



Java SE 8

for the Really
Impatient



Cay S. Horstmann

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Java SE 8 for the Really Impatient

This page intentionally left blank

Java SE 8 for the Really Impatient

Cay S. Horstmann

◆Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

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

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

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Cataloging-in-Publication Data is on file with the Library of Congress.

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-92776-7

ISBN-10: 0-321-92776-1

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing, January 2014

*To Greg Doench, my editor for two decades, whose patience, kindness,
and good judgment I greatly admire*

This page intentionally left blank

Contents

Preface	xiii
About the Author	xv

1	LAMBDA EXPRESSIONS	1
1.1	Why Lambdas?	2
1.2	The Syntax of Lambda Expressions	4
1.3	Functional Interfaces	6
1.4	Method References	8
1.5	Constructor References	9
1.6	Variable Scope	10
1.7	Default Methods	14
1.8	Static Methods in Interfaces	16
	Exercises	18
2	THE STREAM API	21
2.1	From Iteration to Stream Operations	22
2.2	Stream Creation	24
2.3	The <code>filter</code> , <code>map</code> , and <code>flatMap</code> Methods	25
2.4	Extracting Substreams and Combining Streams	26

2.5	Stateful Transformations	27
2.6	Simple Reductions	28
2.7	The Optional Type	29
2.7.1	Working with Optional Values	29
2.7.2	Creating Optional Values	30
2.7.3	Composing Optional Value Functions with flatMap	30
2.8	Reduction Operations	31
2.9	Collecting Results	33
2.10	Collecting into Maps	34
2.11	Grouping and Partitioning	36
2.12	Primitive Type Streams	39
2.13	Parallel Streams	40
2.14	Functional Interfaces	42
	Exercises	44
3	PROGRAMMING WITH LAMBDA	47
3.1	Deferred Execution	48
3.2	Parameters of Lambda Expressions	49
3.3	Choosing a Functional Interface	50
3.4	Returning Functions	53
3.5	Composition	54
3.6	Laziness	56
3.7	Parallelizing Operations	57
3.8	Dealing with Exceptions	58
3.9	Lambdas and Generics	61
3.10	Monadic Operations	63
	Exercises	64
4	JAVAFX	69
4.1	A Brief History of Java GUI Programming	70
4.2	Hello, JavaFX!	71
4.3	Event Handling	72
4.4	JavaFX Properties	73
4.5	Bindings	75

4.6	Layout	80
4.7	FXML	86
4.8	CSS	90
4.9	Animations and Special Effects	91
4.10	Fancy Controls	94
	Exercises	97
5	THE NEW DATE AND TIME API	101
5.1	The Time Line	102
5.2	Local Dates	104
5.3	Date Adjusters	107
5.4	Local Time	108
5.5	Zoned Time	109
5.6	Formatting and Parsing	112
5.7	Interoperating with Legacy Code	115
	Exercises	116
6	CONCURRENCY ENHANCEMENTS	119
6.1	Atomic Values	120
6.2	ConcurrentHashMap Improvements	123
	6.2.1 Updating Values	124
	6.2.2 Bulk Operations	126
	6.2.3 Set Views	128
6.3	Parallel Array Operations	128
6.4	Completable Futures	130
	6.4.1 Futures	130
	6.4.2 Composing Futures	130
	6.4.3 The Composition Pipeline	131
	6.4.4 Composing Asynchronous Operations	132
	Exercises	134
7	THE NASHORN JAVASCRIPT ENGINE	137
7.1	Running Nashorn from the Command Line	138
7.2	Running Nashorn from Java	139

7.3	Invoking Methods	140
7.4	Constructing Objects	141
7.5	Strings	142
7.6	Numbers	143
7.7	Working with Arrays	144
7.8	Lists and Maps	145
7.9	Lambdas	146
7.10	Extending Java Classes and Implementing Java Interfaces	146
7.11	Exceptions	148
7.12	Shell Scripting	148
	7.12.1 Executing Shell Commands	149
	7.12.2 String Interpolation	150
	7.12.3 Script Inputs	151
7.13	Nashorn and JavaFX	152
	Exercises	154

8 MISCELLANEOUS GOODIES 157

8.1	Strings	158
8.2	Number Classes	158
8.3	New Mathematical Functions	159
8.4	Collections	160
	8.4.1 Methods Added to Collection Classes	160
	8.4.2 Comparators	161
	8.4.3 The Collections Class	162
8.5	Working with Files	163
	8.5.1 Streams of Lines	163
	8.5.2 Streams of Directory Entries	165
	8.5.3 Base64 Encoding	166
8.6	Annotations	167
	8.6.1 Repeated Annotations	167
	8.6.2 Type Use Annotations	169
	8.6.3 Method Parameter Reflection	170
8.7	Miscellaneous Minor Changes	171
	8.7.1 Null Checks	171

- 8.7.2 Lazy Messages 171
- 8.7.3 Regular Expressions 172
- 8.7.4 Locales 172
- 8.7.5 JDBC 174
- Exercises 174

9 JAVA 7 FEATURES THAT YOU MAY HAVE MISSED 179

- 9.1 Exception Handling Changes 180
 - 9.1.1 The try-with-resources Statement 180
 - 9.1.2 Suppressed Exceptions 181
 - 9.1.3 Catching Multiple Exceptions 182
 - 9.1.4 Easier Exception Handling for Reflective Methods 183
- 9.2 Working with Files 183
 - 9.2.1 Paths 184
 - 9.2.2 Reading and Writing Files 185
 - 9.2.3 Creating Files and Directories 186
 - 9.2.4 Copying, Moving, and Deleting Files 187
- 9.3 Implementing the equals, hashCode, and compareTo Methods 188
 - 9.3.1 Null-safe Equality Testing 188
 - 9.3.2 Computing Hash Codes 189
 - 9.3.3 Comparing Numeric Types 189
- 9.4 Security Requirements 190
- 9.5 Miscellaneous Changes 193
 - 9.5.1 Converting Strings to Numbers 193
 - 9.5.2 The Global Logger 193
 - 9.5.3 Null Checks 194
 - 9.5.4 ProcessBuilder 194
 - 9.5.5 URLClassLoader 195
 - 9.5.6 BitSet 195
- Exercises 196

Index 199

This page intentionally left blank

Preface

This book gives a concise introduction to the many new features of Java 8 (and a few features of Java 7 that haven't received much attention) for programmers who are already familiar with Java.

This book is written in the “impatient” style that I first tried out in a book called *Scala for the Impatient*. In that book, I wanted to quickly cut to the chase without lecturing the reader about the superiority of one paradigm over another. I presented information in small chunks organized to help you quickly retrieve it when needed. The approach was a big success in the Scala community, and I am employing it again in this book.

With Java 8, the Java programming language and library receive a major refresh. Lambda expressions make it possible to write “snippets of computations” in a concise way, so that you can pass them to other code. The recipient can choose to execute your computation when appropriate and as often as appropriate. This has a profound impact on building libraries.

In particular, working with collections has completely changed. Instead of specifying how to compute a result (“traverse from the beginning to the end, and if an element matches a condition, compute a value from it, and add that value to a sum”), you specify what you want (“give me the sum of all elements that match a condition”). The library is then able to reorder the computation—for example, to take advantage of parallelism. Or, if you just want to have the first hundred matches, it can stop the computation without you having to maintain a counter.

The brand-new *stream* API of Java 8 puts this idea to work. In the first chapter, you learn all about the syntax of lambda expressions, and Chapter 2 gives a complete overview of streams. In Chapter 3, I provide you with tips on how to effectively design your own libraries with lambdas.

With Java 8, developers of client-side applications need to transition to the JavaFX API since Swing is now in “maintenance mode.” Chapter 4 gives a quick introduction to JavaFX for a programmer who needs to put together a graphical program—when a picture speaks louder than 1,000 strings.

Having waited for far too many years, programmers are finally able to use a well-designed date/time library. Chapter 5 covers the `java.time` API in detail.

Each version of Java brings enhancements in the concurrency API, and Java 8 is no exception. In Chapter 6, you learn about improvements in atomic counters, concurrent hash maps, parallel array operations, and composable futures.

Java 8 bundles Nashorn, a high-quality JavaScript implementation. In Chapter 7, you will see how to execute JavaScript on the Java Virtual Machine, and how to interoperate with Java code.

Chapter 8 collects miscellaneous smaller, but nevertheless useful, features of Java 8. Chapter 9 does the same for Java 7, focusing on improved exception handling, the “new I/O” enhancements for working with files and directories, and other library enhancements that you may have missed.

My thanks go, as always, to my editor Greg Doench, who had the idea of a short book that brings experienced programmers up to speed with Java 8. Dmitry Kirsanov and Alina Kirsanova once again turned an XHTML manuscript into an attractive book with amazing speed and attention to detail. I am grateful to the reviewers who spotted many embarrassing errors and gave excellent suggestions for improvement. They are: Gail Anderson, Paul Anderson, James Denvir, Trisha Gee, Brian Goetz (special thanks for the very thorough review), Marty Hall, Angelika Langer, Mark Lawrence, Stuart Marks, Attila Szegedi, and Jim Weaver.

I hope that you enjoy reading this concise introduction to the new features of Java 8, and that it will make you a more successful Java programmer. If you find errors or have suggestions for improvement, please visit <http://horstmann.com/java8> and leave a comment. On that page, you will also find a link to an archive file containing all code examples from the book.

Cay Horstmann

San Francisco, 2013

About the Author

Cay S. Horstmann is the author of *Scala for the Impatient* (Addison-Wesley, 2012), is principal author of *Core Java™, Volumes I and II, Ninth Edition* (Prentice Hall, 2013), and has written a dozen other books for professional programmers and computer science students. He is a professor of computer science at San Jose State University and is a Java Champion.

Programming with Lambdas

Topics in This Chapter

- 3.1 Deferred Execution — page 48
- 3.2 Parameters of Lambda Expressions — page 49
- 3.3 Choosing a Functional Interface — page 50
- 3.4 Returning Functions — page 53
- 3.5 Composition — page 54
- 3.6 Laziness — page 56
- 3.7 Parallelizing Operations — page 57
- 3.8 Dealing with Exceptions — page 58
- 3.9 Lambdas and Generics — page 61
- 3.10 Monadic Operations — page 63
- Exercises — page 64

Chapter

3

In the first two chapters, you saw the basic syntax and semantics of lambda expressions as well as the stream API that makes extensive use of them. In this chapter, you will learn how to create your own libraries that make use of lambda expressions and functional interfaces.

The key points of this chapter are:

- The main reason for using a lambda expression is to defer the execution of the code until an appropriate time.
- When a lambda expression is executed, make sure to provide any required data as inputs.
- Choose one of the existing functional interfaces if you can.
- It is often useful to write methods that return an instance of a functional interface.
- When you work with transformations, consider how you can compose them.
- To compose transformations lazily, you need to keep a list of all pending transformations and apply them in the end.
- If you need to apply a lambda many times, you often have a chance to split up the work into subtasks that execute concurrently.
- Think what should happen when you work with a lambda expression that throws an exception.

