Contents

CHAPTER 1

Streams

In this chapter

Compared to collections, streams provide a view of data that lets you specify computations at a higher conceptual level. With a stream, you specify what you want to have done, not how to do it. You leave the scheduling of operations to the implementation. For example, suppose you want to compute the average of a certain property. You specify the source of data and the property, and the stream library can then optimize the computation, for example by using multiple threads for computing sums and counts and combining the results.

In this chapter, you will learn how to use the Java stream library, which was introduced in Java 8, to process collections in a "what, not how" style.

1.1 From Iterating to Stream Operations

When you process a collection, you usually iterate over its elements and do some work with each of them. For example, suppose we want to count all long words in a book. First, let's put them into a list:

```
var contents = Files.readStr(
   Paths.get("alice.txt")); // Read file into string
List<String> words = List.of(contents.split("\\PL+"));
   // Split into words; nonletters are delimiters
```

Now we are ready to iterate:

```
int count = 0;
for (String w : words) {
   if (w.length() > 12) count++;
}
```

With streams, the same operation looks like this:

```
long count = words.stream()
   .filter(w -> w.length() > 12)
   .count();
```

Now you don't have to scan the loop for evidence of filtering and counting. The method names tell you right away what the code intends to do. Moreover, where the loop prescribes the order of operations in complete detail, a stream is able to schedule the operations any way it wants, as long as the result is correct.

Simply changing `stream` to `parallelStream` allows the stream library to do the filtering and counting in parallel.

```
long count = words.parallelStream()
   .filter(w -> w.length() > 12)
   .count();
```

Streams follow the "what, not how" principle. In our stream example, we describe what needs to be done: get the long words and count them. We don't specify in which order, or in which thread, this should happen. In contrast, the loop at the beginning of this section specifies exactly how the computation should work, and thereby forgoes any chances of optimization.

A stream seems superficially similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

1. A stream does not store its elements. They may be stored in an underlying collection or generated on demand.
2. Stream operations don't mutate their source. For example, the `filter` method does not remove elements from a stream but yields a new stream in which they are not present.
3. Stream operations are *lazy* when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of all, the `filter` method will stop filtering after the fifth match. As a consequence, you can even have infinite streams!

Let us have another look at the example. The `stream` and `parallelStream` methods yield a *stream* for the `words` list. The `filter` method returns another stream that contains only the words of length greater than twelve. The `count` method reduces that stream to a result.

This workflow is typical when you work with streams. You set up a pipeline of operations in three stages:

1. Create a stream.
2. Specify *intermediate operations* for transforming the initial stream into others, possibly in multiple steps.
3. Apply a *terminal operation* to produce a result. This operation forces the execution of the lazy operations that precede it. Afterwards, the stream can no longer be used.

In the example in Listing 1.1, the stream is created with the `stream` or `parallelStream` method. The `filter` method transforms it, and `count` is the terminal operation.

In the next section, you will see how to create a stream. The subsequent three sections deal with stream transformations. They are followed by five sections on terminal operations.

Listing 1.1 `streams/CountLongWords.java`

```java
package streams;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;

public class CountLongWords
{
   public static void main(String[] args) throws IOException
   {
      var contents = Files.readStr(
         Paths.get("../gutenberg/alice30.txt"));
      List<String> words = List.of(contents.split("\\PL+"));

      long count = 0;
      for (String w : words)
      {
         if (w.length() > 12) count++;
      }
      System.out.println(count);

      count = words.stream().filter(w -> w.length() > 12).count();
      System.out.println(count);

      count = words.parallelStream().filter(w -> w.length() > 12).count();
      System.out.println(count);
   }
}
```

java.util.stream.Stream<T> 8

- `Stream<T> filter(Predicate<? super T> p)`

  yields a stream containing all elements of this stream fulfilling `p`.
- `long count()`

  yields the number of elements of this stream. This is a terminal operation.

java.util.Collection<E> 1.2

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

  yields a sequential or parallel stream of the elements in this collection.

1.2 Stream Creation

You have already seen that you can turn any collection into a stream with the `stream` method of the `Collection` interface. If you have an array, use the static `Stream.of` method instead.

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
   // split returns a String[] array
```

The `of` method has a varargs parameter, so you can construct a stream from any number of arguments:

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Use `Arrays.stream(array, from, to)` to make a stream from a part of an array.

To make a stream with no elements, use the static `Stream.empty` method:

```
Stream<String> silence = Stream.empty();
   // Generic type <String> is inferred; same as Stream.<String>empty()
```

The `Stream` interface has two static methods for making infinite streams. The `generate` method takes a function with no arguments (or, technically, an object of the `Supplier<T>` interface). Whenever a stream value is needed, that function is called to produce a value. You can get a stream of constant values as

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

or a stream of random numbers as

```
Stream<Double> randoms = Stream.generate(Math::random);
```

To produce sequences such as `0 1 2 3 . . .`, use the `iterate` method instead. It takes a "seed" value and a function (technically, a `UnaryOperator<T>`) and repeatedly applies the function to the previous result. For example,

```
Stream<BigInteger> integers
   = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

The first element in the sequence is the seed `BigInteger.ZERO`. The second element is `f(seed)` which yields `1` (as a big integer). The next element is `f(f(seed))` which yields `2`, and so on.

To produce a finite stream instead, add a predicate that specifies when the iteration should finish:

```
var limit = new BigInteger("10000000");
Stream<BigInteger> integers
   = Stream.iterate(BigInteger.ZERO,
      n -> n.compareTo(limit) < 0,
      n -> n.add(BigInteger.ONE));
```

As soon as the predicate rejects an iteratively generated value, the stream ends.

Finally, the `Stream.ofNullable` method makes a really short stream from an object. The stream has length 0 if the object is `null` or length 1 otherwise, containing just the object. This is mostly useful in conjunction with `flatMap`—see Section 1.7.7, "Turning an Optional into a Stream," on p. 22 for an example.

NOTE: A number of methods in the Java API yield streams. For example, the `Pattern` class has a method `splitAsStream` that splits a `CharSequence` by a regular expression. You can use the following statement to split a string into words:

```
Stream<String> words = Pattern.compile("\\PL+").splitAsStream(contents);
```

The `Scanner.tokens` method yields a stream of tokens of a scanner. Another way to get a stream of words from a string is

```
Stream<String> words = new Scanner(contents).tokens();
```

The static `Files.lines` method returns a `Stream` of all lines in a file:

```
try (Stream<String> lines = Files.lines(path)) {
   Process lines
}
```

NOTE: If you have an `Iterable` that is not a collection, you can turn it into a stream by calling

```
StreamSupport.stream(iterable.spliterator(), false);
```

If you have an `Iterator` and want a stream of its results, use

```
StreamSupport.stream(Spliterators.spliteratorUnknownSize(
   iterator, Spliterator.ORDERED), false);
```

CAUTION: It is very important that you don't modify the collection backing a stream while carrying out a stream operation. Remember that streams don't collect their data—the data is always in a separate collection. If you modify that collection, the outcome of the stream operations becomes undefined. The JDK documentation refers to this requirement as *noninterference*.

To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point where the terminal operation executes. For example, the following, while certainly not recommended, will work:

```
List<String> wordList = . . .;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

But this code is wrong:

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
   // ERROR--interference
```

The example program in Listing 1.2 shows the various ways of creating a stream.

Listing 1.2 `streams/CreatingStreams.java`

```
package streams;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import java.io.IOException;
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
import java.util.regex.Pattern;
import java.util.stream.*;

public class CreatingStreams
{
   public static <T> void show(String title, Stream<T> stream)
   {
```

(Continues)

Listing 1.2 *(Continued)*

```
      final int SIZE = 10;
      List<T> firstElements = stream
         .limit(SIZE + 1)
         .collect(Collectors.toList());
      System.out.print(title + ": ");
      for (int i = 0; i < firstElements.size(); i++)
      {
         if (i > 0) System.out.print(", ");
         if (i < SIZE) System.out.print(firstElements.get(i));
         else System.out.print("...");
      }
      System.out.println();
   }

   public static void main(String[] args) throws IOException
   {
      Path path = Paths.get("../gutenberg/alice30.txt");
      var contents = Files.readStr(path);

      Stream<String> words = Stream.of(contents.split("\\PL+"));
      show("words", words);
      Stream<String> song = Stream.of("gently", "down", "the", "stream");
      show("song", song);
      Stream<String> silence = Stream.empty();
      show("silence", silence);

      Stream<String> echos = Stream.generate(() -> "Echo");
      show("echos", echos);

      Stream<Double> randoms = Stream.generate(Math::random);
      show("randoms", randoms);

      Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
         n -> n.add(BigInteger.ONE));
      show("integers", integers);

      Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStream(contents);
      show("wordsAnotherWay", wordsAnotherWay);

      try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8))
      {
         show("lines", lines);
      }

      Iterable<Path> iterable = FileSystems.getDefault().getRootDirectories();
      Stream<Path> rootDirectories = StreamSupport.stream(iterable.spliterator(), false);
      show("rootDirectories", rootDirectories);
```

```java

      Iterator<Path> iterator = Paths.get("/usr/share/dict/words").iterator();
      Stream<Path> pathComponents = StreamSupport.stream(Spliterators.spliteratorUnknownSize(
         iterator, Spliterator.ORDERED), false);
      show("pathComponents", pathComponents);
   }
}
```

java.util.stream.Stream 8

- `static <T> Stream<T> of(T... values)`

  yields a stream whose elements are the given values.
- `static <T> Stream<T> empty()`

  yields a stream with no elements.
- `static <T> Stream<T> generate(Supplier<T> s)`

  yields an infinite stream whose elements are constructed by repeatedly invoking the function `s`.
- `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`
- `static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> f)`

  yields a stream whose elements are `seed`, `f` invoked on `seed`, `f` invoked on the preceding element, and so on. The first method yields an infinite stream. The stream of the second method comes to an end before the first element that doesn't fulfill the `hasNext` predicate.
- `static <T> Stream<T> ofNullable(T t)` 9

  returns an empty stream if `t` is `null` or a stream containing `t` otherwise.

java.util.Spliterators 8

- `static <T> Spliterator<T> spliteratorUnknownSize(Iterator<? extends T> iterator, int characteristics)`

  turns an iterator into a splittable iterator of unknown size with the given characteristics (a bit pattern containing constants such as `Spliterator.ORDERED`).

java.util.Arrays 1.2

- `static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)` 8

  yields a stream whose elements are the specified range of the array.

java.util.regex.Pattern 1.4

- `Stream<String> splitAsStream(CharSequence input)` 8

  yields a stream whose elements are the parts of the input that are delimited by this pattern.

java.nio.file.Files 7

- `static Stream<String> lines(Path path)` 8
- `static Stream<String> lines(Path path, Charset cs)` 8

  yields a stream whose elements are the lines of the specified file, with the UTF-8 charset or the given charset.

java.util.stream.StreamSupport 8

- `static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel)`

  yields a stream containing the values produced by the given splittable iterator.

java.lang.Iterable 5

- `Spliterator<T> spliterator()` 8

  yields a splittable iterator for this `Iterable`. The default implementation does not split and does not report a size.

java.util.Scanner 5

- `public Stream<String> tokens()` 9

  yields a stream of strings returned by calling the `next` method of this scanner.

java.util.function.Supplier<T> 8

- `T get()`

  supplies a value.

1.3 The filter, map, and flatMap Methods

A stream transformation produces a stream whose elements are derived from those of another stream. You have already seen the filter transformation that yields a new stream with those elements that match a certain condition. Here, we transform a stream of strings into another stream containing only long words:

```
List<String> words = . . .;
Stream<String> longWords = words.stream().filter(w -> w.length() > 12);
```

The argument of filter is a Predicate<T>—that is, a function from T to boolean.

Often, you want to transform the values in a stream in some way. Use the map method and pass the function that carries out the transformation. For example, you can transform all words to lowercase like this:

```
Stream<String> lowercaseWords = words.stream().map(String::toLowerCase);
```

Here, we used map with a method reference. Often, you will use a lambda expression instead:

```
Stream<String> firstLetters = words.stream().map(s -> s.substring(0, 1));
```

The resulting stream contains the first letter of each word.

When you use map, a function is applied to each element, and the result is a new stream with the results. Now, suppose you have a function that returns not just one value but a stream of values. Here is an example—a method that turns a string into a stream of strings, namely the individual code points:

```
public static Stream<String> codePoints(String s)
{
   var result = new ArrayList<String>();
   int i = 0;
   while (i < s.length())
   {
      int j = s.offsetByCodePoints(i, 1);
      result.add(s.substring(i, j));
      i = j;
   }
   return result.stream();
}
```

This method correctly handles Unicode characters that require two char values because that's the right thing to do. But you don't have to dwell on that.

For example, codePoints("boat") is the stream ["b", "o", "a", "t"].

Now let's map the `codePoints` method on a stream of strings:

```
Stream<Stream<String>> result = words.stream().map(w -> codePoints(w));
```

You will get a stream of streams, like `[. . . ["y", "o", "u", "r"], ["b", "o", "a", "t"], . . .]`. To flatten it out to a single stream `[. . . "y", "o", "u", "r", "b", "o", "a", "t", . . .]`, use the `flatMap` method instead of `map`:

```
Stream<String> flatResult = words.stream().flatMap(w -> codePoints(w));
   // Calls codePoints on each word and flattens the results
```

NOTE: You will find a `flatMap` method in classes other than streams. It is a general concept in computer science. Suppose you have a generic type `G` (such as `Stream`) and functions `f` from some type `T` to `G<U>` and `g` from `U` to `G<V>`. Then you can compose them—that is, first apply `f` and then `g`, by using `flatMap`. This is a key idea in the theory of *monads*. But don't worry—you can use `flatMap` without knowing anything about monads.

java.util.stream.Stream 8

- `Stream<T> filter(Predicate<? super T> predicate)`

  yields a stream containing the elements of this stream that fulfill the predicate.
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`

  yields a stream containing the results of applying `mapper` to the elements of this stream.
- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`

  yields a stream obtained by concatenating the results of applying `mapper` to the elements of this stream. (Note that each result is a stream.)

1.4 Extracting Substreams and Combining Streams

The call *stream*`.limit(n)` returns a new stream that ends after `n` elements (or when the original stream ends if it is shorter). This method is particularly useful for cutting infinite streams down to size. For example,

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

yields a stream with 100 random numbers.

The call *stream*`.skip(n)` does the exact opposite. It discards the first `n` elements. This is handy in our book reading example where, due to the way the `split` method works, the first element is an unwanted empty string. We can make it go away by calling `skip`:

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

The *stream*.takeWhile(*predicate*) call takes all elements from the stream while the predicate is true, and then stops.

For example, suppose we use the codePoints method of the preceding section to split a string into characters, and we want to collect all initial digits. The takeWhile method can do this:

```
Stream<String> initialDigits = codePoints(str).takeWhile(
   s -> "0123456789".contains(s));
```

The dropWhile method does the opposite, dropping elements while a condition is true and yielding a stream of all elements starting with the first one for which the condition was false. For example,

```
Stream<String> withoutInitialWhiteSpace = codePoints(str).dropWhile(
   s -> s.trim().length() == 0);
```

You can concatenate two streams with the static concat method of the Stream class:

```
Stream<String> combined = Stream.concat(
   codePoints("Hello"), codePoints("World"));
   // Yields the stream ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

Of course, the first stream should not be infinite—otherwise the second one wouldn't ever get a chance.

java.util.stream.Stream 8

- Stream<T> limit(long maxSize)

  yields a stream with up to maxSize of the initial elements from this stream.
- Stream<T> skip(long n)

  yields a stream whose elements are all but the initial n elements of this stream.
- Stream<T> takeWhile(Predicate<? super T> predicate) 9

  yields a stream whose elements are the initial elements of this stream that fulfill the predicate.
- Stream<T> dropWhile(Predicate<? super T> predicate) 9

  yields a stream whose elements are the elements of this stream except for the initial ones that do not fulfill the predicate.
- static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)

  yields a stream whose elements are the elements of a followed by the elements of b.

1.5 Other Stream Transformations

The distinct method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The duplicates need not be adjacent.

```
Stream<String> uniqueWords
   = Stream.of("merrily", "merrily", "merrily", "gently").distinct();
   // Only one "merrily" is retained
```

For sorting a stream, there are several variations of the sorted method. One works for streams of Comparable elements, and another accepts a Comparator. Here, we sort strings so that the longest ones come first:

```
Stream<String> longestFirst
   = words.stream().sorted(Comparator.comparing(String::length).reversed());
```

As with all stream transformations, the sorted method yields a new stream whose elements are the elements of the original stream in sorted order.

Of course, you can sort a collection without using streams. The sorted method is useful when the sorting process is part of a stream pipeline.

Finally, the peek method yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved. That is handy for debugging:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
   .peek(e -> System.out.println("Fetching " + e))
   .limit(20).toArray();
```

When an element is actually accessed, a message is printed. This way you can verify that the infinite stream returned by iterate is processed lazily.

TIP: When you use a debugger to debug a stream computation, you can set a breakpoint in a method that is called from one of the transformations. With most IDEs, you can also set breakpoints in lambda expressions. If you just want to know what happens at a particular point in the stream pipeline, add

```
.peek(x -> {
   return; })
```

and set a breakpoint on the second line.

***java.util.stream.Stream* 8**

- `Stream<T> distinct()`

  yields a stream of the distinct elements of this stream.
- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`

  yields a stream whose elements are the elements of this stream in sorted order. The first method requires that the elements are instances of a class implementing `Comparable`.
- `Stream<T> peek(Consumer<? super T> action)`

  yields a stream with the same elements as this stream, passing each element to `action` as it is consumed.

1.6 Simple Reductions

Now that you have seen how to create and transform streams, we will finally get to the most important point—getting answers from the stream data. The methods covered in this section are called *reductions*. Reductions are *terminal operations*. They reduce the stream to a nonstream value that can be used in your program.

You have already seen a simple reduction: the `count` method that returns the number of elements of a stream.

Other simple reductions are `max` and `min` that return the largest or smallest value. There is a twist—these methods return an `Optional<T>` value that either wraps the answer or indicates that there is none (because the stream happened to be empty). In the olden days, it was common to return `null` in such a situation. But that can lead to null pointer exceptions when it happens in an incompletely tested program. The `Optional` type is a better way of indicating a missing return value. We discuss the `Optional` type in detail in the next section. Here is how you can get the maximum of a stream:

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.orElse(""));
```

The `findFirst` returns the first value in a nonempty collection. It is often useful when combined with `filter`. For example, here we find the first word that starts with the letter Q, if it exists:

```
Optional<String> startsWithQ
   = words.filter(s -> s.startsWith("Q")).findFirst();
```

If you are OK with any match, not just the first one, use the `findAny` method. This is effective when you parallelize the stream, since the stream can report any match that it finds instead of being constrained to the first one.

```
Optional<String> startsWithQ
   = words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

If you just want to know if there is a match, use `anyMatch`. That method takes a predicate argument, so you won't need to use `filter`.

```
boolean aWordStartsWithQ
   = words.parallel().anyMatch(s -> s.startsWith("Q"));
```

There are methods `allMatch` and `noneMatch` that return `true` if all or no elements match a predicate. These methods also benefit from being run in parallel.

java.util.stream.Stream 8

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

  yields a maximum or minimum element of this stream, using the ordering defined by the given comparator, or an empty `Optional` if this stream is empty. These are terminal operations.
- `Optional<T> findFirst()`
- `Optional<T> findAny()`

  yields the first, or any, element of this stream, or an empty `Optional` if this stream is empty. These are terminal operations.
- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`

  returns `true` if any, all, or none of the elements of this stream match the given predicate. These are terminal operations.

1.7 The Optional Type

An `Optional<T>` object is a wrapper for either an object of type `T` or no object. In the former case, we say that the value is *present*. The `Optional<T>` type is intended as a safer alternative for a reference of type `T` that either refers to an object or is `null`. But it is only safer if you use it right. The next three sections shows you how.

1.7.1 Getting an Optional Value

The key to using `Optional` effectively is to use a method that either *produces an alternative* if the value is not present, or *consumes the value* only if it is present.

In this section, we look at the first strategy. Often, there is a default that you want to use when there was no match, perhaps the empty string:

```
String result = optionalString.orElse("");
   // The wrapped string, or "" if none
```

You can also invoke code to compute the default:

```
String result = optionalString.orElseGet(() -> System.getProperty("myapp.default"));
   // The function is only called when needed
```

Or you can throw an exception if there is no value:

```
String result = optionalString.orElseThrow(IllegalStateException::new);
   // Supply a method that yields an exception object
```

java.util.Optional 8

- `T orElse(T other)`

  yields the value of this `Optional`, or `other` if this `Optional` is empty.
- `T orElseGet(Supplier<? extends T> other)`

  yields the value of this `Optional`, or the result of invoking `other` if this `Optional` is empty.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`

  yields the value of this `Optional`, or throws the result of invoking `exceptionSupplier` if this `Optional` is empty.

1.7.2 Consuming an Optional Value

In the preceding section, you saw how to produce an alternative if no value is present. The other strategy for working with optional values is to consume the value only if it is present.

The `ifPresent` method accepts a function. If the optional value exists, it is passed to that function. Otherwise, nothing happens.

```
optionalValue.ifPresent(v -> Process v);
```

For example, if you want to add the value to a set if it is present, call

```
optionalValue.ifPresent(v -> results.add(v));
```

or simply

```
optionalValue.ifPresent(results::add);
```

If you want to take one action if the `Optional` has a value and another action if it doesn't, use `ifPresentOrElse`:

```
optionalValue.ifPresentOrElse(
   v -> System.out.println("Found " + v),
   () -> logger.warning("No match"));
```

`java.util.Optional` 8

- `void ifPresent(Consumer<? super T> action)`

  if this `Optional` is nonempty, passes its value to `action`.
- `void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)` 9

  if this `Optional` is nonempty, passes its value to `action`, else invokes `emptyAction`.

1.7.3 Pipelining Optional Values

In the preceding sections, you saw how to get a value out of an `Optional` object. Another useful strategy is to keep the `Optional` intact. You can transform the value inside an `Optional` by using the `map` method:

```
Optional<String> transformed = optionalString.map(String::toUpperCase);
```

If `optionalString` is empty, then `transformed` is also empty.

Here is another example. We add a result to a list if it is present:

```
optionalValue.map(results::add);
```

If `optionalValue` is empty, nothing happens.

NOTE: This `map` method is the analog of the `map` method of the `Stream` interface that you have seen in Section 1.3, "The `filter`, `map`, and `flatMap` Methods," on p. 11. Simply imagine an optional value as a stream of size zero or one. The result again has size zero or one, and in the latter case, the function has been applied.

Similarly, you can use the `filter` method to only consider `Optional` values that fulfill a certain property before or after transforming it. If the property is not fulfilled, the pipeline yields an empty result:

```
Optional<String> transformed = optionalString
   .filter(s -> s.length() >= 8)
   .map(String::toUpperCase);
```

You can substitute an alternative `Optional` for an empty `Optional` with the `or` method. The alternative is computed lazily.

```
Optional<String> result = optionalString.or(() -> // Supply an Optional
   alternatives.stream().findFirst());
```

If `optionalString` has a value, then `result` is `optionalString`. If not, the lambda expression is evaluated, and its result is used.

`java.util.Optional` 8

- `<U> Optional<U> map(Function<? super T,? extends U> mapper)`

  yields an `Optional` whose value is obtained by applying the given function to the value of this `Optional` if present, or an empty `Optional` otherwise.
- `Optional<T> filter(Predicate<? super T> predicate)`

  yields an `Optional` with the value of this `Optional` if it fulfills the given predicate, or an empty `Optional` otherwise.
- `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)` 9

  yields this `Optional` if it is nonempty, or the one produced by the supplier otherwise.

1.7.4 How Not to Work with Optional Values

If you don't use `Optional` values correctly, you have no benefit over the "something or `null`" approach of the past.

The `get` method gets the wrapped element of an `Optional` value if it exists, or throws a `NoSuchElementException` if it doesn't. Therefore,

```
Optional<T> optionalValue = . . .;
optionalValue.get().someMethod()
```

is no safer than

```
T value = . . .;
value.someMethod();
```

The `isPresent` method reports whether an `Optional<T>` object has a value. But

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

is no easier than

```
if (value != null) value.someMethod();
```

NOTE: Java 10 introduces a scarier-sounding synonym for the `get` method. Call `optionalValue.orElseThrow()` to make explicit that the method will throw a `NoSuchElementException` if the `optionalValue` is empty. The hope is that programmers will only call that method when it is absolutely clear that the `Optional` is never empty.

Here are a few more tips for the proper use of the `Optional` type:

- A variable of type `Optional` should *never* be `null`.
- Don't use fields of type `Optional`. The cost is an additional object. Inside a class, using `null` for an absent field is manageable.
- Don't put `Optional` objects in a set, and don't use them as keys for a map. Collect the values instead.

`java.util.Optional` 8

- `T get()`
- `T orElseThrow()` 10

  yields the value of this `Optional`, or throws a `NoSuchElementException` if it is empty.
- `boolean isPresent()`

  returns `true` if this `Optional` is not empty.

1.7.5 Creating Optional Values

So far, we have discussed how to consume an `Optional` object someone else created. If you want to write a method that creates an `Optional` object, there are several static methods for that purpose, including `Optional.of(result)` and `Optional.empty()`. For example,

```
public static Optional<Double> inverse(Double x)
{
   return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

The `ofNullable` method is intended as a bridge from possibly `null` values to optional values. `Optional.ofNullable(obj)` returns `Optional.of(obj)` if `obj` is not `null` and `Optional.empty()` otherwise.

`java.util.Optional` 8

- `static <T> Optional<T> of(T value)`
- `static <T> Optional<T> ofNullable(T value)`

  yields an `Optional` with the given value. If `value` is `null`, the first method throws a `NullPointerException` and the second method yields an empty `Optional`.
- `static <T> Optional<T> empty()`

  yields an empty `Optional`.

1.7.6 Composing Optional Value Functions with `flatMap`

Suppose you have a method `f` yielding an `Optional<T>`, and the target type `T` has a method `g` yielding an `Optional<U>`. If they were normal methods, you could compose them by calling `s.f().g()`. But that composition doesn't work since `s.f()` has type `Optional<T>`, not `T`. Instead, call

```
Optional<U> result = s.f().flatMap(T::g);
```

If `s.f()` is present, then `g` is applied to it. Otherwise, an empty `Optional<U>` is returned.

Clearly, you can repeat that process if you have more methods or lambdas that yield `Optional` values. You can then build a pipeline of steps, simply by chaining calls to `flatMap`, that will succeed only when all parts do.

For example, consider the safe `inverse` method of the preceding section. Suppose we also have a safe square root:

```
public static Optional<Double> squareRoot(Double x)
{
   return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

Then you can compute the square root of the `inverse` as

```
Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);
```

or, if you prefer,

```
Optional<Double> result
   = Optional.of(-4.0).flatMap(Demo::inverse).flatMap(Demo::squareRoot);
```

If either the `inverse` method or the `squareRoot` returns `Optional.empty()`, the result is empty.

