http://www.rejoiceblog.com/

CORE JAVA

Volume II—Advanced Features

ELEVENTH EDITION

CAY S. HORSTMANN

http://www.rejoiceblog.com/

Core Java, Volume II—Advanced Features

Eleventh Edition

Cay S. Horstmann

Pearson

http://www.rejoiceblog.com/

Chapter 1 Streams

In this chapter

- 1.1 From Iterating to Stream Operations
- 1.2 Stream Creation
- 1.3 The `filter`, `map`, and `flatMap` Methods
- 1.4 Extracting Substreams and Combining Streams
- 1.5 Other Stream Transformations
- 1.6 Simple Reductions
- 1.7 The Optional Type
- 1.8 Collecting Results
- 1.9 Collecting into Maps
- 1.10 Grouping and Partitioning
- 1.11 Downstream Collectors
- 1.12 Reduction Operations
- 1.13 Primitive Type Streams
- 1.14 Parallel Streams

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections. 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

http://www.rejoiceblog.com/

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 = new String(Files.readAllBytes(
   Paths.get("alice.txt")), StandardCharsets.UTF_8); // Read
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()
```

http://www.rejoiceblog.com/

```
    .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` into `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 it 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,

http://www.rejoiceblog.com/

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`

```
package streams;

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

http://www.rejoiceblog.com/

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

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

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

http://www.rejoiceblog.com/

```

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

            count = words.parallelStream().filter(w -> w.len
            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.

http://www.rejoiceblog.com/

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", "str
```

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.<Stri
```

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

http://www.rejoiceblog.com/

```
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.ON
```

The first element in the sequence is the seed `BigInteger.ZERO`. The second element is `f(seed)`, or `1` (as a big integer). The next element is `f(f(seed))`, or `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.

http://www.rejoiceblog.com/

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(
```

http://www.rejoiceblog.com/

```
iterator, Spliterator.ORDERED), false);
```

Caution

It is very important that you don't modify the collection that is backing a stream while carrying out a stream operation. Remember that streams don't collect their data—that data is always in a separate collection. If you were to modify that collection, the outcome of the stream operations would be 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 when 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.

http://www.rejoiceblog.com/

Listing 1.2 `streams/CreatingStreams.java`

```
1   package streams;
2
3   /**
4    * @version 1.01 2018-05-01
5    * @author Cay Horstmann
6    */
7
8   import   java.io.IOException;
9   import   java.math.BigInteger;
10  import   java.nio.charset.StandardCharsets;
11  import   java.nio.file.*;
12  import   java.util.*;
13  import   java.util.regex.Pattern;
14  import   java.util.stream.*;
15
16  public class CreatingStreams
17  {
18     public static <T> void show(String title, Stream<T> s
```

http://www.rejoiceblog.com/

```
19      {
20         final int SIZE = 10;
21         List<T> firstElements = stream
22               .limit(SIZE + 1)
23               .collect(Collectors.toList());
24         System.out.print(title + ": ");
25         for (int i = 0; i < firstElements.size(); i++)
26         {
27            if (i > 0) System.out.print(", ");
28            if (i < SIZE) System.out.print(firstElements.ge
29            else System.out.print("...");
30         }
31         System.out.println();
32      }
33
34      public static void main(String[] args) throws IOExcept
35      {
36         Path path = Paths.get("../gutenberg/alice30.txt");
37         var contents = new String(Files.readAllBytes(path),
38               StandardCharsets.UTF_8);
```

http://www.rejoiceblog.com/

```
39
40          Stream<String> words = Stream.of(contents.split("\
41          show("words", words);
42          Stream<String> song = Stream.of("gently", "down",
43          show("song", song);
44          Stream<String> silence = Stream.empty();
45          show("silence", silence);
46
47          Stream<String> echos = Stream.generate(() -> "Echo
48          show("echos", echos);
49
50          Stream<Double> randoms = Stream.generate(Math::ran
51          show("randoms", randoms);
52
53          Stream<BigInteger> integers = Stream.iterate(BigIn
54                n -> n.add(BigInteger.ONE));
55          show("integers", integers);
56
57          Stream<String> wordsAnotherWay = Pattern.compile("
```

http://www.rejoiceblog.com/

```
            contents);
        show("wordsAnotherWay", wordsAnotherWay);

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

        Iterable<Path> iterable = FileSystems.getDefault()
        Stream<Path> rootDirectories = StreamSupport.strea
        show("rootDirectories", rootDirectories);

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

http://www.rejoiceblog.com/

- `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`

http://www.rejoiceblog.com/

`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.

http://www.rejoiceblog.com/

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.lengt
```

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::to
```

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

http://www.rejoiceblog.com/

```
Stream<String> firstLetters = words.stream().map(s -> s.subst
```

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 -> codeP
```

http://www.rejoiceblog.com/

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 -> codeP

   // 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

http://www.rejoiceblog.com/

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(
```

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+")).ski
```

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).dro
```

http://www.rejoiceblog.com/

```
    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", "
```

Of course, the first stream should not be infinite—otherwise the second 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)`

http://www.rejoiceblog.com/

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").dis

   // 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::lengt
```

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))
```

http://www.rejoiceblog.com/

```
    .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 as 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

http://www.rejoiceblog.com/

`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 that we cover 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::compareToIgnoreC

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.

http://www.rejoiceblog.com/

```
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)`

http://www.rejoiceblog.com/

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.getProp

   // The function is only called when needed
```

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

```
String result = optionalString.orElseThrow(IllegalStateExcept

   // Supply a method that yields an exception object
```

http://www.rejoiceblog.com/

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);
```

http://www.rejoiceblog.com/

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 invoke `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::toU
```

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);
```

http://www.rejoiceblog.com/

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 a

   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)`

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

```
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

http://www.rejoiceblog.com/

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);
```

http://www.rejoiceblog.com/

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::squareRo
```

or, if you prefer,

```
Optional<Double> result

   = Optional.of(-4.0).flatMap(Demo::inverse).flatMap(Demo::s
```

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

http://www.rejoiceblog.com/

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 elements.

java.util.Optional 8

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

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

1.7.7 Turning an Optional Into a Stream

The `stream` method turns an `Optional<T>` into a `Stream<T>` with zero or one elements. 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);
```

http://www.rejoiceblog.com/

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 0 or 1 elements. 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.

http://www.rejoiceblog.com/

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 IOExcep
   {
      var contents = new String(Files.readAllBytes(
```

http://www.rejoiceblog.com/

```
            Paths.get("../gutenberg/alice30.txt")), Stan
        List<String> wordList = List.of(contents.split("\\

        Optional<String> optionalValue = wordList.stream()
           .filter(s -> s.contains("fred"))
           .findFirst();
          System.out.println(optionalValue.orElse("No word

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

http://www.rejoiceblog.com/

```
            {
                t.printStackTrace();
            }

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

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

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

http://www.rejoiceblog.com/

```
        }

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

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

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

http://www.rejoiceblog.com/

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. You can call `toArray` and 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 the 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:

http://www.rejoiceblog.com/

```
TreeSet<String> result = stream.collect(Collectors.toCollecti
```

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(Collecto
```

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`

http://www.rejoiceblog.com/

```
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 IOExce
   {
      var contents = new String(Files.readAllBytes(
            Paths.get("../gutenberg/alice30.txt")),
```

http://www.rejoiceblog.com/

```
            StandardCharsets.UTF_8);
        List<String> wordList = List.of(contents.split("\\
        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().getN
        System.out.println("["
                + set.stream().limit(10).map(Object::toString
                        .collect(Collectors.joining(", ")) + "]
    }

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

http://www.rejoiceblog.com/

```

        Object[] numbers = Stream.iterate(0, n -> n + 1).l
        System.out.println("Object array:" + numbers); //

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

        Integer[] numbers3 = Stream.iterate(0, n -> n + 1)
            .toArray(Integer[]::new);
        System.out.println("Integer array: " + numbers3);
```

http://www.rejoiceblog.com/

```

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

        TreeSet<String> noVowelTreeSet = noVowels().collec
              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: " + res

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

http://www.rejoiceblog.com/

```
        System.out.println("Average word length: " + ave
        System.out.println("Max word length: " + maxWord
        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.

http://www.rejoiceblog.com/

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

yields 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)`

http://www.rejoiceblog.com/

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.

- `(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

http://www.rejoiceblog.com/

`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 13 for more information about working with locales.

http://www.rejoiceblog.com/

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.collec

   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(),
```

http://www.rejoiceblog.com/

```
(existingValue, newValue) -> { throw new IllegalStateExc

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

Listing 1.5 `collecting/CollectingIntoMaps.java`

```
package collecting;

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

import   java.io.*;
import   java.util.*;
```

http://www.rejoiceblog.com/

```
10   import   java.util.function.*;
11   import   java.util.stream.*;
12
13   public class CollectingIntoMaps
14   {
15
16     public static class Person
17     {
18        private int id;
19        private String name;
20
21         public Person(int id, String name)
22         {
23            this.id = id;
24            this.name = name;
25         }
26
27         public int getId()
28         {
```

http://www.rejoiceblog.com/

```
            return id;
        }

        public String getName()
        {
            return name;
        }

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

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

```

http://www.rejoiceblog.com/

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

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

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

```

http://www.rejoiceblog.com/

```
        Stream<Locale> locales = Stream.of(Locale.getAva
        Map<String, String> languageNames = locales.coll
            Collectors.toMap(
                Locale::getDisplayLanguage,
                l -> l.getDisplayLanguage(l),
                (existingValue, newValue) -> existingValue
        System.out.println("languageNames: " + languageN

        locales = Stream.of(Locale.getAvailableLocales()
        Map<String, Set<String>> countryLanguageSets = l
            Collectors.toMap(
                Locale::getDisplayCountry,
                l -> List.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: " + cou
    }
```

http://www.rejoiceblog.com/

```
}
```

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>

http://www.rejoiceblog.com/

```
toConcurrentMap(Function<? super T,? extends K> keyMapper,
Function<? super T,? extends U> valueMapper, BinaryOperator<U>
mergeFunction, Supplier<M> mapSupplier)
```

Yield 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, and 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

http://www.rejoiceblog.com/

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.c

   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>`

http://www.rejoiceblog.com/

```
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(
```

http://www.rejoiceblog.com/

```
    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.colle

    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 = string

    groupingBy(s -> s.charAt(0),

      collectingAndThen(toSet(), Set::size)));
```

http://www.rejoiceblog.com/

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

```
Map<Character, Set<Integer>> stringLengthsByStartingLetter =

    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> stateToCityPopulationSummar

   groupingBy(City::getState,

      summarizingInt(City::getPopulation)));
```

http://www.rejoiceblog.com/

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 e
```

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 also 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`

```
1   package collecting;

2
```

http://www.rejoiceblog.com/

```
/**
 * @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;
```

http://www.rejoiceblog.com/

```

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

        public String getName()
        {
            return name;
        }

        public String getState()
        {
            return state;
        }

        public int getPopulation()
```

http://www.rejoiceblog.com/

```
        {
            return population;
        }
    }

    public static Stream<City> readCities(String filename)
    {
        return Files.lines(Paths.get(filename)).map(l -> l.
                .map(a -> new City(a[0], a[1], Integer.parseI
    }

    public static void main(String[] args) throws IOExcept
    {
        Stream<Locale> locales = Stream.of(Locale.getAvaila
        locales = Stream.of(Locale.getAvailableLocales());
        Map<String, Set<Locale>> countryToLocaleSet = local
                Locale::getCountry, toSet()));
        System.out.println("countryToLocaleSet: " + country

        locales = Stream.of(Locale.getAvailableLocales());
```

http://www.rejoiceblog.com/

```
        Map<String, Long> countryToLocaleCounts = locales.
            Locale::getCountry, counting()));
        System.out.println("countryToLocaleCounts: " + cou

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

          cities = readCities("cities.txt");
          Map<String, Optional<String>> stateToLongestCit
              .collect(groupingBy(
                  City::getState,
                  mapping(City::getName,
                      maxBy(Comparator.comparing(St

          System.out.println("stateToLongestCityName: " +

          locales = Stream.of(Locale.getAvailableLocales(
```

http://www.rejoiceblog.com/

```
        Map<String, Set<String>> countryToLanguages = l
              Locale::getDisplayCountry,
              mapping(Locale::getDisplayLanguage, toSet
        System.out.println("countryToLanguages: " + cou

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

        cities = readCities("cities.txt");
        Map<String, String> stateToCityNames = cities.c
              City::getState,
              reducing("", City::getName, (s, t) -> s.l
                    + ", " + t)));

        cities = readCities("cities.txt");
        stateToCityNames = cities.collect(groupingBy(Ci
              mapping(City::getName, joining(", "))));
```

http://www.rejoiceblog.com/

```
            System.out.println("stateToCityNames: " + state
        }
    }
```

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 values 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)`

http://www.rejoiceblog.com/

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 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 +
```

http://www.rejoiceblog.com/

In this case, the `reduce` method computes $v_0 + v_1 + v_2 + \ldots$, where the 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 $v_0\ op\ v_1\ op\ v_2 op \ldots$, 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 and 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 you can use that element as the start of the computation. For example, 0 is the identity for addition. Then call 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,

http://www.rejoiceblog.com/

such as all 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, max, or min. (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 threadsafe. 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:

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

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`, and 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
```

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); // Uppe

IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // U
```

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 , uni
```

```
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

http://www.rejoiceblog.com/

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.

http://www.rejoiceblog.com/

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
{
```

http://www.rejoiceblog.com/

```
19      public static void show(String title, IntStream strea
20      {
21         final int SIZE = 10;
22           int[] firstElements = stream.limit(SIZE + 1).toA
23           System.out.print(title + ": ");
24           for (int i = 0; i < firstElements.length; i++)
25           {
26              if (i > 0) System.out.print(", ");
27              if (i < SIZE) System.out.print(firstElements[
28              else System.out.print("...");
29           }
30           System.out.println();
31       }
32
33       public static void main(String[] args) throws IOExce
34       {
35          IntStream is1 = IntStream.generate(() -> (int) (M
36          show("is1", is1);
37          IntStream is2 = IntStream.range(5, 10);
```

http://www.rejoiceblog.com/

```
        show("is2", is2);

        IntStream is3 = IntStream.rangeClosed(5, 10);

        show("is3", is3);


         Path path = Paths.get("../gutenberg/alice30.txt"

         var contents = new String(Files.readAllBytes(pat


         Stream<String> words = Stream.of(contents.split(

         IntStream is4 = words.mapToInt(String::length);

         show("is4", is4);

         var sentence = "\uD835\uDD46 is the set of octon

         System.out.println(sentence);

         IntStream codes = sentence.codePoints();

         System.out.println(codes.mapToObj(c -> String.fo

               Collectors.joining()));


         Stream<Integer> integers = IntStream.range(0, 10

         IntStream is5 = integers.mapToInt(Integer::intVa

         show("is5", is5);

      }
```

http://www.rejoiceblog.com/

```
}
```

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, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results can be obtained.

- `Stream<Integer> boxed()`

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

java.util.stream.LongStream 8

http://www.rejoiceblog.com/

- `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, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results 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()`

http://www.rejoiceblog.com/

yields an array with the elements of this stream.

- `double sum()`
- `OptionalDouble average()`
- `OptionalDouble max()`
- `OptionalDouble min()`
- `DoubleSummaryStatistics summaryStatistics()`

yields the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results 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`

http://www.rejoiceblog.com/

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)

http://www.rejoiceblog.com/

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.

http://www.rejoiceblog.com/

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 14 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

http://www.rejoiceblog.com/

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().li
```

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().co

   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
```

http://www.rejoiceblog.com/

```
    = words.parallelStream()

      .collect(

        groupingByConcurrent(

          String::length,

          counting()));
```

Don't turn all your streams into parallel streams with the hope of speeding up their 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

http://www.rejoiceblog.com/

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 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;

```

http://www.rejoiceblog.com/

```
 3   /**
 4    * @version 1.01 2018-05-01
 5    * @author Cay Horstmann
 6    */
 7
 8   import static java.util.stream.Collectors.*;
 9
10   import   java.io.*;
11   import   java.nio.charset.*;
12   import   java.nio.file.*;
13   import   java.util.*;
14   import   java.util.stream.*;
15
16   public class ParallelStreams
17   {
18      public static void main(String[] args) throws IOExcep
19      {
20         var contents = new String(Files.readAllBytes(
21               Paths.get("../gutenberg/alice30.txt")), Stan
```

http://www.rejoiceblog.com/

```
22          List<String> wordList = List.of(contents.split("
23
24          // Very bad code ahead
25          var shortWords = new int[10];
26          wordList.parallelStream().forEach(s ->
27              {
28                  if (s.length() < 10) shortWords[s.length(
29              });
30          System.out.println(Arrays.toString(shortWords));
31
32          // Try again--the result will likely be differen
33          Arrays.fill(shortWords, 0);
34          wordList.parallelStream().forEach(s ->
35              {
36                  if (s.length() < 10) shortWords[s.length(
37              });
38          System.out.println(Arrays.toString(shortWords));
39
40          // Remedy: Group and count
41          Map<Integer, Long> shortWordCounts = wordList.pa
```

http://www.rejoiceblog.com/

```
            .filter(s -> s.length() < 10)
            .collect(groupingBy(String::length, counting(

        System.out.println(shortWordCounts);

        // Downstream order not deterministic
        Map<Integer, List<String>> result = wordList.par
            Collectors.groupingByConcurrent(String::lengt

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

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

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

        Map<Integer, Long> wordCounts = wordList.paralle
            groupingByConcurrent(String::length, counting

```

http://www.rejoiceblog.com/

```
            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.

http://www.rejoiceblog.com/

Chapter 2 Input and Output

In this chapter

- 2.1 Input/Output Streams
- 2.2 Reading and Writing Binary Data
- 2.3 Object Input/Output Streams and Serialization
- 2.4 Working with Files
- 2.5 Memory-Mapped Files
- 2.6 File Locking
- 2.7 Regular Expressions

In this chapter, we will 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` form the basis for a hierarchy of input/output

http://www.rejoiceblog.com/

(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 for processing Unicode characters that inherit from the abstract `Reader` and `Writer` classes. 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();
```

http://www.rejoiceblog.com/

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)

{
```

http://www.rejoiceblog.com/

```
      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

http://www.rejoiceblog.com/

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.

http://www.rejoiceblog.com/

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`.

http://www.rejoiceblog.com/

```
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.)

Figure 2.1 Input and output stream hierarchy

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

http://www.rejoiceblog.com/

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

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

http://www.rejoiceblog.com/

> `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)
```

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.)

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

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

http://www.rejoiceblog.com/

```
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)`

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

```
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

http://www.rejoiceblog.com/

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, you need to 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(
```

http://www.rejoiceblog.com/

```
    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 push-back 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 din = new DataInputStream(

   pbin = new PushbackInputStream(

      new BufferedInputStream(

        new FileInputStream("employee.dat"))));
```

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 truly 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.zi

var din = new DataInputStream(zin);
```

http://www.rejoiceblog.com/

Figure 2.4 A sequence of filtered input streams

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

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)`

http://www.rejoiceblog.com/

- `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`.

http://www.rejoiceblog.com/

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. 77.

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 are encoded in 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"
```

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.

http://www.rejoiceblog.com/

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.UT
```

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")`.

http://www.rejoiceblog.com/

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.U
   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(Writer out)`

http://www.rejoiceblog.com/

- `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.

http://www.rejoiceblog.com/

- `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 = new String(Files.readAllBytes(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,

http://www.rejoiceblog.com/

```
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(inputS

{

   String line;

   while ((line = in.readLine()) != null)
```

http://www.rejoiceblog.com/

```
    {
        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 `\n`. This work is done in the following `writeData` method that we add to our `Employee` class:

```
public static void writeEmployee(PrintWriter out, Employee e)
{
```

http://www.rejoiceblog.com/

```
    out.println(e.getName() + "|" + e.getSalary() + "|" + e.ge

}
```

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.

http://www.rejoiceblog.com/

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(BufferedReader 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`

```
1  package textFile;
```

http://www.rejoiceblog.com/

```
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 IOExcept
   {
      var staff = new Employee[3];

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

```

http://www.rejoiceblog.com/

```
22         // save all employee records to the file employee.d
23         try (var out = new PrintWriter("employee.dat", Stan
24         {
25            writeData(staff, out);
26         }
27
28         // retrieve all records into a new array
29         try (var in = new Scanner(
30            new FileInputStream("employee.dat"), "UTF-8"))
31         {
32            Employee[] newStaff = readData(in);
33
34            // print the newly read employee records
35            for (Employee e : newStaff)
36               System.out.println(e);
37      }
38   }
39
40   /**
```

http://www.rejoiceblog.com/

```
41   * Writes all employees in an array to a print writer
42   * @param employees an array of employees
43   * @param out a print writer
44   */
45   private static void writeData(Employee[] employees, Prin
46   {
47      // write number of employees
48      out.println(employees.length);
49
50      for (Employee e : employees)
51        writeEmployee(out, e);
52   }
53
54  /**
55  * Reads an array of employees from a scanner
56  * @param in the scanner
57  * @return the array of employees
58  */
59  private static Employee[] readData(Scanner in)
60  {
```

http://www.rejoiceblog.com/

```
    // 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, Employ
  {
        out.println(e.getName() + "|" + e.getSalary() + "|"
```

http://www.rejoiceblog.com/

```
  }

  /**
  * 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);
  }
}
```

http://www.rejoiceblog.com/

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 big-endian format, the more significant byte comes first: `0x21` followed by `0x22`. In 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 discard it.

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

`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

writeByte

writeInt

writeShort

writeLong

writeFloat

writeDouble

writeChar
```

http://www.rejoiceblog.com/

```
writeBoolean

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 the space needed will be the same 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`, say the decimal number 1234, or 4D2 in hexadecimal (1234 = 4 × 256 + 13 × 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 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.

http://www.rejoiceblog.com/

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

```
readInt

readShort

readLong

readFloat
```

```
readDouble

readChar

readBoolean
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.da
```

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

```
var out = new DataOutputStream(new FileOutputStream("employee
```

java.io.DataInput 1.0

- `boolean readBoolean()`

http://www.rejoiceblog.com/

- `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.

- `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)`

http://www.rejoiceblog.com/

- `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.

http://www.rejoiceblog.com/

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.

http://www.rejoiceblog.com/

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, DataO

   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);
```

http://www.rejoiceblog.com/

```
    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());
```

http://www.rejoiceblog.com/

```
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 contains 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 at the last record first.

Listing 2.2 `randomAccess/RandomAccessTest.java`

```
package randomAccess;

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

http://www.rejoiceblog.com/

```

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

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

    try (var out = new DataOutputStream(new FileOutputStr
    {
      // save all employee records to the file employee.d
      for (Employee e : staff)
```

http://www.rejoiceblog.com/

```
        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
```

http://www.rejoiceblog.com/

```
44        for (Employee e : newStaff)
45          System.out.println(e);
46     }
47  }
48
49  /**
50   * Writes employee data to a data output
51   * @param out the data output
52   * @param e the employee
53   */
54  public static void writeData(DataOutput out, Employee e)
55  {
56    DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE
57    out.writeDouble(e.getSalary());
58
59    LocalDate hireDay = e.getHireDay();
60    out.writeInt(hireDay.getYear());
61    out.writeInt(hireDay.getMonthValue());
62    out.writeInt(hireDay.getDayOfMonth());
```

http://www.rejoiceblog.com/

```
63  }
64
65  /**
66   * Reads employee data from a data input
67   * @param in the data input
68   * @return the employee
69   */
70  public static Employee readData(DataInput in) throws IOEx
71  {
72   String name = DataIO.readFixedString(Employee.NAME_SIZE,
73   double salary = in.readDouble();
74   int y = in.readInt();
75   int m = in.readInt();
76   int d = in.readInt();
77   return new Employee(name, salary, y, m - 1, d);
78  }
79 }
```

java.io.RandomAccessFile 1.0

- RandomAccessFile(String file, String mode)

http://www.rejoiceblog.com/

- `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.

- `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
```

http://www.rejoiceblog.com/

```
        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 you are 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 13) 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

http://www.rejoiceblog.com/
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.

http://www.rejoiceblog.com/

- `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. 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.

http://www.rejoiceblog.com/

- `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

http://www.rejoiceblog.com/

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("employ
```

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.
```

http://www.rejoiceblog.com/

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 that we need to consider: What happens when one object is shared by several objects as part of its state?

To illustrate the problem, let us make a slight modification to the `Manager` class. Let's assume that each manager has a secretary:

http://www.rejoiceblog.com/

```
class Manager extends Employee

{

   private Employee secretary;

   . . .

