Core Java®

Covers Java SE 8

for the Impatient

Cay S. Horstmann

Core Java® for the Impatient

This page intentionally left blank

Core Java®
for the 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

Library of Congress Cataloging-in-Publication Data

Horstmann, Cay S., 1959-
Core Java for the impatient / Cay S. Horstmann.
pages cm
Includes index.
ISBN 978-0-321-99632-9 (pbk. : alk. paper)—ISBN 0-321-99632-1 (pbk. : alk. paper)
1. Java (Computer program language) I. Title.
QA76.73.J38H67535 2015
005.13'3—dc23

2014046523

ISBN-13: 978-0-321-99632-9
ISBN-10: 0-321-99632-1

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, February 2015

To Chi—the most patient person in my life.

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Java is now about twenty years old, and the classic book, *Core Java*™, covers, in meticulous detail, not just the language but all core libraries and a multitude of changes between versions, spanning two volumes and well over 2,000 pages. But Java 8 changes everything. Many of the old Java idioms are no longer required, and there is a much faster, easier pathway for learning Java. In this book, I show you the "good parts" of modern Java so you can put them to work quickly.

As with my previous "Impatient" books, I quickly cut to the chase, showing you what you need to know for solving a programming problem without lecturing about the superiority of one paradigm over another. I also present the information in small chunks, organized so that you can quickly retrieve it when needed.

Assuming you are proficient in some other programming language, such as C++, JavaScript, Objective C, PHP, or Ruby, with this book you will learn how to become a competent Java programmer. I cover all aspects of Java that a developer needs to know, including the powerful lambda expressions and streams that were introduced in Java 8. I tell you where to find out more about old-fashioned concepts that you might still see in legacy code, but I don't dwell on them.

A key reason to use Java is to tackle concurrent programming. With parallel algorithms and threadsafe data structures readily available in the Java library, the way application programmers should handle concurrent programming has completely changed. I provide fresh coverage, showing you how to use the powerful library features instead of error-prone, low-level constructs.

Traditional books on Java are focused on user interface programming—but nowadays, few developers produce user interfaces on desktop computers. If you intend to use Java for server-side programming or Android programming, you will be able to use this book effectively without being distracted by desktop GUI code.

Finally, this book is written for application programmers, not for a college course and not for systems wizards. The book covers issues that application programmers need to wrestle with, such as logging and working with files—but you won't learn how to implement a linked list by hand or how to write a web server.

I hope you enjoy this rapid-fire introduction into modern Java, and I hope it will make your work with Java productive and enjoyable.

If you find errors or have suggestions for improvement, please visit `http://horstmann.com/javaimpatient` and leave a comment. On that page, you will also find a link to an archive file containing all code examples from the book.

Acknowledgments

My thanks go, as always, to my editor Greg Doench, who enthusiastically supported the vision of a short book that gives a fresh introduction to Java 8. Dmitry Kirsanov and Alina Kirsanova once again turned an XHTML manuscript into an attractive book with amazing speed and attention to detail. My special gratitude goes to the excellent team of reviewers who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Brian Goetz, Marty Hall, Mark Lawrence, Doug Lea, Simon Ritter, Yoshiki Shibata, and Christian Ullenboom.

Cay Horstmann
Biel/Bienne, Switzerland
January 2015

This page intentionally left blank

About the Author

Cay S. Horstmann is the author of *Java SE 8 for the Really Impatient* and *Scala for the Impatient* (both from Addison-Wesley), 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.

Chapter 3

Java was designed as an object-oriented programming language in the 1990s when object-oriented programming was the principal paradigm for software development. Interfaces are a key feature of object-oriented programming: They let you specify what should be done, without having to provide an implementation.

Long before there was object-oriented programming, there were functional programming languages, such as Lisp, in which functions and not objects are the primary structuring mechanism. Recently, functional programming has risen in importance because it is well suited for concurrent and event-driven (or "reactive") programming. Java supports function expressions that provide a convenient bridge between object-oriented and functional programming. In this chapter, you will learn about interfaces and lambda expressions.

The key points of this chapter are:

- An interface specifies a set of methods that an implementing class must provide.
- An interface is a supertype of any class that implements it. Therefore, one can assign instances of the class to variables of the interface type.
- An interface can contain static methods. All variables of an interface are automatically static and final.
- An interface can contain default methods that an implementing class can inherit or override.

- The `Comparable` and `Comparator` interfaces are used for comparing objects.
- A lambda expression denotes a block of code that can be executed at a later point in time.
- Lambda expressions are converted to functional interfaces.
- Method and constructor references refer to methods or constructors without invoking them.
- Lambda expressions and local inner classes can access effectively final variables from the enclosing scope.

3.1 Interfaces

An *interface* is a mechanism for spelling out a contract between two parties: the supplier of a service and the classes that want their objects to be usable with the service. In the following sections, you will see how to define and use interfaces in Java.

3.1.1 Declaring an Interface

Consider a service that works on sequences of integers, reporting the average of the first `n` values:

```
public static double average(IntSequence seq, int n)
```

Such sequences can take many forms. Here are some examples:

- A sequence of integers supplied by a user
- A sequence of random integers
- The sequence of prime numbers
- The sequence of elements in an integer array
- The sequence of code points in a string
- The sequence of digits in a number

We want to implement *a single mechanism* for deal with all these kinds of sequences.

First, let us spell out what is common between integer sequences. At a minimum, one needs two methods for working with a sequence:

- Test whether there is a next element
- Get the next element

To declare an interface, you provide the method headers, like this:

```
public interface IntSequence {
    boolean hasNext();
    int next();
}
```

You need not implement these methods, but you can provide default implementations if you like—see Section 3.2.2, "Default Methods," on p. 100. If no implementation is provided, we say that the method is *abstract*.

