivalence Relations” on page 182](#): for the equivalence relation on its keys, it uses the identity relation. In other words, every physically distinct object is a distinct key.

An `IdentityHashMap` differs from an ordinary `HashMap` in that two keys are considered equal only if they are physically the same object: `identity`, rather than `equals`, is used for key comparison. That sets the contract for `IdentityHashMap` at odds with the contract for the interface it implements, namely `Map`, which specifies that equality should be used for key comparison. The problem here is exactly analogous to the one we encountered for `Set`: the Javadoc for `Map` assumes that the equivalence relation for maps is always defined by the `equals` method, whereas in fact `IdentityHashMap` uses a different relation, as do all implementations of `NavigableMap`, as we shall see in [“NavigableMap” on page 267](#).

The main purpose of `IdentityHashMap` is to support operations in which a graph has to be traversed and information stored about each node. Serialization is one such operation. The algorithm used for traversing the graph must be able to check, for each node it encounters, whether that node has already been seen; otherwise, graph cycles could be followed indefinitely. For cyclic graphs, we must use identity rather than equality to check whether nodes are the same. Calculating equality between two graph node objects requires calculating the equality of their fields, which in turn means computing all their successors—and we are back to the original problem. An `IdentityHashMap`, by contrast, will report a node as being present only if that same node has previously been put into the map.

The standard implementation of `IdentityHashMap` handles collisions differently than the chaining method shown in [Figure 12-1](#) and used by all the other variants of `HashSet` and `HashMap`. This implementation uses a technique called *linear probing*, in

which the key and value references are stored directly in adjacent locations in the table itself rather than in cells referenced from it. With linear probing, collisions are handled by simply stepping along the table until the first free pair of locations is found. In fact, this is still chaining, but in a different physical form. [Figure 15-3](#) shows three stages in filling an `IdentityHashMap` with a capacity of 8. In (a) we are storing a key-value pair whose key hashes to 0, and in (b) a pair whose key hashes to 4. The third key, added in (c), also hashes to 4, so the algorithm steps along the table until it finds an unused location. In this case, the first one it tries, with index 6, is free and can be used.

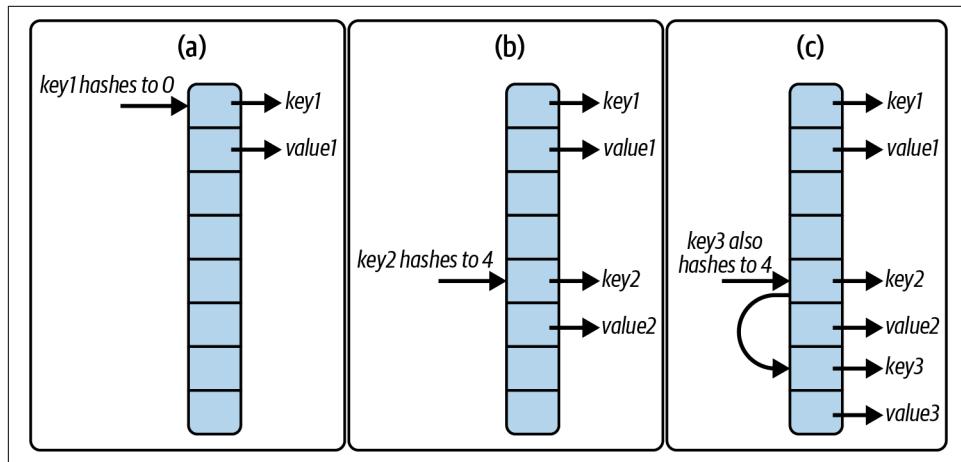


Figure 15-3. Resolving collisions by linear probing

Deletions are trickier than with chaining; if `key2` and `value2` were removed from the table in [Figure 15-3](#), `key3` and `value3` would have to be moved along to take their place. This requires copying; an alternative that avoids this is to replace the deleted entry with a dummy element called a *tombstone marker*, which maintains the chain but can be skipped during searches and can be overwritten for a subsequent insertion.

Why does `IdentityHashMap` use linear probing when `HashMap` and `ConcurrentHashMap` use linked chaining? Each technique has its advantages. Indexing into an array is in general faster than following a linked list, and has better spatial locality (see [“Memory” on page 143](#)). Linked chaining is more resilient in the case of multiple entries sharing the same bucket, usually because of a poorly designed hashing function. This advantage is not relevant to `IdentityHashMap`, since the hash code it uses is implemented in `Object` and cannot be changed by the user. A second advantage of linked chaining is that nodes on the chain can be defined appropriately to hold the hash code of the payload, along with the chain pointer(s) and the payload itself. This is for reasons of efficiency: a `get` operation must check whether it has found the right

key, and since equality is in general an expensive operation, it makes sense to check first whether it even has the right hash code. Again, this advantage does not apply to `IdentityHashCode`, which checks object identity rather than object equality and so has no need to store object hash codes in the table.

There are three constructors for `IdentityHashMap`:

```
public IdentityHashMap()
public IdentityHashMap(Map<? extends K,? extends V> m)
public IdentityHashMap(int expectedMaxSize)
```

The first two are the standard constructors found in every general-purpose `Map` implementation. The third takes the place of the two constructors that in other hash-based collections allow the user to control the initial capacity of the table and the load factor at which it will be rehashed. `IdentityHashMap` doesn't allow this, fixing it instead (at .67 in the standard implementation) in order to protect the user from the consequences of setting the load factor inappropriately. This is because, whereas the cost of finding a key in a table using chaining is proportional to the load factor l , in a table using linear probing it is proportional to $1/(1 - l)$. So avoiding high load factors is too important to be left to the programmer! This decision is in line with the policy, mentioned earlier, of no longer providing configuration parameters when new classes are introduced into the Collections Framework (see “[HashSet constructors](#)” on page 186).

EnumMap

Implementing a mapping from an enumerated type is straightforward and very efficient, for reasons similar to those described for `EnumSet` (see “[EnumSet](#)” on page 187). In an array implementation, the ordinal value of each enumerated type constant can serve as the index of the corresponding value. The basic operations of `get` and `put` can be implemented as array accesses, in constant time. An iterator over the key set takes time proportional to the number of constants in the enumerated type and returns the keys in their natural order (the order in which the enum constants are declared). Iterators over the collection views of this class are weakly consistent.

`EnumMap` has three public constructors:

```
EnumMap(Class<K> keyType)
EnumMap(EnumMap<K,> V> m)
EnumMap(Map<K,> V> m)
```

The first one creates an empty enum map, and the second creates an enum map with the same key type and elements as `m`. The third constructor creates an enum map using the elements of the supplied `Map`, which (unless it is an `EnumMap`) must contain at least one association.

Compile-time type safety requires that K be an enum type. This is guaranteed by the declaration of the class `EnumMap`:

```
Class EnumMap<K extends Enum<K>,V>
```

This ensures that the key is of the correct type, using the same recursive type pattern as for the class `Enum` (see “[Enumerated Types](#)” on page 50).

An `EnumMap` contains a reified key type, which is used at run time for checking the validity of new entries being added to the map. This type is supplied by the three constructors in different ways. In the first, it is supplied as a class object (see “[A Classy Alternative](#)” on page 84); in the second, it is copied from the specified `EnumMap`. In the third, there are two possibilities: either the specified `Map` is actually an `EnumMap`, in which case it behaves like the second constructor, or the class of the first key of the specified `Map` is used (which is why the supplied `Map` may not be empty).

UnmodifiableMap

The family of implementations that we’re calling `UnmodifiableMap<K,V>` is the last of the unmodifiable collections that we will meet in this book. It shares many characteristics with its sister families, `UnmodifiableSet` and `UnmodifiableList`: keys and values cannot be added, removed, or updated, and `null` values are disallowed. The properties of the members of this family are described in the Javadoc for `Map`:

- They are unmodifiable; keys and values cannot be added or removed. Calling any mutator method will always cause `UnsupportedOperationException` to be thrown.
- They are `null`-hostile. Attempts to create them with `null` keys or values will result in a `NullPointerException`.
- They reject duplicate keys at creation time. Duplicate keys passed to a factory method result in an `IllegalArgumentException`.

The factory methods follow those for `UnmodifiableSet` and `UnmodifiableList`. The `of` method has 11 overloads, for creating unmodifiable maps containing up to 10 mappings:

<code><K,V> Map<K,V> of(K k1, V v1)</code>	return an unmodifiable map containing one mapping
<code><K,V> Map<K,V> of(K k1, V v1, K k2, V v2)</code>	return an unmodifiable map containing two mappings
<code><K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)</code>	return an unmodifiable map containing three mappings
further overloads, taking from four to nine key-value pairs	

```
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3,
V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7,
V v7, K k8, V v8, K k9, V v9, K k10, V v10)
```

return an unmodifiable map containing 10 mappings

And two further methods support the creation of unmodifiable maps from any map, or from an array or varargs-supplied list of mappings:

<pre><K,V> Map<K,V> ofEntries(Map.Entry<K,V>... entries)</pre>	return an unmodifiable map with keys and values extracted from <code>entries</code>
<pre><K,V> Map<K,V> copyOf(Map<K,V> map)</pre>	return an unmodifiable map containing the entries of the given map

Because varargs arguments must all be of the same type, while key and value types can differ, the varargs method for creating unmodifiable maps must accept `Map.Entry` elements instead. These can be created with the static convenience method `Map::entry`.

Like the other unmodifiable collections, unmodifiable maps are backed by fixed-length arrays. They have the advantages over hashed collections of reduced memory footprint, faster iteration, and better spatial locality. With a `hashCode` function that provides good distribution, lookup is $O(1)$. As with `UnmodifiableSet`, and for the same reason, the order of iteration over the entry or key set is randomly determined for each virtual machine instance.

SequencedMap

A `SequencedMap` is a `Map` that maintains its entries in a defined order. The position of `SequencedMap` in the hierarchy of `Map` types is shown in [Figure 15-4](#).

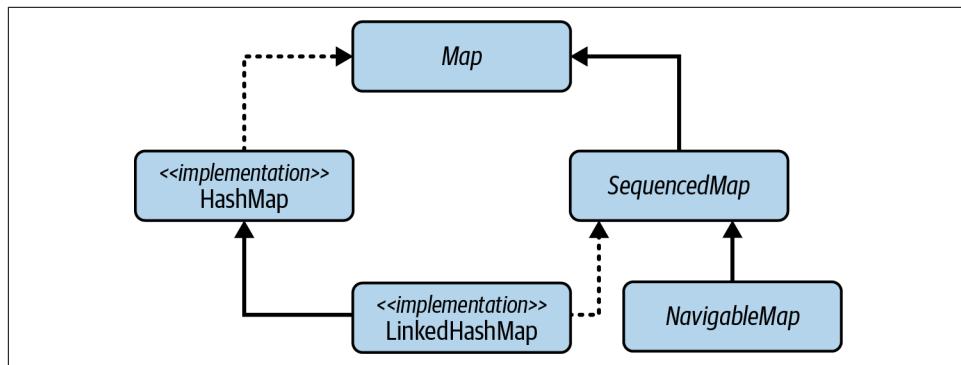


Figure 15-4. SequencedMap and related types

The Methods of SequencedMap

This interface exposes methods to add or inspect the first or last key or entry of a map, and to remove the first or last entry.

Adding or updating entries

`V putFirst(K k, V v)` insert the given mapping, or updates it if it is already present

`V putLast(K k, V v)` insert the given mapping, or updates it if it is already present

These methods add an entry to the start or end of a `SequencedMap`, respectively, or update it if it is already present. In the case of an internally ordered implementation like `TreeMap`, both methods will throw `UnsupportedOperationException`.

The contracts for these methods specify that after their successful execution, the map must contain the supplied entry as the first (or last) entry in the encounter order.

Inspecting entries

`Map.Entry<K,V> firstEntry()` return the first entry, or `null` if the map is empty

`Map.Entry<K,V> lastEntry()` return the last entry, or `null` if the map is empty

These are the methods for inspecting the entries at either end of a `SequencedMap`.

Removing entries

`Map.Entry<K,V> pollFirstEntry()` remove and return the first entry, or `null` if the map is empty

`Map.Entry<K,V> pollLastEntry()` remove and return the last entry, or `null` if the map is empty

These methods remove an entry from the start or end of a `SequencedMap`.

View-generating methods

`SequencedMap<K,V> reversed()` return a reverse-ordered view of the map

`SequencedSet<Map.Entry<K,V>> sequencedEntrySet()` return a `SequencedSet` view of the map's entry Set

`SequencedSet<K> sequencedKeySet()` return a `SequencedSet` view of the map's key Set

`SequencedCollection<V> sequencedValues()` return a `SequencedCollection` view of the map's values collection

These methods provide different views of the map.

LinkedHashMap

Like `LinkedHashSet` (see “[LinkedHashSet](#)” on page 192), the class `LinkedHashMap` refines the contract of its parent class, `HashMap`, by guaranteeing the order in which iterators return its elements. Also like `LinkedHashSet`, it implements the sequenced subinterface (`SequencedMap`) of its main interface. Unlike `LinkedHashSet`, however, `LinkedHashMap` offers a choice of iteration orders; elements can be returned either in the order in which they were inserted in the map—the default—or in the order in which they were accessed (from least recently to most recently accessed). An access-ordered `LinkedHashMap` is created by supplying an argument of `true` for the last parameter of the constructor:

```
public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
```

Supplying `false` will give an insertion-ordered map. The other constructors, which are just like those of `HashMap`, also produce insertion-ordered maps. As with `LinkedHashSet`, iteration over a `LinkedHashMap` takes time proportional only to the number of elements in the map, not its capacity.

Access-ordered maps are especially useful for constructing *least recently used* (LRU) caches. A cache is an area of memory that stores frequently accessed data for fast access. In designing a cache, the key issue is the choice of algorithm that will be used to decide what data to remove in order to conserve memory. When an item from a cached data set needs to be found, the cache will be searched first. Typically, if the item is not found in the cache, it will be retrieved from the main store and added to the cache. But the cache cannot be allowed to continue growing indefinitely, so a strategy must be chosen for removing the least useful item from the cache when a new one is added. If the strategy chosen is LRU, the entry removed will be the one least recently used. This simple strategy is suitable for situations in which access of an element increases the probability of further access in the near future of the same element. Its simplicity and speed have made it the most popular caching strategy. `LinkedHashMap` exposes a method specifically designed to make it easy to use as an LRU cache:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
```

The name of this method is misleading. It is not usually used to itself modify the map: instead, it is called from within the code of `put` or `putAll` each time an element is added. The value it returns is an indication to the calling method of whether it should remove the first entry in the map—that is, the one least recently accessed (or, if some entries have never been accessed, the one amongst these that was least recently added).

The implementation in `LinkedHashMap` simply returns `false`—an indication to the calling method that no action is needed. But you can subclass `LinkedHashMap` and override `removeEldestEntry` to return `true` under specific circumstances. The Java-doc gives the example of a map that should not grow past a given size:

```
org/jcbook/chapter15/F_LinkedHashMap/BoundedSizeMap
```

```
class BoundedSizeMap<K,V> extends LinkedHashMap<K,V> {
    private final int maxEntries;
    public BoundedSizeMap(int maxEntries) {
        super(16, 0.75f, true);
        this.maxEntries = maxEntries;
    }
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return size() > maxEntries;
    }
}
```

Because in an insertion-ordered `LinkedHashMap` the eldest entry will be the one that was least recently added to the map, overriding `removeEldestEntry` as shown here will implement a FIFO strategy. FIFO caching has often been used in preference to LRU because it is much simpler to implement in maps that do not offer access ordering. However, LRU is usually more effective than FIFO, because the reduced cost of cache refreshes outweighs the overhead of maintaining access ordering.

If you need to modify the very specific action for which `removeEldestEntry` is specialized, you can just treat it as a callback, adding whatever action you want and returning `false`. For example, if a method `isReservedKey` returned `true` for elements that should never be removed from the cache, you could implement `removeEldestEntry` like this:

```
org/jcbook/chapter15/F_LinkedHashMap/BoundedSizeMap_1
```

```
private boolean isReservedKey(K keyToRetain) { ... }
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    if (size() > maxEntries) {
        for (var itr = entrySet().iterator(); itr.hasNext(); ) {
            if (!isReservedKey(itr.next().getKey())) {
                itr.remove();
                return false;
            }
        }
    }
    return false;
}
```

From Java 21, `LinkedHashMap` has implemented `SequencedMap`, providing much greater flexibility in caching policies. For example, in some applications recent access to an entry reduces rather than increases the likelihood of it being accessed again soon. In that case, the best strategy is *most recently used* (MRU), which discards the

most recently used entry. This is straightforward to implement in a `SequencedMap`, which exposes a method `pollLastEntry` analogous to `SequencedSet.removeLast`:

org/jgcbook/chapter15/F_linkedhashmap/BoundedSizeMap_2

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {  
    if (size() > maxEntries) {  
        pollLastEntry();  
    }  
    return false;  
}
```

Extending this example one last time, we can combine the previous two variations in a bounded MRU cache with reserved names using the method `sequencedKeySet`, which provides a `SequencedSet` (see “[“SequencedSet” on page 191](#)”) view of the keys:

org/jgcbook/chapter15/F_linkedhashmap/BoundedSizeMap_3

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {  
    if (size() > maxEntries) {  
        for (var itr = sequencedKeySet().reversed().iterator() ; itr.hasNext() ; ) {  
            if (!isReservedKey(itr.next())) {  
                itr.remove();  
                return false;  
            }  
        }  
    }  
    return false;  
}
```

These examples should not be taken as realistic except for the simplest uses; if you subclass `LinkedHashMap` in a real application, you will probably want more flexibility than can be provided by overriding `removeEldestEntry`. In that case, it would make more sense to override `put` and `putAll`—or indeed, to provide other methods to allow users to adjust the cache, for example by using the `SequencedMap` methods `putFirst` or `putLast` to relocate entries to the beginning or end of the map.

Iteration over a collection of keys or values returned by a `LinkedHashMap` is linear in the number of elements. The iterators over such collections are fail-fast.

The factory method to use when you can predict the maximum number of entries is `newLinkedHashMap`:

```
static <K,V> LinkedHashMap<K,V> newLinkedHashMap(int numMappings)
```

NavigableMap

The interface `NavigableMap` adds to `SequencedMap` a guarantee that its iterator will traverse the map in ascending key order and, like `NavigableSet`, adds further methods to find the entries adjacent to a target key value. Also like `NavigableSet`, `NavigableMap` extends and in effect replaces an older interface, `SortedMap`, which

imposes an ordering on its keys: either their natural ordering or that of a Comparator. The equivalence relation on the keys of a NavigableMap is again defined by the ordering relation; two keys that compare as equal—that is, for which the comparison method returns 0—will be regarded as duplicates by a NavigableMap (see “[Defining a Set: Equivalence Relations](#)” on page 182).

The extra methods defined by the NavigableMap interface fall into four groups.

Retrieving the Comparator

<code>Comparator<? super K> comparator()</code>	return the map's key comparator if it has been given one, instead of relying on the natural ordering of the keys; otherwise, return null
---	--

This method can be useful when transferring data between two sorted collections; if the comparators on which they were sorted is the same, no resorting is required.

Getting Range Views

<code>SortedMap<K,V> subMap(K fromKey, K toKey)</code>	return a view of the portion of this map whose keys range from <code>fromKey</code> , inclusive, to <code>toKey</code> , exclusive
<code>SortedMap<K,V> headMap(K toKey)</code>	return a view of the portion of this map whose keys are strictly less than <code>toKey</code>
<code>SortedMap<K,V> tailMap(K fromKey)</code>	return a view of the portion of this map whose keys are greater than or equal to <code>fromKey</code>
<code>NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)</code>	return a view of the portion of this map whose keys range from <code>fromKey</code> to <code>toKey</code>
<code>NavigableMap<K,V> headMap(K toKey, boolean inclusive)</code>	return a view of the portion of this map whose keys are less than (or equal to, if <code>inclusive</code> is true) <code>toKey</code>
<code>NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)</code>	return a view of the portion of this map whose keys are greater than (or equal to, if <code>inclusive</code> is true) <code>fromKey</code>

As with NavigableSet, each of the methods in this group appears in two overloads, one inherited from SortedMap and returning a half-open SortedMap view, and one defined in NavigableMap and returning a NavigableSet view that can be open, half-open, or closed according to the user's choice. These operations work like the corresponding operations in SortedSet: the key arguments do not themselves have to be present in the map, and the sets returned include the `fromKey`—if, in fact, it is present in the map—but not the `toKey`.

Like the `NavigableSet` methods, the `NavigableMap` methods provide more flexibility than the range-view methods of `SortedMap`. Instead of always returning a half-open interval, these methods accept boolean parameters that are used to determine whether to include the key or keys defining the interval.

Getting Closest Matches

<code>Map.Entry<K,V> ceilingEntry(K Key)</code>	return a key-value mapping associated with the least key greater than or equal to the given key, or <code>null</code> if there is no such key
<code>K ceilingKey(K Key)</code>	return the least key greater than or equal to the given key, or <code>null</code> if there is no such key
<code>Map.Entry<K,V> floorEntry(K Key)</code>	return a key-value mapping associated with the greatest key less than or equal to the given key, or <code>null</code> if there is no such key
<code>K floorKey(K Key)</code>	return the greatest key less than or equal to the given key, or <code>null</code> if there is no such key
<code>Map.Entry<K,V> higherEntry(K Key)</code>	return a key-value mapping associated with the least key strictly greater than the given key, or <code>null</code> if there is no such key
<code>K higherKey(K Key)</code>	return the least key strictly greater than the given key, or <code>null</code> if there is no such key
<code>Map.Entry<K,V> lowerEntry(K Key)</code>	return a key-value mapping associated with the greatest key strictly less than the given key, or <code>null</code> if there is no such key
<code>K lowerKey(K Key)</code>	return the greatest key strictly less than the given key, or <code>null</code> if there is no such key

These are similar to the corresponding closest-match methods of `NavigableSet`, but they return `Map.Entry` objects. If you want the key belonging to one of these entries, use the corresponding convenience key-returning method, with the performance benefit of allowing the map to avoid the unnecessary creation of a `Map.Entry` object.

Other Views

<code>NavigableMap<K,V> descendingMap()</code>	return a reverse-order view of the map
<code>NavigableSet<K> descendingKeySet()</code>	return a reverse-order key set
<code>NavigableSet<K> navigableKeySet()</code>	return a forward-order key set

The `SequencedMap` method `reversed` is inherited by `NavigableMap`, with a covariant override to allow it to return a `NavigableMap`. This is a synonym for `descendingMap`, which remains for historical reasons, having been present before `NavigableMap` was retrofitted to inherit from `SequencedMap`. Similarly, the method `descendingKeySet` has almost the same effect as calling `reversed` on the key set of a `SequencedMap`, but again returns the more precise type of `NavigableSet`.

You might wonder why the method `keySet`, inherited from `Map`, could not simply be overridden using a covariant return type to return a `NavigableSet`. Indeed, the platform implementations of `NavigableMap::keySet` do return a `NavigableSet`. But there is a compatibility concern: if `TreeMap::keySet` were to have its return type changed from `Set` to `NavigableSet`, any existing `TreeMap` subclasses that override that method would fail to compile unless they too changed their return type. (This concern is similar to those discussed in “[Maintain Binary Compatibility](#)”.)

TreeMap

`NavigableMap` is implemented in the Collections Framework by `TreeMap`. We met trees as a data structure for storing elements in order when we discussed `TreeSet` (see “[TreeSet](#)” on page 200). In fact, the internal representation of a `TreeSet` is just a `TreeMap` in which every key is associated with the same standard value, so the explanation of the mechanism and performance of red-black trees given there applies equally here.

The constructors for `TreeMap` include, besides the standard ones, one that allows you to supply a `Comparator` and one that allows you to create a `TreeMap` from another `NavigableMap` (strictly speaking, from a `SortedMap`), using both the same comparator and the same mappings:

```
public TreeMap(Comparator<? super K> comparator)
public TreeMap(SortedMap<K, ? extends V> m)
```

Notice that the second of these constructors, which is defined so as to allow the new map to accept the ordering of the one supplied, suffers from a similar problem to the corresponding constructor of `TreeSet`: the standard conversion constructor—the one that takes a `Map` as its argument—always uses the natural ordering of the keys, so if you supply a reference to a `SortedMap` to the conversion constructor of `TreeMap`, the ordering of the constructed map will depend on the static type of that reference.

`TreeMap` has similar performance characteristics to `TreeSet`: the basic operations (`get`, `put`, and `remove`) perform in $O(\log N)$ time. The collection view iterators are fail-fast.

ConcurrentMap

Maps are often used in high-performance server applications—for example, as cache implementations—so a high-throughput thread-safe map implementation is an essential part of the Java platform. This requirement cannot be met by synchronized maps such as those provided by `Collections::synchronizedMap`, because with full synchronization each operation needs to obtain an exclusive lock on the entire map, effectively serializing access to it. Locking only a part of the collection at a time—*lock*

striping—can achieve very large gains in throughput, as we shall see shortly with `ConcurrentHashMap`. But because there is no single lock for a client to hold to gain exclusive access, client-side locking no longer works, and clients need assistance from the collection itself to carry out atomic actions.

That was the purpose of the interface `ConcurrentMap`, when it was introduced along with the other concurrent types in Java 5. It provided declarations for four methods—`putIfAbsent`, `remove`, and two overloads of `replace`—that perform compound operations atomically. At Java 8, new compound operations were introduced (see “[Compound Operations](#)” on page 250), and `ConcurrentMap` was provided with default implementations for these. However, the existing four compound methods were promoted to the `Map` interface, so `ConcurrentMap` no longer exposes any new functionality.

ConcurrentHashMap

The class `ConcurrentHashMap` provides an implementation of `ConcurrentMap` and offers an effective solution to the problem of reconciling throughput with thread safety. It is optimized for reading, so retrievals do not block even while the table is being updated. To allow for this, the contract states that the results of retrievals will reflect the latest update operations completed before the start of the retrieval. Concurrent updates can proceed safely, even while the table is being resized.

The constructors for `ConcurrentHashMap` are similar to those of `HashMap`, but with an extra one that provides the programmer with the ability to hint to the implementation the expected number of concurrently updating threads (its *concurrency level*):

```
ConcurrentHashMap()
ConcurrentHashMap(int initialCapacity)
ConcurrentHashMap(int initialCapacity, float loadFactor)
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)
ConcurrentHashMap(Map<? extends K,? extends V> m)
```

`ConcurrentHashMap` is a useful implementation of `Map` in any concurrent application where it is unnecessary to lock the entire table; this is the one capability of synchronized maps (see “[Synchronized Collections](#)” on page 281) that it does not support. This affects the results of aggregate status methods such as `size`, `isEmpty`, and `containsValue`. A map cannot be locked as a whole, so if it is undergoing concurrent updates it will be changing while these methods are working. In that case, their results will not reflect any consistent state and should be treated as approximations.

As with other maps, the static method `Collections::newSetFromMap` can be used to create a concurrent set from a `ConcurrentHashMap` with the same ordering, concurrency, and performance (see “[Set Views of Maps](#)” on page 191). In addition, `ConcurrentHashMap` has an inner class, `ConcurrentHashMap.KeySetView`, that not only provides set operations but also exposes methods that give access to the backing

map, allowing it to be managed via the view. Two of these methods are overloads of `Map::keySet`, with a covariantly overridden return type of `KeySetView`. The other two are overloads of a factory method `newKeySet`, which creates a new empty map that can be managed—even having key-value pairs added—via the returned `KeySet View`.

Disregarding locking overheads, the cost of the operations of `ConcurrentHashMap` are similar to those of `HashMap`. The collection views return weakly consistent iterators.

ConcurrentNavigableMap

`ConcurrentNavigableMap` inherits from both `ConcurrentMap` and `NavigableMap`. It contains just the methods of these two interfaces, with a few changes to make the return types more precise. First, the range-view methods inherited from `SortedMap` and `NavigableMap` now return views of type `ConcurrentNavigableMap`. The compatibility concerns that prevented `NavigableMap` from overriding the methods of `Sorted Map` don't apply to overriding the range-view methods of `NavigableMap` or `SortedMap`; because neither of these has any implementations that have been retrofitted to the new interface, the danger of breaking implementation subclasses does not arise. For the same reason, it is now possible to override `keySet` to return `NavigableSet`.

ConcurrentSkipListMap

The relationship between `ConcurrentSkipListMap` and `ConcurrentSkipListSet` is like that between `TreeMap` and `TreeSet`. A `ConcurrentSkipListSet` is implemented by a `ConcurrentSkipListMap` in which every key is associated with the same standard value, so the mechanism and performance of the skip list implementation given in “[ConcurrentSkipListSet](#)” on page 203 applies equally here: the basic operations (`get`, `put`, and `remove`) have $O(\log N)$ complexity, and iterators over the collection views execute `next` in constant time. These iterators are weakly consistent.

Comparing Map Implementations

[Table 15-1](#) shows the relative performance of the different platform implementations of `Map` (the column headed “`next`” shows the cost of the `next` operation of iterators over the key set). As with the implementations of `Queue`, your choice of map class is likely to be influenced more by the functional requirements of your application and the concurrency properties that you need than by performance considerations.

Table 15-1. Comparative performance of different Map implementations

	get	containsKey	next	Notes
HashMap	$O(1)$	$O(1)$	$O(h/N)$	h is the table capacity
WeakHashMap	$O(1)$	$O(1)$	$O(h/N)$	h is the table capacity
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$	
IdentityHashMap	$O(1)$	$O(1)$	$O(h/N)$	h is the table capacity
EnumMap	$O(1)$	$O(1)$	$O(1)$	
TreeMap	$O(\log N)$	$O(\log N)$	$O(\log N)$	
ConcurrentHashMap	$O(1)$	$O(1)$	$O(h/N)$	h is the table capacity
ConcurrentSkipListMap	$O(\log N)$	$O(\log N)$	$O(1)$	

Some specialized situations dictate the implementation. `EnumMap` should always (and only) be used for mapping from enums. Problems such as the graph traversals described in “[IdentityHashMap](#)” on page 259 call for `IdentityHashMap`. For a sorted map, use `TreeMap` where thread safety is not required, and `ConcurrentSkipListMap` otherwise.

That leaves the choice of implementation for general-purpose maps. For concurrent applications, `ConcurrentHashMap` is the only choice. Otherwise, favor `LinkedHashMap` over `HashMap` (and accept its slightly worse performance) if you need to make use of the insertion or access order of the map—for example, to use it as a cache.

Conclusion

In this chapter, we explored `Map` and its subinterface `ConcurrentMap`, which provides the best-developed API for concurrent operations offered by the Collections Framework. Implementations of `ConcurrentMap` are widely used in high-performance enterprise applications.

That concludes our survey of the major interfaces and classes of the Collections Framework. But there is more to know: many of the most useful algorithms of the framework are in static methods of the utility class `Collections`. They were mostly placed there before Java 8 introduced static methods on interfaces, but their concentration in the `Collections` class has the advantage of making them findable; in general, this is the place you go to when you need a collection utility. That is the subject of the next chapter.

The Collections Class

The class `java.util.Collections` consists entirely of static methods that operate on or return collections. There are three main categories: generic algorithms, methods that return empty or prepopulated collections, and methods that create wrappers. We'll discuss each of these in turn, then consider a number of other methods that do not fit into a neat classification.

All the methods of `Collections` are public and static, so for readability we will omit these modifiers from the individual declarations.

Generic Algorithms

The generic algorithms fall into four major categories: changing element order in a list, changing the contents of a list, finding extreme values in a collection, and finding specific values in a list. They represent reusable functionality, in that they can be applied to `Lists` (or in some cases to `Collections`) of any type. Generifying the types of these methods has led to some fairly complicated declarations, so each section discusses the declarations briefly after presenting them.

The choice of algorithm used by the methods that act on `Lists` often depends on whether the `List` being processed implements the marker interface `RandomAccess`. Classes implement this interface to indicate to generic methods that a long list of that class is more efficiently processed using `get` than by using an iterator. `ArrayList` implements `RandomAccess`; `LinkedList` does not.

Changing the Order of List Elements

void reverse(List<?> list)	reverse the order of the elements
void rotate(List<?> list, int distance)	rotate the elements of the list; the element at index <i>i</i> is moved to index $(distance + i) \% list.size()$
void shuffle(List<?> list)	randomly permute the list elements
void shuffle(List<?> list, Random rnd)	randomly permute the list using the randomness source <i>rnd</i>
void shuffle(List<?> list, RandomGenerator rndGen)	randomly permute the list using the randomness generator <i>rndGen</i>
<T extends Comparable<? super T>	sort the supplied list using natural ordering
void sort(List<T> list)	
<T> void sort(List<T> list, Comparator<? super T> c)	sort the supplied list using the supplied ordering
void swap(List<?> list, int i, int j)	swap the elements at the specified positions

The simplest of these methods for reordering lists is `swap`, which exchanges two elements and, in the case of a `List` that implements `RandomAccess`, executes in constant time. The most complex is `sort`, which transfers the list elements into an array, sorts them in a worst-case time of $O(N \log N)$, and then returns them to the list. All of the remaining methods execute in time $O(N)$.

For each of these methods (except `sort` and `swap`), there are two algorithms, one using `ListIterator` and the other using `get` and `set`. The method `sort` delegates to `List::sort`, which in the default implementation transfers the list elements to an array, although `ArrayList` overrides this to sort in place. The array is then sorted, maintaining a stable order, using—in the case of the JDK—the [timsort algorithm](#), with a worst-case time of $N \log N$. The method `swap` always uses random access. The standard implementations for the other methods in this section also use either iteration or random access, depending on whether the list implements the `RandomAccess` interface. If it does, the implementation chooses the random-access algorithm; even for a list that does not implement `RandomAccess`, however, the random-access algorithms are used if the list size is below a given threshold, determined on a per-method basis by performance testing.

Changing the Contents of a List

<T> void copy(List<? super T> dest, List<? extends T> src)	copy all of the elements from one list into another
<T> void fill(List<? super T> list, T obj)	replace every element of list with obj
<T> boolean replaceAll(List<T> list, T oldVal, T newVal)	replace all occurrences of oldVal in list with newVal

These methods change some or all of the elements of a list. The method `copy` transfers elements from the source list into an initial sublist of the destination list (which has to be long enough to accommodate them), leaving any remaining elements of the destination list unchanged. The method `fill` replaces every element of a list with a specified object, and `replaceAll` replaces every occurrence of one value in a list with another—where either the old or the new value can be `null`—returning `true` if any replacements were made.

The signatures of these methods can be explained using the Get and Put Principle (see “[The Get and Put Principle](#)” on page 20). The signature of `copy` was discussed in “[Wildcards with super](#)” on page 18. It gets elements from the source list and puts them into the destination, so the types of these lists are, respectively, `? extends T` and `? super T`. This fits with the intuition that the type of the source list elements should be a subtype of the destination list. Although there are simpler alternatives for the signature of `copy`, the discussion in [Chapter 2](#) makes the case that using wildcards where possible widens the range of permissible calls.

For `fill`, the Get and Put Principle dictates that you should use `super` if you are putting values into a parameterized collection, and for `replaceAll` it states that if you are putting values into and getting values out of the same structure, you should not use wildcards at all.

These methods are not often used in practice. `fill` and `copy` rely on the destination list already existing with the right number of elements. If you need to create a new list to receive the contents, a convenient alternative for `copy` is:

```
var dest = new ArrayList<>(src);
```

and for `fill` you can write:

```
var dest = new ArrayList<>(Collections.nCopies(N, obj));
```

The method `replaceAll` has largely been supplanted by the default method `List::replaceAll`.

Finding Extreme Values in a Collection

```
<T extends Object & Comparable<? super T>> return the maximum element using natural ordering  
T max(Collection<? extends T> coll)  
  
<T> T max(Collection<? extends T> coll, Comparator<? super T> comp) return the maximum element using the supplied comparator  
  
<T extends Object & Comparable<? super T>> return the minimum element using natural ordering  
min(Collection<? extends T> coll)  
  
<T> T min(Collection<? extends T> coll, Comparator<? super T> comp) return the minimum element using the supplied comparator
```

The methods `min` and `max` each have two overloads, one using the natural ordering of the elements and one accepting a `Comparator` to impose an ordering. Given an empty collection, they throw `NoSuchElementException`. Both execute in linear time.

[“Multiple Bounds” on page 53](#) and the section “Maintain Binary Compatibility” in the [Appendix](#) explain these methods and the types assigned to them.

Finding Specific Values in a List

```
<T> int binarySearch(List<? extends Comparable<? super T>> list, T key) search for key using binary search  
  
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c) search for key using binary search  
  
int indexOfSubList(List<?> source, List<?> target) find the first sublist of source that matches target  
  
int lastIndexOfSubList(List<?> source, List<?> target) find the last sublist of source that matches target
```

The methods in this group locate elements or groups of elements in a `List`, again choosing between alternative algorithms on the basis of the list’s size and whether it implements `RandomAccess`.

The signature of the first `binarySearch` overload says that you can use it to search for a key of type `T` in a list of objects that can have any type that can be compared with objects of type `T`. The second is like the `Comparator` overloads of `min` and `max`, except that in this case the type parameter of the `Collection` must be a subtype of the type of the key, which in turn must be a subtype of the type parameter of the `Comparator`.

Binary search requires a sorted list for its operation. At the start of a search, the range of indices in which the search value may occur corresponds to the entire list. The binary search algorithm samples an element in the middle of this range, using the

value of the sampled element to determine whether the new range should be the part of the old range above or the part below the index of the element. A third possibility is that the sampled value is equal to the search value, in which case the search is complete. Since each step halves the size of the range, m steps are required to find a search value in a list of length 2^m , and the time complexity for a `RandomAccess` list of length N is $O(\log N)$. If the list does not implement `RandomAccess`, `binarySearch` uses iteration, with linear complexity.

The methods `indexOfSubList` and `lastIndexOfSubList` do not require sorted lists for their operation. Their signatures allow the source and target lists to contain elements of any type (remember that the two wildcards may stand for two different types). The design decision behind these signatures is the same as that behind the `Collection` methods `containsAll`, `retainAll`, and `removeAll` (see “[Bounded or Unbounded?](#)” on page 27).

Collection Factories

<code><T> List<T> emptyList()</code>	return an empty List
<code><K,V> Map<K,V> emptyMap()</code>	return an empty Map
<code><T> Set<T> emptySet()</code>	return an empty Set
<code><T> Iterator<T> emptyIterator()</code>	return an empty Iterator
<code><T> ListIterator<T> emptyListIterator()</code>	return an empty ListIterator
<code><T> NavigableSet<T> emptyNavigableSet()</code>	return an empty NavigableSet
<code><K,V> NavigableMap<K,V> emptyNavigableMap()</code>	return an empty NavigableMap

The `Collections` class provides convenient ways of creating some kinds of collections containing zero or more references to the same object. The simplest possible such collections are empty.

In this section, and in the following section on wrapper factories, we have omitted the methods relating to `SortedSet` and `SortedMap`, as these have been made effectively obsolete by the introduction of `NavigableSet` and `NavigableMap`.

Empty collections can be useful in implementing methods to return collections of values, where they can be used to signify that there were no values to return. Each method returns a reference to an instance of a singleton inner class of `Collections`. Because these instances are immutable, they can safely be shared, so calling one of these factory methods does not result in object creation. The `Collections` fields `EMPTY_SET`, `EMPTY_LIST`, and `EMPTY_MAP` were commonly used for the same purpose in Java before generics, but they’re less useful now because their raw types generate

unchecked warnings whenever they are used. The methods `emptyList`, `emptyMap`, and `emptySet` have themselves become less useful as their functions have been duplicated by the unmodifiable factory method parameterless overloads of `Set::of`, `List::of`, and `Map::of`. The methods `emptyNavigableSet` and `emptyNavigableMap` may become similarly redundant in the future if factory methods for unmodifiable `NavigableSets` and `NavigableMaps` are introduced.

The `Collections` class also provides ways of creating collection objects containing only a single member:

<code><T> Set<T> singleton(T o)</code>	return an immutable set containing only the specified object
<code><T> List<T> singletonList(T o)</code>	return an immutable list containing only the specified object
<code><K,V> Map<K,V> singletonMap(K key, V value)</code>	return an immutable map, mapping only the key K to the value V

Again, these can be useful in providing a single input value to a method that is written to accept a `Collection` of values, and again, they have been duplicated by the unmodifiable factory methods.

Finally, it is possible to create a list containing a number of copies of a given object:

```
<T> List<T> nCopies(int n, T o) return an immutable list containing n references to the object o
```

Because the list produced by `nCopies` is immutable, it need contain only a single physical element to provide a list view of the required length. Such lists are often used as the basis for building further collections—for example, as the argument to a constructor or an `addAll` method.

Wrappers

The `Collections` class provides wrapper objects that modify the behavior of standard collection classes in one of three ways: by synchronizing them, by making them unmodifiable, or by checking the type of elements being added to them. These wrapper objects implement the same interfaces as the wrapped objects, and they delegate their work to them. Their purpose is to restrict the circumstances under which that work will be carried out. These are examples of the use of *protection proxies* ([Gamma et al. 1995](#)), a variant of the Proxy pattern in which the proxy controls access to the real subject.

Proxies can be created in different ways. Here, they are created by factory methods that wrap the supplied collection object in an inner class of `Collections` that implements the collection's interface. Subsequently, method calls to the proxy are

(mostly) delegated to the collection object, but the proxy controls the conditions of the call. In the case of the synchronized wrappers, all method calls are delegated, but the proxy uses synchronization to ensure that the collection is accessed by only one thread at a time. In the case of unmodifiable and checked collections, method calls that break the contract for the proxy fail, throwing the appropriate exception.

Synchronized Collections

As we explained in “[Collections and Thread Safety](#)” on page 155, most of the collections framework classes are not thread-safe—by design—in order to avoid the overhead of unnecessary synchronization (as incurred by the legacy classes `Vector` and `Hashtable`). But for the occasions when you do need to provide a small number of threads with concurrent access to the same collection, these synchronized wrappers are provided by the `Collections` class:

```
<T> Collection<T> synchronizedCollection(Collection<T> c)
<T> Set<T> synchronizedSet(Set<T> s)
<T> List<T> synchronizedList(List<T> list)
<K,V> Map<K,V> synchronizedMap(Map<K,V> m)
<T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> s)
<K,V> NavigableMap<K,V> synchronizedNavigableMap(NavigableMap<K,V> m);
```

The thread safety offered by these wrappers depends on no unsynchronized access being made to the underlying collection. A common usage pattern is to wrap a collection at construction time without retaining any reference to the wrapped collection:

```
var synchList = Collections.synchronizedList(new ArrayList<>( ... ));
```

The classes that provide these synchronized views are conditionally thread-safe (see “[Collections and Thread Safety](#)” on page 155); although each of their operations is guaranteed to be atomic, you may need to synchronize multiple method calls on the wrapper object itself in order to obtain consistent behavior. In particular, iterators must be created and used entirely within a code block synchronized on the collection; otherwise, the result will at best be failure with `ConcurrentModificationException`. This is very coarse-grained synchronization; if your application makes heavy use of synchronized collections, its effective concurrency will be greatly reduced. In this situation, you should turn to the appropriate concurrent collection (see “[Avoid Synchronized Wrapper Collections](#)” on page 302).

Unmodifiable Collections

An unmodifiable collection will throw `UnsupportedOperationException` in response to any attempt to change its structure or the elements that compose it. Used with care, this can be useful when you want to allow clients read access to an internal data structure; passing the structure in an unmodifiable wrapper will prevent a client from changing it but it will not prevent the client from changing the objects it contains, if they are modifiable. Often, you will have to protect your internal data structure by

providing clients instead with a defensive copy made for that purpose, or by placing these objects in unmodifiable wrappers. We explore these options in more detail in “[Respect the ‘Ownership’ of Collections](#)” on page 289.

There are unmodifiable wrapper factory methods corresponding to the major interfaces of the Collections Framework:

```
<T> Collection<T> unmodifiableCollection(Collection<? extends T> c)
<T> Set<T> unmodifiableSet(Set<? extends T> s)
<T> List<T> unmodifiableList(List<? extends T> list)
<K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m)
<T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<? extends T> s)
<K,V> NavigableMap<K,V> unmodifiableNavigableMap(NavigableMap<K,? extends V> m)
<T> SequencedCollection<T>
    unmodifiableSequencedCollection(SequencedCollection<? extends T> c)
<K,V> SequencedMap<K,V>
    unmodifiableSequencedMap(SequencedMap<? extends K,? extends V> m)
<T> SequencedSet<T> unmodifiableSequencedSet(SequencedSet<? extends T> s)
```

Checked Collections

Unchecked warnings from the compiler are a signal to take special care to avoid runtime type violations. For example, after we have passed a typed collection reference to an ungenerified library method, we can’t be sure that it has added only correctly typed elements to the collection. Instead of losing confidence in the collection’s type safety, we can pass in a checked wrapper, which will test every element added to the collection for membership of the type supplied when it was created. “[Enforce Type Safety When Calling Untrusted Code](#)” on page 115 shows an example of this technique.

The following methods each create an object containing an instance of the appropriate interface, together with a type token for the element. All operations are delegated to the instance, but before a new element is added to the collection, it is first tested against the type token, and a `ClassCastException` is thrown if it does not match:

```
<E> Collection<E> checkedCollection(Collection<E> c, Class<E> elementType)
<E> List<E> checkedList(List<E> c, Class<E> elementType)
<E> Set<E> checkedSet(Set<E> c, Class<E> elementType)
<E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> c, Class<E> elementType)
<K,V> Map<K,V> checkedMap(Map<K,V> c, Class<K> keyType, Class<V> valueType)
<K,V> NavigableMap<K,V>
    checkedNavigableMap(NavigableMap<K,V> c, Class<K> keyType, Class<V> valueType)
<E> Queue<E> checkedQueue(Queue<E> c, Class<E> elementType)
```

Other Methods

The `Collections` class provides a number of utility methods, some of which we have already seen in use. Here we review them in alphabetical order.

addAll

```
boolean addAll(Collection<? super T> c, adds all of the specified elements to the specified collection  
T... elements)
```

We have used this method a number of times as a convenient and efficient way of initializing a collection with individual elements or with the contents of an array.

asLifoQueue

```
<T> Queue<T> asLifoQueue(Deque<T> deque) returns a view of a Deque as a last in, first out (LIFO) Queue
```

Recall from [Chapter 13](#) that while queues can impose various different orderings on their elements, there is no standard `Queue` implementation that provides LIFO ordering. `Deque` implementations, on the other hand, all support LIFO ordering if elements are removed from the same end of the `deque` as they were added. The method `asLifoQueue` allows you to use this functionality through the conveniently concise `Queue` interface.

disjoint

```
boolean disjoint(Collection<?> c1, Collection<?> c2) returns true if c1 and c2 have no elements in  
common
```

The implementation of this method in OpenJDK iterates over one of these collections, testing each element for membership in the other and returning `true` if none are found. This is straightforward for non-set collections, which generally use the `equals` method to test for membership. Some sets, however, use other equivalence relations to characterize membership (see [“Defining a Set: Equivalence Relations” on page 182](#)); if two sets using different equivalence relations are being tested for disjointness, the result may depend on which of the two has been chosen by the implementation to test for membership. As with equality of sets, you should avoid using `disjoint` on sets with different equivalence relations. [“Inconsistent with equals” on page 322](#) discusses this issue in detail.

enumeration

```
<T> Enumeration<T> enumeration(Collection<T> c) returns an enumeration over the specified collection
```

This method is provided for interoperation with APIs whose methods take arguments of type `Enumeration`, a legacy version of `Iterator`. The `Enumeration` it returns yields the same elements, in the same order, as the `Iterator` provided by `c`. This method forms a pair with the method `list`, which constructs an `ArrayList` from the elements yielded in order by the `Enumeration`.

frequency

```
int frequency(Collection<?> c, Object o) returns the number of elements in c that are equal to o
```

If the supplied value `o` is `null`, then `frequency` returns the number of `null` elements in the collection `c`.

list

```
<T> ArrayList<T> list(Enumeration<T> e) returns an ArrayList containing the elements returned by the specified Enumeration
```

This method is provided for interoperation with APIs whose methods return results of type `Enumeration`, a legacy version of `Iterator`. The `ArrayList` that it returns contains the same elements, in the same order, as provided by the `Enumeration` `e`. This method forms a pair with the method `enumeration`, which creates an `ArrayList` from an `Enumeration`.

newSequencedSetFromMap and newSetFromMap

```
<E> SequencedSet<E> newSequencedSetFromMap (SequencedMap<E,Boolean> map) returns a SequencedSet backed by the specified SequencedMap
```

```
<E> Set<E> newSetFromMap(Map<E,Boolean> map) returns a Set backed by the specified Map
```

As we saw in [Chapter 15](#), many sets (such as `TreeSet` and `ConcurrentSkipListSet`) are implemented by maps and share their ordering, concurrency, and performance characteristics. Some maps, however (such as `WeakHashMap` and `IdentityHashMap`), do not have standard set equivalents. The purpose of the method `newSetFromMap` is to provide equivalent set implementations for such maps. The method `newSetFromMap` wraps its argument, which must be empty when supplied and should never be

subsequently accessed directly. This code shows the standard idiom for using it to create a weak HashSet, one whose elements are held via weak references:

```
Set<Object> weakHashSet = Collections.newSetFromMap(  
    new WeakHashMap<Object, Boolean>());
```

At first sight, `newSequencedSetFromMap` seems to have limited usefulness within the Collections Framework since, unlike the Map implementations, all SequencedMap implementations in the JDK have standard set equivalents. However, `LinkedHashSet`, which corresponds to `LinkedHashMap`, lacks one important feature that the corresponding map provides: the ability to define an eviction policy (or a callback) to be invoked every time a new entry is made. This ability can be provided to a Sequenced Set by creating it from a `LinkedHashMap`. For example, once this set has grown to contain five elements, adding a new one will cause the oldest one to be discarded:

```
SequencedSet<String> set = Collections.newSequencedSetFromMap(  
    new LinkedHashMap<String, Boolean>() {  
        protected boolean removeEldestEntry(Map.Entry<String, Boolean> e) {  
            return this.size() > 5;  
        }  
    });
```

reverseOrder

<T> Comparator<T> reverseOrder() returns a comparator that reverses natural ordering

The `reverseOrder` method provides a simple way of sorting or maintaining a collection of `Comparable` objects in reverse natural order. Here is an example of its use:

```
SortedSet<Integer> s = new TreeSet<>(Collections.reverseOrder());  
Collections.addAll(s, 1, 2, 3);  
assert s.equals(new TreeSet<>(Set.of(1, 2, 3)).reversed());
```

This method predates Java 8, which introduced the feature of static interface methods. The `Comparator` method `reverseOrder`, which has exactly the same functionality as `Collections::reverseOrder`, was provided in Java 8 as an alternative that developers would be able to find more easily when looking for ways of manipulating Comparators. It also corrects a problem with the `Collections` method, whose parametric type is not precisely defined so that it can create comparators that will produce run-time errors. For example, a comparator created like this:

```
Comparator<Object> objReverseCompr = Collections.reverseOrder();
```

will work correctly applied to a collection of comparable elements:

```
var strings = new ArrayList<>(List.of("a", "b", "c"));  
strings.sort(objReverseCompr);  
assert strings.equals(List.of("c", "b", "a"));
```

but will fail with a run-time error when applied to a collection of non-comparable objects:

```
var classes = new ArrayList<>(List.of(String.class,
    Integer.class));           // throws ClassCastException
classes.sort(objReverseCompr);
```

This problem can't arise with the `Comparator` method, which requires a `Comparable` as its type parameter:

```
<T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

The `Collections` class has a second overload of this method:

```
<T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

This method is like the preceding one, but instead of reversing the natural order of an object collection, it reverses the order of the `Comparator` supplied as its argument. Its behavior when supplied with `null` is unusual for a method of the `Collections` class. The contract for `Collections` states that its methods throw a `NullPointerException` if the collections or class objects provided to them are `null`, but if this method is supplied with `null` it returns the same result as a call of `reverseOrder`—that is, it returns a `Comparator` that reverses the natural order of a collection of objects.

This method too has an equivalent in the `Comparator` interface: in this case, it is the default (i.e., instance) method `Comparator::reversed`, in which the receiver takes the place of the parameter to the static `Collections` method.

Conclusion

If you have read this far, you should now have a fairly comprehensive picture of the features that the Java Collections Framework has to offer. But there is a difference between knowing what is available and knowing how to use it. The long life of the Collections Framework—more than a quarter-century at the time of this writing—has created the opportunity to record some of the experience of the millions of Java programmers who have used it during that time. The next chapter offers some guidance distilled from that experience.

Guidance for Using the Java Collections Framework

The Java Collections Framework has been heavily used by millions of working Java programmers since it first appeared in 1998, a quarter-century ago at the time of this writing. The material in this chapter identifies some lessons we can derive from this collective experience. Even items that may seem elementary are worth stating explicitly; all the guidance in this chapter is based on experience with real-world code, where failing to follow it has resulted in systems that are fault-prone or difficult to maintain.

In this chapter, the terms *library*, *API*, and *client* are used with the following meanings: a *library* is any class that exposes a public method, its *API* is the collection of public methods that it exposes, and a *client* is any class that calls methods in the API.



The code examples for this chapter can be found at:

[https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/
main/src/main/java/org/jgcbook/chapter17](https://github.com/MauriceNaftalin/JGC_2e_Book_Code/blob/main/src/main/java/org/jgcbook/chapter17)

Avoid Anemic Domain Models

The term *anemic domain model* was coined by Martin Fowler (2003) to describe designs in which objects of the domain model are essentially data carriers, containing little or no business logic. In the standard model of object-oriented design, this leads to business logic becoming concentrated in the service layer. Fowler notes that “the problem with anemic domain models is that they incur all the costs of a domain model [in particular the need for persistence mappings], without yielding any of the benefits.” The scenario he describes allows for these objects to be named for nouns

in the domain space, whereas in the Collections Framework this anti-pattern appears as nameless collection structures directly containing other collections, sometimes deeply nested, with no provision for custom behavior. The object design failure is fundamentally the same in each case.

A simple example should help to clarify this. Suppose that you decide to introduce a new entity into the task management system from earlier chapters: projects, which group tasks together under a single named heading. The service layer to manage projects for a user could simply model them as a map from project name to a collection of tasks. Including a single sample method—to add a task—the service layer could look, in outline, like this:

```
public class ServiceLayer {  
  
    Map<String, Set<Task>> projects = new HashMap<>();  
  
    public void addTask(String projectName, Task task) {  
        projects.merge(projectName, new HashSet<>(Set.of(task)),  
                      (oldValue,newValue) -> {oldValue.addAll(newValue); return oldValue;});  
    }  
    ...  
}
```

Here, the central problem of the anemic domain model is apparent: logic that should belong to the domain object is not expressed there, and indeed, in this representation, cannot be expressed there. So instead, it must be captured by the service layer. This problem grows as the domain logic becomes more complex. For example, suppose that the design of tasks changes to include a duration:

```
org/jgcbook/chapter17/A_avoid_anemic_domain_models/Task  
public interface Task extends Comparable<Task> {  
    @Override  
    default int compareTo(Task t) {  
        return toString().compareTo(t.toString());  
    }  
    Duration duration();  
}
```

```
org/jgcbook/chapter17/A_avoid_anemic_domain_models/CodingTask  
public record CodingTask(String spec, Duration duration) implements Task {}
```

```
org/jgcbook/chapter17/A_avoid_anemic_domain_models/PhoneTask  
public record PhoneTask(String name, String number, Duration duration)  
    implements Task {}
```

and that projects are frequently queried for the total duration of the tasks they contain, so it makes sense for this total to be cached. Now the service layer has to contain a separate data structure to record the cached value for each project:

```

public class ServiceLayer {

    Map<String,Set<Task>> tasksPerProject = new HashMap<>();
    Map<String,Duration> durationsPerProject = new HashMap<>();

    public void addTask(String projectName, Task task) {
        tasksPerProject.merge(projectName, new HashSet<>(Set.of(task)),
            (oldValue,newValue) -> {oldValue.addAll(newValue); return oldValue;});
        durationsPerProject.merge(projectName, task.duration(), Duration::plus);
    }
    ...
}

```

The implicit invariant for the application includes the requirement that every value in the `durations` map should be the sum of the corresponding task durations in the `projects` map—an invariant that is quite difficult to express and that, without encapsulation, will be even more difficult to maintain consistently. This invariant governs the components of a single project, so encapsulating the logic for a project within a single object can focus the requirement in a more comprehensible and maintainable way:

```

public class Project {

    private final String name;
    private final Set<Task> tasks;
    private Duration totalDuration;

    public void addTask(Task task) {
        tasks.add(task);
        totalDuration = totalDuration.plus(task.duration());
    }
    ...
}

```

Although locating the responsibilities of `Project` objects in the `Project` class conforms to standard object-oriented design techniques, it is useful to be able to recognize nested collection types as a code smell, alerting you to the need to refactor your design in this way.

Respect the ‘Ownership’ of Collections

The most significant decisions in the design of the Collections Framework, once it had been resolved that collections were to be mutable, were around the control of mutation. Of these decisions, the most important one was that the interfaces should expose both read and write operations, rather than separating them in different interfaces. In this section, we explore the consequences for developers wishing to ensure encapsulation of collection data in their objects.

These consequences include the presence of optional operations, in some cases unsupported by unmodifiable or partially modifiable collections and collection views. The presence of optional operations has led to criticism of the decision to combine read and write operations in the same interfaces: we shall see, however, when we investigate this question in [“Fundamental Issues in the Collections Framework Design” on page 311](#) that no approach to the problem of guaranteeing encapsulation is without trade-offs.

So given that collections are mutable and the interfaces expose mutation methods, how can an object safely encapsulate its data? We can explore answers to this question through the `Project` class introduced in the previous section. In designing the API for this class, we would want to consider the operations a client might need to perform. Obviously, it might be necessary to add a task to or remove one from a `Project`:

```
public class Project {  
    ...  
    public void addTask(Task task) {  
        tasks.add(task);  
        totalDuration = totalDuration.plus(task.duration());  
    }  
    public void removeTask(Task task) {  
        tasks.remove(task);  
        totalDuration = totalDuration.minus(task.duration());  
    }  
}
```

But suppose that we anticipate that a client might also need to iterate over the contents of `tasks`, or to stream them. Anticipating all possible uses of the data is difficult, so we might be tempted to replace all these methods by one method that simply exposes the field itself:

```
public class Project {  
    ...  
    public Set<Task> getTasks() {  
        return tasks;  
    }  
    ...  
}
```

The problem with this (very common) approach is that there is no way to ensure that a client will respect the need to maintain the class invariant. Perhaps it will only perform read operations, in which case there is no problem. But writing to the collection can create subtle bugs that are hard to detect, with symptoms that can appear with no obvious connection to their actual cause. In this example, if a client adds or removes a task without updating `totalDuration`, that field will no longer provide an accurate reflection of the total duration of the tasks in the project.

As the API designer, do you trust the client? You may, if it's part of the same closely coupled system that will be maintained by the same person or team as your library

code. But if the client is unaware of the internal logic of your code, or if you're concerned about accidental mutation, then you need to make an extra effort to maintain encapsulation. Encapsulation is especially important in a large system: it allows you to reason about the code using only local knowledge, instead of requiring knowledge about the entire system. The best way is to protect Project's state by only exposing it in an unmodifiable wrapper. That then makes it necessary to restore the individual mutation methods, as in [Example 17-1](#). (We have added `equals` and `hashCode` implementations to this example, to be used in a later discussion.)

Example 17-1. Accessor returning an unmodifiable collection

org/jgcbook/chapter17/B_ownership/Project

```
public class Project {  
  
    private final String name;  
    private final Set<Task> tasks;  
    private Duration totalDuration;  
  
    public Project(String name, Set<Task> tasks) {  
        this.name = name;  
        this.tasks = tasks;  
        totalDuration = tasks.stream()  
            .map(Task::duration)  
            .reduce(Duration.ZERO, Duration::plus);  
    }  
  
    public String getName() { return name; }  
  
    public Duration getTotalDuration() { return totalDuration; }  
  
    public Set<Task> getTasks() {  
        return Collections.unmodifiableSet(tasks);  
    }  
    public void addTask(Task task) {  
        tasks.add(task);  
        totalDuration = totalDuration.plus(task.duration());  
    }  
    public void removeTask(Task task) {  
        tasks.remove(task);  
        totalDuration = totalDuration.minus(task.duration());  
    }  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Project project)) return false;  
        if (!name.equals(project.name)) return false;  
        return tasks.equals(project.tasks);  
    }  
    @Override  
    public int hashCode() {  
        int result = name.hashCode();  
        result = 31 * result + tasks.hashCode();  
    }  
}
```