}
```

Each `Manager` object now contains a reference to the `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);
```

http://www.rejoiceblog.com/

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).

http://www.rejoiceblog.com/

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 back the objects, 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.

http://www.rejoiceblog.com/

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 communicating 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`

```
package objectStream;

import java.io.*;

/**
 * @version 1.11 2018-05-01
 * @author Cay Horstmann
 */
class ObjectStreamTest
```

http://www.rejoiceblog.com/

```
10  {
11     public static void main(String[] args) throws IOExcept
12     {
13       var harry = new Employee("Harry Hacker", 50000, 1989
14       var carl = new Manager("Carl Cracker", 80000, 1987,
15       carl.setSecretary(harry);
16       var tony = new Manager("Tony Tester", 40000, 1990, 3
17       tony.setSecretary(harry);
18
19       var staff = new Employee[3];
20
21       staff[0] = carl;
22       staff[1] = harry;
23       staff[2] = tony;
24
25       // save all employee records to the file employee.da
26       try (var out = new ObjectOutputStream(new FileOutput
27       {
28         out.writeObject(staff);
29       }
```

http://www.rejoiceblog.com/

```

      try (var in = new ObjectInputStream(new FileInputStr
      {
        // 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)`

http://www.rejoiceblog.com/

creates an `ObjectOutputStream` so that 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, non-transient 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 nontransient and nonstatic fields of the class and all its superclasses. It does deserializing to allow multiple object references to 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

http://www.rejoiceblog.com/

```
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” to 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

http://www.rejoiceblog.com/

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 d

static final byte SC_SERIALIZABLE = 2;

   // class implements the Serializable interface

static final byte SC_EXTERNALIZABLE = 4;
```

http://www.rejoiceblog.com/

```
// 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` inter-face 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
```

http://www.rejoiceblog.com/

```
[    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
```

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

```
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

http://www.rejoiceblog.com/

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)

http://www.rejoiceblog.com/

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 that 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, and these methods are called instead.

http://www.rejoiceblog.com/

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());

}
```

http://www.rejoiceblog.com/

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(ObjectInputStream in)

    throws IOException, ClassNotFoundException;

public void writeExternal(ObjectOutputStream out)
```

http://www.rejoiceblog.com/

```
    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());
```

http://www.rejoiceblog.com/

```
}
```

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 Orientati

   public static final Orientation VERTICAL = new Orientation



   private int value;



   private Orientation(int v) { value = v; }

}
```

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

`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 happe

}
```

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 finger-prints. 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 = -1814239825517
```

http://www.rejoiceblog.com/

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 = -1814239825517

}
```

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,

http://www.rejoiceblog.com/

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 knew 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.

http://www.rejoiceblog.com/

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`

```
package serialClone;

/**
 * @version 1.22 2018-05-01
 * @author Cay Horstmann
```

http://www.rejoiceblog.com/

```
 */

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

public class SerialCloneTest
{
   public static void main(String[] args) throws CloneNot
   {
      var harry = new Employee("Harry Hacker", 35000, 198
      // 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);
```

http://www.rejoiceblog.com/

```
    }
}

/**
 * A class whose clone method uses serialization.
 */
class SerialCloneable implements Cloneable, Serializable
{
    public Object clone() throws CloneNotSupportedExceptio
    {
        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.toBy
```

http://www.rejoiceblog.com/

```
        {
            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.
 */
class Employee extends SerialCloneable
```

http://www.rejoiceblog.com/

```
{
    private String namae;
    private double salary;
    private LocalDate hireDay;

    public Employee(String n, double s, int year, int mont
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
```

http://www.rejoiceblog.com/

```
    }

    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()
  {
```

http://www.rejoiceblog.com/

```
    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 the 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");
```

http://www.rejoiceblog.com/

```
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

Path basePath = Paths.get(baseDir); // OK that baseDir has se
```

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");
```

http://www.rejoiceblog.com/

```
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/myXapp/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 /
```

http://www.rejoiceblog.com/

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`.

http://www.rejoiceblog.com/

- `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:

http://www.rejoiceblog.com/

```
byte[] bytes = Files.readAllBytes(path);
```

If you want to read the file as a string, call `readAllBytes` followed by

```
var content = new String(bytes, 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), StandardOpenOpti
```

You can also write a collection of lines with

```
Files.write(path, lines);
```

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);
```

http://www.rejoiceblog.com/

```
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 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, 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

http://www.rejoiceblog.com/

```
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

http://www.rejoiceblog.com/

- `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

http://www.rejoiceblog.com/
COPY_ATTRIBUTES option. You can supply both like this:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXIST

   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.

http://www.rejoiceblog.com/

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|SYNC`	Requires that each update to the file data|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

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)`

http://www.rejoiceblog.com/

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.

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

http://www.rejoiceblog.com/

• 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, B
```

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, P
```

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)`

http://www.rejoiceblog.com/

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))
{
```

http://www.rejoiceblog.com/

```
    . . .

}
```

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
```

http://www.rejoiceblog.com/

```
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.

http://www.rejoiceblog.com/

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 first delete the children before deleting the parent. The next section shows you how to overcome that problem.

http://www.rejoiceblog.com/

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
{
   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
```

Table 2.4 shows all glob patterns.

Table 2.4 Glob Patterns

http://www.rejoiceblog.com/

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 Java files (not counting the extension)
[. . .]	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)`

http://www.rejoiceblog.com/

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 but the `visitFileFailed` method do nothing and continue. The `visitFileFailed` method throws the exception that caused the failure, thereby terminating the visit.

For example, here is how you can print out all subdirectories of a given directory:

```
Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path

  {

     public FileVisitResult preVisitDirectory(Path path, Basi

        throws IOException
```

http://www.rejoiceblog.com/

```
        {
            System.out.println(path);
            return FileVisitResult.CONTINUE;
        }
        public FileVisitResult postVisitDirectory(Path dir, IOEx
        {
            return FileVisitResult.CONTINUE;
        }
        public FileVisitResult visitFileFailed(Path path, IOExce
        {
            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 that it's not allowed to access.

Also note that the attributes of the path are passed as a parameter to the `preVisitDirectory` and `visitFile` method. 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.

http://www.rejoiceblog.com/

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, BasicFileAtt

     {

        Files.delete(file);

        return FileVisitResult.CONTINUE;

     }

     public FileVisitResult postVisitDirectory(Path dir, IOEx

     {

        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,`

http://www.rejoiceblog.com/

`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, BasicFileAttributes attrs)`

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*.

http://www.rejoiceblog.com/

If `zipname` is the name of a ZIP file, then the call

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname),
```

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),

Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Pat

  {

     public FileVisitResult visitFile(Path file, BasicFileAtt

     {

      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.

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

```
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[], int offset, int length)
```

Finally, there are methods

```
getInt

getLong

getShort
```

http://www.rejoiceblog.com/

```
getChar

getFloat

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

putLong

putShort

putChar

putFloat
```

http://www.rejoiceblog.com/

```
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();
```

Note

For a nice explanation of the CRC algorithm, see `www.relisoft.com/Science/CrcMath.html`.

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`

http://www.rejoiceblog.com/

```
 1  package memoryMap;
 2
 3  import java.io.*;
 4  import java.nio.*;
 5  import java.nio.channels.*;
 6  import java.nio.file.*;
 7  import java.util.zip.*;
 8
 9  /**
10   * This program computes the CRC checksum of a file in fo
11   * Usage: java memoryMap.MemoryMapTest filename
12   * @version 1.02 2018-05-01
13   * @author Cay Horstmann
14   */
15  public class MemoryMapTest
16  {
17     public static long checksumInputStream(Path filename)
18     {
19        try (InputStream in = Files.newInputStream(filename
20        {
```

http://www.rejoiceblog.com/

```
            var crc = new CRC32();

            int c;
            while ((c = in.read()) != -1)
                crc.update(c);
            return crc.getValue();
        }
    }

    public static long checksumBufferedInputStream(Path fi
    {
        try (var in = new BufferedInputStream(Files.newInpu
        {
            var crc = new CRC32();

            int c;
            while ((c = in.read()) != -1)
                crc.update(c);
            return crc.getValue();
```

http://www.rejoiceblog.com/

```
        }
    }

    public static long checksumRandomAccessFile(Path filen
    {
        try (var file = new RandomAccessFile(filename.toFil
        {
            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();
    }
}

```

http://www.rejoiceblog.com/

```
public static long checksumMappedFile(Path filename) thr
{
    try (FileChannel channel = FileChannel.open(filename)
    {
        var crc = new CRC32();
        int length = (int) channel.size();
        MappedByteBuffer buffer = channel.map(FileChannel.

        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 IOExceptio
{
```

http://www.rejoiceblog.com/

```
        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));
```

http://www.rejoiceblog.com/

```
        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

http://www.rejoiceblog.com/

- `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.

http://www.rejoiceblog.com/

- `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`.

http://www.rejoiceblog.com/

- `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.

- `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)`

http://www.rejoiceblog.com/

- `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

http://www.rejoiceblog.com/

Figure 2.9 A buffer

These values fulfill the condition

0 = mark = position = limit = 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);
```

http://www.rejoiceblog.com/

```
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)`

http://www.rejoiceblog.com/

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)
```

http://www.rejoiceblog.com/

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.

http://www.rejoiceblog.com/

• 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.

http://www.rejoiceblog.com/

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, you will see the regular expression syntax used by the Java API, and 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 `]`,

http://www.rejoiceblog.com/

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`

http://www.rejoiceblog.com/

.	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

http://www.rejoiceblog.com/

`\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`
`(?`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

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/

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

http://www.rejoiceblog.com/
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:

http://www.rejoiceblog.com/

```
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

http://www.rejoiceblog.com/

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
```

http://www.rejoiceblog.com/

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
 * or hit Cancel to exit. If the pattern contains groups,
```

http://www.rejoiceblog.com/

```
 * in the match.
 * @version 1.03 2018-05-01
 * @author Cay Horstmann
 */
public class RegexTest
{
   public static void main(String[] args) throws PatternS
   {
      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);
```

http://www.rejoiceblog.com/

```
        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 == matc
                            System.out.print("()");
                    // Print ( for non-empty groups starting
                    for (int j = 1; j <= g; j++)
                        if (i == matcher.start(j) && i != matc
                            System.out.print('(');
                    System.out.print(input.charAt(i));
                    // Print ) for non-empty groups ending he
                    for (int j = 1; j <= g; j++)
```

http://www.rejoiceblog.com/

```
                    if (i + 1 != matcher.start(j) && i + 1
                        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();
```

http://www.rejoiceblog.com/

```
    String match = input.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, then 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
```

http://www.rejoiceblog.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 matchi
 * <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)
   {
```

http://www.rejoiceblog.com/

```
        try
        {
            // get URL string from command line or use defau
            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(), Standa

            // search for all occurrences of pattern
            var patternString = "<a\\s+href\\s*=\\s*(\"[^\"]
            Pattern pattern = Pattern.compile(patternString,
            pattern.matcher(input)
                .results()
                .map(MatchResult::group)
                .forEach(System.out::println);
        }
```

http://www.rejoiceblog.com/

```
        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*");
```

http://www.rejoiceblog.com/

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, then 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")
```

http://www.rejoiceblog.com/

```
      .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.

- `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, where the pattern specifies 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()`

http://www.rejoiceblog.com/

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.

- `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

http://www.rejoiceblog.com/

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 grop

- `int start()`

- `int end()`

- `int start(int group)`

- `int end(int group)`

yield the start and end offsets of the matched string or the string matched by the given group.

http://www.rejoiceblog.com/

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.

http://www.rejoiceblog.com/

Chapter 3 XML

https://avxhm.se/blogs/hill0

In this chapter

- 3.1 Introducing XML
- 3.2 The Structure of an XML Document
- 3.3 Parsing an XML Document
- 3.4 Validating XML Documents
- 3.5 Locating Information with XPath
- 3.6 Using Namespaces
- 3.7 Streaming Parsers
- 3.8 Generating XML Documents
- 3.9 XSL Transformations

The preface of the book *Essential XML* by Don Box et al. (Addison-Wesley, 2000) states only half-jokingly: “The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry’s solution to world hunger.” Indeed, as you will see in this chapter, XML is a very useful technology for describing structured information. XML tools make it easy to process and transform that information. However, XML is not a silver bullet. You need domain-specific standards and code libraries to use it effectively. Moreover, far from making Java obsolete, XML works very well with Java. Since the late 1990s, IBM, Apache, and others have been instrumental in producing high-quality Java libraries for XML processing. Many of these libraries have now been integrated into the Java platform.

This chapter introduces XML and covers the XML features of the Java library. As always, we’ll point out along the way when the hype surrounding XML is justified—and

http://www.rejoiceblog.com/
when you have to take it with a grain of salt and try solving your problems the old-fashioned way: through good design and code.

3.1 INTRODUCING XML

In Chapter 13 of Volume I, you have seen the use of *property files* to describe the configuration of a program. A property file contains a set of name/value pairs, such as

```
fontname=Times Roman

fontsize=12

windowsize=400 200

color=0 50 100
```

You can use the `Properties` class to read in such a file with a single method call. That's a nice feature, but it doesn't really go far enough. In many cases, the information you want to describe has more structure than the property file format can comfortably handle. Consider the `fontname/fontsize` entries in the example. It would be more object-oriented to have a single entry:

```
font=Times Roman 12
```

But then, parsing the font description gets ugly as you have to figure out when the font name ends and the font size starts.

Property files have a single flat hierarchy. You can often see programmers work around that limitation with key names like

```
title.fontname=Helvetica

title.fontsize=36

body.fontname=Times Roman

body.fontsize=12
```

http://www.rejoiceblog.com/

Another shortcoming of the property file format is the requirement that keys must be unique. To store a sequence of values, you need another workaround, such as

```
menu.item.1=Times Roman

menu.item.2=Helvetica

menu.item.3=Goudy Old Style
```

The XML format solves these problems. It can express hierarchical structures and is thus more flexible than the flat table structure of a property file.

An XML file for describing a program configuration might look like this:

```
<config>

   <entry id="title">

      <font>

         <name>Helvetica</name>

         <size>36</size>

      </font>

   </entry>

   <entry id="body">

      <font>

         <name>Times Roman</name>

         <size>12</size>

   </font>
```

http://www.rejoiceblog.com/

```
    </entry>

    <entry id="background">

        <color>

            <red>0</red>

            <green>50</green>

        <blue>100</blue>

        </color>

    </entry>

</config>
```

The XML format allows you to express the hierarchy and record repeated elements without contortions.

The format of an XML file is straightforward. It looks similar to an HTML file. There is a good reason for that—both the XML and HTML formats are descendants of the venerable Standard Generalized Markup Language (SGML).

SGML has been around since the 1970s for describing the structure of complex documents. It has been used with success in some industries that require ongoing maintenance of massive documentation—in particular, the aircraft industry. However, SGML is quite complex, so it has never caught on in a big way. Much of that complexity arises because SGML has two conflicting goals. SGML wants to make sure that documents are formed according to the rules for their document type, but it also wants to make data entry easy by allowing shortcuts that reduce typing. XML was designed as a simplified version of SGML for use on the Internet. As is often true, simpler is better, and XML has enjoyed the immediate and enthusiastic reception that has eluded SGML for so long.

http://www.rejoiceblog.com/

Note

You can find a very nice version of the XML standard, with annotations by Tim Bray, at `www.xml.com/axml/axml.html`.

Even though XML and HTML have common roots, there are important differences between the two.

• Unlike HTML, XML is case-sensitive. For example, `<H1>` and `<h1>` are different XML tags.

• In HTML, you can omit end tags, such as `</p>` or `</li>`, if it is clear from the context where a paragraph or list item ends. In XML, you can never omit an end tag.

• In XML, elements that have a single tag without a matching end tag must end in a `/`, as in `<img src="coffeecup.png"/>`. That way, the parser knows not to look for a `</img>` tag.

• In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional. For example, `<applet code="MyApplet.class" width=300 height=300>` is legal HTML but not legal XML. In XML, you have to use quotation marks: `width="300"`.

• In HTML, you can have attribute names without values, such as `<input type="radio" name="language" value="Java" checked>`. In XML, all attributes must have values, such as `checked="true"` or (ugh) `checked="checked"`.

3.2 THE STRUCTURE OF AN XML DOCUMENT

An XML document should start with a header such as

```
<?xml version="1.0"?>
```

or

http://www.rejoiceblog.com/

```
<?xml version="1.0" encoding="UTF-8"?>
```

Strictly speaking, a header is optional, but it is highly recommended.

> **Note**
>
> Since SGML was created for processing of real documents, XML files are called *documents* even though many of them describe data sets that one would not normally call documents.

The header can be followed by a *document type definition* (DTD), such as

```
<!DOCTYPE web-app PUBLIC

   "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"

   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

DTDs are an important mechanism to ensure the correctness of a document, but they are not required. We will discuss them later in this chapter.

Finally, the body of the XML document contains the *root element*, which can contain other elements. For example,

```
<?xml version="1.0"?>

<!DOCTYPE config . . .>

<config>

   <entry id="title">

      <font>
```

http://www.rejoiceblog.com/

```
        <name>Helvetica</name>

        <size>36</size>

      </font>

    </entry>

    . . .

  </config>
```

An element can contain *child elements*, text, or both. In the preceding example, the `font` element has two child elements, `name` and `size`. The `name` element contains the text `"Helvetica"`.

Tip

It is best to structure your XML documents so that an element contains *either* child elements *or* text. In other words, you should avoid situations such as

```
<font>

  Helvetica

  <size>36</size>

</font>
```

This is called *mixed content* in the XML specification. As you will see later in this chapter, you can simplify parsing if you avoid mixed content.

XML elements can contain attributes, such as

http://www.rejoiceblog.com/

```
<size unit="pt">36</size>
```

There is some disagreement among XML designers about when to use elements and when to use attributes. For example, it would seem easier to describe a font as

```
<font name="Helvetica" size="36"/>
```

than

```
<font>

   <name>Helvetica</name>

   <size>36</size>

</font>
```

However, attributes are much less flexible. Suppose you want to add units to the size value. If you use attributes, you will have to add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

Ugh! Now you have to parse the string `"36 pt"`, just the kind of hassle that XML was designed to avoid. Adding an attribute to the `size` element is much cleaner:

```
<font>

   <name>Helvetica</name>

   <size unit="pt">36</size>

</font>
```

A commonly used rule of thumb is that attributes should be used only to modify the

http://www.rejoiceblog.com/

interpretation of a value, not to specify values. If you find yourself engaged in metaphysical discussions about whether a particular setting is a modification of the interpretation of a value or not, just say “no” to attributes and use elements throughout. Many useful XML documents don’t use attributes at all.

Note

In HTML, the rule for attribute usage is simple: If it isn’t displayed on the web page, it’s an attribute. For example, consider the hyperlink

```
<a href="http://java.sun.com">Java Technology</a>
```

The string `Java Technology` is displayed on the web page, but the URL of the link is not a part of the displayed page. However, the rule isn’t all that helpful for most XML files because the data in an XML file aren’t normally meant to be viewed by humans.

Elements and text are the “bread and butter” of XML documents. Here are a few other markup instructions that you might encounter:

• *Character references* have the form `&#`*decimalValue*`;` or `&#x`*hexValue*`;`. For example, the é character can be denoted with either of the following:

```
&#233; &#xE9;
```

• *Entity references* have the form `&`*name*`;`. The entity references

```
&lt; &gt; &amp; &quot; &apos;
```

have predefined meanings: the less-than, greater-than, ampersand, quotation mark, and apostrophe characters. You can define other entity references in a DTD.

• *CDATA sections* are delimited by `<![CDATA[` and `]]>`. They are a special form of character data. You can use them to include strings that contain characters such as `<` `>`

http://www.rejoiceblog.com/

`&` without having them interpreted as markup, for example:

```
<![CDATA[< & > are my favorite delimiters]]>
```

CDATA sections cannot contain the string `]]>`. Use this feature with caution! It is too often used as a back door for smuggling legacy data into XML documents.

• *Processing instructions* are instructions for applications that process XML documents. They are delimited by `<?` and `?>`, for example

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Every XML document starts with a processing instruction

```
<?xml version="1.0"?>
```

• *Comments* are delimited by `<!--` and `-->`, for example

```
<!-- This is a comment. -->
```

Comments should not contain the string `--`. Comments should only be information for human readers. They should never contain hidden commands; use processing instructions for commands.

3.3 PARSING AN XML DOCUMENT

To process an XML document, you need to *parse* it. A parser is a program that reads a file, confirms that the file has the correct format, breaks it up into the constituent elements, and lets a programmer access those elements. The Java library supplies two kinds of XML parsers:

• Tree parsers, such as the Document Object Model (DOM) parser, that read an XML document into a tree structure.

• Streaming parsers, such as the Simple API for XML (SAX) parser, that generate events as they read an XML document.

http://www.rejoiceblog.com/

The DOM parser is easier to use for most purposes, and we explain it first. You may consider a streaming parser if you process very long documents whose tree structures would use up a lot of memory, or if you are only interested in a few elements and don't care about their context. For more information, see Section 3.7, "Streaming Parsers," on p. 195.

The DOM parser interface is standardized by the World Wide Web Consortium (W3C). The `org.w3c.dom` package contains the definitions of interface types such as `Document` and `Element`. Different suppliers, such as the Apache Organization and IBM, have written DOM parsers whose classes implement these interfaces. The Java API for XML Processing (JAXP) library actually makes it possible to

plug in any of these parsers. But the JDK also comes with a DOM parser that is derived from the Apache parser.

To read an XML document, you need a `DocumentBuilder` object that you get from a `DocumentBuilderFactory` like this:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newIn

DocumentBuilder builder = factory.newDocumentBuilder();
```

You can now read a document from a file:

```
File f = . . .

Document doc = builder.parse(f);
```

Alternatively, you can use a URL:

```
URL u = . . .

Document doc = builder.parse(u);
```

You can even specify an arbitrary input stream:

http://www.rejoiceblog.com/

```
InputStream in = . . .

Document doc = builder.parse(in);
```

Note

If you use an input stream as an input source, the parser will not be able to locate other files that are referenced relative to the location of the document, such as a DTD in the same directory. You can install an "entity resolver" to overcome that problem. See `www.xml.com/pub/a/2004/03/03/catalogs.html` or `www.ibm.com/developerworks/xml/library/x-mxd3.html` for more information.

The `Document` object is an in-memory representation of the tree structure of the XML document. It is composed of objects whose classes implement the `Node` interface and its various subinterfaces. Figure 3.1 shows the inheritance hierarchy of the subinterfaces.

Figure 3.1 The `Node` interface and its subinterfaces

http://www.rejoiceblog.com/

Start analyzing the contents of a document by calling the `getDocumentElement` method. It returns the root element.

```
Element root = doc.getDocumentElement();
```

For example, if you are processing a document

```
<?xml version="1.0"?>

<font>

   . . .

</font>
```

then calling `getDocumentElement` returns the `font` element.

The `getTagName` method returns the tag name of an element. In the preceding example, `root.getTagName()` returns the string `"font"`.

To get the element's children (which may be subelements, text, comments, or other nodes), use the `getChildNodes` method. That method returns a collection of type `NodeList`. That type was standardized before the standard Java collections, so it has a different access protocol. The `item` method gets the item with a given index, and the `getLength` method gives the total count of the items. Therefore, you can enumerate all children like this:

```
NodeList children = root.getChildNodes();

for (int i = 0; i < children.getLength(); i++)

{
   Node child = children.item(i);

   . . .
}
```

http://www.rejoiceblog.com/

Be careful when analyzing children. Suppose, for example, that you are processing the document

```
<font>

   <name>Helvetica</name>

   <size>36</size>

</font>
```

You would expect the `font` element to have two children, but the parser reports five:

- The whitespace between `<font>` and `<name>`
- The `name` element
- The whitespace between `</name>` and `<size>`
- The `size` element
- The whitespace between `</size>` and `</font>`

Figure 3.2 shows the DOM tree.

http://www.rejoiceblog.com/

Figure 3.2 A simple DOM tree

If you expect only subelements, you can ignore the whitespace:

```
for (int i = 0; i < children.getLength(); i++)
{
   Node child = children.item(i);
   if (child instanceof Element)
   {
     var childElement = (Element) child;
     . . .
   }
}
```

http://www.rejoiceblog.com/

Now you look at only two elements, with tag names `name` and `size`.

As you will see in the next section, you can do even better if your document has a DTD. Then the parser knows which elements don't have text nodes as children, and it can suppress the whitespace for you.

When analyzing the `name` and `size` elements, you want to retrieve the text strings that they contain. Those text strings are themselves contained in child nodes of type `Text`. You know that these `Text` nodes are the only children, so you can use the `getFirstChild` method without having to traverse another `NodeList`. Then, use the `getData` method to retrieve the string stored in the `Text` node.

```
for (int i = 0; i < children.getLength(); i++)
{
   Node child = children.item(i);
   if (child instanceof Element)
   {
     var childElement = (Element) child;
     var textNode = (Text) childElement.getFirstChild();
     String text = textNode.getData().trim();
     if (childElement.getTagName().equals("name"))
        name = text;
     else if (childElement.getTagName().equals("size"))
        size = Integer.parseInt(text);
   }
}
```

http://www.rejoiceblog.com/

Tip

It is a good idea to call `trim` on the return value of the `getData` method. If the author of an XML file puts the beginning and the ending tags on separate lines, such as

```
<size>
  36
</size>
```

then the parser will include all line breaks and spaces in the text node data. Calling the `trim` method removes the whitespace surrounding the actual data.

You can also get the last child with the `getLastChild` method, and the next sibling of a node with `getNextSibling`. Therefore, another way of traversing a node's children is

```
for (Node childNode = element.getFirstChild();
      childNode != null;
      childNode = childNode.getNextSibling())
{
   . . .
}
```

To enumerate the attributes of a node, call the `getAttributes` method. It returns a `NamedNodeMap` object that contains `Node` objects describing the attributes. You can

http://www.rejoiceblog.com/

traverse the nodes in a `NamedNodeMap` in the same way as a `NodeList`. Then, call the `getNodeName` and `getNodeValue` methods to get the attribute names and values.

```
NamedNodeMap attributes = element.getAttributes();

for (int i = 0; i < attributes.getLength(); i++)

{

   Node attribute = attributes.item(i);

   String name = attribute.getNodeName();

   String value = attribute.getNodeValue();

   . . .

}
```

Alternatively, if you know the name of an attribute, you can retrieve the corresponding value directly:

```
String unit = element.getAttribute("unit");
```

You have now seen how to analyze a DOM tree. The program in Listing 3.1 puts these techniques to work by converting an XML document to JSON format.

The tree display clearly shows how child elements are surrounded by text containing whitespace and comments. You can clearly see the newline and return characters as `\n`.

You don't have to be familiar with JSON to understand how the program works with the DOM tree. Simply observe the following:

• We use a `DocumentBuilder` to read a `Document` from a file.

• For each element, we print the tag name, attributes, and elements.

• For character data, we produce a string with the data. If the data comes from a

http://www.rejoiceblog.com/

comment, we add a "`Comment: `" prefix.