- When working with generic functional interfaces, use `?` super wildcards for argument types, `?` extends wildcards for return types.
- When working with generic types that can be transformed by functions, consider supplying `map` and `flatMap`.

3.1 Deferred Execution

The point of all lambdas is *deferred execution*. After all, if you wanted to execute some code right now, you'd do that, without wrapping it inside a lambda. There are many reasons for executing code later, such as

- Running the code in a separate thread
- Running the code multiple times
- Running the code at the right point in an algorithm (for example, the comparison operation in sorting)
- Running the code when something happens (a button was clicked, data has arrived, and so on)
- Running the code only when necessary

It is a good idea to think through what you want to achieve when you set out programming with lambdas.

Let us look at a simple example. Suppose you log an event:

```
logger.info("x: " + x + ", y: " + y);
```

What happens if the log level is set to suppress INFO messages? The message string is computed and passed to the `info` method, which then decides to throw it away. Wouldn't it be nicer if the string concatenation only happened when necessary?

Running code only when necessary is a use case for lambdas. The standard idiom is to wrap the code in a no-arg lambda:

```
() -> "x: " + x + ", y: " + y
```

Now we need to write a method that

1. Accepts the lambda
2. Checks whether it should be called
3. Calls it when necessary

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. We discuss the process of choosing an interface in more detail in Section 3.3, "Choosing a Functional Interface," on page 50. Here, a good choice is a `Supplier<String>`. The following method provides lazy logging:

```
public static void info(Logger logger, Supplier<String> message) {
    if (logger.isLoggable(Level.INFO))
        logger.info(message.get());
}
```

We use the `isLoggable` method of the `Logger` class to decide whether `INFO` messages should be logged. If so, we invoke the lambda by calling its abstract method, which happens to be called `get`.



NOTE: Deferring logging messages is such a good idea that the Java 8 library designers beat me to it. The `info` method, as well as the other logging methods, now have variants that accept a `Supplier<String>`. You can directly call `logger.info(() -> "x: " + x + ", y:" + y)`. However, see Exercise 1 for a potentially useful refinement.

3.2 Parameters of Lambda Expressions

When you ask your user to supply a comparator, it is pretty obvious that the comparator has two arguments—the values to be compared.

```
Arrays.sort(names,
    (s, t) -> Integer.compare(s.length(), t.length())); // Compare strings s and t
```

Now consider a different example. This method repeats an action multiple times:

```
public static void repeat(int n, IntConsumer action) {
    for (int i = 0; i < n; i++) action.accept(i);
}
```

Why an `IntConsumer` and not a `Runnable`? We tell the action in which iteration it occurs, which might be useful information. The action needs to capture that input in a parameter

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

Another example is an event handler

```
button.setOnAction(event -> action);
```

The event object carries information that the action may need.

In general, you want to design your algorithm so that it passes any required information as arguments. For example, when editing an image, it makes sense to have the user supply a function that computes the color for a pixel. Such a function might need to know not just the current color, but also where the pixel is in the image, or what the neighboring pixels are.

However, if these arguments are rarely needed, consider supplying a second version that doesn't force users into accepting unwanted arguments:

```
public static void repeat(int n, Runnable action) {
    for (int i = 0; i < n; i++) action.run();
}
```

This version can be called as

```
repeat(10, () -> System.out.println("Hello, World!"));
```

3.3 Choosing a Functional Interface

In most functional programming languages, function types are *structural*. To specify a function that maps two strings to an integer, you use a type that looks something like `Function2<String, String, Integer>` or `(String, String) -> int`. In Java, you instead declare the intent of the function, using a functional interface such as `Comparator<String>`. In the theory of programming languages this is called *nominal* typing.

Of course, there are many situations where you want to accept “any function” without particular semantics. There are a number of generic function types for that purpose (see Table 3–1), and it's a very good idea to use one of them when you can.

For example, suppose you write a method to process files that match a certain criterion. Should you use the descriptive `java.io.FileFilter` class or a `Predicate<File>`? I strongly recommend that you use the standard `Predicate<File>`. The only reason not to do so would be if you already have many useful methods producing `FileFilter` instances.



NOTE: Most of the standard functional interfaces have nonabstract methods for producing or combining functions. For example, `Predicate.isEqual(a)` is the same as `a::equals`, provided `a` is not null. And there are default methods `and`, `or`, `negate` for combining predicates. For example, `Predicate.isEqual(a).or(Predicate.isEqual(b))` is the same as `x -> a.equals(x) || b.equals(x)`.

Consider another example. We want to transform images, applying a `Color -> Color` function to each pixel. For example, the brightened image in Figure 3–1 is obtained by calling

```
Image brightenedImage = transform(image, Color::brighter);
```

Table 3–1 Common Functional Interfaces

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
Runnable	none	void	run	Runs an action without arguments or return value	
Supplier<T>	none	T	get	Supplies a value of type T	
Consumer<T>	T	void	accept	Consumes a value of type T	chain
BiConsumer<T, U>	T, U	void	accept	Consumes values of types T and U	chain
Function<T, R>	T	R	apply	A function with argument of type T	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	A function with arguments of types T and U	andThen
UnaryOperator<T>	T	T	apply	A unary operator on the type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	A binary operator on the type T	andThen
Predicate<T>	T	boolean	test	A Boolean-valued function	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	A Boolean-valued function with two arguments	and, or, negate



Figure 3-1 The original and transformed image

There is a standard functional interface for this purpose: `UnaryOperator<Color>`. That is a good choice, and there is no need to come up with a `ColorTransformer` interface. Here is the implementation of the `transform` method. Note the call to the `apply` method.

```
public static Image transform(Image in, UnaryOperator<Color> f) {
    int width = (int) in.getWidth();
    int height = (int) in.getHeight();
    WritableImage out = new WritableImage(width, height);
    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++)
            out.getPixelWriter().setColor(x, y,
                f.apply(in.getPixelReader().getColor(x, y)));
    return out;
}
```



NOTE: This method uses the `Color` and `Image` classes from JavaFX, not from `java.awt`. See Chapter 4 for more information on JavaFX.

Table 3-2 lists the 34 available specializations for primitive types `int`, `long`, and `double`. Use the specializations when you can to reduce autoboxing.

Sometimes, you need to supply your own functional interface because there is nothing in the standard library that works for you. Suppose you want to modify colors in an image, allowing users to specify a function $(int, int, Color) \rightarrow Color$ that computes a new color depending on the (x, y) location in the image. In that case, you can define your own interface:

Table 3–2 Functional Interfaces for Primitive Types

p, q is int, long, double;
 P, Q is Int, Long, Double

Functional Interface	Parameter Types	Return Type	Abstract Method Name
BooleanSupplier	none	boolean	getAsBoolean
P Supplier	none	p	getAs P
P Consumer	p	void	accept
Obj P Consumer< T >	T, p	void	accept
P Function< T >	p	T	apply
P To Q Function	p	q	applyAs Q
To P Function< T >	T	p	applyAs P
To P BiFunction< T, U >	T, U	p	applyAs P
P UnaryOperator	p	p	applyAs P
P BinaryOperator	p, p	p	applyAs P
P Predicate	p	boolean	test

```
@FunctionalInterface
public interface ColorTransformer {
    Color apply(int x, int y, Color colorAtXY);
}
```



NOTE: I called the abstract method `apply` because that is used for the majority of standard functional interfaces. Should you call the method `process` or `transform` or `getColor` instead? It doesn't matter much to users of the color manipulation code—they will usually supply a lambda expression. Sticking with the standard name simplifies the life of the implementor.

3.4 Returning Functions

In a functional programming language, functions are first-class citizens. Just like you can pass numbers to methods and have methods that produce numbers, you can have arguments and return values that are functions. This sounds abstract, but it is very useful in practice. Java is not quite a functional language because it uses functional interfaces, but the principle is the same. You have seen many

methods that accept functional interfaces. In this section, we consider methods whose return type is a functional interface.

Consider again image transformations. If you call

```
Image brightenedImage = transform(image, Color::brighter);
```

the image is brightened by a fixed amount. What if you want it even brighter, or not quite so bright? Could you supply the desired brightness as an additional parameter to transform?

```
Image brightenedImage = transform(image,
    (c, factor) -> c.deriveColor(0, 1, factor, 1), // Brighten c by factor
    1.2); // Use a factor of 1.2
```

One would have to overload transform:

```
public static <T> Image transform(Image in, BiFunction<Color, T> f, T arg)
```

That can be made to work (see Exercise 6), but what if one wants to supply two arguments? Or three? There is another way. We can make a method that returns the appropriate `UnaryOperator<Color>`, with the brightness set:

```
public static UnaryOperator<Color> brighten(double factor) {
    return c -> c.deriveColor(0, 1, factor, 1);
}
```

Then we can call

```
Image brightenedImage = transform(image, brighten(1.2));
```

The `brighten` method returns a function (or, technically, an instance of a functional interface). That function can be passed to another method (here, `transform`) that expects such an interface.

In general, don't be shy to write methods that produce functions. This is useful to customize the functions that you pass to methods with functional interfaces. For example, consider the `Arrays.sort` method with a `Comparator` argument. There are many ways of comparing values, and you can write a method that yields a comparator for your needs—see Exercise 7. Then you can call `Arrays.sort(values, comparatorGenerator(customization arguments))`.



NOTE: As you will see in Chapter 8, the `Comparator` class has several methods that yield or modify comparators.

3.5 Composition

A single-argument function transforms one value into another. If you have two such transformations, then doing one after the other is also a transformation.

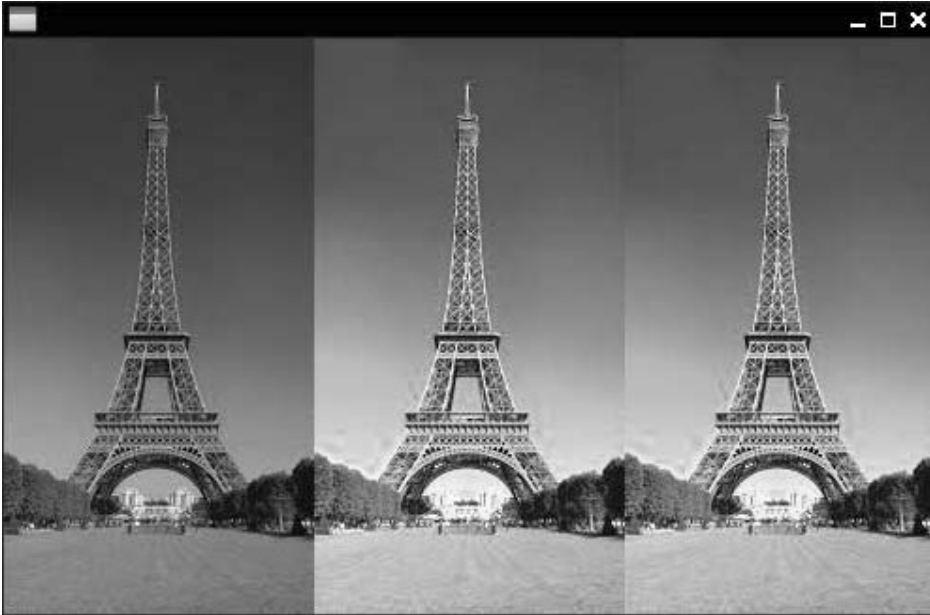


Figure 3–2 First, the image is brightened, and then grayscale is applied.