NOTE: You have already seen a `flatMap` method in the `Stream` interface (see Section 1.3, "The `filter`, `map`, and `flatMap` Methods," on p. 11). That method was used to compose two methods that yield streams, by flattening out the resulting stream of streams. The `Optional.flatMap` method works in the same way if you interpret an optional value as having zero or one element.

`java.util.Optional` 8

- `<U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper)`

  yields the result of applying `mapper` to the value in this `Optional` if present, or an empty optional otherwise.

1.7.7 Turning an Optional into a Stream

The `stream` method turns an `Optional<T>` into a `Stream<T>` with zero or one element. Sure, why not, but why would you ever want that?

This becomes useful with methods that return an `Optional` result. Suppose you have a stream of user IDs and a method

```
Optional<User> lookup(String id)
```

How do you get a stream of users, skipping those IDs that are invalid?

Of course, you can filter out the invalid IDs and then apply `get` to the remaining ones:

```
Stream<String> ids = . . .;
Stream<User> users = ids.map(Users::lookup)
   .filter(Optional::isPresent)
   .map(Optional::get);
```

But that uses the `isPresent` and `get` methods that we warned about. It is more elegant to call

```
Stream<User> users = ids.map(Users::lookup)
   .flatMap(Optional::stream);
```

Each call to `stream` returns a stream with zero or one element. The `flatMap` method combines them all. That means the nonexistent users are simply dropped.

NOTE: In this section, we consider the happy scenario in which we have a method that returns an `Optional` value. These days, many methods return `null` when there is no valid result. Suppose `Users.classicLookup(id)` returns a `User` object or `null`, not an `Optional<User>`. Then you can of course filter out the `null` values:

```
Stream<User> users = ids.map(Users::classicLookup)
   .filter(Objects::nonNull);
```

But if you prefer the `flatMap` approach, you can use

```
Stream<User> users = ids.flatMap(
   id -> Stream.ofNullable(Users.classicLookup(id)));
```

or

```
Stream<User> users = ids.map(Users::classicLookup)
   .flatMap(Stream::ofNullable);
```

The call `Stream.ofNullable(obj)` yields an empty stream if `obj` is `null` or a stream just containing `obj` otherwise.

The example program in Listing 1.3 demonstrates the `Optional` API.

Listing 1.3 `optional/OptionalTest.java`

```
package optional;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;

public class OptionalTest
{
   public static void main(String[] args) throws IOException
   {
      var contents = Files.readStr(
         Paths.get("../gutenberg/alice30.txt"));
      List<String> wordList = List.of(contents.split("\\PL+"));

      Optional<String> optionalValue = wordList.stream()
         .filter(s -> s.contains("fred"))
         .findFirst();
```

(Continues)

Listing 1.3 *(Continued)*

```
      System.out.println(optionalValue.orElse("No word") + " contains fred");

      Optional<String> optionalString = Optional.empty();
      String result = optionalString.orElse("N/A");
      System.out.println("result: " + result);
      result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
      System.out.println("result: " + result);
      try
      {
         result = optionalString.orElseThrow(IllegalStateException::new);
         System.out.println("result: " + result);
      }
      catch (Throwable t)
      {
         t.printStackTrace();
      }

      optionalValue = wordList.stream()
         .filter(s -> s.contains("red"))
         .findFirst();
      optionalValue.ifPresent(s -> System.out.println(s + " contains red"));

      var results = new HashSet<String>();
      optionalValue.ifPresent(results::add);
      Optional<Boolean> added = optionalValue.map(results::add);
      System.out.println(added);

      System.out.println(inverse(4.0).flatMap(OptionalTest::squareRoot));
      System.out.println(inverse(-1.0).flatMap(OptionalTest::squareRoot));
      System.out.println(inverse(0.0).flatMap(OptionalTest::squareRoot));
      Optional<Double> result2 = Optional.of(-4.0)
         .flatMap(OptionalTest::inverse).flatMap(OptionalTest::squareRoot);
      System.out.println(result2);
   }

   public static Optional<Double> inverse(Double x)
   {
      return x == 0 ? Optional.empty() : Optional.of(1 / x);
   }

   public static Optional<Double> squareRoot(Double x)
   {
      return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
   }
}
```

java.util.Optional 8

- `<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)` 9

  yields the result of applying `mapper` to the value of this `Optional`, or an empty `Optional` if this `Optional` is empty.

1.8 Collecting Results

When you are done with a stream, you will often want to look at the results. You can call the `iterator` method, which yields an old-fashioned iterator that you can use to visit the elements.

Alternatively, you can call the `forEach` method to apply a function to each element:

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

On a parallel stream, the `forEach` method traverses elements in arbitrary order. If you want to process them in stream order, call `forEachOrdered` instead. Of course, you might then give up some or all of the benefits of parallelism.

But more often than not, you will want to collect the result in a data structure. Call `toArray` to get an array of the stream elements.

Since it is not possible to create a generic array at runtime, the expression `stream.toArray()` returns an `Object[]` array. If you want an array of the correct type, pass in the array constructor:

```
String[] result = stream.toArray(String[]::new);
   // stream.toArray() has type Object[]
```

For collecting stream elements to another target, there is a convenient `collect` method that takes an instance of the `Collector` interface. A *collector* is an object that accumulates elements and produces a result. The `Collectors` class provides a large number of factory methods for common collectors. To collect stream elements into a list, use the collector produced by `Collectors.toList()`:

```
List<String> result = stream.collect(Collectors.toList());
```

Similarly, here is how you can collect stream elements into a set:

```
Set<String> result = stream.collect(Collectors.toSet());
```

If you want to control which kind of set you get, use the following call instead:

```
TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));
```

Suppose you want to collect all strings in a stream by concatenating them. You can call

```
String result = stream.collect(Collectors.joining());
```

If you want a delimiter between elements, pass it to the `joining` method:

```
String result = stream.collect(Collectors.joining(", "));
```

If your stream contains objects other than strings, you need to first convert them to strings, like this:

```
String result = stream.map(Object::toString).collect(Collectors.joining(", "));
```

If you want to reduce the stream results to a sum, count, average, maximum, or minimum, use one of the `summarizing(Int|Long|Double)` methods. These methods take a function that maps the stream objects to numbers and yield a result of type `(Int|Long|Double)SummaryStatistics`, simultaneously computing the sum, count, average, maximum, and minimum.

```
IntSummaryStatistics summary = stream.collect(
   Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

The example program in Listing 1.4 shows how to collect elements from a stream.

Listing 1.4 `collecting/CollectingResults.java`

```
package collecting;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class CollectingResults
{
   public static Stream<String> noVowels() throws IOException
   {
      var contents = Files.readStr(
         Paths.get("../gutenberg/alice30.txt"));

      List<String> wordList = List.of(contents.split("\\PL+"));
      Stream<String> words = wordList.stream();
      return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
   }

   public static <T> void show(String label, Set<T> set)
   {
      System.out.print(label + ": " + set.getClass().getName());
      System.out.println("["
         + set.stream().limit(10).map(Object::toString).collect(Collectors.joining(", "))
         + "]");
   }

   public static void main(String[] args) throws IOException
   {
      Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10).iterator();
      while (iter.hasNext())
         System.out.println(iter.next());

      Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
      System.out.println("Object array:" + numbers);
         // Note it's an Object[] array

      try
      {
         var number = (Integer) numbers[0]; // OK
         System.out.println("number: " + number);
         System.out.println("The following statement throws an exception:");
         var numbers2 = (Integer[]) numbers; // Throws exception
      }
      catch (ClassCastException ex)
      {
         System.out.println(ex);
      }

      Integer[] numbers3 = Stream.iterate(0, n -> n + 1)
         .limit(10)
         .toArray(Integer[]::new);
      System.out.println("Integer array: " + numbers3);
         // Note it's an Integer[] array

```

(Continues)

Listing 1.4 *(Continued)*

```java
      Set<String> noVowelSet = noVowels().collect(Collectors.toSet());
      show("noVowelSet", noVowelSet);

      TreeSet<String> noVowelTreeSet = noVowels().collect(
         Collectors.toCollection(TreeSet::new));
      show("noVowelTreeSet", noVowelTreeSet);

      String result = noVowels().limit(10).collect(Collectors.joining());
      System.out.println("Joining: " + result);
      result = noVowels().limit(10)
         .collect(Collectors.joining(", "));
      System.out.println("Joining with commas: " + result);

      IntSummaryStatistics summary = noVowels().collect(
         Collectors.summarizingInt(String::length));
      double averageWordLength = summary.getAverage();
      double maxWordLength = summary.getMax();
      System.out.println("Average word length: " + averageWordLength);
      System.out.println("Max word length: " + maxWordLength);
      System.out.println("forEach:");
      noVowels().limit(10).forEach(System.out::println);
   }
}
```

java.util.stream.BaseStream 8

- `Iterator<T> iterator()`

  yields an iterator for obtaining the elements of this stream. This is a terminal operation.

java.util.stream.Stream 8

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

  invokes `action` on each element of the stream. This is a terminal operation.

- `Object[] toArray()`
- `<A> A[] toArray(IntFunction<A[]> generator)`

  yields an array of objects, or of type `A` when passed a constructor reference `A[]::new`. These are terminal operations.

- `<R,A> R collect(Collector<? super T,A,R> collector)`

  collects the elements in this stream, using the given collector. The `Collectors` class has factory methods for many collectors.

java.util.stream.Collectors 8

- `static <T> Collector<T,?,List<T>> toList()`
- `static <T> Collector<T,?,List<T>> toUnmodifiableList()` 10
- `static <T> Collector<T,?,Set<T>> toSet()`
- `static <T> Collector<T,?,Set<T>> toUnmodifiableSet()` 10

  yield collectors that collect elements in a list or set.
- `static <T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory)`

  yields a collector that collects elements into an arbitrary collection. Pass a constructor reference such as `TreeSet::new`.
- `static Collector<CharSequence,?,String> joining()`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter)`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`

  yields a collector that joins strings. The delimiter is placed between strings, and the prefix and suffix before the first and after the last string. When not specified, these are empty.
- `static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)`
- `static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)`
- `static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)`

  yields collectors that produce an `(Int|Long|Double)SummaryStatistics` object, from which you can obtain the count, sum, average, maximum, and minimum of the results of applying `mapper` to each element.

IntSummaryStatistics 8
LongSummaryStatistics 8
DoubleSummaryStatistics 8

- `long getCount()`

  yields the count of the summarized elements.
- `(int|long|double) getSum()`
- `double getAverage()`

  yields the sum or average of the summarized elements, or zero if there are no elements.

(Continues)

DoubleSummaryStatistics 8 *(Continued)*

- `(int|long|double) getMax()`
- `(int|long|double) getMin()`

  yields the maximum or minimum of the summarized elements, or `(Integer|Long|Double).(MAX|MIN)_VALUE` if there are no elements.

1.9 Collecting into Maps

Suppose you have a `Stream<Person>` and want to collect the elements into a map so that later you can look up people by their ID. The `Collectors.toMap` method has two function arguments that produce the map's keys and values. For example,

```
Map<Integer, String> idToName = people.collect(
   Collectors.toMap(Person::getId, Person::getName));
```

In the common case when the values should be the actual elements, use `Function.identity()` for the second function.

```
Map<Integer, Person> idToPerson = people.collect(
   Collectors.toMap(Person::getId, Function.identity()));
```

If there is more than one element with the same key, there is a conflict, and the collector will throw an `IllegalStateException`. You can override that behavior by supplying a third function argument that resolves the conflict and determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

Here, we construct a map that contains, for each language in the available locales, as key its name in your default locale (such as `"German"`), and as value its localized name (such as `"Deutsch"`).

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
   Collectors.toMap(
      Locale::getDisplayLanguage,
      loc -> loc.getDisplayLanguage(loc),
      (existingValue, newValue) -> existingValue));
```

We don't care that the same language might occur twice (for example, German in Germany and in Switzerland), so we just keep the first entry.

NOTE: In this chapter, I use the `Locale` class as a source of an interesting data set. See Chapter 7 for more information on working with locales.

Now suppose we want to know all languages in a given country. Then we need a `Map<String, Set<String>>`. For example, the value for `"Switzerland"` is the set `[French, German, Italian]`. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

```
Map<String, Set<String>> countryLanguageSets = locales.collect(
   Collectors.toMap(
      Locale::getDisplayCountry,
      l -> Collections.singleton(l.getDisplayLanguage()),
      (a, b) -> { // Union of a and b
         var union = new HashSet<String>(a);
         union.addAll(b);
         return union; }));
```

You will see a simpler way of obtaining this map in the next section.

If you want a `TreeMap`, supply the constructor as the fourth argument. You must provide a merge function. Here is one of the examples from the beginning of the section, now yielding a `TreeMap`:

```
Map<Integer, Person> idToPerson = people.collect(
   Collectors.toMap(
      Person::getId,
      Function.identity(),
      (existingValue, newValue) -> { throw new IllegalStateException(); },
      TreeMap::new));
```

NOTE: For each of the `toMap` methods, there is an equivalent `toConcurrentMap` method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps. Note that elements are no longer collected in stream order, but that doesn't usually make a difference.

The program in Listing 1.5 gives examples of collecting stream results into maps.

Listing 1.5 `collecting/CollectingIntoMaps.java`

```java
package collecting;

/**
 * @version 1.00 2016-05-10
 * @author Cay Horstmann
 */

import java.io.*;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class CollectingIntoMaps
{

   public static class Person
   {
      private int id;
      private String name;

      public Person(int id, String name)
      {
         this.id = id;
         this.name = name;
      }

      public int getId()
      {
         return id;
      }

      public String getName()
      {
         return name;
      }

      public String toString()
      {
         return getClass().getName() + "[id=" + id + ",name=" + name + "]";
      }
   }

   public static Stream<Person> people()
   {
      return Stream.of(new Person(1001, "Peter"), new Person(1002, "Paul"),
         new Person(1003, "Mary"));
   }

   public static void main(String[] args) throws IOException
   {
      Map<Integer, String> idToName = people().collect(
         Collectors.toMap(Person::getId, Person::getName));
      System.out.println("idToName: " + idToName);

      Map<Integer, Person> idToPerson = people().collect(
         Collectors.toMap(Person::getId, Function.identity()));
      System.out.println("idToPerson: " + idToPerson.getClass().getName()
         + idToPerson);

      idToPerson = people().collect(
         Collectors.toMap(Person::getId, Function.identity(),
            (existingValue, newValue) -> { throw new IllegalStateException(); },
            TreeMap::new));
      System.out.println("idToPerson: " + idToPerson.getClass().getName()
         + idToPerson);

      Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
      Map<String, String> languageNames = locales.collect(
         Collectors.toMap(
            Locale::getDisplayLanguage,
            l -> l.getDisplayLanguage(l),
            (existingValue, newValue) -> existingValue));
      System.out.println("languageNames: " + languageNames);

      locales = Stream.of(Locale.getAvailableLocales());
      Map<String, Set<String>> countryLanguageSets = locales.collect(
         Collectors.toMap(
            Locale::getDisplayCountry,
            l -> Set.of(l.getDisplayLanguage()),
            (a, b) ->
            { // union of a and b
               Set<String> union = new HashSet<>(a);
               union.addAll(b);
               return union;
            }));
      System.out.println("countryLanguageSets: " + countryLanguageSets);
   }
}
```

`java.util.stream.Collectors` 8

- `static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
- `static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`
- `static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)`
- `static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)` 10
- `static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)` 10
- `static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
- `static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`
- `static <T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)`

  yields a collector that produces a map, unmodifiable map, or concurrent map. The `keyMapper` and `valueMapper` functions are applied to each collected element, yielding a key/value entry of the resulting map. By default, an `IllegalStateException` is thrown when two elements give rise to the same key. You can instead supply a `mergeFunction` that merges values with the same key. By default, the result is a `HashMap` or `ConcurrentHashMap`. You can instead supply a `mapSupplier` that yields the desired map instance.

1.10 Grouping and Partitioning

In the preceding section, you saw how to collect all languages in a given country. But the process was a bit tedious. You had to generate a singleton set for each map value and then specify how to merge the existing and new values. Forming groups of values with the same characteristic is very common, so the `groupingBy` method supports it directly.

Let's look at the problem of grouping locales by country. First, form this map:

```
Map<String, List<Locale>> countryToLocales = locales.collect(
   Collectors.groupingBy(Locale::getCountry));
```

The function `Locale::getCountry` is the *classifier function* of the grouping. You can now look up all locales for a given country code, for example

```
List<Locale> swissLocales = countryToLocales.get("CH");
   // Yields locales de_CH, fr_CH, it_CH and maybe more
```

NOTE: A quick refresher on locales: Each locale has a language code (such as `en` for English) and a country code (such as `US` for the United States). The locale `en_US` describes English in the United States, and `en_IE` is English in Ireland. Some countries have multiple locales. For example, `ga_IE` is Gaelic in Ireland, and, as the preceding example shows, the JDK knows at least three locales in Switzerland.

When the classifier function is a predicate function (that is, a function returning a `boolean` value), the stream elements are partitioned into two lists: those where the function returns `true` and the complement. In this case, it is more efficient to use `partitioningBy` instead of `groupingBy`. For example, here we split all locales into those that use English and all others:

```
Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(
   Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale> englishLocales = englishAndOtherLocales.get(true);
```

NOTE: If you call the `groupingByConcurrent` method, you get a concurrent map that, when used with a parallel stream, is concurrently populated. This is entirely analogous to the `toConcurrentMap` method.

`java.util.stream.Collectors` 8

- `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`
- `static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)`

  yields a collector that produces a map or concurrent map whose keys are the results of applying `classifier` to all collected elements, and whose values are lists of elements with the same key.
- `static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)`

  yields a collector that produces a map whose keys are `true`/`false`, and whose values are lists of the elements that fulfill/do not fulfill the predicate.

1.11 Downstream Collectors

The `groupingBy` method yields a map whose values are lists. If you want to process those lists in some way, supply a *downstream collector*. For example, if you want sets instead of lists, you can use the `Collectors.toSet` collector that you saw in the preceding section:

```
Map<String, Set<Locale>> countryToLocaleSet = locales.collect(
   groupingBy(Locale::getCountry, toSet()));
```

NOTE: In this example, as well as the remaining examples of this section, I assume a static import of `java.util.stream.Collectors.*` to make the expressions easier to read.

Several collectors are provided for reducing collected elements to numbers:

- `counting` produces a count of the collected elements. For example,

  ```
  Map<String, Long> countryToLocaleCounts = locales.collect(
     groupingBy(Locale::getCountry, counting()));
  ```

  counts how many locales there are for each country.
- `summing(Int|Long|Double)` takes a function argument, applies the function to the downstream elements, and produces their sum. For example,

  ```
  Map<String, Integer> stateToCityPopulation = cities.collect(
     groupingBy(City::getState, summingInt(City::getPopulation)));
  ```

  computes the sum of populations per state in a stream of cities.
- `maxBy` and `minBy` take a comparator and produce maximum and minimum of the downstream elements. For example,

  ```
  Map<String, Optional<City>> stateToLargestCity = cities.collect(
     groupingBy(City::getState,
        maxBy(Comparator.comparing(City::getPopulation))));
  ```

  produces the largest city per state.

The `collectingAndThen` collector adds a final processing step behind a collector. For example, if you want to know how many distinct results there are, collect them into a set and then compute the size:

```
Map<Character, Integer> stringCountsByStartingLetter = strings.collect(
   groupingBy(s -> s.charAt(0),
      collectingAndThen(toSet(), Set::size)));
```

The `mapping` collector does the opposite. It applies a function to each collected element and passes the results to a downstream collector.

```
Map<Character, Set<Integer>> stringLengthsByStartingLetter = strings.collect(
   groupingBy(s -> s.charAt(0),
      mapping(String::length, toSet())));
```

Here, we group strings by their first character. Within each group, we produce the lengths and collect them in a set.

The `mapping` method also yields a nicer solution to a problem from the preceding section—gathering a set of all languages in a country.

```
Map<String, Set<String>> countryToLanguages = locales.collect(
   groupingBy(Locale::getDisplayCountry,
      mapping(Locale::getDisplayLanguage,
         toSet())));
```

There is a `flatMapping` method as well, for use with functions that return streams.

If the grouping or mapping function has return type `int`, `long`, or `double`, you can collect elements into a summary statistics object, as discussed in Section 1.8, "Collecting Results," on p. 25. For example,

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities.collect(
   groupingBy(City::getState,
      summarizingInt(City::getPopulation)));
```

Then you can get the sum, count, average, minimum, and maximum of the function values from the summary statistics objects of each group.

The `filtering` collector applies a filter to each group, for example:

```
Map<String, Set<City>> largeCitiesByState
   = cities.collect(
      groupingBy(City::getState,
         filtering(c -> c.getPopulation() > 500000,
            toSet()))); // States without large cities have empty sets
```

NOTE: There are also three versions of a `reducing` method that apply general reductions, as described in the next section.

Composing collectors is powerful, but it can lead to very convoluted expressions. The best use is with `groupingBy` or `partitioningBy` to process the "downstream" map values. Otherwise, simply apply methods such as `map`, `reduce`, `count`, `max`, or `min` directly on streams.

The example program in Listing 1.6 demonstrates downstream collectors.

Listing 1.6 collecting/DownstreamCollectors.java

```java
package collecting;

/**
 * @version 1.00 2016-05-10
 * @author Cay Horstmann
 */

import static java.util.stream.Collectors.*;

import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class DownstreamCollectors
{

   public static class City
   {
      private String name;
      private String state;
      private int population;

      public City(String name, String state, int population)
      {
         this.name = name;
         this.state = state;
         this.population = population;
      }

      public String getName()
      {
         return name;
      }

      public String getState()
      {
         return state;
      }

      public int getPopulation()
      {
         return population;
      }
   }

   public static Stream<City> readCities(String filename) throws IOException
   {
      return Files.lines(Paths.get(filename))
         .map(l -> l.split(", "))
         .map(a -> new City(a[0], a[1], Integer.parseInt(a[2])));
   }

   public static void main(String[] args) throws IOException
   {
      Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());

      Map<String, Set<Locale>> countryToLocaleSet = locales.collect(groupingBy(
         Locale::getCountry, toSet()));
      System.out.println("countryToLocaleSet: " + countryToLocaleSet);

      locales = Stream.of(Locale.getAvailableLocales());
      Map<String, Long> countryToLocaleCounts = locales.collect(groupingBy(
         Locale::getCountry, counting()));
      System.out.println("countryToLocaleCounts: " + countryToLocaleCounts);

      Stream<City> cities = readCities("cities.txt");
      Map<String, Integer> stateToCityPopulation = cities.collect(groupingBy(
         City::getState, summingInt(City::getPopulation)));
      System.out.println("stateToCityPopulation: " + stateToCityPopulation);

      cities = readCities("cities.txt");
      Map<String, Optional<String>> stateToLongestCityName = cities
         .collect(groupingBy(City::getState,
            mapping(City::getName, maxBy(Comparator.comparing(String::length)))));
      System.out.println("stateToLongestCityName: " + stateToLongestCityName);

      locales = Stream.of(Locale.getAvailableLocales());
      Map<String, Set<String>> countryToLanguages = locales.collect(groupingBy(
         Locale::getDisplayCountry, mapping(Locale::getDisplayLanguage, toSet())));
      System.out.println("countryToLanguages: " + countryToLanguages);

      cities = readCities("cities.txt");
      Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities
         .collect(groupingBy(City::getState, summarizingInt(City::getPopulation)));
      System.out.println(stateToCityPopulationSummary.get("NY"));

      cities = readCities("cities.txt");
      Map<String, String> stateToCityNames = cities.collect(groupingBy(
         City::getState,
         reducing("", City::getName, (s, t) -> s.length() == 0 ? t : s + ", " + t)));

      cities = readCities("cities.txt");
      stateToCityNames = cities.collect(groupingBy(City::getState,
         mapping(City::getName, joining(", "))));
      System.out.println("stateToCityNames: " + stateToCityNames);
   }
}
```

java.util.stream.Collectors 8

- `public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`

  yields a collector that produces a map. The keys are the results of applying `classifier` to all collected elements. The values are the results of collecting elements with the same key, using the downstream collector.

- `static <T> Collector<T,?,Long> counting()`

  yields a collector that counts the collected elements.

- `static <T> Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)`
- `static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)`
- `static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)`

  yields a collector that computes the sum of the results of applying `mapper` to the collected elements.

- `static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)`
- `static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)`

  yields a collector that computes the maximum or minimum of the collected elements, using the ordering specified by `comparator`.

- `static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)`

  yields a collector that sends elements to the downstream collector and then applies the finisher function to its result.

- `static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)`

  yields a collector that calls `mapper` on each element and passes the results to the downstream collector.

- `static <T,U,A,R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)`

  yields a collector that calls `mapper` on each element and passes the elements of the results to the downstream collector.

- `static <T,A,R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)`

  yields a collector that passes the elements fulfilling the predicate to the downstream collector.

1.12 Reduction Operations

The `reduce` method is a general mechanism for computing a value from a stream. The simplest form takes a binary function and keeps applying it, starting with the first two elements. It's easy to explain this if the function is the sum:

```
List<Integer> values = . . .;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

In this case, the `reduce` method computes $v_0 + v_1 + v_2 + \ . . .$, where v_i are the stream elements. The method returns an `Optional` because there is no valid result if the stream is empty.

NOTE: In this case, you can write `reduce(Integer::sum)` instead of `reduce((x, y) -> x + y)`.

More generally, you can use any operation that combines a partial result `x` with the next value `y` to yield a new partial result.

Here is another way of looking at reductions. Given a reduction operation *op*, the reduction yields v_0 *op* v_1 *op* v_2 *op* . . . , where v_i *op* v_{i+1} denotes the function call *op*(v_i, v_{i+1}). There are many operations that might be useful in practice—such as sum, product, string concatenation, maximum and minimum, set union or intersection.