Listing 3.1 `dom/JSONConverter.java`

```
package dom;

import java.io.*;
import java.util.*;

import javax.xml.parsers.*;

import org.w3c.dom.*;
import org.w3c.dom.CharacterData;
import org.xml.sax.*;

/**
 * This program displays an XML document as a tree in JS
 * @version 1.2 2018-04-02
 * @author Cay Horstmann
 */
public class JSONConverter
{
```

http://www.rejoiceblog.com/

```
19        public static void main(String[] args) throws SAXEx
20              ParserConfigurationException
21        {
22         String filename;
23         if (args.length == 0)
24         {
25            try (var in = new Scanner(System.in))
26            {
27               System.out.print("Input file: ");
28               filename = in.nextLine();
29            }
30         }
31         else
32            filename = args[0];
33         DocumentBuilderFactory factory = DocumentBuilderFa
34         DocumentBuilder builder = factory.newDocumentBuild
35
36         Document doc = builder.parse(filename);
37         Element root = doc.getDocumentElement();
```

http://www.rejoiceblog.com/

```
        System.out.println(convert(root, 0));
    }

    public static StringBuilder convert(Node node, int lev
    {
        if (node instanceof Element)
        {
            return elementObject((Element) node, level);
        }
        else if (node instanceof CharacterData)
        {
            return characterString((CharacterData) node, lev
        }
        else
        {
            return pad(new StringBuilder(), level).append(
                    jsonEscape(node.getClass().getName()));
        }
    }

```

http://www.rejoiceblog.com/

```
    private static Map<Character, String> replacements = M
          '\n', "\\n", '\r', "\\r", '\t', "\\t", '"', "\\\

    private static StringBuilder jsonEscape(String str)
    {
       var result = new StringBuilder("\"");
       for (int i = 0; i < str.length(); i++)
       {
          char ch = str.charAt(i);
          String replacement = replacements.get(ch);
          if (replacement == null) result.append(ch);
          else result.append(replacement);
         }
         result.append("\"");
         return result;
     }

     private static StringBuilder characterString(Characte
     {
```

http://www.rejoiceblog.com/

```
        var result = new StringBuilder();
        StringBuilder data = jsonEscape(node.getData());
        if (node instanceof Comment) data.insert(1, "Comme
        pad(result, level).append(data);
        return result;
    }

    private static StringBuilder elementObject(Element el
    {
        var result = new StringBuilder();
        pad(result, level).append("{\n");
        pad(result, level + 1).append("\"name\": ");
        result.append(jsonEscape(elem.getTagName()));
        NamedNodeMap attrs = elem.getAttributes();
        if (attrs.getLength() > 0)
        {
            pad(result.append(",\n"), level + 1).append("\"
            result.append(attributeObject(attrs));
        }
        NodeList children = elem.getChildNodes();
```

http://www.rejoiceblog.com/

```
        if (children.getLength() > 0)
        {
            pad(result.append(",\n"), level + 1).append("\"
            for (int i = 0; i < children.getLength(); i++)
            {
                if (i > 0) result.append(",\n");
                result.append(convert(children.item(i), leve
            }
            result.append("\n");
            pad(result, level + 1).append("]\n");
        }
        pad(result, level).append("}");
        return result;
    }

    private static StringBuilder pad(StringBuilder builde
    {
        for (int i = 0; i < level; i++) builder.append(" "
        return builder;
```

http://www.rejoiceblog.com/

```
    }

    private static StringBuilder attributeObject(NamedN
    {
        var result = new StringBuilder("{");
        for (int i = 0; i < attrs.getLength(); i++)
        {
            if (i > 0) result.append(", ");
            result.append(jsonEscape(attrs.item(i).getNod
            result.append(": ");
            result.append(jsonEscape(attrs.item(i).getNod
        }
        result.append("}");
        return result;
    }
}
```

javax.xml.parsers.DocumentBuilderFactory 1.4

- `static DocumentBuilderFactory newInstance()`

returns an instance of the `DocumentBuilderFactory` class.

http://www.rejoiceblog.com/

- `DocumentBuilder newDocumentBuilder()`

returns an instance of the `DocumentBuilder` class.

javax.xml.parsers.DocumentBuilder 1.4

- `Document parse(File f)`
- `Document parse(String url)`
- `Document parse(InputStream in)`

parses an XML document from the given file, URL, or input stream and returns the parsed document.

org.w3c.dom.Document 1.4

- `Element getDocumentElement()`

returns the root element of the document.

org.w3c.dom.Element 1.4

- `String getTagName()`

returns the name of the element.

- `String getAttribute(String name)`

returns the value of the attribute with the given name, or the empty string if there is no such attribute.

org.w3c.dom.Node 1.4

- `NodeList getChildNodes()`

http://www.rejoiceblog.com/

returns a node list that contains all children of this node.

- `Node getFirstChild()`

- `Node getLastChild()`

gets the first or last child node of this node, or `null` if this node has no children.

- `Node getNextSibling()`

- `Node getPreviousSibling()`

gets the next or previous sibling of this node, or `null` if this node has no siblings.

- `Node getParentNode()`

gets the parent of this node, or `null` if this node is the document node.

- `NamedNodeMap getAttributes()`

returns a node map that contains `Attr` nodes that describe all attributes of this node.

- `String getNodeName()`

returns the name of this node. If the node is an `Attr` node, the name is the attribute name.

- `String getNodeValue()`

returns the value of this node. If the node is an `Attr` node, the value is the attribute value.

org.w3c.dom.CharacterData 1.4

- `String getData()`

returns the text stored in this node.

org.w3c.dom.NodeList 1.4

http://www.rejoiceblog.com/

- `int getLength()`

returns the number of nodes in this list.

- `Node item(int index)`

returns the node with the given index. The index is between `0` and `getLength() - 1`.

org.w3c.dom.NamedNodeMap 1.4

- `int getLength()`

returns the number of nodes in this map.

- `Node item(int index)`

returns the node with the given index. The index is between `0` and `getLength() - 1`.

3.4 VALIDATING XML DOCUMENTS

In the previous section, you saw how to traverse the tree structure of a DOM document. However, if you simply follow that approach, you'll have to do quite a bit of tedious programming and error checking. Not only will you have to deal with whitespace between elements, but you will also need to check whether the document contains the nodes that you expect. For example, suppose you are reading an element:

```
<font>
    <name>Helvetica</name>
   <size>36</size>
</font>
```

You get the first child. Oops . . . it is a text node containing whitespace `"\n "`. You skip

http://www.rejoiceblog.com/

text nodes and find the first element node. Then, you need to check that its tag name is `"name"` and that it has one child node of type `Text`. You move on to the next nonwhitespace child and make the same check. What if the author of the document switched the order of the children or added another child element? It is tedious to code all this error checking—but reckless to skip the checks.

Fortunately, one of the major benefits of an XML parser is that it can automatically verify that a document has the correct structure. Then, parsing becomes much simpler. For example, if you know that the `font` fragment has passed validation, you can simply get the two grandchildren, cast them as `Text` nodes, and get the text data, without any further checking.

To specify the document structure, you can supply a DTD or an XML Schema definition. A DTD or schema contains rules that explain how a document should be formed, by specifying the legal child elements and attributes for each element. For example, a DTD might contain a rule:

```
<!ELEMENT font (name,size)>
```

This rule expresses that a `font` element must always have two children, which are `name` and `size` elements. The XML Schema language expresses the same constraint as

```
<xsd:element name="font">
   <xsd:sequence>
     <xsd:element name="name" type="xsd:string"/>
     <xsd:element name="size" type="xsd:int"/>
   </xsd:sequence>
</xsd:element>
```

XML Schema can express more sophisticated validation conditions (such as the fact that the `size` element must contain an integer) than can DTDs. Unlike the DTD syntax, the XML Schema syntax itself uses XML, which is a benefit if you need to process

http://www.rejoiceblog.com/

schema files.

In the next section, we will discuss DTDs in detail, then briefly cover the basics of XML Schema support. Finally, we will present a complete application that demonstrates how validation simplifies XML programming.

3.4.1 Document Type Definitions

There are several methods for supplying a DTD. You can include a DTD in an XML document like this:

```
<?xml version="1.0"?>
<!DOCTYPE config [
   <!ELEMENT config . . .>
   more rules
   . . .
]>
<config>
   . . .
</config>
```

As you can see, the rules are included inside a `DOCTYPE` declaration, in a block delimited by `[. . .]`. The document type must match the name of the root element, such as `config` in our example.

Supplying a DTD inside an XML document is somewhat uncommon because DTDs can grow lengthy. It makes more sense to store the DTD externally. The `SYSTEM` declaration can be used for that purpose. Specify a URL that contains the DTD, for example:

```
<!DOCTYPE config SYSTEM "config.dtd">
```

or

```
<!DOCTYPE config SYSTEM "http://myserver.com/config.dtd">
```

http://www.rejoiceblog.com/

Caution

If you use a relative URL for the DTD (such as `"config.dtd"`), give the parser a `File` or `URL` object, not an `InputStream`. If you must parse from an input stream, supply an entity resolver (see the following note).

The mechanism for identifying well-known DTDs has its origin in SGML. Here is an example:

```
<!DOCTYPE web-app

   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2

   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

If an XML processor knows how to locate the DTD with the public identifier, it need not go to the URL.

Note

The system identifier URL of a DTD may not actually be working, or be purposefully slowed down. An example for the latter is the system identifier of the XHTML 1.0 Strict DTD, `http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd`. If you parse an XHTML file, it may take a minute or two for the DTD to be served.

The remedy is to use an *entity resolver* that maps public identifiers to local files. Prior to Java 9, you had to provide an object of a class that implements the `EntityResolver` interface and provides a `resolveEntity` method.

Nowadays, you can use *XML catalogs* to manage the mapping. Provide one or more *catalog files* of the form

```
<?xml version="1.0"?>
```

http://www.rejoiceblog.com/

```
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.0//EN"

    "http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd"

<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog" prefer="pul

  <public publicId=". . ." uri=". . ."/>

 . . .

</catalog>
```

Then construct and install a resolver like this:

```
builder.setEntityResolver(CatalogManager.catalogResolver(

  CatalogFeatures.defaults(),

  Paths.get("catalog.xml").toAbsolutePath().toUri()));
```

See Listing 3.6 for a complete example.

Instead of setting the catalog file locations in your program, you can specify them on the command line with the `javax.xml.catalog.files` system property. Provide absolute `file` URLs separated by semicolons.

Now that you have seen how the parser locates the DTD, let us consider the various kinds of rules.

The `ELEMENT` rule specifies what children an element can have. Use a regular expression, made up of the components shown in Table 3.1.

Table 3.1 Rules for Element Content

http://www.rejoiceblog.com/

Rule	Meaning
E*	0 or more occurrences of E
E+	1 or more occurrences of E
E?	0 or 1 occurrences of E
$E_1 \mid E_2 \mid \ldots \mid E_n$	One of $E_1, E_2, \ldots, E_n$
$E_1, E_2, \ldots, E_n$	E_1 followed by $E_2, \ldots, E_n$
#PCDATA	Text
(#PCDATA$\mid E_1 \mid E_2 \mid \ldots \mid E_n$)*	0 or more occurrences of text and $E_1, E_2, \ldots, E_n$ in any order (mixed content)
ANY	Any children allowed
EMPTY	No children allowed

Here are several simple but typical examples. The following rule states that a `menu` element contains 0 or more `item` elements:

```
<!ELEMENT menu (item)*>
```

This set of rules states that a font is described by a name followed by a size, each of which contain text:

```
<!ELEMENT font (name,size)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT size (#PCDATA)>
```

The abbreviation `PCDATA` denotes *parsed character data*. It is "parsed" because the parser interprets the text string, looking for < characters that denote the start of a new tag, or `&` characters that denote the start of an entity.

An element specification can contain regular expressions that are nested and complex. For example, here is a rule that describes the makeup of a chapter in a book:

```
<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)
```

http://www.rejoiceblog.com/

Each chapter starts with an introduction, which is followed by one or more sections consisting of a heading and one or more paragraphs, images, tables, or notes.

However, in one common case you can't define the rules to be as flexible as you might like. Whenever an element can contain text, there are only two valid cases. Either the element contains nothing but text, such as

```
<!ELEMENT name (#PCDATA)>
```

or the element contains *any combination of text and tags in any order*, such as

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

It is not legal to specify any other types of rules that contain `#PCDATA`. For example, the following is illegal:

```
<!ELEMENT captionedImage (image,#PCDATA)>
```

You have to rewrite such a rule, either by introducing another `caption` element or by allowing any combination of `image` elements and text.

This restriction simplifies the job of the XML parser when parsing *mixed content* (a mixture of tags and text). Since you lose some control by allowing mixed content, it is best to design DTDs so that all elements contain either other elements or nothing but text.

Note

Actually, it isn't quite true that you can specify arbitrary regular expressions of elements in a DTD rule. An XML parser may reject certain complex rule sets that lead to nondeterministic parsing. For example, a regular expression `((x,y)|(x,z))` is nondeterministic. When the parser sees `x`, it doesn't know which of the two alternatives to take. This expression can be rewritten in a deterministic form as `(x,(y|z))`. However, some expressions can't be

http://www.rejoiceblog.com/

reformulated, such as `((x,y)*|x?)`. The parser in the Java XML library gives no warnings when presented with an ambiguous DTD; it simply picks the first matching alternative when parsing, which causes it to reject some correct inputs. The parser is well within its rights to do so because the XML standard allows a parser to assume that the DTD is unambiguous.

You can also specify rules to describe the legal attributes of elements. The general syntax is

`<!ATTLIST` *element attribute type default*`>`

Table 3.2 shows the legal attribute types, and Table 3.3 shows the syntax for the defaults.

Table 3.2 Attribute Types

Type	Meaning
CDATA	Any character string
$(A_1\|A_2\| \ldots \|A_n)$	One of the string attributes $A_1, A_2, \ldots, A_n$
NMTOKEN, NMTOKENS	One or more name tokens
ID	A unique ID
IDREF, IDREFS	One or more references to a unique ID
ENTITY, ENTITIES	One or more unparsed entities

Table 3.3 Attribute Defaults

Default	Meaning
#REQUIRED	Attribute is required.
#IMPLIED	Attribute is optional.
A	Attribute is optional; the parser reports it to be *A* if it is not specified.
#FIXED *A*	The attribute must either be unspecified or *A*; in either case, the parser reports it to be *A*.

Here are two typical attribute specifications:

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
```

http://www.rejoiceblog.com/

```
<!ATTLIST size unit CDATA #IMPLIED>
```

The first specification describes the `style` attribute of a `font` element. There are four legal attribute values, and the default value is `plain`. The second specification expresses that the `unit` attribute of the `size` element can contain any character data sequence.

> **Note**
>
> We generally recommend the use of elements, not attributes, to describe data. Thus, the font style should be a separate element, such as `<font> <style>plain</style>. . .</font>`. However, attributes have an undeniable advantage for enumerated types because the parser can verify that the values are legal. For example, if the font style is an attribute, the parser checks that it is one of the four allowed values, and supplies a default if no value was given.

The handling of a `CDATA` attribute value is subtly different from the processing of `#PCDATA` that you have seen before, and quite unrelated to the `<![CDATA[. . .]]>` sections. The attribute value is first *normalized*—that is, the parser processes character and entity references (such as `é` or `<`) and replaces whitespace with spaces.

An `NMTOKEN` (or name token) is similar to `CDATA`, but most nonalphanumeric characters and internal whitespace are disallowed, and the parser removes leading and trailing whitespace. `NMTOKENS` is a whitespace-separated list of name tokens.

The `ID` construct is quite useful. An `ID` is a name token that must be unique in the document—the parser checks the uniqueness. You will see an application in the next sample program. An `IDREF` is a reference to an ID that exists in the same document, which the parser also checks. `IDREFS` is a whitespace-separated list of ID references.

An `ENTITY` attribute value refers to an "unparsed external entity." That is a holdover from SGML that is rarely used in practice. The annotated XML specification at `www.xml.com/axml/axml.html` has an example.

A DTD can also define *entities*, or abbreviations that are replaced during parsing. You

http://www.rejoiceblog.com/

can find a good example for the use of entities in the user interface descriptions of the Firefox browser. Those descriptions are formatted in XML and contain entity definitions such as

```
<!ENTITY back.label "Back">
```

Elsewhere, text can contain an entity reference, for example:

```
<menuitem label="&back.label;"/>
```

The parser replaces the entity reference with the replacement string. To internationalize the application, only the string in the entity definition needs to be changed. Other uses of entities are more complex and less common; look at the XML specification for details.

This concludes the introduction to DTDs. Now that you have seen how to use DTDs, you can configure your parser to take advantage of them.

First, tell the document builder factory to turn on validation:

```
factory.setValidating(true);
```

All builders produced by this factory validate their input against a DTD. The most useful benefit of validation is ignoring whitespace in element content. For example, consider the XML fragment

```
<font>

   <name>Helvetica</name>

   <size>36</size>

</font>
```

A nonvalidating parser reports the whitespace between the `font`, `name`, and `size`

http://www.rejoiceblog.com/
elements because it has no way of knowing if the children of `font` are

```
(name,size)

(#PCDATA,name,size)*
```

or perhaps

```
ANY
```

Once the DTD specifies that the children are `(name,size)`, the parser knows that the whitespace between them is not text. Call

```
factory.setIgnoringElementContentWhitespace(true);
```

and the builder will stop reporting the whitespace in text nodes. That means you can now *rely on* the fact that a `font` node has two children. You no longer need to program a tedious loop:

```
for (int i = 0; i < children.getLength(); i++)

{

   Node child = children.item(i);

   if (child instanceof Element)

   {

     var childElement = (Element) child;

     if (childElement.getTagName().equals("name")) . . .

     else if (childElement.getTagName().equals("size")) . . .

   }
```

http://www.rejoiceblog.com/

```
}
```

Instead, you can simply access the first and second child:

```
var nameElement = (Element) children.item(0);

var sizeElement = (Element) children.item(1);
```

That is why DTDs are so useful. You don't overload your program with rule-checking code—the parser has already done that work by the time you get the document.

When the parser reports an error, your application will want to do something about it—log it, show it to the user, or throw an exception to abandon the parsing. Therefore, you should install an error handler whenever you use validation. Supply an object that implements the `ErrorHandler` interface. That interface has three methods:

```
void warning(SAXParseException exception)

void error(SAXParseException exception)

void fatalError(SAXParseException exception)
```

Install the error handler with the `setErrorHandler` method of the `DocumentBuilder` class:

```
builder.setErrorHandler(handler);
```

javax.xml.parsers.DocumentBuilder 1.4

- `void setEntityResolver(EntityResolver resolver)`

sets the resolver to locate entities that are referenced in the XML documents being parsed.

http://www.rejoiceblog.com/

- `void setErrorHandler(ErrorHandler handler)`

sets the handler to report errors and warnings that occur during parsing.

org.xml.sax.EntityResolver 1.4

- `public InputSource resolveEntity(String publicID, String systemID)`

returns an input source that contains the data referenced by the given ID(s), or `null` to indicate that this resolver doesn't know how to resolve the particular name. The `publicID` parameter may be `null` if no public ID was supplied.

org.xml.sax.InputSource 1.4

- `InputSource(InputStream in)`
- `InputSource(Reader in)`
- `InputSource(String systemID)`

constructs an input source from a stream, reader, or system ID (usually a relative or absolute URL).

org.xml.sax.ErrorHandler 1.4

- `void fatalError(SAXParseException exception)`
- `void error(SAXParseException exception)`
- `void warning(SAXParseException exception)`

Override these methods to provide handlers for fatal errors, nonfatal errors, and warnings.

http://www.rejoiceblog.com/

org.xml.sax.SAXParseException 1.4

- `int getLineNumber()`
- `int getColumnNumber()`

return the line and column numbers of the end of the processed input that caused the exception.

javax.xml.catalog.CatalogManager 9

- static CatalogResolver catalogResolver(CatalogFeatures features, URI . . . uris)

produces a resolver that uses the catalog files located at the provided URIs. This class implements `EntityResolver` as well as the resolver classes used by StAX, schema validation, and XSL transforms.

javax.xml.catalog.CatalogFeatures 9

- `static CatalogFeatures defaults()`

yields an instance with default settings.

javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isValidating()`
- `void setValidating(boolean value)`

gets or sets the `validating` property of the factory. If set to `true`, the parsers that this factory generates validate their input.

- `boolean isIgnoringElementContentWhitespace()`
- `void setIgnoringElementContentWhitespace(boolean value)`

http://www.rejoiceblog.com/

gets or sets the `ignoringElementContentWhitespace` property of the factory. If set to `true`, the parsers that this factory generates ignore whitespace between element nodes that don't have mixed content (i.e., a mixture of elements and `#PCDATA`).

3.4.2 XML Schema

XML Schema is quite a bit more complex than the DTD syntax, so we will only cover the basics. For more information, we recommend the tutorial at `www.w3.org/TR/xmlschema-0`.

To reference a Schema file in a document, add attributes to the root element, for example:

```
<?xml version="1.0"?>

<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

     xsi:noNamespaceSchemaLocation="config.xsd">

   . . .

</config>
```

This declaration states that the schema file `config.xsd` should be used to validate the document. If your document uses namespaces, the syntax is a bit more complex—see the XML Schema tutorial for details. (The prefix `xsi` is a *namespace alias*; see Section 3.6, "Using Namespaces," on p. 192 for more information.)

A schema defines a *type* for each element and attribute. A *simple type* is a string, perhaps with restrictions on its contents. Everything else is a *complex* type. An element with simple type can have no attributes and no child elements. Otherwise, it must have a complex type. Conversely, attributes always have a simple type.

Some simple types are built into XML Schema, including

```
xsd:string
```

http://www.rejoiceblog.com/

```
xsd:int

xsd:boolean
```

Note

We use the prefix `xsd:` to denote the XML Schema Definition namespace. Some authors use the prefix `xs:` instead.

You can define your own simple types. For example, here is an enumerated type:

```
<xsd:simpleType name="StyleType">

   <xsd:restriction base="xsd:string">

     <xsd:enumeration value="PLAIN" />

     <xsd:enumeration value="BOLD" />

     <xsd:enumeration value="ITALIC" />

     <xsd:enumeration value="BOLD_ITALIC" />

   </xsd:restriction>

</xsd:simpleType>
```

When you define an element, you specify its type:

```
<xsd:element name="name" type="xsd:string"/>

<xsd:element name="size" type="xsd:int"/>
```

http://www.rejoiceblog.com/

```
<xsd:element name="style" type="StyleType"/>
```

The type constrains the element content. For example, the elements

```
<size>10</size>

<style>PLAIN</style>
```

will validate correctly, but the elements

```
<size>default</size>

<style>SLANTED</style>
```

will be rejected by the parser.

You can compose types into complex types, for example:

```
<xsd:complexType name="FontType">

   <xsd:sequence>

     <xsd:element ref="name"/>

     <xsd:element ref="size"/>

     <xsd:element ref="style"/>

   </xsd:sequence>

</xsd:complexType>
```

A `FontType` is a sequence of `name`, `size`, and `style` elements. In this type definition, we use the `ref` attribute and refer to definitions that are located elsewhere in the schema. You can also nest definitions, like this:

http://www.rejoiceblog.com/

```
<xsd:complexType name="FontType">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="size" type="xsd:int"/>
      <xsd:element name="style">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="PLAIN" />
              <xsd:enumeration value="BOLD" />
              <xsd:enumeration value="ITALIC" />
              <xsd:enumeration value="BOLD_ITALIC" />
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
```

Note the *anonymous type definition* of the `style` element.

The `xsd:sequence` construct is the equivalent of the concatenation notation in DTDs. The `xsd:choice` construct is the equivalent of the `|` operator. For example,

```
<xsd:complexType name="contactinfo">
```

http://www.rejoiceblog.com/

```
    <xsd:choice>

        <xsd:element ref="email"/>

        <xsd:element ref="phone"/>

    </xsd:choice>

  </xsd:complexType>
```

This is the equivalent of the DTD type `email|phone`.

To allow repeated elements, use the `minoccurs` and `maxoccurs` attributes. For example, the equivalent of the DTD type `item*` is

```
  <xsd:element name="item" type=". . ." minoccurs="0" maxoccurs
```

To specify attributes, add `xsd:attribute` elements to `complexType` definitions:

```
  <xsd:element name="size">

    <xsd:complexType>

        . . .

        <xsd:attribute name="unit" type="xsd:string" use="optio

    </xsd:complexType>

  </xsd:element>
```

This is the equivalent of the DTD statement

```
  <!ATTLIST size unit CDATA #IMPLIED "cm">
```

http://www.rejoiceblog.com/

Enclose element and type definitions of your schema inside an `xsd:schema` element:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   . . .

   </xsd:schema>
```

Parsing an XML file with a schema is similar to parsing a file with a DTD, but with three differences:

1. You need to turn on support for namespaces, even if you don't use them in your XML files.

```
factory.setNamespaceAware(true);
```

2. You need to prepare the factory for handling schemas, with the following magic incantation:

```
final String JAXP_SCHEMA_LANGUAGE = "http://java.sun.com/xml/

final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSche

factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

3.4.3 A Practical Example

In this section, we work through a practical example that shows the use of XML in a realistic setting.

Suppose an application needs configuration data that specifies arbitrary objects, not just text strings. We provide two mechanisms for instatiating the object: with a constructor, and with a factory method. Here is how to make a `Color` object using a constructor:

http://www.rejoiceblog.com/

```
<construct class="java.awt.Color">

   <int>55</int>

   <int>200</int>

   <int>100</int>

</construct>
```

Here is an example with a factory method:

```
<factory class="java.util.logging.Logger" method="getLogger">

   <string>com.horstmann.corejava</string>

</factory>
```

If the factory method name is omitted, it defaults to `getInstance`.

As you can see, there are elements for describing strings and integers. We also support the `boolean` type, and other primitive types can be added in the same way.

Just to show off the syntax, there is a second mechanism for primitive types:

```
<value type="int">30</value>
```

A configuration is a sequence of entries. Each entry has an ID and an object:

```
<config>

   <entry id="background">

     <construct class="java.awt.Color">

       <value type="int">55</value>
```

http://www.rejoiceblog.com/

```
        <value type="int">200</value>

        <value type="int">100</value>

      </construct>

    </entry>

    . . .

  </config>
```

The parser checks that IDs are unique.