Consider image manipulation: Let's first brighten an image, then turn it to grayscale (see Figure 3–2).



NOTE: In the printed book, everything is in grayscale. Just run the program in the companion code to see the effect.

That is easy to do with our transform method:

```
Image image = new Image("eiffel-tower.jpg");  
Image image2 = transform(image, Color::brighter);  
Image finalImage = transform(image2, Color::grayscale);
```

But this is not very efficient. We need to make an intermediate image. For large images, that requires a considerable amount of storage. If we could compose the image operations and then apply the composite operation to each pixel, that would be better.

In this case, the image operations are instances of `UnaryOperator<Color>`. That type has a method `compose` that, for rather depressing reasons that are explored in Exercise 10, is not useful for us. But it is easy to roll our own:

```
public static <T> UnaryOperator<T> compose(UnaryOperator<T> op1,
    UnaryOperator<T> op2) {
    return t -> op2.apply(op1.apply(t));
}
```

Now we can call

```
Image finalImage = transform(image, compose(Color::brighter, Color::grayscale));
```

That is much better. Now the composed transformation is directly applied to each pixel, and there is no need for an intermediate image.

Generally, when you build a library where users can carry out one effect after another, it is a good idea to give library users the ability to compose these effects. See Exercise 11 for another example.

3.6 Laziness

In the preceding section, you saw how users of an image transformation method can precompose operations to avoid intermediate images. But why should they have to do that? Another approach is for the library to accumulate all operations and then fuse them. This is, of course, what the stream library does.

If you do lazy processing, your API needs to distinguish between intermediate operations, which accumulate the tasks to be done, and terminal operations which deliver the result. In the image processing example, we can make `transform` lazy, but then it needs to return another object that is not an `Image`. For example,

```
LatentImage latent = transform(image, Color::brighter);
```

A `LatentImage` can simply store the original image and a sequence of image operations.

```
public class LatentImage {
    private Image in;
    private List<UnaryOperator<Color>> pendingOperations;
    ...
}
```

This class also needs a `transform` method:

```
LatentImage transform(UnaryOperator<Color> f) {
    pendingOperations.add(f);
    return this;
}
```

To avoid duplicate `transform` methods, you can follow the approach of the stream library where an initial `stream()` operation is required to turn a collection into a

stream. Since we can't add a method to the `Image` class, we can provide a `LatentImage` constructor or a static factory method.

```
LatentImage latent = LatentImage.from(image)
    .transform(Color::brighter).transform(Color::grayscale);
```

You can only be lazy for so long. Eventually, the work needs to be done. We can provide a `toImage` method that applies all operations and returns the result:

```
Image finalImage = LatentImage.from(image)
    .transform(Color::brighter).transform(Color::grayscale)
    .toImage();
```

Here is the implementation of the method:

```
public Image toImage() {
    int width = (int) in.getWidth();
    int height = (int) in.getHeight();
    WritableImage out = new WritableImage(width, height);
    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            Color c = in.getPixelReader().getColor(x, y);
            for (UnaryOperator<Color> f : pendingOperations) c = f.apply(c);
            out.getPixelWriter().setColor(x, y, c);
        }
    return out;
}
```



CAUTION: In real life, implementing lazy operations is quite a bit harder. Usually you have a mixture of operations, and not all of them can be applied lazily. See Exercises 12 and 13.

3.7 Parallelizing Operations

When expressing operations as functional interfaces, the caller gives up control over the processing details. As long as the operations are applied so that the correct result is achieved, the caller has nothing to complain about. In particular, the library can make use of concurrency. For example, in image processing we can split the image into multiple strips and process each strip separately.

Here is a simple way of carrying out an image transformation in parallel. This code operates on `Color[][]` arrays instead of `Image` objects because the JavaFX `PixelWriter` is not threadsafe.

```
public static Color[][] parallelTransform(Color[][] in, UnaryOperator<Color> f) {
    int n = Runtime.getRuntime().availableProcessors();
    int height = in.length;
    int width = in[0].length;
    Color[][] out = new Color[height][width];
    try {
        ExecutorService pool = Executors.newCachedThreadPool();
        for (int i = 0; i < n; i++) {
            int fromY = i * height / n;
            int toY = (i + 1) * height / n;
            pool.submit(() -> {
                for (int x = 0; x < width; x++)
                    for (int y = fromY; y < toY; y++)
                        out[y][x] = f.apply(in[y][x]);
            });
        }
        pool.shutdown();
        pool.awaitTermination(1, TimeUnit.HOURS);
    }
    catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    return out;
}
```

This is, of course, just a proof of concept. Supporting image operations that combine multiple pixels would be a major challenge.

In general, when you are given an object of a functional interface and you need to invoke it many times, ask yourself whether you can take advantage of concurrency.

3.8 Dealing with Exceptions

When you write a method that accepts lambdas, you need to spend some thought on handling and reporting exceptions that may occur when the lambda expression is executed.

When an exception is thrown in a lambda expression, it is propagated to the caller. There is nothing special about executing lambda expressions, of course. They are simply method calls on some object that implements a functional interface. Often it is appropriate to let the expression bubble up to the caller.

Consider, for example:

```
public static void doInOrder(Runnable first, Runnable second) {
    first.run();
    second.run();
}
```

If `first.run()` throws an exception, then the `doInOrder` method is terminated, `second` is never run, and the caller gets to deal with the exception.

But now suppose we execute the tasks asynchronously.

```
public static void doInOrderAsync(Runnable first, Runnable second) {
    Thread t = new Thread() {
        public void run() {
            first.run();
            second.run();
        }
    };
    t.start();
}
```

If `first.run()` throws an exception, the thread is terminated, and `second` is never run. However, the `doInOrderAsync` returns right away and does the work in a separate thread, so it is not possible to have the method rethrow the exception. In this situation, it is a good idea to supply a handler:

```
public static void doInOrderAsync(Runnable first, Runnable second,
    Consumer<Throwable> handler) {
    Thread t = new Thread() {
        public void run() {
            try {
                first.run();
                second.run();
            } catch (Throwable t) {
                handler.accept(t);
            }
        }
    };
    t.start();
}
```

Now suppose that `first` produces a result that is consumed by `second`. We can still use the handler.

```

public static <T> void doInOrderAsync(Supplier<T> first, Consumer<T> second,
    Consumer<Throwable> handler) {
    Thread t = new Thread() {
        public void run() {
            try {
                T result = first.get();
                second.accept(result);
            } catch (Throwable t) {
                handler.accept(t);
            }
        }
    };
    t.start();
}

```

Alternatively, we could make `second` a `BiConsumer<T, Throwable>` and have it deal with the exception from `first`—see Exercise 16.

It is often inconvenient that methods in functional interfaces don't allow checked exceptions. Of course, your methods can accept functional interfaces whose methods allow checked exceptions, such as `Callable<T>` instead of `Supplier<T>`. A `Callable<T>` has a method that is declared as `T call() throws Exception`. If you want an equivalent for a `Consumer` or a `Function`, you have to create it yourself.

You sometimes see suggestions to “fix” this problem with a generic wrapper, like this:

```

public static <T> Supplier<T> unchecked(Callable<T> f) {
    return () -> {
        try {
            return f.call();
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
        catch (Throwable t) {
            throw t;
        }
    };
}

```

Then you can pass a

```

unchecked(() -> new String(Files.readAllBytes(
    Paths.get("/etc/passwd")), StandardCharsets.UTF_8))

```

to a `Supplier<String>`, even though the `readAllBytes` method throws an `IOException`.

That is a solution, but not a complete fix. For example, this method cannot generate a `Consumer<T>` or a `Function<T, U>`. You would need to implement a variation of `unchecked` for each functional interface.

3.9 Lambdas and Generics

Generally, lambdas work well with generic types. You have seen a number of examples where we wrote generic mechanisms, such as the `unchecked` method of the preceding section. There are just a couple of issues to keep in mind.

One of the unhappy consequences of type erasure is that you cannot construct a generic array at runtime. For example, the `toArray()` method of `Collection<T>` and `Stream<T>` cannot call `T[] result = new T[n]`. Therefore, these methods return `Object[]` arrays. In the past, the solution was to provide a second method that accepts an array. That array was either filled or used to create a new one via reflection. For example, `Collection<T>` has a method `toArray(T[] a)`. With lambdas, you have a new option, namely to pass the constructor. That is what you do with streams:

```
String[] result = words.toArray(String[]::new);
```

When you implement such a method, the constructor expression is an `IntFunction<T[]>`, since the size of the array is passed to the constructor. In your code, you call `T[] result = constr.apply(n)`.

In this regard, lambdas help you overcome a limitation of generic types. Unfortunately, in another common situation lambdas suffer from a different limitation. To understand the problem, recall the concept of type variance.

Suppose `Employee` is a subtype of `Person`. Is a `List<Employee>` a special case of a `List<Person>`? It seems that it should be. But actually, it would be unsound. Consider this code:

```
List<Employee> staff = ...;
List<Person> tenants = staff; // Not legal, but suppose it was
tenants.add(new Person("John Q. Public")); // Adds Person to staff!
```

Note that `staff` and `tenants` are references to the same list. To make this type error impossible, we must disallow the conversion from `List<Employee>` to `List<Person>`. We say that the type parameter `T` of `List<T>` is *invariant*.

If `List` was immutable, as it is in a functional programming language, then the problem would disappear, and one could have a *covariant* list. That is what is done in languages such as Scala. However, when generics were invented, Java had very few immutable generic classes, and the language designers instead embraced a different concept: use-site variance, or “wildcards.”

A method can decide to accept a `List<? extends Person>` if it only reads from the list. Then you can pass either a `List<Person>` or a `List<Employee>`. Or it can accept a

List<? super Employee> if it only writes to the list. It is okay to write employees into a List<Person>, so you can pass such a list. In general, reading is covariant (subtypes are okay) and writing is contravariant (supertypes are okay). Use-site variance is just right for mutable data structures. It gives each service the choice which variance, if any, is appropriate.

However, for function types, use-site variance is a hassle. A function type is *always* contravariant in its arguments and covariant in its return value. For example, if you have a Function<Person, Employee>, you can safely pass it on to someone who needs a Function<Employee, Person>. They will only call it with employees, whereas your function can handle any person. They will expect the function to return a person, and you give them something even better.