NOTE: All methods of an interface are automatically `public`. Therefore, it is not necessary to declare `hasNext` and `next` as `public`. Some programmers do it anyway for greater clarity.

The methods in the interface suffice to implement the `average` method:

```
public static double average(IntSequence seq, int n) {
    int count = 0;
    double sum = 0;
    while (seq.hasNext() && count < n) {
        count++;
        sum += seq.next();
    }
    return count == 0 ? 0 : sum / count;
}
```

3.1.2 Implementing an Interface

Now let's look at the other side of the coin: the classes that want to be usable with the `average` method. They need to *implement* the `IntSequence` interface. Here is such a class:

```
public class SquareSequence implements IntSequence {
    private int i;

    public boolean hasNext() {
        return true;
    }

    public int next() {
        i++;
        return i * i;
    }
}
```

There are infinitely many squares, and an object of this class delivers them all, one at a time.

The `implements` keyword indicates that the `SquareSequence` class intends to conform to the `IntSequence` interface.

CAUTION: The implementing class must declare the methods of the interface as `public`. Otherwise, they would default to package access. Since the interface requires public access, the compiler would report an error.

This code get the average of the first 100 squares:

```
SquareSequence squares = new SquareSequence();
double avg = average(squares, 100);
```

There are many classes that can implement the `IntSequence` interface. For example, this class yields a finite sequence, namely the digits of a positive integer starting with the least significant one:

```
public class DigitSequence implements IntSequence {
    private int number;

    public DigitSequence(int n) {
        number = n;
    }

    public boolean hasNext() {
        return number != 0;
    }

    public int next() {
        int result = number % 10;
        number /= 10;
        return result;
    }

    public int rest() {
        return number;
    }
}
```

An object `new DigitSequence(1729)` delivers the digits `9 2 7 1` before `hasNext` returns `false`.

NOTE: The `SquareSequence` and `DigitSequence` classes implement all methods of the `IntSequence` interface. If a class only implements some of the methods, then it must be declared with the `abstract` modifier. See Chapter 4 for more information on abstract classes.

3.1.3 Converting to an Interface Type

This code fragment computes the average of the digit sequence values:

```
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
   // Will only look at the first four sequence values
```

Look at the `digits` variable. Its type is `IntSequence`, not `DigitSequence`. A variable of type `IntSequence` refers to an object of some class that implements the `IntSequence` interface. You can always assign an object to a variable whose type is an implemented interface, or pass it to a method expecting such an interface.

Here is a bit of useful terminology. A type `S` is a *supertype* of the type `T` (the *subtype*) when any value of the subtype can be assigned to a variable of the supertype without a conversion. For example, the `IntSequence` interface is a supertype of the `DigitSequence` class.

NOTE: Even though it is possible to declare variables of an interface type, you can never have an object whose type is an interface. All objects are instances of classes.

3.1.4 Casts and the `instanceof` Operator

Occasionally, you need the opposite conversion—from a supertype to a subtype. Then you use a *cast*. For example, if you happen to know that the object stored in an `IntSequence` is actually a `DigitSequence`, you can convert the type like this:

```
IntSequence sequence = ...;
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

In this scenario, the cast was necessary because `rest` is a method of `DigitSequence` but not `IntSequence`.

See Exercise 2 for a more compelling example.

You can only cast an object to its actual class or one of its supertypes. If you are wrong, a compile-time error or class cast exception will occur:

```
String digitString = (String) sequence;
    // Cannot possibly work—IntSequence is not a supertype of String
RandomSequence randoms = (RandomSequence) sequence;
    // Could work, throws a class cast exception if not
```

To avoid the exception, you can first test whether the object is of the desired type, using the `instanceof` operator. The expression

object `instanceof` *Type*

returns `true` if *object* is an instance of a class that has *Type* as a supertype. It is a good idea to make this check before using a cast.

```
if (sequence instanceof DigitSequence) {
    DigitSequence digits = (DigitSequence) sequence;
    ...
}
```

3.1.5 Extending Interfaces

An interface can *extend* another, providing additional methods on top of the original ones. For example, `Closeable` is an interface with a single method:

```
public interface Closeable {
    void close();
}
```

As you will see in Chapter 5, this is an important interface for closing resources when an exception occurs.

The `Channel` interface extends this interface:

```
public interface Channel extends Closeable {
    boolean isOpen();
}
```

A class that implements the `Channel` interface must provide both methods, and its objects can be converted to both interface types.

3.1.6 Implementing Multiple Interfaces

A class can implement any number of interfaces. For example, a `FileSequence` class that reads integers from a file can implement the `Closeable` interface in addition to `IntSequence`:

```
public class FileSequence implements IntSequence, Closeable {
    ...
}
```

Then the `FileSequence` class has both `IntSequence` and `Closeable` as supertypes.

3.1.7 Constants

Any variable defined in an interface is automatically `public static final`.

For example, the `SwingConstants` interface defines constants for compass directions:

```
public interface SwingConstants {
    int NORTH = 1;
    int NORTH_EAST = 2;
    int EAST = 3;
    ...
}
```

You can refer to them by their qualified name, `SwingConstants.NORTH`. If your class chooses to implement the `SwingConstants` interface, you can drop the `SwingConstants` qualifier and simply write `NORTH`. However, this is not a common idiom. It is far better to use enumerations for a set of constants; see Chapter 4.

NOTE: You cannot have instance variables in an interface. An interface specifies behavior, not object state.

3.2 Static and Default Methods

In earlier versions of Java, all methods of an interface had to be abstract—that is, without a body. Nowadays you can add two kinds of methods with a concrete implementation: static and default methods. The following sections describe these methods.

3.2.1 Static Methods

There was never a technical reason why an interface could not have static methods, but they did not fit into the view of interfaces as abstract specifications. That thinking has now evolved. In particular, factory methods make a lot of sense in interfaces. For example, the `IntSequence` interface can have a static method `digitsOf` that generates a sequence of digits of a given integer:

```
IntSequence digits = IntSequence.digitsOf(1729);
```

The method yields an instance of some class implementing the `IntSequence` interface, but the caller need not care which one it is.