If you want to use reduction with parallel streams, the operation must be *associative*: It shouldn't matter in which order you combine the elements. In math notation, (x *op* y) *op* z must be equal to x *op* (y *op* z). An example of an operation that is not associative is subtraction. For example, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Often, there is an *identity* e such that e *op* $x = x$, and that element can be used as the start of the computation. For example, 0 is the identity for addition, and you can use the second form of `reduce`:

```
List<Integer> values = . . .;
Integer sum = values.stream().reduce(0, (x, y) -> x + y);
   // Computes 0 + v0 + v1 + v2 + . . .
```

The identity value is returned if the stream is empty, and you no longer need to deal with the `Optional` class.

Now suppose you have a stream of objects and want to form the sum of some property, such as lengths in a stream of strings. You can't use the simple form of `reduce`. It requires a function `(T, T) -> T`, with the same types for the arguments and the result, but in this situation you have two types: The stream elements have type `String`, and the accumulated result is an integer. There is a form of `reduce` that can deal with this situation.

First, you supply an "accumulator" function `(total, word) -> total + word.length()`. That function is called repeatedly, forming the cumulative total. But when the computation is parallelized, there will be multiple computations of this kind, and you need to combine their results. You supply a second function for that purpose. The complete call is

```
int result = words.reduce(0,
   (total, word) -> total + word.length(),
   (total1, total2) -> total1 + total2);
```

NOTE: In practice, you probably won't use the `reduce` method a lot. It is usually easier to map to a stream of numbers and use one of its methods to compute sum, maximum, or minimum. (We discuss streams of numbers in Section 1.13, "Primitive Type Streams," on p. 43.) In this particular example, you could have called `words.mapToInt(String::length).sum()`, which is both simpler and more efficient since it doesn't involve boxing.

NOTE: There are times when `reduce` is not general enough. For example, suppose you want to collect the results in a `BitSet`. If the collection is parallelized, you can't put the elements directly into a single `BitSet` because a `BitSet` object is not thread-safe. For that reason, you can't use `reduce`. Each segment needs to start out with its own empty set, and `reduce` only lets you supply one identity value. Instead, use `collect`. It takes three arguments:

1. A *supplier* to make new instances of the target object—for example, a constructor for a hash set
2. An *accumulator* that adds an element to the target, such as an `add` method
3. A *combiner* that merges two objects into one, such as `addAll`

Here is how the `collect` method works for a bit set:

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or);
```

java.util.Stream **8**

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

  forms a cumulative total of the stream elements with the given `accumulator` function. If `identity` is provided, then it is the first value to be accumulated. If `combiner` is provided, it can be used to combine totals of segments that are accumulated separately.
- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`

  collects elements in a result of type `R`. On each segment, `supplier` is called to provide an initial result, `accumulator` is called to mutably add elements to it, and `combiner` is called to combine two results.

1.13 Primitive Type Streams

So far, we have collected integers in a `Stream<Integer>`, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types—`double`, `float`, `long`, `short`, `char`, `byte`, and `boolean`. The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers. If you want to store `short`, `char`, `byte`, and `boolean`, use an `IntStream`; for `float`, use a `DoubleStream`.

To create an `IntStream`, call the `IntStream.of` and `Arrays.stream` methods:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // values is an int[] array
```

As with object streams, you can also use the static `generate` and `iterate` methods. In addition, `IntStream` and `LongStream` have static methods `range` and `rangeClosed` that generate integer ranges with step size one:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100); // Upper bound is excluded
IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper bound is included
```

The `CharSequence` interface has methods `codePoints` and `chars` that yield an `IntStream` of the Unicode codes of the characters or of the code units in the UTF-16 encoding. (See Chapter 1 for the sordid details.)

```
String sentence = "\uD835\uDD46 is the set of octonions.";
   // \uD835\uDD46 is the UTF-16 encoding of the letter 𝕆, unicode U+1D546

IntStream codes = sentence.codePoints();
   // The stream with hex values 1D546 20 69 73 20 . . .
```

When you have a stream of objects, you can transform it to a primitive type stream with the `mapToInt`, `mapToLong`, or `mapToDouble` methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an `IntStream`:

```
Stream<String> words = . . .;
IntStream lengths = words.mapToInt(String::length);
```

To convert a primitive type stream to an object stream, use the `boxed` method:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

- The `toArray` methods return primitive type arrays.
- Methods that yield an optional result return an `OptionalInt`, `OptionalLong`, or `OptionalDouble`. These classes are analogous to the `Optional` class, but they have methods `getAsInt`, `getAsLong`, and `getAsDouble` instead of the `get` method.
- There are methods `sum`, `average`, `max`, and `min` that return the sum, average, maximum, and minimum. These methods are not defined for object streams.
- The `summaryStatistics` method yields an object of type `IntSummaryStatistics`, `LongSummaryStatistics`, or `DoubleSummaryStatistics` that can simultaneously report the sum, count, average, maximum, and minimum of the stream.

NOTE: The `Random` class has methods `ints`, `longs`, and `doubles` that return primitive type streams of random numbers. If you need random numbers in parallel streams, use the `SplittableRandom` class instead.

The program in Listing 1.7 gives examples for the API of primitive type streams.

Listing 1.7 `streams/PrimitiveTypeStreams.java`

```
package streams;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class PrimitiveTypeStreams
{
   public static void show(String title, IntStream stream)
   {
      final int SIZE = 10;
      int[] firstElements = stream.limit(SIZE + 1).toArray();
      System.out.print(title + ": ");
      for (int i = 0; i < firstElements.length; i++)
      {
         if (i > 0) System.out.print(", ");
         if (i < SIZE) System.out.print(firstElements[i]);
         else System.out.print("...");
      }
      System.out.println();
   }

   public static void main(String[] args) throws IOException
   {
      IntStream is1 = IntStream.generate(() -> (int) (Math.random() * 100));
      show("is1", is1);
      IntStream is2 = IntStream.range(5, 10);
      show("is2", is2);
      IntStream is3 = IntStream.rangeClosed(5, 10);
      show("is3", is3);

      Path path = Paths.get("../gutenberg/alice30.txt");
      var contents = Files.readStr(path);

      Stream<String> words = Stream.of(contents.split("\\PL+"));
      IntStream is4 = words.mapToInt(String::length);
      show("is4", is4);
      var sentence = "\uD835\uDD46 is the set of octonions.";
      System.out.println(sentence);
      IntStream codes = sentence.codePoints();
      System.out.println(codes.mapToObj(c -> String.format("%X ", c)).collect(
         Collectors.joining()));

      Stream<Integer> integers = IntStream.range(0, 100).boxed();
      IntStream is5 = integers.mapToInt(Integer::intValue);
      show("is5", is5);
   }
}
```

java.util.stream.IntStream 8

- `static IntStream range(int startInclusive, int endExclusive)`
- `static IntStream rangeClosed(int startInclusive, int endInclusive)`

  yields an `IntStream` with the integers in the given range.
- `static IntStream of(int... values)`

  yields an `IntStream` with the given elements.
- `int[] toArray()`

  yields an array with the elements of this stream.
- `int sum()`
- `OptionalDouble average()`
- `OptionalInt max()`
- `OptionalInt min()`
- `IntSummaryStatistics summaryStatistics()`

  yields the sum, count, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- `Stream<Integer> boxed()`

  yields a stream of wrapper objects for the elements in this stream.

java.util.stream.LongStream 8

- `static LongStream range(long startInclusive, long endExclusive)`
- `static LongStream rangeClosed(long startInclusive, long endInclusive)`

  yields a `LongStream` with the integers in the given range.
- `static LongStream of(long... values)`

  yields a `LongStream` with the given elements.
- `long[] toArray()`

  yields an array with the elements of this stream.
- `long sum()`
- `OptionalDouble average()`
- `OptionalLong max()`
- `OptionalLong min()`
- `LongSummaryStatistics summaryStatistics()`

  yields the sum, count, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- `Stream<Long> boxed()`

  yields a stream of wrapper objects for the elements in this stream.

java.util.stream.DoubleStream 8

- `static DoubleStream of(double... values)`

  yields a `DoubleStream` with the given elements.
- `double[] toArray()`

  yields an array with the elements of this stream.
- `double sum()`
- `OptionalDouble average()`
- `OptionalDouble max()`
- `OptionalDouble min()`
- `DoubleSummaryStatistics summaryStatistics()`

  yields the sum, count, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- `Stream<Double> boxed()`

  yields a stream of wrapper objects for the elements in this stream.

java.lang.CharSequence 1.0

- `IntStream codePoints()` 8

  yields a stream of all Unicode code points of this string.

java.util.Random 1.0

- `IntStream ints()`
- `IntStream ints(int randomNumberOrigin, int randomNumberBound)` 8
- `IntStream ints(long streamSize)` 8
- `IntStream ints(long streamSize, int randomNumberOrigin, int randomNumberBound)` 8
- `LongStream longs()` 8
- `LongStream longs(long randomNumberOrigin, long randomNumberBound)` 8
- `LongStream longs(long streamSize)` 8
- `LongStream longs(long streamSize, long randomNumberOrigin, long randomNumberBound)` 8
- `DoubleStream doubles()` 8
- `DoubleStream doubles(double randomNumberOrigin, double randomNumberBound)` 8
- `DoubleStream doubles(long streamSize)` 8
- `DoubleStream doubles(long streamSize, double randomNumberOrigin, double randomNumberBound)` 8

  yields streams of random numbers. If `streamSize` is provided, the stream is finite with the given number of elements. When bounds are provided, the elements are between `randomNumberOrigin` (inclusive) and `randomNumberBound` (exclusive).

java.util.Optional(Int|Long|Double) 8

- `static Optional(Int|Long|Double) of((int|long|double) value)`

  yields an optional object with the supplied primitive type value.
- `(int|long|double) getAs(Int|Long|Double)()`

  yields the value of this optional object, or throws a `NoSuchElementException` if it is empty.
- `(int|long|double) orElse((int|long|double) other)`
- `(int|long|double) orElseGet((Int|Long|Double)Supplier other)`

  yields the value of this optional object, or the alternative value if this object is empty.
- `void ifPresent((Int|Long|Double)Consumer consumer)`

  If this optional object is not empty, passes its value to `consumer`.

java.util.(Int|Long|Double)SummaryStatistics 8

- `long getCount()`
- `(int|long|double) getSum()`
- `double getAverage()`
- `(int|long|double) getMax()`
- `(int|long|double) getMin()`

  yields the count, sum, average, maximum, and minimum of the collected elements.

1.14 Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. You can get a parallel stream from any collection with the `Collection.parallelStream()` method:

```
Stream<String> parallelWords = words.parallelStream();
```

Moreover, the `parallel` method converts any sequential stream into a parallel one.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

As long as the stream is in parallel mode when the terminal method executes, all intermediate stream operations will be parallelized.

When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations are *stateless* and can be executed in an arbitrary order.

Here is an example of something you cannot do. Suppose you want to count all short words in a stream of strings:

```
var shortWords = new int[12];
words.parallelStream().forEach(
   s -> { if (s.length() < 12) shortWords[s.length()]++; });
      // ERROR--race condition!
System.out.println(Arrays.toString(shortWords));
```

This is very, very bad code. The function passed to `forEach` runs concurrently in multiple threads, each updating a shared array. As you saw in Chapter 12 of Volume I, that's a classic *race condition*. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run—each of them wrong.

It is your responsibility to ensure that any functions you pass to parallel stream operations are safe to execute in parallel. The best way to do that is to stay away from mutable state. In this example, you can safely parallelize the computation if you group strings by length and count them:

```
Map<Integer, Long> shortWordCounts
   = words.parallelStream()
      .filter(s -> s.length() < 12)
      .collect(groupingBy(
         String::length,
         counting()));
```

By default, streams that arise from ordered collections (arrays and lists), from ranges, generators, and iterators, or from calling `Stream.sorted`, are *ordered*. Results are accumulated in the order of the original elements, and are entirely predictable. If you run the same operations twice, you will get exactly the same results.

Ordering does not preclude efficient parallelization. For example, when computing `stream.map(fun)`, the stream can be partitioned into n segments, each of which is concurrently processed. Then the results are reassembled in order.

Some operations can be more effectively parallelized when the ordering requirement is dropped. By calling the `Stream.unordered` method, you indicate that you are not interested in ordering. One operation that can benefit from this is `Stream.distinct`. On an ordered stream, `distinct` retains the first of all equal elements. That impedes parallelization—the thread processing a segment can't know which elements to discard until the preceding segment has been processed. If it is acceptable to retain *any* of the unique elements, all segments can be processed concurrently (using a shared set to track duplicates).

You can also speed up the `limit` method by dropping ordering. If you just want any `n` elements from a stream and you don't care which ones you get, call

```
Stream<String> sample = words.parallelStream().unordered().limit(n);
```

As discussed in Section 1.9, "Collecting into Maps," on p. 30, merging maps is expensive. For that reason, the `Collectors.groupingByConcurrent` method uses a shared concurrent map. To benefit from parallelism, the order of the map values will not be the same as the stream order.

```
Map<Integer, List<String>> result = words.parallelStream().collect(
   Collectors.groupingByConcurrent(String::length));
   // Values aren't collected in stream order
```

Of course, you won't care if you use a downstream collector that is independent of the ordering, such as

```
Map<Integer, Long> wordCounts
   = words.parallelStream()
      .collect(
         groupingByConcurrent(
            String::length,
            counting()));
```

Don't turn all your streams into parallel streams in the hope of speeding up operations. Keep these issues in mind:

- There is a substantial overhead to parallelization that will only pay off for very large data sets.
- Parallelizing a stream is only a win if the underlying data source can be effectively split into multiple parts.
- The thread pool that is used by parallel streams can be starved by blocking operations such as file I/O or network access.

Parallel streams work best with huge in-memory collections of data and computationally intensive processing.

TIP: Prior to Java 9, parallelizing the stream returned by the `Files.lines` method made no sense. The data was not splittable—you had to read the first half of the file before the second half. Now the method uses a memory-mapped file, and splitting is effective. If you process the lines of a huge file, parallelizing the stream may improve performance.

NOTE: By default, parallel streams use the global fork-join pool returned by `ForkJoinPool.commonPool`. That is fine if your operations don't block and you don't share the pool with other tasks. There is a trick to substitute a different pool. Place your operations inside the `submit` method of a custom pool:

```
ForkJoinPool customPool = . . .;
result = customPool.submit(() ->
   stream.parallel().map(. . .).collect(. . .)).get();
```

Or, asynchronously:

```
CompletableFuture.supplyAsync(() ->
   stream.parallel().map(. . .).collect(. . .),
   customPool).thenAccept(result -> . . .);
```

NOTE: If you want to parallelize stream computations based on random numbers, don't start out with a stream obtained from the `Random.ints`, `Random.longs`, or `Random.doubles` methods. Those streams don't split. Instead, use the `ints`, `longs`, or `doubles` methods of the `SplittableRandom` class.

The example program in Listing 1.8 demonstrates how to work with parallel streams.

Listing 1.8 `parallel/ParallelStreams.java`

```
package parallel;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import static java.util.stream.Collectors.*;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class ParallelStreams
{
   public static void main(String[] args) throws IOException
   {
```

(Continues)

Listing 1.8 *(Continued)*

```
      var contents = new String(Files.readAllBytes(
         Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
      List<String> wordList = List.of(contents.split("\\PL+"));

      // Very bad code ahead
      var shortWords = new int[10];
      wordList.parallelStream().forEach(s ->
         {
            if (s.length() < 10) shortWords[s.length()]++;
         });
      System.out.println(Arrays.toString(shortWords));

      // Try again--the result will likely be different (and also wrong)
      Arrays.fill(shortWords, 0);
      wordList.parallelStream().forEach(s ->
         {
            if (s.length() < 10) shortWords[s.length()]++;
         });
      System.out.println(Arrays.toString(shortWords));

      // Remedy: Group and count
      Map<Integer, Long> shortWordCounts = wordList.parallelStream()
         .filter(s -> s.length() < 10)
         .collect(groupingBy(String::length, counting()));

      System.out.println(shortWordCounts);

      // Downstream order not deterministic
      Map<Integer, List<String>> result = wordList.parallelStream().collect(
         Collectors.groupingByConcurrent(String::length));

      System.out.println(result.get(14));

      result = wordList.parallelStream().collect(
         Collectors.groupingByConcurrent(String::length));

      System.out.println(result.get(14));

      Map<Integer, Long> wordCounts = wordList.parallelStream().collect(
         groupingByConcurrent(String::length, counting()));

      System.out.println(wordCounts);
   }
}
```

java.util.stream.BaseStream<T,S extends BaseStream<T,S>> 8

- `S parallel()`

  yields a parallel stream with the same elements as this stream.
- `S unordered()`

  yields an unordered stream with the same elements as this stream.

java.util.Collection<E> 1.2

- `Stream<E> parallelStream()` 8

  yields a parallel stream with the elements of this collection.

In this chapter, you have learned how to put the stream library of Java 8 to use. The next chapter covers another important topic: processing input and output.

CHAPTER 2

Input and Output

In this chapter

In this chapter, we cover the Java Application Programming Interfaces (APIs) for input and output. You will learn how to access files and directories and how to read and write data in binary and text format. This chapter also shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data. Next, we will turn to working with files and directories. We finish the chapter with a discussion of regular expressions, even though they are not actually related to input and output. We couldn't find a better place to handle that topic, and apparently neither could the Java team—the regular expression API specification was attached to a specification request for "new I/O" features.

2.1 Input/Output Streams

In the Java API, an object from which we can read a sequence of bytes is called an *input stream*. An object to which we can write a sequence of bytes is called an *output stream*. These sources and destinations of byte sequences can be—and often are—files, but they can also be network connections and even blocks of memory. The abstract classes `InputStream` and `OutputStream` are the basis for a hierarchy of input/output (I/O) classes.

NOTE: These input/output streams are unrelated to the streams that you saw in the preceding chapter. For clarity, we will use the terms input stream, output stream, or input/output stream whenever we discuss streams that are used for input and output.

Byte-oriented input/output streams are inconvenient for processing information stored in Unicode (recall that Unicode uses multiple bytes per character). Therefore, a separate hierarchy provides classes, inheriting from the abstract `Reader` and `Writer` classes, for processing Unicode characters. These classes have read and write operations that are based on two-byte `char` values (that is, UTF-16 code units) rather than `byte` values.

2.1.1 Reading and Writing Bytes

The `InputStream` class has an abstract method:

```
abstract int read()
```

This method reads one byte and returns the byte that was read, or `-1` if it encounters the end of the input source. The designer of a concrete input stream class overrides this method to provide useful functionality. For example, in the `FileInputStream` class, this method reads one byte from a file. `System.in` is a predefined object of a subclass of `InputStream` that allows you to read information from "standard input," that is, the console or a redirected file.

The `InputStream` class also has nonabstract methods to read an array of bytes or to skip a number of bytes. Since Java 9, there is a very useful method to read all bytes of a stream:

```
byte[] bytes = in.readAllBytes();
```

There are also methods to read a given number of bytes—see the API notes.

These methods call the abstract `read` method, so subclasses need to override only one method.

Similarly, the `OutputStream` class defines the abstract method

```
abstract void write(int b)
```

which writes one byte to an output location.

If you have an array of bytes, you can write them all at once:

```
byte[] values = . . .;
out.write(values);
```

The `transferTo` method transfers all bytes from an input stream to an output stream:

```
in.transferTo(out);
```

Both the `read` and `write` methods *block* until the byte is actually read or written. This means that if the input stream cannot immediately be accessed (usually because of a busy network connection), the current thread blocks. This gives other threads the chance to do useful work while the method is waiting for the input stream to become available again.

The `available` method lets you check the number of bytes that are currently available for reading. This means a fragment like the following is unlikely to block:

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
   var data = new byte[bytesAvailable];
   in.read(data);
}
```

When you have finished reading or writing to an input/output stream, close it by calling the `close` method. This call frees up the operating system resources that are in limited supply. If an application opens too many input/output streams without closing them, system resources can become depleted. Closing an output stream also *flushes* the buffer used for the output stream: Any bytes that were temporarily placed in a buffer so that they could be delivered as a larger packet are sent off. In particular, if you do not close a file, the last packet of bytes might never be delivered. You can also manually flush the output with the `flush` method.

Even if an input/output stream class provides concrete methods to work with the raw `read` and `write` functions, application programmers rarely use them. The data that you are interested in probably contain numbers, strings, and objects, not raw bytes.

Instead of working with bytes, you can use one of many input/output classes that build upon the basic `InputStream` and `OutputStream` classes.

`java.io.InputStream` 1.0

- `abstract int read()`

  reads a byte of data and returns the byte read; returns `-1` at the end of the input stream.
- `int read(byte[] b)`

  reads into an array of bytes and returns the actual number of bytes read, or `-1` at the end of the input stream; this method reads at most `b.length` bytes.
- `int read(byte[] b, int off, int len)`
- `int readNBytes(byte[] b, int off, int len)` 9

  reads up to `len` bytes, if available without blocking (`read`), or blocking until all values have been read (`readNBytes`). Values are placed into `b`, starting at `off`. Returns the actual number of bytes read, or `-1` at the end of the input stream.
- `byte[] readAllBytes()` 9

  yields an array of all bytes that can be read from this stream.
- `long transferTo(OutputStream out)` 9

  transfers all bytes from this input stream to the given output stream, returning the number of bytes transferred. Neither stream is closed.
- `long skip(long n)`

  skips `n` bytes in the input stream, returns the actual number of bytes skipped (which may be less than `n` if the end of the input stream was encountered).
- `int available()`

  returns the number of bytes available, without blocking (recall that blocking means that the current thread loses its turn).
- `void close()`

  closes the input stream.
- `void mark(int readlimit)`

  puts a marker at the current position in the input stream (not all streams support this feature). If more than `readlimit` bytes have been read from the input stream, the stream is allowed to forget the marker.
- `void reset()`

  returns to the last marker. Subsequent calls to `read` reread the bytes. If there is no current marker, the input stream is not reset.
- `boolean markSupported()`

  returns `true` if the input stream supports marking.

java.io.OutputStream 1.0

- `abstract void write(int n)`

  writes a byte of data.
- `void write(byte[] b)`
- `void write(byte[] b, int off, int len)`

  writes all bytes, or `len` bytes starting at `off`, in the array `b`.
- `void close()`

  flushes and closes the output stream.
- `void flush()`

  flushes the output stream—that is, sends any buffered data to its destination.

2.1.2 The Complete Stream Zoo

Unlike C, which gets by just fine with a single type `FILE*`, Java has a whole zoo of more than 60 (!) different input/output stream types (see Figures 2.1 and 2.2).

Let's divide the animals in the input/output stream zoo by how they are used. There are separate hierarchies for classes that process bytes and characters. As you saw, the `InputStream` and `OutputStream` classes let you read and write individual bytes and arrays of bytes. These classes form the basis of the hierarchy shown in Figure 2.1. To read and write strings and numbers, you need more capable subclasses. For example, `DataInputStream` and `DataOutputStream` let you read and write all the primitive Java types in binary format. Finally, there are input/output streams that do useful stuff; for example, the `ZipInputStream` and `ZipOutputStream` let you read and write files in the familiar ZIP compression format.

For Unicode text, on the other hand, you can use subclasses of the abstract classes `Reader` and `Writer` (see Figure 2.2). The basic methods of the `Reader` and `Writer` classes are similar to those of `InputStream` and `OutputStream`.

```
abstract int read()
abstract void write(int c)
```

The `read` method returns either a UTF-16 code unit (as an integer between `0` and `65535`) or `-1` when you have reached the end of the file. The `write` method is called with a Unicode code unit. (See Volume I, Chapter 3 for a discussion of Unicode code units.)

There are four additional interfaces: `Closeable`, `Flushable`, `Readable`, and `Appendable` (see Figure 2.3). The first two interfaces are very simple, with methods

```
void close() throws IOException
```

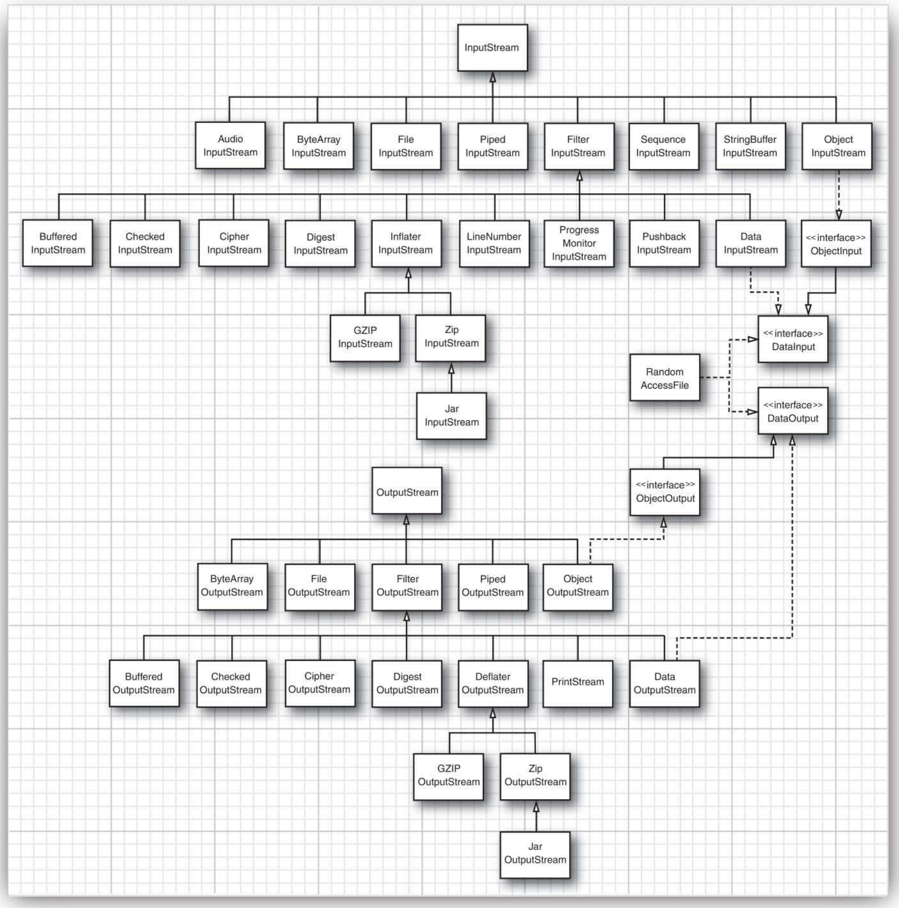

Figure 2.1 Input and output stream hierarchy

and

```
void flush()
```

respectively. The classes `InputStream`, `OutputStream`, `Reader`, and `Writer` all implement the `Closeable` interface.