In Java, when you declare a generic functional interface, you can't specify that function arguments are always contravariant and return types always covariant. Instead, you have to repeat it for each use. For example, look at the javadoc for Stream<T>:

```
void forEach(Consumer<? super T> action)
Stream<T> filter(Predicate<? super T> predicate)
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

The general rule is that you use `super` for argument types, `extends` for return types. That way, you can pass a Consumer<Object> to forEach on a Stream<String>. If it is willing to consume any object, surely it can consume strings.

But the wildcards are not always there. Look at

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

Since `T` is the argument *and* return type of BinaryOperator, the type does not vary. In effect, the contravariance and covariance cancel each other out.

As the implementor of a method that accepts lambda expressions with generic types, you simply add `? super` to any argument type that is not also a return type, and `? extends` to any return type that is not also an argument type.

For example, consider the doInOrderAsync method of the preceding section. Instead of

```
public static <T> void doInOrderAsync(Supplier<T> first,
    Consumer<T> second, Consumer<Throwable> handler)
```

it should be

```
public static <T> void doInOrderAsync(Supplier<? extends T> first,
    Consumer<? super T> second, Consumer<? super Throwable> handler)
```

3.10 Monadic Operations

When you work with generic types, and with functions that yield values from these types, it is useful to supply methods that let you *compose* these functions—that is, carry out one after another. In this section, you will see a design pattern for providing such compositions.

Consider a generic type $G\langle T \rangle$ with one type parameter, such as $List\langle T \rangle$ (zero or more values of type T), $Optional\langle T \rangle$ (zero or one values of type T), or $Future\langle T \rangle$ (a value of type T that will be available in the future).

Also consider a function $T \rightarrow U$, or a $Function\langle T, U \rangle$ object.

It often makes sense to apply this function to a $G\langle T \rangle$ (that is, a $List\langle T \rangle$, $Optional\langle T \rangle$, $Future\langle T \rangle$, and so on). How this works exactly depends on the nature of the generic type G . For example, applying a function f to a $List$ with elements e_1, \dots, e_n means creating a list with elements $f(e_1), \dots, f(e_n)$.

Applying f to an $Optional\langle T \rangle$ containing v means creating an $Optional\langle U \rangle$ containing $f(v)$. But if f is applied to an empty $Optional\langle T \rangle$ without a value, the result is an empty $Optional\langle U \rangle$.

Applying f to a $Future\langle T \rangle$ simply means to apply it whenever it is available. The result is a $Future\langle U \rangle$.

By tradition, this operation is usually called `map`. There is a `map` method for `Stream` and `Optional`. The `CompletableFuture` class that we will discuss in Chapter 6 has an operation that does just what `map` should do, but it is called `thenApply`. There is no `map` for a plain `Future\langle V \rangle`, but it is not hard to supply one (see Exercise 21).

So far, that is a fairly straightforward idea. It gets more complex when you look at functions $T \rightarrow G\langle U \rangle$ instead of functions $T \rightarrow U$. For example, consider getting the web page for a URL. Since it takes some time to fetch the page, that is a function $URL \rightarrow Future\langle String \rangle$. Now suppose you have a $Future\langle URL \rangle$, a URL that will arrive sometime. Clearly it makes sense to map the function to that `Future`. Wait for the URL to arrive, then feed it to the function and wait for the string to arrive. This operation has traditionally been called `flatMap`.

The name `flatMap` comes from sets. Suppose you have a “many-valued” function—a function computing a set of possible answers. And then you have another such function. How can you compose these functions? If $f(x)$ is the set $\{y_1, \dots, y_n\}$, you apply g to each element, yielding $\{g(y_1), \dots, g(y_n)\}$. But each of the $g(y_i)$ is a *set*. You want to “flatten” the set of sets so that you get the set of all possible values of both functions.

There is a `flatMap` for `Optional<T>` as well. Given a function `T -> Optional<U>`, `flatMap` unwraps the value in the `Optional` and applies the function, except if either the source or target option was not present. It does exactly what the set-based `flatMap` would have done on sets with size 0 or 1.

Generally, when you design a type `G<T>` and a function `T -> U`, think whether it makes sense to define a `map` that yields a `G<U>`. Then, generalize to functions `T -> G<U>` and, if appropriate, provide `flatMap`.



NOTE: These operations are important in the theory of monads, but you don't need to know the theory to understand `map` and `flatMap`. The concept of mapping a function is both straightforward and useful, and the point of this section is to make you aware of it.

Exercises

1. Enhance the lazy logging technique by providing conditional logging. A typical call would be `logIf(Level.FINEST, () -> i == 10, () -> "a[10] = " + a[10])`. Don't evaluate the condition if the logger won't log the message.
2. When you use a `ReentrantLock`, you are required to lock and unlock with the idiom

```
myLock.lock();
try {
    some action
} finally {
    myLock.unlock();
}
```

Provide a method `withLock` so that one can call

```
withLock(myLock, () -> { some action })
```

3. Java 1.4 added assertions to the language, with an `assert` keyword. Why were assertions not supplied as a library feature? Could they be implemented as a library feature in Java 8?
4. How many functional interfaces with `Filter` in their name can you find in the Java API? Which ones add value over `Predicate<T>`?
5. Here is a concrete example of a `ColorTransformer`. We want to put a frame around an image, like this:



First, implement a variant of the `transform` method of Section 3.3, “Choosing a Functional Interface,” on page 50, with a `ColorTransformer` instead of an `UnaryOperator<Color>`. Then call it with an appropriate lambda expression to put a 10 pixel gray frame replacing the pixels on the border of an image.

6. Complete the method

```
public static <T> Image transform(Image in, BiFunction<Color, T> f, T arg)
```

from Section 3.4, “Returning Functions,” on page 53.

7. Write a method that generates a `Comparator<String>` that can be normal or reversed, case-sensitive or case-insensitive, space-sensitive or space-insensitive, or any combination thereof. Your method should return a lambda expression.
8. Generalize Exercise 5 by writing a static method that yields a `ColorTransformer` that adds a frame of arbitrary thickness and color to an image.
9. Write a method `lexicographicComparator(String... fieldNames)` that yields a comparator that compares the given fields in the given order. For example, a `lexicographicComparator("lastname", "firstname")` takes two objects and, using reflection, gets the values of the `lastname` field. If they are different, return the difference, otherwise move on to the `firstname` field. If all fields match, return `0`.
10. Why can't one call