```
        return result;
    }
}
```

Wrapping the exposed state in this way ensures that `Project` objects cannot be corrupted by clients receiving references to it via accessor methods.

The principle that clients should not be able to mutate the internal state of an object is quite general, only to be relaxed in the case of closely cooperating system components. When, as here, the state being exposed is a member of the Collections Framework, the unmodifiable wrappers of the `Collections` class are exactly what is needed to implement the principle. This technique may not be available for other kinds of mutable data; in these cases, *defensive copying* can instead be used to protect exposed object state. With defensive copying, accessors create a copy of the object's state and return a reference to that. This copy is distinct from the state itself and so can be modified by a client without danger of corrupting the object.

One situation in which this is necessary is when an array is a component of object state. Arrays are mutable, but the inconvenience of having to copy them at every accessor call is lessened by the fact that they expose a public `clone` method. If the task collection of projects has to be represented by an array, for reasons of compatibility or performance, the code should look something like this:

```
public class ProjectWithArray {

    private final String name;
    private final Task[] tasks;
    private Duration totalDuration;

    public ProjectWithArray(String name, Task[] tasks) {
        this.name = name;
        this.tasks = tasks;
        totalDuration = Arrays.stream(tasks)
            .map(Task::duration)
            .reduce(Duration.ZERO, Duration::plus);
    }

    public Task[] getTasks() {
        return tasks.clone();
    }
    ...
}
```

In general, providing a public `clone` method requires a class to implement the `Cloneable` interface, and is not straightforward (see Bloch 2017, item 13). Instead, you can provide a copy constructor or, in the modern idiom, a static copy factory.

Defensive copying is also used—more commonly, in fact—to protect an object's state from mutation of input data. For example, the encapsulation of `Project` state is not yet complete, as shown by this code:

```

var tasks = new HashSet<Task>();
tasks.add(new CodingTask("code ui", Duration.ofHours(4)));
tasks.add(new CodingTask("code db", Duration.ofHours(6)));
var project1 = new Project("Project1", tasks);
var project2 = new Project("Project2", tasks);

```

Supplying two different projects with the same collection of tasks has coupled them so that adding or removing a task in `project1` will unexpectedly affect `project2`, and vice versa. Here is another way in which objects can fail to control their own state, in this case because it is not unambiguously their own state: rather, it is shared with another object. This is the problem known in the computing literature as *aliasing*. You can avoid it by making a defensive copy, decoupling each project's state from the collection supplied to its constructor. This also allows the constructor to choose its own representation of the data. For example, we might choose to maintain the tasks in sorted order within the project, without necessarily affecting clients in any way:

```

private final NavigableSet<Task> tasks;
...
public Project(String name, Set<Task> tasks) {
    this.name = name;
    this.tasks = new TreeSet<>(tasks);
    totalDuration = tasks.stream()
        .map(Task::Duration)
        .reduce(Duration.ZERO, Duration::plus);
}

```

In summary, we have considered two techniques for encapsulating object data:

- To protect an object's state from mutation of input data, there is no alternative to defensive copying for ensuring exclusive access to an object reference.
- To protect state that would otherwise be exposed on output, there are two possibilities:
 - Where possible, as with collections, return them wrapped in an unmodifiable wrapper. This is the better alternative when it is available, as defensive copying is a potentially expensive operation to repeat for every accessor call.
 - When no unmodifiable wrapper is available, defensive copying on output may be necessary.

Prefer Immutable Objects as Set Elements or Map Keys

Many errors in code using the Collections Framework can be traced to the same cause: a failure to prevent mutation of elements of hashed and ordered collections. Data structures that store their elements by value rely on using the same value to retrieve them. Although it might seem obvious, this rule is easily broken, and the results can be surprising. For example, suppose that we wished to store a collection of

Project objects, as defined in [Example 17-1](#), in a HashSet. This reasonable-seeming goal can produce strange results:

```
org/jgcbook/chapter17/C_use_immutable_objects_as_set_elements_map_keys/Program_1
var tasks = new HashSet<Task>(Set.of(new CodingTask("code ui", Duration.ofHours(4))));
var myProject = new Project("My Project", tasks);
var projectSet = new HashSet<>(Set.of(myProject));
assert projectSet.contains(myProject);
myProject.addTask(new CodingTask("code db", Duration.ofHours(6)));
assert ! projectSet.contains(myProject);
```

The final assertion may seem even more surprising in light of the fact that the first line of `Project::equals` is an identity check (added to that method for efficiency). The set `myProjects` is failing to find the very object that was present in it only two lines earlier! This failure is easily understood by reference to [Figure 12-1](#). The choice of the bucket that determines the starting point for the search for a matching element is made by the hash code of the object: the stored object can never be found if the wrong overflow chain is being searched. In this case, `Project::hashCode` includes the hash code for the `tasks` component in its calculation, so when that component changes, the hash code changes, and a different bucket may be chosen for the search starting point.

A natural train of thought leads to the idea of excluding the mutable `tasks` component from the hash code calculation. That may solve the problem in some cases, but only at the cost of reducing the performance of the hashing algorithm; here, we might be reasonably confident that different projects will have different names, but excluding significant parts of an object may lead in real-life examples to unacceptable concentrations of entries in a small number of buckets.

Something similar can happen with `TreeSet`:

```
org/jgcbook/chapter17/C_use_immutable_objects_as_set_elements_map_keys/Program_2
Task codeUi = new CodingTask("code ui", Duration.ofHours(6));
var project1 = new Project("project1", new HashSet<>(Set.of(codeUi)));

Task codeDb = new CodingTask("code db", Duration.ofHours(4));
var project2 = new Project("project2", new HashSet<>(Set.of(codeDb)));

var projectSet = new TreeSet<>(Comparator.comparing(Project::getTotalDuration)
    .thenComparing(Project::getName));

projectSet.addAll(List.of(project1, project2));❶
assert projectSet.contains(project2);
project2.addTask(new CodingTask("code ai", Duration.ofHours(5)));❷
assert ! projectSet.contains(project2);❸
```

[Figure 17-1](#) shows the state of the `TreeSet` before and after the call to `addTask` at ❶ and ❷. ❸

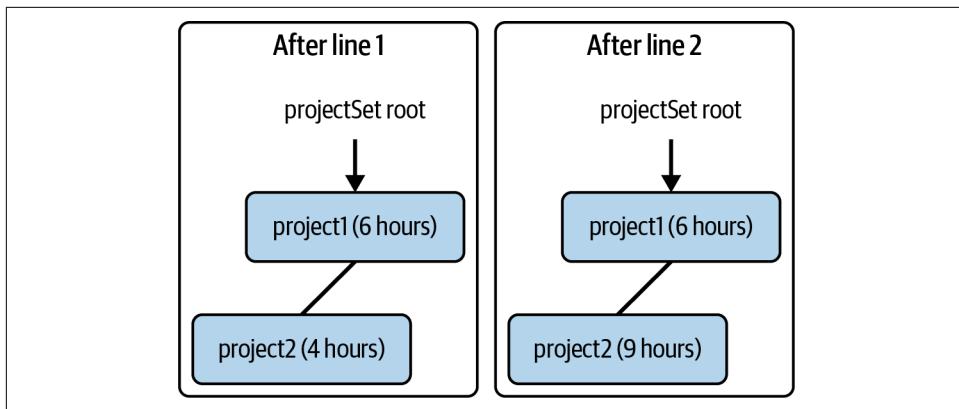


Figure 17-1. The TreeSet before and after its invariant is broken

After the call to `addAll` at ❶, `project2` is placed in the lefthand subtree of the `TreeSet`, because the duration of the tasks in it is less than that of the root element, `project1`. But with the changed state of `project2` after a new task is added to it at ❷, it belongs properly in the righthand subtree. That is where the `contains` method looks for it at ❸, and of course fails to find it. The search algorithm for the `TreeSet` depends on its invariant, which asserts, in part, that elements comparing as greater than a node should always be in its righthand subtree. That invariant has been broken by the call to `addTask` at ❷, so the search algorithm is no longer guaranteed to work.

Any collection that organizes its elements according to their values will show the same effects. This also applies to maps like `HashMap` and `TreeMap` that organize their keys according to their values. Of the common collection classes, only `List` implementations are immune to these problems, since their elements are organized by the actions of client code. The other collection classes are simply not suitable for storage of elements that will be mutated; in fact, to be certain of avoiding these problems, elements to be stored in them should, where possible, be immutable.

Balancing Client and Library Interests in API Design

In one sense, this section is about API design in general, rather than about collections. The common slogan for object-oriented design is “program to the interface”; the question addressed here is “which interface?” This question arises anytime you are designing a method that receives a parameter or returns a value, when the type of either lies somewhere within a type hierarchy. This happens more often with APIs that transmit collections than elsewhere, because the collections type hierarchy is unusually deep. For example, possible return types for the method `Project::getTasks` in the previous section include `Iterable`, `Collection`, `SequencedCollection`, `Set`, `SequencedSet`, and `NavigableSet`, and even the concrete type `TreeSet`.

Deciding where in the type hierarchy to place the parameter or return type is an exercise in balancing the interests of the destination—which lie in the direction of the most specific type—with those of the source, which are about retaining maximum flexibility by supplying the most abstract type. For example, the developer of the `Project` class might prefer to return a `Collection<Task>`, as that would provide the most flexibility to the implementation. It would leave the library maintainers free to change the implementation to any other kind of collection—a list, say—if that provided an advantage later on. The problem for the client is that this choice deprives it of possibly useful information about the returned collection: that it is ordered and that it has no duplicates.

Going to the opposite extreme, we could type the value returned from `getTasks` with the concrete implementation type, `TreeSet`. Now the problem is that the `Project` class is severely constrained. For example, a later stage of the development might require the task collection to be thread-safe, forcing a change of implementation to the thread-safe collection `ConcurrentSkipListSet`. However, it would be infeasible to make that change to `Project` if the return type exposed had previously been `TreeSet`, allowing existing clients to depend on that specific type.

So the key question is how to balance the flexibility requirements of the library implementation(s) with the semantics the destination actually needs. In this case, will it benefit a client to know that the returned collection is sorted, in which case `getTasks` should return a `NavigableSet`? Or is it only necessary for it to be able to access the first and last elements, in which case we could return a `SequencedSet` and leave open the possibility of changing the implementation to a `LinkedHashSet`? (Refer to [Figure 12-2](#) for a refresher on the relationship between these types.) If clients will not require the information that the task elements are unique, we could return a `SequencedCollection`.

Similar considerations apply to the choice of parameter types. To maximize the usability of the API for a wide diversity of clients, it should accept a type high in the hierarchy. This also makes sense in terms of the balance of development effort: the work of handling a supertype need only be done once in the library code, whereas adapting client code to library requirements must be done at every method call. In the case of the `Project` constructor, this reasoning might lead to accepting a `Collection` of tasks.

In general, you can achieve the best balance with a type chosen from among the middle layers of the hierarchy, but the deciding question in each specific case should be the semantics of the API.

Exploit the Features of Records

It wasn't easy to define tuples in Java before records were given final and permanent status in Java 16. In general, the primary use of tuples is as data carriers. If you wanted a data carrier before Java 16, you had to define a class for the purpose, requiring explicit declarations of a constructor to initialize the fields, together with methods for equality, hash code calculation, field access, and string representation. Further, it is desirable for data carriers to be immutable (see “[Immutability and Unmodifiability](#)” on page 147). Satisfying these requirements makes for a very heavy-weight representation of a simple tuple, so Java developers used various workarounds in order to avoid tuples altogether. With the advent of records, which satisfy those requirements automatically—as well as being able to define behavior when that is required—these workarounds are no longer needed.

Prefer Records to Parallel Lists

One useful role for records is in replacing parallel lists. Here, the term *parallel* refers not to the action of multiple threads, but to data structures that implicitly combine several lists with parallel structure, in which elements at the same index position in each list represent components of the same object.¹ For example, consider the problem of modeling a club competition ladder in which a player can move upward by successfully challenging the player directly above them. One plausible representation is by two parallel lists: a list of players and, in parallel, a list of their rankings on the ladder. In [Example 17-2](#), the `exchange` method is invoked when one player successfully challenges another, one position higher in the rankings. The `players` list remains the same, but their rankings are exchanged.

Example 17-2. Representing a competition ladder by parallel lists

org/jgcbook/chapter17/E_records/Ladder

```
record Person(String name, int age) {}
public class Ladder {
    final private List<Person> players;
    final private List<Integer> rankings;
    public Ladder(List<Person> players, List<Integer> rankings) {
        this.players = players;
        this.rankings = rankings;
    }
    public List<Integer> getRankings() {
        return rankings;
    }
}
```

¹ In other languages, parallel arrays may have performance advantages, but these are small compared to their other problems and are in any case not available to Java lists, of which the component elements are always objects rather than primitive values.

```

public void exchange(Person winner, Person loser) {
    int winnerLocation = players.indexOf(winner);
    int loserLocation = players.indexOf(loser);
    int winnerRanking = rankings.get(loserLocation);
    rankings.set(winnerLocation, winnerRanking);
    rankings.set(loserLocation, winnerRanking + 1);
}
public static void main(String[] args) {
    final Person george = new Person("george", 33);
    final Person john = new Person("john", 31);
    final Person paul = new Person("paul", 30);
    Ladder ladder = new Ladder(new ArrayList<>(List.of(george, john, paul)),
        new ArrayList<>(List.of(2, 3, 1)));
    ladder.exchange(john, george);
    assert ladder.getRankings().equals(List.of(3, 2, 1));
}
}

```

The drawbacks of this representation become clear when we consider other operations on the ladder. For example, sorting the data differently—say, by the age of the players—requires you to first sort the `players` list, recording the changes that you have made in a permutation vector (that is, a new list whose value at each index corresponds to the new position of the element formerly at that index). The permutation can then be used to make the same changes to the `rankings` list. A more attractive alternative is to transfer the data to a sorted map, suggesting that a sorted map might be a better representation; however, this carries the disadvantage that any given sorted map you might choose would restrict you to a single sort key.

A record declared for the purpose provides greater flexibility:

org/jgcbook/chapter17/E_records/RecordLadder

```

record PlayerRanking(Person player, int ranking) {}
public class RecordLadder {
    final private List<PlayerRanking> ladder = new ArrayList<>();
    public RecordLadder(List<Person> players, List<Integer> rankings) {
        for (int i = 0; i < players.size(); i++) {
            ladder.add(new PlayerRanking(players.get(i), rankings.get(i)));
        }
    }
    public List<PlayerRanking> getLadder() {
        return ladder;
    }
    ...
}

```

Now the `exchange` method is a little clumsier, because a new helper method must be defined to replace `indexOf` as the way to locate an element on the ladder:

```
private int locate(Person p) {
    for (int i = 0; i < ladder.size(); i++) {
        if (ladder.get(i).player().equals(p)) return i;
    }
    return -1;
}
public void exchange(Person winner, Person loser) {
    int winnerLocation = locate(winner);
    int loserLocation = locate(loser);
    int winnerRanking = ladder.get(loserLocation).ranking();
    ladder.set(winnerLocation, new PlayerRanking(winner, winnerRanking));
    ladder.set(loserLocation, new PlayerRanking(loser, winnerRanking + 1));
}
```

But sorting is now straightforward:

```
public static void main(String[] args) {
    final Person peter = new Person("peter", 33);
    final Person paul = new Person("paul", 31);
    final Person mary = new Person("mary", 30);
    RecordLadder ladder = new RecordLadder(new ArrayList<>(List.of(peter, paul, mary)),
        new ArrayList<>(List.of(2, 3, 1)));
    ladder.exchange(paul, peter);
    assert ladder.getLadder().get(ladder.locate(paul)).ranking() == 2 &&
        ladder.getLadder().get(ladder.locate(peter)).ranking() == 3;
    ladder.getLadder().sort(Comparator.comparing(pp -> pp.player().age()));
    assert ladder.getLadder().stream().map(PlayerRanking::player).toList().equals(
        List.of(mary, paul, peter));
}
```

It is not surprising that sorting, along with other operations, is simpler in the version using records, because records explicitly represent the relationship between the data elements, requiring no extra work to maintain it. This is in contrast to the implicit relation in parallel lists, which has to be manually maintained.

Use Records as Composite Keys

Records provide a good solution to the common problem of representing collections of objects that are uniquely identified by a subset of their properties. For example, consider how to represent a library consisting of instances of a Book class:

```
class Book {
    private String title;
    private String author;
    private String publisher;
    private int pageCount;
    ...
}
```

Instances of `Book` are uniquely identified by a combination of `title` and `author`. Suppose that the primary use case is to quickly retrieve the publisher of a `Book`, as identified by its author and title. What fields should equality be defined on, and what collection of `Book` should be chosen to serve that purpose best?

For a large collection, a hashed data structure like a set or a map will be preferable to one with linear search time, like a list. In the interest of speed, you will want to reduce the number of fields used to define equality and the hash code, and since `author` and `title` together define a `Book` uniquely, it seems reasonable to use only those fields. This is especially compelling in the common case where there are many other fields besides the ones that uniquely characterize instances.

So one common solution to this problem is to provide `Book` with these methods:

```
class Book {  
    ...  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Book book)) return false;  
        if (!title.equals(book.title)) return false;  
        return author.equals(book.author);  
    }  
    public int hashCode() {  
        return Objects.hash(title, author);  
    }  
}
```

But now consider how, given an author and title, you would query a set of `Book` objects to discover whether it contained that book. The only way to do that is to create a dummy `Book` object containing only the author and title, with the values of other fields being irrelevant. Now there are two kinds of books in your system: real books and dummy books. It is easy to see how this can lead to errors. Worse still, this design is not resilient to modification. For example, suppose `Book` is enhanced with a new field to represent the edition:

```
class Book {  
    private String title;  
    private String author;  
    private int edition;  
    private String publisher;  
    private int pageCount;  
    ...  
}
```

Now it is the combination of `title`, `author`, and `edition` that uniquely identifies a `Book` object. And in addition to the need to find the object with those fields, you may now also have a requirement to quickly find all the editions of a given book.

A solution to this problem is to define a record containing the composite key:

```
record TitleAuthor(String title, String author) {}
```

In the first part of the example, when the title and author uniquely identify a Book object, a collection of these can be modeled with a map:

```
Map<TitleAuthor, Book>
```

allowing fast retrieval given only the key fields. Now, when the edition field is added, the representation can be modified (there will likely only be a relatively small number of editions):

```
Map<TitleAuthor, List<Book>>
```

Of course, the same effect can be achieved without records, by defining a class TitleAuthor, but records have the great advantage of concisely providing exactly the characteristics we want: unmodifiability and appropriate definitions of equals and hashCode.

Manage the Mutability of Records

When we discussed good practice for mutation control in “[Respect the ‘Ownership’ of Collections](#)” on page 289, we were dealing with the general case. Records have some special features that make it easier to apply the principles that we outlined there. Records are not immutable, as is sometimes thought, and collections held in their fields are not even automatically unmodifiable, but it is relatively easy to make them so. The first step outlined in that section is to make a defensive copy of data supplied to the object. For example, suppose we wanted to store snapshots of projects, recording the set of tasks associated with each one on a given date. As a data carrier, a record is ideal for this purpose. We are immediately freed from having to consider mutation of the fields, since there are no setter methods. In this case, there is no need for defensive copying for the name or the date, since both of these are represented by immutable types. The only remaining requirement is to make a defensive copy on input of the set of tasks. The compact constructor allows us to use very concise code to achieve this:

```
record ProjectWithSet(String projectName, LocalDate date, Set<Task> tasks) {  
    ProjectWithNavigableSet {  
        tasks = Set.copyOf(tasks);  
    }  
}
```

In this case, the method `Set::copyOf` has a dual role. The defensive copy that it makes is unmodifiable, so no further wrapping is required. And in the absence of setters or other code to alter the contents of `tasks`, the same unaltered copy can be repeatedly returned from calls to the getter.

The `copyOf` method is a convenience not available with every collection. If we wanted the project snapshots to maintain the component tasks in sorted order, we could define the record to accept a `NavigableSet` of tasks. But in this case, the copy constructor that we would use to make the defensive copy does not supply

unmodifiability, so a further step is required to wrap the modifiable copy that it makes in a unmodifiable wrapper.

And when we return a reference via a getter, it must be to an unmodifiable copy:

```
record ProjectWithNavigableSet(String name, NavigableSet<Task> tasks) {  
    ProjectWithNavigableSet {  
        var defensiveCopy = new TreeSet<>(tasks);  
        tasks = Collections.unmodifiableNavigableSet(defensiveCopy);  
    }  
}
```

If the mutable objects to be stored are not collections, defensive copying on input may require more work in the constructor. And if no unmodifiable wrapper is available, it may be necessary to override the accessor to make a new defensive copy on each call.

Avoid Legacy Implementations

When an API as large and complex as the Collections Framework develops over a period of many years, it is inevitable that new ideas will appear and evolve, sometimes replacing older ones. Indeed, it was the appearance in Java 2 of the Collections Framework that made the collections from earlier Java versions—`Vector`, `Stack`, and `Hashtable`—obsolete. We'll omit discussion of the design defects of these classes, as you're unlikely ever to encounter them unless you have the misfortune to need to interoperate with Java code written in the dawn of history, before 1998.

Instead, this section discusses some implementations from the Collections Framework itself that are also now best avoided.

Avoid Synchronized Wrapper Collections

We saw in “[Synchronized Collections and Fail-Fast Iterators](#)” on page 158 how relinquishing thread safety in the newly introduced Java 2 collections—`ArrayList`, `HashSet`, and so on—created the necessity for a layer of thread safety that could be added on to these collections when the situation demanded it.

This layer is provided by factory methods in the `Collections` class, which wrap the supplied collection in an object that acts as a global lock, permitting access to only one thread at a time. Until Java 5, these objects provided the only approximation to thread safety in the Collections Framework. For example, to make a synchronized `List`, you could supply an instance of `ArrayList` to be wrapped. The wrapper object returned from `Collections::synchronizedList` implements the interface by delegating method calls to the collection you supplied, but the calls are synchronized on the wrapper object itself.

To illustrate how this works, the class `SynchronizedArrayStack` shows a very simple synchronized wrapper for the interface `Stack` of [Example 9-1](#) (repeated here for convenience):

[org/jgcbook/chapter17/F_legacy/SynchronizedArrayStack](#)

```
interface Stack {  
    public void push(int elt);  
    public int pop();  
    public boolean isEmpty();  
}  
  
class SynchronizedArrayStack implements Stack {  
    private final Stack stack;  
    public SynchronizedArrayStack(Stack stack) {  
        this.stack = stack;  
    }  
    public synchronized void push(int elt) { stack.push(elt); }  
    public synchronized int pop() { return stack.pop(); }  
    public synchronized boolean isEmpty() { return stack.isEmpty(); }  
}
```

To get a thread-safe `Stack`, you would write:

```
Stack threadSafeStack = new SynchronizedArrayStack(new ArrayStack());
```

This is the preferred idiom for using synchronized wrappers; the only reference to the wrapped object is held by the wrapper, so all calls on the wrapped object will be synchronized on the same lock—that belonging to the wrapper object itself.

Using synchronized collections safely

Even a class like `SynchronizedArrayStack`, which has fully synchronized methods and is itself thread-safe, must still be used with care in a concurrent environment. For example, this client code is not thread-safe:

```
Stack stack = new SynchronizedArrayStack(new ArrayStack());  
...  
// don't do this in a multithreaded environment  
if (!stack.isEmpty()) {  
    stack.pop();           // can throw IllegalStateException  
}
```

The exception would be thrown if the last element on the stack were removed by another thread in the time between the evaluation of `isEmpty` and the execution of `pop`. This is an example of a common concurrent program bug, variously called *test-then-act*, *check-then-act*, or *TOCTOU* (time-of-check to time-of-use), in which program behavior is guided by information that will sometimes be out of date. To avoid it, the test and action must be executed atomically. For synchronized wrapper collections, this must be enforced with *client-side locking*. For example, if we decided to make `Stack` extend `Iterable` so as to allow iteration over `SynchronizedArray`

Stack, the Javadoc recommendation (for a different but equivalent form of iteration) would be for client code to take this form, warning that failure to do so may result in nondeterministic behavior:

```
synchronized(stack) {  
    for (int element : stack) {  
        // process element  
    }  
}
```

From this, it should be clear that the thread safety provided by client-side locking comes at a high cost. Since other threads cannot use any of the collection's methods while the action is being performed, guarding a long-lasting action (say, iterating over an entire array) will have an impact on throughput, potentially losing all the benefits of concurrency.

A further disadvantage of client-side locking is that all clients must use the same lock. The convention is to use the same lock that is used by the methods of the synchronized collection—that is, the wrapper object itself. The problem is enforcing that discipline on every client: this kind of convention is notoriously fragile under maintenance. And in general, when it is passed a reference to the interface type (Stack, in this case), a client has no way of telling whether the object it refers to requires locking or not on iteration and compound operations.

There are circumstances in which synchronized collections are still useful; for example, if you need a list that is relatively frequently modified (so that `CopyOnWriteArrayList` is not suitable) but not heavily contended, or if your application needs a feature of the synchronized collections such as exclusive locking. In these circumstances, you should usually encapsulate client-side locking in a higher-level application object that coordinates locking of the synchronized wrapper (although in this case the synchronized wrapper may provide advantages of the nonsynchronized collection). However, outside of these use cases, the concurrent collections are almost always a better option.

Avoid `LinkedList`

[Table 14-2](#) provides an excellent example of the reasons why the time complexity of individual data structure operations cannot be sensibly considered in isolation. On the face of the values in that table, it would appear that `LinkedList` should be the list implementation of choice when the primary use case will be addition or removal of elements at or near the beginning of the list—for example, in implementing a stack. In a `LinkedList` such operations, taken in isolation, have constant time complexity, whereas in an `ArrayList` they take linear time, owing to the need to maintain contiguity and zero-indexing by moving all the higher-indexed elements of the collection. This fact, however, must be taken in context with other characteristics of `LinkedList`, which taken together weigh the balance decisively against its use.

First, a `LinkedList` is very memory intensive. Because it is doubly linked, every node has to contain two references, one each to the previous and the next node in the list. As a result, each node requires at least 24 bytes (the exact amount depends on how the JVM is representing object pointers and on memory alignment considerations), compared to a typical 4-byte overhead for `ArrayList`. Even allowing for the fact that the backing array for `ArrayList` uses more storage than the collection actually requires, this still gives `ArrayList` a typically 3x advantage in raw memory utilization. This may well be significant for very large collections and, besides requiring larger heaps and potentially more physical memory, can lead to increased garbage collection overheads and pause times.

Even for smaller collections, however, this greater memory usage is significant, especially if progressive allocation of nodes results in them being widely spaced in memory. As described in “[Memory](#) on page 143”, a key element of modern hardware architectures is a memory hierarchy, in which CPU caches play a crucial role. Refreshing caches following a cache miss can cost hundreds of CPU cycles, so minimizing cache misses is a vital part of performance optimization. The units of cache storage, called cache lines, have limited capacity—typically 64 bytes—and their contents always reflect a contiguous block of memory. So in a favorable case, a single cache line could hold 16 `ArrayList` references, compared to—at best—two `LinkedList` nodes. Since in many practical situations the cost of cache misses dominates program performance, this is a huge disadvantage for `LinkedList`.

The most compelling argument in favor of `LinkedList` is its ability to add and remove a single element in constant time. However, any realistic scenario involving these operations always involves accessing the list position to be managed either by index or by a linear search of the collection. Even in the case of a linear search, `ArrayList` will give better performance (for the reasons discussed previously), and if it can be sorted, binary search on an `ArrayList` gives $O(\log N)$ performance. In practice, the cost of iterating over a large collection far outweighs the cost of a single modification, even if for `ArrayList` that involves a call of `System.arraycopy` to keep the elements contiguous.

One scenario that is often proposed as a compelling reason to keep `LinkedList` in a developer’s toolbox is its use as a stack or a queue. But the Collections Framework has better alternatives to offer for that purpose, in `ArrayDeque` and, since Java 21, a reversed `ArrayList`.

Customize Collections Using the Abstract Classes

In *Effective Java*, Joshua Bloch (2017, item 18) sets out the arguments against the indiscriminate use of implementation inheritance to achieve software reuse. The most compelling of these is the *fragile base class problem*, in which the dependence of a

subclass on implementation details of its superclass leads to unexpected behavior when the superclass implementation changes. This simplified code illustrates the problem:

```
class Super {  
    public void foo() { System.out.println("called a superclass method"); }  
    public void bar() { System.out.println("called a superclass method"); }  
}  
  
class Sub extends Super {  
    @Override  
    public void bar() {  
        super.foo();  
        System.out.println("called a subclass method");  
    }  
}
```

Calling `Sub::bar` results in this output:

```
called a superclass method  
called a subclass method
```

but the maintainers of `Super::foo` could change its implementation to conform with the DRY principle (don't repeat yourself):

```
public void foo() { bar(); }
```

Although the behavior of `Super` is unchanged by this implementation change, calling `Sub::bar` now results in an infinite recursion, because its declaration overrides the declaration of `bar` in `Super`. The fragile base class problem has the same basic cause in all variations: implementation detail of a class is exposed to classes inheriting from it, so their behavior is liable to change unexpectedly when the superclass implementation changes. For this reason, *Effective Java* (Bloch 2017, item 17) advises strongly that classes should either be designed for subclassing or prevent it. As the primary initial designer of the Collections Framework, Bloch went some distance toward implementing this principle: even though the concrete classes of the framework are neither designed for inheritance nor do they prevent it, the framework does provide abstract classes that are specifically designed for this purpose and form an excellent basis for customization of existing collection classes.

Let's look at an example to help illustrate this concept. Suppose that your application needs a collection that behaves like `ArrayList` except for a new invariant: a single element, supplied at construction time, is to be held at all times in the zeroth position (the position with index 0). You might initially think that you only have to override the two `remove` overloads, along with `clear`, `removeIf`, and `removeAll`. But since the value of the fixed element isn't permitted to change, the `set` method must also be overridden. Then you might remember that there is an overload of `add` that would allow a new element to be inserted at the zeroth position. After that, you should recall that the iterators returned from `iterator` and `listIterator` both expose a `remove`

method, so these must take account of the collection’s invariant. And list iterators also expose an `add` method; that too must check for insertion at the zeroth position.

Clearly, there is a great deal of room for errors and omissions in making these changes. But the really compelling argument against this strategy is the fragile base class problem. Suppose you had made these changes before Java 21. Even though you know that in that release the introduction of sequenced collections provided `ArrayList` with new methods `addFirst`, `removeFirst`, and `removeLast`, you might reasonably hope that these would be implemented on top of the methods that you have overridden, and so would not affect the invariant. If that was your hope, you would be disappointed: `removeFirst` follows an optimized path that avoids calling `remove`. Instead, it omits checking validity of the index and directly calls `System::arraycopy`. As a result of this implementation detail in the new release of `ArrayList`, the invariant of your collection no longer always holds—unless you override `removeFirst`.

The implication is that you will have to inspect every future release of `ArrayList` to ensure that no similar problems recur. And because inheritance couples a subclass to the exact behavior of its superclass methods, you will have to inspect not only new superclass methods like `removeFirst`, but all the existing ones as well, in order to ensure that refactoring or optimization has not exposed your subclass to changed behavior.

Fortunately, there is a better alternative. When designing the Collections Framework, Bloch took his own advice, providing each of the main interfaces—`Collection`, `Set`, `List`, `Map`, and `Queue`—with a corresponding abstract class: `AbstractSet`, `AbstractCollection`, and so on. These partial implementations—the Javadoc calls them “skeletal implementations”—are designed for subclassing, most importantly by providing implementations for all but a few of the corresponding interface methods. For our current example, the Javadoc for `AbstractList` tells us that to extend it to create a concrete modifiable list, we need only override `size` and the indexed variants of `get`, `set`, `add`, and `remove`. [Example 17-3](#) shows a minimal implementation.

Example 17-3. Implementing a custom list

[org/jgcbook/chapter17/G_customize_collections_using_the_abstract_classes/](#)`ListWithFixedFirstElement`

```
public class ListWithFixedFirstElement<E> extends AbstractList<E>
    implements RandomAccess {

    private final List<E> backingList;

    public ListWithFixedFirstElement(E fixedElement) {
        this.backingList = new ArrayList<E>();
        backingList.add(fixedElement);
    }
}
```

```

@Override
public E set(int index, E element) {
    if (index != 0) {
        return backingList.set(index, element);
    } else {
        throw new IllegalArgumentException("Cannot change fixed first element");
    }
}

@Override
public void add(int index, E element) {
    if (index != 0) {
        backingList.add(index, element);
    } else {
        throw new IllegalArgumentException("Cannot change fixed first element");
    }
}

@Override
public E remove(int index) {
    if (index != 0) {
        return backingList.remove(index);
    } else {
        throw new IllegalArgumentException("Cannot change fixed first element");
    }
}

@Override
public E get(int index) {
    return backingList.get(index);
}

@Override
public int size() {
    return backingList.size();
}
}

```

The class `ListWithFixedFirstElement` creates a backing list, which in this case is an `ArrayList`, although any `List` implementation would be functionally equivalent. It handles a call to one of the overridden methods by first checking, if necessary, that it doesn't break the class invariant, then delegating the required action to this backing list. For other methods from the `List` interface, `AbstractList` provides implementations, all of which call the overriding methods at some point. Since the class is array-backed, it implements `RandomAccess` so that index-based algorithms will be chosen in preference to iterator-based ones.

This implementation is immune to the fragile base class problem. Taking the example mentioned earlier, besides its optimized implementation in `ArrayList`, `removeFirst` is also implemented as a default method declared in the `List` interface, to support subclasses that—unlike `ArrayList`—have not been modified to take account of its

introduction. This default method calls `remove(0)`, thus encountering the invariant check in `ListWithFixedFirstElement`.

You might have noticed that the reliance on implementations provided by the abstract collection means that this technique may fail to take advantage of any performance optimizations provided by the backing collection. But it does provide a quick, easy, and reliable way of customizing a collection. In case of performance problems, there is still the option to override performance-critical methods with delegated calls to optimized methods of the backing data structure. For example, `AbstractList::addAll` repeatedly calls `add(size(), e)` for each element of the supplied collection. Calling `size` in this way performs a separate index check for each element. It would be simple to override `addAll` and delegate the call directly to the backing `ArrayList`.

Conclusion

The guidelines in this chapter attempt to distill some lessons from the long experience accumulated by the Java community in using the Collections Framework, in the hope that they will provide useful techniques and help you to avoid some pitfalls in building systems that use the framework.

The next and final chapter reviews the design and evolution of the Java Collections Framework, and especially of some of its more controversial features, in the light of this collective experience. Clearly, the basic design will not change at this point! But it is still a very interesting exercise to review the decisions that were made, and to evaluate how they have worked out in practice. We'll consider five questions still being debated today.

Design Retrospective

At the start of [Chapter 17](#), we noted that the Java community's long experience with the Collections Framework provides the opportunity to derive some practical lessons in how to use it. This chapter uses that experience for a different purpose: to review some of the design decisions and trade-offs that shaped the Collections Framework at its inception and have continued to influence its evolution ever since.

Fundamental Issues in the Collections Framework Design

[The Java Collections API Design FAQ](#), written in 1998, when the framework was first published, begins with this question:

Why don't you support immutability directly in the core collection interfaces so that you can do away with optional operations (and `UnsupportedOperationException`)?

and answers it in this way:

This is the most controversial design decision in the whole API. Clearly, static (compile time) type checking is highly desirable, and is the norm in Java. We would have supported it if we believed it were feasible...

Over twenty-five years later, the first sentence is still undoubtedly true. A review of the proposal for the second edition of this book can stand for many comments on this decision:

Even the modest additions, such as immutable collections, are hamstrung by the requirement to implement the mutation methods of the Collections APIs—and just throwing `UnsupportedOperationException` is a horrible hack, to put it politely. The immutability of Java's new collections is not visible at type level at all, and this reflects a basic incompatibility between the now over 20-year-old and highly imperative design of the Collections, and the current trend toward a more [functional programming] style.

This comment, like the FAQ, contains an assumption that immutability is about *structural* concerns—principally the question of where mutation methods should appear in the class hierarchy. These should be distinguished from the *semantic* issues; namely, how precisely mutation can be controlled, and how information about an object’s mutability can be communicated to its users. In this section, we will attempt to analyze these issues in a wider context. Our aim is not just to review individual decisions in the design, but to survey the broad possibilities for the design of any framework for collections written in Java. Along the way, we hope to set criticisms of the Collections Framework in perspective.

The FAQ goes on to answer the first question by imagining the consequences of replacing the single all-purpose interfaces, like `List`, with modifiable, unmodifiable, and immutable variants—for example, `ModifiableList`, `UnmodifiableList`, and `ImmutableList`. The implication of the discussion is that unmodifiable interfaces are simply ones that lack mutator methods. But an interface cannot control its implementations; it can only express the capabilities it exposes. So interfaces that expose only read methods should be named by the fact that they are readable. In the following discussion, we will use the names `ReadableList`, `ReadableSet`, and so on, until we have had an opportunity to explore ways of adding constraints to enforce or communicate the semantic properties of unmodifiability and immutability.

Desirable Characteristics of a Collections Framework

For thoroughness, a retrospective should begin with a review of the aims of the design. What are the desirable characteristics of a collections framework, specifically one written in Java?

1. It should have a small surface area. This was a priority of the original design, on the basis that the framework would be in everyday use by millions of developers, of varying experience. Making it small enough to be easily internalized increases the likelihood that a developer will understand it well enough to use it appropriately in application design. In answering its first question, the Collections API Design FAQ argues that the possibility of fully comprehending the entire type hierarchy would be lost if an unmodifiable version of every existing interface were provided, together with all those required by partly modifiable and unmodifiable views.
2. It should be readily extensible. This requirement follows from characteristic 1: by definition, a minimal framework will not satisfy specialized use cases, so the path to customization should be straightforward.
3. As many errors as possible should appear at compile time rather than run time. In particular, we would like if possible to avoid run-time errors of the `UnsupportedOperationException` variety. This is an application of the Interface

Segregation Principle ([Martin 2002](#)): “a client should not be exposed to methods that it does not need.”

4. It should maximize the ability to enforce semantic constraints. An example of a constraint that can be enforced is uniqueness in a collection, which is specified in the contract of the `Set` interface. One that cannot be enforced in the Collections Framework is for a `List` that can only ever contain an even number of elements.
5. The properties of a collection should be deducible from the type of its interface. For example, in the Collections Framework, a user of `SortedSet` can be confident that iteration over the elements of the collection will return them in their natural order or the order defined by the comparator of the set.

An important part of designing a framework is weighing up the relative importance of these characteristics when they conflict. For example, a fine-grained API designed to satisfy characteristic 3 would have very many specialized interfaces, making characteristic 1—a small API—unachievable. Similarly, an extensible framework respecting characteristic 2 will allow implementation of immutable or unmodifiable interfaces by mutable classes.

Unmodifiability via Subtyping

Suppose, then, that we set out to design a collections framework using the type hierarchy to prioritize characteristic 1, while respecting the others as far as possible. Intuitively, it appears that obedience to the Interface Segregation Principle would lead to separate interfaces for read and write operations on collections (ignoring for the moment the complications of the partially writable collection views mentioned in [“Contracts” on page 149](#)). How could this separation be achieved? One can imagine three possibilities:

Readable types as supertypes of read/write types

For example, `ReadableCollection` could be a supertype of `Collection`, exposing only nonmutating methods and allowing `Collection` to extend `ReadableCollection` by adding mutation methods. In the same way, `ReadableList` would be a supertype of `List`, `ReadableSet` a supertype of `Set`, and so on, satisfying the Interface Segregation Principle. But a readable list is not the same as an unmodifiable one; we cannot replace the name `ReadableList` with `UnmodifiableList` without, according to the Substitution Principle, defining an `ArrayList` as a kind of `UnmodifiableList`, which is clearly unacceptable.

Read-only types as subtypes of read/write types

This is the design chosen by the [Guava library](#). There, for example, `ImmutableCollection` (in Guava terms, “shallowly immutable”; in our terms, “unmodifiable”) is a class that inherits from `java.util.Collection`. The Guava designers use classes to represent types instead of interfaces because, before the

introduction of sealed types in Java 17, this was the only way to prevent external subtyping. As a result, when you receive a reference to a Guava immutable collection, you can be confident that it refers to a class in which the mutation methods will all throw `UnsupportedOperationException`, rather than a class that inherits from the Guava type, overriding those methods; in other words, there is a guarantee of (shallow) immutability.

Parallel type hierarchies

This is the model for [Eclipse Collections](#). Each interface exists in two versions: `MutableCollection` is paired with `ImmutableCollection`, `MutableList` with `ImmutableList`, and so on. The top-level mutable interfaces extend the corresponding Collections Framework interfaces, with `MutableCollection` extending `java.util.Collection`, etc. Methods are supplied to convert from mutable to immutable types and from immutable types to types in the Collections Framework. In our terms, the immutable interfaces are readable: there is nothing to prevent you from implementing them with a mutable class. In this framework, the names of the immutable types serve as an indication of design intent rather than a guarantee of immutability.

Notice that all three possibilities involve at least doubling the number of interfaces in the type hierarchy. In the first of these three options, for the top-level interfaces to conform to the Substitution Principle they have to be called `ReadableList`, `ReadableSet`, and so on; the name indicates that this design has quite different semantics from the immutable interfaces that were our initial goal. The second option, realized by the Guava library, does guarantee the unmodifiability of some interfaces but, again in obedience to the Substitution Principle, must limit extensibility and provide implementations of mutator methods, throwing `UnsupportedOperationException`. The third option adds the difficulties of interconversion between the parallel hierarchies to the choice between extensibility and conformance with characteristic 5 in the previous section—in other words, a client receiving a reference of type `ImmutableXXX` cannot know whether the referent is genuinely unmodifiable or instead a member of a subclass with mutation methods.

Limitations of Subtyping

The type system is the primary vehicle for detection of compile-time errors, so it should be as precise as possible, other things being equal. But it is clear from the discussion in the preceding section that in practice other things never will be equal: each of the three possibilities explored there carries consequences for the semantics of the types, for extensibility, and for the size of the framework. Overreliance on the type hierarchy for solving API design problems encounters—or ignores—other issues. Here are three important ones:

- Many desirable system properties cannot be represented by the Java type system. For example, for an object to be immutable, its entire object graph—that is, the object itself and everything it refers to, directly or indirectly—must not observably change after construction, as we saw in “[Immutability and Unmodifiability](#)” [on page 147](#). For a type checker to verify this at compile time, it must be able to determine for any mutable component that the object graph has exclusive access to it. To see how significant this problem is, you only need to observe that nearly all collections are based ultimately on arrays, and that Java arrays are always mutable.

Determining exclusive access is closely related to the problem of escape analysis ([Gay and Steensgaard 2000](#)), a compiler optimization that can only be performed at run time after dynamic method binding has taken place. So it is not feasible to enforce immutability in Java (or any object-oriented language) by means of static typing. It is possible, of course, for type systems to *represent* immutability without enforcing it; for example, the proposal for immutable collections in the Kotlin extension library [kotlinx](#) states that “immutability is enforced solely by...contract: the implementors are required to keep the collection elements unchanged after the collection is constructed.”

Even for less ambitious constraints, extraordinary measures may be needed in order for them to be described by the type system. For example, consider the possibility, alluded to in the reviewer’s comment that opened this section, of introducing an `Unmodifiable` type to refer to the unmodifiable collections introduced in Java 9. Should this type also be returned from the unmodifiable wrapper methods of the `Collections` class? If so, a client receiving a reference to an `Unmodifiable` collection would not be able to determine whether it really was unmodifiable or could instead be modified by any code that has a reference to the backing collection. Distinguishing these two situations by means of the type structure requires still greater refinement; following this route will lead to a very fine-grained (and therefore large) type hierarchy. In the next section, we consider alternatives that may be better adapted to representing semantic properties.

- If you attempt to use the type system to counter completely the most common criticism of the Collections Framework—that is, the “horrible hack” of `UnsupportedOperationException`—you encounter an arguably worse problem: the size explosion in the APIs that would go along with the first and third options described in the preceding section. In fact, as mentioned earlier, the Collections API Design FAQ goes further in justifying the absence of unmodifiable (in the language of the FAQ) interfaces by imagining the number that would be required to reflect all the various restrictions on collection view operations. For example, the list returned from `Arrays::asList` would need an interface `FixedSizeList`, which in turn would require a new list iterator interface `FixedSizeListIterator`, and so on. A less extreme position could be argued; for

example, providing unmodifiable interfaces for the Java 9 unmodifiable classes, but still allowing `UnsupportedOperationException` to be thrown from view methods.¹

Since eliminating optional methods via the type system comes at such a high price, it's worth considering the alternatives to focusing on the language and API level. In the next section, we will consider other ways of defining the behavior of collections, including development practices that we have already seen (in “[Respect the ‘Ownership’ of Collections](#)” on page 289) that can eliminate the practical problems associated with `UnsupportedOperationException`.

- The strongest criticism in the reviewer's comment at the beginning of this section is that the unmodifiability of the Java 9 collections is not reflected in the type system. This criticism has some force: as we have seen, it is in fact possible to define readable types precisely, by excluding mutator methods. But the cost of doing this effectively is mentioned less often: in an object-oriented language, inheritance normally allows the possibility of creating an implementation with added mutator methods. Preventing this possibility by the use of sealed interfaces or by defining collection types as final classes (as Guava does) also excludes the possibility of extending the framework.

Obviously, the static type hierarchy has an important place in the framework design. But it is not the only tool available to the API designer; others may be better suited to providing the semantic properties that developers need in practice. The next section reviews the possibilities.

Complementing Static Typing

Taking a broad view of the ways in which an API design can be specified, we see that static type constraints are only one of the tools available to the designer: others include static analysis, annotations, type names, and contracts. Type names are important, as the example of `kotlinx` immutable types shows, and many simple semantic rules can be encoded for use by static analyzers.² But in practice, the most important complement to static typing is provided by the contracts of software components. In fact, you can see static type constraints as very coarse-grained contracts.

¹ This does not address the formal criticism that the existence of optional methods violates the Substitution Principle (see “[Subtyping and the Substitution Principle](#)” on page 13). This formal criticism can be met with a formal response: the “principle” is not in fact an unbreakable *principle*, but a *description* of systems in which any use of a supertype, such as an interface, can be replaced by an instance of one of its subtypes, such as an implementing class, without changing the meaning of the program. In fact, since the interface contracts specify mutation operations as optional, the Substitution Principle does indeed describe the behavior of the views and unmodifiable collections. The real question, though, is the practical one: Does the existence of optional methods represent a real problem in practical software development?

² For an overview of Java static analyzers and their use, see Valeev ([2024](#), §1.4).

For example, in Java a collection can only be typed as either modifiable or unmodifiable, whereas a contract can specify its semantics in detail—that it can be modified, but only in certain ways or within certain limits. The boundary between the two may even move: until Java 21, encounter order was specified by contracts on the classes and interfaces implementing it, but the introduction of sequenced collections brought it into the type system.

Contracts may seem like a weak kind of enforcement compared to compile-time type checking, but in fact the Collections Framework has used it from the outset without encountering criticism. The interface `Set` extends `Collection` without any type differentiation; the methods of `Set` are identical to those of `Collection`. All that ensures the property of element uniqueness is contracts: the general contract for `Set` and the contracts for its methods. Element uniqueness is like the precise mutability constraints just mentioned, in that it is a collection property that can only be communicated through a contract or a formal specification.

Contracts as we usually think of them (and as they are described in “[Contracts](#) on [page 149](#)”) specify pre- and post-conditions for a software component. The precondition represents what it can rely on in the state of its environment and its input values when execution starts, and the post-condition represents what it promises for the environment state and any return value when execution finishes. This style of contract is not always adequate for describing the kind of constraint we discussed in “[Respect the ‘Ownership’ of Collections](#)” on [page 289](#). Consider this code:

```
void foo(Collection<Object> c, Object x) {  
    assert c.contains(x);  
    bar();  
    System.out.println(c.contains(x)); // true or false?  
}
```

Given that `bar` may call methods on other objects and that the collection `c` may be widely shared, the pre/post-condition contract for `bar` cannot specify whether the contents of `c` can be affected by its execution. However, `c` can be governed by a different style of contract, which can state for a collection what it requires from its environment and how it will behave when it is accessed from code other than its “owning” object. This is called a *rely/guarantee* contract and, like a pre/post-condition contract, it is made up of two parts: a *rely* condition, which in the previous example specifies what changes may be made to `c` during the execution of `bar`, and a *guarantee* condition, which specifies the outcome of `c`’s interaction with its environment during the execution of `bar`.

Rely/guarantee terminology originates from the field of formal specification ([Jones 1983](#)), which provides precise application-level semantics for program components that run in an interfering environment. This is not applicable for the general description of collections, whose precise behavior obviously varies between applications. But in constructing programs in which collections are transferred between system

components, it is very helpful if the collections themselves can respect minimal rely/guarantee conditions. In particular, it should be possible to communicate, to a class or method being supplied with a collection reference, the information that this collection:

- *Relies* for its correct behavior on not being mutated
- *Guarantees* either that it will not be modified, or that no part of its object graph will be mutated

This information helps to enable the designer of a system component to know *what* is permissible to do with a collection that has been passed across a component boundary, and *how* to do it. For example, a program requiring repeatable reads from a collection needs to know whether it can rely on a collection that it has been passed remaining unchanged—otherwise, it would have to take a snapshot and read from that. Similarly, it needs to know, given a collection that it has been handed, whether the action of mutating it is acceptable to the collection’s owner (see “[Respect the ‘Ownership’ of Collections](#)” on page 289).

More fine-grained constraints are in a gray area between collection and application semantics. For example, it is possible to imagine communicating the constraint that a collection (e.g., a collection of log records) should only be appended by a client, either by assigning the collection a type specialized to allow only append operations or by specifying that requirement on the program by another means, such as by static analysis rules, by documentation of the contract in natural language, or (with greater precision but also greater difficulty) in a formal specification language.

The API design should start with the choice of the semantic properties to be represented; the question of *how* they should be represented is important but secondary. The answer to that question may even change over time, as we have seen with the introduction of sequenced collections. Static type constraints and contracts are only a means to an end: that of communicating and enforcing the semantics of the API.

Lumping and Splitting

In *The Principles of Classification and a Classification of Mammals*, the naturalist George Simpson ([1945](#)) wrote:

Splitters make very small units—their critics say that if they can tell two animals apart, they place them in different genera...and if they cannot tell them apart, they place them in different species....Lumpers make large units—their critics say that if a carnivore is neither a dog nor a bear, they call it a cat.

The contrasting tendencies of lumping and splitting, usually seen in classification systems, can also describe forces in software design. As a design mode, lumping produces models with powerful general rules, relatively simple and easy to understand,

but with many special cases. Splitting produces more precise models, but with the loss of generality leading to more complexity and possible duplication. For example, consider `IdentityHashMap`, a class with many properties in common with other maps, but with the important difference that its equivalence relation is identity rather than value equality. Was it the right decision to make `IdentityHashMap` extend `Map`? Compared to other maps, it has both similarities (its lookup mechanism) and differences (its equivalence relation; see “[Defining a Set: Equivalence Relations](#)” on page 182). It is currently lumped with other maps; to have split it would have provided greater precision, at the expense of duplication of many `Map` methods in this new type.

One of the main goals of the original Collections Framework design was a small and easily understandable API, which naturally led to a lumping model. Everyday use of an API exposes the downside of the trade-offs inherent in its design, so the exceptions in the rules of the lumped-together Collections Framework classes can be irksome. But a balanced retrospective should assess individual problems in the context of the overall design: in the round, would a split model have been a better alternative than the lumped one that was chosen? As the quotation at the start of this section implies, your answer may well reflect your personal view of the relative value of generalization compared to precision.

Summary

This section has been long, as we explored the conflicting design goals of the Collections Framework, the different tools with which they could be achieved, and the underlying design philosophies by which you may judge them. For most of the issues we have considered, there are no right or wrong answers, because every design decision implies trade-offs that are, in the end, value judgments. We feel that the principles laid down in this book, especially in [Chapter 17](#), should enable you to work safely around `UnsupportedOperationException` and other problems. But, however you assess the individual choices that formed the Collections Framework, it should be acknowledged that it was a considerable achievement, primarily by Joshua Bloch, to navigate through the conflicting forces that we have described. Without needing to minimize the shortcomings of the design, we can recognize that they have not seriously impacted the achievements of a framework that has been successfully used to build countless information systems over the last three decades. Design ideas will continue to evolve, as they should, but that achievement remains.

nulls

Few programming language features arouse more heated controversy—or, more precisely, condemnation—than `null`. A web search for articles listing problems with `null` returns numerous five- and ten-thousand-word blog posts with titles like “The

Worst Mistake of Computer Science.” Articles presenting the opposite viewpoint are much less common. Despite this, `null` is everywhere, in application programs and in databases. Unfortunately, the Java Collections Framework fully reflects this ambivalence in its complex and inconsistent treatment of `null`.

The case against `null` actually falls into two parts. The more general one concerns type systems that do not exclude `null` values as valid references.³ In any language that allows `null` to be a reference value, programs must include tests for nullity as a special case. Probably the most commonly occurring run-time exceptions in Java are `NullPointerExceptions`, the result of attempting to dereference `null` references, but there are other situations in which `null` must be given special treatment. For example, linear probing, as used in `IdentityHashMap` (see Figure 15-3), must be able to distinguish `null` keys from unused table locations. So `null`, as a key value, must be replaced internally by a special non-`null` object, and `nulls` must be translated to and from this object on storage and retrieval. This workaround avoids the central issue concerning `null` for the Java Collections Framework design: it often has two roles, as a valid collection element and as a sentinel, an indicator of the absence of an element.

Inconsistent handling of `null` can be seen as the original sin of Java collections, preceding even the birth of the Collections Framework. The Java 1.0 class `Hashtable` (the precursor of `HashMap`) refused `null` as either key or value, whereas `Vector` (the precursor of `ArrayList`) accepted `null` values as elements. The Collections Framework, introduced in Java 2, attempted to resolve the issue in a consistent way, initially by accepting `nulls` as valid values in almost all collections. This attempt at consistency arguably worsened the situation: the Javadoc for the method `Map::get` specifies its return value like this:

If this map permits `null` values, then a return value of `null` does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to `null`. The `containsKey` operation may be used to distinguish these two cases.

Although the contract for the interface `Map` explicitly acknowledges that implementations might in future be `null`-hostile refusing to accept `null` as a valid element, the original Collections Framework implementations—`HashMap`, `IdentityHashMap`, and `LinkedHashMap`—were in fact all `null`-tolerant.

This was the situation until Java 5 introduced `ConcurrentMap` and its implementation `ConcurrentHashMap`. At this point, the chicken of the `Map::get` contract came home to roost. The implication of the Javadoc is that *following* a call of `Map::get`

³ Tony Hoare (2009), who introduced `null` to the type system for references in Algol W, described this as his “Billion Dollar Mistake.”

that returns null, the client should *then* call `Map::containsKey` to disambiguate the result. But in a concurrent environment, this check-then-act sequence (see “[Using synchronized collections safely](#)” on page 303) cannot be executed reliably without locking the collection. The solution adopted was to forbid `ConcurrentHashMap` to contain null keys, although the interface `ConcurrentMap` did not override the contract for `Map` methods to exclude null-tolerant implementations.

It did, however, introduce the compound atomic operations `remove`, `replace`, and `putIfAbsent`, which in Java 8 were lifted to `Map` at the same time as `merge`, `getOrDefault`, and the `compute*` methods were introduced. Like the older methods—`get`, `put`, and `containsKey`—`remove`, `replace`, and `getOrDefault` treat an existing null as a valid value. The other Java 8 methods do not distinguish between an existing null value and no mapping at all, but they differ in their behavior given a null value to place in the map: if the lambda supplied to `compute`, `computeIfPresent`, or `merge` evaluates to null, they will remove the existing mapping. (`computeIfAbsent`, however, will not remove an existing null value; this is consistent, in a way, with its view of such a mapping as being nonexistent.)

The two Java 8 methods that accept a value rather than a lambda are `putIfAbsent` and `merge`. Given a null value, `putIfAbsent` will place it in the map (although it will not subsequently recognize it as an existing value). The behavior of `merge`, given a null value, is consistent, in a way, with its null hostility, but not with the map API in general, as it throws a `NullPointerException`.

The root cause of the many contradictions in the treatment of null is that at Java 5, the Collections Framework merged with the concurrent library that implemented the ideas of Doug Lea’s (1999) *Concurrent Programming in Java*. The classes of this library were consistently null-hostile (for the reason given previously), so lifting methods from a null-hostile class like `ConcurrentHashMap` into an existing interface like `Map` inevitably introduced inconsistencies within the single interface. This faced the designers of the Java 8 streams and Java 9 unmodifiable collections with an unpalatable choice—but whichever direction they chose, it could only propagate the existing inconsistencies in the framework.⁴ (The eventual decision for streams was mostly null tolerance; for unmodifiable collections it was null hostility.)

Even hindsight does not provide a clear moral to this story. The concurrent interfaces and classes that arrived in Java 5 have been central to Java’s success and have powered innumerable concurrent systems in the 20 years of their use; it is impossible to

⁴ An alternative design could have sidestepped this choice by using the type system to distinguish null-hostile from null-tolerant collections, in the same way that `Collection` and `Set` expose the same methods but with different semantics. This choice of splitting over lumping (see “[Lumping and Splitting](#)” on page 318) would have traded the current inconsistency for significant interface proliferation.

imagine modern Java without them. Redesigning them to bring their treatment of `null` into line with the existing Collections Framework types would have required extraordinary (and disproportionate) effort.

Some of today's Java designers believe that `null tolerance was always a mistake`. But when a design mistake is discovered, should new features provide improved behavior at the cost of inconsistency, or does consistency condemn the mistake to be repeated in perpetuity? This question arises repeatedly in the evolution of a major software platform like Java. Sometimes the balance falls on the side of accepting inconsistency. Of course, a third alternative does exist: it is possible to incorporate improved behavior and also maintain consistency, at the cost of making incompatible changes. But such changes impose a very high cost on users, and for that reason Java has always rejected this strategy.

Inconsistent with equals

In “[Consistent with equals](#)” on page 37, we noted that the documentation for the `Comparable` interface recommends that the `compareTo` method should be *consistent with equals*—that is, that two objects should satisfy the `equals` method if and only if they compare as the same. But the documentation does not mention either the reasons why you might choose to depart from this recommendation or the consequences of doing so. The counter-example given in “[Consistent with equals](#)” on page 37 is that of `BigDecimal`. A `BigDecimal` instance consists of two parts: the numerical value and the precision with which it is stored. Rounding behavior depends on the latter, so equality of two `BigDecimal` instances depends on both parts. For example, the result of dividing 2 by 3 is $2/3$; to write this as a decimal fraction, it must be rounded. The result of `BigDecimal` division is rounded to the precision of the dividend, as the following two `assert` statements show:

```
assert new BigDecimal("2.0").divide(BigDecimal.valueOf(3), HALF_UP)
    .equals(new BigDecimal("0.7"));
assert new BigDecimal("2.00").divide(BigDecimal.valueOf(3), HALF_UP)
    .equals(new BigDecimal("0.67"));
```

Two values that satisfy the equality relation should behave the same way, so the two instances created by `new BigDecimal("2.0")` and `new BigDecimal("2.00")` should not be equal, and indeed `BigDecimal("2.0").equals(BigDecimal("2.00"))` evaluates to `false`. But, of course, the numerical values that they represent are equal, so `BigDecimal("2.0").compareTo(BigDecimal("2.00"))` evaluates to `0`, and data structures that use numerical comparison in place of the equality relation will be unable to distinguish them. For example, this `assert` statement executes without error:

```
assert new TreeSet<>(Set.of(new BigDecimal("2.0")))
    .contains(new BigDecimal("2.00"));
```

while of course:

```
assert ! new HashSet<>(Set.of(new BigDecimal("2.0"))).contains(new BigDecimal("2.00"));
```

Since set equality is computed by testing containment, it is liable to lose symmetry when sets with different equivalence relations are being compared:

```
Set<BigDecimal> hs = new HashSet<>(Set.of(new BigDecimal("2.0")));  
Set<BigDecimal> ts = new TreeSet<>(Set.of(new BigDecimal("2.00")));  
assert ts.equals(hs);  
assert ! hs.equals(ts);
```

This is the kind of “strange” behavior that the documentation for `Comparable` warns will result from defining natural orders inconsistent with `equals`.

`BigDecimal` is unusual in having a natural order that is inconsistent with `equals`. By contrast, it is very easy to create a comparator with an inconsistent `compare` method. This occurs in a common scenario in which a collection of objects is to be maintained in the order of a single field. For example, to maintain a collection of `Book` objects (see “[Use Records as Composite Keys](#)” on page 299) in order of their page counts, we might use an ordered collection:

```
new TreeSet<Book>(Comparator.comparingInt(Book::getPageCount))
```

The consequence has been accidentally rediscovered by countless hapless developers: the result of adding two `Book` objects with the same page count to this set is that the second one, being regarded as a duplicate, is “lost.”

A comparator can be redesigned, but the natural order of a class is fixed. So to store `BigDecimal` values in an ordered collection, you would need to define a comparator that would impose a *total order* on the values—that is, an ordering that connects the elements of every possible pair of values. Of course, you could define a comparator that would make the `BigDecimal` value corresponding to “2.0” greater—or less—than that corresponding to “2.00”, but it is not obvious how useful such an order would be.

Does this problem reveal a flaw in the design of ordered collections, of `BigDecimal`, or of both? It certainly explains the advice to define orderings that are consistent with `equals`, wherever possible—but we have seen that, for `BigDecimal` at least, there is no such reasonable ordering. Then, should the ordered collections use equality for their equivalence relation? If so, the consequence would be that ordered collections could contain the `BigDecimal` values corresponding to both “2.0” and “2.00”, when the contract of every method of these collections assumes the existence of a total ordering over the elements.

The problem does not lie in either of these design decisions, but in the misleading expectation (reinforced by the contract) that every `Set` will use the `equals` method as its equivalence relation. We discussed this problem in “[Defining a Set](#):

[Equivalence Relations](#) on page 182; although `equals` is the right choice for the equivalence relation of `HashSet` and most other set implementations, it is nonetheless only one candidate. The equivalence relation for ordered collections can, if it is inconsistent with `equals`, regard some nonequal objects as duplicates, whereas the equivalence relation for a set created from `IdentityHashMap` regards equal objects as non-duplicates. These equivalence relations simply view the same data differently for different purposes. Problems arise when sets (and maps) are processed with incorrect assumptions about their equivalence relation. For example, applying binary operations like `equals`, `addAll`, `removeAll`, or `retainAll` to sets with different equivalence relations leads to unexpected results. Equally, if your code is passed an empty `Set<String>`, you might be surprised to discover that adding the strings "one" and "two" to that set increases its size by only one. But that is what will happen if the implementation is in fact a `TreeSet` whose comparator orders strings by length only, so that two strings of the same length are duplicates. The problem here is that the code is relying on the general contract for `Set`, which takes no account of different equivalence relations, and so is violated by `TreeSet`.

As with `null` tolerance versus `null` hostility, a splitting rather than a lumping design for sets (and maps) with different equivalence relations could have avoided these problems—at the cost, again, of significant type proliferation.

Object Versus E

At several points in this book, we've noted the apparent strangeness of the signatures of the `Collection` methods `contains`, `remove`, and `retain`, each of which take a parameter of type `Object`, and the corresponding collection-oriented versions `containsAll`, `removeAll`, and `retainAll`, which take a `Collection<?>`. Why are these methods not generified to take parameters of type `E` or (respectively) `Collection<E>`?

A moment's thought shows that these methods, which remove elements from the collection or test for membership, do not have the potential to compromise its type safety—unlike `add` and `addAll`, whose signatures do indeed specify the type `E`. So the designers *could* safely choose these parameter types. But why did they? Different reasons have been proposed (including by the designers themselves), including:

- Backward compatibility. To minimize the number of breaking changes caused by the introduction of generics, the only methods generified were those that were *necessary* to ensure type safety, like `add` and `addAll`.
- Allowing the use of bounded wildcard types. Suppose that a collection parameter to a method is given a type `Set<? extends Foo>`. If `contains` required that parametric type, it could never be used in this situation, since no argument

could be supplied of type `? extends Foo` (except `null`). Giving the parameter of `contains` the type `Object` ensures that it can be used with collections of any type.

- Accommodating unrelated types. The most compelling reason is the one that we saw in “[Removing Elements](#)” on page 171: that in many situations, such as calculating the intersection of two collections, the types of the elements may be completely unrelated. In that example, the intersection being calculated was between a collection of type `PhoneTask` and a collection of its supertype `Task`, but even this relationship is unnecessary. For example, think of calculating the intersection of two collections of `List` objects. Two `Lists` are equal if they contain the same elements, so `retainAll` could plausibly be used to preserve in a `Collection<ArrayList>` only those `ArrayLists` that were equal to some element of a different `Collection<LinkedList>`. The only parameter type for `retainAll` that makes that possible is `Collection<?>`.

The trade-off between the two alternatives for these collection method types is quite subtle. On the one hand, it can be argued that using `Object` as the parameter type means that some errors will be missed that would be caught by the more precise typing. After all, one of the key advantages of generics is that their precise typing enables more errors to be caught at compile time. But in cases like those mentioned here, where the precise typing would be inappropriate, it would have to be evaded with additional unchecked casts—one of the major sources of imprecision in the generics type system. Whether you feel that the designers made the right or the wrong choice, it is worth knowing the issues that lay behind the decision.

Concurrent Modification

A central concern of the design of a collections library is *concurrent modification*. We have previously discussed the various concurrent modification policies in the JDK. The core collections implement a fail-fast policy, as described in “[Synchronized Collections and Fail-Fast Iterators](#)” on page 158. The concurrent collections introduce two additional concurrent modification policies: a snapshot policy and a weakly consistent policy. These are discussed in “[Mechanisms of concurrent collections](#)” on page 159. In this section we’ll try to reconstruct the reasoning that took the Collections Framework to its present position with these concurrent modification policies, why the fail-fast policy was chosen for most non-concurrent collections, and how well these policies have worked since their introduction.

The original collection classes, predating the Collections Framework, were `Vector` and `Hashtable`. These were iterated using the `Enumeration` class. There was no concurrent modification policy mentioned at all in any documentation. Depending upon the implementation, concurrent modification could result in elements being skipped during processing, elements being processed multiple times, or even infinite loops.

The development of the Collections Framework was an opportunity to improve on this situation.

A snapshot policy seems like the easiest policy to understand and to implement. Unfortunately, it is also the most expensive policy to implement. It essentially requires a full copy of the collection to be made whenever an `Iterator` is created. The snapshot policy is thus unsuitable for most collections. In the Collections Framework it's used only for the collections `CopyOnWriteArrayList` and `CopyOnWriteArraySet`, for which modifications are presumed to be rare.

The main alternative at the time, a weakly consistent policy, would be applicable to the non-concurrent collections as well as the concurrent ones. The primary issue is that the semantics are so weak that they can result in unpredictable behavior that would be startling to find in a non-concurrent program. Consider a single-threaded program iterating a `HashSet`. The `Iterator` implementation maintains state pointing to its current position in the `HashSet`'s internal table. If an element is added in the middle of the iteration, it might end up earlier or later in the internal table, depending on the element's hash code. Worse, if the addition caused the table to be rehashed, elements might end up in completely different relative positions in the new table, invalidating the iterator's current position. If the iteration were to continue, some existing elements might be repeated, some might be skipped, and the newly added element might or might not appear later in the iteration. These are indeed weak semantics! While this sort of inconsistency is acceptable in a concurrent system, it seems unreasonable for a single-threaded system to produce such unpredictable behavior.

In the quest to provide predictable behavior for concurrent modification, one might reasonably ask whether in-progress iterators could be adjusted appropriately when the underlying collection is mutated. While this might not work for a `HashSet`, it could work for a `List`. If an element is added to or removed from a `List`, it could conceivably update its iterators, which would reposition themselves depending upon whether the modification occurred before or after the iterator's current position. Naturally, this would require the collection to keep track of its iterators. This is a common approach for implementing *robust iterators*, as described in *Design Patterns* (Gamma et al. 1995, 261).

Applications create fresh iterators every time they iterate a collection. This would result in iterator objects building up indefinitely, unless some kind of “close” operation were added to the `Iterator` interface. This style of API would be quite inconvenient for clients, as it would require them to enclose every iteration within a `try-finally` or `try-with-resources` statement. Not requiring clients to close iterators is a feature. It makes the API considerably easier to use, and it allows clients to abandon an iteration at any time, such as when returning an element immediately when it's found.

Without a “close” operation, it is still possible for a collection to keep track of its iterators by using a `WeakReference` to detect when a client is no longer using an iterator. A `WeakReference` allows code to hold a reference to an object as long as strong references to the object exist. When there are no longer any strong references to the object, the `WeakReference` is automatically cleared (set to `null`). This technique is used by the `ArrayBlockingQueue` concurrent collection. It adds considerable complexity, however, and it seems unreasonable to burden every collection implementation with the responsibility of tracking its iterators using weak references.

Other platforms deal with this issue in different ways. The Smalltalk collection classes, for example, don’t have any general policy for concurrent modification; such cases can exhibit unpredictable results. In this way, they are similar to the original `Vector` and `Hashtable` collections from JDK 1.0. The situation in Smalltalk is recognized as an issue, though, and a common idiom is to copy the collection and iterate the copy, allowing modifications to the original. In some Smalltalk implementations there are extensions or tools available to help detect concurrent modification.

In the [C++ Containers Library](#), concurrent modification is referred to as *iterator invalidation*. Operations on a collection may never invalidate iterators on that collection, or may do so sometimes, or always. When an iterator has been invalidated, subsequent operations on it can result in *undefined behavior*. Almost anything can happen: invalid results can be returned, data corruption can occur, or the program can crash. It is the programmer’s responsibility to avoid using any iterator that has been invalidated.

Instead of trying to provide predictable behavior, the Collections Framework’s *fail-fast* policy considers concurrent modification to be a programming error. The fail-fast policy attempts to detect when concurrent modification has occurred, and throws a `ConcurrentModificationException` when it succeeds. Note that this detection is not guaranteed: there are cases where concurrent modification has occurred but an exception is *not* thrown. Thus, this mechanism is useful for detecting errors, but it cannot be relied upon to do so in all cases.

As an alternative to direct mutation of the collection, a mechanism is provided to mutate the collection using the `Iterator` itself. This simultaneously updates the iterator and the underlying collection, allowing them to remain consistent. But the modern idiom for iteration is to use a `foreach` loop, which hides the `Iterator`. Of course, the classic `for` loop is still available, but many programmers prefer the alternative of avoiding in-place mutation entirely by copying elements into a new collection as they are processed. The Stream API encourages this idiom.

The fail-fast policy is rather unusual for a library. The exception is thrown not in response to a precondition violation, but rather when a likely programming error has been detected. Further, the exception is not thrown by the code committing the

programming error, but rather by the “victim” of the error. If *this* code is iterating a collection, and other code modifies the collection concurrently, it is *this* code that detects the modification and throws the exception. Beginning programmers, in particular, often find this behavior confusing.

`ConcurrentModificationException` can also pose difficult issues for seasoned programmers, though. It is thrown when the detection mechanism is triggered, which is on a “best effort” basis (this really means “minimal effort”). As such, there are times when the exception is thrown only intermittently, and this can result in an unpleasant surprise in production even after testing has completed successfully.

Throwing an exception is sometimes inconvenient. An alternative to fail-fast might be called *terminate-fast*: instead of throwing an exception, the iteration code would arrange to terminate the loop as quickly as possible and log a message. This is not in any sense more correct than throwing an exception; the iteration will still be incomplete, and this might result in data being partially updated or left unprocessed. However, logging a message might be a better way to notify the developers that something has gone wrong, without the disruption caused by an exception.

The primary contribution of the Collections Framework in this area is to have developed the concept of a *concurrent modification policy*. Different collection implementations have different policies depending on their expected usage. This has created an overall expectation among Java developers that concurrent modification is handled in a reasonably well-defined way. This is clearly preferable to having no policy at all, or to having the results be undefined, where the burden of detection and diagnosis of errors falls completely on the programmer.

The fail-fast policy can be confusing at times, but it has the advantage of a simple and low-overhead implementation. Throwing an exception is sometimes inconvenient, but that inconvenience is counterbalanced by the policy-surfacing bugs that might otherwise have gone unnoticed. The designers believed that it was one of the most valuable innovations of the framework, and on balance, we can say that time has justified that judgment.

Afterword

This book began life when Phil Wadler returned to Scotland from America in 2003 to take up a chair at Edinburgh University. At that time, the Generic Java proposal ([Bracha et al. 1998](#)), which had been struggling for acceptance in the Java Community Process for five years, had finally been slated for inclusion in Java 5. Phil was one of the authors of Generic Java, and I had some experience with functional programming, so a collaborative book project on this major upcoming feature seemed like an obvious opportunity. As it turned out, I couldn't match Phil's deep background in type theory, so he suggested that a complement to the relatively theoretical treatment of generics should be a part on its most significant practical application, in the Java Collections Framework. This wasn't one of my specialisms, and my initial efforts were poor, to say the least. I am forever indebted to Josh Bloch, who wrote an extraordinarily generous and detailed review of a late draft, correcting every minor error and transforming the quality of the material.

Returning to the book nearly 20 years later has been an interesting experience. Before starting work, I was apprehensive that I might find that neither generics nor collections had changed enough to justify a new edition: after all, Java is famous for its backward compatibility, so surely the material should have been timeless? Of course, once I engaged with it in detail, I was surprised that I could have been so naïve: after all, isn't it ridiculous to think that programming idioms and style could stay unchanged over two decades? But most surprising of all was that both of these things could be true—Java has been outstandingly stable over that time, and yet the style and idioms of Java programs have changed a great deal. I find this especially interesting because I grew up, professionally speaking, in a world in which we repeatedly saw that when a programming style became unpopular, languages associated with it would die, or at least wither to insignificance. That doesn't hold any longer; languages of the 1990s—including Java—are still popular 30 years later. And that presents a major challenge: how to ensure that a language with its roots in a different period can still feel appropriate to use in a modern environment.

Perhaps the biggest single change has been the huge increase in processor speed and power relative to memory. This has meant that multiprocessor systems with multiple cache levels have become commonplace, and memory latency effects, especially cache behavior, often dominate program performance. So spatial locality has become a prime requirement for good performance, challenging the assumption originally underlying Java's automatic memory management philosophy: that the physical location of objects in memory isn't of primary importance. Continued evolution of JVM and garbage collector implementations have helped to protect application programmers from the full consequences of this, but it still means that every use of linked data structures comes under careful scrutiny. The first edition of this book contained a section—one that hasn't aged well—advocating the deprecation of arrays as data structures! As a sign of Java's continuing adaptation, one goal of [Project Valhalla](#) is to address this problem by flattening objects into cache-friendly value objects.

This second edition combines the work of three people. Phil's influence on [Part I](#) remains very strong. But many of the revisions to it, and many of the new ideas in [Part II](#), are the result of long discussions with Stuart Marks, nominally technical editor on this edition but in many ways an equal collaborator. His ideas, encouragement, support, and good judgment have been indispensable to the project. Among the many excellent people in the Java community that I have been fortunate to have collaborated with and to have learned from, I can't think of anyone better equipped to lead the development of the Java Collections Framework into the future. I'm sure that it will still be going strong in another 20 years!

—Maurice Naftalin,
Edinburgh
February 2025

Bibliography

- Bloch, Joshua. 2017. *Effective Java*. 3rd ed. Boston: Addison-Wesley.
- Bracha, Gilad, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. “Making the Future Safe for the Past: Adding Genericity to the Java Programming Language.” In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 183–200. New York: ACM. <https://doi.org/10.1145/286936.286957>.
- Calaprice, Alice, ed. 2011. *The Ultimate Quotable Einstein*. Princeton, NJ: Princeton University Press.
- Farach-Colton, Martin, Andrew Krapivin, and William Kuszmaul. 2025. “Optimal Bounds for Open Addressing Without Reordering.” *arXiv*, <https://arxiv.org/abs/2501.02305v2>.
- Fowler, Martin. 2003. “Anemic Domain Model.” <https://martinfowler.com/bliki/AnemicDomainModel.html>.
- Fuller, Thomas. 1732. *Gnomologia: Adages and Proverbs; Wise Sentences and Witty Sayings, Ancient and Modern, Foreign and British*. London: Barker, Bettesworth and Hitch.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Gay, David, and Bjarne Steensgaard. 2000. “Fast Escape Analysis and Stack Allocation for Object-Based Programs.” In *Compiler Construction*. CC 2000, edited by D. A. Watt. Lecture Notes in Computer Science 1781. Springer. https://doi.org/10.1007/3-540-46423-9_6.
- Goetz, Brian, with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Boston: Addison-Wesley.

Goetz, Brian. 2020. “Background: How We Got the Generics We Have (Or, How I Learned to Stop Worrying and Love Erasure).” <https://openjdk.org/projects/valhalla/design-notes/in-defense-of-erasure>.

Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2023. *The Java Language Specification, Java SE 21 Edition*. Oracle, Inc. <https://docs.oracle.com/javase/specs/jls/se21/html/index.html>.

Hoare, Tony. 2009. “Null References: The Billion Dollar Mistake.” *InfoQ*. <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>.

Igarashi, Atsushi, and Mirko Viroli. 2006. “Variant Parametric Types: A Flexible Subtyping Scheme for Generics.” *ACM Transactions on Programming Languages and Systems*, 28 (5): 795–847. <https://doi.org/10.1145/1152649.1152650>.

Jones, C. B. 1983. “Tentative Steps Toward a Development Method for Interfering Programs.” *ACM Transactions on Programming Languages and Systems*, 5 (4): 596–619. <https://doi.org/10.1145/69575.69577>.

Kabutz, Heinz. 2001. “Circular Array List.” *The JavaSpecialists’ Newsletter*, 27. <https://www.javaspecialists.eu/archive/Issue027-Circular-Array-List.html>.