The DTD, shown in Listing 3.4, is straightforward.

Listing 3.5 contains the equivalent schema. In the schema, we can provide additional checking: an `int` or `boolean` element can only contain integer or boolean content. Note the use of the `xsd:group` construct to define parts of complex types that are used repeatedly.

The program in Listing 3.2 shows how to use TODO A sample configuration is defined in Listing 3.3.

The program uses the schema instead of the DTD if you choose a file that contains the string `-schema`.

This example is a typical use of XML. The XML format is robust enough to express complex relationships. The XML parser adds value by taking over the routine job of validity checking and supplying defaults.

Listing 3.2 `read/XMLReadTest.java`

```
1   package read;

2

3   import java.io.*;
```

http://www.rejoiceblog.com/

```
import java.lang.reflect.*;
import java.util.*;

import javax.xml.parsers.*;

import org.w3c.dom.*;
import org.xml.sax.*;

/**
 * This program shows how to use an XML file to describe
 * @version 1.0 2018-04-03
 * @author Cay Horstmann
 */
public class XMLReadTest
{
      public static void main(String[] args)
            throws ParserConfigurationException, SAXExcep
            ReflectiveOperationException
      {
```

http://www.rejoiceblog.com/

```
23          String filename;
24          if (args.length == 0)
25          {
26             try (var in = new Scanner(System.in))
27             {
28                System.out.print("Input file: ");
29                filename = in.nextLine();
30             }
31          }
32          else
33             filename = args[0];
34
35       DocumentBuilderFactory factory = DocumentBuilderFa
36       factory.setValidating(true);
37
38       if (filename.contains("-schema"))
39       {
40          factory.setNamespaceAware(true);
41          final String JAXP_SCHEMA_LANGUAGE =
42                "http://java.sun.com/xml/jaxp/properties/
```

http://www.rejoiceblog.com/

```
            final String W3C_XML_SCHEMA = "http://www.w3.or
            factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_
        }

        factory.setIgnoringElementContentWhitespace(true);

        DocumentBuilder builder = factory.newDocumentBuild

        builder.setErrorHandler(new ErrorHandler()
            {
                public void warning(SAXParseException e) thr
                {
                    System.err.println("Warning: " + e.getM
                }

                public void error(SAXParseException e) thro
                {
                    System.err.println("Error: " + e.getMess
                    System.exit(0);
```

http://www.rejoiceblog.com/

```
                }

                public void fatalError(SAXParseException e)
                {
                    System.err.println("Fatal error: " + e.g
                    System.exit(0);
                }
            });

            Document doc = builder.parse(filename);
            Map<String, Object> config = parseConfig(doc.get
            System.out.println(config);
        }

        private static Map<String, Object> parseConfig(Elemen
        {
            var result = new HashMap<String, Object>();
            NodeList children = e.getChildNodes();
            for (int i = 0; i < children.getLength(); i++)
            {
```

http://www.rejoiceblog.com/

```
            var child = (Element) children.item(i);
            String name = child.getAttribute("id");
            Object value = parseObject((Element) child.getF
            result.put(name, value);
         }
         return result;
      }

      private static Object parseObject(Element e) throws R
      {
         String tagName = e.getTagName();
         if (tagName.equals("factory")) return parseFactory
         else if (tagName.equals("construct")) return parse
         else
         {
            String childData = ((CharacterData) e.getFirstC
            if (tagName.equals("int"))
               return Integer.valueOf(childData);
            else if (tagName.equals("boolean"))
```

http://www.rejoiceblog.com/

```
            return Boolean.valueOf(childData);
        else
            return childData;
        }
    }

    private static Object parseFactory(Element e) throw
    {
        String className = e.getAttribute("class");
        String methodName = e.getAttribute("method");
        Object[] args = parseArgs(e.getChildNodes());
        Class<?>[] parameterTypes = getParameterTypes(ar
        Method method = Class.forName(className).getMeth
        return method.invoke(null, args);
    }

    private static Object parseConstruct(Element e) thr
    {
        String className = e.getAttribute("class");
        Object[] args = parseArgs(e.getChildNodes());
```

http://www.rejoiceblog.com/

```
121         Class<?>[] parameterTypes = getParameterTypes(ar
122         Constructor<?> constructor = Class.forName(class
123         return constructor.newInstance(args);
124      }
125
126      private static Object[] parseArgs(NodeList elements
127      {
128         var result = new Object[elements.getLength()];
129         for (int i = 0; i < result.length; i++)
130            result[i] = parseObject((Element) elements.it
131         return result;
132      }
133
134      private static Map<Class<?>, Class<?>> toPrimitive
135            Integer.class, int.class,
136            Boolean.class, boolean.class);
137
138      private static Class<?>[] getParameterTypes(Object[
139      {
```

http://www.rejoiceblog.com/

```
        var result = new Class<?>[args.length];
        for (int i = 0; i < result.length; i++)
        {
            Class<?> cl = args[i].getClass();
            result[i] = toPrimitive.get(cl);
            if (result[i] == null) result[i] = cl;
        }
        return result;
    }
}
```

Listing 3.3 `read/config.xml`

```
<?xml version="1.0"?>
<!DOCTYPE config SYSTEM "config.dtd">
<config>
  <entry id="background">
    <construct class="java.awt.Color">
      <int>55</int>
      <int>200</int>
      <int>100</int>
```

http://www.rejoiceblog.com/

```
      </construct>
    </entry>
    <entry id="currency">
      <factory class="java.util.Currency">
        <string>USD</string>
      </factory>
    </entry>
  </config>
```

Listing 3.4 `read/config.xsd`

```
<!ELEMENT config (entry)*>

<!ELEMENT entry (string|int|boolean|construct|factory)>
<!ATTLIST entry id ID #IMPLIED>

<!ELEMENT construct (string|int|boolean|construct|factor
<!ATTLIST construct class CDATA #IMPLIED>

<!ELEMENT factory (string|int|boolean|construct|factory)
<!ATTLIST factory class CDATA #IMPLIED>
```

http://www.rejoiceblog.com/

```
<!ATTLIST factory method CDATA "getInstance">

<!ELEMENT string (#PCDATA)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Listing 3.5 `read/config.xsd`

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   <xsd:element name="config">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="entry" minOccurs="0" maxO
            <xsd:complexType>
               <xsd:group ref="Object"/>
               <xsd:attribute name="id" type="xsd:ID"/>
            </xsd:complexType>
            </xsd:element>
         </xsd:sequence>
      </xsd:complexType>
```

http://www.rejoiceblog.com/

```
13      </xsd:element>

14

15      <xsd:element name="construct">

16         <xsd:complexType>

17            <xsd:group ref="Arguments"/>

18            <xsd:attribute name="class" type="xsd:string"/>

19         </xsd:complexType>

20      </xsd:element>

21

22      <xsd:element name="factory">

23         <xsd:complexType>

24            <xsd:group ref="Arguments"/>

25            <xsd:attribute name="class" type="xsd:string"/>

26            <xsd:attribute name="method" type="xsd:string" 

27         </xsd:complexType>

28      </xsd:element>

29

30      <xsd:group name="Object">

31         <xsd:choice>

32            <xsd:element ref="construct"/>
```

http://www.rejoiceblog.com/

```
            <xsd:element ref="factory"/>
            <xsd:element name="string" type="xsd:string"/>
            <xsd:element name="int" type="xsd:int"/>
            <xsd:element name="boolean" type="xsd:boolean"/
         </xsd:choice>
      </xsd:group>

      <xsd:group name="Arguments">
         <xsd:sequence>
         <xsd:group ref="Object" minOccurs="0" maxOccurs="u
         </xsd:sequence>
      </xsd:group>
   </xsd:schema>
```

3.5 LOCATING INFORMATION WITH XPATH

If you want to locate a specific piece of information in an XML document, it can be a bit of a hassle to navigate the nodes of the DOM tree. The XPath language makes it simple to access tree nodes. For example, suppose you have this XHTML document:

```
<html>
   <head>
      . . .
```

http://www.rejoiceblog.com/

```
        <title>. . .</title>

        . . .

      </database>

      . . .

    </html>
```

You can get the title text by evaluating the XPath expression

```
/html/head/title/text()
```

That's a lot simpler than the plain DOM approach:

1. Get the document root.

2. Get the first child and cast is as an `Element`.

3. Locate the `title` element among its children.

4. Get its first child and cast it as a `CharacterData` node.

5. Get its data.

An XPath can describe *a set of nodes* in an XML document. For example, the XPath

```
/html/body/form
```

describes the set of all `form` elements that are children of the `body` element in an XHTML file. You can select a particular element with the `[]` operator:

```
/html/body/form[1]
```

is the first form. (The index values start at 1.)

http://www.rejoiceblog.com/

Use the `@` operator to get attribute values. The XPath expression

```
/html/body/form[1]/@action
```

describes the `action` attribute of the first table. The XPath expression

```
/html/body/form/@action
```

describes all `action` attribute nodes of all `form` elements that are children of the `body` element.

There are a number of useful XPath functions. For example,

```
count(/html/body/form)
```

returns the number of `form` children of the `body` element. There are many more elaborate XPath expressions; see the specification at `www.w3c.org/TR/xpath` or the online tutorial at `www.zvon.org/xxl/XPathTutorial/General/examples.html`.

To evaluate XPath expressions, first create an `XPath` object from an `XPathFactory`:

```
XPathFactory xpfactory = XPathFactory.newInstance();

path = xpfactory.newXPath();
```

Then, call the `evaluate` method to evaluate XPath expressions:

```
String username = path.evaluate("/html/head/title/text()", do
```

You can use the same `XPath` object to evaluate multiple expressions.

This form of the `evaluate` method returns a string result. It is suitable for retrieving text, such as the text child of the `title` element in the preceding example. If an XPath

http://www.rejoiceblog.com/
expression yields multiple nodes, make a call such as the following:

```
XPathNodes result = path.evaluateExpression("/html/body/form"
```

The `XPathNodes` class is similar to a `NodeList`, but it extends the `Iterable` interface, allowing you to use an enhanced `for` loop.

This method was added in Java 9. In older releases, use the following call instead:

```
var nodes = (NodeList) path.evaluate("/html/body/form", doc,
```

If the result is a single node, use one of the following calls:

```
Node node = path.evaluateExpression("/html/body/form[1]", doc

node = (Node) path.evaluate("/html/body/form[1]", doc, XPathC
```

If the result is a number, use:

```
int count = path.evaluateExpression("count(/html/body/form)",

count = ((Number) path.evaluate("count(/html/body/form)", doc
```

You don't have to start the search at the document root; you can start at any node or node list. For example, if you have a node from a previous evaluation, you can call

```
String result = path.evaluate(expression, node);
```

If you do not know the result of evaluating an XPath expression (perhaps because it comes from a user), then call

http://www.rejoiceblog.com/

```
XPathEvaluationResult<?> result = path.evaluateExpression(exp
```

The expression `result.type()` is one of the constants

```
STRING

NODESET

NODE

NUMBER

BOOLEAN
```

of the `XPathEvaluationResult.XPathResultType` enumeration. Call `result.value()` to get the value.

The program in Listing 3.6 demonstrates evaluation of arbitrary XPath expressions. Load an XML file and type an expression. The result of the expression is displayed.

Listing 3.6 `xpath/XPathTest.java`

```
1   package xpath;

2

3   import java.io.*;

4   import java.nio.file.*;

5   import java.util.*;

6

7   import javax.xml.catalog.*;
```

http://www.rejoiceblog.com/

```
 8   import javax.xml.parsers.*;
 9   import javax.xml.xpath.*;
10
11   import org.w3c.dom.*;
12   import org.xml.sax.*;
13
14   /**
15    * This program evaluates XPath expressions.
16    * @version 1.1 2018-04-06
17    * @author Cay Horstmann
18    */
19   public class XPathTest
20   {
21        public static void main(String[] args)
22              throws ParserConfigurationException, SAXExcep
23        {
24           DocumentBuilderFactory factory = DocumentBuilder
25           DocumentBuilder builder = factory.newDocumentBui
26
27         // Avoid a delay in parsing an XHTML file--see the
```

http://www.rejoiceblog.com/

```
        builder.setEntityResolver(CatalogManager.catalogRe
           CatalogFeatures.defaults(),
           Paths.get("xpath/catalog.xml").toAbsolutePath()

        XPathFactory xpfactory = XPathFactory.newInstance(
        XPath path = xpfactory.newXPath();
        try (var in = new Scanner(System.in))
        {
           String filename;
           if (args.length == 0)
           {
                 System.out.print("Input file: ");
                 filename = in.nextLine();
              }
              else
                 filename = args[0];

              Document doc = builder.parse(filename);
              var done = false;
```

http://www.rejoiceblog.com/

```
47                while (!done)
48                {
49                   System.out.print("XPath expression (empty
50                   String expression = in.nextLine();
51                   if (expression.trim().isEmpty()) done = t
52                   else
53                   {
54                      try
55                      {
56                         XPathEvaluationResult<?> result = 
57                         if (result.type() == XPathEvaluati
58                         {
59                            for (Node n : (XPathNodes) resu
60                               System.out.println(descripti
61                         }
62                         else if (result.type() == XPathEva
63                            System.out.println((Node) resul
64                         else
65                            System.out.println(result.value
66                      }
```

http://www.rejoiceblog.com/

```
                    catch (XPathExpressionException e)
                    {
                        System.out.println(e.getMessage())
                    }
                }
            }
        }
    }

    public static String description(Node n)
    {
        if (n instanceof Element) return "Element " + n.g
        else if (n instanceof Attr) return "Attribute " +
        else return n.toString();
    }
}
```

javax.xml.xpath.XPathFactory 5.0

- `static XPathFactory newInstance()`

http://www.rejoiceblog.com/

returns an `XPathFactory` instance for creating `XPath` objects.

- `XPath newXpath()`

constructs an `XPath` object for evaluating XPath expressions.

javax.xml.xpath.XPath 5.0

- `String evaluate(String expression, Object startingPoint)`

evaluates an expression, beginning at the given starting point. The starting point can be a node or node list. If the result is a node or node set, the returned string consists of the data of all text node children.

- `Object evaluate(String expression, Object startingPoint, QName resultType)`

evaluates an expression, beginning at the given starting point. The starting point can be a node or node list. The `resultType` is one of the constants `STRING`, `NODE`, `NODESET`, `NUMBER`, or `BOOLEAN` in the `XPathConstants` class. The return value is a `String`, `Node`, `NodeList`, `Number`, or `Boolean`.

- `<T> T evaluateExpression(String expression, Object item, Class<T> type)` 9

evaluates the given expression and yields the answer as a value of the given type.

- `XPathEvaluationResult<?> evaluateExpression(String expression, InputSource source)` 9

evaluates the given expression.

javax.xml.xpath.XPathEvaluationResult<T> 9

- `XPathEvaluationResult.XPathResultType type()`

returns one of the enumeration constants `STRING`, `NODESET`, `NODE`, `NUMBER`, `BOOLEAN`.

http://www.rejoiceblog.com/

- `T value()`

returns the result value.

3.6 USING NAMESPACES

The Java language uses packages to avoid name clashes. Programmers can use the same name for different classes as long as they aren't in the same package. XML has a similar *namespace* mechanism for element and attribute names.

A namespace is identified by a Uniform Resource Identifier (URI), such as

```
http://www.w3.org/2001/XMLSchema

uuid:1c759aed-b748-475c-ab68-10679700c4f2

urn:com:books-r-us
```

The HTTP URL form is the most common. Note that the URL is just used as an identifier string, not as a locator for a document. For example, the namespace identifiers

```
http://www.horstmann.com/corejava

http://www.horstmann.com/corejava/index.html
```

denote *different* namespaces, even though a web server would serve the same document for both URLs.

There need not be any document at a namespace URL—the XML parser doesn't attempt to find anything at that location. However, as a help to programmers who encounter a possibly unfamiliar namespace, it is customary to place a document explaining the purpose of the namespace at the URL location. For example, if you point your browser to the namespace URL for the XML Schema namespace (`http://www.w3.org/2001/XMLSchema`), you will find a document describing the XML Schema standard.

http://www.rejoiceblog.com/

Why use HTTP URLs for namespace identifiers? It is easy to ensure that they are unique. If you choose a real URL, the host part's uniqueness is guaranteed by the domain name system. Your organization can then arrange for the uniqueness of the remainder of the URL. This is the same rationale that underlies the use of reversed domain names in Java package names.

Of course, although long namespace identifiers are good for uniqueness, you don't want to deal with long identifiers any more than you have to. In the Java programming language, you use the `import` mechanism to specify the long names of packages, and then use just the short class names. In XML, there is a similar mechanism:

```
<element xmlns="namespaceURI">
  children
</element>
```

The element and its children are now part of the given namespace.

A child can provide its own namespace, for example:

```
<element xmlns="namespaceURI1">
     <child xmlns="namespaceURI2">
   grandchildren
  </child>
  more children
</element>
```

Then the first child and the grandchildren are part of the second namespace.

This simple mechanism works well if you need only a single namespace or if the namespaces are naturally nested. Otherwise, you will want to use a second mechanism that has no analog in Java. You can have a *prefix* for a namespace—a short identifier that you choose for a particular document. Here is a typical example—the `xsd` prefix in an XML Schema file:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsd:element name="config"/>

   . . .
```

http://www.rejoiceblog.com/

```
</xsd:schema>
```

The attribute

`xmlns:`*`prefix`*`="`*`namespaceURI`*`"`

defines a namespace and a prefix. In our example, the prefix is the string `xsd`. Thus, `xsd:schema` really means `schema` in the namespace `http://www.w3.org/2001/XMLSchema`.

Note

Only child elements inherit the namespace of their parent. Attributes without an explicit prefix are never part of a namespace. Consider this contrived example:

```
<configuration xmlns="http://www.horstmann.com/corejava"
  xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
  <size value="210" si:unit="mm"/>
  . . .
</configuration>
```

In this example, the elements `configuration` and `size` are part of the namespace with URI `http://www.horstmann.com/corejava`. The attribute `si:unit` is part of the namespace with URI `http://www.bipm.fr/enus/3_SI/si.html`. However, the attribute `value` is not part of any namespace.

You can control how the parser deals with namespaces. By default, the DOM parser of the Java XML library is not namespace-aware.

To turn on namespace handling, call the `setNamespaceAware` method of the `DocumentBuilderFactory`:

http://www.rejoiceblog.com/

```
factory.setNamespaceAware(true);
```

Now, all builders the factory produces support namespaces. Each node has three properties:

• The *qualified name*, with a prefix, returned by `getNodeName`, `getTagName`, and so on

• The namespace URI, returned by the `getNamespaceURI` method

• The *local name*, without a prefix or a namespace, returned by the `getLocalName` method

Here is an example. Suppose the parser sees the following element:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

It then reports the following:

• Qualified name = `xsd:schema`

• Namespace URI = `http://www.w3.org/2001/XMLSchema`

• Local name = `schema`

Note

If namespace awareness is turned off, `getNamespaceURI` and `getLocalName` return `null`.

org.w3c.dom.Node 1.4

• `String getLocalName()`

http://www.rejoiceblog.com/

returns the local name (without prefix), or `null` if the parser is not namespace-aware.

- `String getNamespaceURI()`

returns the namespace URI, or `null` if the node is not part of a namespace or if the parser is not namespace-aware.

javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isNamespaceAware()`
- `void setNamespaceAware(boolean value)`

gets or sets the `namespaceAware` property of the factory. If set to `true`, the parsers that this factory generates are namespace-aware.

3.7 STREAMING PARSERS

The DOM parser reads an XML document in its entirety into a tree data structure. For most practical applications, DOM works fine. However, it can be inefficient if the document is large and if your processing algorithm is simple enough that you can analyze nodes on the fly, without having to see all of the tree structure. In these cases, you should use a streaming parser.

In the following sections, we discuss the streaming parsers supplied by the Java library: the venerable SAX parser and the more modern StAX parser that was added to Java 6. The SAX parser uses event callbacks, and the StAX parser provides an iterator through the parsing events. The latter is usually a bit more convenient.

3.7.1 Using the SAX Parser

The SAX parser reports events as it parses the components of the XML input, but it does not store the document in any way—it is up to the event handlers to build a data structure. In fact, the DOM parser is built on top of the SAX parser. It builds the DOM tree as it receives the parser events.

Whenever you use a SAX parser, you need a handler that defines the event actions for the various parse events. The `ContentHandler` interface defines several callback methods that the parser executes as it parses the document. Here are the most

http://www.rejoiceblog.com/

important ones:

• `startElement` and `endElement` are called each time a start tag or end tag is encountered.

• `characters` is called whenever character data are encountered.

• `startDocument` and `endDocument` are called once each, at the start and the end of the document.

For example, when parsing the fragment

```
<font>

   <name>Helvetica</name>

   <size units="pt">36</size>

</font>
```

the parser makes the following callbacks:

1. `startElement`, element name: `font`

2. `startElement`, element name: `name`

3. `characters`, content: `Helvetica`

4. `endElement`, element name: `name`

5. `startElement`, element name: `size`, attributes: `units="pt"`

6. `characters`, content: `36`

7. `endElement`, element name: `size`

8. `endElement`, element name: `font`

Your handler needs to override these methods and have them carry out whatever action you want to carry out as you parse the file. The program at the end of this section prints all links `<a href=". . .">` in an HTML file. It simply overrides the `startElement`

http://www.rejoiceblog.com/

method of the handler to check for links with name `a` and an attribute with name `href`. This is potentially useful for implementing a "web crawler"—a program that reaches more and more web pages by following links.

Note

HTML doesn't have to be valid XML, and many web pages deviate so much from proper XML that the example programs will not be able to parse them. However, most pages authored by the W3C are written in XHTML (an HTML dialect that is proper XML). You can use those pages to test the example program. For example, if you run

```
java SAXTest http://www.w3c.org/MarkUp
```

you will see a list of the URLs of all links on that page.

The sample program is a good example for the use of SAX. We don't care at all in which context the `a` elements occur, and there is no need to store a tree structure.

Here is how you get a SAX parser:

```
SAXParserFactory factory = SAXParserFactory.newInstance();

SAXParser parser = factory.newSAXParser();
```

You can now process a document:

```
parser.parse(source, handler);
```

Here, `source` can be a file, URL string, or input stream. The `handler` belongs to a subclass of `DefaultHandler`. The `DefaultHandler` class defines do-nothing methods for the four interfaces:

http://www.rejoiceblog.com/

```
ContentHandler

DTDHandler

EntityResolver

ErrorHandler
```

The example program defines a handler that overrides the `startElement` method of the `ContentHandler` interface to watch out for `a` elements with an `href` attribute:

```
var handler = new DefaultHandler()

   {

     public void startElement(String namespaceURI, String lna

        throws SAXException

     {

        if (lname.equalsIgnoreCase("a") && attrs != null)

        {

           for (int i = 0; i < attrs.getLength(); i++)

           {

              String aname = attrs.getLocalName(i);

              if (aname.equalsIgnoreCase("href"))

                 System.out.println(attrs.getValue(i));

           }
```

http://www.rejoiceblog.com/

```
            }

        }

    };
```

The `startElement` method has three parameters that describe the element name. The `qname` parameter reports the qualified name of the form `prefix:localname`. If namespace processing is turned on, then the `namespaceURI` and `lname` parameters provide the namespace and local (unqualified) name.

As with the DOM parser, namespace processing is turned off by default. To activate namespace processing, call the `setNamespaceAware` method of the factory class:

```
SAXParserFactory factory = SAXParserFactory.newInstance();

factory.setNamespaceAware(true);

SAXParser saxParser = factory.newSAXParser();
```

In this program, we cope with another common issue. An XHTML file starts with a tag that contains a DTD reference, and the parser will want to load it. Understandably, the W3C isn't too happy to serve billions of copies of files such as `www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd`. At one point, they refused altogether, but at the time of this writing, they serve the DTD at a glacial pace. If you don't need to validate the document, just call

```
factory.setFeature("http://apache.org/xml/features/nonvalidat
```

Listing 3.7 contains the code for the web crawler program. Later in this chapter, you will see another interesting use of SAX. An easy way of turning a non-XML data source into XML is to report the SAX events that an XML parser would report. See Section 3.9, "XSL Transformations," on p. 215 for details.

http://www.rejoiceblog.com/

Listing 3.7 `sax/SAXTest.java`

```
1   package sax;
2
3   import  java.io.*;
4   import  java.net.*;
5   import  javax.xml.parsers.*;
6   import  org.xml.sax.*;
7   import  org.xml.sax.helpers.*;
8
9   /**
10   * This program demonstrates how to use a SAX parser. Th
11   * XHTML web page. <br>
12   * Usage: java sax.SAXTest URL
13   * @version 1.01 2018-05-01
14   * @author Cay Horstmann
15   */
16  public class SAXTest
17  {
18       public static void main(String[] args) throws Excep
```

http://www.rejoiceblog.com/

```
19        {
20            String url;
21            if (args.length == 0)
22            {
23                url = "http://www.w3c.org";
24                System.out.println("Using " + url);
25            }
26            else url = args[0];
27
28            var handler = new DefaultHandler()
29                {
30                    public void startElement(String namespaceU
31                        Attributes attrs)
32                    {
33                        if (lname.equals("a") && attrs != null)
34                        {
35                            for (int i = 0; i < attrs.getLength(
36                            {
37                                String aname = attrs.getLocalName
38                                if (aname.equals("href")) System.
```

http://www.rejoiceblog.com/

```
                        }
                    }
                }
            };