```
UnaryOperator op = Color::brighter;
Image finalImage = transform(image, op.compose(Color::grayscale));
```

Look carefully at the return type of the `compose` method of `UnaryOperator<T>`. Why is it not appropriate for the `transform` method? What does that say about

the utility of structural and nominal types when it comes to function composition?

11. Implement static methods that can compose two `ColorTransformer` objects, and a static method that turns a `UnaryOperator<Color>` into a `ColorTransformer` that ignores the x - and y -coordinates. Then use these methods to add a gray frame to a brightened image. (See Exercise 5 for the gray frame.)
12. Enhance the `LatentImage` class in Section 3.6, “Laziness,” on page 56, so that it supports both `UnaryOperator<Color>` and `ColorTransformer`. Hint: Adapt the former to the latter.
13. Convolution filters such as blur or edge detection compute a pixel from neighboring pixels. To blur an image, replace each color value by the average of itself and its eight neighbors. For edge detection, replace each color value c with $4c - n - e - s - w$, where the other colors are those of the pixel to the north, east, south, and west. Note that these cannot be implemented lazily, using the approach of Section 3.6, “Laziness,” on page 56, since they require the image from the previous stage (or at least the neighboring pixels) to have been computed. Enhance the lazy image processing to deal with these operations. Force computation of the previous stage when one of these operators is evaluated.
14. To deal with lazy evaluation on a per-pixel basis, change the transformers so that they are passed a `PixelReader` object from which they can read other pixels in the image. For example, $(x, y, reader) \rightarrow reader.get(width - x, y)$ is a mirroring operation. The convolution filters from the preceding exercises can be easily implemented in terms of such a reader. The straightforward operations would simply have the form $(x, y, reader) \rightarrow reader.get(x, y).grayscale()$, and you can provide an adapter from `UnaryOperation<Color>`. A `PixelReader` is at a particular level in the pipeline of operations. Keep a cache of recently read pixels at each level in the pipeline. If a reader is asked for a pixel, it looks in the cache (or in the original image at level 0); if that fails, it constructs a reader that asks the previous transform.
15. Combine the lazy evaluation of Section 3.6, “Laziness,” on page 56, with the parallel evaluation of Section 3.7, “Parallelizing Operations,” on page 57.
16. Implement the `doInOrderAsync` of Section 3.8, “Dealing with Exceptions,” on page 58, where the second parameter is a `BiConsumer<T, Throwable>`. Provide a plausible use case. Do you still need the third parameter?
17. Implement a `doInParallelAsync(Runnable first, Runnable second, Consumer<Throwable>)` method that executes `first` and `second` in parallel, calling the handler if either method throws an exception.

18. Implement a version of the unchecked method in Section 3.8, “Dealing with Exceptions,” on page 58, that generates a `Function<T, U>` from a lambda that throws checked exceptions. Note that you will need to find or provide a functional interface whose abstract method throws arbitrary exceptions.
19. Look at the `Stream<T>` method `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`. Should `U` be declared as `? super U` in the first type argument to `BiFunction`? Why or why not?
20. Supply a static method `<T, U> List<U> map(List<T>, Function<T, U>)`.
21. Supply a static method `<T, U> Future<U> map(Future<T>, Function<T, U>)`. Return an object of an anonymous class that implements all methods of the `Future` interface. In the `get` methods, invoke the function.
22. Is there a `flatMap` operation for `CompletableFuture`? If so, what is it?
23. Define a `map` operation for a class `Pair<T>` that represents a pair of objects of type `T`.
24. Can you define a `flatMap` method for `Pair<T>`? If so, what is it? If not, why not?

This page intentionally left blank

Index

Symbols

- operator, for numbers, 159
- , in shell scripts, 151
- >, in lambda expressions, 4–6
- ;; (semicolon), in JavaScript, 141
- :: operator, in method references, 8
- / (slash), in Unix paths, 184
- `...` (back quotes), in shell scripts, 149
- ^ (caret), for denoting free variables, 4
- '...' and '..." (single and double quotes)
 - in JavaScript, 138
 - in shell scripts, 150
- [...] (square brackets), in JavaScript, 141, 144–145
- {...} (curly braces), in lambdas, 5
- \$ (dollar sign), in JavaScript, 142
- \${...}, in shell scripts, 150
- * (asterisk), in locales, 173
- \ (backslash), in Windows paths, 184
- #!, in shell scripts, 151
- + operator, for numbers, 159
- < operator, in JavaScript, 143

A

- abstract methods, in functional
 - interfaces, 6
- acceptEither method (CompletableFuture), 134
- accumulate method (LongAccumulator), 122
- accumulateAndGet method (AtomicXXX), 121
- actions, repeating, 49
- add method
 - of Bindings, 78
 - of LongAdder, 121
- addExact method (Math), 159
- addListener method (JavaFX), 72, 153
- addSuppressed method (Throwable), 182
- allOf method (CompletableFuture), 134
- AnchorPane class (JavaFX), 84
- and method
 - of Bindings, 78–79
 - of Predicate, 50
- Android, 80
- AnnotatedElement interface, getAnnotation, getAnnotationsByType methods, 168

- annotations, 167–171
 - container, 167
 - in lambdas, 5
 - no annotations for, 170
 - repeated, 167–169
 - type use, 169–170
- anyMatch method (Stream), 28
- anyOf method (CompletableFuture), 134
- Apollo 11, launch of, 104, 109
- applets, 190–192
- Application class
 - init method, 153
 - start method, 71
 - stop method, 153
- applyToEither method (CompletableFuture), 134
- \$ARG, in shell scripts, 151
- ArithmeticException, 159
- arrays
 - and generic types, 10, 61
 - computing values of, 129
 - from stream elements, 33
 - in Nashorn, 144–145
 - sorting, 128
 - type use annotations in, 170
- Arrays class
 - parallelXXX methods, 128–129
 - sort method, 3, 6
 - stream method, 39
- ASCII characters, printable, 166
- Ask.com toolbar, 192
- asPredicate method (Pattern), 172
- asynchronous applications, 131
- atomic values, 120–123
 - and performance, 121
 - in concurrent hash maps, 124–126
- AtomicXXX classes
 - accumulateAndGet method, 121
 - compareAndSet method, 120
 - getAndXXX methods, 121
 - incrementAndGet method, 120
 - updateAndGet method, 121
- Atwood's law, 153
- atZone method (LocalDateTime), 109
- autoboxing, reducing, 52
- AutoCloseable interface, 25, 163, 180–181, 195
- average method (XXXStream), 40
- AWT (Abstract Window Toolkit), 70
- B**
- Base64, BASE64Encoder classes, 166
- BeanInfo class, 73
- between method (Duration), 103
- BiConsumer interface, 43, 51
- BiFunction interface, 7, 43, 51
- BigInteger class, 159
 - constructor for, 193
- BinaryOperator interface, 43, 51
- bind, bindBidirectional methods (XXXProperty), 76
- Binding interface, 78
- bindings, 75–80
 - lambdas for, 79
 - with JavaScript, 140
- Bindings class, methods of, 78–79
- BiPredicate interface, 51
- BitSet class, 195
 - constructor for, 195
 - stream method, 161, 196
 - toXXXArray methods, 196
 - valueOf method, 196
- books, counting words in, 22
- Boolean class, logicalXXX methods of, 158
- BooleanProperty class, 74
- BooleanSupplier interface, 53
- BorderLayout control (Swing), 81
- BorderPane class (JavaFX), 80–81, 84
- boxed method (XXXStream), 40
- BufferedReader class, lines method, 164
- BufferedXXX classes, 186
- buttons
 - disabling, at the ends of a gauge, 77
 - event handling for, 3, 8
- Byte class
 - compare method, 190
 - decode method, 193
 - parseByte method, 193
 - toUnsignedXXX methods, 158
 - valueOf method, 193
- BYTE static field, 158
- byteValueExact method (BigInteger), 159

C

- C++ programming language
 - default methods in, 15
 - unsigned types in, 159
- Calendar class, 101
- Callable interface
 - call method, 8
 - checked exceptions in, 60
- casts, type use annotations in, 170
- catch statement, 181
 - multiple exceptions in, 182–183
- certificates, signing, 190
- ChangeListener interface, 75, 77, 153
- charAt method (String), 25
- CharSequence interface
 - chars, codePoints methods, 39, 158
 - splitting by regular expressions, 24, 172
- checked exceptions, 8
- checkedNavigableXXX methods (Collections), 162
- checkedQueue method (Collections), 162
- Checker Framework, 169–170
- Church, Alonzo, 4, 104
- “class wins” rule, 16
- ClassCastException, 162
- classes
 - and classloader, 195
 - companion, 16
 - extending, in JavaScript, 146–148
- classifier functions, 36
- clone method (Object), 6
- close method (AutoCloseable), 163, 180–181
- Closeable interface, 180
- closures, 11
- code units, 39, 158
- codePoints method (CharSequence), 39, 158
- collect method (Stream), 33–34
- Collection interface, 160–163
 - parallelStream method, 23, 40
 - removeIf methods, 160–161
 - stream method, 22–24
 - toArray method, 61
- collections, 160–163
 - and lambda expressions, 160
 - processing, 22–23
 - sorting, 27
 - threadsafe, 119
 - vs. streams, 22
- Collections class, 17, 162–163
 - checkedQueue method, 162
 - emptySortedXXX methods, 163
 - sort method, 27
 - xxxNavigableXXX methods, 162
- collections library
 - adding forEach method to, 14
 - and function expressions, 1
- Collector interface, 33
- Collectors class, 33
 - counting method, 37
 - groupingBy method, 36–39
 - groupingByConcurrent method, 41
 - joining method, 34
 - mapping method, 38
 - maxBy, minBy methods, 37
 - partitioningBy method, 36–37, 39
 - reducing method, 38
 - summarizingXXX methods, 34
 - summingXXX methods, 37
 - toCollection method, 33
 - toConcurrentMap method, 36
 - toMap method, 34–36
 - toSet method, 33, 37
- collectors, downstream, 37–39
- com global object (JavaScript), 141
- Comparator interface, 2, 161–162
 - and lambdas, 6
 - compare method, 17
 - comparing method, 17, 161–162
 - naturalOrder method, 162
 - nullXXX methods, 162
 - reverseOrder, reversed methods, 162
 - thenComparing method, 161–162
- comparators, 49
 - chaining, 161
 - comparing integers in, 189–190
 - customizing, 54
- compare method (integer types), 3–6, 189–190
- compareAndSet method (AtomicXXX), 120
- compareUnsigned method (Integer, Long), 159

- CompletableFuture class, 130–131
 - acceptEither, applyToEither methods, 134
 - allOf, anyOf methods, 134
 - exceptions in, 133
 - handle method, 133
 - runAfterXXX methods, 134
 - runAsync method, 131
 - supplyAsync method, 131–132
 - thenAccept method, 132–133
 - thenAcceptBoth method, 134
 - thenApply method, 63, 130–133
 - thenApplyAsync method, 131–133
 - thenCombine method, 134
 - thenCompose method, 132–133
 - thenRun method, 133
 - whenComplete method, 133
- CompletionStage interface, 134
- compose method (UnaryOperator), 55–56
- composition pipeline, 131–132
- compute, computeIfXXX methods (Map), 161