```
public interface IntSequence {
    ...
    public static IntSequence digitsOf(int n) {
         return new DigitSequence(n);
    }
}
```

NOTE: In the past, it had been common to place static methods in a companion class. You find pairs of interfaces and utility classes, such as `Collection/Collections` or `Path/Paths`, in the standard library. This split is no longer necessary.

3.2.2 Default Methods

You can supply a *default* implementation for any interface method. You must tag such a method with the `default` modifier.

```
public interface IntSequence {
    default boolean hasNext() { return true; }
        // By default, sequences are infinite
    int next();
}
```

A class implementing this interface can choose to override the `hasNext` method or to inherit the default implementation.

NOTE: Default methods put an end to the classic pattern of providing an interface and a companion class that implements most or all of its methods, such as `Collection/AbstractCollection` or `WindowListener/WindowAdapter` in the Java API. Nowadays you should just implement the methods in the interface.

An important use for default methods is *interface evolution*. Consider for example the `Collection` interface that has been a part of Java for many years. Suppose that way back when, you provided a class

```
public class Bag implements Collection
```

Later, in Java 8, a `stream` method was added to the interface.

Suppose the `stream` method was not a default method. Then the `Bag` class no longer compiles since it doesn't implement the new method. Adding a nondefault method to an interface is not *source-compatible*.

But suppose you don't recompile the class and simply use an old JAR file containing it. The class will still load, even with the missing method. Programs can still construct `Bag` instances, and nothing bad will happen. (Adding a method to an interface is *binary-compatible*.) However, if a program calls the `stream` method on a `Bag` instance, an `AbstractMethodError` occurs.

Making the method a `default` method solves both problems. The `Bag` class will again compile. And if the class is loaded without being recompiled and the `stream` method is invoked on a `Bag` instance, the `Collection.stream` method is called.

3.2.3 Resolving Default Method Conflicts

If a class implements two interfaces, one of which has a default method and the other a method (default or not) with the same name and parameter types, then you must resolve the conflict. This doesn't happen very often, and it is usually easy to deal with the situation.

Let's look at an example. Suppose we have an interface `Person` with a `getId` method:

```
public interface Person {
    String getName();
    default int getId() { return 0; }
}
```

And suppose there is an interface `Identified`, also with such a method.

```
public interface Identified {
    default int getId() { return Math.abs(hashCode()); }
}
```

You will see what the `hashCode` method does in Chapter 4. For now, all that matters is that it returns some integer that is derived from the object.

What happens if you form a class that implements both of them?

```
public class Employee implements Person, Identified {
    ...
}
```

The class inherits two `getId` methods provided by the `Person` and `Identified` interfaces. There is no way for the Java compiler to choose one over the other. The compiler reports an error and leaves it up to you to resolve the ambiguity. Provide a `getId` method in the `Employee` class and either implement your own ID scheme, or delegate to one of the conflicting methods, like this:

```
public class Employee implements Person, Identified {
    public int getId() { return Identified.super.getId(); }
    ...
}
```

NOTE: The `super` keyword lets you call a supertype method. In this case, we need to specify which supertype we want. The syntax may seem a bit odd, but it is consistent with the syntax for invoking a superclass method that you will see in Chapter 4.

Now assume that the `Identified` interface does not provide a default implementation for `getId`:

```
interface Identified {
    int getId();
}
```

Can the `Employee` class inherit the default method from the `Person` interface? At first glance, this might seem reasonable. But how does the compiler know whether the `Person.getId` method actually does what `Identified.getId` is expected to do? After all, it might return the level of the person's Freudian id, not an ID number.

The Java designers decided in favor of safety and uniformity. It doesn't matter how two interfaces conflict; if at least one interface provides an implementation, the compiler reports an error, and it is up to the programmer to resolve the ambiguity.

NOTE: If neither interface provides a default for a shared method, then there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented and declare the class as `abstract`.

NOTE: If a class extends a superclass (see Chapter 4) and implements an interface, inheriting the same method from both, the rules are easier. In that case, only the superclass method matters, and any default method from the interface is simply ignored. This is actually a more common case than conflicting interfaces. See Chapter 4 for the details.

3.3 Examples of Interfaces

At first glance, interfaces don't seem to do very much. An interface is just a set of methods that a class promises to implement. To make the importance of interfaces more tangible, the following sections show you four examples of commonly used interfaces from the standard Java library.

3.3.1 The Comparable Interface

Suppose you want to sort an array of objects. A sorting algorithm repeatedly compares elements and rearranges them if they are out of order. Of course, the rules for doing the comparison are different for each class, and the sorting algorithm should just call a method supplied by the class. As long as all classes can agree on what that method is called, the sorting algorithm can do its job. That is where interfaces come in.

If a class wants to enable sorting for its objects, it should implement the `Comparable` interface. There is a technical point about this interface. We want to compare strings against strings, employees against employees, and so on. For that reason, the `Comparable` interface has a type parameter.

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

For example, the `String` class implements `Comparable<String>` so that its `compareTo` method has the signature

```
int compareTo(String other)
```

NOTE: A type with a type parameter such as `Comparable` or `ArrayList` is a *generic* type. You will learn all about generic types in Chapter 6.

When calling `x.compareTo(y)`, the `compareTo` method returns an integer value to indicate whether `x` or `y` should come first. A positive return value (not necessarily `1`) indicates that `x` should come after `y`. A negative integer (not necessarily `-1`) is returned when `x` should come before `y`. If `x` and `y` are considered equal, the returned value is `0`.

Note that the return value can be any integer. That flexibility is useful because it allows you to return a difference of non-negative integers.