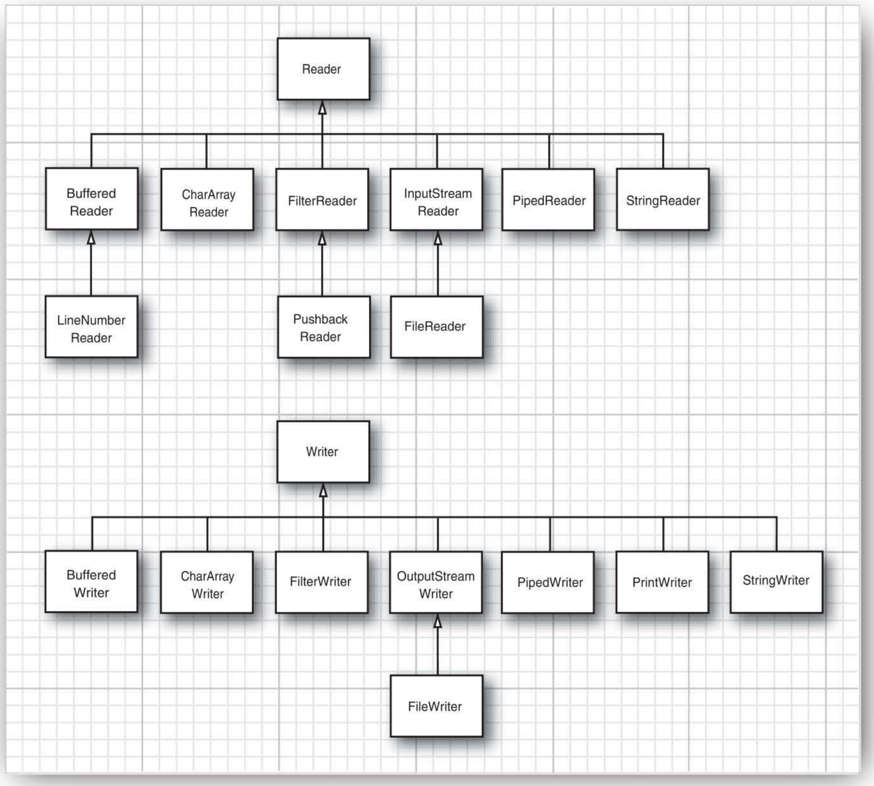

Figure 2.2 Reader and writer hierarchy

NOTE: The `java.io.Closeable` interface extends the `java.lang.AutoCloseable` interface. Therefore, you can use the `try`-with-resources statement with any `Closeable`. Why have two interfaces? The `close` method of the `Closeable` interface only throws an `IOException`, whereas the `AutoCloseable.close` method may throw any exception.

`OutputStream` and `Writer` implement the `Flushable` interface.

The `Readable` interface has a single method

```
int read(CharBuffer cb)
```

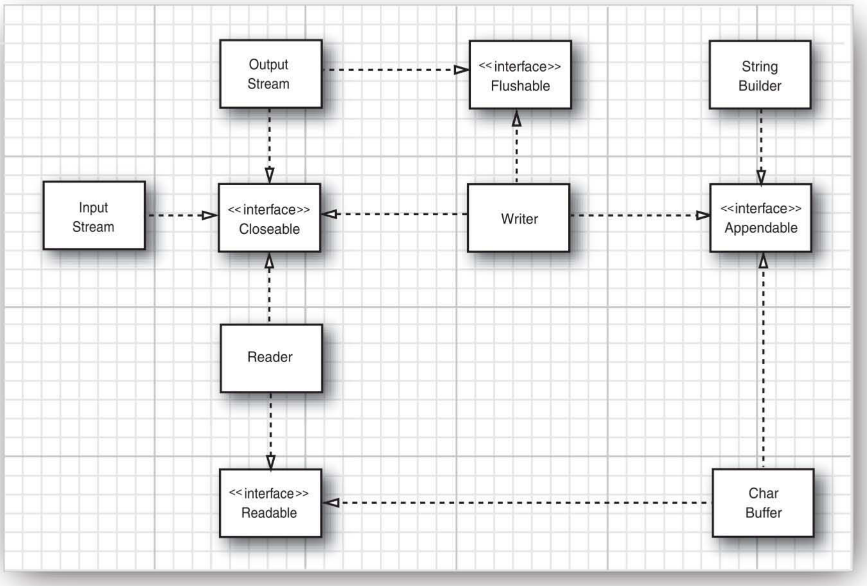

Figure 2.3 The `Closeable`, `Flushable`, `Readable`, and `Appendable` interfaces

The `CharBuffer` class has methods for sequential and random read/write access. It represents an in-memory buffer or a memory-mapped file. (See Section 2.5.2, "The Buffer Data Structure," on p. 132 for details.)

The `Appendable` interface has two methods for appending single characters and character sequences:

```
Appendable append(char c)
Appendable append(CharSequence s)
```

The `CharSequence` interface describes basic properties of a sequence of `char` values. It is implemented by `String`, `CharBuffer`, `StringBuilder`, and `StringBuffer`.

Of the input/output stream classes, only `Writer` implements `Appendable`.

java.io.Closeable 5.0

- `void close()`

  closes this `Closeable`. This method may throw an `IOException`.

java.io.Flushable 5.0

- `void flush()`

  flushes this `Flushable`.

java.lang.Readable 5.0

- `int read(CharBuffer cb)`

  attempts to read as many `char` values into `cb` as it can hold. Returns the number of values read, or `-1` if no further values are available from this `Readable`.

java.lang.Appendable 5.0

- `Appendable append(char c)`
- `Appendable append(CharSequence cs)`

  appends the given code unit, or all code units in the given sequence, to this `Appendable`; returns `this`.

java.lang.CharSequence 1.4

- `char charAt(int index)`

  returns the code unit at the given index.
- `int length()`

  returns the number of code units in this sequence.
- `CharSequence subSequence(int startIndex, int endIndex)`

  returns a `CharSequence` consisting of the code units stored from index `startIndex` to `endIndex - 1`.
- `String toString()`

  returns a string consisting of the code units of this sequence.

2.1.3 Combining Input/Output Stream Filters

`FileInputStream` and `FileOutputStream` give you input and output streams attached to a disk file. You need to pass the file name or full path name of the file to the constructor. For example,

```
var fin = new FileInputStream("employee.dat");
```

looks in the user directory for a file named `employee.dat`.

TIP: All the classes in `java.io` interpret relative path names as starting from the user's working directory. You can get this directory by a call to `System.getProperty("user.dir")`.

CAUTION: Since the backslash character is the escape character in Java strings, be sure to use \\ for Windows-style path names (for example, `C:\\Windows\\win.ini`). In Windows, you can also use a single forward slash (`C:/Windows/win.ini`) because most Windows file-handling system calls will interpret forward slashes as file separators. However, this is not recommended—the behavior of the Windows system functions is subject to change. Instead, for portable programs, use the file separator character for the platform on which your program runs. It is available as the constant string `java.io.File.separator`.

Like the abstract `InputStream` and `OutputStream` classes, these classes only support reading and writing at the byte level. That is, we can only read bytes and byte arrays from the object `fin`.

```
byte b = (byte) fin.read();
```

As you will see in the next section, if we just had a `DataInputStream`, we could read numeric types:

```
DataInputStream din = . . .;
double x = din.readDouble();
```

But just as the `FileInputStream` has no methods to read numeric types, the `DataInputStream` has no method to get data from a file.

Java uses a clever mechanism to separate two kinds of responsibilities. Some input streams (such as the `FileInputStream` and the input stream returned by the `openStream` method of the URL class) can retrieve bytes from files and other more exotic locations. Other input streams (such as the `DataInputStream`) can assemble bytes into more useful data types. The Java programmer has to combine the two. For example, to be able to read numbers from a file, first create a `FileInputStream` and then pass it to the constructor of a `DataInputStream`.

```
var fin = new FileInputStream("employee.dat");
var din = new DataInputStream(fin);
double x = din.readDouble();
```

If you look at Figure 2.1 again, you can see the classes `FilterInputStream` and `FilterOutputStream`. The subclasses of these classes are used to add capabilities to input/output streams that process bytes.

You can add multiple capabilities by nesting the filters. For example, by default, input streams are not buffered. That is, every call to `read` asks the operating system to dole out yet another byte. It is more efficient to request blocks of data instead and store them in a buffer. If you want buffering *and* the data input methods for a file, use the following rather monstrous sequence of constructors:

```
var din = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
```

Notice that we put the `DataInputStream` *last* in the chain of constructors because we want to use the `DataInputStream` methods, and we want *them* to use the buffered `read` method.

Sometimes you'll need to keep track of the intermediate input streams when chaining them together. For example, when reading input, you often need to peek at the next byte to see if it is the value that you expect. Java provides the `PushbackInputStream` for this purpose.

```
var pbin = new PushbackInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
```

Now you can speculatively read the next byte

```
int b = pbin.read();
```

and throw it back if it isn't what you wanted.

```
if (b != '<') pbin.unread(b);
```

However, reading and unreading are the *only* methods that apply to a pushback input stream. If you want to look ahead and also read numbers, then you need both a pushback input stream and a data input stream reference.

```
var pbin = new PushbackInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
var din = new DataInputStream(pbin);
```

Of course, in the input/output libraries of other programming languages, niceties such as buffering and lookahead are automatically taken care of—so it is a bit of a hassle to resort, in Java, to combining stream filters. However, the ability to mix and match filter classes to construct useful sequences of input/output streams does give you an immense amount of flexibility. For example, you can read numbers from a compressed ZIP file by using the following sequence of input streams (see Figure 2.4):

```
var zin = new ZipInputStream(new FileInputStream("employee.zip"));
var din = new DataInputStream(zin);
```

(See Section 2.2.3, "ZIP Archives," on p. 85 for more on Java's handling of ZIP files.)

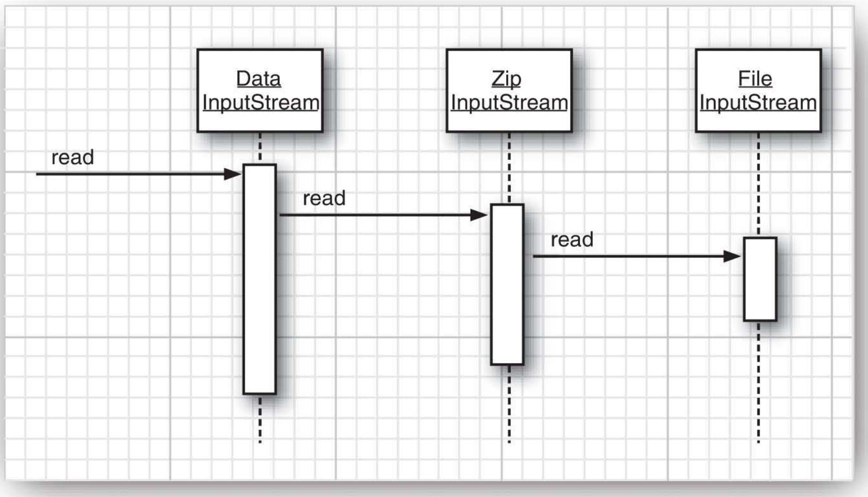

Figure 2.4 A sequence of filtered input streams

`java.io.FileInputStream` 1.0

- `FileInputStream(String name)`
- `FileInputStream(File file)`

  creates a new file input stream using the file whose path name is specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) Path names that are not absolute are resolved relative to the working directory that was set when the VM started.

java.io.FileOutputStream 1.0

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`
- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

  creates a new file output stream specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) If the `append` parameter is `true`, an existing file with the same name will not be deleted and data will be added at the end of the file. Otherwise, this method deletes any existing file with the same name.

java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

  creates a buffered input stream. A buffered input stream reads bytes from a stream without causing a device access every time. When the buffer is empty, a new block of data is read into the buffer.

java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

  creates a buffered output stream. A buffered output stream collects bytes to be written without causing a device access every time. When the buffer fills up or when the stream is flushed, the data are written.

java.io.PushbackInputStream 1.0

- `PushbackInputStream(InputStream in)`
- `PushbackInputStream(InputStream in, int size)`

  constructs an input stream with one-byte lookahead or a pushback buffer of specified size.

- `void unread(int b)`

  pushes back a byte, which is retrieved again by the next call to `read`.

2.1.4 Text Input and Output

When saving data, you have the choice between binary and text formats. For example, if the integer 1234 is saved in binary, it is written as the sequence of bytes `00 00 04 D2` (in hexadecimal notation). In text format, it is saved as the string `"1234"`. Although binary I/O is fast and efficient, it is not easily readable by humans. We first discuss text I/O and cover binary I/O in Section 2.2, "Reading and Writing Binary Data," on p. 78.

When saving text strings, you need to consider the *character encoding*. In the UTF-16 encoding that Java uses internally, the string `"José"` is encoded as `00 4A 00 6F 00 73 00 E9` (in hex). However, many programs expect that text files use a different encoding. In UTF-8, the encoding most commonly used on the Internet, the string would be written as `4A 6F 73 C3 A9`, without the zero bytes for the first three letters and with two bytes for the é character.

The `OutputStreamWriter` class turns an output stream of Unicode code units into a stream of bytes, using a chosen character encoding. Conversely, the `InputStreamReader` class turns an input stream that contains bytes (specifying characters in some character encoding) into a reader that emits Unicode code units.

For example, here is how you make an input reader that reads keystrokes from the console and converts them to Unicode:

```
var in = new InputStreamReader(System.in);
```

This input stream reader assumes the default character encoding used by the host system. On desktop operating systems, that can be an archaic encoding such as Windows 1252 or MacRoman. You should always choose a specific encoding in the constructor for the `InputStreamReader`, for example:

```
var in = new InputStreamReader(new FileInputStream("data.txt"), StandardCharsets.UTF_8);
```

See Section 2.1.8, "Character Encodings," on p. 75 for more information on character encodings.

The `Reader` and `Writer` classes have only basic methods to read and write individual characters. As with streams, you use subclasses for processing strings and numbers.

2.1.5 How to Write Text Output

For text output, use a `PrintWriter`. That class has methods to print strings and numbers in text format. In order to print to a file, construct a `PrintStream` from a file name and a character encoding:

```
var out = new PrintWriter("employee.txt", StandardCharsets.UTF_8);
```

To write to a print writer, use the same `print`, `println`, and `printf` methods that you used with `System.out`. You can use these methods to print numbers (`int`, `short`, `long`, `float`, `double`), characters, `boolean` values, strings, and objects.

For example, consider this code:

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

This writes the characters

```
Harry Hacker 75000.0
```

to the writer `out`. The characters are then converted to bytes and end up in the file `employee.txt`.

The `println` method adds the correct end-of-line character for the target system (`"\r\n"` on Windows, `"\n"` on UNIX) to the line. This is the string obtained by the call `System.getProperty("line.separator")`.

If the writer is set to *autoflush mode*, all characters in the buffer are sent to their destination whenever `println` is called. (Print writers are always buffered.) By default, autoflushing is *not* enabled. You can enable or disable autoflushing by using the `PrintWriter(Writer writer, boolean autoFlush)` constructor:

```
var out = new PrintWriter(
   new OutputStreamWriter(
      new FileOutputStream("employee.txt"), StandardCharsets.UTF_8),
   true); // autoflush
```

The `print` methods don't throw exceptions. You can call the `checkError` method to see if something went wrong with the output stream.

NOTE: Java veterans might wonder whatever happened to the `PrintStream` class and to `System.out`. In Java 1.0, the `PrintStream` class simply truncated all Unicode characters to ASCII characters by dropping the top byte. (At the time, Unicode was still a 16-bit encoding.) Clearly, that was not a clean or portable approach, and it was fixed with the introduction of readers and writers in Java 1.1. For compatibility with existing code, `System.in`, `System.out`, and `System.err` are still input/output streams, not readers and writers. But now the `PrintStream` class internally converts Unicode characters to the default host encoding in the same way the `PrintWriter` does. Objects of type `PrintStream` act exactly like print writers when you use the `print` and `println` methods, but unlike print writers they allow you to output raw bytes with the `write(int)` and `write(byte[])` methods.

java.io.PrintWriter 1.1

- `PrintWriter(OutputStream out)`
- `PrintWriter(Writer writer)`

  creates a new `PrintWriter` that writes to the given writer.
- `PrintWriter(String filename, String encoding)`
- `PrintWriter(File file, String encoding)`

  creates a new `PrintWriter` that writes to the given file, using the given character encoding.
- `void print(Object obj)`

  prints an object by printing the string resulting from `toString`.
- `void print(String s)`

  prints a string containing Unicode code units.
- `void println(String s)`

  prints a string followed by a line terminator. Flushes the output stream if it is in autoflush mode.
- `void print(char[] s)`

  prints all Unicode code units in the given array.
- `void print(char c)`

  prints a Unicode code unit.
- `void print(int i)`
- `void print(long l)`
- `void print(float f)`
- `void print(double d)`
- `void print(boolean b)`

  prints the given value in text format.
- `void printf(String format, Object... args)`

  prints the given values as specified by the format string. See Volume I, Chapter 3 for the specification of the format string.
- `boolean checkError()`

  returns `true` if a formatting or output error occurred. Once the output stream has encountered an error, it is tainted and all calls to `checkError` return `true`.

2.1.6 How to Read Text Input

The easiest way to process arbitrary text is the `Scanner` class that we used extensively in Volume I. You can construct a `Scanner` from any input stream.

Alternatively, you can read a short text file into a string like this:

```
var content = Files.readString(path, charset);
```

But if you want the file as a sequence of lines, call

```
List<String> lines = Files.readAllLines(path, charset);
```

If the file is large, process the lines lazily as a `Stream<String>`:

```
try (Stream<String> lines = Files.lines(path, charset))
{
   . . .
}
```

You can also use a scanner to read *tokens*—strings that are separated by a delimiter. The default delimiter is white space. You can change the delimiter to any regular expression. For example,

```
Scanner in = . . .;
in.useDelimiter("\\PL+");
```

accepts any non-Unicode letters as delimiters. The scanner then accepts tokens consisting only of Unicode letters.

Calling the `next` method yields the next token:

```
while (in.hasNext())
{
   String word = in.next();
   . . .
}
```

Alternatively, you can obtain a stream of all tokens as

```
Stream<String> words = in.tokens();
```

In early versions of Java, the only game in town for processing text input was the `BufferedReader` class. Its `readLine` method yields a line of text, or `null` when no more input is available. A typical input loop looks like this:

```
InputStream inputStream = . . .;
try (var in = new BufferedReader(new InputStreamReader(inputStream, charset)))
{
   String line;
   while ((line = in.readLine()) != null)
   {
      do something with line
   }
}
```

Nowadays, the `BufferedReader` class also has a `lines` method that yields a `Stream<String>`. However, unlike a `Scanner`, a `BufferedReader` has no methods for reading numbers.

2.1.7 Saving Objects in Text Format

In this section, we walk you through an example program that stores an array of `Employee` records in a text file. Each record is stored in a separate line. Instance fields are separated from each other by delimiters. We use a vertical bar (`|`) as our delimiter. (A colon (`:`) is another popular choice. Part of the fun is that everyone uses a different delimiter.) Naturally, we punt on the issue of what might happen if a `|` actually occurs in one of the strings we save.

Here is a sample set of records:

```
Harry Hacker|35500|1989-10-01
Carl Cracker|75000|1987-12-15
Tony Tester|38000|1990-03-15
```

Writing records is simple. Since we write to a text file, we use the `PrintWriter` class. We simply write all fields, followed by either a `|` or, for the last field, a newline character. This work is done in the following `writeData` method that we add to our `Employee` class:

```
public static void writeEmployee(PrintWriter out, Employee e)
{
   out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
}
```

To read records, we read in a line at a time and separate the fields. We use a scanner to read each line and then split the line into tokens with the `String.split` method.

```
public static Employee readEmployee(Scanner in)
{
   String line = in.nextLine();
   String[] tokens = line.split("\\|");
   String name = tokens[0];
   double salary = Double.parseDouble(tokens[1]);
   LocalDate hireDate = LocalDate.parse(tokens[2]);
   int year = hireDate.getYear();
   int month = hireDate.getMonthValue();
   int day = hireDate.getDayOfMonth();
   return new Employee(name, salary, year, month, day);
}
```

The parameter of the `split` method is a regular expression describing the separator. We discuss regular expressions in more detail at the end of this chapter. As it happens, the vertical bar character has a special meaning in regular expressions, so it needs to be escaped with a \ character. That character needs to be escaped by another \, yielding the "\\|" expression.

The complete program is in Listing 2.1. The static method

```
void writeData(Employee[] e, PrintWriter out)
```

first writes the length of the array, then writes each record. The static method

```
Employee[] readData(Scanner in)
```

first reads in the length of the array, then reads in each record. This turns out to be a bit tricky:

```
int n = in.nextInt();
in.nextLine(); // consume newline
var employees = new Employee[n];
for (int i = 0; i < n; i++)
{
   employees[i] = new Employee();
   employees[i].readData(in);
}
```

The call to `nextInt` reads the array length but not the trailing newline character. We must consume the newline so that the `readData` method can get the next input line when it calls the `nextLine` method.

Listing 2.1 `textFile/TextFileTest.java`

```
package textFile;

import java.io.*;
import java.nio.charset.*;
import java.time.*;
import java.util.*;

/**
 * @version 1.15 2018-03-17
 * @author Cay Horstmann
 */
public class TextFileTest
{
   public static void main(String[] args) throws IOException
   {
      var staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

(Continues)

Listing 2.1 *(Continued)*

```

      // save all employee records to the file employee.dat
      try (var out = new PrintWriter("employee.dat", StandardCharsets.UTF_8))
      {
         writeData(staff, out);
      }

      // retrieve all records into a new array
      try (var in = new Scanner(
            new FileInputStream("employee.dat"), "UTF-8"))
      {
         Employee[] newStaff = readData(in);

         // print the newly read employee records
         for (Employee e : newStaff)
            System.out.println(e);
      }
   }

   /**
    * Writes all employees in an array to a print writer
    * @param employees an array of employees
    * @param out a print writer
    */
   private static void writeData(Employee[] employees, PrintWriter out)
         throws IOException
   {
      // write number of employees
      out.println(employees.length);

      for (Employee e : employees)
         writeEmployee(out, e);
   }

   /**
    * Reads an array of employees from a scanner
    * @param in the scanner
    * @return the array of employees
    */
   private static Employee[] readData(Scanner in)
   {
      // retrieve the array size
      int n = in.nextInt();
      in.nextLine(); // consume newline

      var employees = new Employee[n];
      for (int i = 0; i < n; i++)
      {
         employees[i] = readEmployee(in);
      }
      return employees;
   }

   /**
    * Writes employee data to a print writer
    * @param out the print writer
    */
   public static void writeEmployee(PrintWriter out, Employee e)
   {
      out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
   }

   /**
    * Reads employee data from a buffered reader
    * @param in the scanner
    */
   public static Employee readEmployee(Scanner in)
   {
      String line = in.nextLine();
      String[] tokens = line.split("\\|");
      String name = tokens[0];
      double salary = Double.parseDouble(tokens[1]);
      LocalDate hireDate = LocalDate.parse(tokens[2]);
      int year = hireDate.getYear();
      int month = hireDate.getMonthValue();
      int day = hireDate.getDayOfMonth();
      return new Employee(name, salary, year, month, day);
   }
}
```

2.1.8 Character Encodings

Input and output streams are for sequences of bytes, but in many cases you will work with texts—that is, sequences of characters. It then matters how characters are encoded into bytes.

Java uses the Unicode standard for characters. Each character, or "code point," has a 21-bit integer number. There are different *character encodings*—methods for packaging those 21-bit numbers into bytes.

The most common encoding is UTF-8, which encodes each Unicode code point into a sequence of one to four bytes (see Table 2.1). UTF-8 has the advantage that the characters of the traditional ASCII character set, which contains all characters used in English, only take up one byte each.

Table 2.1 UTF-8 Encoding

Character Range	Encoding
`0. . .7F`	$0a_6a_5a_4a_3a_2a_1a_0$
`80. . .7FF`	$110a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
`800. . .FFFF`	$1110a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
`10000. . .10FFFF`	$11110a_{20}a_{19}a_{18}$ $10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$

Table 2.2 UTF-16 Encoding

Character Range	Encoding
`0. . .FFFF`	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`10000. . .10FFFF`	$110110b_{19}b_{18}$ $b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}$ $110111a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$ where $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

Another common encoding is UTF-16, which encodes each Unicode code point into one or two 16-bit values (see Table 2.2). This is the encoding used in Java strings. Actually, there are two forms of UTF-16, called "big-endian" and "little-endian." Consider the 16-bit value `0x2122`. In the big-endian format, the more significant byte comes first: `0x21` followed by `0x22`. In the little-endian format, it is the other way around: `0x22 0x21`. To indicate which of the two is used, a file can start with the "byte order mark," the 16-bit quantity `0xFEFF`. A reader can use this value to determine the byte order and then discard it.

CAUTION: Some programs, including Microsoft Notepad, add a byte order mark at the beginning of UTF-8 encoded files. Clearly, this is unnecessary since there are no byte ordering issues in UTF-8. But the Unicode standard allows it, and even suggests that it's a pretty good idea since it leaves little doubt about the encoding. It is supposed to be removed when reading a UTF-8 encoded file. Sadly, Java does not do that, and bug reports against this issue are closed as "will not fix." Your best bet is to strip out any leading `\uFEFF` that you find in your input.

In addition to the UTF encodings, there are partial encodings that cover a character range suitable for a given user population. For example, ISO 8859-1 is a one-byte code that includes accented characters used in Western European languages. Shift-JIS is a variable-length code for Japanese characters. A large number of these encodings are still in widespread use.

There is no reliable way to automatically detect the character encoding from a stream of bytes. Some API methods let you use the "default charset"—the character encoding preferred by the operating system of the computer. Is that the same encoding that is used by your source of bytes? These bytes may well originate from a different part of the world. Therefore, you should always explicitly specify the encoding. For example, when reading a web page, check the `Content-Type` header.

NOTE: The platform encoding is returned by the static method `Charset.defaultCharset`. The static method `Charset.availableCharsets` returns all available `Charset` instances, as a map from canonical names to `Charset` objects.

CAUTION: The Oracle implementation of Java has a system property `file.encoding` for overriding the platform default. This is not an officially supported property, and it is not consistently followed by all parts of Oracle's implementation of the Java library. You should not set it.

The `StandardCharsets` class has static variables of type `Charset` for the character encodings that every Java virtual machine must support:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

To obtain the `Charset` for another encoding, use the static `forName` method:

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

Use the `Charset` object when reading or writing text. For example, you can turn an array of bytes into a string as