- concat method
 - of Bindings, 79
 - of Stream, 26
- concurrent programming, 1, 119–135
- ConcurrentHashMap class, 123–128
 - atomic updates in, 124–126
 - compute, computeIfXXX methods, 125–126
 - forEach, forEachXXX methods, 126–128
 - get method, 124
 - mappingCount method, 123
 - merge method, 125–126
 - newKeySet, keySet methods, 128
 - organizing buckets as trees in, 124
 - put method, 124
 - putIfAbsent method, 125
 - reduce, reduceXXX methods, 126–128
 - replace method, 124
 - search, searchXXX methods, 126–128
- config method (Logger), 171
- constructor references, 9–10
 - for arrays, 10
 - type use annotations in, 170
- Consumer interface, 43, 51
- convert method (Bindings), 79
- copy method (Files), 186–187
- count method (Stream), 22–23, 28
- counting method (Collectors), 37
- createBindings method (ScriptEngine), 140
- createXXX methods (Files), 186–187
- createXXXBindings methods (Bindings), 79
- CSS (Cascading Style Sheets), 80
 - using with JavaFX, 90–91
- D**
- Date class, 115, 174
- Date and Time API, 101–117
 - and legacy code, 115–116
- dates
 - computing, 107–108
 - difference between, 104–106
 - formatting, 112–115
 - local, 104–106
 - local preferences for, 172
 - parsing, 115
- DateTimeFormatter class, 112–115
 - and legacy classes, 116
 - format method, 112
 - ofLocalizedXXX methods, 112
 - ofPattern method, 114
 - parse method, 115
 - toFormat method, 114
 - withLocale method, 112
- DayOfWeek class, 106
- dayOfWeekInMonth method (TemporalAdjusters), 107
- deadlocks, 119
 - in loggers, 193
- debugging
 - layouts, 84, 91
 - streams, 27
 - with checked wrappers, 162
- decrementExact method (Math), 159
- default methods, 14–16
 - adding to interfaces, 16
 - resolving ambiguities in, 15
- deferred execution, 2–4, 48–49
- delete, deleteIfExists methods (Files), 187
- deployment rule sets, 193
- directories
 - checking existence of, 187
 - creating, 186
 - deleting, 188

- paths for, 184
 - streams of, 165
 - temporary, 187
 - working, changing, 194
 - DirectoryStream interface, 165
 - disableProperty method (JavaFX), 77
 - distinct method (Stream), 27, 41, 160
 - divide method (Bindings), 76, 78
 - dividedBy method (Instant, Duration), 103
 - divideUnsigned method (Integer, Long), 159
 - Double class
 - compare method, 190
 - decode method, 193
 - isXXX methods, 159
 - parseDouble method, 193
 - sum, max, min methods, 158
 - valueOf method, 193
 - DoubleAccumulator, DoubleAdder classes, 122
 - DoubleProperty class, 74–75
 - doubles method (Random), 40
 - DoubleStream class, 39–40
 - boxed method, 40
 - mapToDouble method, 39
 - range, rangeClosed methods, 39
 - sum, average, max, min methods, 40
 - summaryStatistics method, 40
 - toArray method, 40
 - DoubleSummaryStatistics class, 34, 40
 - DoubleXXX interfaces, 43, 53
 - downstream collectors, 37–39
 - DropShadow class (JavaFX), 93
 - Duration class
 - arithmetic operations, 103
 - between method, 103
 - immutable, 104
 - toXXX methods, 103
 - dynamically typed languages, 143
- E**
- ECMAScript standard, 137, 146
 - edu global object (JavaScript), 141
 - Emacs text editor, running jjs inside, 139
 - emails, binary data in, 166
 - empty method (Optional), 30
 - emptyNavigableXXX methods (Collections), 162
 - emptySortedXXX methods (Collections), 163
 - <<END, in shell scripts, 150
 - \$ENV, in shell scripts, 151
 - environment variables, 151
 - epoch, definition of, 102
 - equal, equalIgnoreCase methods (Bindings), 78
 - equals method (Object), 16, 188
 - \$ERR, in shell scripts, 149
 - Error, Exception classes, disabling
 - suppressed exceptions in, 182
 - eval method (ScriptEngine), 140
 - event handlers
 - deferred execution in, 3
 - for asynchronous actions, 131
 - passing methods to, 8
 - event-driven programming, 1
 - exception specifications, type use
 - annotations in, 170
 - exceptions, 58–61
 - catching multiple, 182–183
 - checked, 7–8
 - in functional interfaces, 60–61
 - in reflective methods, 183
 - suppressed, 181–182
 - exec method (Runtime), 194
 - executeLargeUpdate method (Statement), 174
 - ExecutionException, 133
 - exists method (Files), 187
 - exit function (shell scripts), 152
 - \$EXIT, in shell scripts, 149
 - expression closure, 146
 - extends keyword, for function types, 62
- F**
- FadeTransition class (JavaFX), 92
 - fat clients, 70
 - File class, toPath method, 185
 - FileReader class, 163
 - files
 - checking existense of, 187
 - closing, 163
 - copying/moving/deleting, 187
 - creating, 184, 186
 - processing, 50
 - reading, 185
 - all words of, 181
 - lazily, 163–164

- files (*continued*)
 - redirecting standard I/O streams to, 194
 - saving streams into, 186
 - specifying encoding for, 163
 - temporary, 187
 - working with, 183–188
 - writing, 185
- Files class, 185–188
 - copy method, 186–187
 - createXXX methods, 186–187
 - delete, deleteIfExists methods, 187
 - encodings in, 186
 - exists method, 187
 - lines method, 25, 163
 - list method, 165
 - move method, 187
 - newBufferedXXX, newXXXStream methods, 186
 - readAllXXX methods, 185
 - walk method, 165
 - write method, 185
- FileTime class, and legacy classes, 116
- FileVisitor interface, 188
- FileXXXStream classes, 186
- fillInStackTrace method (Throwable), 182
- filter method
 - of Locale, 173
 - of Stream, 22–23, 25, 28, 42–43, 160
- final modifier, in lambdas, 5
- finally statement, 181
- findXXX methods (Stream), 28
- fine, finer, finest methods (Logger), 171
- firstDayOfXXX methods (TemporalAdjusters), 107
- Flash, 70
- flatMap method
 - of Optional, 30–31, 64
 - of Stream, 26
- Float class
 - compare method, 190
 - isXXX methods, 159
 - sum, max, min methods, 158
- FloatProperty class, 74
- floorXXX methods (Math), 159–160
- FlowPane class (JavaFX), 84
- for loop, enhanced, 165
- forEach method
 - adding to collection library, 14
 - of ConcurrentHashMap, 126–128
 - of Iterable, 14, 161
 - of Map, 161
 - of Stream, 34
- forEachOrdered method (Stream), 34
- forEachRemaining method (Iterator), 161
- forEachXXX methods (ConcurrentHashMap), 126–128
- forLanguageTag method (Locale), 173
- format method
 - of Bindings, 79
 - of DateTimeFormatter, 112
- formatters, for date/time values
 - custom, 114
 - predefined, 112–113
- from method (Instant, ZonedDateTime), 115
- Function interface, 43, 51
 - identity method, 35
- function keyword (JavaScript), 146
- function types
 - generic, 50
 - using wildcards for, 62
- functional interfaces, 6–8, 42–43
 - annotating, 7
 - as return type, 53–54
 - choosing, 48, 50–53
 - composing, 54, 63
 - conversion to, 6
 - defining, 52
 - exceptions in, 60–61
 - generic, 7
 - methods inabstract, 6
 - methods innonabstract, 50
 - parallelizing, 57–58
 - processed lazily, 56–57
- functional programming, 1
- @FunctionalInterface annotation, 7, 52
- Future interface, 130
- futures
 - combining multiple, 133–134
 - completable, 130–134
 - composing, 132–134
 - fx:id attribute (FXML), 89

FXML, 86–90
 constructing elements in, 87
 initialization in, 89
 writing files in, 87
 @FXML annotation, 88

G

GaussianBlur class (JavaFX), 94
 generate method (Stream), 24, 39
 generic types
 and arrays, 10
 and lambdas, 61–62
 type use annotations in, 170
 generic wrappers, 60
 get method
 of ConcurrentHashMap, 124
 of LongAccumulator, 122
 of ObservableXXXValue, 78
 of Path, 184–185
 of property classes, 75
 getAndXXX methods (AtomicXXX), 121
 getAnnotation, getAnnotationsByType methods
 (AnnotatedElement), 168
 getAsXXX methods (OptionalXXX classes),
 40
 getAverage method (XXXSummaryStatistics), 34
 getBytes method (String), 185–186
 getEncoder, getXXXEncoder methods (Base64),
 166
 getFileName method (Path), 185
 getGlobal, getLogger methods (Logger), 193
 getMax method (XXXSummaryStatistics), 34
 getObject method (ResultSet, Statement), 174
 getParent, getRoot (Path), 185
 getStackTrace method (Throwable), 182
 getSuppressed method (Throwable), 181–182
 getters/setters
 in JavaFX, 73–75
 in Nashorn, 141
 getValue method (property classes), 75, 78
 getXXX methods (Date and Time API),
 105–106, 108, 111–112
 GlassFish administration tool, 150
 Glow class (JavaFX), 94
 greaterThan, greaterThanOrEqualTo methods
 (Bindings), 78

GregorianCalendar class, 115
 toZonedDateTime method, 115
 GridBagLayout control (Swing), 82
 GridPane class (JavaFX), 82–84, 87
 alignment in, 83
 using CSS with, 90–91
 Groovy programming language
 executing scripts in, 140
 JavaFX bindings in, 86
 groupingBy method (Collectors), 36–39
 groupingByConcurrent method (Collectors), 41
 GStreamer framework, 95–97

H

handle method (CompletableFuture), 133
 hash method (Arrays, Objects), 189
 hash tables, 123
 hashCode method
 of Objects, 189
 of primitive types, 158
 hasNext method (JavaScript), 146–147
 HBox class (JavaFX), 81–82, 84, 87
 alignment and padding in, 83
 using CSS with, 91
 here documents, 150
 HTML (HyperText Markup Language),
 80
 HTML 5, 190, 192
 HTTP authentication, 166

I

IANA (Internet Assigned Numbers
 Authority), 109
 identity method (Function), 35
 identity values, 32
 ifPresent method (Optional), 29
 IllegalStateException, 35
 images, transforming, 49–57
 parallel, 57–58
 implements specification, type use
 annotations in, 170
 in-car displays, user interfaces for, 71
 increment method (LongAdder), 121–122
 incrementAndGet method (AtomicXXX), 120
 incrementExact method (Math), 159
 info method (Logger), 49, 171

- inheritIO method (ProcessBuilder), 194
 - init method (Application), 153
 - Initializable interface, 88
 - inner classes
 - capturing values in, 9, 12
 - vs. lambdas, 6
 - InputStream class, 186
 - instanceof keyword, and type use
 - annotations, 170
 - Instant class, 102
 - and legacy classes, 116
 - arithmetic operations, 103–104
 - from method, 115
 - immutable, 104
 - now method, 102
 - Integer class
 - compare method, 3–6, 189–190
 - decode method, 193
 - parseInt method, 193
 - sum, max, min methods, 158
 - toUnsignedLong method, 158
 - valueOf method, 193
 - xxxUnsigned methods, 159
 - integer ranges, 39
 - integer remainders, 159–160
 - IntegerProperty class, 74–75
 - interfaces
 - functional, 42–43, 48, 50–53
 - implemented in JavaScript, 146–148
 - methods in, 16–17
 - default, 14–16
 - name clashes between, 15
 - nonabstract, 6
 - Introspector class, 73
 - ints method (Random), 40
 - IntStream class, 39–40
 - boxed method, 40
 - mapToInt method, 39–40
 - of method, 39
 - range, rangeClosed methods, 39
 - sum, average, max, min methods, 40
 - summaryStatistics method, 40
 - toArray method, 40
 - IntSummaryStatistics class, 34, 40
 - intValueExact method (BigInteger), 159
 - IntXXX interfaces, 43, 53
 - InvalidationListener interface, 75, 77, 153
 - InvalidPathException, 184
 - IOException, 164
 - isEmpty, isEmpty, isNull, isNotNull methods (Bindings), 78
 - isEqual method (Predicate), 50
 - isFinite, isInfinite, isNaN methods (Double, Float), 159
 - isLoggable method (Logger), 49
 - isNull method (Objects), 171