```
public class Employee implements Comparable<Employee> {
    ...
    public int compareTo(Employee other) {
        return getId() - other.getId(); // Ok if IDs always ≥ 0
    }
}
```

CAUTION: Returning a difference of integers does not work if the integers can be negative. Then the difference can overflow for large operands of opposite sign. In that case, use the `Integer.compare` method that works correctly for all integers.

When comparing floating-point values, you cannot just return the difference. Instead, use the static `Double.compare` method. It does the right thing, even for ±∞ and NaN.

Here is how the `Employee` class can implement the `Comparable` interface, ordering employees by salary:

```
public class Employee implements Comparable<Employee> {
    ...
    public int compareTo(Employee other) {
        return Double.compare(salary, other.salary);
    }
}
```

NOTE: It is perfectly legal for the `compare` method to access `other.salary`. In Java, a method can access private features of *any* object of its class.

The `String` class, as well as over a hundred other classes in the Java library, implements the `Comparable` interface. You can use the `Arrays.sort` method to sort an array of `Comparable` objects:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends); // friends is now ["Mary", "Paul", "Peter"]
```

NOTE: Strangely, the `Arrays.sort` method does not check at compile time whether the argument is an array of `Comparable` objects. Instead, it throws an exception if it encounters an element of a class that doesn't implement the `Comparable` interface.

3.3.2 The `Comparator` Interface

Now suppose we want to sort strings by increasing length, not in dictionary order. We can't have the `String` class implement the `compareTo` method in two ways—and at any rate, the `String` class isn't ours to modify.

To deal with this situation, there is a second version of the `Arrays.sort` method whose parameters are an array and a *comparator*—an instance of a class that implements the `Comparator` interface.

```
public interface Comparator<T> {
    int compare(T first, T second);
}
```

To compare strings by length, define a class that implements `Comparator<String>`:

```
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

To actually do the comparison, you need to make an instance:

```
Comparator<String> comp = new LengthComparator();
if (comp.compare(words[i], words[j]) > 0) ...
```

Contrast this call with `words[i].compareTo(words[j])`. The `compare` method is called on the comparator object, not the string itself.

NOTE: Even though the `LengthComparator` object has no state, you still need to make an instance of it. You need the instance to call the `compare` method—it is not a static method.

To sort an array, pass a `LengthComparator` object to the `Arrays.sort` method:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

Now the array is either `["Paul", "Mary", "Peter"]` or `["Mary", "Paul", "Peter"]`.

You will see in Section 3.4.2, "Functional Interfaces," on p. 109 how to use a `Comparator` much more easily, using a lambda expression.

3.3.3 The `Runnable` Interface

At a time when just about every processor has multiple cores, you want to keep those cores busy. You may want to run certain tasks in a separate thread, or give them to a thread pool for execution. To define the task, you implement the `Runnable` interface. This interface has just one method.

```
class HelloTask implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("Hello, World!");
        }
    }
}
```

If you want to execute this task in a new thread, create the thread from the `Runnable` and start it.

```
Runnable task = new HelloTask();
Thread thread = new Thread(task);
thread.start();
```

Now the `run` method executes in a separate thread, and the current thread can proceed with other work.

NOTE: In Chapter 10, you will see other ways of executing a `Runnable`.

NOTE: There is also a `Callable<T>` interface for tasks that return a result of type `T`.

3.3.4 User Interface Callbacks

In a graphical user interface, you have to specify actions to be carried out when the user clicks a button, selects a menu option, drags a slider, and so on. These actions are often called *callbacks* because some code gets called back when a user action occurs.

In Java-based GUI libraries, interfaces are used for callbacks. For example, in JavaFX, the following interface is used for reporting events:

```
public interface EventHandler<T> {
    void handle(T event);
}
```

This too is a generic interface where `T` is the type of event that is being reported, such as an `ActionEvent` for a button click.

To specify the action, implement the interface:

```
class CancelAction implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        System.out.println("Oh noes!");
    }
}
```

Then, make an object of that class and add it to the button:

```
Button cancelButton = new Button("Cancel");
cancelButton.setOnAction(new CancelAction());
```

NOTE: Since Oracle positions JavaFX as the successor to the Swing GUI toolkit, I use JavaFX in these examples. (Don't worry—you need not know any more about JavaFX than the couple of statements you just saw.) The details don't matter; in every user interface toolkit, be it Swing, JavaFX, or Android, you give a button some code that you want to run when the button is clicked.

Of course, this way of defining a button action is rather tedious. In other languages, you just give the button a function to execute, without going through the detour of making a class and instantiating it. The next section shows how you can do the same in Java.

3.4 Lambda Expressions

A "lambda expression" is a block of code that you can pass around so it can be executed later, once or multiple times. In the preceding sections, you have seen many situations where it is useful to specify such a block of code:

- To pass a comparison method to `Arrays.sort`
- To run a task in a separate thread
- To specify an action that should happen when a button is clicked

However, Java is an object-oriented language where (just about) everything is an object. There are no function types in Java. Instead, functions are expressed as objects, instances of classes that implement a particular interface. Lambda expressions give you a convenient syntax for creating such instances.

3.4.1 The Syntax of Lambda Expressions

Consider again the sorting example from Section 3.3.2, "The `Comparator` Interface," on p. 104. We pass code that checks whether one string is shorter than another. We compute