```
var str = new String(bytes, StandardCharsets.UTF_8);
```

TIP: As of Java 10, all methods in the `java.io` package allow you to specify a character encoding with a `Charset` object or a string. Choose the `StandardCharsets` constants, so that any spelling errors are caught at compile time.

CAUTION: Some methods (such as the `String(byte[])` constructor) use the default platform encoding if you don't specify any; others (such as `Files.readAllLines`) use UTF-8.

2.2 Reading and Writing Binary Data

Text format is convenient for testing and debugging because it is humanly readable, but it is not as efficient as transmitting data in binary format. In the following sections, you will learn how to perform input and output with binary data.

2.2.1 The `DataInput` and `DataOutput` interfaces

The `DataOutput` interface defines the following methods for writing a number, a character, a `boolean` value, or a string in binary format:

```
writeChars        writeFloat
writeByte         writeDouble
writeInt          writeChar
writeShort        writeBoolean
writeLong         writeUTF
```

For example, `writeInt` always writes an integer as a 4-byte binary quantity regardless of the number of digits, and `writeDouble` always writes a `double` as an 8-byte binary quantity. The resulting output is not human-readable, but it will use the same space for each value of a given type and reading it back in will be faster than parsing text.

NOTE: There are two different methods of storing integers and floating-point numbers in memory, depending on the processor you are using. Suppose, for example, you are working with a 4-byte `int`, such as the decimal number 1234, or 4D2 in hexadecimal ($1234 = 4 \times 256 + 13 \times 16 + 2$). This value can be stored in such a way that the first of the four bytes in memory holds the most significant byte (MSB) of the value: `00 00 04 D2`. This is the so-called big-endian method. Or, we can start with the least significant byte (LSB) first: `D2 04 00 00`. This is called, naturally enough, the little-endian method. For example, the SPARC uses big-endian; the Pentium, little-endian. This can lead to problems. When a file is saved from C or C++ file, the data are saved exactly as the processor stores them. That makes it challenging to move even the simplest data files from one platform to another. In Java, all values are written in the big-endian fashion, regardless of the processor. That makes Java data files platform-independent.

The `writeUTF` method writes string data using a modified version of the 8-bit Unicode Transformation Format. Instead of simply using the standard UTF-8 encoding, sequences of Unicode code units are first represented in UTF-16, and then the result is encoded using the UTF-8 rules. This modified encoding is different for characters with codes higher than `0xFFFF`. It is used for backward compatibility with virtual machines that were built when Unicode had not yet grown beyond 16 bits.

Since nobody else uses this modification of UTF-8, you should only use the `writeUTF` method to write strings intended for a Java virtual machine—for example, in a program that generates bytecodes. Use the `writeChars` method for other purposes.

To read the data back in, use the following methods defined in the `DataInput` interface:

```
readInt          readDouble
readShort        readChar
readLong         readBoolean
readFloat        readUTF
```

The `DataInputStream` class implements the `DataInput` interface. To read binary data from a file, combine a `DataInputStream` with a source of bytes such as a `FileInputStream`:

```
var in = new DataInputStream(new FileInputStream("employee.dat"));
```

Similarly, to write binary data, use the `DataOutputStream` class that implements the `DataOutput` interface:

```
var out = new DataOutputStream(new FileOutputStream("employee.dat"));
```

java.io.DataInput 1.0

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`
- `short readShort()`

  reads in a value of the given type.
- `void readFully(byte[] b)`

  reads bytes into the array `b`, blocking until all bytes are read.
- `void readFully(byte[] b, int off, int len)`

  places up to `len` bytes into the array `b`, starting at `off`, blocking until all bytes are read.

(Continues)

java.io.DataInput **1.0** *(Continued)*

- `String readUTF()`

  reads a string of characters in the "modified UTF-8" format.
- `int skipBytes(int n)`

  skips `n` bytes, blocking until all bytes are skipped.

java.io.DataOutput **1.0**

- `void writeBoolean(boolean b)`
- `void writeByte(int b)`
- `void writeChar(int c)`
- `void writeDouble(double d)`
- `void writeFloat(float f)`
- `void writeInt(int i)`
- `void writeLong(long l)`
- `void writeShort(int s)`

  writes a value of the given type.
- `void writeChars(String s)`

  writes all characters in the string.
- `void writeUTF(String s)`

  writes a string of characters in the "modified UTF-8" format.

2.2.2 Random-Access Files

The `RandomAccessFile` class lets you read or write data anywhere in a file. Disk files are random-access, but input/output streams that communicate with a network socket are not. You can open a random-access file either for reading only or for both reading and writing; specify the option by using the string `"r"` (for read access) or `"rw"` (for read/write access) as the second argument in the constructor.

```
var in = new RandomAccessFile("employee.dat", "r");
var inOut = new RandomAccessFile("employee.dat", "rw");
```

When you open an existing file as a `RandomAccessFile`, it does not get deleted.

A random-access file has a *file pointer* that indicates the position of the next byte to be read or written. The `seek` method can be used to set the file pointer to an arbitrary byte position within the file. The argument to `seek` is a `long` integer between zero and the length of the file in bytes.

The `getFilePointer` method returns the current position of the file pointer.

The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces. To read and write from a random-access file, use methods such as `readInt`/`writeInt` and `readChar`/`writeChar` that we discussed in the preceding section.

Let's walk through an example program that stores employee records in a random-access file. Each record will have the same size. This makes it easy to read an arbitrary record. Suppose you want to position the file pointer to the third record. Simply set the file pointer to the appropriate byte position and start reading.

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
var e = new Employee();
e.readData(in);
```

If you want to modify the record and save it back into the same location, remember to set the file pointer back to the beginning of the record:

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

To determine the total number of bytes in a file, use the `length` method. The total number of records is the length divided by the size of each record.

```
long nbytes = in.length(); // length in bytes
int nrecords = (int) (nbytes / RECORD_SIZE);
```

Integers and floating-point values have a fixed size in binary format, but we have to work harder for strings. We provide two helper methods to write and read strings of a fixed size.

The `writeFixedString` writes the specified number of code units, starting at the beginning of the string. If there are too few code units, the method pads the string, using zero values.

```
public static void writeFixedString(String s, int size, DataOutput out)
      throws IOException
{
   for (int i = 0; i < size; i++)
   {
      char ch = 0;
      if (i < s.length()) ch = s.charAt(i);
      out.writeChar(ch);
   }
}
```

The `readFixedString` method reads characters from the input stream until it has consumed `size` code units or until it encounters a character with a zero value.

Then, it skips past the remaining zero values in the input field. For added efficiency, this method uses the `StringBuilder` class to read in a string.

```
public static String readFixedString(int size, DataInput in)
      throws IOException
{
   var b = new StringBuilder(size);
   int i = 0;
   var done = false;
   while (!done && i < size)
   {
      char ch = in.readChar();
      i++;
      if (ch == 0) done = true;
      else b.append(ch);
   }
   in.skipBytes(2 * (size - i));
   return b.toString();
}
```

We placed the `writeFixedString` and `readFixedString` methods inside the `DataIO` helper class.

To write a fixed-size record, we simply write all fields in binary.

```
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());
```

Reading the data back is just as simple.

```
String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
double salary = in.readDouble();
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();
```

Let us compute the size of each record. We will use 40 characters for the name strings. Therefore, each record will contain 100 bytes:

- 40 characters = 80 bytes for the name
- 1 `double` = 8 bytes for the salary
- 3 `int` = 12 bytes for the date

The program shown in Listing 2.2 writes three records into a data file and then reads them from the file in reverse order. To do this efficiently requires random access—we need to get to the last record first.

Listing 2.2 randomAccess/RandomAccessTest.java

```java
package randomAccess;

import java.io.*;
import java.time.*;

/**
 * @version 1.14 2018-05-01
 * @author Cay Horstmann
 */
public class RandomAccessTest
{
   public static void main(String[] args) throws IOException
   {
      var staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try (var out = new DataOutputStream(new FileOutputStream("employee.dat")))
      {
         // save all employee records to the file employee.dat
         for (Employee e : staff)
            writeData(out, e);
      }

      try (var in = new RandomAccessFile("employee.dat", "r"))
      {
         // retrieve all records into a new array

         // compute the array size
         int n = (int)(in.length() / Employee.RECORD_SIZE);
         var newStaff = new Employee[n];

         // read employees in reverse order
         for (int i = n - 1; i >= 0; i--)
         {
            newStaff[i] = new Employee();
            in.seek(i * Employee.RECORD_SIZE);
            newStaff[i] = readData(in);
         }

         // print the newly read employee records
         for (Employee e : newStaff)
            System.out.println(e);
      }
   }
```

(Continues)

Listing 2.2 *(Continued)*

```java

   /**
    * Writes employee data to a data output
    * @param out the data output
    * @param e the employee
    */
   public static void writeData(DataOutput out, Employee e) throws IOException
   {
      DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
      out.writeDouble(e.getSalary());

      LocalDate hireDay = e.getHireDay();
      out.writeInt(hireDay.getYear());
      out.writeInt(hireDay.getMonthValue());
      out.writeInt(hireDay.getDayOfMonth());
   }

   /**
    * Reads employee data from a data input
    * @param in the data input
    * @return the employee
    */
   public static Employee readData(DataInput in) throws IOException
   {
      String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
      double salary = in.readDouble();
      int y = in.readInt();
      int m = in.readInt();
      int d = in.readInt();
      return new Employee(name, salary, y, m - 1, d);
   }
}
```

`java.io.RandomAccessFile` 1.0

- `RandomAccessFile(String file, String mode)`
- `RandomAccessFile(File file, String mode)`

  opens the given file for random access. The `mode` string is `"r"` for read-only mode, `"rw"` for read/write mode, `"rws"` for read/write mode with synchronous disk writes of data and metadata for every update, and `"rwd"` for read/write mode with synchronous disk writes of data only.

- `long getFilePointer()`

  returns the current location of the file pointer.

(Continues)

java.io.RandomAccessFile 1.0 *(Continued)*

- `void seek(long pos)`

  sets the file pointer to `pos` bytes from the beginning of the file.
- `long length()`

  returns the length of the file in bytes.

2.2.3 ZIP Archives

ZIP archives store one or more files in a (usually) compressed format. Each ZIP archive has a header with information such as the name of each file and the compression method that was used. In Java, you can use a `ZipInputStream` to read a ZIP archive. You need to look at the individual *entries* in the archive. The `getNextEntry` method returns an object of type `ZipEntry` that describes the entry. Read from the stream until the end, which is actually the end of the current entry. Then call `closeEntry` to read the next entry. Do not close `zin` until you read the last entry. Here is a typical code sequence to read through a ZIP file:

```
var zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
   read the contents of zin
   zin.closeEntry();
}
zin.close();
```

To write a ZIP file, use a `ZipOutputStream`. For each entry that you want to place into the ZIP file, create a `ZipEntry` object. Pass the file name to the `ZipEntry` constructor; it sets the other parameters such as file date and decompression method. You can override these settings if you like. Then, call the `putNextEntry` method of the `ZipOutputStream` to begin writing a new file. Send the file data to the ZIP output stream. When done, call `closeEntry`. Repeat for all the files you want to store. Here is a code skeleton:

```
var fout = new FileOutputStream("test.zip");
var zout = new ZipOutputStream(fout);
for all files
{
   var ze = new ZipEntry(filename);
   zout.putNextEntry(ze);
   send data to zout
   zout.closeEntry();
}
zout.close();
```

NOTE: JAR files (which were discussed in Volume I, Chapter 4) are simply ZIP files with a special entry—the so-called manifest. Use the `JarInputStream` and `JarOutputStream` classes to read and write the manifest entry.

ZIP input streams are a good example of the power of the stream abstraction. When you read data stored in compressed form, you don't need to worry that the data are being decompressed as they are being requested. Moreover, the source of the bytes in a ZIP stream need not be a file—the ZIP data can come from a network connection.

NOTE: Section 2.4.8, "ZIP File Systems," on p. 123 shows how to access a ZIP archive without a special API, using the `FileSystem` class of Java 7.

`java.util.zip.ZipInputStream` 1.1

- `ZipInputStream(InputStream in)`

  creates a `ZipInputStream` that allows you to inflate data from the given `InputStream`.
- `ZipEntry getNextEntry()`

  returns a `ZipEntry` object for the next entry, or `null` if there are no more entries.
- `void closeEntry()`

  closes the current open entry in the ZIP file. You can then read the next entry by using `getNextEntry()`.

`java.util.zip.ZipOutputStream` 1.1

- `ZipOutputStream(OutputStream out)`

  creates a `ZipOutputStream` that you can use to write compressed data to the specified `OutputStream`.
- `void putNextEntry(ZipEntry ze)`

  writes the information in the given `ZipEntry` to the output stream and positions the stream for the data. The data can then be written by calling the `write()` method.

(Continues)

`java.util.zip.ZipOutputStream` 1.1 *(Continued)*

- `void closeEntry()`

  closes the currently open entry in the ZIP file. Use the `putNextEntry` method to start the next entry.
- `void setLevel(int level)`

  sets the default compression level of subsequent `DEFLATED` entries to a value from `Deflater.NO_COMPRESSION` to `Deflater.BEST_COMPRESSION`. The default value is `Deflater.DEFAULT_COMPRESSION`. Throws an `IllegalArgumentException` if the level is not valid.
- `void setMethod(int method)`

  sets the default compression method for this `ZipOutputStream` for any entries that do not specify a method; can be either `DEFLATED` or `STORED`.

`java.util.zip.ZipEntry` 1.1

- `ZipEntry(String name)`

  constructs a ZIP entry with a given name.
- `long getCrc()`

  returns the CRC32 checksum value for this `ZipEntry`.
- `String getName()`

  returns the name of this entry.
- `long getSize()`

  returns the uncompressed size of this entry, or `-1` if the uncompressed size is not known.
- `boolean isDirectory()`

  returns `true` if this entry is a directory.
- `void setMethod(int method)`

  sets the compression method for the entry to `DEFLATED` or `STORED`.
- `void setSize(long size)`

  sets the size of this entry. Only required if the compression method is `STORED`.
- `void setCrc(long crc)`

  sets the CRC32 checksum of this entry. Use the `CRC32` class to compute this checksum. Only required if the compression method is `STORED`.

java.util.zip.ZipFile 1.1

- `ZipFile(String name)`
- `ZipFile(File file)`

  creates a `ZipFile` for reading from the given string or `File` object.
- `Enumeration entries()`

  returns an `Enumeration` object that enumerates the `ZipEntry` objects that describe the entries of the `ZipFile`.
- `ZipEntry getEntry(String name)`

  returns the entry corresponding to the given name, or `null` if there is no such entry.
- `InputStream getInputStream(ZipEntry ze)`

  returns an `InputStream` for the given entry.
- `String getName()`

  returns the path of this ZIP file.

2.3 Object Input/Output Streams and Serialization

Using a fixed-length record format is a good choice if you need to store data of the same type. However, objects that you create in an object-oriented program are rarely all of the same type. For example, you might have an array called `staff` that is nominally an array of `Employee` records but contains objects that are actually instances of a subclass such as `Manager`.

It is certainly possible to come up with a data format that allows you to store such polymorphic collections—but, fortunately, we don't have to. The Java language supports a very general mechanism, called *object serialization*, that makes it possible to write any object to an output stream and read it again later. (You will see in this chapter where the term "serialization" comes from.)

2.3.1 Saving and Loading Serializable Objects

To save object data, you first need to open an `ObjectOutputStream` object:

```
var out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

Now, to save an object, simply use the `writeObject` method of the `ObjectOutputStream` class as in the following fragment:

```
var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
var boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
```

```
out.writeObject(harry);
out.writeObject(boss);
```

To read the objects back in, first get an `ObjectInputStream` object:

```
var in = new ObjectInputStream(new FileInputStream("employee.dat"));
```

Then, retrieve the objects in the same order in which they were written, using the `readObject` method:

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

There is, however, one change you need to make to any class that you want to save to an output stream and restore from an object input stream. The class must implement the `Serializable` interface:

```
class Employee implements Serializable { . . . }
```

The `Serializable` interface has no methods, so you don't need to change your classes in any way. In this regard, it is similar to the `Cloneable` interface that we discussed in Volume I, Chapter 6. However, to make a class cloneable, you still had to override the `clone` method of the `Object` class. To make a class serializable, you do not need to do anything else.

NOTE: You can write and read only *objects* with the `writeObject`/`readObject` methods. For primitive type values, use methods such as `writeInt`/`readInt` or `writeDouble`/`readDouble`. (The object input/output stream classes implement the `DataInput`/`DataOutput` interfaces.)

Behind the scenes, an `ObjectOutputStream` looks at all the fields of the objects and saves their contents. For example, when writing an `Employee` object, the name, date, and salary fields are written to the output stream.

However, there is one important situation to consider: What happens when one object is shared by several objects as part of their state?

To illustrate the problem, let us make a slight modification to the `Manager` class. Let's assume that each manager has a secretary:

```
class Manager extends Employee
{
   private Employee secretary;
   . . .
}
```

Each `Manager` object now contains a reference to an `Employee` object that describes the secretary. Of course, two managers can share the same secretary, as is the case in Figure 2.5 and the following code:

```
var harry = new Employee("Harry Hacker", . . .);
var carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
var tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

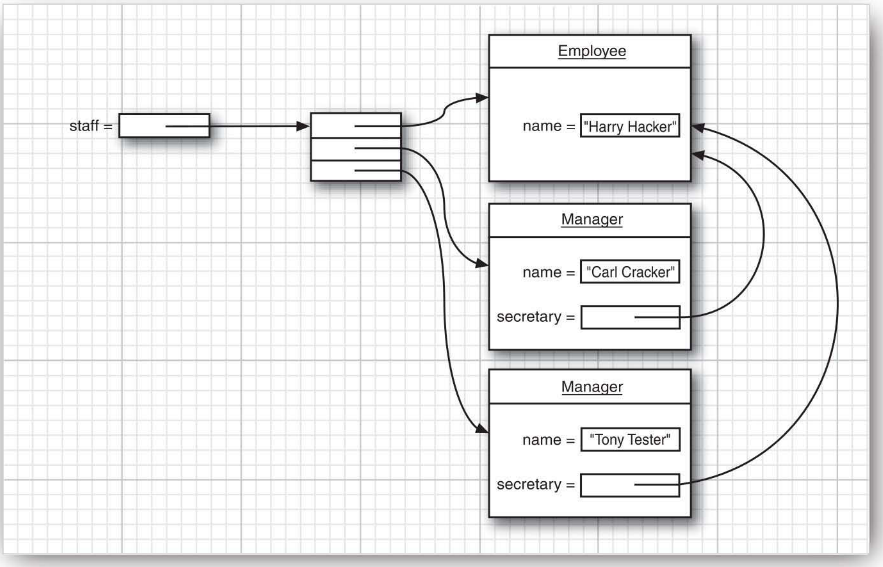

Figure 2.5 Two managers can share a mutual employee.

Saving such a network of objects is a challenge. Of course, we cannot save and restore the memory addresses for the secretary objects. When an object is reloaded, it will likely occupy a completely different memory address than it originally did.

Instead, each object is saved with the *serial number*—hence the name *object serialization* for this mechanism. Here is the algorithm:

1. Associate a serial number with each object reference that you encounter (as shown in Figure 2.6).

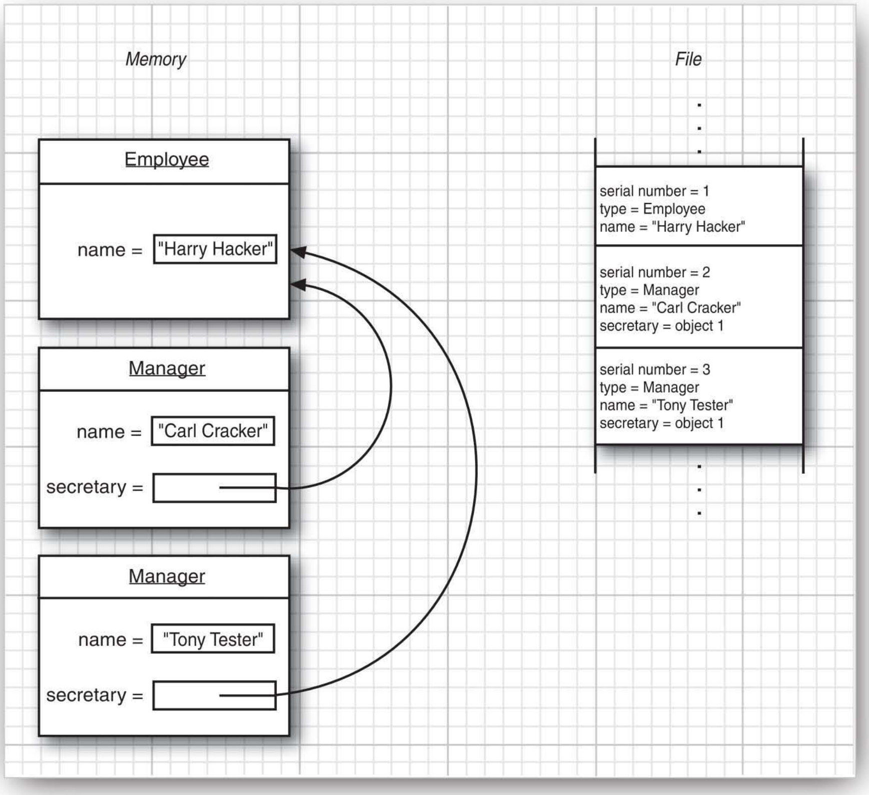

Figure 2.6 An example of object serialization

2. When encountering an object reference for the first time, save the object data to the output stream.
3. If it has been saved previously, just write "same as the previously saved object with serial number *x*."

When reading the objects back, the procedure is reversed.

1. When an object is specified in an object input stream for the first time, construct it, initialize it with the stream data, and remember the association between the serial number and the object reference.
2. When the tag "same as the previously saved object with serial number *x*" is encountered, retrieve the object reference for the sequence number.

NOTE: In this chapter, we will use serialization to save a collection of objects to a disk file and retrieve it exactly as we stored it. Another very important application is the transmittal of a collection of objects across a network connection to another computer. Just as raw memory addresses are meaningless in a file, they are also meaningless when you communicate with a different processor. By replacing memory addresses with serial numbers, serialization permits the transport of object collections from one machine to another.

Listing 2.3 is a program that saves and reloads a network of `Employee` and `Manager` objects (some of which share the same employee as a secretary). Note that the secretary object is unique after reloading—when `newStaff[1]` gets a raise, that is reflected in the `secretary` fields of the managers.

Listing 2.3 `objectStream/ObjectStreamTest.java`

```java
package objectStream;

import java.io.*;

/**
 * @version 1.11 2018-05-01
 * @author Cay Horstmann
 */
class ObjectStreamTest
{
   public static void main(String[] args) throws IOException, ClassNotFoundException
   {
      var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      var carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      carl.setSecretary(harry);
      var tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
      tony.setSecretary(harry);

      var staff = new Employee[3];

      staff[0] = carl;
      staff[1] = harry;
      staff[2] = tony;

      // save all employee records to the file employee.dat
      try (var out = new ObjectOutputStream(new FileOutputStream("employee.dat")))
      {
         out.writeObject(staff);
      }

      try (var in = new ObjectInputStream(new FileInputStream("employee.dat")))
      {
         // retrieve all records into a new array

         var newStaff = (Employee[]) in.readObject();

         // raise secretary's salary
         newStaff[1].raiseSalary(10);

         // print the newly read employee records
         for (Employee e : newStaff)
            System.out.println(e);
      }
   }
}
```

`java.io.ObjectOutputStream` 1.1

- `ObjectOutputStream(OutputStream out)`

  creates an `ObjectOutputStream` so you can write objects to the specified `OutputStream`.
- `void writeObject(Object obj)`

  writes the specified object to the `ObjectOutputStream`. This method saves the class of the object, the signature of the class, and the values of any nonstatic, nontransient fields of the class and its superclasses.

`java.io.ObjectInputStream` 1.1

- `ObjectInputStream(InputStream in)`

  creates an `ObjectInputStream` to read back object information from the specified `InputStream`.
- `Object readObject()`

  reads an object from the `ObjectInputStream`. In particular, this method reads back the class of the object, the signature of the class, and the values of the non-transient and nonstatic fields of the class and all its superclasses. It does deserializing so that multiple object references can be recovered.

2.3.2 Understanding the Object Serialization File Format

Object serialization saves object data in a particular file format. Of course, you can use the `writeObject`/`readObject` methods without having to know the exact sequence of bytes that represents objects in a file. Nonetheless, we found studying the data format extremely helpful for gaining insight into the object serialization process. As the details are somewhat technical, feel free to skip this section if you are not interested in the implementation.

Every file begins with the two-byte "magic number"

```
AC ED
```

followed by the version number of the object serialization format, which is currently

```
00 05
```

(We use hexadecimal numbers throughout this section to denote bytes.) Then, it contains a sequence of objects, in the order in which they were saved.

String objects are saved as

`74`	two-byte length	characters

For example, the string "Harry" is saved as

```
74 00 05 Harry
```

The Unicode characters of the string are saved in the "modified UTF-8" format.

When an object is saved, the class of that object must be saved as well. The class description contains

- The name of the class
- The *serial version unique ID*, which is a fingerprint of the data field types and method signatures
- A set of flags describing the serialization method
- A description of the data fields

The fingerprint is obtained by ordering the descriptions of the class, superclass, interfaces, field types, and method signatures in a canonical way, and then applying the so-called Secure Hash Algorithm (SHA) to that data.

SHA is a fast algorithm that gives a "fingerprint" of a larger block of information. This fingerprint is always a 20-byte data packet, regardless of the size of the original data. It is created by a clever sequence of bit operations on the data that makes it essentially 100 percent certain that the fingerprint will change if the information is altered in any way. (For more details on SHA, see, for example, *Cryptography and Network Security, Seventh Edition* by William Stallings, Prentice Hall, 2016.) However, the serialization mechanism uses only the first eight bytes of the SHA code as a class fingerprint. It is still very likely that the class fingerprint will change if the data fields or methods change.

When reading an object, its fingerprint is compared against the current fingerprint of the class. If they don't match, it means the class definition has changed after the object was written, and an exception is generated. Of course, in practice, classes do evolve, and it might be necessary for a program to read in older versions of objects. We will discuss this in Section 2.3.5, "Versioning," on p. 103.