            SAXParserFactory factory = SAXParserFactory.newI
            factory.setNamespaceAware(true);
            factory.setFeature("http://apache.org/xml/featur
            SAXParser saxParser = factory.newSAXParser();
            InputStream in = new URL(url).openStream();
            saxParser.parse(in, handler);
        }
    }
```

javax.xml.parsers.SAXParserFactory 1.4

- `static SAXParserFactory newInstance()`

returns an instance of the `SAXParserFactory` class.

- `SAXParser newSAXParser()`

returns an instance of the `SAXParser` class.

- `boolean isNamespaceAware()`

http://www.rejoiceblog.com/

- `void setNamespaceAware(boolean value)`

gets or sets the `namespaceAware` property of the factory. If set to `true`, the parsers that this factory generates are namespace-aware.

- `boolean isValidating()`

- `void setValidating(boolean value)`

gets or sets the `validating` property of the factory. If set to `true`, the parsers that this factory generates validate their input.

javax.xml.parsers.SAXParser 1.4

- `void parse(File f, DefaultHandler handler)`

- `void parse(String url, DefaultHandler handler)`

- `void parse(InputStream in, DefaultHandler handler)`

parses an XML document from the given file, URL, or input stream and reports parse events to the given handler.

org.xml.sax.ContentHandler 1.4

- `void startDocument()`

- `void endDocument()`

is called at the start or the end of the document.

- `void startElement(String uri, String lname, String qname, Attributes attr)`

- `void endElement(String uri, String lname, String qname)`

is called at the start or the end of an element. If the parser is namespace-aware, it reports the URI of the namespace, the local name without prefix, and the qualified

http://www.rejoiceblog.com/

name with prefix.

- `void characters(char[] data, int start, int length)`

is called when the parser reports character data.

org.xml.sax.Attributes 1.4

- `int getLength()`

returns the number of attributes stored in this attribute collection.

- `String getLocalName(int index)`

returns the local name (without prefix) of the attribute with the given index, or the empty string if the parser is not namespace-aware.

- `String getURI(int index)`

returns the namespace URI of the attribute with the given index, or the empty string if the node is not part of a namespace or if the parser is not namespace-aware.

- `String getQName(int index)`

returns the qualified name (with prefix) of the attribute with the given index, or the empty string if the qualified name is not reported by the parser.

- `String getValue(int index)`

- `String getValue(String qname)`

- `String getValue(String uri, String lname)`

returns the attribute value from a given index, qualified name, or namespace URI + local name. Returns `null` if the value doesn't exist.

3.7.2 Using the StAX Parser

The StAX parser is a "pull parser." Instead of installing an event handler, you simply iterate through the events, using this basic loop:

http://www.rejoiceblog.com/

```
InputStream in = url.openStream();

XMLInputFactory factory = XMLInputFactory.newInstance();

XMLStreamReader parser = factory.createXMLStreamReader(in);

while (parser.hasNext())

{

    int event = parser.next();

    Call parser methods to obtain event details

}
```

For example, when parsing the fragment

```
<font>

    <name>Helvetica</name>

    <size units="pt">36</size>

</font>
```

the parser yields the following events:

1. `START_ELEMENT`, element name: `font`

2. `CHARACTERS`, content: white space

3. `START_ELEMENT`, element name: `name`

4. `CHARACTERS`, content: `Helvetica`

5. `END_ELEMENT`, element name: `name`

http://www.rejoiceblog.com/

6. `CHARACTERS`, content: white space

7. `START_ELEMENT`, element name: `size`

8. `CHARACTERS`, content: `36`

9. `END_ELEMENT`, element name: `size`

10. `CHARACTERS`, content: white space

11. `END_ELEMENT`, element name: `font`

To analyze the attribute values, call the appropriate methods of the `XMLStreamReader` class. For example,

```
String units = parser.getAttributeValue(null, "units");
```

gets the `units` attribute of the current element.

By default, namespace processing is enabled. You can deactivate it by modifying the factory:

```
XMLInputFactory factory = XMLInputFactory.newInstance();

factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false
```

Listing 3.8 contains the code for the web crawler program implemented with the StAX parser. As you can see, the code is simpler than the equivalent SAX code because you don't have to worry about event handling.

Listing 3.8 `stax/StAXTest.java`

```
package stax;

import java.io.*;
```

http://www.rejoiceblog.com/

```
 4   import java.net.*;
 5   import javax.xml.stream.*;
 6
 7   /**
 8    * This program demonstrates how to use a StAX parser. T
 9    * XHTML web page. <br>
10    * Usage: java stax.StAXTest URL
11    * @author Cay Horstmann
12    * @version 1.1 2018-05-01
13    */
14   public class StAXTest
15   {
16       public static void main(String[] args) throws Excep
17       {
18          String urlString;
19          if (args.length == 0)
20          {
21             urlString = "http://www.w3c.org";
22             System.out.println("Using " + urlString);
```

http://www.rejoiceblog.com/

```
23            }
24            else urlString = args[0];
25            var url = new URL(urlString);
26            InputStream in = url.openStream();
27            XMLInputFactory factory = XMLInputFactory.newIns
28            XMLStreamReader parser = factory.createXMLStream
29            while (parser.hasNext())
30            {
31               int event = parser.next();
32               if (event == XMLStreamConstants.START_ELEMENT
33               {
34                  if (parser.getLocalName().equals("a"))
35                  {
36                     String href = parser.getAttributeValue(
37                     if (href != null)
38                        System.out.println(href);
39                  }
40               }
41            }
42         }
```

http://www.rejoiceblog.com/

```
}
```

javax.xml.stream.XMLInputFactory 6

- `static XMLInputFactory newInstance()`

returns an instance of the `XMLInputFactory` class.

- `void setProperty(String name, Object value)`

sets a property for this factory, or throws an `IllegalArgumentException` if the property is not supported or cannot be set to the given value. The JDK implementation supports the following Boolean-valued properties:

`"javax.xml.stream.isValidating"`	When `false` (the default), the document is not validated. Not required by the specification.
`"javax.xml.stream.isNamespaceAware"`	When `true` (the default), namespaces are processed. Not required by the specification.
`"javax.xml.stream.isCoalescing"`	When `false` (the default), adjacent character data are not coalesced.
`"javax.xml.stream.isReplacingEntityReferences"`	When `true` (the default), entity references are replaced and reported as character data.
`"javax.xml.stream.isSupportingExternalEntities"`	When `true` (the default), external entities are resolved. The specification gives no default for this property.
`"javax.xml.stream.supportDTD"`	When `true` (the default), DTDs are reported as events.

- `XMLStreamReader createXMLStreamReader(InputStream in)`

- `XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)`

- `XMLStreamReader createXMLStreamReader(Reader in)`

- `XMLStreamReader createXMLStreamReader(Source in)`

http://www.rejoiceblog.com/

creates a parser that reads from the given stream, reader, or JAXP source.

javax.xml.stream.XMLStreamReader 6

- `boolean hasNext()`

returns `true` if there is another parse event.

- `int next()`

sets the parser state to the next parse event and returns one of the following constants: `START_ELEMENT`, `END_ELEMENT`, `CHARACTERS`, `START_DOCUMENT`, `END_DOCUMENT`, `CDATA`, `COMMENT`, `SPACE` (ignorable whitespace), `PROCESSING_INSTRUCTION`, `ENTITY_REFERENCE`, `DTD`.

- `boolean isStartElement()`

- `boolean isEndElement()`

- `boolean isCharacters()`

- `boolean isWhiteSpace()`

returns `true` if the current event is a start element, end element, character data, or whitespace.

- `QName getName()`

- `String getLocalName()`

gets the name of the element in a `START_ELEMENT` or `END_ELEMENT` event.

- `String getText()`

returns the characters of a `CHARACTERS`, `COMMENT`, or `CDATA` event, the replacement value for an `ENTITY_REFERENCE`, or the internal subset of a `DTD`.

- `int getAttributeCount()`

- `QName getAttributeName(int index)`

http://www.rejoiceblog.com/

- `String getAttributeLocalName(int index)`

- `String getAttributeValue(int index)`

gets the attribute count and the names and values of the attributes, provided the current event is `START_ELEMENT`.

- `String getAttributeValue(String namespaceURI, String name)`

gets the value of the attribute with the given name, provided the current event is `START_ELEMENT`. If `namespaceURI` is `null`, the namespace is not checked.

3.8 GENERATING XML DOCUMENTS

You now know how to write Java programs that read XML. Let us now turn to the opposite process: producing XML output. Of course, you could write an XML file simply by making a sequence of `print` calls, printing the elements, attributes, and text content, but that would not be a good idea. The code is rather tedious, and you can easily make mistakes if you don't pay attention to special symbols (such as `"` or <) in the attribute values and text content.

A better approach is to build up a DOM tree with the contents of the document and then write out the tree contents. The following sections discuss the details.

3.8.1 Documents without Namespaces

To build a DOM tree, you start out with an empty document. You can get an empty document by calling the `newDocument` method of the `DocumentBuilder` class:

```
Document doc = builder.newDocument();
```

Use the `createElement` method of the `Document` class to construct the elements of your document:

```
Element rootElement = doc.createElement(rootName); Element ch
```

Use the `createTextNode` method to construct text nodes:

http://www.rejoiceblog.com/

```
Text textNode = doc.createTextNode(textContents);
```

Add the root element to the document, and add the child nodes to their parents:

```
doc.appendChild(rootElement); rootElement.appendChild(childEl
```

As you build up the DOM tree, you may also need to set element attributes. Simply call the `setAttribute` method of the `Element` class:

```
rootElement.setAttribute(name, value);
```

3.8.2 Documents with Namespaces

If you use namespaces, the procedure for creating a document is slightly different.

First, set the builder factory to be namespace-aware, then create the builder:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newIn
```

Then use `createElementNS` instead of `createElement` to create any nodes:

```
String namespace = "http://www.w3.org/2000/svg";
```

```
Element rootElement = doc.createElementNS(namespace, "svg");
```

If your node has a qualified name with a namespace prefix, then any necessary `xmlns`-prefixed attributes are created automatically. For example, if you need SVG inside XHTML, you can construct an element like this:

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

http://www.rejoiceblog.com/

When the element is written, it turns into

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

If you need to set element attributes whose names are in a namespace, use the `setAttributeNS` method of the `Element` class:

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

3.8.3 Writing Documents

Somewhat curiously, it is not so easy to write a DOM tree to an output stream. The easiest approach is to use the Extensible Stylesheet Language Transformations (XSLT) API. For more information about XSLT, turn to Section 3.9, "XSL Transformations," on p. 215. Right now, consider the code that follows a magic incantation to produce XML output.

We apply the do-nothing transformation to the document and capture its output. To include a `DOCTYPE` node in the output, we also need to set the `SYSTEM` and `PUBLIC` identifiers as output properties.

```
// construct the do-nothing transformation

Transformer t = TransformerFactory.newInstance().newTransform

// set output properties to get a DOCTYPE node

t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, systemIdentifi

t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, publicIdentifi

// set indentation

t.setOutputProperty(OutputKeys.INDENT, "yes");
```

http://www.rejoiceblog.com/

```
t.setOutputProperty(OutputKeys.METHOD, "xml");

t.setOutputProperty("{http://xml.apache.org/xslt}indent-amoun

// apply the do-nothing transformation and send the output to

t.transform(new DOMSource(doc), new StreamResult(new FileOutp
```

Another approach is to use the `LSSerializer` interface. To get an instance, you have to use the following magic incantation:

```
DOMImplementation impl = doc.getImplementation();

var implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0

LSSerializer ser = implLS.createLSSerializer();
```

If you want spaces and line breaks, set this flag:

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

Then it's simple enough to convert a document to a string:

```
String str = ser.writeToString(doc);
```

If you want to write the output directly to a file, you need an `LSOutput`:

```
LSOutput out = implLS.createLSOutput();

out.setEncoding("UTF-8");

out.setByteStream(Files.newOutputStream(path));
```

http://www.rejoiceblog.com/

```
   ser.write(doc, out);
```

javax.xml.parsers.DocumentBuilder 1.4

- `Document newDocument()`

returns an empty document.

org.w3c.dom.Document 1.4

- `Element createElement(String name)`
- `Element createElementNS(String uri, String qname)`

creates an element with the given name.

- `Text createTextNode(String data)`

creates a text node with the given data.

org.w3c.dom.Node 1.4

- `Node appendChild(Node child)`

appends a node to the list of children of this node. Returns the appended node.

org.w3c.dom.Element 1.4

- `void setAttribute(String name, String value)`
- `void setAttributeNS(String uri, String qname, String value)`

sets the attribute with the given name to the given value. If the qualified name has an alias prefix, then `uri` must not be `null`.

http://www.rejoiceblog.com/

javax.xml.transform.TransformerFactory 1.4

- `static TransformerFactory newInstance()`

returns an instance of the `TransformerFactory` class.

- `Transformer newTransformer()`

returns an instance of the `Transformer` class that carries out an identity (do-nothing) transformation.

javax.xml.transform.Transformer 1.4

- `void setOutputProperty(String name, String value)`

sets an output property. See `www.w3.org/TR/xslt#output` for a listing of the standard output properties. The most useful ones are shown here:

doctype-public	The public ID to be used in the DOCTYPE declaration
doctype-system	The system ID to be used in the DOCTYPE declaration
indent	"yes" or "no"
method	"xml", "html", "text", or a custom string

- `void transform(Source from, Result to)`

transforms an XML document.

javax.xml.transform.dom.DOMSource 1.4

- `DOMSource(Node n)`

constructs a source from the given node. Usually, `n` is a document node.

javax.xml.transform.stream.StreamResult 1.4

http://www.rejoiceblog.com/

- `StreamResult(File f)`
- `StreamResult(OutputStream out)`
- `StreamResult(Writer out)`
- `StreamResult(String systemID)`

constructs a stream result from a file, stream, writer, or system ID (usually a relative or absolute URL).

3.8.4 Writing an XML Document with StAX

In the preceding section, you saw how to produce an XML document by writing a DOM tree. If you have no other use for the DOM tree, that approach is not very efficient.

The StAX API lets you write an XML tree directly. Construct an `XMLStreamWriter` from an `OutputStream`:

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();

XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

To produce the XML header, call

```
writer.writeStartDocument()
```

Then call

```
writer.writeStartElement(name);
```

Add attributes by calling

```
writer.writeAttribute(name, value);
```

http://www.rejoiceblog.com/

Now you can add child elements by calling `writeStartElement` again, or write characters with

```
writer.writeCharacters(text);
```

When you have written all child nodes, call

```
writer.writeEndElement();
```

This causes the current element to be closed.

To write an element without children (such as `<img . . ./>`), use the call

```
writer.writeEmptyElement(name);
```

Finally, at the end of the document, call

```
writer.writeEndDocument();
```

This call closes any open elements.

You still need to close the `XMLStreamWriter`, and you need to do it manually since the `XMLStreamWriter` interface does not extend the `AutoCloseable` interface.

As with the DOM/XSLT approach, you don't have to worry about escaping characters in attribute values and character data. However, it is possible to produce malformed XML, such as a document with multiple root nodes. Also, the current version of StAX has no support for producing indented output.

The program in Listing 3.9 shows you both approaches for writing XML.

Listing 3.9 `write/XMLWriteTest.java`

```
package write;
```

http://www.rejoiceblog.com/

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

import   javax.xml.parsers.*;
import   javax.xml.stream.*;
import   javax.xml.transform.*;
import   javax.xml.transform.dom.*;
import   javax.xml.transform.stream.*;

import org.w3c.dom.*;

/**
 * This program shows how to write an XML file. It produ
 * @version 1.12 2016-04-27
 * @author Cay Horstmann
 */
public class XMLWriteTest
{
```

http://www.rejoiceblog.com/

```
22        public static void main(String[] args)
23              throws ParserConfigurationException, Transfor
24        {
25            Document doc = newDrawing(600, 400);
26            writeDocument(doc, "drawing1.svg");
27            writeNewDrawing(600, 400, "drawing2.svg");
28        }
29
30      private static Random generator = new Random();
31
32      /**
33       * Creates a new random drawing.
34       * @return the DOM tree of the SVG document
35       */
36      public static Document newDrawing(int drawingWidth, i
37              throws ParserConfigurationException
38      {
39          DocumentBuilderFactory factory = DocumentBuilder
40          factory.setNamespaceAware(true);
```

http://www.rejoiceblog.com/

```
        DocumentBuilder builder = factory.newDocumentBuild
        var namespace = "http://www.w3.org/2000/svg";
        Document doc = builder.newDocument();
        Element svgElement = doc.createElementNS(namespace
        doc.appendChild(svgElement);
        svgElement.setAttribute("width", "" + drawingWidth
        svgElement.setAttribute("height", "" + drawingHeig
        int n = 10 + generator.nextInt(20);
        for (int i = 1; i <= n; i++)
        {
            int x = generator.nextInt(drawingWidth);
            int y = generator.nextInt(drawingHeight);
            int width = generator.nextInt(drawingWidth - x)
            int height = generator.nextInt(drawingHeight -
            int r = generator.nextInt(256);
            int g = generator.nextInt(256);
            int b = generator.nextInt(256);

            Element rectElement = doc.createElementNS(names
            rectElement.setAttribute("x", "" + x);
```

http://www.rejoiceblog.com/

```
            rectElement.setAttribute("y", "" + y);
            rectElement.setAttribute("width", "" + width);
            rectElement.setAttribute("height", "" + height)
            rectElement.setAttribute("fill", String.format(
            svgElement.appendChild(rectElement);
        }
        return doc;
    }

    /**
     * Saves a document using DOM/XSLT
     */
    public static void writeDocument(Document doc, String
            throws TransformerException, IOException
    {
        Transformer t = TransformerFactory.newInstance().
        t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
            "http://www.w3.org/TR/2000/CR-SVG-20000802/
        t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "-
```

http://www.rejoiceblog.com/

```
        t.setOutputProperty(OutputKeys.INDENT, "yes");
        t.setOutputProperty(OutputKeys.METHOD, "xml");
        t.setOutputProperty("{http://xml.apache.org/xslt}
        t.transform(new DOMSource(doc), new StreamResult(
            Files.newOutputStream(Paths.get(filename)))
    }

    /**
     * Writes an SVG document of the current drawing.
        * @param writer the document destination
        * @throws IOException
        */
      public static void writeNewDrawing(int drawingWidth
            throws XMLStreamException, IOException
      {
          XMLOutputFactory factory = XMLOutputFactory.ne
          XMLStreamWriter writer = factory.createXMLStre
              Files.newOutputStream(Paths.get(filename
          writer.writeStartDocument();
          writer.writeDTD("<!DOCTYPE svg PUBLIC \"-//W3C
```

http://www.rejoiceblog.com/

```
                + "\"http://www.w3.org/TR/2000/CR-SVG-20
            writer.writeStartElement("svg");
            writer.writeDefaultNamespace("http://www.w3.or
            writer.writeAttribute("width", "" + drawingWid
            writer.writeAttribute("height", "" + drawingHe
            int n = 10 + generator.nextInt(20);
            for (int i = 1; i <= n; i++)
            {
                int x = generator.nextInt(drawingWidth);
                int y = generator.nextInt(drawingHeight);
                int width = generator.nextInt(drawingWidth
                int height = generator.nextInt(drawingHeigh
                int r = generator.nextInt(256);
                int g = generator.nextInt(256);
                int b = generator.nextInt(256);
                writer.writeEmptyElement("rect");
                writer.writeAttribute("x", "" + x);
                writer.writeAttribute("y", "" + y);
                writer.writeAttribute("width", "" + width);
```

http://www.rejoiceblog.com/

```
                writer.writeAttribute("height", "" + height
                writer.writeAttribute("fill", String.format
            }
            writer.writeEndDocument(); // closes svg eleme
        }
    }
```

javax.xml.stream.XMLOutputFactory 6

- `static XMLOutputFactory newInstance()`

returns an instance of the `XMLOutputFactory` class.

- `XMLStreamWriter createXMLStreamWriter(OutputStream in)`
- `XMLStreamWriter createXMLStreamWriter(OutputStream in, String characterEncoding)`
- `XMLStreamWriter createXMLStreamWriter(Writer in)`
- `XMLStreamWriter createXMLStreamWriter(Result in)`

creates a writer that writes to the given stream, writer, or JAXP result.

javax.xml.stream.XMLStreamWriter 6

- `void writeStartDocument()`
- `void writeStartDocument(String xmlVersion)`
- `void writeStartDocument(String encoding, String xmlVersion)`

http://www.rejoiceblog.com/

writes the XML processing instruction at the top of the document. Note that the `encoding` parameter is only used to write the attribute. It does not set the character encoding of the output.

- `void setDefaultNamespace(String namespaceURI)`

- `void setPrefix(String prefix, String namespaceURI)`

sets the default namespace or the namespace associated with a prefix. The declaration is scoped to the current element or, if no element has been written, to the document root.

- `void writeStartElement(String localName)`

- `void writeStartElement(String namespaceURI, String localName)`

writes a start tag, replacing the `namespaceURI` with the associated prefix.

- `void writeEndElement()`

closes the current element.

- `void writeEndDocument()`

closes all open elements.

- `void writeEmptyElement(String localName)`

- `void writeEmptyElement(String namespaceURI, String localName)`

writes a self-closing tag, replacing the `namespaceURI` with the associated prefix.

- `void writeAttribute(String localName, String value)`

- `void writeAttribute(String namespaceURI, String localName, String value)`

writes an attribute for the current element, replacing the `namespaceURI` with the associated prefix.

- `void writeCharacters(String text)`

writes character data.

http://www.rejoiceblog.com/

- `void writeCData(String text)`

writes a `CDATA` block.

- `void writeDTD(String dtd)`

writes the `dtd` string, which is assumed to contain a `DOCTYPE` declaration.

- `void writeComment(String comment)`

writes a comment.

- `void close()`

closes this writer.

3.8.5 An Example: Generating an SVG File

Listing 3.9 is a typical program that produces XML output. The program draws a modernist painting—a random set of colored rectangles (see Figure 3.3). To save a masterpiece, we use the Scalable Vector Graphics (SVG) format. SVG is an XML format to describe complex graphics in a device-independent fashion. You can find more information about SVG at `www.w3c.org/Images/SVG`. To view SVG files, simply use any modern browser.

Figure 3.3 Generating modern art

The program demonstrates two ways of producing XML: by constructing and saving a DOM tree, and by directly writing the XML with the StAX API.

http://www.rejoiceblog.com/

We don't need to go into details about SVG; for our purposes, we just need to know how to express a set of colored rectangles. Here is a sample:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"

   "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-2000080

<svg xmlns="http://www.w3.org/2000/svg" width="300" height="1

   <rect x="231" y="61" width="9" height="12" fill="#6e4a13"/

   <rect x="107" y="106" width="56" height="5" fill="#c406be"

   . . .

</svg>
```

As you can see, each rectangle is described as a `rect` node. The position, width, height, and fill color are attributes. The fill color is an RGB value in hexadecimal.

Note

SVG uses attributes heavily. In fact, some attributes are quite complex. For example, here is a path element:

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

The `M` denotes a "moveto" command, `L` is "lineto," and `z` is "closepath" (!). Apparently, the designers of this data format didn't have much confidence in using XML for structured data. In your own XML formats, you might want to use elements instead of complex attributes.

http://www.rejoiceblog.com/

3.9 XSL TRANSFORMATIONS

The XSL Transformations (XSLT) mechanism allows you to specify rules for transforming XML documents into other formats, such as plain text, XHTML, or any other XML format. XSLT is commonly used to translate from one machine-readable XML format to another, or to translate XML into a presentation format for human consumption.

You need to provide an XSLT stylesheet that describes the conversion of XML documents into some other format. An XSLT processor reads an XML document and the stylesheet and produces the desired output (see Figure 3.4).

Figure 3.4 Applying XSL transformations

The XSLT specification is quite complex, and entire books have been written on the subject. We can't possibly discuss all the features of XSLT, so we will just work through a representative example. You can find more information in the book *Essential XML* by Don Box et al. The XSLT specification is available at `www.w3.org/TR/xslt`.

Suppose we want to transform XML files with employee records into HTML documents. Consider this input file:

```
<staff>

   <employee>

      <name>Carl Cracker</name>
```

http://www.rejoiceblog.com/

```
        <salary>75000</salary>

        <hiredate year="1987" month="12" day="15"/>

    </employee>

    <employee>

        <name>Harry Hacker</name>

        <salary>50000</salary>

        <hiredate year="1989" month="10" day="1"/>

    </employee>

    <employee>

        <name>Tony Tester</name>

        <salary>40000</salary>

        <hiredate year="1990" month="3" day="15"/>

    </employee>

</staff>
```

The desired output is an HTML table:

```
<table border="1">

<tr>

<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>

</tr>
```

http://www.rejoiceblog.com/

```
<tr>

<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>

</tr>

<tr>

<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>

</tr>

</table>
```

A stylesheet with transformation templates has this form:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet

   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

   version="1.0">

   <xsl:output method="html"/>

   template 1



   template2

   . . .