 - isPresent method (Optional), 28–29
 - isXXX methods (Date and Time API), 105, 108, 112
 - isZero, isNegative methods (Instant, Duration), 103
 - Iterable interface, 165
 - forEach method, 14, 161
 - iterate method (Stream), 24, 27, 39
 - Iterator interface, forEachRemaining method, 161
 - iterators, 33
 - for random numbers, 146–147
- ## J
- Java Media Framework, 95
 - Java programming language
 - executing external commands from, 194
 - implementation bugs in, 191
 - simplicity and consistency of, 4
 - Java Web Start, 190–192
 - java, javax, javax global objects (JavaScript), 141
 - Java.extend function (JavaScript), 146–147
 - Java.from function (JavaScript), 145
 - Java.super function (JavaScript), 148
 - Java.to function (JavaScript), 144–145
 - Java.type function (JavaScript), 141–142
 - java.util.concurrent package, 119, 130
 - java.util.concurrent.atomic package, 120
 - java.util.function package, 7
 - JavaBeans, 73
 - javadoc comments, redeclaring Object methods for, 6
 - JavaFX, 69–98
 - controls in, 94–97
 - debugging in, 84, 91

- dimensions in, 82
 - event handling in, 72–73
 - getters/setters in, 73–75
 - labels in, 71
 - launching, 72
 - from Nashorn, 152–154
 - layouts in, 80–86
 - alignment, 83
 - cell styling, 84, 91
 - markup, 86–90
 - padding property, 82
 - panes, 80
 - properties in, 72–75, 153
 - scenes in, 71
 - setting fonts in, 71
 - sliders in, 72–73
 - special effects in, 91–94
 - stages in, 71, 140
 - transitions in, 92
 - using CSS in, 90–91
 - versions of, 70
- JavaFX Script programming language, 85
- JavaScript programming language
- accessing Java applications from, 153
 - anonymous functions in, 146
 - anonymous subclasses in, 147
 - bracket notation in, 141, 144–145
 - calling applets from, 191
 - catching Java exceptions in, 148
 - constructing objects in, 141–142
 - delimiters in, 138
 - extending Java classes in, 146–148
 - implementing Java interfaces in, 146–148
 - inner classes in, 142
 - instance variables in, 147
 - invoking:
 - Java methods in, 140–141
 - superclasses in, 148
 - making JavaFX stages visible in, 140
 - Mozilla implementation of, 146
 - no method overloading in, 141
 - numbers in, 143
 - objects vs. strings in, 143
 - REPL in, 138–139
 - semicolons in, 141
 - static methods in, 142
 - using with Nashorn, 137–155
 - `javax.annotation.processing` package, 169
 - `javax.lang.model` package, 169
 - JDBC (Java Database Connectivity), 174
 - JDK (Java Development Kit), installed by users, 192
 - JEditorPane control (Swing), 95
 - jjs tool, 138–139
 - command-line arguments in, 151
 - executing commands in, 149
 - fx option, 152
 - join method (String), 158
 - joining method (Collectors), 34
 - JRuby programming language, 140
 - jrnscrip script shell, 149, 151
 - JUnit test, automated execution of, 195
 - JUnitCore class, 195
 - JVM (Java Virtual Machine), installed by users, 192
 - Jython programming language, 140
- ## K
- keySet method (ConcurrentHashMap), 128
 - kiosks, user interfaces for, 71
- ## L
- Label class, setFont method, 71
 - lambda expressions, 1–17, 48–49
 - accessing variables in, 10–13
 - and collections, 160
 - and computed bindings, 79
 - and functional interfaces, 6
 - and generics, 61–62
 - and JavaScript, 146
 - and method references, 8
 - annotations in, 5
 - capturing values by, 11
 - event handling with, 72
 - modifiers in, 5
 - no assigning to a variable of type Object, 7
 - no-arg, 48
 - parameter types in, 5
 - parameters of, 49–50
 - result type of, 6

- lambda expressions (*continued*)
 - scope of, 13
 - syntax of, 4–5
 - this keyword in, 13
 - throwing exceptions in, 58–61
 - updating counters with, 13
 - using with `map` method, 25
 - vs. inner classes, 6
 - language range, 173
 - lastXXX methods (TemporalAdjusters), 107
 - leap seconds, 102
 - leap years, 105
 - length method (Bindings), 78
 - lessThan, lessThanOrEqual methods (Bindings), 78
 - limit method (Stream), 26, 41
 - lines method
 - of `BufferedReader`, 164
 - of `Files`, 25, 163
 - lines, reading, 25, 163–164, 185
 - Lisp programming language, 1
 - list method (`Files`), 165
 - List interface, 17
 - `replaceAll`, sort methods, 160–161
 - ListProperty class, 74
 - lists
 - declaring non-null elements of, 169
 - in Nashorn, 145
 - LocalDate class, 174
 - and legacy classes, 116
 - methods of, 105–106
 - LocalDateTime class, 109, 174
 - and legacy classes, 116
 - `atZone` method, 109
 - Locale class
 - `filter` method, 173
 - `forLanguageTag` method, 173
 - `lookup` method, 173
 - locales, 36, 172–174
 - default, 112
 - finding, 173
 - formatting styles for, 114
 - LocalTime class, 108–109, 174
 - and legacy classes, 116
 - methods of, 108
 - locks, 122
 - Logger class
 - `getGlobal`, `getLogger` methods, 193
 - `isLoggable` method, 49
 - `log`, `logp`, `severe`, `warning`, `info`, `config`, `fine`, `finer`, `finest` methods, 171
 - `Logger.global` instance, 193
 - logging, lazily, 48–49, 171–172
 - logicalXXX methods (`Boolean`), 158
 - Long class
 - `compare` method, 190
 - `decode` method, 193
 - `parseLong` method, 193
 - `sum`, `max`, `min` methods, 158
 - `valueOf` method, 193
 - `xxxUnsigned` methods, 159
 - LongAccumulator class, 121
 - `accumulate`, `get` methods, 122
 - LongAdder class, 121, 125
 - `add`, `sum` methods, 121
 - `increment` method, 121–122
 - LongProperty class, 74
 - longs method (`Random`), 40
 - LongStream class, 39–40
 - `boxed` method, 40
 - `mapToLong` method, 39
 - `range`, `rangeClosed` methods, 39
 - `sum`, `average`, `max`, `min` methods, 40
 - `summaryStatistics` method, 40
 - `toArray` method, 40
 - LongSummaryStatistics class, 34, 40
 - LongValueExact method (`BigInteger`), 159
 - LongXXX interfaces, 43, 53
- ## M
- `map` method
 - of `Optional`, 29, 63
 - of `Stream`, 25, 63
 - `Map` interface, methods of, 161
 - `mapping` method (`Collectors`), 38
 - `mappingCount` method (`ConcurrentHashMap`), 123
 - `MapProperty` class, 74
 - `maps`
 - `concurrent`, 36
 - in Nashorn, 145
 - `merging`, 34–36, 41
 - `mapToInt` method (`Stream`), 32

- mapToXXX methods (XXXStream), 39–40
 - Math class, 159–160
 - max method
 - of Bindings, 78
 - of integer types, 158
 - of streams, 28, 40
 - maxBy method (Collectors), 37
 - Media, MediaXXX classes (JavaFX), 95–96
 - merge method (ConcurrentHashMap), 125–126
 - messages, constructed lazily, 171–172
 - method references, 8–9
 - this, super parameters in, 9
 - type use annotations in, 170
 - methods
 - abstract, in functional interfaces, 6
 - customizing functions passed to, 54
 - default, 14–16
 - adding to interfaces, 16
 - parameters of, available through reflection, 170–171
 - reflective, exceptions in, 183
 - resolving ambiguities in, 15–16
 - static, adding to interfaces, 16–17
 - Microsoft Office, 139
 - min method
 - of Bindings, 78
 - of integer types, 158
 - of streams, 28, 40
 - minus, minusXXX methods (Date and Time API), 103, 105–106, 108, 111
 - monads, 26, 63–64
 - MonthDay class, 106
 - move method (Files), 187
 - Mozilla JavaScript implementation, 146
 - multipliedBy method (Instant, Duration), 103
 - multiply method (Bindings), 78
 - multiplyExact method (Math), 159
- N**
- named capturing groups, 172
 - Nashorn engine, 137–155
 - anonymous subclasses in, 147
 - arrays in, 144–145
 - catching Java exceptions in, 148
 - class objects in, 142
 - extending Java classes in, 146–148
 - getters/setters in, 141
 - implementing Java interfaces in, 146–148
 - instance variables in, 147
 - invoking:
 - Java methods in, 140–141
 - superclasses in, 148
 - launching JavaFX from, 152–154
 - lists and maps in, 145
 - numbers in, 143
 - running from:
 - command line, 138–139
 - Java, 139–140
 - shell scripting in, 148–152
 - strings in, 142–143
 - naturalOrder method (Comparator), 162
 - NavigableXXX classes, 162
 - negate method
 - of Bindings, 78
 - of Predicate, 50
 - negated method (Instant, Duration), 103
 - negateExact method (Math), 159
 - new keyword, in constructor references, 9
 - new operator (JavaScript), 142, 144, 147
 - newBufferedXXX, newXXXStream methods (Files), 186
 - newKeySet method (ConcurrentHashMap), 128
 - next method (JavaScript), 146–147
 - next, nextOrSame methods (TemporalAdjusters), 107
 - nextXXX methods (Math), 160
 - NIO (New I/O) library, 183
 - nominal typing, 50
 - noneMatch method (Stream), 28
 - noninterference, of stream operations, 42
 - @NonNull annotation, 169
 - nonNull method (Objects), 171
 - normalize method (Path), 185
 - NoSuchElementException, 29
 - not method (Bindings), 78
 - notEqual, notEqualIgnoreCase methods (Bindings), 78
 - now method (Date and Time API), 102–103, 105, 108, 111
 - @Nullable annotation, 169
 - NullPointerException, 28, 169, 194

null-safe equality testing, 188
 nullXXX methods (Comparator), 162
 Number type (JavaScript), 143
 numbers, 158–159

- arithmetic operations on, 158
- comparing, 162, 189–190
- converting from strings, 193
- unsigned, 158–159

O

Object class

- clone method, 6
- equals method, 16
- no redefining for methods of, 16
- toString method, 6, 16, 189

object-oriented programming, 1

ObjectProperty class, 74

Objects class

- equals method, 188
- hash method, 189
- hashCode method, 189
- isNull, nonNull methods, 171
- requireNonNull method, 172, 194

ObjXXXConsumer interfaces, 53

Observable, ObservableValue interfaces, 77

ObservableXXXValue interfaces, 78

of method

- of Date and Time API, 105, 108–109–111
- of IntStream, 39
- of Optional, 30
- of Stream, 24

ofDateAdjuster method (TemporalAdjusters), 107

OffsetDateTime class, 112

ofInstant method (ZonedDateTime), 111

ofLocalizedXXX methods (DateTimeFormatter), 112

ofNullable method (Optional), 30

ofPattern method (DateTimeFormatter), 114

Optional class, 28–31

- creating values of, 30
- empty method, 30
- flatMap method, 30–31, 64
- ifPresent method, 29
- isPresent method, 28–29
- map method, 29, 63
- of, ofNullable methods, 30

OptionalXXX classes, 40

or method

- of Bindings, 78
- of Predicate, 50

org global object (JavaScript), 141

\$OUT, in shell scripts, 149

P

Package object (JavaScript), 141

parallel method (Stream), 40

parallelism threshold, 126

parallelStream method (Collection), 23, 40, 160

ParallelTransition class (JavaFX), 93

parallelXXX methods (Arrays), 128–129

Parameter class, 171

parse method (DateTimeFormatter), 115

parseXXX methods (integer types), 193

partitioningBy method (Collectors), 36–37, 39

Path interface, 16–17, 184–185

- get, getXXX methods, 184–185
- normalize method, 185
- relativize method, 184
- resolve, resolveSibling methods, 184
- toAbsolutePath, toFile methods, 185

paths

- absolute, 185
- combining, 185
- for directories, 184
- resolving, 184

Paths class, 16–17

Pattern class

- asPredicate method, 172
- splitAsStream method, 24, 172

peek method (Stream), 27, 34

performance, and atomic values, 121

Period class, 106

PHP programming language, 140

plus, plusXXX methods (Date and Time API), 103, 105–106, 108, 110–111