Here is how a class identifier is stored:

- `72`
- 2-byte length of class name
- Class name
- 8-byte fingerprint
- 1-byte flag
- 2-byte count of data field descriptors
- Data field descriptors
- `78` (end marker)
- Superclass type (`70` if none)

The flag byte is composed of three bit masks, defined in `java.io.ObjectStreamConstants`:

```
static final byte SC_WRITE_METHOD = 1;
   // class has a writeObject method that writes additional data
static final byte SC_SERIALIZABLE = 2;
   // class implements the Serializable interface
static final byte SC_EXTERNALIZABLE = 4;
   // class implements the Externalizable interface
```

We discuss the `Externalizable` interface later in this chapter. Externalizable classes supply custom read and write methods that take over the output of their instance fields. The classes that we write implement the `Serializable` interface and will have a flag value of `02`. The serializable `java.util.Date` class defines its own `readObject`/`writeObject` methods and has a flag of `03`.

Each data field descriptor has the format:

- 1-byte type code
- 2-byte length of field name
- Field name
- Class name (if the field is an object)

The type code is one of the following:

`B`	`byte`
`C`	`char`
`D`	`double`
`F`	`float`
`I`	`int`
`J`	`long`
`L`	object
`S`	`short`
`Z`	`boolean`
`[`	array

When the type code is `L`, the field name is followed by the field type. Class and field name strings do not start with the string code `74`, but field types do. Field types use a slightly different encoding of their names—namely, the format used by native methods.

For example, the `salary` field of the `Employee` class is encoded as

```
D 00 06 salary
```

Here is the complete class descriptor of the `Employee` class:

`72 00 08 Employee`	
`E6 D2 86 7D AE AC 18 1B 02`	Fingerprint and flags
`00 03`	Number of instance fields
`D 00 06 salary`	Instance field type and name
`L 00 07 hireDay`	Instance field type and name
`74 00 10 Ljava/util/Date;`	Instance field class name: `Date`
`L 00 04 name`	Instance field type and name
`74 00 12 Ljava/lang/String;`	Instance field class name: `String`
`78`	End marker
`70`	No superclass

These descriptors are fairly long. If the *same* class descriptor is needed again in the file, an abbreviated form is used:

`71`	4-byte serial number

The serial number refers to the previous explicit class descriptor. We discuss the numbering scheme later.

An object is stored as

`73`	class descriptor	object data

For example, here is how an Employee object is stored:

`40 E8 6A 00 00 00 00 00`	`salary` field value: `double`
`73`	`hireDay` field value: new object
`  71 00 7E 00 08`	Existing class `java.util.Date`
`  77 08 00 00 00 91 1B 4E B1 80 78`	External storage (details later)
`74 00 0C Harry Hacker`	`name` field value: `String`

As you can see, the data file contains enough information to restore the Employee object.

Arrays are saved in the following format:

`75`	class descriptor	4-byte number of entries	entries

The array class name in the class descriptor is in the same format as that used by native methods (which is slightly different from the format used by class names in other class descriptors). In this format, class names start with an L and end with a semicolon.

For example, an array of three Employee objects starts out like this:

`75`	Array
`  72 00 0B [LEmployee;`	New class, string length, class name `Employee[]`
`    FC BF 36 11 C5 91 11 C7 02`	Fingerprint and flags
`    00 00`	Number of instance fields
`    78`	End marker
`    70`	No superclass
`    00 00 00 03`	Number of array entries

Note that the fingerprint for an array of Employee objects is different from a fingerprint of the Employee class itself.

All objects (including arrays and strings) and all class descriptors are given serial numbers as they are saved in the output file. The numbers start at 00 7E 00 00.

We already saw that a full class descriptor for any given class occurs only once. Subsequent descriptors refer to it. For example, in our previous example, a repeated reference to the Date class was coded as

```
71 00 7E 00 08
```

The same mechanism is used for objects. If a reference to a previously saved object is written, it is saved in exactly the same way—that is, 71 followed by the serial number. It is always clear from the context whether a particular serial reference denotes a class descriptor or an object.

Finally, a null reference is stored as

```
70
```

Here is the commented output of the ObjectRefTest program of the preceding section. Run the program, look at a hex dump of its data file employee.dat, and compare it with the commented listing. The important lines toward the end of the output show a reference to a previously saved object.

```
AC ED 00 05                       File header
75                                Array staff (serial #1)
   72 00 0B [LEmployee;           New class, string length, class name
                                  Employee[] (serial #0)
      FC BF 36 11 C5 91 11 C7 02  Fingerprint and flags
      00 00                       Number of instance fields
      78                          End marker
      70                          No superclass
      00 00 00 03                 Number of array entries
73                                staff[0]— new object (serial #7)
   72 00 07 Manager               New class, string length, class name
                                  (serial #2)
      36 06 AE 13 63 8F 59 B7 02  Fingerprint and flags
      00 01                       Number of data fields
      L 00 09 secretary           Instance field type and name
      74 00 0A LEmployee;         Instance field class name: String (serial #3)
      78                          End marker
      72 00 08 Employee           Superclass: new class, string length, class
                                  name (serial #4)
         E6 D2 86 7D AE AC 18 1B 02  Fingerprint and flags
         00 03                       Number of instance fields
         D 00 06 salary              Instance field type and name
         L 00 07 hireDay             Instance field type and name
         74 00 10 Ljava/util/Date;   Instance field class name: String (serial #5)
         L 00 04 name                Instance field type and name
         74 00 12 Ljava/lang/String; Instance field class name: String (serial #6)
         78                          End marker
         70                          No superclass
```

```
40 F3 88 00 00 00 00 00           salary field value: double
73                                hireDay field value: new object (serial #9)
  72 00 0E java.util.Date         New class, string length, class name
                                  (serial #8)
    68 6A 81 01 4B 59 74 19 03    Fingerprint and flags
    00 00                         No instance variables
    78                            End marker
    70                            No superclass
  77 08                           External storage, number of bytes
  00 00 00 83 E9 39 E0 00         Date
  78                              End marker
74 00 0C Carl Cracker             name field value: String (serial #10)
73                                secretary field value: new object
                                  (serial #11)
  71 00 7E 00 04                  existing class (use serial #4)
  40 E8 6A 00 00 00 00 00         salary field value: double
  73                              hireDay field value: new object (serial #12)
    71 00 7E 00 08                Existing class (use serial #8)
    77 08                         External storage, number of bytes
    00 00 00 91 1B 4E B1 80       Date
    78                            End marker
    74 00 0C Harry Hacker         name field value: String (serial #13)
71 00 7E 00 0B                    staff[1]: existing object (use serial #11)
73                                staff[2]: new object (serial #14)
  71 00 7E 00 02                  Existing class (use serial #2)
  40 E3 88 00 00 00 00 00         salary field value: double
  73                              hireDay field value: new object (serial #15)
    71 00 7E 00 08                Existing class (use serial #8)
    77 08                         External storage, number of bytes
    00 00 00 94 6D 3E EC 00 00    Date
    78                            End marker
  74 00 0B Tony Tester            name field value: String (serial #16)
  71 00 7E 00 0B                  secretary field value: existing object (use
                                  serial #11)
```

Of course, studying these codes can be about as exciting as reading a phone book. It is not important to know the exact file format (unless you are trying to create an evil effect by modifying the data), but it is still instructive to know that the serialized format has a detailed description of all the objects it contains, with sufficient detail to allow reconstruction of both objects and arrays of objects.

What you should remember is this:

- The serialized format contains the types and data fields of all objects.
- Each object is assigned a serial number.
- Repeated occurrences of the same object are stored as references to that serial number.

2.3.3 Modifying the Default Serialization Mechanism

Certain data fields should never be serialized—for example, integer values that store file handles or handles of windows that are only meaningful to native methods. Such information is guaranteed to be useless when you reload an object at a later time or transport it to a different machine. In fact, improper values for such fields can actually cause native methods to crash. Java has an easy mechanism to prevent such fields from ever being serialized: Mark them with the keyword `transient`. You also need to tag fields as `transient` if they belong to nonserializable classes. Transient fields are always skipped when objects are serialized.

The serialization mechanism provides a way for individual classes to add validation or any other desired action to the default read and write behavior. A serializable class can define methods with the signature

```
private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
      throws IOException;
```

Then, the data fields are no longer automatically serialized—these methods are called instead.

Here is a typical example. A number of classes in the `java.awt.geom` package, such as `Point2D.Double`, are not serializable. Now, suppose you want to serialize a class `LabeledPoint` that stores a `String` and a `Point2D.Double`. First, you need to mark the `Point2D.Double` field as `transient` to avoid a `NotSerializableException`.

```
public class LabeledPoint implements Serializable
{
   private String label;
   private transient Point2D.Double point;
   . . .
}
```

In the writeObject method, we first write the object descriptor and the String field, label, by calling the defaultWriteObject method. This is a special method of the ObjectOutputStream class that can only be called from within a writeObject method of a serializable class. Then we write the point coordinates, using the standard DataOutput calls.

```
private void writeObject(ObjectOutputStream out)
      throws IOException
{
   out.defaultWriteObject();
   out.writeDouble(point.getX());
   out.writeDouble(point.getY());
}
```

In the readObject method, we reverse the process:

```
private void readObject(ObjectInputStream in)
      throws IOException
{
   in.defaultReadObject();
   double x = in.readDouble();
   double y = in.readDouble();
   point = new Point2D.Double(x, y);
}
```

Another example is the java.util.Date class that supplies its own readObject and writeObject methods. These methods write the date as a number of milliseconds from the epoch (January 1, 1970, midnight UTC). The Date class has a complex internal representation that stores both a Calendar object and a millisecond count to optimize lookups. The state of the Calendar is redundant and does not have to be saved.

The readObject and writeObject methods only need to save and load their data fields. They should not concern themselves with superclass data or any other class information.

Instead of letting the serialization mechanism save and restore object data, a class can define its own mechanism. To do this, a class must implement the Externalizable interface. This, in turn, requires it to define two methods:

```
public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutput out)
      throws IOException;
```

Unlike the readObject and writeObject methods that were described in the previous section, these methods are fully responsible for saving and restoring the entire object, *including the superclass data*. When writing an object, the serialization mechanism merely records the class of the object in the output stream. When reading an externalizable object, the object input stream creates an object with the no-argument constructor and then calls the `readExternal` method. Here is how you can implement these methods for the `Employee` class:

```
public void readExternal(ObjectInput s)
      throws IOException
{
   name = s.readUTF();
   salary = s.readDouble();
   hireDay = LocalDate.ofEpochDay(s.readLong());
}

public void writeExternal(ObjectOutput s)
      throws IOException
{
   s.writeUTF(name);
   s.writeDouble(salary);
   s.writeLong(hireDay.toEpochDay());
}
```

CAUTION: Unlike the `readObject` and `writeObject` methods, which are private and can only be called by the serialization mechanism, the `readExternal` and `writeExternal` methods are public. In particular, `readExternal` potentially permits modification of the state of an existing object.

2.3.4 Serializing Singletons and Typesafe Enumerations

You have to pay particular attention to serializing and deserializing objects that are assumed to be unique. This commonly happens when you are implementing singletons and typesafe enumerations.

If you use the `enum` construct of the Java language, you need not worry about serialization—it just works. However, suppose you maintain legacy code that contains an enumerated type such as

```
public class Orientation
{
   public static final Orientation HORIZONTAL = new Orientation(1);
   public static final Orientation VERTICAL  = new Orientation(2);

   private int value;

   private Orientation(int v) { value = v; }
}
```

This idiom was common before enumerations were added to the Java language. Note that the constructor is private. Thus, no objects can be created beyond `Orientation.HORIZONTAL` and `Orientation.VERTICAL`. In particular, you can use the `==` operator to test for object equality:

```
if (orientation == Orientation.HORIZONTAL) . . .
```

There is an important twist that you need to remember when a typesafe enumeration implements the `Serializable` interface. The default serialization mechanism is not appropriate. Suppose we write a value of type `Orientation` and read it in again:

```
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.write(original);
out.close();
ObjectInputStream in = . . .;
var saved = (Orientation) in.read();
```

Now the test

```
if (saved == Orientation.HORIZONTAL) . . .
```

will fail. In fact, the `saved` value is a completely new object of the `Orientation` type that is not equal to any of the predefined constants. Even though the constructor is private, the serialization mechanism can create new objects!

To solve this problem, you need to define another special serialization method, called `readResolve`. If the `readResolve` method is defined, it is called after the object is deserialized. It must return an object which then becomes the return value of the `readObject` method. In our case, the `readResolve` method will inspect the `value` field and return the appropriate enumerated constant:

```
protected Object readResolve() throws ObjectStreamException
{
   if (value == 1) return Orientation.HORIZONTAL;
   if (value == 2) return Orientation.VERTICAL;
   throw new ObjectStreamException(); // this shouldn't happen
}
```

Remember to add a `readResolve` method to all typesafe enumerations in your legacy code and to all classes that follow the singleton design pattern.

2.3.5 Versioning

If you use serialization to save objects, you need to consider what happens when your program evolves. Can version 1.1 read the old files? Can the users who still use 1.0 read the files that the new version is producing? Clearly, it would be desirable if object files could cope with the evolution of classes.

At first glance, it seems that this would not be possible. When a class definition changes in any way, its SHA fingerprint also changes, and you know that object input streams will refuse to read in objects with different fingerprints. However, a class can indicate that it is *compatible* with an earlier version of itself. To do this, you must first obtain the fingerprint of the *earlier* version of the class. Use the standalone `serialver` program that is part of the JDK to obtain this number. For example, running

```
serialver Employee
```

prints

```
Employee: static final long serialVersionUID = -1814239825517340645L;
```

All *later* versions of the class must define the `serialVersionUID` constant to the same fingerprint as the original.

```
class Employee implements Serializable // version 1.1
{
   . . .
   public static final long serialVersionUID = -1814239825517340645L;
}
```

When a class has a static data member named `serialVersionUID`, it will not compute the fingerprint manually but will use that value instead.

Once that static data member has been placed inside a class, the serialization system is now willing to read in different versions of objects of that class.

If only the methods of the class change, there is no problem with reading the new object data. However, if the data fields change, you may have problems. For example, the old file object may have more or fewer data fields than the one in the program, or the types of the data fields may be different. In that case, the object input stream makes an effort to convert the serialized object to the current version of the class.

The object input stream compares the data fields of the current version of the class with those of the version in the serialized object. Of course, the object input stream considers only the nontransient and nonstatic data fields. If two fields have matching names but different types, the object input stream makes no effort to convert one type to the other—the objects are incompatible. If the serialized object has data fields that are not present in the current version, the object input stream ignores the additional data. If the current version has data fields that are not present in the serialized object, the added fields are set to their default (`null` for objects, zero for numbers, and `false` for `boolean` values).

Here is an example. Suppose we have saved a number of employee records on disk, using the original version (1.0) of the class. Now we change the `Employee` class to version 2.0 by adding a data field called `department`. Figure 2.7 shows what happens when a 1.0 object is read into a program that uses 2.0 objects. The `department` field is set to `null`. Figure 2.8 shows the opposite scenario: A program using 1.0 objects reads a 2.0 object. The additional `department` field is ignored.

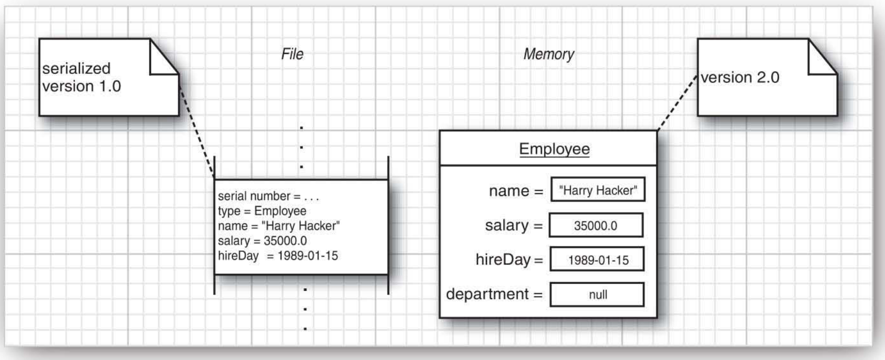

Figure 2.7 Reading an object with fewer data fields

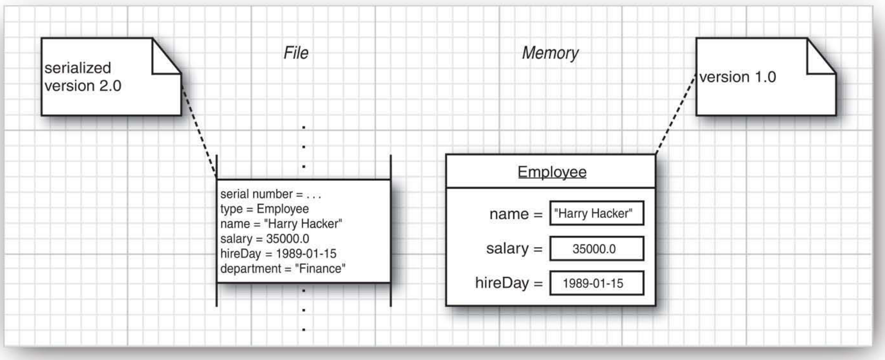

Figure 2.8 Reading an object with more data fields

Is this process safe? It depends. Dropping a data field seems harmless—the recipient still has all the data that it knows how to manipulate. Setting a data field to `null` might not be so safe. Many classes work hard to initialize all data fields in all constructors to non-`null` values, so that the methods don't have to be prepared to handle `null` data. It is up to the class designer to implement additional code in the `readObject` method to fix version incompatibilities or to make sure the methods are robust enough to handle `null` data.

✓ **TIP:** Before you add a `serialVersionUID` field to a class, ask yourself why you made your class serializable. If serialization is used only for short-term persistence, such as distributed method calls in an application server, there is no need to worry about versioning and the `serialVersionUID`. The same applies if you extend a class that happens to be serializable, but you have no intent to ever persist its instances. If your IDE gives you pesky warnings, change the IDE preferences to turn them off, or add an annotation `@SuppressWarnings("serial")`. This is safer than adding a `serialVersionUID` that you may later forget to change.

2.3.6 Using Serialization for Cloning

There is an amusing use for the serialization mechanism: It gives you an easy way to clone an object, provided the class is serializable. Simply serialize it to an output stream and then read it back in. The result is a new object that is a deep copy of the existing object. You don't have to write the object to a file—you can use a `ByteArrayOutputStream` to save the data into a byte array.

As Listing 2.4 shows, to get `clone` for free, simply extend the `SerialCloneable` class, and you are done.

You should be aware that this method, although clever, will usually be much slower than a clone method that explicitly constructs a new object and copies or clones the data fields.

Listing 2.4 `serialClone/SerialCloneTest.java`

```java
package serialClone;

/**
 * @version 1.22 2018-05-01
 * @author Cay Horstmann
 */

import java.io.*;
import java.time.*;

public class SerialCloneTest
{
   public static void main(String[] args) throws CloneNotSupportedException
   {
      var harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
      // clone harry
      var harry2 = (Employee) harry.clone();

      // mutate harry
      harry.raiseSalary(10);

      // now harry and the clone are different
      System.out.println(harry);
      System.out.println(harry2);
   }
}

/**
 * A class whose clone method uses serialization.
 */
class SerialCloneable implements Cloneable, Serializable
{
   public Object clone() throws CloneNotSupportedException
   {
      try {
         // save the object to a byte array
         var bout = new ByteArrayOutputStream();
         try (var out = new ObjectOutputStream(bout))
         {
            out.writeObject(this);
         }

         // read a clone of the object from the byte array
         try (var bin = new ByteArrayInputStream(bout.toByteArray()))
         {
            var in = new ObjectInputStream(bin);
            return in.readObject();
         }
      }
      catch (IOException | ClassNotFoundException e)
      {
         var e2 = new CloneNotSupportedException();
         e2.initCause(e);
         throw e2;
      }
   }
}

/**
 * The familiar Employee class, redefined to extend the
 * SerialCloneable class.
 */
```

(Continues)

Listing 2.4 *(Continued)*

```
class Employee extends SerialCloneable
{
   private String name;
   private double salary;
   private LocalDate hireDay;

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = LocalDate.of(year, month, day);
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public LocalDate getHireDay()
   {
      return hireDay;
   }

   /**
      Raises the salary of this employee.
      @byPercent the percentage of the raise
   */
   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }
}
```

2.4 Working with Files

You have learned how to read and write data from a file. However, there is more to file management than reading and writing. The Path interface and Files class encapsulate the functionality required to work with the file system on the user's machine. For example, the Files class can be used to remove or rename a file, or to find out when a file was last modified. In other words, the input/output stream classes are concerned with the contents of files, whereas the classes that we discuss here are concerned with the storage of files on a disk.

The Path interface and Files class were added in Java 7. They are much more convenient to use than the File class which dates back all the way to JDK 1.0. We expect them to be very popular with Java programmers and discuss them in depth.

2.4.1 Paths

A Path is a sequence of directory names, optionally followed by a file name. The first component of a path may be a *root component* such as / or C:\. The permissible root components depend on the file system. A path that starts with a root component is *absolute*. Otherwise, it is *relative*. For example, here we construct an absolute and a relative path. For the absolute path, we assume a UNIX-like file system.

```
Path absolute = Paths.get("/home", "harry");
Path relative = Paths.get("myprog", "conf", "user.properties");
```

The static Paths.get method receives one or more strings, which it joins with the path separator of the default file system (/ for a UNIX-like file system, \ for Windows). It then parses the result, throwing an InvalidPathException if the result is not a valid path in the given file system. The result is a Path object.

The get method can get a single string containing multiple components. For example, you can read a path from a configuration file like this:

```
String baseDir = props.getProperty("base.dir");
   // May be a string such as /opt/myprog or c:\Program Files\myprog
Path basePath = Paths.get(baseDir); // OK that baseDir has separators
```

NOTE: A path does not have to correspond to a file that actually exists. It is merely an abstract sequence of names. As you will see in the next section, when you want to create a file, you first make a path and then call a method to create the corresponding file.

It is very common to combine or *resolve* paths. The call `p.resolve(q)` returns a path according to these rules:

- If `q` is absolute, then the result is `q`.
- Otherwise, the result is "`p` then `q`," according to the rules of the file system.

For example, suppose your application needs to find its working directory relative to a given base directory that is read from a configuration file, as in the preceding example.

```
Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);
```

There is a shortcut for the `resolve` method that takes a string instead of a path:

```
Path workPath = basePath.resolve("work");
```

There is a convenience method `resolveSibling` that resolves against a path's parent, yielding a sibling path. For example, if `workPath` is `/opt/myapp/work`, the call

```
Path tempPath = workPath.resolveSibling("temp");
```

creates `/opt/myapp/temp`.

The opposite of `resolve` is `relativize`. The call `p.relativize(r)` yields the path `q` which, when resolved with `p`, yields `r`. For example, relativizing `/home/harry` against `/home/fred/input.txt` yields `../fred/input.txt`. Here, we assume that `..` denotes the parent directory in the file system.

The `normalize` method removes any redundant `.` and `..` components (or whatever the file system may deem redundant). For example, normalizing the path `/home/harry/../fred/./input.txt` yields `/home/fred/input.txt`.

The `toAbsolutePath` method yields the absolute path of a given path, starting at a root component, such as `/home/fred/input.txt` or `c:\Users\fred\input.txt`.

The `Path` interface has many useful methods for taking paths apart. This code sample shows some of the most useful ones:

```
Path p = Paths.get("/home", "fred", "myprog.properties");
Path parent = p.getParent(); // the path /home/fred
Path file = p.getFileName(); // the path myprog.properties
Path root = p.getRoot(); // the path /
```

As you have already seen in Volume I, you can construct a `Scanner` from a `Path` object:

```
var in = new Scanner(Paths.get("/home/fred/input.txt"));
```

NOTE: Occasionally, you may need to interoperate with legacy APIs that use the `File` class instead of the `Path` interface. The `Path` interface has a `toFile` method, and the `File` class has a `toPath` method.

`java.nio.file.Paths` 7

- `static Path get(String first, String... more)`

  makes a path by joining the given strings.

`java.nio.file.Path` 7

- `Path resolve(Path other)`
- `Path resolve(String other)`

  if `other` is absolute, returns `other`; otherwise, returns the path obtained by joining `this` and `other`.

- `Path resolveSibling(Path other)`
- `Path resolveSibling(String other)`

  if `other` is absolute, returns `other`; otherwise, returns the path obtained by joining the parent of `this` and `other`.

- `Path relativize(Path other)`

  returns the relative path that, when resolved with `this`, yields `other`.

- `Path normalize()`

  removes redundant path elements such as `.` and `..`

- `Path toAbsolutePath()`

  returns an absolute path that is equivalent to this path.

- `Path getParent()`

  returns the parent, or `null` if this path has no parent.

- `Path getFileName()`

  returns the last component of this path, or `null` if this path has no components.

- `Path getRoot()`

  returns the root component of this path, or `null` if this path has no root components.

- `toFile()`

  makes a `File` from this path.

java.io.File 1.0

- `Path toPath()` 7

  makes a `Path` from this file.

2.4.2 Reading and Writing Files

The `Files` class makes quick work of common file operations. For example, you can easily read the entire contents of a file:

```
byte[] bytes = Files.readAllBytes(path);
```

As already mentioned in Section 2.1.6, "How to Read Text Input," on p. 70, you can read the content of a text file as

```
var content = Files.readString(path, charset);
```

But if you want the file as a sequence of lines, call

```
List<String> lines = Files.readAllLines(path, charset);
```

Conversely, if you want to write a string, call

```
Files.write(path, content.getBytes(charset));
```

To append to a given file, use

```
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
```

You can also write a collection of lines with

```
Files.write(path, lines, charset);
```

These simple methods are intended for dealing with text files of moderate length. If your files are large or binary, you can still use the familiar input/output streams or readers/writers:

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

These convenience methods save you from dealing with `FileInputStream`, `FileOutputStream`, `BufferedReader`, or `BufferedWriter`.

java.nio.file.Files 7

- `static byte[] readAllBytes(Path path)`
- `static String readString(Path path, Charset charset)`
- `static List<String> readAllLines(Path path, Charset charset)`

  reads the contents of a file.
- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `static Path write(Path path, String contents, Charset charset, OpenOption... options)`
- `static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)`

  writes the given contents to a file and returns `path`.
- `static InputStream newInputStream(Path path, OpenOption... options)`
- `static OutputStream newOutputStream(Path path, OpenOption... options)`
- `static BufferedReader newBufferedReader(Path path, Charset charset)`
- `static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)`

  opens a file for reading or writing.

2.4.3 Creating Files and Directories

To create a new directory, call

```
Files.createDirectory(path);
```

All but the last component in the path must already exist. To create intermediate directories as well, use

```
Files.createDirectories(path);
```

You can create an empty file with

```
Files.createFile(path);
```

The call throws an exception if the file already exists. The check for existence and creation are atomic. If the file doesn't exist, it is created before anyone else has a chance to do the same.