```
first.length() - second.length()
```

What are `first` and `second`? They are both strings. Java is a strongly typed language, and we must specify that as well:

```
(String first, String second) -> first.length() - second.length()
```

You have just seen your first *lambda expression*. Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter lambda (λ) to mark parameters, somewhat like

```
λfirst. λsecond. first.length() - second.length()
```

NOTE: Why the letter λ? Did Church run out of letters of the alphabet? Actually, the venerable *Principia Mathematica* (see `http://plato.stanford.edu/entries/principia-mathematica`) used the ^ accent to denote function parameters, which inspired Church to use an uppercase lambda Λ. But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

If the body of a lambda expression carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method: enclosed in {} and with explicit `return` statements. For example,

```
(String first, String second) -> {
   int difference = first.length() < second.length();
   if (difference < 0) return -1;
   else if (difference > 0) return 1;
   else return 0;
}
```

If a lambda expression has no parameters, supply empty parentheses, just as with a parameterless method:

```
Runnable task = () -> { for (int i = 0; i < 1000; i++) doWork(); }
```

If the parameter types of a lambda expression can be inferred, you can omit them. For example,

```
Comparator<String> comp
    = (first, second) -> first.length() - second.length();
    // Same as (String first, String second)
```

Here, the compiler can deduce that `first` and `second` must be strings because the lambda expression is assigned to a string comparator. (We will have a closer look at this assignment in the next section.)

If a method has a single parameter with inferred type, you can even omit the parentheses:

```
EventHandler<ActionEvent> listener = event ->
    System.out.println("Oh noes!");
        // Instead of (event) -> or (ActionEvent event) ->
```

You never specify the result type of a lambda expression. However, the compiler infers it from the body and checks that it matches the expected type. For example, the expression

```
(String first, String second) -> first.length() - second.length()
```

can be used in a context where a result of type `int` is expected (or a compatible type such as `Integer`, `long`, or `double`).

3.4.2 Functional Interfaces

As you already saw, there are many interfaces in Java that express actions, such as `Runnable` or `Comparator`. Lambda expressions are compatible with these interfaces.

You can supply a lambda expression whenever an object of an interface with a *single abstract method* is expected. Such an interface is called a *functional interface*.

To demonstrate the conversion to a functional interface, consider the `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

```
Arrays.sort(words,
    (first, second) -> first.length() - second.length());
```

Behind the scenes, the second parameter variable of the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation-dependent and highly optimized.

In most programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, put functions into those variables, and invoke them. In Java, there is *only one thing* you can do with a lambda expression: put it in a variable whose type is a functional interface, so that it is converted to an instance of that interface.

NOTE: You cannot assign a lambda expression to a variable of type `Object`, the common supertype of all classes in Java (see Chapter 4). `Object` is a class, not a functional interface.

The standard library provides a large number of functional interfaces (see Section 3.6.2, "Choosing a Functional Interface," on p. 113). One of them is

```
public interface Predicate<T> {
    boolean test(T t);
    // Additional default and static methods
}
```

The ArrayList class has a removeIf method whose parameter is a Predicate. It is specifically designed to pass a lambda expression. For example, the following statement removes all null values from an array list:

```
list.removeIf(e -> e == null);
```

3.5 Method and Constructor References

Sometimes, there is already a method that carries out exactly the action that you'd like to pass on to some other code. There is special syntax for a *method reference* that is even shorter than a lambda expression calling the method. A similar shortcut exists for constructors. You will see both in the following sections.

3.5.1 Method References

Suppose you want to sort strings regardless of letter case. You could call

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));
```

Instead, you can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase);
```

The expression String::compareToIgnoreCase is a *method reference* that is equivalent to the lambda expression (x, y) -> x.compareToIgnoreCase(y).

Here is another example. The Objects class defines a method isNull. The call Objects.isNull(x) simply returns the value of x == null. It seems hardly worth having a method for this case, but it was designed to be passed as a method expression. The call

```
list.removeIf(Objects::isNull);
```

removes all null values from a list.

As another example, suppose you want to print all elements of a list. The ArrayList class has a method forEach that applies a function to each element. You could call

```
list.forEach(x -> System.out.println(x));
```

It would be nicer, however, if you could just pass the println method to the forEach method. Here is how to do that:

```
list.forEach(System.out::println);
```

As you can see from these examples, the :: operator separates the method name from the name of a class or object. There are three variations:

1. *Class*::*instanceMethod*
2. *Class*::*staticMethod*
3. *object*::*instanceMethod*

In the first case, the first parameter becomes the receiver of the method, and any other parameters are passed to the method. For example, `String::compareToIgnoreCase` is the same as `(x, y) -> x.compareToIgnoreCase(y)`.

In the second case, all parameters are passed to the static method. The method expression `Objects::isNull` is equivalent to `x -> Objects.isNull(x)`.

In the third case, the method is invoked on the given object, and the parameters are passed to the instance method. Therefore, `System.out::println` is equivalent to `x -> System.out.println(x)`.

NOTE: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are multiple versions of the `println` method. When passed to the `forEach` method of an `ArrayList<String>`, the `println(String)` method is picked.

You can capture the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`.

NOTE: In an inner class, you can capture the `this` reference of an enclosing class as *EnclosingClass*`.this::`*method*. You can also capture `super`—see Chapter 4.

3.5.2 Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Employee::new` is a reference to an `Employee` constructor. If the class has more than one constructor, then it depends on the context which constructor is chosen.

Here is an example for using such a constructor reference. Suppose you have a list of strings

```
List<String> names = ...;
```

You want a list of employees, one for each name. As you will see in Chapter 8, you can use streams to do this without a loop: Turn the list into a stream, and then call the `map` method. It applies a function and collects all results.