</xsl:stylesheet>
```

In our example, the `xsl:output` element specifies the method as HTML. Other valid

http://www.rejoiceblog.com/

method settings are `xml` and `text`.

Here is a typical template:

```
<xsl:template match="/staff/employee">

   <tr><xsl:apply-templates/></tr>

</xsl:template>
```

The value of the `match` attribute is an XPath expression. The template states: Whenever you see a node in the XPath set `/staff/employee`, do the following:

1. Emit the string `<tr>`.

2. Keep applying templates as you process its children.

3. Emit the string `</tr>` after you are done with all children.

In other words, this template generates the HTML table row markers around every employee record.

The XSLT processor starts processing by examining the root element. Whenever a node matches one of the templates, it applies the template. (If multiple templates match, the best matching one is used; see the specification at `www.w3.org/TR/xslt` for the gory details.) If no template matches, the processor carries out a default action. For text nodes, the default is to include the contents in the output. For elements, the default action is to create no output but to keep processing the children.

Here is a template for transforming `name` nodes in an employee file:

```
<xsl:template match="/staff/employee/name">

   <td><xsl:apply-templates/></td>

</xsl:template>
```

As you can see, the template produces the `<td>. . .</td>` delimiters, and it asks the

http://www.rejoiceblog.com/

processor to recursively visit the children of the `name` element. There is just one child—the text node. When the processor visits that node, it emits the text contents (provided, of course, that there is no other matching template).

You have to work a little harder if you want to copy attribute values into the output. Here is an example:

```
<xsl:template match="/staff/employee/hiredate">

   <td><xsl:value-of select="@year"/>-<xsl:value-of

   select="@month"/>-<xsl:value-of select="@day"/></td>

</xsl:template>
```

When processing a `hiredate` node, this template emits

1. The string `<td>`

2. The value of the `year` attribute

3. A hyphen

4. The value of the `month` attribute

5. A hyphen

6. The value of the `day` attribute

7. The string `</td>`

The `xsl:value-of` statement computes the string value of a node set. The node set is specified by the XPath value of the `select` attribute. In this case, the path is relative to the currently processed node. The node set is converted to a string by concatenation of the string values of all nodes. The string value of an attribute node is its value. The string value of a text node is its contents. The string value of an element node is the concatenation of the string values of its child nodes (but not its attributes).

Listing 3.10 contains the stylesheet for turning an XML file with employee records into an HTML table.

http://www.rejoiceblog.com/

Listing 3.11 shows a different set of transformations. The input is the same XML file, and the output is plain text in the familiar property file format:

```
employee.1.name=Carl Cracker

employee.1.salary=75000.0

employee.1.hiredate=1987-12-15

employee.2.name=Harry Hacker

employee.2.salary=50000.0

employee.2.hiredate=1989-10-1

employee.3.name=Tony Tester

employee.3.salary=40000.0

employee.3.hiredate=1990-3-15
```

That example uses the `position()` function which yields the position of the current node as seen from its parent. We thus get an entirely different output simply by switching the stylesheet. This means you can safely use XML to describe your data; if some applications need the data in another format, just use XSLT to generate the alternative format.

It is simple to generate XSL transformations in the Java platform. Set up a transformer factory for each stylesheet. Then, get a transformer object and tell it to transform a source to a result:

```
var styleSheet = new File(filename);

var styleSource = new StreamSource(styleSheet);

Transformer t = TransformerFactory.newInstance().newTransform

t.transform(source, result);
```

http://www.rejoiceblog.com/

The parameters of the `transform` method are objects of classes that implement the `Source` and `Result` interfaces. Several classes implement the `Source` interface:

```
DOMSource

SAXSource

StAXSource

StreamSource
```

You can construct a `StreamSource` from a file, stream, reader, or URL, and a `DOMSource` from the node of a DOM tree. For example, in the preceding section, we invoked the identity transformation as

```
t.transform(new DOMSource(doc), result);
```

In our example program, we do something slightly more interesting. Instead of starting out with an existing XML file, we produce a SAX XML reader that gives the illusion of parsing an XML file by emitting appropriate SAX events. Actually, our XML reader reads a flat file, as described in Chapter 1. The input file looks like this:

```
Carl Cracker|75000.0|1987|12|15

Harry Hacker|50000.0|1989|10|1

Tony Tester|40000.0|1990|3|15
```

Our XML reader generates SAX events as it processes the input. Here is a part of the `parse` method of the `EmployeeReader` class that implements the `XMLReader` interface:

```
var attributes = new AttributesImpl();
```

http://www.rejoiceblog.com/

```
handler.startDocument();

handler.startElement("", "staff", "staff", attributes);

while ((line = in.readLine()) != null)

{

   handler.startElement("", "employee", "employee", attribute

   var tokenizer = new StringTokenizer(line, "|");

   handler.startElement("", "name", "name", attributes);

   String s = tokenizer.nextToken();

   handler.characters(s.toCharArray(), 0, s.length());

   handler.endElement("", "name", "name");

   . . .

   handler.endElement("", "employee", "employee");

}

handler.endElement("", rootElement, rootElement);

handler.endDocument();
```

The `SAXSource` for the transformer is constructed from the XML reader:

```
t.transform(new SAXSource(new EmployeeReader(),

   new InputSource(new FileInputStream(filename))), result);
```

http://www.rejoiceblog.com/

This is an ingenious trick to convert non-XML legacy data into XML. Of course, most XSLT applications will already have XML input data, and you can simply invoke the `transform` method on a `StreamSource`:

```
t.transform(new StreamSource(file), result);
```

The transformation result is an object of a class that implements the `Result` interface. The Java library supplies three classes:

```
DOMResult

SAXResult

StreamResult
```

To store the result in a DOM tree, use a `DocumentBuilder` to generate a new document node and wrap it into a `DOMResult`:

```
Document doc = builder.newDocument();

t.transform(source, new DOMResult(doc));
```

To save the output in a file, use a `StreamResult`:

```
t.transform(source, new StreamResult(file));
```

Listing 3.12 contains the complete source code.

Listing 3.10 `transform/makehtml.xsl`

```
<?xml version="1.0" encoding="ISO-8859-1"?>

```

http://www.rejoiceblog.com/

```
3   <xsl:stylesheet
4       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5       version="1.0">
6
7       <xsl:output method="html"/>
8
9       <xsl:template match="/staff">
10         <table border="1"><xsl:apply-templates/></table>
11      </xsl:template>
12
13      <xsl:template match="/staff/employee">
14         <tr><xsl:apply-templates/></tr>
15      </xsl:template>
16
17      <xsl:template match="/staff/employee/name">
18         <td><xsl:apply-templates/></td>
19      </xsl:template>
20
21      <xsl:template match="/staff/employee/salary">
22         <td>$<xsl:apply-templates/></td>
```

http://www.rejoiceblog.com/

```
    </xsl:template>

    <xsl:template match="/staff/employee/hiredate">
        <td><xsl:value-of select="@year"/>-<xsl:value-of
        select="@month"/>-<xsl:value-of select="@day"/></t
    </xsl:template>

</xsl:stylesheet>
```

Listing 3.11 `transform/makeprop.xsl`

```
<?xml version="1.0"?>

<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

    <xsl:output method="text" omit-xml-declaration="yes"/
```

http://www.rejoiceblog.com/

```
   <xsl:template match="/staff/employee">
employee.<xsl:value-of select="position()"
/>.name=<xsl:value-of select="name/text()"/>
employee.<xsl:value-of select="position()"
/>.salary=<xsl:value-of select="salary/text()"/>
employee.<xsl:value-of select="position()"
/>.hiredate=<xsl:value-of select="hiredate/@year"
/>-<xsl:value-of select="hiredate/@month"
/>-<xsl:value-of select="hiredate/@day"/>
   </xsl:template>

</xsl:stylesheet>
```

Listing 3.12 `transform/TransformTest.java`

```
package transform;

import  java.io.*;
import  java.nio.file.*;
import  java.util.*;
import  javax.xml.transform.*;
```

http://www.rejoiceblog.com/

```
 7   import  javax.xml.transform.sax.*;
 8   import  javax.xml.transform.stream.*;
 9   import  org.xml.sax.*;
10   import  org.xml.sax.helpers.*;
11
12   /**
13    * This program demonstrates XSL transformations. It app
14    * records. The records are stored in the file employee.
15    * stylesheet on the command line, e.g.
16    *     java transform.TransformTest transform/makeprop.x
17    * @version 1.04 2018-04-10
18    * @author Cay Horstmann
19    */
20   public class TransformTest
21   {
22        public static void main(String[] args) throws Excep
23        {
24           Path path;
25           if (args.length > 0) path = Paths.get(args[0]);
```

http://www.rejoiceblog.com/

```
            else path = Paths.get("transform", "makehtml.xsl
            try (InputStream styleIn = Files.newInputStream(
            {
               var styleSource = new StreamSource(styleIn);

                Transformer t = TransformerFactory.newInstan
                t.setOutputProperty(OutputKeys.INDENT, "yes"
                t.setOutputProperty(OutputKeys.METHOD, "xml"
                t.setOutputProperty("{http://xml.apache.org/

                try (InputStream docIn = Files.newInputStrea
                {
                   t.transform(new SAXSource(new EmployeeRea
                      new StreamResult(System.out));
                }
            }
        }
    }

    /**
```

http://www.rejoiceblog.com/

```
46     * This class reads the flat file employee.dat and repo
47     * parsing an XML file.
48     */
49   class EmployeeReader implements XMLReader
50   {
51        private ContentHandler handler;
52
53       public void parse(InputSource source) throws IOExcep
54       {
55          InputStream stream = source.getByteStream();
56          var in = new BufferedReader(new InputStreamReader
57          String rootElement = "staff";
58          var atts = new AttributesImpl();
59
60           if (handler == null) throw new SAXException("No
61
62          handler.startDocument();
63          handler.startElement("", rootElement, rootElement
64          String line;
```

http://www.rejoiceblog.com/

```
65          while ((line = in.readLine()) != null)
66          {
67             handler.startElement("", "employee", "employee
68             var t = new StringTokenizer(line, "|");
69
70              handler.startElement("", "name", "name", atts
71              String s = t.nextToken();
72              handler.characters(s.toCharArray(), 0, s.leng
73              handler.endElement("", "name", "name");
74
75              handler.startElement("", "salary", "salary",
76              s = t.nextToken();
77              handler.characters(s.toCharArray(), 0, s.leng
78              handler.endElement("", "salary", "salary");
79
80              atts.addAttribute("", "year", "year", "CDATA"
81              atts.addAttribute("", "month", "month", "CDAT
82              atts.addAttribute("", "day", "day", "CDATA",
83              handler.startElement("", "hiredate", "hiredat
84              handler.endElement("", "hiredate", "hiredate"
```

http://www.rejoiceblog.com/

```
            atts.clear();

            handler.endElement("", "employee", "employee"
        }

        handler.endElement("", rootElement, rootElement);
        handler.endDocument();
    }

    public void setContentHandler(ContentHandler newValue
    {
        handler = newValue;
    }

    public ContentHandler getContentHandler()
    {
        return handler;
    }

```

http://www.rejoiceblog.com/

```
    // the following methods are just do-nothing implem
    public void parse(String systemId) throws IOExcepti
    public void setErrorHandler(ErrorHandler handler) {
    public ErrorHandler getErrorHandler() { return null
    public void setDTDHandler(DTDHandler handler) {}
    public DTDHandler getDTDHandler() { return null; }
      public void setEntityResolver(EntityResolver reso
      public EntityResolver getEntityResolver() { retur
      public void setProperty(String name, Object value
      public Object getProperty(String name) { return n
      public void setFeature(String name, boolean value
      public boolean getFeature(String name) { return f
  }
```

javax.xml.transform.TransformerFactory 1.4

- `Transformer newTransformer(Source styleSheet)`

returns an instance of the `Transformer` class that reads a stylesheet from the given source.

javax.xml.transform.stream.StreamSource 1.4

http://www.rejoiceblog.com/

- `StreamSource(File f)`
- `StreamSource(InputStream in)`
- `StreamSource(Reader in)`
- `StreamSource(String systemID)`

constructs a stream source from a file, stream, reader, or system ID (usually a relative or absolute URL).

javax.xml.transform.sax.SAXSource 1.4

- `SAXSource(XMLReader reader, InputSource source)`

constructs a SAX source that obtains data from the given input source and uses the given reader to parse the input.

org.xml.sax.XMLReader 1.4

- `void setContentHandler(ContentHandler handler)`

sets the handler that is notified of parse events as the input is parsed.

- `void parse(InputSource source)`

parses the input from the given input source and sends parse events to the content handler.

javax.xml.transform.dom.DOMResult 1.4

- `DOMResult(Node n)`

constructs a source from the given node. Usually, `n` is a new document node.

http://www.rejoiceblog.com/

org.xml.sax.helpers.AttributesImpl 1.4

- `void addAttribute(String uri, String lname, String qname, String type, String value)`

adds an attribute to this attribute collection. The `lname` parameter is the local name without prefix, and `qname` ist he qualified name with prefix. The `type` parameter is one of `"CDATA"`, `"ID"`, `"IDREF"`, `"IDREFS"`, `"NMTOKEN"`, `"NMTOKENS"`, `"ENTITY"`, `"ENTITIES"`, or `"NOTATION"`

- `void clear()`

removes all attributes from this attribute collection.

This example concludes our discussion of XML support in the Java library. You should now have a good perspective on the major strengths of XML—in particular, for automated parsing and validation and as a powerful transformation mechanism. Of course, all this technology is only going to work for you if you design your XML formats well. You need to make sure that the formats are rich enough to express all your business needs, that they are stable over time, and that your business partners are willing to accept your XML documents. Those issues can be far more challenging than dealing with parsers, DTDs, or transformations.

In the next chapter, we will discuss network programming on the Java platform, starting with the basics of network sockets and moving on to higher-level protocols for e-mail and the World Wide Web.

http://www.rejoiceblog.com/

Chapter 4 Networking

In this chapter

- 4.1 Connecting to a Server
- 4.2 Implementing Servers
- 4.3 Getting Web Data
- 4.4 The HTTP Client
- 4.5 Sending E-Mail

We begin this chapter by reviewing basic networking concepts, then move on to writing Java programs that connect to network services. We will show you how network clients and servers are implemented. Finally, you will see how to send e-mail from a Java program and how to harvest information from a web server.

4.1 CONNECTING TO A SERVER

In the following sections, you will connect to a server, first by hand and with telnet, and then with a Java program.

4.1.1 Using Telnet

The telnet program is a great debugging tool for network programming. You should be able to launch it by typing `telnet` from a command shell.

Note

In Windows, you need to activate telnet. Go to the Control Panel, select Programs, click Turn Windows Features On or Off, and select the Telnet client checkbox. The Windows firewall also blocks quite a few network ports

http://www.rejoiceblog.com/

that we use in this chapter; you might need an administrator account to unblock them.

You may have used telnet to connect to a remote computer, but you can use it to communicate with other services provided by Internet hosts as well. Here is an example of what you can do. Type

```
telnet time-a.nist.gov 13
```

As Figure 4.1 shows, you should get back a line like this:

```
57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *
```

Figure 4.1 Output of the "time of day" service

What is going on? You have connected to the "time of day" service that most UNIX machines constantly run. The particular server that you connected to is operated by the National Institute of Standards and Technology and gives the measurement of a Cesium atomic clock. (Of course, the reported time is not completely accurate due to network delays.)

http://www.rejoiceblog.com/

By convention, the “time of day” service is always attached to “port” number 13.

Note

In network parlance, a port is not a physical device, but an abstraction facilitating communication between a server and a client (see Figure 4.2).

Figure 4.2 A client connecting to a server port

The server software is continuously running on the remote machine, waiting for any network traffic that wants to chat with port 13. When the operating system on the remote computer receives a network package that contains a request to connect to port number 13, it wakes up the listening server process and establishes the connection. The connection stays up until it is terminated by one of the parties.

When you began the telnet session with `time-a.nist.gov` at port 13, a piece of network software knew enough to convert the string `"time-a.nist.gov"` to its correct Internet Protocol (IP) address, 129.6.15.28. The telnet software then sent a connection request to that address, asking for a connection to port 13. Once the connection was established, the remote program sent back a line of data and closed the connection. In general, of course, clients and servers engage in a more extensive dialog before one or the other closes the connection.

Here is another experiment along the same lines—but a bit more interesting. Type

http://www.rejoiceblog.com/

```
telnet horstmann.com 80
```

Then type very carefully the following:

```
GET / HTTP/1.1

Host: horstmann.com

blank line
```

That is, hit the Enter key twice at the end.

Figure 4.3 shows the response. It should look eerily familiar—you got a page of HTML-formatted text, namely Cay Horstmann's home page.

```
Terminal

~$ telnet horstmann.com 80
Trying 67.210.118.65...
Connected to horstmann.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: horstmann.com

HTTP/1.1 200 OK
Date: Sun, 10 Apr 2016 04:36:27 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/0.9.8e-fips-rhel5 mod_auth_p
assthrough/2.1 mod_bwlimited/1.4 mod_fcgid/2.3.6 Sun-ONE-ASP/4.0.3
Last-Modified: Thu, 17 Mar 2016 18:32:18 GMT
ETag: "2590e1c-1c47-52e42d9a8f080"
Accept-Ranges: bytes
Content-Length: 7239
Content-Type: text/html

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/x
html1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
  <title>Cay Horstmann's Home Page</title>
  <link href="styles.css" rel="stylesheet" type="text/css"/>
```

Figure 4.3 Using telnet to access an HTTP port

This is exactly the same process that your web browser goes through to get a web page. It uses HTTP to request web pages from servers. Of course, the browser displays the HTML code more nicely.

http://www.rejoiceblog.com/

Note

The `Host` key/value pair is required when you connect to a web server that hosts multiple domains with the same IP address. You can omit it if the server hosts a single domain.

4.1.2 Connecting to a Server with Java

Our first network program in Listing 4.1 will do the same thing we did using telnet—connect to a port and print out what it finds.

Listing 4.1 `socket/SocketTest.java`

```
package socket;

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

/**
 * This program makes a socket connection to the atomic
 * the time that the server sends.
 *
 * @version 1.22 2018-03-17
```

http://www.rejoiceblog.com/

```
 * @author Cay Horstmann
 */
public class SocketTest
{
    public static void main(String[] args) throws IOExc
    {
        try (var s = new Socket("time-a.nist.gov", 13);
            var in = new Scanner(s.getInputStream(), Stan
        {
            while (in.hasNextLine())
            {
                String line = in.nextLine();
                System.out.println(line);
            }
        }
    }
}
```

The key statements of this simple program are these:

```
var s = new Socket("time-a.nist.gov", 13);
```

http://www.rejoiceblog.com/

```
InputStream inStream = s.getInputStream();
```

The first line opens a *socket*, which is a network software abstraction that enables communication out of and into this program. We pass the remote address and the port number to the socket constructor. If the connection fails, an `UnknownHostException` is thrown. If there is another problem, an `IOException` occurs. Since `UnknownHostException` is a subclass of `IOException` and this is a sample program, we just catch the superclass.

Once the socket is open, the `getInputStream` method in `java.net.Socket` returns an `InputStream` object that you can use just like any other stream. Once you have grabbed the stream, this program simply prints each input line to standard

output. This process continues until the stream is finished and the server disconnects.

This program works only with very simple servers, such as a "time of day" service. In more complex networking programs, the client sends request data to the server, and the server might not immediately disconnect at the end of a response. You will see how to implement that behavior in several examples throughout this chapter.

The `Socket` class is pleasant and easy to use because the Java library hides the complexities of establishing a networking connection and sending data across it. The `java.net` package essentially gives you the same programming interface you would use to work with a file.

Note

In this book, we cover only the Transmission Control Protocol (TCP). The Java platform also supports the User Datagram Protocol (UDP), which can be used to send packets (also called *datagrams*) with much less overhead than TCP. The drawback is that packets need not be delivered in sequential order to the receiving application and can even be dropped altogether. It is up to the recipient to put the packets in order and to request retransmission of missing packets. UDP is well suited for applications in which missing packets can be tolerated—for example, for audio or video streams or continuous measurements.

http://www.rejoiceblog.com/

java.net.Socket 1.0

- `Socket(String host, int port)`

constructs a socket to connect to the given host and port.

- `InputStream getInputStream()`
- `OutputStream getOutputStream()`

gets the stream to read data from the socket or write data to the socket.

4.1.3 Socket Timeouts

Reading from a socket blocks until data are available. If the host is unreachable, your application waits for a long time and you are at the mercy of the underlying operating system to eventually time out.

You can decide what timeout value is reasonable for your particular application. Then, call the `setSoTimeout` method to set a timeout value (in milliseconds).

```
var s = new Socket(. . .);
s.setSoTimeout(10000); // time out after 10 seconds
```

If the timeout value has been set for a socket, all subsequent read operations throw a `SocketTimeoutException` when the timeout has been reached before the operation has completed its work. You can catch that exception and react to the timeout.

```
try {
   InputStream in = s.getInputStream(); // read from in
   . . .
}
```

http://www.rejoiceblog.com/

```
catch (SocketTimeoutException e)
{
    react to timeout
}
```

There is no timeout for write operations.

There is one additional timeout issue that you need to address. The constructor

```
Socket(String host, int port)
```

can block indefinitely until an initial connection to the host is established.

You can overcome this problem by first constructing an unconnected socket and then connecting it with a timeout:

```
var s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

See Section 4.2.4, "Interruptible Sockets," on p. 244 for how to allow users to interrupt the socket connection at any time.

java.net.Socket 1.0

- `Socket()` **1.1**

creates a socket that has not yet been connected.

- `void connect(SocketAddress address)` **1.4**

connects this socket to the given address.

- `void connect(SocketAddress address, int timeoutInMilliseconds)`

http://www.rejoiceblog.com/

1.4

connects this socket to the given address, or returns if the time interval expired.

- `void setSoTimeout(int timeoutInMilliseconds)` 1.1

sets the blocking time for read requests on this socket. If the timeout is reached, a `SocketTimeoutException` is raised.

- `boolean isConnected()` 1.4

returns `true` if the socket is connected.

- `boolean isClosed()` 1.4

returns `true` if the socket is closed.

4.1.4 Internet Addresses

Usually, you don't have to worry too much about Internet addresses—the numerical host addresses that consist of 4 bytes (or, with IPv6, 16 bytes) such as 129.6.15.28. However, you can use the `InetAddress` class if you need to convert between host names and Internet addresses.

The `java.net` package supports IPv6 Internet addresses, provided the host operating system does.

The static `getByName` method returns an `InetAddress` object of a host. For example,

```
InetAddress address = InetAddress.getByName("time-a.nist.gov"
```

returns an `InetAddress` object that encapsulates the sequence of four bytes 129.6.15.28. You can access the bytes with the `getAddress` method.

```
byte[] addressBytes = address.getAddress();
```

Some host names with a lot of traffic correspond to multiple Internet addresses, to facilitate load balancing. For example, at the time of this writing, the host name

http://www.rejoiceblog.com/

`google.com` corresponds to twelve different Internet addresses. One of them is picked at random when the host is accessed. You can get all hosts with the `getAllByName` method.

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Finally, you sometimes need the address of the local host. If you simply ask for the address of `localhost`, you always get the local loopback address 127.0.0.1, which cannot be used by others to connect to your computer. Instead, use the static `getLocalHost` method to get the address of your local host.

```
InetAddress address = InetAddress.getLocalHost();
```

Listing 4.2 is a simple program that prints the Internet address of your local host if you do not specify any command-line parameters, or all Internet addresses of another host if you specify the host name on the command line, such as

```
java inetAddress/InetAddressTest www.horstmann.com
```

Listing 4.2 `inetAddress/InetAddressTest.java`

```
1   package inetAddress;

2

3   import java.io.*;

4   import java.net.*;

5

6   /**

7    * This program demonstrates the InetAddress class. Supp
```

http://www.rejoiceblog.com/

```
 8    * run without command-line arguments to see the address
 9    * @version 1.02 2012-06-05
10    * @author Cay Horstmann
11    */
12   public class InetAddressTest
13   {
14       public static void main(String[] args) throws IOExc
15       {
16          if (args.length > 0)
17          {
18             String host = args[0];
19             InetAddress[] addresses = InetAddress.getAllB
20             for (InetAddress a : addresses)
21                System.out.println(a);
22          }
23          else
24          {
25             InetAddress localHostAddress = InetAddress.ge
26             System.out.println(localHostAddress);
27          }
```

http://www.rejoiceblog.com/

```
        }
    }
```

java.net.InetAddress 1.0

- `static InetAddress getByName(String host)`

- `static InetAddress[] getAllByName(String host)`

constructs an `InetAddress`, or an array of all Internet addresses, for the given host name.

- `static InetAddress getLocalHost()`

constructs an `InetAddress` for the local host.

- `byte[] getAddress()`

returns an array of bytes that contains the numerical address.

- `String getHostAddress()`

returns a string with decimal numbers, separated by periods, for example `"129.6.15.28"`.

- `String getHostName()`

returns the host name.

4.2 IMPLEMENTING SERVERS

Now that we have implemented a basic network client that receives data from the Internet, let's program a simple server that can send information to clients. In the previous section, we have implemented a basic network client that receives data from the Internet. In the following sections, we will program a simple server that can send information to clients.

http://www.rejoiceblog.com/

4.2.1 Server Sockets

A server program, when started, waits for a client to attach to its port. For our example program, we chose port number 8189, which is not used by any of the standard services. The `ServerSocket` class establishes a socket. In our case, the command

```
var s = new ServerSocket(8189);
```

establishes a server that monitors port 8189. The command

```
Socket incoming = s.accept();
```

tells the program to wait indefinitely until a client connects to that port. Once someone connects to this port by sending the correct request over the network, this method returns a `Socket` object that represents the connection that was made. You can use this object to get input and output streams, as is shown in the following code:

```
InputStream inStream = incoming.getInputStream();

OutputStream outStream = incoming.getOutputStream();
```

Everything that the server sends to the server output stream becomes the input of the client program, and all the output from the client program ends up in the server input stream.

In all the examples in this chapter, we transmit text through sockets. We therefore turn the streams into scanners and writers.

```
var in = new Scanner(inStream, StandardCharsets.UTF_8);

var out = new PrintWriter(new OutputStreamWriter(outStream, S

   true /* autoFlush */);
```

Let's send the client a greeting:

http://www.rejoiceblog.com/

```
out.println("Hello! Enter BYE to exit.");
```

When you use telnet to connect to this server program at port 8189, you will see this greeting on the terminal screen.

In this simple server, we just read the client's input, a line at a time, and echo it. This demonstrates that the program receives the input. An actual server would obviously compute and return an answer depending on the input.

```
String line = in.nextLine();

out.println("Echo: " + line);

if (line.trim().equals("BYE")) done = true;
```

In the end, we close the incoming socket.

```
incoming.close();
```

That is all there is to it. Every server program, such as an HTTP web server, continues performing this loop:

1. It receives a command from the client ("get me this information") through an incoming data stream.

2. It decodes the client command.

3. It gathers the information that the client requested.

4. It sends the information to the client through the outgoing data stream.

Listing 4.3 is the complete program.