Kabutz, Heinz. 2004. “References.” *The JavaSpecialists’ Newsletter*, 98. <https://www.javaspecialists.eu/archive/Issue098-References.html>.

Knuth, Donald E. 1974. “Structured Programming with *go to* Statements.” *ACM Computing Surveys*, 6 (4): 261–301. <https://doi.org/10.1145/356635.356640>.

Knuth, Donald E. 1998. *The Art of Computer Programming: Sorting and Searching*, vol. 3, 2nd ed. Reading, MA: Addison-Wesley.

Lea, Doug. 1999. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Boston: Addison-Wesley.

Liskov, Barbara. 1987. Keynote Address—Data Abstraction and Hierarchy. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA ’87)*, 17–34. <https://dl.acm.org/doi/10.1145/62138.62141>.

Martin, Robert C. 2002. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall.

Naftalin, Maurice. n.d. “Maurice Naftalin’s Lambda FAQ.” <https://www.lambdafaq.org>.

Naftalin, Maurice. 2015. *Mastering Lambdas: Java Programming in a Multicore World*. New York: McGraw-Hill.

Newton, Isaac. 1675. “Letter to Robert Hooke.” In *The Correspondence of Isaac Newton*, edited by H.W. Turnbull, vol. 1, 116–118. Cambridge: Cambridge University Press, 1959.

Odersky, Martin, Enno Runne, and Philip Wadler. 2000. “Two Ways to Bake Your Pizza—Translating Parameterized Types into Java.” Technical Report CIS-97-016, University of South Australia. <https://pizzacompiler.sourceforge.net/doc/pizza-translation.pdf>.

Sedgewick, Robert, and Kevin Wayne. 2011. *Algorithms*. 4th ed. Boston: Addison-Wesley.

Simpson, George Gaylord. 1945. *The Principles of Classification and a Classification of Mammals*. New York: American Museum of Natural History.

Smith, Brian, and Ross Cartwright. 2008. “Java Type Inference Is Broken: Can We Fix It?” In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA ’08)*, 505–524. <https://doi.org/10.1145/1449764.1449804>.

Steele, Julie, and Noah Iliinsky, eds. 2010. *Beautiful Visualization: Looking at Data through the Eyes of Experts*. Sebastopol, CA: O’Reilly Media.

Torgersen, Mads, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. 2004. “Adding Wildcards to the Java Programming Language.” In *Proceedings of the 2004 ACM Symposium on Applied Computing*, 1289–1296. <https://doi.org/10.1145/967900.968162>.

Valeev, Tagir. 2024. *100 Java Mistakes and How to Avoid Them*. Shelter Island, NY: Manning Publications.

Index

A

abstract classes, customization and, 305-309
AbstractList, 86
aliasing, 293
anemic domain models, 287-289
annotations, 101
 erasure, 5
 scope, 115
API design, 295-296
APIs (application programming interfaces)
 term use, 287
 using enumeration, 284
Appendable interface, 53-55
ArrayDeque, 209
ArrayList
 defining, 86-87
 List interface, 241-243
arrays, 139
 circular, 219
 creating, 79-81
 from arrays, 83-86
 by reflection, 83-86
 varargs, 91-93
 erasure, 6
 generic, 92
 of nonreifiable type, 89
 reification, 70
 subtyping
 covariant, 24
 varargs, 8-9
 versus collection classes, 26
atomic operations, 159
 optimistic style, 252-253
 pessimistic style, 250-252

autoboxing, 10

B

binary trees, 200-201
 priority heaps, 212
BlockingQueue, 215
 adding elements, 215
 ArrayBlockingQueue, 219-221
 circular array implementation, 219
 DelayQueue, 221-223
 element delay time, 221-223
 LinkedBlockingQueue, 219
 methods, 217-219
 PriorityBlockingQueue, 221
 querying contents, 216
 removing elements, 215-216
 retrieving contents, 216
 SynchronousQueue, 223
 TransferQueue, 223-224
bound type variables, 39
bounded wildcards, 16, 28-29
 bounded below, 18
bounds
 multiple, 53-55
 intersection types, 54
 queues, 207
 recursive, 40
boxing, 10
bridge methods (see bridges)
bridges, 55-57

C

C++ templates, 6-7
cache memory, 144

cast-iron guarantee, 5, 74
casts, 71-75
 inserted by erasure, 5, 83
 through raw types, 121
 to reifiable types, 93
unchecked, 73, 75-76, 114
 reflection library, 104-106
check-then-act, 303
circular arrays, 219
Class class, 99
 equality test, 100
 parameterization, 84-86
 primitive types, 104
 reifiable types, 102-103
 Type interface, 108-111
 type parameter, 101-102
 type tokens, 84-86, 123
 type-safe library, 105-106
class literals, 85, 100, 104
 restricted to raw types, 103
client-side locking, 303
clients, term use, 287
Closeable interface, 53-55
code bloat, 6
Collection interface, 128
 contents
 processing availability, 165-167
 querying, 165
 elements
 adding, 163
 removing, 164
 implementation, 175
List collection, 129-130
Queue, 131-132
Sequenced Collection, 131
sequenced collections, 132
Set collection, 128-129
collection types
 externally ordered, 132
 internally ordered, 132
List, 129-130
Map, 130-131
Queue, 131-132
sequenced, 132
SequencedMap, 131
Set, 128-129
collections
 and lambdas, 152
 and streams, 152
array-based, 139
checked wrappers, 282
concurrent modification, 325-328
constructors, 176
contents
 making available, 172-175
 querying, 172
contracts, 149-152
conversion to arrays, 78
customization, abstract classes, 305-309
empty, streams and, 40
finding the maximum of, 39
hash-based, 140
immutability, 147-148
linked structure-based, 139
locking operations, 187
ownership, 289-293
performance, 142
 instruction count, 145-147
 memory, 143-144
 O-notation, 145-147
streams
 parallel, 153-154
 synchronized, 281
 unmodifiable, 148, 281
views, 141-142
 views of maps, 249
Collections class
 addAll, 283
 asLifoQueue, 283
 disjoint, 283
 elements, ordering, 275-276
 enumeration, 284
 extreme values, 277
 factory methods, 279-280
 frequency, 284
 lists, 275-279
 modifying, 275-278
 querying, 278-279
 newSequenceSetFromMap, 284
 reverseOrder, 285
 single member objects, 280
wrappers
 checked wrappers, 282
 synchronization, 281
 unmodifiability, 281
Collections Framework, 128
 (see also specific collections and interfaces)
anemic domain models, 287-289

API design, 295-296
collection ownership, 289-293
customization, abstract classes, 305-309
desirable characteristics, 312-313
immutability, 293-295
legacy implementations
 synchronized wrapper collections, 302-304
Comparable, 35
 comparisons
 permitting, 41-45
 prohibiting, 41-45
contract, 37
correspondence with Comparator, 46
equals and, 37-38
 finding a maximum element, 39
Comparator
 abstract compare method, 45
 default methods, 47
 comparing, 47-50
 naturalOrder, 35
 reversed, 49
 thenComparing, 49
comparisons
 antisymmetric, 37
 congruence, 37
 versus equality, 36
integer with string, 36
integral values, 38-39
nonsensical, 36
null argument, 36
permitting, 41-45
prohibiting, 41-45
transitive, 37
component types
 arrays, 78-79
 refication, 70
ConcurrentHashMap, 271
ConcurrentMap, 270-271
 ConcurrentHashMap, 271
ConcurrentNavigableMap, 272
 ConcurrentSkipListMap, 272
ConcurrentSkipListSet interface, 203-204
constructors
 ArrayBlockingQueue, 219
 ConcurrentHashMap, 271
 EnumMap, 261-262
 HashSet, 186
 LinkedHashMap, 265-267

of collections, 176
of generic classes, 59-60
SynchronousQueue, 223
TreeMap, 270
TreeSet, 201-202
contracts, 149-152, 317-318
 rely/guarantee, 317
contravariant subtyping, 25
CopyOnWriteArrayList, 243-244
CopyOnWriteArraySet class, 186-187
covariant method overriding, 57-58
covariant subtyping, 24

D

declarations
 classes, 59-60
 constructors, 59-60
 methods, 41
 nested classes, 63-65
deep immutability, 148
defensive copying, 292
Deque, 134, 224
 ArrayDeque, 227
 BlockingDeque, 229-230
 collection-like methods, 225
 inherited methods, 226
 LinkedList, 228
 queue-like methods, 226
design choices
 collections, 311-328
 erasure, 94-95
 generic array declarations, 95
 unbounded wildcards, 95
 varargs, 96

E

empty collections, 279
empty collections, handling, 40
encapsulation, 291
encounter order, 177
Enum, 51
EnumMap class, 261-262
enums (enumerated types), 50-53
EnumSet class, 187-189
equals, consistency, 37, 322-324
equivalence relations, sets, 182-183, 323-324
erasure, 5, 65-67
 bridges, inserted by, 55-57
 casts, inserted by, 5, 83

design, 94-95
versus expansion, 6
parameterized types, 6
Principle of Truth in Advertising, 82
 reification and, 95
exception handling, 76-77
expansion
 templates, 94
 versus erasure, 6

F

fail-fast iterators, 138, 158
fail-fast policies, 327
FIFO (first in, first out), 207
foreach statement, 127, 136-137
fragile base class problem, 305

G

generic array creation error, 80
generic array types, 88-90, 122-123
generic classes, 4
generic methods, 7
 method calls, 32
generics for reflection, 100-102
generics versus C++ templates, 6-7
Get and Put Principle, 20-24, 30, 277
giants, shoulders, xix

H

hardware design, memory and, 143
hash table collisions, handling, 184-185
 IdentityHashMap, 259
hash tables, 140, 183-185, 259-260
 collisions, 184-185
 hash codes, 183
 hash functions, 183
 iteration, 185
HashMap class, 257
HashSet class
 constructors, 186
 hash functions, 183
 hash tables, 183
 collisions, 184-185
 hash codes, 183
 iteration, 185
heap pollution, 91
hierarchies
 memory, 143

of parallel types, 314

I

IdentityHashMap, 259-261
immutability, 147-148, 293-295, 312
 type systems, 315
inheritance, 305-307
 use to create reifiable types, 118-119
inner classes, type parameters, 64
instance tests, 71-75
instanceof, 71-73
instruction count, 145-147
integral values, comparing, 38-39
interfaces (see specific interfaces)
intersection types, 9, 54
invariant subtyping, 25
invariants, 155
Iterable interface, 135-138
iteration, 135-138, 172, 185
iterators
 collections, 40
 fail-fast iterators, 138
 invalidation, 327
 List interface, 235-236, 240
 robust iterators, 326
 snapshot iterators, 161
 weakly consistent iterators, 161

J

Java Collections Framework (see Collections Framework)
JVM (Java Virtual Machine), 95
 generics implementation, 5

L

lambdas, 152
libraries
 generic versus legacy, 6
 reflection, 104-106
 term use, 287
linked lists
 linear, 139
 skip lists, 203-204
LinkedHashMap, 265-267
LinkedHashSet class, 192-193
LinkedList, 243, 304-305
List interface, 4, 129-130, 233
 adding tasks, 239

ArrayList, 241-243
CopyOnWriteArrayList, 243-244
factory methods, 236
implementation comparison, 245-246
iteration, 235-236, 240
LinkedList, 243
positional access, 234
properties, 244
searches, 234
SequencedCollection inherited methods, 236
UnmodifiableList, 244-245
view generation, 234-235
views, 240
load balancing, 229
locality properties, 142
lock striping, 270
locking operations, 157-158, 187
 client-side, 303
LRU (least recently used) caches, 265
lumping versus splitting, 318-319

M

Map interface, 130-131
 collection-like operations, 248-249
 compound operations, 250
 atomic operations, 250
 ConcurrentMap, 270-271
 ConcurrentHashMap, 271
 ConcurrentNavigableMap, 272
 ConcurrentSkipListMap, 272
 EnumMap, 261-262
 HashMap, 257
 IdentityHashMap, 259-261
 implementation comparison, 272-273
 iterable-like operations, 248
 LinkedHashMap, 265-267
 Map.Entry interface, 253
 methods, using, 254-255
 NavigableMap, 267
 closest matches, 269
 comparators, 268
 TreeMap, 270
 properties, 262
 SequencedMap interface
 adding entries, 264
 inspecting entries, 264
 removing entries, 264
 updating entries, 264

 view generating, 265
 UnmodifiableMap, 262-263
 view collections, 249
 WeakHashMap, 258-259
memory
 cache lines, 144
 hierarchies, 143
methods
 declarations, 41
 generic, 7
 generic method calls, 32
 signatures, 19
 static, 40
money, begetting, 83
MRU (most recently used), 266
multithreading, 155-156
 concurrency, 155
 concurrent collections, 159-161
 fail-fast iterators, 158
 invariants, 155
 locking operations, 187
 race conditions, 156
 synchronization, legacy collections, 157

N

natural ordering, 35, 46
 Comparable, 35
 Comparator, 47
NavigableMap, 38, 134, 267
 closest matches, 269
 Comparator, 268
 navigating in reverse order, 269
 range views, 268
 TreeMap, 270
NavigableSet, 38, 133, 193
 closest matches, 197
 Comparator, 195
 element inspection, 195
 element removal, 195
 extending SequencedSet, 199
 navigating in reverse order, 198
 range views, 196-197
 TreeSet, 199
nested classes, 63-65
nested wildcards, 31
nonsensical comparisons, 36
nulls, 319-322
 comparisons and, 36
 null-tolerant maps, 249

O

O-notation, 145-147
operators
 airport check-in operator, 207
 pattern match operator, 71
 type comparison, 71
overriding, covariant methods, 57-58

P

parallel streams, 153-154
parameterization, Class class, 84-86
parameterized types
 erasure, 6
 heap pollution, 91
 reifiable or not, 70
 specialized, reification, 118-119
pattern match operator, 71
PECS (producer-extends, consumer-super), 20
performance, 142
 instruction count, 145-147
 memory, 143
 cache lines, 144
 hierarchy, 143
 O-notation, 145-147
primitive types, 9-11
 reflection, 104
 reification, 70
Principle of Indecent Exposure, 70, 88-91
Principle of Truth in Advertising, 70, 81-83
priority heaps, 212
producer-extends, consumer-super (PECS), 20
properties
 of compareTo, 37
 of UnmodifiableList, 244
 of UnmodifiableMap, 262
 of UnmodifiableSet, 189
 related to type, 313
proxies, protection proxies, 280

Q

Queue interface, 131-132, 207
 adding elements, 208
 ArrayDeque, 209
 BlockingQueue, 215
 adding elements, 215
 ArrayBlockingQueue, 219-221
 DelayQueue, 221-223
 LinkedBlockingQueue, 219

methods, 217-219
PriorityBlockingQueue, 221
querying contents, 216
removing elements, 215-216
retrieving contents, 216
SynchronousQueue, 223
TransferQueue, 223-224
ConcurrentLinkedQueue, 214
Deque, 224
 ArrayDeque, 227
 BlockingDeque, 229-230
 collection-like methods, 225
 inherited methods, 226
 LinkedList, 228
 queue-like methods, 226
implementation comparison, 230-232
methods, 209-211
PriorityQueue, 211-214
 retrieving elements, 209
queues, 207
 adding elements, 208, 215
 bounded, 207
 element delay time, 221-223
 FIFO (first in, first out), 207
 insertion/removal, 219
 producer/consumer, 218
 querying contents, 216
 removing elements, 215-216
 retrieving contents, 216
 retrieving elements, 209

R

range views, 169, 196-197
raw types, 60
 casting through, 121
 class literals, 103
 reifiable, 70
read-only types, 313
read/write types, 313
Readable interface, 53-55
readable types, 313
records, 297
 as composite keys, 299-301
 mutability, 301-302
 versus parallel lists, 297-299
recursive bounds, 40
recursive decomposition, parallel streams, 153
red-black trees, 201, 202
reference types versus primitive types, 9-11

reflection
generics for reflection, 99, 100-102
library
 handling reflected types, 108-111
 minimizing unchecked casts, 104-106
primitive types, 104
reflection for generics, 99, 107-108
reification, 102-103
type parameters, 101
unbounded wildcards, 101
reifiable types, 70
 and casts, 71-76, 93
 arrays, component types, 78-79, 88-91
 class literals, 103
 exception handling, 76-77
 instance tests, 71-76
 parameterized types, specialized, 118-119
 pattern match operator, 71
Principle of Truth in Advertising, 82
required, 93-94
 type comparison operator, 71
reification, 69
 alternative to erasure, 95
 array component types, 6
 reflection, 102-103
rely/guarantee contracts, 317
robust iterators, 326

S

semantics
 API, enforcing, 97
 application-level, 317-318
 versus syntax, 6
 weak, 326
semaphores, 218
SequencedCollection, 132-133, 177-178
 elements
 adding, 178
 inspecting, 178
 removing, 179
 reverse-ordered view, 179
 List interface methods, 236
SequencedMap interface, 131, 134
 adding entries, 264
 inspecting entries, 264
 removing entries, 264
 updating entries, 264
 view generating, 265
SequencedSet, 133

LinkedHashSet class, 192-193
NavigableSet, 199
Set interface, 128-129, 181, 204
CopyOnWriteArraySet class, 186-187
EnumSet class, 187-189
HashSet class, 183-186
SequencedSet, 192
 ConcurrentSkipListSet, 203-204
 LinkedHashSet class, 192-193
 NavigableSet, 193-200
 TreeSet, 200-202
UnmodifiableSet, 189-190
 properties, 189
sets, 181
 element inspection, 195
 element removal, 195
 enums, 187-189
 equivalence relations, 182-183
 hash tables, 184-185
 linked, 192-193
 map views, 191
 navigable sets
 range views, 196-197
 reverse order navigation, 198
 trees, 200-202
shallow immutability (see unmodifiability)
signatures, 19
single-use type variables, 119
skip lists, 203-204
snapshot iterators, 161, 326
spatial locality, 144
specialized types, reification, 118-119
Spliterator, 154
splitting versus lumping, 318-319
static members of classes, 60-62
static methods on interfaces, 40
static typing, alternatives to, 316-318
Stream API, 153, 168
streams, 152
 empty collections, handling, 40
 parallel, 153-154
Substitution Principle, subtyping, 14-15
subtypes
 read-only types, 313
subtyping, 13, 18
 contravariant, 25
 covariant, 24
 implementing unmodifiability, 313-314
 invariant, 25

- limitations, 314-316
Substitution Principle, 14-15
supertypes, 14, 18, 32
 readable types, 313
suppressing warnings, 115
synchronization, 157, 281
 fail-fast iterators, 158
 legacy collections, 157
 wrapper collections, 302-304
syntax versus semantics, 6
- T**
- template metaprogramming in C++, 7
templates, expansion, 94
test-then-act, 303
threads, 155-156
 time slicing, 155
 concurrent collections, 159-161
 invariants, 155
 race conditions, 156
 synchronization, fail-fast iterators, 158
 synchronization, legacy collections, 157
time slicing, 155
time-of-check to time-of-use (TOCTOU), 303
TOCTOU (time-of-check to time-of-use), 303
tombstone marker, 260
TreeMap, 270
TreeSet, 200-202
type comparison operator, 71
Type interface, 108-111
type parameters, 3, 4
 consistency checks, 4
 constructors, 59
 inner classes, 64
 reflection, 101
type safety, untrusted code and, 115-117
type tokens, 83-86, 123-124
type variables, 4
 bound, 39
 recursive, 40
 extends keyword, 39
 generic methods, 7
 single-use, 119
types
 autoboxing, 10
 boxing, 10
 collection types
 List, 129-130
 Map, 130-131
Queue, 131-132
sequenced, 132
Set, 128-129
enumerated, 50-53
erasure, 5, 65-67
 (see also erasure)
generic array types, 122-123
generic types, reflecting, 108-111
generics for reflection, 100
intersection types, 54
parallel hierarchies, 314
parameterized, 91
 erasure, 6
 reifiable, 70
 specialized, 118-119
primitive types, 9-11
 reflection, 104
 reifiable, 71
raw types, 60-62, 70, 121
read-only, 313
read/write, 313
readable, 313
reference types, 9-11
reifiable, 70-71
static typing, alternatives to, 316-318
subtypes, 13, 18
supertypes, 14, 18, 32
unboxing, 10
- U**
- unbounded wildcards, 16, 27-28, 95
 reflection, 101
unboxing, 10
unchecked casts, 73, 114
unchecked warnings, 5, 113-115
unmodifiability, 148, 281
UnmodifiableList, 244-245
 properties, 244
UnmodifiableMap, 262-263
 properties, 262
UnmodifiableSet, 189-190
- V**
- varargs, 8-9
 array creation, 91-93
 design, 96
 heap pollution, 91
 unmodifiable lists, 245
 unmodifiable maps, 263

unmodifiable sets, 190
variable arity parameters (see varargs)
variables
 atomic, 159
 type variables, 4
 bound, 39, 40
 extracting, 109
 generic method, 7
 single-use, 119
 wildcards, 17
views, 141-142
 List interface, 234-235, 240
 NavigableMap, 267
 NavigableSet interface, 196-197

W

wait-free algorithms, 214
warnings, unchecked, 5, 73-74, 113-115
WeakHashMap, 258-259
wildcards, 16
 bounded, 16, 28-29
 bounded above, 17
 bounded below, 18

capture, 29-30
contravariant subtyping, 24
covariant subtyping, 25
Get and Put Principle, 20-24, 30
helper methods to capture, 120-121
instance creation
 nested, permitted, 30-31
 top-level, prohibited, 30-31
PECS (producer-extends, consumer-super), 20-24
restrictions on, 30
with super, 18-19
unbounded, 16, 27-28, 95
 reflection, 101
with extends, 16-18

word cloud generators, 128
work stealing, 229
wrappers
 Collections class
 checked wrappers, 282
 synchronization, 281, 302-304
 unmodifiability, 281

About the Authors

Maurice Naftalin has over 50 years' experience in IT as a developer, designer, architect, manager, teacher, and author. He has been using and teaching Java since JDK 1.0. He is a Java Champion, the author of *Mastering Lambdas* (2015), and a frequent presenter at conferences worldwide. He disorganizes the annual Java unconference JAlba.

Philip Wadler is professor of theoretical computer science at the University of Edinburgh, Scotland, where his research focuses on functional and logic programming, and the design of programming languages. He was a principal designer of the Haskell programming language, and a codesigner of GJ, work that became the basis for Java generics. Professor Wadler received his PhD in computer science from Carnegie-Mellon.

Stuart Marks is the JDK Core Libraries project lead in the Java Platform Group at Oracle. He is currently the maintainer of the Java Collections Framework. As his alter ego “Dr Deprecator,” he also works on Java’s deprecation mechanism. He has over 30 years of software platform product development experience in the areas of window systems, interactive graphics, and mobile and embedded systems. He holds a master’s degree in computer science from Stanford University.

Colophon

The animal on the cover of *Java Generics and Collections* is an alligator. Alligators are found only in southern parts of the US and in China. They are rare in China, native only to the Yangtze River Basin. Alligators generally cannot tolerate salt water and therefore live in freshwater ponds, swamps, and the like.

When first born, alligators are tiny, measuring only about six inches. However, they grow extremely fast in the first years of life—a foot each year. A fully grown female is usually around 9 feet long and weighs between 150 and 200 pounds, while an adult male typically reaches 11 feet and weighs about 350 to 400 pounds. The largest known alligator on record, found in Louisiana in the early 1900s, measured 19 feet, 2 inches. A key identifying characteristic of an alligator's appearance is its short, broad snout. An adult alligator's skin is a gray-black color, which turns dark black when wet, and it has a white underbelly. Young alligators have yellow and white stripes across their backs. The shape of the snout and skin color provide physical characteristics that differentiate alligators from crocodiles, which have long, thin snouts and are a tan color.

Alligators are mainly nocturnal and do most of their hunting and feeding after the sun sets. They are carnivores and eat a large variety of food, such as turtles, fish, frogs, birds, snakes, small mammals, and even smaller alligators. However, once an alligator grows into adulthood, it really faces no threats—other than humans.

The cover illustration is by José Marzan Jr., based on an antique line engraving from *The Animal Kingdom Illustrated*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

**Learn from experts.
Become one yourself.**

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation

Try the O'Reilly learning platform free for 10 days.