There are convenience methods for creating a temporary file or directory in a given or system-specific location.

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

Here, `dir` is a `Path`, and `prefix`/`suffix` are strings which may be `null`. For example, the call `Files.createTempFile(null, ".txt")` might return a path such as `/tmp/1234405522364837194.txt`.

When you create a file or directory, you can specify attributes, such as owners or permissions. However, the details depend on the file system, and we won't cover them here.

`java.nio.file.Files` 7

- `static Path createFile(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectory(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectories(Path path, FileAttribute<?>... attrs)`

  creates a file or directory. The `createDirectories` method creates any intermediate directories as well.

- `static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`
- `static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)`
- `static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)`
- `static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)`

  creates a temporary file or directory, in a location suitable for temporary files or in the given parent directory. Returns the path to the created file or directory.

2.4.4 Copying, Moving, and Deleting Files

To copy a file from one location to another, simply call

```
Files.copy(fromPath, toPath);
```

To move the file (that is, copy and delete the original), call

```
Files.move(fromPath, toPath);
```

The copy or move will fail if the target exists. If you want to overwrite an existing target, use the `REPLACE_EXISTING` option. If you want to copy all file attributes, use the `COPY_ATTRIBUTES` option. You can supply both like this:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
   StandardCopyOption.COPY_ATTRIBUTES);
```

You can specify that a move should be atomic. Then you are assured that either the move completed successfully, or the source continues to be present. Use the `ATOMIC_MOVE` option:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

You can also copy an input stream to a `Path`, which just means saving the input stream to disk. Similarly, you can copy a `Path` to an output stream. Use the following calls:

```
Files.copy(inputStream, toPath);
Files.copy(fromPath, outputStream);
```

As with the other calls to `copy`, you can supply copy options as needed.

Finally, to delete a file, simply call

```
Files.delete(path);
```

This method throws an exception if the file doesn't exist, so instead you may want to use

```
boolean deleted = Files.deleteIfExists(path);
```

The deletion methods can also be used to remove an empty directory.

See Table 2.3 for a summary of the options that are available for file operations.

`java.nio.file.Files` 7

- `static Path copy(Path from, Path to, CopyOption... options)`
- `static Path move(Path from, Path to, CopyOption... options)`

  copies or moves `from` to the given target location and returns `to`.

- `static long copy(InputStream from, Path to, CopyOption... options)`
- `static long copy(Path from, OutputStream to, CopyOption... options)`

  copies from an input stream to a file, or from a file to an output stream, returning the number of bytes copied.

- `static void delete(Path path)`
- `static boolean deleteIfExists(Path path)`

  deletes the given file or empty directory. The first method throws an exception if the file or directory doesn't exist. The second method returns `false` in that case.

Table 2.3 Standard Options for File Operations

Option	Description
`StandardOpenOption`; use with `newBufferedWriter`, `newInputStream`, `newOutputStream`, `write`	
`READ`	Open for reading
`WRITE`	Open for writing
`APPEND`	If opened for writing, append to the end of the file
`TRUNCATE_EXISTING`	If opened for writing, remove existing contents
`CREATE_NEW`	Create a new file and fail if it exists
`CREATE`	Atomically create a new file if it doesn't exist
`DELETE_ON_CLOSE`	Make a "best effort" to delete the file when it is closed
`SPARSE`	A hint to the file system that this file will be sparse
`DSYNC` or `SYNC`	Requires that each update to the file data or data and metadata be written synchronously to the storage device
`StandardCopyOption`; use with `copy`, `move`	
`ATOMIC_MOVE`	Move the file atomically
`COPY_ATTRIBUTES`	Copy the file attributes
`REPLACE_EXISTING`	Replace the target if it exists
`LinkOption`; use with all of the above methods and `exists`, `isDirectory`, `isRegularFile`	
`NOFOLLOW_LINKS`	Do not follow symbolic links
`FileVisitOption`; use with `find`, `walk`, `walkFileTree`	
`FOLLOW_LINKS`	Follow symbolic links

2.4.5 Getting File Information

The following static methods return a `boolean` value to check a property of a path:

- `exists`
- `isHidden`
- `isReadable`, `isWritable`, `isExecutable`
- `isRegularFile`, `isDirectory`, `isSymbolicLink`

The `size` method returns the number of bytes in a file.

```
long fileSize = Files.size(path);
```

The `getOwner` method returns the owner of the file, as an instance of `java.nio.file.attribute.UserPrincipal`.

All file systems report a set of basic attributes, encapsulated by the `BasicFileAttributes` interface, which partially overlaps with that information. The basic file attributes are

- The times at which the file was created, last accessed, and last modified, as instances of the class `java.nio.file.attribute.FileTime`
- Whether the file is a regular file, a directory, a symbolic link, or none of these
- The file size
- The file key—an object of some class, specific to the file system, that may or may not uniquely identify a file

To get these attributes, call

```
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);
```

If you know that the user's file system is POSIX-compliant, you can instead get an instance of `PosixFileAttributes`:

```
PosixFileAttributes attributes = Files.readAttributes(path, PosixFileAttributes.class);
```

Then you can find out the group owner and the owner, group, and world access permissions of the file. We won't dwell on the details since so much of this information is not portable across operating systems.

`java.nio.file.Files` 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`

  checks for the given property of the file given by the path.
- `static long size(Path path)`

  gets the size of the file in bytes.
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`

  reads the file attributes of type `A`.

***java.nio.file.attribute.BasicFileAttributes* 7**

- `FileTime creationTime()`
- `FileTime lastAccessTime()`
- `FileTime lastModifiedTime()`
- `boolean isRegularFile()`
- `boolean isDirectory()`
- `boolean isSymbolicLink()`
- `long size()`
- `Object fileKey()`

  gets the requested attribute.

2.4.6 Visiting Directory Entries

The static `Files.list` method returns a `Stream<Path>` that reads the entries of a directory. The directory is read lazily, making it possible to efficiently process directories with huge numbers of entries.

Since reading a directory involves a system resource that needs to be closed, you should use a `try` block:

```
try (Stream<Path> entries = Files.list(pathToDirectory))
{
   . . .
}
```

The `list` method does not enter subdirectories. To process all descendants of a directory, use the `Files.walk` method instead.

```
try (Stream<Path> entries = Files.walk(pathToRoot))
{
   // Contains all descendants, visited in depth-first order
}
```

Here is a sample traversal of the unzipped `src.zip` tree:

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
. . .
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
```

```
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
. . .
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
. . .
```

As you can see, whenever the traversal yields a directory, it is entered before continuing with its siblings.

You can limit the depth of the tree that you want to visit by calling `Files.walk(pathToRoot, depth)`. Both `walk` methods have a varargs parameter of type `FileVisitOption...`, but there is only one option you can supply: `FOLLOW_LINKS` to follow symbolic links.

NOTE: If you filter the paths returned by `walk` and your filter criterion involves the file attributes stored with a directory, such as size, creation time, or type (file, directory, symbolic link), then use the `find` method instead of `walk`. Call that method with a predicate function that accepts a path and a `BasicFileAttributes` object. The only advantage is efficiency. Since the directory is being read anyway, the attributes are readily available.

This code fragment uses the `Files.walk` method to copy one directory to another:

```
Files.walk(source).forEach(p ->
   {
      try
      {
         Path q = target.resolve(source.relativize(p));
         if (Files.isDirectory(p))
            Files.createDirectory(q);
         else
            Files.copy(p, q);
      }
      catch (IOException ex)
      {
         throw new UncheckedIOException(ex);
      }
   });
```

Unfortunately, you cannot easily use the `Files.walk` method to delete a tree of directories since you need to delete the children before deleting the parent. The next section shows you how to overcome that problem.

2.4.7 Using Directory Streams

As you saw in the preceding section, the `Files.walk` method produces a `Stream<Path>` that traverses the descendants of a directory. Sometimes, you need more fine-grained control over the traversal process. In that case, use the `Files.newDirectoryStream` object instead. It yields a `DirectoryStream`. Note that this is not a subinterface of `java.util.stream.Stream` but an interface that is specialized for directory traversal. It is a subinterface of `Iterable` so that you can use directory stream in an enhanced `for` loop. Here is the usage pattern:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
{
   for (Path entry : entries)
      Process entries
}
```

The `try`-with-resources block ensures that the directory stream is properly closed.

There is no specific order in which the directory entries are visited.

You can filter the files with a glob pattern:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

Table 2.4 shows all glob patterns.

Table 2.4 Glob Patterns

Pattern	Description	Example
`*`	Matches zero or more characters of a path component.	`*.java` matches all Java files in the current directory.
`**`	Matches zero or more characters, crossing directory boundaries.	`**.java` matches all Java files in any subdirectory.
`?`	Matches one character.	`????.java` matches all four-character (not counting the extension) Java files.
`[. . .]`	Matches a set of characters. You can use hyphens `[0-9]` and negation `[!0-9]`.	`Test[0-9A-F].java` matches `Test`*x*`.java`, where *x* is one hexadecimal digit.
`{. . .}`	Matches alternatives, separated by commas.	`*.{java,class}` matches all Java and class files.
`\`	Escapes any of the above as well as `\`.	`*\**` matches all files with a `*` in their name.

CAUTION: If you use the glob syntax on Windows, you have to escape backslashes *twice*: once for the glob syntax, and once for the Java string syntax: `Files.newDirectoryStream(dir, "C:\\\\")`.

If you want to visit all descendants of a directory, call the `walkFileTree` method instead and supply an object of type `FileVisitor`. That object gets notified

- When a file is encountered: `FileVisitResult visitFile(T path, BasicFileAttributes attrs)`
- Before a directory is processed: `FileVisitResult preVisitDirectory(T dir, IOException ex)`
- After a directory is processed: `FileVisitResult postVisitDirectory(T dir, IOException ex)`
- When an error occurred trying to visit a file or directory, such as trying to open a directory without the necessary permissions: `FileVisitResult visitFileFailed(T path, IOException ex)`

In each case, you can specify whether you want to

- Continue visiting the next file: `FileVisitResult.CONTINUE`
- Continue the walk, but without visiting the entries in this directory: `FileVisitResult.SKIP_SUBTREE`
- Continue the walk, but without visiting the siblings of this file: `FileVisitResult.SKIP_SIBLINGS`
- Terminate the walk: `FileVisitResult.TERMINATE`

If any of the methods throws an exception, the walk is also terminated, and that exception is thrown from the `walkFileTree` method.

NOTE: The `FileVisitor` interface is a generic type, but it isn't likely that you'll ever want something other than a `FileVisitor<Path>`. The `walkFileTree` method is willing to accept a `FileVisitor<? super Path>`, but `Path` does not have an abundance of supertypes.

A convenience class `SimpleFileVisitor` implements the `FileVisitor` interface. All methods except `visitFileFailed` do nothing and continue. The `visitFileFailed` method throws the exception that caused the failure, thereby terminating the visit.

For example, here is how to print out all subdirectories of a given directory:

```
Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path>()
   {
      public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes attrs)
            throws IOException
      {
         System.out.println(path);
         return FileVisitResult.CONTINUE;
      }

      public FileVisitResult postVisitDirectory(Path dir, IOException exc)
      {
         return FileVisitResult.CONTINUE;
      }

      public FileVisitResult visitFileFailed(Path path, IOException exc)
            throws IOException
      {
         return FileVisitResult.SKIP_SUBTREE;
      }
   });
```

Note that we need to override `postVisitDirectory` and `visitFileFailed`. Otherwise, the visit would fail as soon as it encounters a directory that it's not allowed to open or a file it's not allowed to access.

Also note that the attributes of the path are passed as a parameter to the `preVisitDirectory` and `visitFile` methods. The visitor already had to make an OS call to get the attributes, since it needs to distinguish between files and directories. This way, you don't need to make another call.

The other methods of the `FileVisitor` interface are useful if you need to do some work when entering or leaving a directory. For example, when you delete a directory tree, you need to remove the current directory after you have removed all of its files. Here is the complete code for deleting a directory tree:

```
// Delete the directory tree starting at root
Files.walkFileTree(root, new SimpleFileVisitor<Path>()
   {
      public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
            throws IOException
      {
         Files.delete(file);
         return FileVisitResult.CONTINUE;
      }
      public FileVisitResult postVisitDirectory(Path dir, IOException e) throws IOException
      {
         if (e != null) throw e;
         Files.delete(dir);
         return FileVisitResult.CONTINUE;
      }
   });
```

`java.nio.file.Files` 7

- `static DirectoryStream<Path> newDirectoryStream(Path path)`
- `static DirectoryStream<Path> newDirectoryStream(Path path, String glob)`

  gets an iterator over the files and directories in a given directory. The second method only accepts those entries matching the given glob pattern.
- `static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`

  walks all descendants of the given path, applying the visitor to all descendants.

`java.nio.file.SimpleFileVisitor<T>` 7

- `static FileVisitResult visitFile(T path, BasicFileAttributes attrs)`

  is called when a file or directory is visited; returns one of `CONTINUE`, `SKIP_SUBTREE`, `SKIP_SIBLINGS`, or `TERMINATE`. The default implementation does nothing and continues.
- `static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`
- `static FileVisitResult postVisitDirectory(T dir, IOException exc)`

  are called before and after visiting a directory. The default implementation does nothing and continues.
- `static FileVisitResult visitFileFailed(T path, IOException exc)`

  is called if an exception was thrown in an attempt to get information about the given file. The default implementation rethrows the exception, which causes the visit to terminate with that exception. Override the method if you want to continue.

2.4.8 ZIP File Systems

The `Paths` class looks up paths in the default file system—the files on the user's local disk. You can have other file systems. One of the more useful ones is a *ZIP file system*. If `zipname` is the name of a ZIP file, then the call

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
```

establishes a file system that contains all files in the ZIP archive. It's an easy matter to copy a file out of that archive if you know its name:

```
Files.copy(fs.getPath(sourceName), targetPath);
```

Here, `fs.getPath` is the analog of `Paths.get` for an arbitrary file system.

To list all files in a ZIP archive, walk the file tree:

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Path>()
   {
      public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
            throws IOException
      {
         System.out.println(file);
         return FileVisitResult.CONTINUE;
      }
   });
```

That is nicer than the API described in Section 2.2.3, "ZIP Archives," on p. 85 which required a set of new classes just to deal with ZIP archives.

`java.nio.file.FileSystems` 7

- `static FileSystem newFileSystem(Path path, ClassLoader loader)`

  iterates over the installed file system providers and, provided that `loader` is not `null`, the file systems that the given class loader can load. Returns the file system created by the first file system provider that accepts the given path. By default, there is a provider for ZIP file systems that accepts files whose names end in `.zip` or `.jar`.

`java.nio.file.FileSystem` 7

- `static Path getPath(String first, String... more)`

  makes a path by joining the given strings.

2.5 Memory-Mapped Files

Most operating systems can take advantage of a virtual memory implementation to "map" a file, or a region of a file, into memory. Then the file can be accessed as if it were an in-memory array, which is much faster than the traditional file operations.

2.5.1 Memory-Mapped File Performance

At the end of this section, you can find a program that computes the CRC32 checksum of a file using traditional file input and a memory-mapped file. On one machine, we got the timing data shown in Table 2.5 when computing the checksum of the 37MB file `rt.jar` in the `jre/lib` directory of the JDK.

Table 2.5 Timing Data for File Operations

Method	Time
Plain input stream	110 seconds
Buffered input stream	9.9 seconds
Random access file	162 seconds
Memory-mapped file	7.2 seconds

As you can see, on this particular machine, memory mapping is a bit faster than using buffered sequential input and dramatically faster than using a `RandomAccessFile`.

Of course, the exact values will differ greatly from one machine to another, but it is obvious that the performance gain, compared to random access, can be substantial. For sequential reading of files of moderate size, on the other hand, there is no reason to use memory mapping.

The `java.nio` package makes memory mapping quite simple. Here is what you do.

First, get a *channel* for the file. A channel is an abstraction for a disk file that lets you access operating system features such as memory mapping, file locking, and fast data transfers between files.

```
FileChannel channel = FileChannel.open(path, options);
```

Then, get a `ByteBuffer` from the channel by calling the `map` method of the `FileChannel` class. Specify the area of the file that you want to map and a *mapping mode*. Three modes are supported:

- `FileChannel.MapMode.READ_ONLY`: The resulting buffer is read-only. Any attempt to write to the buffer results in a `ReadOnlyBufferException`.
- `FileChannel.MapMode.READ_WRITE`: The resulting buffer is writable, and the changes will be written back to the file at some time. Note that other programs that have mapped the same file might not see those changes immediately. The exact behavior of simultaneous file mapping by multiple programs depends on the operating system.

- `FileChannel.MapMode.PRIVATE`: The resulting buffer is writable, but any changes are private to this buffer and not propagated to the file.

Once you have the buffer, you can read and write data using the methods of the `ByteBuffer` class and the `Buffer` superclass.

Buffers support both sequential and random data access. A buffer has a *position* that is advanced by `get` and `put` operations. For example, you can sequentially traverse all bytes in the buffer as

```
while (buffer.hasRemaining())
{
   byte b = buffer.get();
   . . .
}
```

Alternatively, you can use random access:

```
for (int i = 0; i < buffer.limit(); i++)
{
   byte b = buffer.get(i);
   . . .
}
```

You can also read and write arrays of bytes with the methods

```
get(byte[] bytes)
get(byte[] bytes, int offset, int length)
```

Finally, there are methods

```
getInt          getChar
getLong         getFloat
getShort        getDouble
```

to read primitive-type values that are stored as *binary* values in the file. As we already mentioned, Java uses big-endian ordering for binary data. However, if you need to process a file containing binary numbers in little-endian order, simply call

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

To find out the current byte order of a buffer, call

```
ByteOrder b = buffer.order();
```

CAUTION: This pair of methods does not use the `set`/`get` naming convention.

To write numbers to a buffer, use one of the methods

```
putInt          putChar
putLong         putFloat
putShort        putDouble
```

At some point, and certainly when the channel is closed, these changes are written back to the file.

Listing 2.5 computes the 32-bit cyclic redundancy checksum (CRC32) of a file. That checksum is often used to determine whether a file has been corrupted. Corruption of a file makes it very likely that the checksum has changed. The `java.util.zip` package contains a class `CRC32` that computes the checksum of a sequence of bytes, using the following loop:

```
var crc = new CRC32();
while (more bytes)
   crc.update(next byte);
long checksum = crc.getValue();
```

The details of the CRC computation are not important. We just use it as an example of a useful file operation. (In practice, you would read and update data in larger blocks, not a byte at a time. Then the speed differences are not as dramatic.)

Run the program as

```
java memoryMap.MemoryMapTest filename
```

Listing 2.5 `memoryMap/MemoryMapTest.java`

```
package memoryMap;

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.util.zip.*;

/**
 * This program computes the CRC checksum of a file in four ways. <br>
 * Usage: java memoryMap.MemoryMapTest filename
 * @version 1.02 2018-05-01
 * @author Cay Horstmann
 */
```

(Continues)

Listing 2.5 *(Continued)*

```java
public class MemoryMapTest
{
   public static long checksumInputStream(Path filename) throws IOException
   {
      try (InputStream in = Files.newInputStream(filename))
      {
         var crc = new CRC32();

         int c;
         while ((c = in.read()) != -1)
            crc.update(c);
         return crc.getValue();
      }
   }

   public static long checksumBufferedInputStream(Path filename) throws IOException
   {
      try (var in = new BufferedInputStream(Files.newInputStream(filename)))
      {
         var crc = new CRC32();

         int c;
         while ((c = in.read()) != -1)
            crc.update(c);
         return crc.getValue();
      }
   }

   public static long checksumRandomAccessFile(Path filename) throws IOException
   {
      try (var file = new RandomAccessFile(filename.toFile(), "r"))
      {
         long length = file.length();
         var crc = new CRC32();

         for (long p = 0; p < length; p++)
         {
            file.seek(p);
            int c = file.readByte();
            crc.update(c);
         }
         return crc.getValue();
      }
   }

   public static long checksumMappedFile(Path filename) throws IOException
   {
      try (FileChannel channel = FileChannel.open(filename))
      {
         var crc = new CRC32();
         int length = (int) channel.size();
         MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);

         for (int p = 0; p < length; p++)
         {
            int c = buffer.get(p);
            crc.update(c);
         }
         return crc.getValue();
      }
   }

   public static void main(String[] args) throws IOException
   {
      System.out.println("Input Stream:");
      long start = System.currentTimeMillis();
      Path filename = Paths.get(args[0]);
      long crcValue = checksumInputStream(filename);
      long end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");

      System.out.println("Buffered Input Stream:");
      start = System.currentTimeMillis();
      crcValue = checksumBufferedInputStream(filename);
      end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");

      System.out.println("Random Access File:");
      start = System.currentTimeMillis();
      crcValue = checksumRandomAccessFile(filename);
      end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");

      System.out.println("Mapped File:");
      start = System.currentTimeMillis();
      crcValue = checksumMappedFile(filename);
      end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");
   }
}
```

`java.io.FileInputStream` 1.0

- `FileChannel getChannel()` 1.4

  returns a channel for accessing this input stream.

`java.io.FileOutputStream` 1.0

- `FileChannel getChannel()` 1.4

  returns a channel for accessing this output stream.

`java.io.RandomAccessFile` 1.0

- `FileChannel getChannel()` 1.4

  returns a channel for accessing this file.

`java.nio.channels.FileChannel` 1.4

- `static FileChannel open(Path path, OpenOption... options)` 7

  opens a file channel for the given path. By default, the channel is opened for reading. The parameter `options` is one of the values `WRITE`, `APPEND`,`TRUNCATE_EXISTING`, `CREATE` in the `StandardOpenOption` enumeration.
- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`

  maps a region of the file to memory. The parameter `mode` is one of the constants `READ_ONLY`, `READ_WRITE`, or `PRIVATE` in the `FileChannel.MapMode` class.

`java.nio.Buffer` 1.4

- `boolean hasRemaining()`

  returns `true` if the current buffer position has not yet reached the buffer's limit position.
- `int limit()`

  returns the limit position of the buffer—that is, the first position at which no more values are available.

`java.nio.ByteBuffer` 1.4

- `byte get()`

  gets a byte from the current position and advances the current position to the next byte.

- `byte get(int index)`

  gets a byte from the specified index.

- `ByteBuffer put(byte b)`

  puts a byte at the current position and advances the current position to the next byte. Returns a reference to this buffer.

- `ByteBuffer put(int index, byte b)`

  puts a byte at the specified index. Returns a reference to this buffer.

- `ByteBuffer get(byte[] destination)`
- `ByteBuffer get(byte[] destination, int offset, int length)`

  fills a byte array, or a region of a byte array, with bytes from the buffer, and advances the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are read, and a `BufferUnderflowException` is thrown. Returns a reference to this buffer.

- `ByteBuffer put(byte[] source)`
- `ByteBuffer put(byte[] source, int offset, int length)`

  puts all bytes from a byte array, or the bytes from a region of a byte array, into the buffer, and advances the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are written, and a `BufferOverflowException` is thrown. Returns a reference to this buffer.

- *Xxx* `get`*Xxx*`()`
- *Xxx* `get`*Xxx*`(int index)`
- `ByteBuffer put`*Xxx*`(`*Xxx* `value)`
- `ByteBuffer put`*Xxx*`(int index,` *Xxx* `value)`

  gets or puts a binary number. *Xxx* is one of `Int`, `Long`, `Short`, `Char`, `Float`, or `Double`.

- `ByteBuffer order(ByteOrder order)`
- `ByteOrder order()`

  sets or gets the byte order. The value for `order` is one of the constants `BIG_ENDIAN` or `LITTLE_ENDIAN` of the `ByteOrder` class.

- `static ByteBuffer allocate(int capacity)`

  constructs a buffer with the given capacity.

(Continues)

java.nio.ByteBuffer 1.4 *(Continued)*

- `static ByteBuffer wrap(byte[] values)`

  constructs a buffer that is backed by the given array.
- `CharBuffer asCharBuffer()`

  constructs a character buffer that is backed by this buffer. Changes to the character buffer will show up in this buffer, but the character buffer has its own position, limit, and mark.

java.nio.CharBuffer 1.4

- `char get()`
- `CharBuffer get(char[] destination)`
- `CharBuffer get(char[] destination, int offset, int length)`

  gets one `char` value, or a range of `char` values, starting at the buffer's position and moving the position past the characters that were read. The last two methods return `this`.
- `CharBuffer put(char c)`
- `CharBuffer put(char[] source)`
- `CharBuffer put(char[] source, int offset, int length)`
- `CharBuffer put(String source)`
- `CharBuffer put(CharBuffer source)`

  puts one `char` value, or a range of `char` values, starting at the buffer's position and advancing the position past the characters that were written. When reading from a `CharBuffer`, all remaining characters are read. All methods return `this`.

2.5.2 The Buffer Data Structure

When you use memory mapping, you make a single buffer that spans the entire file or the area of the file that you're interested in. You can also use buffers to read and write more modest chunks of information.

In this section, we briefly describe the basic operations on `Buffer` objects. A buffer is an array of values of the same type. The `Buffer` class is an abstract class with concrete subclasses `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, and `ShortBuffer`.

NOTE: The `StringBuffer` class is not related to these buffers.

In practice, you will most commonly use `ByteBuffer` and `CharBuffer`. As shown in Figure 2.9, a buffer has

- A *capacity* that never changes
- A *position* at which the next value is read or written
- A *limit* beyond which reading and writing is meaningless
- Optionally, a *mark* for repeating a read or write operation

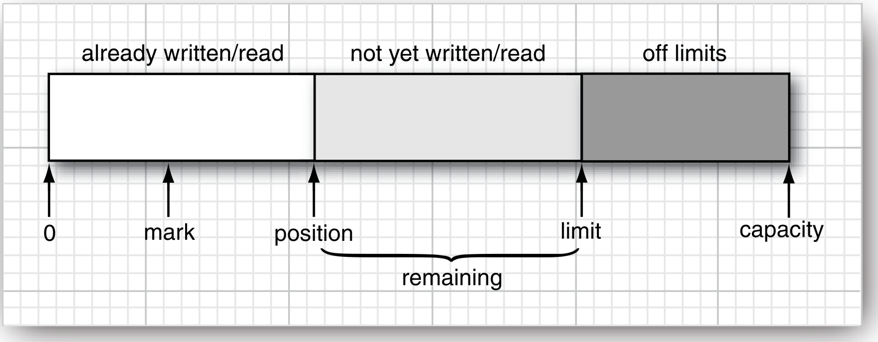

Figure 2.9 A buffer

These values fulfill the condition

$0 \le mark \le position \le limit \le capacity$

The principal purpose of a buffer is a "write, then read" cycle. At the outset, the buffer's position is 0 and the limit is the capacity. Keep calling `put` to add values to the buffer. When you run out of data or reach the capacity, it is time to switch to reading.

Call `flip` to set the limit to the current position and the position to `0`. Now keep calling `get` while the `remaining` method (which returns *limit* – *position*) is positive. When you have read all values in the buffer, call `clear` to prepare the buffer for the next writing cycle. The `clear` method resets the position to `0` and the limit to the capacity.

If you want to reread the buffer, use `rewind` or `mark/reset` (see the API notes for details).

To get a buffer, call a static method such as `ByteBuffer.allocate` or `ByteBuffer.wrap`.

Then, you can fill a buffer from a channel, or write its contents to a channel. For example,

```
ByteBuffer buffer = ByteBuffer.allocate(RECORD_SIZE);
channel.read(buffer);
channel.position(newpos);
buffer.flip();
channel.write(buffer);
```

This can be a useful alternative to a random-access file.

`java.nio.Buffer` 1.4

- `Buffer clear()`

  prepares this buffer for writing by setting the position to `0` and the limit to the capacity; returns `this`.
- `Buffer flip()`

  prepares this buffer for reading after writing, by setting the limit to the position and the position to `0`; returns `this`.
- `Buffer rewind()`

  prepares this buffer for rereading the same values by setting the position to `0` and leaving the limit unchanged; returns `this`.
- `Buffer mark()`

  sets the mark of this buffer to the position; returns `this`.
- `Buffer reset()`

  sets the position of this buffer to the mark, thus allowing the marked portion to be read or written again; returns `this`.
- `int remaining()`

  returns the remaining number of readable or writable values—that is, the difference between the limit and position.
- `int position()`
- `void position(int newValue)`

  gets and sets the position of this buffer.
- `int capacity()`

  returns the capacity of this buffer.

2.6 File Locking

When multiple simultaneously executing programs need to modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem. A file lock controls access to a file or a range of bytes within a file.

Suppose your application saves a configuration file with user preferences. If a user invokes two instances of the application, it could happen that both of them want to write the configuration file at the same time. In that situation, the first instance should lock the file. When the second instance finds the file locked, it can decide to wait until the file is unlocked or simply skip the writing process.

To lock a file, call either the `lock` or `tryLock` methods of the `FileChannel` class.