```
Stream<Employee> stream = names.stream().map(Employee::new);
```

Since `names.stream()` contains `String` objects, the compiler knows that `Employee::new` refers to the constructor `Employee(String)`.

You can form constructor references with array types. For example, `int[]::new` is a constructor reference with one parameter: the length of the array. It is equivalent to the lambda expression `n -> new int[n]`.

Array constructor references are useful to overcome a limitation of Java: It is not possible to construct an array of a generic type. (See Chapter 6 for details.) For that reason, methods such `Stream.toArray` return an `Object` array, not an array of the element type:

```
Object[] employees = stream.toArray();
```

But that is unsatisfactory. The user wants an array of employees, not objects. To solve this problem, another version of `toArray` accepts a constructor reference:

```
Employee[] buttons = stream.toArray(Employee[]::new);
```

The `toArray` method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.

3.6 Processing Lambda Expressions

Up to now, you have seen how to produce lambda expressions and pass them to a method that expects a functional interface. In the following sections, you will see how to write your own methods that can consume lambda expressions.

3.6.1 Implementing Deferred Execution

The point of using 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

Let's look at a simple example. Suppose you want to repeat an action `n` times. The action and the count are passed to a `repeat` method:

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

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. In this case, we can just use `Runnable`:

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

Note that the body of the lambda expression is executed when `action.run()` is called.

Now let's make this example a bit more sophisticated. We want to tell the action in which iteration it occurs. For that, we need to pick a functional interface that has a method with an `int` parameter and a `void` return. Instead of rolling your own, I strongly recommend that you use one of the standard ones described in the next section. The standard interface for processing `int` values is

```
public interface IntConsumer {
    void accept(int value);
}
```

Here is the improved version of the `repeat` method:

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

And here is how you call it:

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

3.6.2 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.

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`	`andThen`
`BiConsumer<T, U>`	`T, U`	`void`	`accept`	Consumes values of types `T` and `U`	`andThen`
`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`, `maxBy`, `minBy`
`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`

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`, but it also works if `a` is `null`. 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)`.

Table 3–2 lists the 34 available specializations for primitive types `int`, `long`, and `double`. It is a good idea to use these specializations to reduce autoboxing. For that reason, I used an `IntConsumer` instead of a `Consumer<Integer>` in the example of the preceding section.

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`

3.6.3 Implementing Your Own Functional Interfaces

Ever so often, you will be in a situation where none of the standard functional interfaces work for you. Then you need to roll your own.

Suppose you want to fill an image with color patterns, where the user supplies a function yielding the color for each pixel. There is no standard type for a mapping `(int, int) -> Color`. You could use `BiFunction<Integer, Integer, Color>`, but that involves autoboxing.

In this case, it makes sense to define a new interface

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

NOTE: You should tag functional interfaces with the `@FunctionalInterface` annotation. This has two advantages. First, the compiler checks that the annotated entity is an interface with a single abstract method. Second, the javadoc page includes a statement that your interface is a functional interface.

Now you are ready to implement a method:

```
BufferedImage createImage(int width, int height, PixelFunction f) {
    BufferedImage image = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_RGB);

    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            Color color = f.apply(x, y);
            image.setRGB(x, y, color.getRGB());
        }
    return image;
}
```

To call it, supply a lambda expression that yields a color value for two integers:

```
BufferedImage frenchFlag = createImage(150, 100,
    (x, y) -> x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```

3.7 Lambda Expressions and Variable Scope

In the following sections, you will learn how variables work inside lambda expressions. This information is somewhat technical but essential for working with lambda expressions.

3.7.1 Scope of a Lambda Expression

The body of a lambda expression has *the same scope as a nested block*. The same rules for name conflicts and shadowing apply. It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
int first = 0;
Comparator<String> comp = (first, second) -> first.length() - second.length();
    // Error: Variable first already defined
```

Inside a method, you can't have two local variables with the same name, therefore you can't introduce such variables in a lambda expression either.

As another consequence of the "same scope" rule, the `this` keyword in a lambda expression denotes the `this` parameter of the method that creates the lambda. For example, consider

```
public class Application() {
    public void doWork() {
        Runnable runner = () -> { ...; System.out.println(this.toString()); ... };
        ...
    }
}
```

The expression `this.toString()` calls the `toString` method of the `Application` object, *not* the `Runnable` instance. There is nothing special about the use of `this` in a lambda expression. The scope of the lambda expression is nested inside the `doWork` method, and `this` has the same meaning anywhere in that method.

3.7.2 Accessing Variables from the Enclosing Scope

Often, you want to access variables from an enclosing method or class in a lambda expression. Consider this example:

```
public static void repeatMessage(String text, int count) {
    Runnable r = () -> {
        for (int i = 0; i < count; i++) {
            System.out.println(text);
        }
    };
    new Thread(r).start();
}
```

Note that the lambda expression accesses the parameter variables defined in the enclosing scope, not in the lambda expression itself.

Consider a call

```
repeatMessage("Hello", 1000); // Prints Hello 1000 times in a separate thread
```

Now look at the variables `count` and `text` inside the lambda expression. If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the

parameter variables are gone. How do the `text` and `count` variables stay around when the lambda expression is ready to execute?

To understand what is happening, we need to refine our understanding of a lambda expression. A lambda expression has three ingredients:

1. A block of code
2. Parameters
3. Values for the *free* variables—that is, the variables that are not parameters and not defined inside the code

In our example, the lambda expression has two free variables, `text` and `count`. The data structure representing the lambda expression must store the values for these variables—in our case, `"Hello"` and `1000`. We say that these values have been *captured* by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)

NOTE: The technical term for a block of code together with the values of free variables is a *closure*. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. To ensure that the captured value is well defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. This is sometimes described by saying that lambda expressions capture values, not variables. For example, the following is a compile-time error:

```
for (int i = 0; i < n; i++) {
    new Thread(() -> System.out.println(i)).start();
        // Error—cannot capture i
}
```

The lambda expression tries to capture `i`, but this is not legal because `i` changes. There is no single value to capture. The rule is that a lambda expression can only access local variables from an enclosing scope that are *effectively final*. An effectively final variable is never modified—it either is or could be declared as `final`.

NOTE: The same rule applies to variables captured by local inner classes (see Section 3.9, "Local Inner Classes," on p. 122). In the past, the rule was more draconian and required captured variables to actually be declared `final`. This is no longer the case.

NOTE: The variable of an enhanced `for` loop is effectively final since its scope is a single iteration. The following is perfectly legal:

```
for (String arg : args) {
    new Thread(() -> System.out.println(arg)).start();
        // OK to capture arg
}
```

A new variable `arg` is created in each iteration and assigned the next value from the `args` array. In contrast, the scope of the variable `i` in the preceding example was the entire loop.

As a consequence of the "effectively final" rule, a lambda expression cannot mutate any captured variables. For example,

```
public static void repeatMessage(String text, int count, int threads) {
    Runnable r = () -> {
        while (count > 0) {
            count--; // Error: Can't mutate captured variable
            System.out.println(text);
        }
    };
    for (int i = 0; i < threads; i++) new Thread(r).start();
}
```

This is actually a good thing. As you will see in Chapter 10, if two threads update `count` at the same time, its value is undefined.

NOTE: Don't count on the compiler to catch all concurrent access errors. The prohibition against mutation only holds for local variables. If `count` is an instance variable or static variable of an enclosing class, then no error is reported even though the result is just as undefined.

CAUTION: One can circumvent the check for inappropriate mutations by using an array of length 1:

```
int[] counter = new int[1];
button.setOnAction(event -> counter[0]++);
```

The `counter` variable is effectively final—it is never changed since it always refers to the same array, so you can access it in the lambda expression.

Of course, code like this is not threadsafe. Except possibly for a callback in a single-threaded UI, this is a terrible idea. You will see how to implement a threadsafe shared counter in Chapter 10.

3.8 Higher-Order 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. Functions that process or return functions are called *higher-order 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. In the following sections, we will look at some examples and examine the higher-order functions in the `Comparator` interface.

3.8.1 Methods that Return Functions

Suppose sometimes we want to sort an array of strings in ascending order and other times in descending order. We can make a method that produces the correct comparator:

```
public static Comparator<String> compareInDirecton(int direction) {
    return (x, y) -> direction * x.compareTo(y);
}
```

The call `compareInDirection(1)` yields an ascending comparator, and the call `compareInDirection(-1)` a descending comparator.

The result can be passed to another method (such as `Arrays.sort`) that expects such an interface.

```
Arrays.sort(friends, compareInDirection(-1));
```

In general, don't be shy to write methods that produce functions (or, technically, instances of classes that implement a functional interface). This is useful to generate custom functions that you pass to methods with functional interfaces.

3.8.2 Methods That Modify Functions

In the preceding section, you saw a method that yields an increasing or decreasing string comparator. We can generalize this idea by reversing any comparator:

```
public static Comparator<String> reverse(Comparator<String> comp) {
    return (x, y) -> comp.compare(x, y);
}
```

This method operates on functions. It receives a function and returns a modified function. To get case-insensitive descending order, use

```
reverse(String::compareToIgnoreCase)
```

NOTE: The `Comparator` interface has a default method `reversed` that produces the reverse of a given comparator in just this way.

3.8.3 Comparator Methods

The `Comparator` interface has a number of useful static methods that are higher-order functions generating comparators.

The `comparing` method takes a "key extractor" function that maps a type `T` to a comparable type (such as `String`). The function is applied to the objects to be compared, and the comparison is then made on the returned keys. For example, suppose you have an array of `Person` objects. Here is how you can sort them by name:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

You can chain comparators with the `thenComparing` method for breaking ties. For example,

```
Arrays.sort(people, Comparator
    .comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

If two people have the same last name, then the second comparator is used.

There are a few variations of these methods. You can specify a comparator to be used for the keys that the `comparing` and `thenComparing` methods extract. For example, here we sort people by the length of their names:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    (s, t) -> s.length() - t.length()));
```

Moreover, both the `comparing` and `thenComparing` methods have variants that avoid boxing of `int`, `long`, or `double` values. An easier way of sorting by name length would be

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

If your key function can return `null`, you will like the `nullsFirst` and `nullsLast` adapters. These static methods take an existing comparator and modify it so that it doesn't throw an exception when encountering `null` values but ranks them as smaller or larger than regular values. For example, suppose `getMiddleName` returns a `null` when a person has no middle name. Then you can use `Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...))`.

The `nullsFirst` method needs a comparator—in this case, one that compares two strings. The `naturalOrder` method makes a comparator for any class implementing `Comparable`. Here is the complete call for sorting by potentially null middle names.

I use a static import of `java.util.Comparator.*` to make the expression more legible. Note that the type for `naturalOrder` is inferred.