Listing 4.3 `server/EchoServer.java`

```
package server;
```

http://www.rejoiceblog.com/

```
import   java.io.*;
import   java.net.*;
import   java.nio.charset.*;
import   java.util.*;

/**
  * This program implements a simple server that listens
  * input.
  * @version 1.22 2018-03-17
  * @author Cay Horstmann
  */
public class EchoServer
{
      public static void main(String[] args) throws IOExc
      {
         // establish server socket
         try (var s = new ServerSocket(8189))
         {
            // wait for client connection
```

http://www.rejoiceblog.com/

```
22                 try (Socket incoming = s.accept())
23                 {
24                    InputStream inStream = incoming.getInputS
25                    OutputStream outStream = incoming.getOutp
26
27                     try (var in = new Scanner(inStream, Stan
28                     {
29                        var out = new PrintWriter(
30                           new OutputStreamWriter(outStream,
31                           true /* autoFlush */);
32
33                        out.println("Hello! Enter BYE to exi
34
35                        // echo client input
36                        var done = false;
37                        while (!done && in.hasNextLine())
38                        {
39                           String line = in.nextLine();
40                           out.println("Echo: " + line);
```

http://www.rejoiceblog.com/

```
                        if (line.trim().equals("BYE")) do
                    }
                }
            }
        }
    }
```

To try it out, compile and run the program. Then use telnet to connect to the server `localhost` (or IP address 127.0.0.1) and port 8189.

If you are connected directly to the Internet, anyone in the world can access your echo server, provided they know your IP address and the magic port number.

When you connect to the port, you will see the message shown in Figure 4.4:

```
Hello! Enter BYE to exit.
```

http://www.rejoiceblog.com/

```
Terminal
File Edit View Terminal Tabs Help
~$ telnet localhost 8189
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello! Enter BYE to exit.
Hello Sailor!
Echo: Hello Sailor!
BYE
Echo: BYE
Connection closed by foreign host.
~$
```

Figure 4.4 Accessing an echo server

Type anything and watch the input echo on your screen. Type `BYE` (all upper-case letters) to disconnect. The server program will terminate as well.

java.net.ServerSocket 1.0

- `ServerSocket(int port)`

creates a server socket that monitors a port.

- `Socket accept()`

waits for a connection. This method blocks (i.e., idles) the current thread until the connection is made. The method returns a `Socket` object through which the program can communicate with the connecting client.

- `void close()`

closes the server socket.

4.2.2 Serving Multiple Clients

http://www.rejoiceblog.com/

There is one problem with the simple server in the preceding example. Suppose we want to allow multiple clients to connect to our server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet might want to use it at the same time. Rejecting multiple connections allows any one client to monopolize the service by connecting to it for a long time. We can do much better through the magic of threads.

Every time we know the program has established a new socket connection—that is, every time the call to `accept()` returns a socket—we will launch a new thread to take care of the connection between the server and *that* client. The main program will just go back and wait for the next connection. For this to happen, the main loop of the server should look like this:

```
while (true)
{
   Socket incoming = s.accept();
   var r = new ThreadedEchoHandler(incoming);


   var t = new Thread(r);
   t.start();
}
```

The `ThreadedEchoHandler` class implements `Runnable` and contains the communication loop with the client in its `run` method.

```
class ThreadedEchoHandler implements Runnable
{
   . . .
```

http://www.rejoiceblog.com/

```
    public void run()
    {
      try (InputStream inStream = incoming.getInputStream();
        OutputStream outStream = incoming.getOutputStream())
      {
        Process input and send response
      }
      catch(IOException e)
      {
        Handle exception
      }
    }
  }
```

When each connection starts a new thread, multiple clients can connect to the server at the same time. You can easily check this out.

1. Compile and run the server program (Listing 4.4).

2. Open several telnet windows as we have in Figure 4.5.

3. Switch between windows and type commands. Note that you can communicate through all of them simultaneously.

4. When you are done, switch to the window from which you launched the server program and press Ctrl+C to kill it.

http://www.rejoiceblog.com/

Figure 4.5 Several telnet windows communicating simultaneously

Note

In this program, we spawn a separate thread for each connection. This approach is not satisfactory for high-performance servers. You can achieve greater server throughput by using features of the `java.nio` package. See `www.ibm.com/developerworks/java/library/j-javaio` for more information.

Listing 4.4 `threaded/ThreadedEchoServer.java`

```
1   package threaded;

2
```

http://www.rejoiceblog.com/

```
import  java.io.*;
import  java.net.*;
import  java.nio.charset.*;
import  java.util.*;

/**
 * This program implements a multithreaded server that l
 * all client input.
 * @author Cay Horstmann
 * @version 1.23 2018-03-17
 */
public class ThreadedEchoServer
{
      public static void main(String[] args )
      {
         try (var s = new ServerSocket(8189))
         {
            int i = 1;

            while (true)
```

http://www.rejoiceblog.com/

```
            {
                Socket incoming = s.accept();
                System.out.println("Spawning " + i);
                Runnable r = new ThreadedEchoHandler(incom
                var t = new Thread(r);
                t.start();
                i++;
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

/**
  * This class handles the client input for one server s
  */
```

http://www.rejoiceblog.com/

```
class ThreadedEchoHandler implements Runnable
{
     private Socket incoming;

    /**
        Constructs a handler.
        @param incomingSocket the incoming socket
    */
    public ThreadedEchoHandler(Socket incomingSocket)
    {
       incoming = incomingSocket;
    }

    public void run()
    {
       try (InputStream inStream = incoming.getInputStre
          OutputStream outStream = incoming.getOutputStr
          var in = new Scanner(inStream, StandardCharset
            var out = new PrintWriter(
               new OutputStreamWriter(outStream, Standar
```

http://www.rejoiceblog.com/

```
                true /* autoFlush */))
            {
                out.println( "Hello! Enter BYE to exit."

                // echo client input
                var done = false;
                while (!done && in.hasNextLine())
                {
                    String line = in.nextLine();
                    out.println("Echo: " + line);
                    if (line.trim().equals("BYE"))
                        done = true;
                }
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
```

http://www.rejoiceblog.com/

```
}
```

4.2.3 Half-Close

The *half-close* allows one end of a socket connection to terminate its output while still receiving data from the other end.

Here is a typical situation. Suppose you transmit data to the server but you don't know at the outset how much data you have. With a file, you'd just close the file at the end of the data. However, if you close a socket, you immediately disconnect from the server and cannot read the response.

The half-close overcomes this problem. You can close the output stream of a socket, thereby indicating to the server the end of the requested data, but keep the input stream open.

The client side looks like this:

```
try (var socket = new Socket(host, port))

{

   var in = new Scanner(socket.getInputStream(), StandardChar

   var writer = new PrintWriter(socket.getOutputStream());

   // send request data

   writer.print(. . .);

   writer.flush();

   socket.shutdownOutput();

   // now socket is half-closed

   // read response data
```

http://www.rejoiceblog.com/

```
      while (in.hasNextLine() != null) { String line = in.nextLi

   }
```

The server side simply reads input until the end of the input stream is reached. Then it sends the response.

Of course, this protocol is only useful for one-shot services such as HTTP where the client connects, issues a request, catches the response, and then disconnects.

java.net.Socket 1.0

- `void shutdownOutput()` 1.3

sets the output stream to "end of stream."

- `void shutdownInput()` 1.3

sets the input stream to "end of stream."

- `boolean isOutputShutdown()` 1.4

returns `true` if output has been shut down.

- `boolean isInputShutdown()` 1.4

returns `true` if input has been shut down.

4.2.4 Interruptible Sockets

When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed. Similarly, when you read data through a socket, the current thread blocks until the operation is successful or has timed out. (There is no timeout for writing data.)

In interactive applications, you would like to give users an option to simply cancel a socket connection that does not appear to produce results. However, if a thread blocks on an unresponsive socket, you cannot unblock it by calling `interrupt`.

http://www.rejoiceblog.com/

To interrupt a socket operation, use a `SocketChannel`, a feature of the `java.nio` package. Open the `SocketChannel` like this:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddr
```

A channel does not have associated streams. Instead, it has `read` and `write` methods that make use of `Buffer` objects. (See Chapter 1 for more information about NIO buffers.) These methods are declared in the interfaces `ReadableByteChannel` and `WritableByteChannel`.

If you don't want to deal with buffers, you can use the `Scanner` class to read from a `SocketChannel` because `Scanner` has a constructor with a `ReadableByteChannel` parameter:

```
var in = new Scanner(channel, StandardCharsets.UTF_8);
```

To turn a channel into an output stream, use the static `Channels.newOutputStream` method.

```
OutputStream outStream = Channels.newOutputStream(channel);
```

That's all you need to do. Whenever a thread is interrupted during an open, read, or write operation, the operation does not block, but is terminated with an exception.

The program in Listing 4.5 contrasts interruptible and blocking sockets. A server sends numbers and pretends to be stuck after the tenth number. Click on either button, and a thread is started that connects to the server and prints the output. The first thread uses an interruptible socket; the second thread uses a blocking socket. If you click the Cancel button within the first ten numbers, you can interrupt either thread.

However, after the first ten numbers, you can only interrupt the first thread. The second thread keeps blocking until the server finally closes the connection (see Figure 4.6).

http://www.rejoiceblog.com/

Figure 4.6 Interrupting a socket

Listing 4.5 `interruptible/InterruptibleSocketTest.java`

```
package interruptible;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;
import java.nio.charset.*;
import java.nio.channels.*;
import javax.swing.*;

```

http://www.rejoiceblog.com/

```
/**
 * This program shows how to interrupt a socket channel.
 * @author Cay Horstmann
 * @version 1.05 2018-03-17
 */
public class InterruptibleSocketTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(() ->
            {
                var frame = new InterruptibleSocketFrame(
                frame.setTitle("InterruptibleSocketTest")
                frame.setDefaultCloseOperation(JFrame.EXI
                frame.setVisible(true);
            });
    }
}

class InterruptibleSocketFrame extends JFrame
```

http://www.rejoiceblog.com/

```
32   {
33      private Scanner in;
34      private JButton interruptibleButton;
35      private JButton blockingButton;
36      private JButton cancelButton;
37      private JTextArea messages;
38      private TestServer server;
39      private Thread connectThread;
40
41      public InterruptibleSocketFrame()
42      {
43         var northPanel = new JPanel();
44         add(northPanel, BorderLayout.NORTH);
45
46         final int TEXT_ROWS = 20;
47         final int TEXT_COLUMNS = 60;
48         messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
49        add(new JScrollPane(messages));
50
```

http://www.rejoiceblog.com/

```
        interruptibleButton = new JButton("Interruptible");
        blockingButton = new JButton("Blocking");

        northPanel.add(interruptibleButton);
        northPanel.add(blockingButton);

        interruptibleButton.addActionListener(event ->
           {
               interruptibleButton.setEnabled(false);
               blockingButton.setEnabled(false);
               cancelButton.setEnabled(true);
               connectThread = new Thread(() ->
                  {
                      try
                      {
                          connectInterruptibly();
                      }
                      catch (IOException e)
                      {
                          messages.append("\nInterruptibleS
```

http://www.rejoiceblog.com/

```
                    }
                });
            connectThread.start();
        });

        blockingButton.addActionListener(event ->
        {
            interruptibleButton.setEnabled(false);
            blockingButton.setEnabled(false);
            cancelButton.setEnabled(true);
            connectThread = new Thread(() ->
                {
                    try
                    {
                        connectBlocking();
                    }
                    catch (IOException e)
                    {
                        messages.append("\nInterruptibleS
```

http://www.rejoiceblog.com/

```
                    }
                });
            connectThread.start();
        });

        cancelButton = new JButton("Cancel");
        cancelButton.setEnabled(false);
        northPanel.add(cancelButton);
        cancelButton.addActionListener(event ->
            {
                connectThread.interrupt();
                cancelButton.setEnabled(false);
            });
        server = new TestServer();
        new Thread(server).start();
        pack();
    }

    /**
     * Connects to the test server, using interruptible I
```

http://www.rejoiceblog.com/

```
     */
    public void connectInterruptibly() throws IOException
    {
        messages.append("Interruptible:\n");
        try (SocketChannel channel = SocketChannel.open(
        {
            in = new Scanner(channel, StandardCharsets.UT
            while (!Thread.currentThread().isInterrupted(
            {
                messages.append("Reading ");
                if (in.hasNextLine())
                {
                    String line = in.nextLine();
                    messages.append(line);
                    messages.append("\n");
                }
            }
        }
        finally
```

http://www.rejoiceblog.com/

```
        {
            EventQueue.invokeLater(() ->
                {
                    messages.append("Channel closed\n");
                    interruptibleButton.setEnabled(true);
                    blockingButton.setEnabled(true);
                });
        }
    }

    /**
     * Connects to the test server, using balocking I/O
     */
    public void connectBlocking() throws IOException
    {
        messages.append("Blocking:\n");
        try (var sock = new Socket("localhost", 8189))
        {
            in = new Scanner(sock.getInputStream(), Standa
            while (!Thread.currentThread().isInterrupted()
```

http://www.rejoiceblog.com/

```
            {
                messages.append("Reading ");
                if (in.hasNextLine())
                {
                    String line = in.nextLine();
                    messages.append(line);
                    messages.append("\n");
                }
            }
        }
        finally
        {
            EventQueue.invokeLater(() ->
                {
                    messages.append("Socket closed\n");
                    interruptibleButton.setEnabled(true);
                    blockingButton.setEnabled(true);
                });
        }
```

http://www.rejoiceblog.com/

```
    }

    /**
     * A multithreaded server that listens to port 8189 a
     * a hanging server after 10 numbers.
     */
    class TestServer implements Runnable
    {
        public void run()
        {
            try (var s = new ServerSocket(8189))
            {
                while (true)
                {
                    Socket incoming = s.accept();
                    Runnable r = new TestServerHandler(inco
                    new Thread(r).start();
                }
            }
            catch (IOException e)
```

http://www.rejoiceblog.com/

```
            {
               messages.append("\nTestServer.run: " + e);
            }
         }
      }

      /**
       * This class handles the client input for one serv
       */
      class TestServerHandler implements Runnable
      {
           private Socket incoming;
           private int counter;

          /**
           * Constructs a handler.
           * @param i the incoming socket
           */
          public TestServerHandler(Socket i)
```

http://www.rejoiceblog.com/

```
207             {
208                 incoming = i;
209             }
210
211             public void run()
212             {
213                 try
214                 {
215                     try
216                     {
217                         OutputStream outStream = incoming.ge
218                         var out = new PrintWriter(
219                             new OutputStreamWriter(outStream,
220                             true /* autoFlush */);
221                         while (counter < 100)
222                         {
223                             counter++;
224                             if (counter <= 10) out.println(co
225                             Thread.sleep(100);
226                         }
```

http://www.rejoiceblog.com/

```
                    }
                    finally
                    {
                        incoming.close();
                        messages.append("Closing server\n");
                    }
                }
                catch (Exception e)
                {
                    messages.append("\nTestServerHandler.run
                }
            }
        }
    }
```

java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`

constructs an address object with the given host and port, resolving the host name during construction. If the host name cannot be resolved, the address object's `unresolved` property is set to `true`.

http://www.rejoiceblog.com/

- `boolean isUnresolved()`

returns `true` if this address object could not be resolved.

java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress address)`

opens a socket channel and connects it to a remote address.

java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel channel)`

constructs an input stream that reads from the given channel.

- `static OutputStream newOutputStream(WritableByteChannel channel)`

constructs an output stream that writes to the given channel.

4.3 GETTING WEB DATA

To access web servers in a Java program, you will want to work at a higher level than socket connections and HTTP requests. In the following sections, we discuss the classes that the Java library provides for this purpose.

4.3.1 URLs and URIs

The `URL` and `URLConnection` classes encapsulate much of the complexity of retrieving information from a remote site. You can construct a `URL` object from a string:

```
var url = new URL(urlString);
```

If you simply want to fetch the contents of the resource, use the `openStream` method of the `URL` class. This method yields an `InputStream` object. Use it in the usual way—for example, to construct a `Scanner`:

http://www.rejoiceblog.com/

```
InputStream inStream = url.openStream(); var in = new Scanner
```

The `java.net` package makes a useful distinction between URLs (uniform resource *locators*) and URIs (uniform resource *identifiers*).

A URI is a purely syntactical construct that contains the various parts of the string specifying a web resource. A URL is a special kind of URI, namely, one with sufficient information to *locate* a resource. Other URIs, such as

```
mailto:cay@horstmann.com
```

are not locators—there is no data to locate from this identifier. Such a URI is called a URN (uniform resource *name*).

In the Java library, the `URI` class has no methods for accessing the resource that the identifier specifies—its sole purpose is parsing. In contrast, the `URL` class can open a stream to the resource. For that reason, the `URL` class only works with schemes that the Java library knows how to handle, such as `http:`, `https:`, `ftp:`, the local file system (`file:`), and JAR files (`jar:`).

To see why parsing is not trivial, consider how complex URIs can be. For example,

```
http:/google.com?q=Beach+Chalet

ftp://username:password@ftp.yourserver.com/pub/file.txt
```

The URI specification gives the rules for the makeup of these identifiers. A URI has the syntax

`[`*scheme*`:]`*schemeSpecificPart*`[#`*fragment*`]`

Here, the `[. . .]` denotes an optional part, and the `:` and `#` are included literally in the identifier.

If the *scheme*`:` part is present, the URI is called *absolute*. Otherwise, it is called *relative*.

http://www.rejoiceblog.com/

An absolute URI is *opaque* if the *schemeSpecificPart* does not begin with a `/` such as

```
mailto:cay@horstmann.com
```

All absolute nonopaque URIs and all relative URIs are *hierarchical*. Examples are

```
http://horstmann.com/index.html

../../java/net/Socket.html#Socket()
```

The *schemeSpecificPart* of a hierarchical URI has the structure

`[//`*authority*`][`*path*`][?`*query*`]`

where, again, `[. . .]` denotes optional parts.

For server-based URIs, the *authority* part has the form

`[`*user-info*`@]`*host*`[:`*port*`]`

The *port* must be an integer.

RFC 2396, which standardizes URIs, also supports a registry-based mechanism in which the *authority* has a different format, but this is not in common use.

One of the purposes of the `URI` class is to parse an identifier and break it up into its components. You can retrieve them with the methods

```
getScheme

getSchemeSpecificPart

getAuthority

getUserInfo

getHost
```

http://www.rejoiceblog.com/

```
getPort

getPath

getQuery

getFragment
```

The other purpose of the `URI` class is the handling of absolute and relative identifiers. If you have an absolute URI such as

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

and a relative URI such as

```
../../java/net/Socket.html#Socket()
```

then you can combine the two into an absolute URI.

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

This process is called *resolving* a relative URL.

The opposite process is called *relativization*. For example, suppose you have a *base* URI

```
http://docs.mycompany.com/api
```

and a URI

```
http://docs.mycompany.com/api/java/lang/String.html
```

http://www.rejoiceblog.com/

Then the relativized URI is

```
java/lang/String.html
```

The URI class supports both of these operations:

```
relative = base.relativize(combined); combined = base.resolve
```

4.3.2 Using a URLConnection to Retrieve Information

If you want additional information about a web resource, you should use the `URLConnection` class, which gives you much more control than the basic `URL` class.

When working with a `URLConnection` object, you must carefully schedule your steps.

1. Call the `openConnection` method of the `URL` class to obtain the `URLConnection` object:

```
URLConnection connection = url.openConnection();
```

2. Set any request properties, using the methods

```
setDoInput

setDoOutput

setIfModifiedSince

setUseCaches

setAllowUserInteraction

setRequestProperty

setConnectTimeout
```

http://www.rejoiceblog.com/

```
setReadTimeout
```

We discuss these methods later in this section and in the API notes.

3. Connect to the remote resource by calling the `connect` method.

```
connection.connect();
```

Besides making a socket connection to the server, this method also queries the server for *header information*.

4. After connecting to the server, you can query the header information. Two methods, `getHeaderFieldKey` and `getHeaderField`, enumerate all fields of the header. The method `getHeaderFields` gets a standard `Map` object containing the header fields. For your convenience, the following methods query standard fields:

```
getContentType

getContentLength

getContentEncoding

getDate

getExpiration

getLastModified
```

5. Finally, you can access the resource data. Use the `getInputStream` method to obtain an input stream for reading the information. (This is the same input stream that the `openStream` method of the `URL` class returns.) The other method, `getContent`, isn't very useful in practice. The objects that are returned by standard content types such as `text/plain` and `image/gif` require classes in the `com.sun` hierarchy for processing. You could register your own content handlers, but we do not discuss this technique in our book.

http://www.rejoiceblog.com/

Caution

Some programmers form the wrong mental image when using the `URLConnection` class, thinking that the `getInputStream` and `getOutputStream` methods are similar to those of the `Socket` class. But that isn't quite true. The `URLConnection` class does quite a bit of magic behind the scenes—in particular, the handling of request and response headers. For that reason, it is important that you follow the setup steps for the connection.

Let us now look at some of the `URLConnection` methods in detail. Several methods set properties of the connection before connecting to the server. The most important ones are `setDoInput` and `setDoOutput`. By default, the connection yields an input stream for reading from the server but no output stream for writing. If you want an output stream (for example, for posting data to a web server), you need to call

```
connection.setDoOutput(true);
```

Next, you may want to set some of the request headers. The request headers are sent together with the request command to the server. Here is an example:

```
GET www.server.com/index.html HTTP/1.0

Referer: http://www.somewhere.com/links.html

Proxy-Connection: Keep-Alive

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.

Host: www.server.com

Accept: text/html, image/gif, image/jpeg, image/png, */*

Accept-Language: en

Accept-Charset: iso-8859-1,*,utf-8
```

http://www.rejoiceblog.com/

```
Cookie: orangemilano=192218887821987
```

The `setIfModifiedSince` method tells the connection that you are only interested in data modified since a certain date.

Finally, you can use the catch-all `setRequestProperty` method to set any name/value pair that is meaningful for the particular protocol. For the format of the HTTP request headers, see RFC 2616. Some of these parameters are not well documented and are passed around by word of mouth from one programmer to the next. For example, if you want to access a password-protected web page, you must do the following:

1. Concatenate the user name, a colon, and the password.

```
String input = username + ":" + password;
```

2. Compute the Base64 encoding of the resulting string. (The Base64 encoding encodes a sequence of bytes into a sequence of printable ASCII characters.)

```
Base64.Encoder encoder = Base64.getEncoder();

String encoding = encoder.encodeToString(input.getBytes(Stand
```

3. Call the `setRequestProperty` method with a name of `"Authorization"` and the value `"Basic " + encoding`:

```
connection.setRequestProperty("Authorization", "Basic " + enc
```

Tip

You just saw how to access a password-protected web page. To access a

http://www.rejoiceblog.com/

password-protected file by FTP, use an entirely different method: Construct a URL of the form

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Once you call the `connect` method, you can query the response header information. First, let's see how to enumerate all response header fields. The implementors of this class felt a need to express their individuality by introducing yet another iteration protocol. The call

```
String key = connection.getHeaderFieldKey(n);
```

gets the `n`th key from the response header, where `n` starts from `1`! It returns `null` if `n` is zero or greater than the total number of header fields. There is no method to return the number of fields; you simply keep calling `getHeaderFieldKey` until you get `null`. Similarly, the call

```
String value = connection.getHeaderField(n);
```

returns the `n`th value.

The method `getHeaderFields` returns a `Map` of response header fields.

```
Map<String,List<String>> headerFields = connection.getHeaderF
```

Here is a set of response header fields from a typical HTTP request:

```
Date: Wed, 27 Aug 2008 00:15:48 GMT

Server: Apache/2.2.2 (Unix)

Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
```

http://www.rejoiceblog.com/

```
Accept-Ranges: bytes

Content-Length: 4813

Connection: close

Content-Type: text/html
```

Note

You can obtain the response status line (such as `"HTTP/1.1 200 OK"`) as `connection.getHeaderField(0)` or `headerFields.get(null)`.

As a convenience, six methods query the values of the most common header types and convert them to numeric types when appropriate. Table 4.1 shows these convenience methods. The methods with return type `long` return the number of seconds since January 1, 1970 GMT.

Table 4.1 Convenience Methods for Response Header Values

Key Name	Method Name	Return Type
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

The program in Listing 4.6 lets you experiment with URL connections. Supply a URL and an optional user name and password on the command line when running the program, for example:

```
java urlConnection.URLConnectionTest http://www.yourserver.co
```

http://www.rejoiceblog.com/

The program prints

- All keys and values of the header
- The return values of the six convenience methods in Table 4.1
- The first ten lines of the requested resource

Listing 4.6 `urlConnection/URLConnectionTest.java`

```
package urlConnection;

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

/**
 * This program connects to an URL and displays the resp
 * the requested data.
 *
 * Supply the URL and an optional username and password
 * command line.
 * @version 1.12 2018-03-17
 * @author Cay Horstmann
```

http://www.rejoiceblog.com/

```
16     */
17   public class URLConnectionTest
18   {
19        public static void main(String[] args)
20        {
21           try
22           {
23              String urlName;
24              if (args.length > 0) urlName = args[0];
25              else urlName = "http://horstmann.com";
26
27            var url = new URL(urlName);
28            URLConnection connection = url.openConnection()
29
30            // set username, password if specified on comma
31
32            if (args.length > 2)
33            {
34               String username = args[1];
```

http://www.rejoiceblog.com/

```
            String password = args[2];
            String input = username + ":" + password;
            Base64.Encoder encoder = Base64.getEncoder()
            String encoding = encoder.encodeToString(inp
            connection.setRequestProperty("Authorization
        }

        connection.connect();

        // print header fields

        Map<String, List<String>> headers = connection.
        for (Map.Entry<String, List<String>> entry : he
        {
            String key = entry.getKey();
            for (String value : entry.getValue())
                System.out.println(key + ": " + value);
        }

        // print convenience functions
```

http://www.rejoiceblog.com/

```
55
56            System.out.println("----------");
57            System.out.println("getContentType: " + connect
58              System.out.println("getContentLength: " + con
59              System.out.println("getContentEncoding: " + c
60              System.out.println("getDate: " + connection.g
61              System.out.println("getExpiration: " + connec
62              System.out.println("getLastModifed: " + conne
63              System.out.println("----------");
64
65              String encoding = connection.getContentEncodi
66              if (encoding == null) encoding = "UTF-8";
67              try (var in = new Scanner(connection.getInput
68              {
69                 // print first ten lines of contents
70
71                 for (int n = 1; in.hasNextLine() && n <= 1
72                    System.out.println(in.nextLine());
73                 if (in.hasNextLine()) System.out.println("
```

http://www.rejoiceblog.com/

```
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

java.net.URL 1.0

- `InputStream openStream()`

opens an input stream for reading the resource data.