```
FileChannel = FileChannel.open(path);
FileLock lock = channel.lock();
```

or

```
FileLock lock = channel.tryLock();
```

The first call blocks until the lock becomes available. The second call returns immediately, either with the lock or with `null` if the lock is not available. The file remains locked until the channel is closed or the `release` method is invoked on the lock.

You can also lock a portion of the file with the call

```
FileLock lock(long start, long size, boolean shared)
```

or

```
FileLock tryLock(long start, long size, boolean shared)
```

The `shared` flag is `false` to lock the file for both reading and writing. It is `true` for a *shared* lock, which allows multiple processes to read from the file, while preventing any process from acquiring an exclusive lock. Not all operating systems support shared locks. You may get an exclusive lock even if you just asked for a shared one. Call the `isShared` method of the `FileLock` class to find out which kind you have.

NOTE: If you lock the tail portion of a file and the file subsequently grows beyond the locked portion, the additional area is not locked. To lock all bytes, use a size of `Long.MAX_VALUE`.

Be sure to unlock the lock when you are done. As always, this is best done with a `try`-with-resources statement:

```
try (FileLock lock = channel.lock())
{
   access the locked file or segment
}
```

Keep in mind that file locking is system-dependent. Here are some points to watch for:

- On some systems, file locking is merely *advisory*. If an application fails to get a lock, it may still write to a file that another application has currently locked.
- On some systems, you cannot simultaneously lock a file and map it into memory.
- File locks are held by the entire Java virtual machine. If two programs are launched by the same virtual machine (such as an applet or application launcher), they can't each acquire a lock on the same file. The `lock` and `tryLock` methods will throw an `OverlappingFileLockException` if the virtual machine already holds another overlapping lock on the same file.
- On some systems, closing a channel releases all locks on the underlying file held by the Java virtual machine. You should therefore avoid multiple channels on the same locked file.
- Locking files on a networked file system is highly system-dependent and should probably be avoided.

`java.nio.channels.FileChannel` 1.4

- `FileLock lock()`

  acquires an exclusive lock on the entire file. This method blocks until the lock is acquired.
- `FileLock tryLock()`

  acquires an exclusive lock on the entire file, or returns `null` if the lock cannot be acquired.
- `FileLock lock(long position, long size, boolean shared)`
- `FileLock tryLock(long position, long size, boolean shared)`

  acquires a lock on a region of the file. The first method blocks until the lock is acquired, and the second method returns `null` if the lock cannot be acquired. The parameter `shared` is `true` for a shared lock, `false` for an exclusive lock.

`java.nio.channels.FileLock` 1.4

- `void close()` 1.7

  releases this lock.

2.7 Regular Expressions

Regular expressions are used to specify string patterns. You can use regular expressions whenever you need to locate strings that match a particular pattern. For example, one of our sample programs locates all hyperlinks in an HTML file by looking for strings of the pattern `<a href=". . .">`.

Of course, when specifying a pattern, the `. . .` notation is not precise enough. You need to specify exactly what sequence of characters is a legal match, using a special syntax to describe a pattern.

In the following sections, we cover the regular expression syntax used by the Java API and discuss how to put regular expressions to work.

2.7.1 The Regular Expression Syntax

Let us start with a simple example. The regular expression

```
[Jj]ava.+
```

matches any string of the following form:

- The first letter is a `J` or `j`.
- The next three letters are `ava`.
- The remainder of the string consists of one or more arbitrary characters.

For example, the string `"javanese"` matches this particular regular expression, but the string `"Core Java"` does not.

As you can see, you need to know a bit of syntax to understand the meaning of a regular expression. Fortunately, for most purposes, a few straightforward constructs are sufficient.

- A *character class* is a set of character alternatives, enclosed in brackets, such as `[Jj]`, `[0-9]`, `[A-Za-z]`, or `[^0-9]`. Here the `-` denotes a range (all characters whose Unicode values fall between the two bounds), and `^` denotes the complement (all characters except those specified).
- To include a `-` inside a character class, make it the first or last item. To include a `]`, make it the first item. To include a `^`, put it anywhere but the beginning. You only need to escape `[` and `\`.
- There are many predefined character classes such as `\d` (digits) or `\p{Sc}` (Unicode currency symbol). See Tables 2.6 and 2.7.

Table 2.6 Regular Expression Syntax

Expression	Description	Example
Characters		
c, not one of `. * + ? { | ( ) [ \ ^ $`	The character *c*	`J`
`.`	Any character except line terminators, or any character if the `DOTALL` flag is set	
`\x{`*p*`}`	The Unicode code point with hex code *p*	`\x{1D546}`
`\u`*hhhh*, `\x`*hh*, `\0`*o*, `\0`*oo*, `\0`*ooo*	The UTF-16 code unit with the given hex or octal value	`\uFEFF`
`\a`, `\e`, `\f`, `\n`, `\r`, `\t`	Alert (`\x{7}`), escape (`\x{1B}`), form feed (`\x{B}`), newline (`\x{A}`), carriage return (`\x{D}`), tab (`\x{9}`)	`\n`
`\c`*c*, where *c* is in `[A-Z]` or one of `@ [ \ ] ^ _ ?`	The control character corresponding to the character *c*	`\cH` is a backspace (`\x{8}`)
`\`*c*, where *c* is not in `[A-Za-z0-9]`	The character *c*	`\\`
`\Q. . .\E`	Everything between the start and the end of the quotation	`\Q(. . .)\E` matches the string `(. . .)`
Character Classes		
[C_1C_2. . .], where C_i are characters, ranges *c*-*d*, or character classes	Any of the characters represented by C_1, C_2, . . .	`[0-9+-]`
`[^. . .]`	Complement of a character class	`[^\d\s]`
`[. . .&&. . .]`	Intersection of character classes	`[\p{L}&&[^A-Za-z]]`
`\p{. . .}`, `\P{. . .}`	A predefined character class (see Table 2.7); its complement	`\p{L}` matches a Unicode letter, and so does `\pL`—you can omit braces around a single letter

(Continues)

Table 2.6 *(Continued)*

Expression	Description	Example
`\d`, `\D`	Digits (`[0-9]`, or `\p{Digit}` when the `UNICODE_CHARACTER_CLASS` flag is set); the complement	`\d+` is a sequence of digits
`\w`, `\W`	Word characters (`[a-zA-Z0-9_]`, or Unicode word characters when the `UNICODE_CHARACTER_CLASS` flag is set); the complement	
`\s`, `\S`	Spaces (`[ \n\r\t\f\x{B}]`, or `\p{IsWhite_Space}` when the `UNICODE_CHARACTER_CLASS` flag is set); the complement	`\s*,\s*` is a comma surrounded by optional white space
`\h`, `\v`, `\H`, `\V`	Horizontal whitespace, vertical whitespace, their complements	
Sequences and Alternatives		
XY	Any string from `X`, followed by any string from `Y`	`[1-9][0-9]*` is a positive number without leading zero
X|*Y*	Any string from `X` or `Y`	`http|ftp`
Grouping		
(*X*)	Captures the match of *X*	`'([^']*)'` captures the quoted text
\n	The *n*th group	`(['"]).*\1` matches `'Fred'` or `"Fred"` but not `"Fred'`
(?<*name*>*X*)	Captures the match of *X* with the given name	`'(?<id>[A-Za-z0-9]+)'` captures the match with name `id`
\k<*name*>	The group with the given name	`\k<id>` matches the group with name `id`
(?:*X*)	Use parentheses without capturing *X*	In `(?:http|ftp)://(.*)`, the match after `://` is `\1`

(Continues)

Table 2.6 *(Continued)*

Expression	Description	Example
`(?`f_1f_2`. . .:`X`)`, `(?`f_1`. . .-`f_k`. . .:`X`)`, with f_i in `[dimsuUx]`	Matches, but does not capture, X with the given flags on or off (after `-`)	`(?i:jpe?g)` is a case-insensitive match
Other `(?. . .)`	See the `Pattern` API documentation	
Quantifiers		
X?	Optional *X*	`\+?` is an optional + sign
*X**, *X*+	0 or more *X*, 1 or more *X*	`[1-9][0-9]+` is an integer ≥ 10
X{*n*}, *X*{*n*,}, *X*{*m*,*n*}	*n* times *X*, at least *n* times `X`, between *m* and *n* times *X*	`[0-7]{1,3}` are one to three octal digits
Q?, where *Q* is a quantified expression	Reluctant quantifier, attempting the shortest match before trying longer matches	`.*(<.+?>).*` captures the shortest sequence enclosed in angle brackets
Q+, where *Q* is a quantified expression	Possessive quantifier, taking the longest match without backtracking	`'[^']*+'` matches strings enclosed in single quotes and fails quickly on strings without a closing quote
Boundary Matches		
`^`, `$`	Beginning, end of input (or beginning, end of line in multiline mode)	`^Java$` matches the input or line `Java`
`\A`, `\Z`, `\z`	Beginning of input, end of input, absolute end of input (unchanged in multiline mode)	
`\b`, `\B`	Word boundary, nonword boundary	`\bJava\b` matches the word `Java`
`\R`	A Unicode line break	
`\G`	The end of the previous match	

Table 2.7 Predefined Character Class Names Used with `\p`

Character Class Name	Explanation
posixClass	*posixClass* is one of `Lower`, `Upper`, `Alpha`, `Digit`, `Alnum`, `Punct`, `Graph`, `Print`, `Cntrl`, `XDigit`, `Space`, `Blank`, `ASCII`, interpreted as POSIX or Unicode class, depending on the `UNICODE_CHARACTER_CLASS` flag
`Is`*Script*, `sc=`*Script*, `script=`*Script*	A script accepted by `Character.UnicodeScript.forName`
`In`*Block*, `blk=`*Block*, `block=`*Block*	A block accepted by `Character.UnicodeBlock.forName`
Category, `In`*Category*, `gc=`*Category*, `general_category=`*Category*	A one- or two-letter name for a Unicode general category
`Is`*Property*	*Property* is one of `Alphabetic`, `Ideographic`, `Letter`, `Lowercase`, `Uppercase`, `Titlecase`, `Punctuation`, `Control`, `White_Space`, `Digit`, `Hex_Digit`, `Join_Control`, `Noncharacter_Code_Point`, `Assigned`
`java`*Method*	Invokes the method `Character.is`*Method* (must not be deprecated)

- Most characters match themselves, such as the `ava` characters in the preceding example.
- The `.` symbol matches any character (except possibly line terminators, depending on flag settings).
- Use `\` as an escape character. For example, `\.` matches a period and `\\` matches a backslash.
- `^` and `$` match the beginning and end of a line, respectively.
- If *X* and *Y* are regular expressions, then *XY* means "any match for *X* followed by a match for *Y*." *X* `|` *Y* means "any match for *X* or *Y*."
- You can apply *quantifiers* *X*`+` (1 or more), *X*`*` (0 or more), and *X*`?` (0 or 1) to an expression *X*.
- By default, a quantifier matches the largest possible repetition that makes the overall match succeed. You can modify that behavior with suffixes `?` (reluctant, or stingy, match: match the smallest repetition count) and `+` (possessive, or greedy, match: match the largest count even if that makes the overall match fail).

  For example, the string `cab` matches `[a-z]*ab` but not `[a-z]*+ab`. In the first case, the expression `[a-z]*` only matches the character `c`, so that the characters `ab` match the remainder of the pattern. But the greedy version `[a-z]*+` matches the characters `cab`, leaving the remainder of the pattern unmatched.

- You can use *groups* to define subexpressions. Enclose the groups in `( )`, for example, `([+-]?)([0-9]+)`. You can then ask the pattern matcher to return the match of each group or to refer back to a group with `\`*n* where *n* is the group number, starting with `\1`.

For example, here is a somewhat complex but potentially useful regular expression that describes decimal or hexadecimal integers:

```
[+-]?[0-9]+|0[Xx][0-9A-Fa-f]+
```

Unfortunately, the regular expression syntax is not completely standardized between various programs and libraries; there is a consensus on the basic constructs but many maddening differences in the details. The Java regular expression classes use a syntax that is similar to, but not quite the same as, the one used in the Perl language. Table 2.6 shows all constructs of the Java syntax. For more information on the regular expression syntax, consult the API documentation for the `Pattern` class or the book *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly and Associates, 2006).

2.7.2 Matching a String

The simplest use for a regular expression is to test whether a particular string matches it. Here is how you program that test in Java. First, construct a `Pattern` object from a string containing the regular expression. Then, get a `Matcher` object from the pattern and call its `matches` method:

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

The input of the matcher is an object of any class that implements the `CharSequence` interface, such as a `String`, `StringBuilder`, or `CharBuffer`.

When compiling the pattern, you can set one or more flags, for example:

```
Pattern pattern = Pattern.compile(expression,
   Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

Or you can specify them inside the pattern:

```
String regex = "(?iU:expression)";
```

Here are the flags:

- `Pattern.CASE_INSENSITIVE` or `i`: Match characters independently of the letter case. By default, this flag takes only US ASCII characters into account.

- `Pattern.UNICODE_CASE` or `u`: When used in combination with `CASE_INSENSITIVE`, use Unicode letter case for matching.
- `Pattern.UNICODE_CHARACTER_CLASS` or `U`: Select Unicode character classes instead of POSIX. Implies `UNICODE_CASE`.
- `Pattern.MULTILINE` or `m`: Make `^` and `$` match the beginning and end of a line, not the entire input.
- `Pattern.UNIX_LINES` or `d`: Only `'\n'` is a line terminator when matching `^` and `$` in multiline mode.
- `Pattern.DOTALL` or `s`: Make the `.` symbol match all characters, including line terminators.
- `Pattern.COMMENTS` or `x`: Whitespace and comments (from `#` to the end of a line) are ignored.
- `Pattern.LITERAL`: The pattern is taken literally and must be matched exactly, except possibly for letter case.
- `Pattern.CANON_EQ`: Take canonical equivalence of Unicode characters into account. For example, u followed by ¨ (diaeresis) matches ü.

The last two flags cannot be specified inside a regular expression.

If you want to match elements in a collection or stream, turn the pattern into a predicate:

```
Stream<String> strings = . . .;
Stream<String> result = strings.filter(pattern.asPredicate());
```

The result contains all strings that match the regular expression.

If the regular expression contains groups, the `Matcher` object can reveal the group boundaries. The methods

```
int start(int groupIndex)
int end(int groupIndex)
```

yield the starting index and the past-the-end index of a particular group.

You can simply extract the matched string by calling

```
String group(int groupIndex)
```

Group 0 is the entire input; the group index for the first actual group is 1. Call the `groupCount` method to get the total group count. For named groups, use the methods

```
int start(String groupName)
int end(String groupName)
String group(String groupName)
```

Nested groups are ordered by the opening parentheses. For example, given the pattern

```
(([1-9]|1[0-2]):([0-5][0-9]))[ap]m
```

and the input

```
11:59am
```

the matcher reports the following groups

Group Index	Start	End	String
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

Listing 2.6 prompts for a pattern, then for strings to match. It prints out whether or not the input matches the pattern. If the input matches and the pattern contains groups, the program prints the group boundaries as parentheses, for example:

```
((11):(59))am
```

Listing 2.6 regex/RegexTest.java

```
package regex;

import java.util.*;
import java.util.regex.*;

/**
 * This program tests regular expression matching. Enter a pattern and strings to match,
 * or hit Cancel to exit. If the pattern contains groups, the group boundaries are displayed
 * in the match.
 * @version 1.03 2018-05-01
 * @author Cay Horstmann
 */
public class RegexTest
{
   public static void main(String[] args) throws PatternSyntaxException
   {
      var in = new Scanner(System.in);
      System.out.println("Enter pattern: ");
      String patternString = in.nextLine();

      Pattern pattern = Pattern.compile(patternString);

      while (true)
      {
         System.out.println("Enter string to match: ");
         String input = in.nextLine();
         if (input == null || input.equals("")) return;
         Matcher matcher = pattern.matcher(input);
         if (matcher.matches())
         {
            System.out.println("Match");
            int g = matcher.groupCount();
            if (g > 0)
            {
               for (int i = 0; i < input.length(); i++)
               {
                  // Print any empty groups
                  for (int j = 1; j <= g; j++)
                     if (i == matcher.start(j) && i == matcher.end(j))
                        System.out.print("()");
                  // Print ( for non-empty groups starting here
                  for (int j = 1; j <= g; j++)
                     if (i == matcher.start(j) && i != matcher.end(j))
                        System.out.print('(');
                  System.out.print(input.charAt(i));
                  // Print ) for non-empty groups ending here
                  for (int j = 1; j <= g; j++)
                     if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
                        System.out.print(')');
               }
               System.out.println();
            }
         }
         else
            System.out.println("No match");
      }
   }
}
```

2.7.3 Finding Multiple Matches

Usually, you don't want to match the entire input against a regular expression, but to find one or more matching substrings in the input. Use the `find` method of the `Matcher` class to find the next match. If it returns `true`, use the `start` and `end` methods to find the extent of the match or the `group` method without an argument to get the matched string.

```
while (matcher.find())
{
   int start = matcher.start();
   int end = matcher.end();
   String match = matcher.group();
   . . .
}
```

In this way, you can process each match in turn. As shown in the code fragment, you can get the matched string as well as its position in the input string.

More elegantly, you can call the `results` method to get a `Stream<MatchResult>`. The `MatchResult` interface has methods `group`, `start`, and `end`, just like `Matcher`. (In fact, the `Matcher` class implements this interface.) Here is how you get a list of all matches:

```
List<String> matches = pattern.matcher(input)
   .results()
   .map(Matcher::group)
   .collect(Collectors.toList());
```

If you have the data in a file, you can use the `Scanner.findAll` method to get a `Stream<MatchResult>`, without first having to read the contents into a string. You can pass a `Pattern` or a pattern string:

```
var in = new Scanner(path, StandardCharsets.UTF_8);
Stream<String> words = in.findAll("\\pL+")
   .map(MatchResult::group);
```

Listing 2.7 puts this mechanism to work. It locates all hypertext references in a web page and prints them. To run the program, supply a URL on the command line, such as

```
java match.HrefMatch http://horstmann.com
```

Listing 2.7 `match/HrefMatch.java`

```
package match;

import java.io.*;
import java.net.*;
import java.nio.charset.*;
import java.util.regex.*;

/**
 * This program displays all URLs in a web page by matching a regular expression that
 * describes the <a href=...> HTML tag. Start the program as <br>
 * java match.HrefMatch URL
 * @version 1.03 2018-03-19
 * @author Cay Horstmann
 */
public class HrefMatch
{
   public static void main(String[] args)
   {
      try
      {
         // get URL string from command line or use default
         String urlString;
         if (args.length > 0) urlString = args[0];
         else urlString = "http://openjdk.java.net/";

         // read contents of URL
         InputStream in = new URL(urlString).openStream();
         var input = new String(in.readAllBytes(), StandardCharsets.UTF_8);

         // search for all occurrences of pattern
         var patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>]*)\\s*>";
         Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
         pattern.matcher(input)
            .results()
            .map(MatchResult::group)
            .forEach(System.out::println);
      }
      catch (IOException | PatternSyntaxException e)
      {
         e.printStackTrace();
      }
   }
}
```

2.7.4 Splitting along Delimiters

Sometimes, you want to break an input along matched delimiters and keep everything else. The `Pattern.split` method automates this task. You obtain an array of strings, with the delimiters removed:

```
String input = . . .;
Pattern commas = Pattern.compile("\\s*,\\s*");
String[] tokens = commas.split(input);
   // "1, 2, 3" turns into ["1", "2", "3"]
```

If there are many tokens, you can fetch them lazily:

```
Stream<String> tokens = commas.splitAsStream(input);
```

If you don't care about precompiling the pattern or lazy fetching, you can just use the `String.split` method:

```
String[] tokens = input.split("\\s*,\\s*");
```

If the input is in a file, use a scanner:

```
var in = new Scanner(path, StandardCharsets.UTF_8);
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

2.7.5 Replacing Matches

The `replaceAll` method of the `Matcher` class replaces all occurrences of a regular expression with a replacement string. For example, the following instructions replace all sequences of digits with a `#` character:

```
Pattern pattern = Pattern.compile("[0-9]+");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");
```

The replacement string can contain references to the groups in the pattern: `$`*n* is replaced with the *n*th group, and `${`*name*`}` is replaced with the group that has the given name. Use `\$` to include a `$` character in the replacement text.

If you have a string that may contain `$` and `\`, and you don't want them to be interpreted as group replacements, call `matcher.replaceAll(Matcher.quoteReplacement(str))`.

If you want to carry out a more complex operation than splicing in group matches, you can provide a replacement function instead of a replacement string. The function accepts a `MatchResult` and yields a string. For example, here we replace all words with at least four letters with their uppercase version:

```
String result = Pattern.compile("\\pL{4,}")
   .matcher("Mary had a little lamb")
   .replaceAll(m -> m.group().toUpperCase());
   // Yields "MARY had a LITTLE LAMB"
```

The `replaceFirst` method replaces only the first occurrence of the pattern.

java.util.regex.Pattern 1.4

- `static Pattern compile(String expression)`
- `static Pattern compile(String expression, int flags)`

  compiles the regular expression string into a pattern object for fast processing of matches. The `flags` parameter has one or more of the bits `CASE_INSENSITIVE`, `UNICODE_CASE`, `MULTILINE`, `UNIX_LINES`, `DOTALL`, and `CANON_EQ` set.

(Continues)

`java.util.regex.Pattern` 1.4 *(Continued)*

- `Matcher matcher(CharSequence input)`

  returns a `matcher` object that you can use to locate the matches of the pattern in the input.
- `String[] split(CharSequence input)`
- `String[] split(CharSequence input, int limit)`
- `Stream<String> splitAsStream(CharSequence input)` 8

  splits the input string into tokens, with the pattern specifying the form of the delimiters. Returns an array or stream of tokens. The delimiters are not part of the tokens. The second form has a parameter `limit` denoting the maximum number of strings to produce. If `limit - 1` matching delimiters have been found, then the last entry of the returned array contains the remaining unsplit input. If `limit` is ≤ `0`, then the entire input is split. If `limit` is `0`, then trailing empty strings are not placed in the returned array.

`java.util.regex.Matcher` 1.4

- `boolean matches()`

  returns `true` if the input matches the pattern.
- `boolean lookingAt()`

  returns `true` if the beginning of the input matches the pattern.
- `boolean find()`
- `boolean find(int start)`

  attempts to find the next match and returns `true` if another match is found.
- `int start()`
- `int end()`

  returns the start or past-the-end position of the current match.
- `String group()`

  returns the current match.
- `int groupCount()`

  returns the number of groups in the input pattern.

(Continues)

`java.util.regex.Matcher` 1.4 *(Continued)*

- `int start(int groupIndex)`
- `int start(String name)` 8
- `int end(int groupIndex)`
- `int end(String name)` 8

  returns the start or past-the-end position of a group in the current match. The group is specified by an index starting with `1`, or `0` to indicate the entire match, or by a string identifying a named group.

- `String group(int groupIndex)`
- `String group(String name)` 7

  returns the string matching a given group, denoted by an index starting with `1`, or `0` to indicate the entire match, or by a string identifying a named group.

- `String replaceAll(String replacement)`
- `String replaceFirst(String replacement)`

  returns a string obtained from the matcher input by replacing all matches, or the first match, with the replacement string. The replacement string can contain references to pattern groups as `$`*n*. Use `\$` to include a `$` symbol.

- `static String quoteReplacement(String str)` 5.0

  quotes all `\` and `$` in `str`.

- `String replaceAll(Function<MatchResult,String> replacer)` 9

  replaces every match with the result of the `replacer` function applied to the `MatchResult`.

- `Stream<MatchResult> results()` 9

  yields a stream of all match results.

`java.util.regex.MatchResult` 5

- `String group()`
- `String group(int group)`

  yields the matched string or the string matched by the given group.

- `int start()`
- `int end()`
- `int start(int group)`
- `int end(int group)`

  yields the start and end offsets of the matched string or the string matched by the given group.

java.util.Scanner 5.0

- `Stream<MatchResult> findAll(Pattern pattern)` 9

  yields a stream of all matches of the given pattern in the input produced by this scanner.

You have now seen how to carry out input and output operations in Java, and had an overview of the regular expression package that was a part of the "new I/O" specification. In the next chapter, we turn to the processing of XML data.