Predicate interface, 42–43, 50–51

- and, or, negate, isEqual methods, 50

previous, previousOrSame methods (TemporalAdjusters), 107

primitive types

- comparing, 162
- specializations for, 52

- streams of, 39–40, 161
- transforming hash map values to, 128
- wrappers for, 158
- println method (System.out), 8
- Process class, waitFor method, 194
- ProcessBuilder class, 194–195
 - inheritIO method, 194
 - redirectXXX methods, 194
- Programmer’s Day, 105
- properties (JavaFX), 72–80
 - bound, 73
 - computed, 76
 - enumerating, 73
 - final, 74
 - implementing, 74
 - listeners for, 72, 74, 153
 - numeric, using Changelistener for, 75
 - updating automatically, 75–80
- Property interface, 74
- put method (ConcurrentHashMap), 124
- putIfAbsent method
 - of ConcurrentHashMap, 125
 - of Map, 161

R

- race conditions, 41, 119–120
- Random class, methods of, 40
- random numbers, 40, 146–147
- range, rangeClosed methods (XXXStream), 39
- readAllXXX methods (Files), 185
- readLine function (shell scripts), 151
- redirectXXX methods (ProcessBuilder), 194
- reduce, reduceXXX methods (ConcurrentHashMap), 31–33, 126–128
- reducing method (Collectors), 38
- reductions, 28, 31–32
- reflection, 170–171
- reflective methods, 183
- ReflectiveOperationException, 183
- regular expressions, 172
- relativize method (Path), 184
- rem units, 82
- remainderUnsigned method (Integer, Long), 159
- remove method (Map), 161
- removeIf method (Collection), 160–161
- @Repeatable annotation, 168

- REPL (“read-eval-print” loop), 138–139
- replace method
 - of ConcurrentHashMap, 124
 - of Map, 161
- replaceAll method
 - of List, 160–161
 - of Map, 161
- repurposing attacks, 191
- requireNonNull method (Objects), 172, 194
- resolve, resolveSibling methods (Path), 184
- ResultSet class, get/setObject methods, 174
- return statement, in lambdas, 5
- reverseOrder, reversed methods (Comparator), 162
- Rhino engine, 137
- rlwrap tool, 139
- RotateTransition class (JavaFX), 92–93
- runAfterXXX methods (CompletableFuture), 134
- runAsync method (CompletableFuture), 131
- Runnable interface, 51
 - and lambdas, 6
 - run method, 2
- Runtime class, exec method, 194
- RuntimeException class, disabling suppressed exceptions in, 182

S

- sandbox, 190–191
- Scala programming language
 - covariant type parameters in, 61
 - default methods in, 15
 - JavaFX bindings in, 86
 - REPL in, 139
- ScaleTransition class (JavaFX), 92
- Scanner class, 164, 182
- SceneBuilder program, 80, 87
- scheduling applications
 - and time zones, 104, 109
 - computing dates for, 107–108
- Scheme programming language, 1
 - executing scripts in, 140
- ScriptEngine interface, createBindings, eval methods, 140
- search, searchXXX methods (ConcurrentHashMap), 126–128
- seconds, leap, 102
- security, 190–193

- select, selectXXX methods (Bindings), 79
- SequentialTransition class (JavaFX), 93
- set, setValue methods (property classes), 75
- setFont method (Label), 71
- setObject method (ResultSet, Statement), 174
- SetProperty class, 74
- sets
 - flattening, 26, 63
 - operations on, for integers, 195
 - threadsafe, 128
- severe method (Logger), 171
- shebang, 151
- shell scripts, 148–152
 - command-line arguments in, 151
 - environment variables in, 151
 - string interpolation in, 150
- Short class
 - compare method, 190
 - decode method, 193
 - parseShort method, 193
 - sum, max, min methods, 158
 - toUnsignedXXX methods, 158
 - valueOf method, 193
- shortValueExact method (BigInteger), 159
- SimpleXXXProperty classes, 74
- size method (Bindings), 78
- SIZE static field, 158
- skip method (Stream), 26
- sleep method (Thread), 8
- slice method (JavaScript), 142
- sort method
 - of Arrays, 3, 6
 - of Collections, 27
 - of List, 160–161
- sorted method (Stream), 27
- sorting
 - people, by name, 161–162
 - strings by length, 2–5
- split method (String), 158
- splitAsStream method (Pattern), 24, 172
- splitIterator method (Collection), 160
- square root, computing, 31
- stack traces
 - disabling, 182
 - working back from, 194
- StackPane class (JavaFX), 84
- \$STAGE, in shell scripts, 152
- StampedLock class, 122–123
- start method
 - of Application, 71
 - of Thread, 2
- Statement class, methods of, 174
- static methods, adding to interfaces, 16–17
- stop method (Application), 153
- stream method
 - of Arrays, 39
 - of BitSet, 161, 196
 - of Collection, 22–24, 160
- Stream interface
 - collect method, 33
 - concat method, 26
 - count method, 22–23, 28
 - distinct method, 27, 41, 160
 - filter method, 22–23, 25, 28, 42–43, 160
 - findXXX methods, 28
 - flatMap method, 26
 - generate method, 24, 39
 - iterate method, 24, 27, 39
 - limit method, 26, 41
 - map method, 25, 63
 - mapToInt method, 32
 - max, min methods, 28
 - of method, 24
 - parallel method, 40
 - peek method, 27, 34
 - skip method, 26
 - sorted method, 27
 - toArray method, 10, 33
 - unordered method, 41
 - xxxMatch methods, 28
- streams, 21–43
 - closing, 25, 164
 - combining, 26
 - converting:
 - between objects and primitive types
 - of, 39–40
 - to arrays, 33
 - creating, 24–25
 - debugging, 27
 - empty, 32
 - extracting substreams from, 26
 - flattening, 26

- infinite, 23–24, 27
 - cutting, 26
 - intermediate operations for, 23
 - noninterference of, 42
 - null checks for, 171
 - of directory entries, 165
 - of primitive type values, 39–40, 161
 - of random numbers, 40
 - ordered, 41
 - parallel, 23, 28, 34, 40–42
 - pipeline of, 27
 - processed lazily, 23, 27, 42
 - reductions of, 28
 - sorting, 27
 - standard I/O, redirecting to files, 194
 - terminal operation for, 23
 - threadsafe operations on, 41
 - transformations of, 25–26
 - stateful, 27
 - stateless, 26–27
 - vs. collections, 22
 - with no elements, 24
- StrictMath class, 160
- String class
- charAt method, 25
 - getBytes method, 185–186
 - join, split methods, 158
 - toLowerCase method, 25
 - valueOf method, 189
- string interpolation, 150
- StringProperty class, 74–75
- strings
- combining, 158
 - converting to numbers, 193
 - filtering by regular expressions, 172
 - sorting by length, 2–5
 - splitting, 24, 158, 172
 - transforming to lowercase, 25
- subtract method (Bindings), 78
- subtractExact method (Math), 159
- sum method
- of integer types, 158
 - of LongAdder, 121
 - of XXXStream, 40
- summarizingXXX methods (Collectors), 34
- summaryStatistics method (XXXStream), 40
- summingXXX methods (Collectors), 37
- super keyword
- capturing in method references, 9
 - for function types, 62
- superclasses
- for related exceptions, 183
 - type use annotations in, 170
- Supplier interface, 43, 48, 51
- supplyAsync method (CompletableFuture), 131–132
- Swing, 70, 80
- naming controls in, 71
 - showing HTML in, 95
- synchronizedNavigableXXX methods (Collections), 162
- T**
- Temporal interface, with method, 107
- TemporalAdjuster interface, 107
- TemporalAdjusters class, 107
- thenAccept method (CompletableFuture), 132–133
- thenAcceptBoth method (CompletableFuture), 134
- thenApply method (CompletableFuture), 63, 130–133
- thenApplyAsync method (CompletableFuture), 131–133
- thenCombine method (CompletableFuture), 134
- thenComparing method (Comparator), 161–162
- thenCompose method (CompletableFuture), 132–133
- thenRun method (CompletableFuture), 133
- this keyword
- capturing in method references, 9
 - in lambda expressions, 13
- Thread class
- constructor for, 147
 - sleep method, 8
 - start method, 2
- threads
- atomic mutations in, 120–123
 - concurrency enhancements for, 119–135
 - executing:
 - code in, 2
 - increments concurrently, 12–13
 - locking, 120–123
 - race conditions in, 41

- threads (*continued*)
 - reading web pages in, 130
 - starting new, 2, 9
 - terminating upon an exception, 59
 - updating hash tables in, 123–128
 - Throwable class
 - addSuppressed method, 182
 - getStackTrace, fillInStackTrace methods, 182
 - getSuppressed method, 181–182
 - TilePane class (JavaFX), 84
 - time
 - between two instants, 103
 - current, 102
 - daylight savings, 109–112
 - formatting, 112–115
 - local, 108–109
 - measuring, 103
 - parsing, 115
 - zoned, 109–112
 - Time class, 115, 174
 - Timestamp class, 115, 174
 - timestamps, 112
 - using instants as, 103
 - TimeZone class, and legacy classes, 116
 - toAbsolutePath method (Path), 185
 - toArray method
 - of Collection, 61
 - of Stream, 10, 33
 - of XXXStream, 40
 - toCollection method (Collectors), 33
 - toConcurrentMap method (Collectors), 36
 - toFile method (Path), 185
 - toFormat method (DateTimeFormatter), 114
 - toInstant method
 - of Date, 115
 - of ZonedDateTime, 109, 112
 - toIntExact method (Math), 159
 - toLocalXXX methods (ZonedDateTime), 112
 - toLowerCase method (String), 25
 - toMap method (Collectors), 34–36
 - toPath method (File), 185
 - toSet method (Collectors), 33, 37
 - toString method (Object), 16
 - null-safe calling, 189
 - redeclaring, 6
 - toUnsignedXXX methods (integer types), 158
 - toXXX methods (Duration), 103
 - toXXXArray methods (BitSet), 196
 - ToXXXBiFunction interfaces, 53
 - ToXXXFunction interfaces, 43, 53
 - toXXXOfDay methods (LocalTime), 108
 - toZonedDateTime method (GregorianCalendar), 115
 - tryOptimisticRead method (StampedLock), 122–123
 - try-with-resources statement, 180–181
 - closing:
 - files with, 163–165
 - streams with, 25, 164
 - for JUnit tests, 195
 - suppressed exceptions in, 181–182
 - type parameters, 61
 - type use annotations, 169–170
- U**
- UnaryOperator interface, 43, 51–52
 - compose method, 55–56
 - unbind, unbindBidirectional methods (XXXProperty), 76
 - UncheckedIOException, 164
 - unmodifiableNavigableXXX methods (Collections), 162
 - unordered method (Stream), 41
 - until method (LocalDate), 105–106
 - updateAndGet method (AtomicXXX), 121
 - URLClassLoader class, 195
 - use-site variance. *See* wildcards
 - UTC (coordinated universal time), 110
- V**
- valueAt, XXXValueAt methods (Bindings), 79
 - valueOf method
 - of BitSet, 196
 - of integer types, 193
 - of String, 189
 - valueProperty method (JavaFX), 72
 - values
 - captured:
 - by inner classes, 12
 - by lambda expressions, 11
 - grouping, 36–38
 - partitioning, 36–37

variables

- accessing in lambdas, 10–13
- atomic mutations of, 120–123
- effectively final, 12

VB Script programming language,
139

VBox class (JavaFX), 82, 84

vendor lock, 139

videos, playing, 95–97

W

waitFor method (Process), 194

walk method (Files), 165

warning method (Logger), 171

web pages

- layout of, 80

- reading:

 - from URL, 63, 132

 - in a separate thread, 130

WebKit engine, 95

WebView class (JavaFX), 95

whenComplete method (CompletableFuture), 133

wildcards, 61

- type use annotations in, 170

- with method (Temporal), 107

- withLocale method (DateTimeFormatter), 112

- withXXX methods (Date and Time API),
105, 108, 111

- write method (Files), 185

Y

Year, YearMonth classes, 106

years, leap, 105

Z

ZonedDateTime class

- and legacy classes, 115–116

- methods of, 109–112, 115