```
Arrays.sort(people, comparing(Person::getMiddleName,
    nullsFirst(naturalOrder())));
```

The static `reverseOrder` method gives the reverse of the natural order.

3.9 Local Inner Classes

Long before there were lambda expressions, Java had a mechanism for concisely defining classes that implement an interface (functional or not). For functional interfaces, you should definitely use lambda expressions, but once in a while, you may want a concise form for an interface that isn't functional. You will also encounter the classic constructs in legacy code.

3.9.1 Local Classes

You can define a class inside a method. Such a class is called a *local class*. You would do this for classes that are just tactical. This occurs often when a class implements an interface and the caller of the method only cares about the interface, not the class.

For example, consider a method

```
public static IntSequence randomInts(int low, int high)
```

that generates an infinite sequence of random integers with the given bounds.

Since `IntSequence` is an interface, the method must return an object of some class implementing that interface. The caller doesn't care about the class, so it can be declared inside the method:

```
private static Random generator = new Random();

public static IntSequence randomInts(int low, int high) {
    class RandomSequence implements IntSequence {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }

    return new RandomSequence();
}
```

NOTE: A local class is not declared as `public` or `private` since it is never accessible outside the method.

There are two advantages of making a class local. First, its name is hidden in the scope of the method. Second, the methods of the class can access variables from the enclosing scope, just like the variables of a lambda expression.

In our example, the `next` method captures three variables: `low`, `high`, and `generator`. If you turned `RandomInt` into a nested class, you would have to provide an explicit constructor that receives these values and stores them in instance variables (see Exercise 15).

3.9.2 Anonymous Classes

In the example of the preceding section, the name `RandomSequence` was used exactly once: to construct the return value. In this case, you can make the class *anonymous*:

```
public static IntSequence randomInts(int low, int high) {
    return new IntSequence() {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }
}
```

The expression

```
new Interface() { methods }
```

means: Define a class implementing the interface that has the given methods, and construct one object of that class.

NOTE: As always, the `()` in the `new` expression indicate the construction arguments. A default constructor of the anonymous class is invoked.

Before Java had lambda expressions, anonymous inner classes were the most concise syntax available for providing runnables, comparators, and other functional objects. You will often see them in legacy code.

Nowadays, they are only necessary when you need to provide two or more methods, as in the preceding example. If the `IntSequence` interface has a default `hasNext` method, as in Exercise 15, you can simply use a lambda expression:

```
public static IntSequence randomInts(int low, int high) {
    return () -> low + generator.nextInt(high - low + 1);
}
```

Exercises

1. Provide an interface `Measurable` with a method `double getMeasure()` that measures an object in some way. Make `Employee` implement `Measurable`. Provide a method `double average(Measurable[] objects)` that computes the average measure. Use it to compute the average salary of an array of employees.
2. Continue with the preceding exercise and provide a method `Measurable largest(Measurable[] objects)`. Use it to find the name of the employee with the largest salary. Why do you need a cast?
3. What are all the supertypes of `String`? Of `Scanner`? Of `ImageOutputStream`? Note that each type is its own supertype. A class or interface without declared supertype has supertype `Object`.
4. Implement a static `of` method of the `IntSequence` class that yields a sequence with the arguments. For example, `IntSequence.of(3, 1, 4, 1, 5, 9)` yields a sequence with six values. Extra credit if you return an instance of an anonymous inner class.
5. Implement a static `constant` method of the `IntSequence` class that yields an infinite constant sequence. For example, `IntSequence.constant(1)` yields values `1 1 1 . . .`, ad infinitum. Extra credit if you do this with a lambda expression.
6. In this exercise, you will try out what happens when a method is added to an interface. In Java 7, implement a class `DigitSequence` that implements `Iterator<Integer>`, not `IntSequence`. Provide methods `hasNext`, `next`, and a do-nothing `remove`. Write a program that prints the elements of an instance. In Java 8, the `Iterator` class gained another method, `forEachRemaining`. Does your code still compile when you switch to Java 8? If you put your Java 7 class in a JAR file and don't recompile, does it work in Java 8? What if you call the `forEachRemaining` method? Also, the `remove` method has become a default method in Java 8, throwing an `UnsupportedOperationException`. What happens when `remove` is called on an instance of your class?
7. Implement the method `void luckySort(ArrayList<String> strings, Comparator<String> comp)` that keeps calling `Collections.shuffle` on the array list until the elements are in increasing order, as determined by the comparator.
8. Implement a class `Greeter` that implements `Runnable` and whose `run` method prints `n` copies of `"Hello, " + target`, where `n` and `target` are set in the constructor. Construct two instances with different messages and execute them concurrently in two threads.
9. Implement methods

```
public static void runTogether(Runnable... tasks)
public static void runInOrder(Runnable... tasks)
```

The first method should run each task in a separate thread and then return. The second method should run all methods in the current thread and return when the last one has completed.

10. Using the `listFiles(FileFilter)` and `isDirectory` methods of the `java.io.File` class, write a method that returns all subdirectories of a given directory. Use a lambda expression instead of a `FileFilter` object. Repeat with a method expression and an anonymous inner class.
11. Using the `list(FilenameFilter)` method of the `java.io.File` class, write a method that returns all files in a given directory with a given extension. Use a lambda expression, not a `FilenameFilter`. Which variable from the enclosing scope does it capture?
12. Given an array of `File` objects, sort it so that directories come before files, and within each group, elements are sorted by path name. Use a lambda expression to specify the `Comparator`.
13. Write a method that takes an array of `Runnable` instances and returns a `Runnable` whose `run` method executes them in order. Return a lambda expression.
14. Write a call to `Arrays.sort` that sorts employees by salary, breaking ties by name. Use `Comparator.thenComparing`. Then do this in reverse order.
15. Implement the `RandomSequence` in Section 3.9.1, "Local Classes," on p. 122 as a nested class, outside the `randomInts` method.

Index

E

F

G

M

O

P

S

T

U

V

W