- `URLConnection openConnection();`

returns a `URLConnection` object that manages the connection to the resource.

java.net.URLConnection 1.0

- `void setDoInput(boolean doInput)`
- `boolean getDoInput()`

If `doInput` is `true`, the user can receive input from this `URLConnection`.

- `void setDoOutput(boolean doOutput)`
- `boolean getDoOutput()`

http://www.rejoiceblog.com/

If `doOutput` is `true`, the user can send output to this `URLConnection`.

- `void setIfModifiedSince(long time)`

- `long getIfModifiedSince()`

The `ifModifiedSince` property configures this `URLConnection` to fetch only data modified since a given time. The time is given in seconds since midnight, GMT, January 1, 1970.

- `void setConnectTimeout(int timeout)` **5.0**

- `int getConnectTimeout()` **5.0**

sets or gets the timeout for the connection (in milliseconds). If the timeout has elapsed before a connection was established, the `connect` method of the associated input stream throws a `SocketTimeoutException`.

- `void setReadTimeout(int timeout)` **5.0**

- `int getReadTimeout()` **5.0**

sets or gets the timeout for reading data (in milliseconds). If the timeout has elapsed before a read operation was successful, the `read` method throws a `SocketTimeoutException`.

- `void setRequestProperty(String key, String value)`

sets a request header field.

- `Map<String,List<String>> getRequestProperties()` **1.4**

returns a map of request properties. All values for the same key are placed in a list.

- `void connect()`

connects to the remote resource and retrieves response header information.

- `Map<String,List<String>> getHeaderFields()` **1.4**

returns a map of response headers. All values for the same key are placed in a list.

- `String getHeaderFieldKey(int n)`

http://www.rejoiceblog.com/

gets the key for the `n`th response header field, or `null` if `n` is ≠ `0` or greater than the number of response header fields.

- `String getHeaderField(int n)`

gets value of the `n`th response header field, or `null` if `n` is ≠ `0` or greater than the number of response header fields.

- `int getContentLength()`

gets the content length if available, or `-1` if unknown.

- `String getContentType()`

gets the content type, such as `text/plain` or `image/gif`.

- `String getContentEncoding()`

gets the content encoding, such as `gzip`. This value is not commonly used, because the default `identity` encoding is not supposed to be specified with a `Content-Encoding` header.

- `long getDate()`

- `long getExpiration()`

- `long getLastModifed()`

gets the date of creation, expiration, and last modification of the resource. The dates are specified as seconds since midnight, GMT, January 1, 1970.

- `InputStream getInputStream()`

- `OutputStream getOutputStream()`

returns a stream for reading from the resource or writing to the resource.

- `Object getContent()`

selects the appropriate content handler to read the resource data and convert it into an object. This method is not useful for reading standard types such as `text/plain` or `image/gif` unless you install your own content handler.

http://www.rejoiceblog.com/

4.3.3 Posting Form Data

In the preceding section, you saw how to read data from a web server. Now we will show you how your programs can send data back to a web server and to programs that the web server invokes.

To send information from a web browser to the web server, a user fills out a *form*, like the one in Figure 4.7.

Figure 4.7 An HTML form

When the user clicks the Submit button, the text in the text fields and the settings of any checkboxes, radio buttons, and other input elements are sent back to the web server. The web server invokes a program that processes the user input.

Many technologies enable web servers to invoke programs. Among the best known ones are Java servlets, JavaServer Faces, Microsoft Active Server Pages (ASP), and Common

http://www.rejoiceblog.com/

Gateway Interface (CGI) scripts.

The server-side program processes the form data and produces another HTML page that the web server sends back to the browser. This sequence is illustrated in Figure 4.8. The response page can contain new information (for ex-

ample, in an information-search program) or just an acknowledgment. The web browser then displays the response page.

Figure 4.8 Data flow during execution of a server-side program

We do not discuss the implementation of server-side programs in this book. Our interest is merely in writing client programs that interact with existing server-side programs.

When form data are sent to a web server, it does not matter whether the data are interpreted by a servlet, a CGI script, or some other server-side technology. The client sends the data to the web server in a standard format, and the web server takes care of passing it on to the program that generates the response.

Two commands, called `GET` and `POST`, are commonly used to send information to a web server.

In the `GET` command, you simply attach query parameters to the end of the URL. The

http://www.rejoiceblog.com/

URL has the form

`http://host/path?query`

Each parameter has the form *name=value*. Parameters are separated by `&` characters. Parameter values are encoded using the *URL encoding* scheme, following these rules:

• Leave the characters `A` through `Z`, `a` through `z`, `0` through `9`, and `. - ~ _` unchanged.

• Replace all spaces with `+` characters.

• Encode all other characters into UTF-8 and encode each byte by a `%`, followed by a two-digit hexadecimal number.

For example, to transmit *San Francisco, CA*, you use `San+Francisco%2c+CA`, as the hexadecimal number `2c` is the UTF-8 code of the `','` character.

This encoding keeps any intermediate programs from messing with spaces and interpreting other special characters.

At the time of this writing, the Google Maps site (`www.google.com/maps`) accepts query parameters with names `q` and `hl` whose values are the location query and the human language of the response. To get a map of 1 Market Street in San Francisco, with a response in German, use the following URL:

```
http://www.google.com/maps?q=1+Market+Street+San+Francisco&hl
```

Very long query strings can look unattractive in browsers, and older browsers and proxies have a limit on the number of characters that you can include in a `GET` request. For that reason, a `POST` request is often used for forms with a lot of data. In a `POST` request, you do not attach parameters to a URL. Instead, you get an output stream from the `URLConnection` and write name/value pairs to the output stream. You still have to URL-encode the values and separate them with `&` characters.

Let us look at this process in detail. To post data to a server-side program, first establish a `URLConnection`:

http://www.rejoiceblog.com/

```
var url = new URL("http://host/path");

URLConnection connection = url.openConnection();
```

Then, call the `setDoOutput` method to set up the connection for output:

```
connection.setDoOutput(true);
```

Next, call `getOutputStream` to get a stream through which you can send data to the server. If you are sending text to the server, it is convenient to wrap that stream into a `PrintWriter`.

```
var out = new PrintWriter(connection.getOutputStream(), Stand
```

Now you are ready to send data to the server:

```
out.print(name1 + "=" + URLEncoder.encode(value1, StandardCha

out.print(name2 + "=" + URLEncoder.encode(value2, StandardCha
```

Close the output stream.

```
out.close();
```

Finally, call `getInputStream` and read the server response.

Let's run through a practical example. The web site at `https://tools.usps.com/zipcodelookup/byaddress` contains a form to find the zip code for a street address (see Figure 4.7). To use this form in a Java program, you need to know the URL and the parameters of the `POST` request.

You could get that information by looking at the HTML code of the form, but it is usually easier to "spy" on a request with a network monitor. Most browsers have a

http://www.rejoiceblog.com/

network monitor as part of their development toolkit. For example, Figure 4.9 shows a screen capture of the Firefox network monitor when submitting data to our example web site. You can find out the submission URL as well as the parameter names and values.

Figure 4.9 Monitoring the submission of a form

When posting form data, the HTTP header includes the content type:

```
Content-Type: application/x-www-form-urlencoded
```

You can also post data in other formats. For example, when sending data in JavaScript Object Notation (JSON), set the content type to `application/json`.

The header for a `POST` must also include the content length, for example

```
Content-Length: 124
```

The program in Listing 4.7 sends `POST` form data to any server-side program. Place the data into a `.properties` file such as the following:

```
url=https://tools.usps.com/tools/app/ziplookup/zipByAddress
```

http://www.rejoiceblog.com/

```
User-Agent=HTTPie/0.9.2

address1=1 Market Street

address2=

city=San Francisco

state=CA

companyName=

. . .
```

The program removes the `url` and `User-Agent` entries and sends all others to the `doPost` method.

In the `doPost` method, we first open the connection and set the user agent. (The zip code service does not work with the default `User-Agent` request parameter which contains the string `Java`, perhaps because the postal service doesn't want to serve programmatic requests.)

Then we call `setDoOutput(true)`, and open the output stream. We then enumerate all keys and values. For each of them, we send the `key`, = character, `value`, and `&` separator character:

```
out.print(key);

out.print('=');

out.print(URLEncoder.encode(value, StandardCharsets.UTF_8));

if (more pairs) out.print('&');
```

When switching from writing to reading any part of the response, the actual interaction with the server happens. The `Content-Length` header is set to the size of the output.

http://www.rejoiceblog.com/

The `Content-Type` header is set to `application/x-www-form-urlencoded` unless a different content type was specified. The headers and data are sent to the server. Then the response headers and server response are read and can be queried. In our example program, this switch happens in the call to `connection.getContentEncoding()`.

There is one twist with reading the response. If a server-side error occurs, the call to `connection.getInputStream()` throws a `FileNotFoundException`. However, the server still sends an error page back to the browser (such as the ubiquitous "Error 404—page not found"). To capture this error page, call the `getErrorStream` method:

```
InputStream err = connection.getErrorStream();
```

Note

The `getErrorStream` method, as well as several other methods in this program, belong to the `HttpURLConnection` subclass of `URLConnection`. If you make a request to an URL that starts with `http://` or `https://`, you can cast the resulting connection object to `HttpURLConnection`.

When you send `POST` data to a server, it can happen that the server-side program responds with a *redirect*: a different URL that should be called to get the actual information. The server could do that because the information is available elsewhere, or to provide a bookmarkable URL. The `HttpURLConnection` class can handle redirects in most cases.

Note

If cookies need to be sent from one site to another in a redirect, you can configure the global cookie handler like this:

```
CookieHandler.setDefault(new CookieManager(null, CookiePolicy.ACCE
```

http://www.rejoiceblog.com/

Then cookies will be properly included in the redirect.

Even though redirects are usually automatically handled, there are some situations where you need to do them yourself. Automatic redirects between HTTP and HTTPS are not supported for security reasons. Redirects can also fail for more subtle reasons. For example, an earlier version of the zip code service used a redirect. Recall that we set the `User-Agent` request parameter so that the post office didn't think we made a request via the Java API. While it is possible to set the user agent to a different string in the initial request, that setting is not used in a automatic redirects. Instead, automatic redirects always send a generic user agent string that contains the word `Java`.

In such situations, you can manually carry out the redirects. Before connecting the server, turn off automatic redirects:

```
connection.setInstanceFollowRedirects(false);
```

After making the request, get the response code:

```
int responseCode = connection.getResponseCode();
```

Check if it is one of

```
HttpURLConnection.HTTP_MOVED_PERM

HttpURLConnection.HTTP_MOVED_TEMP

HttpURLConnection.HTTP_SEE_OTHER
```

In that case, get the `Location` response header to obtain the URL for the redirect. Then disconnect and make another connection to the new URL:

```
String location = connection.getHeaderField("Location");

if (location != null)
```

http://www.rejoiceblog.com/

```
{
   URL base = connection.getURL();
   connection.disconnect();
   connection = (HttpURLConnection) new URL(base, location).o
   . . .
}
```

The techniques that this program illustrates can be useful whenever you need to query information from an existing web site. Simply find out the parameters that you need to send, and then strip out the HTML tags and other unnecessary information from the reply.

Listing 4.7 `post/PostTest.java`

```
1   package post;
2
3   import  java.io.*;
4   import  java.net.*;
5   import  java.nio.charset.*;
6   import  java.nio.file.*;
7   import  java.util.*;
8
9   /**
```

http://www.rejoiceblog.com/

```
 * This program demonstrates how to use the URLConnectio
 * @version 1.42 2018-03-17
 * @author Cay Horstmann
 */
public class PostTest
{
    public static void main(String[] args) throws IOExc
    {
        String propsFilename = args.length > 0 ? args[0]
        var props = new Properties();
        try (InputStream in = Files.newInputStream(Paths
        {
            props.load(in);
        }
        String urlString = props.remove("url").toString(
        Object userAgent = props.remove("User-Agent");
        Object redirects = props.remove("redirects");
        CookieHandler.setDefault(new CookieManager(null,
        String result = doPost(new URL(urlString), props
```

http://www.rejoiceblog.com/

```
            userAgent == null ? null : userAgent.toString
            redirects == null ? -1 : Integer.parseInt(red
        System.out.println(result);
    }

    /**
     * Do an HTTP POST.
     * @param url the URL to post to
     * @param nameValuePairs the query parameters
     * @param userAgent the user agent to use, or null for
     * @param redirects the number of redirects to follow
     * @return the data returned from the server
     */
    public static String doPost(URL url, Map<Object, Objec
            throws IOException
    {
        var connection = (HttpURLConnection) url.openConn
        if (userAgent != null)
            connection.setRequestProperty("User-Agent", us

```

http://www.rejoiceblog.com/

```
        if (redirects >= 0)
            connection.setInstanceFollowRedirects(false);

        connection.setDoOutput(true);

        try (var out = new PrintWriter(connection.getOutpu
        {
            var first = true;
            for (Map.Entry<Object, Object> pair : nameValue
            {
                if (first) first = false;
                else out.print('&');
                String name = pair.getKey().toString();
                String value = pair.getValue().toString();
                out.print(name);
                out.print('=');
                out.print(URLEncoder.encode(value, StandardC
            }
        }
```

http://www.rejoiceblog.com/

```
68          String encoding = connection.getContentEncoding();
69          if (encoding == null) encoding = "UTF-8";
70
71          if (redirects > 0)
72          {
73             int responseCode = connection.getResponseCode()
74             if (responseCode == HttpURLConnection.HTTP_MOVE
75                   || responseCode == HttpURLConnection.HTTP
76                   || responseCode == HttpURLConnection.HTTP
77             {
78                String location = connection.getHeaderField(
79                   if (location != null)
80                   {
81                      URL base = connection.getURL();
82                      connection.disconnect();
83                      return doPost(new URL(base, location),
84                   }
85                }
86             }
87             else if (redirects == 0)
```

http://www.rejoiceblog.com/

```
            {
                throw new IOException("Too many redirects");
            }

            var response = new StringBuilder();
            try (var in = new Scanner(connection.getInputSt
            {
                while (in.hasNextLine())
                {
                    response.append(in.nextLine());
                    response.append("\n");
                }
            }
            catch (IOException e)
            {
                InputStream err = connection.getErrorStream(
                if (err == null) throw e;
                try (var in = new Scanner(err))
                {
```

http://www.rejoiceblog.com/

```
                response.append(in.nextLine());
                response.append("\n");
            }
        }

        return response.toString();
    }
}
```

java.net.HttpURLConnection 1.0

- `InputStream getErrorStream()`

returns a stream from which you can read web server error messages.

java.net.URLEncoder 1.0

- `static String encode(String s, String encoding)` 1.4

returns the URL-encoded form of the string `s`, using the given character encoding scheme. (The recommended scheme is `"UTF-8"`.) In URL encoding, the characters `'A'-'Z'`, `'a'-'z'`, `'0'-'9'`, `'-'`, `'_'`, `'.'`, and `'~'` are left unchanged. Space is encoded into `'+'`, and all other characters are encoded into sequences of encoded bytes of the form "%*XY*", where `0x`*XY* is the hexadecimal value of the byte.

java.net.URLDecoder 1.2

http://www.rejoiceblog.com/

- `static string decode(String s, String encoding)` 1.4

returns the decoding of the URL encoded string `s` under the given character encoding scheme.

4.4 THE HTTP CLIENT

The `URLConnection` class was designed before HTTP was the universal protocol of the web. It provides support for a number of protocols, but its HTTP support is somewhat cumbersome. When the decision was made to support HTTP/2, it became clear that it would be best to provide a modern client interface instead of reworking the existing API. The `HttpClient` provides a more convenient API and HTTP/2 support. In Java 9 and 10, the API classes are located in the `jdk.incubator.http` package, to give the API a chance to evolve as a result of user feedback. As of Java 11, the `HttpClient` is in the `java.net.http` package.

Note

With Java 9 and 10, you need to run your program with the command line option

```
--add-modules jdk.incubator.httpclient
```

The HTTP client API provides a simpler mechanism for connecting to a web server than the `URLConnection` class with its rather fussy set of stages.

An `HttpClient` can issue requests and receive responses. You get a client by calling

```
HttpClient client = HttpClient.newHttpClient()
```

Alternatively, if you need to configure the client, you use a builder API, like this:

```
HttpClient client = HttpClient.newBuilder()
```

http://www.rejoiceblog.com/

```
    .followRedirects(HttpClient.Redirect.ALWAYS)

    .build();
```

That is, you get a builder, call methods to customize the item that is going to be built, and then call the `build` method to finalize the building process. This is a common pattern for constructing immutable objects.

You also follow the builder pattern for formulating requests. Here is a `GET` re-quest:

```
HttpRequest request = HttpRequest.newBuilder()

    .uri(new URI("http://horstmann.com"))

    .GET()

    .build();
```

The URI is the “uniform resource identifier” which is, when using HTTP, the same as a URL. However, in Java, the `URL` class has methods for actually opening a connection to a URL, whereas the `URI` class is only concerned with the syntax (scheme, host, port, path, query, fragment, and so on).

With a `POST` request, you need a “body publisher” that turns the request data into the data that is being posted. There are body publishers for strings, byte arrays, and files. For example, if your request is in JSON, you just provide the JSON string to a string body publisher.

```
HttpRequest request = HttpRequest.newBuilder()

    .uri(new URI(url))

    .header("Content-Type", "application/json")

    .POST(HttpRequest.BodyPublishers.ofString(jsonString))
```

http://www.rejoiceblog.com/

```
    .build();
```

It is unfortunate that the API does not support the required formatting for common content types. The sample program in Listing 4.8 provides body publishers for form data and file uploads.

When sending the request, you have to tell the client how to handle the response. If you just want the body as a string, send the request with a `HttpResponse.BodyHandler.asString()`, like this:

```
HttpResponse<String> response = client.send(request, HttpResp
```

The `HttpResponse` class is a generic type whose type parameter denotes the type of the body. You get the response body string simply as

```
String bodyString = response.body();
```

There are other response body handlers that get the response as a byte array or input stream. `BodyHandlers.ofFile(filePath)` yields a handler that saves the response to the given file, and `BodyHandlers.ofFileDownload(directoryPath)` saves the response in the given directory, using the file name from the `Content-Disposition` header. Finally, the handler obtained from `BodyHandler.discarding()` simply discards the response.

Processing the contents of the response is not considered part of the API. For example, if you receive JSON data, you need a JSON library to parse the contents.

The `HttpResponse` object also yields the status code and the response headers.

```
int status = response.statusCode();

HttpHeaders responseHeaders = response.headers();
```

You can turn the `HttpHeaders` object into a map:

http://www.rejoiceblog.com/

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

The map values are lists since in HTTP, each key can have multiple values.

If you just want the value of a particular key, and you know that there won't be multiple values, then call the `firstValue` method:

```
Optional<String> lastModified = headerMap.firstValue("Last-Mo
```

You get the response value or an empty optional if none was supplied.

You can process the response asynchronously. When building the client, provide an executor:

```
ExecutorService executor = Executors.newCachedThreadPool();

HttpClient client = HttpClient.newBuilder().executor(executor
```

Build a request and then invoke the `sendAsync` method on the client. You receive a `CompletableFuture<HttpResponse<T>>`, where `T` is the type of the body handler. Use the `CompletableFuture` API as described in Chapter 12 of Volume I:

```
HttpRequest request = HttpRequest.newBuilder().uri(uri).GET()

client.sendAsync(request, HttpResponse.BodyHandler.asString()

   thenAccept(response -> . . .);
```

Tip

To enable logging for the `HttpClient`, add this line to `net.properties` in

http://www.rejoiceblog.com/

your JDK:

```
jdk.httpclient.HttpClient.log=all
```

Instead of `all`, you can specify a comma-separated list of `headers`, `requests`, `content`, `errors`, `ssl`, `trace`, and `frames`, optionally followed by `:control`, `:data`, `:window`, or `:all`. Don't use any spaces.

Then set the logging level for the logger with name `jdk.httpclient.HttpClient` to `INFO`, for example by adding this line to the `logging.properties` file in your JDK:

```
jdk.httpclient.HttpClient.level=INFO
```

Listing 4.8 `client/HttpClientTest.java`

```
package client;

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

import java.net.http.*;
```

http://www.rejoiceblog.com/

```
11   import java.net.http.HttpRequest.*;

12

13   class MoreBodyPublishers {

14

15      public   static BodyPublisher ofFormData(Map<Object, 

16      {

17         var   first = true;

18         var   builder = new StringBuilder();

19         for   (Map.Entry<Object, Object> entry : data.entr

20         {

21               if (first) first = false;

22               else builder.append("&");

23               builder.append(URLEncoder.encode(entry.getKe

24               builder.append("=");

25               builder.append(URLEncoder.encode(entry.getVa

26          }

27          return BodyPublishers.ofString(builder.toString()

28      }

29

30      private static byte[] bytes(String s) { return s.getB
```

http://www.rejoiceblog.com/

```
31
32      public static BodyPublisher ofMimeMultipartData(Map<O
33          var byteArrays = new ArrayList<byte[]>();
34          byte[] separator = bytes("--" + boundary + "\nCont
35          for (Map.Entry<Object, Object> entry : data.entryS
36          {
37              byteArrays.add(separator);
38
39              if (entry.getValue() instanceof Path) {
40                  var path = (Path) entry.getValue();
41                  String mimeType = Files.probeContentType(pa
42                  byteArrays.add(bytes("\"" + entry.getKey()
43                      + "\"\nContent-Type: " + mimeType + "\n
44                  byteArrays.add(Files.readAllBytes(path));
45              }
46              else {
47                  byteArrays.add(bytes("\"" + entry.getKey()
48              }
49          }
```

http://www.rejoiceblog.com/

```
        byteArrays.add(bytes("--" + boundary + "--"));

        return BodyPublishers.ofByteArrays(byteArrays);

    }


    public static BodyPublisher ofSimpleJSON(Map<Object,

    {

        var builder = new StringBuilder();

        builder.append("{");

        var first = true;

        for (Map.Entry<Object, Object> entry : data.entry

        {

            if (first) first = false;

            else

                builder.append(",");

            builder.append(jsonEscape(entry.getKey().toStr

                .append(jsonEscape(entry.getValue().toStrin

        }

        builder.append("}");

        return BodyPublishers.ofString(builder.toString()

    }
```

http://www.rejoiceblog.com/

```

    private static Map<Character, String> replacements =
            '\n', "\\n", '\r', "\\r", '\t', "\\t", '"', "\

    private static StringBuilder jsonEscape(String str)
    {
        var result = new StringBuilder("\"");
        for (int i = 0; i < str.length(); i++)
        {
            char ch = str.charAt(i);
            String replacement = replacements.get(ch);
            if (replacement == null) result.append(ch);
            else result.append(replacement);
        }
        result.append("\"");
        return result;
    }
}

```

http://www.rejoiceblog.com/

```
89   public class HttpClientTest
90   {
91      public static void main(String[] args)
92            throws IOException, URISyntaxException, Interru
93      {
94         System.setProperty("jdk.httpclient.HttpClient.log"
95         String propsFilename = args.length > 0 ? args[0] :
96         Path propsPath = Paths.get(propsFilename);
97          var props = new Properties();
98          try (InputStream in = Files.newInputStream(propsP
99          {
100            props.load(in);
101         }
102         String urlString = "" + props.remove("url");
103         String contentType = "" + props.remove("Content-T
104         if (contentType.equals("multipart/form-data"))
105         {
106            var generator = new Random();
107            String boundary = new BigInteger(256, generato
108            contentType += ";boundary=" + boundary;
```

http://www.rejoiceblog.com/

```
            props.replaceAll((k, v) ->
                v.toString().startsWith("file://")
                    ? propsPath.getParent().resolve(Paths.ge
                    : v);
        }
        String result = doPost(urlString, contentType, pr
        System.out.println(result);
    }

    public static String doPost(String url, String conten
            throws IOException, URISyntaxException, Inte
    {
        HttpClient client = HttpClient.newBuilder()
            .followRedirects(HttpClient.Redirect.ALWAYS)

        BodyPublisher publisher = null;
        if (contentType.startsWith("multipart/form-data")
        {
            String boundary = contentType.substring(conten
```

http://www.rejoiceblog.com/

```
            publisher = MoreBodyPublishers.ofMimeMultipart
        }
        else if (contentType.equals("application/x-www-fo
            publisher = MoreBodyPublishers.ofFormData(data
        else
        {
            contentType = "application/json";
            publisher = MoreBodyPublishers.ofSimpleJSON(da
        }

        HttpRequest request = HttpRequest.newBuilder()
                .uri(new URI(url))
                .header("Content-Type", contentType)
                .POST(publisher)
                .build();
        HttpResponse<String> response = client.send(reque
        return response.body();
    }
}
```

http://www.rejoiceblog.com/

java.net.http.HttpClient 11

- `static HttpClient newHttpClient()`

yields an `HttpClient` with a default configuration.

- `static HttpClient.Builder newBuilder()`

yields a builder for building an `HttpClient`

- `<T> HttpResponse<T> send(HttpRequest request, HttpResponse.BodyHandler<T> responseBodyHandler)`

- `<T> CompletableFuture<HttpResponse<T>> sendAsync(HttpRequest request, HttpResponse.BodyHandler<T> responseBodyHandler)`

makes a synchronous or asynchronous request and processes the response body with the given handler.

java.net.http.HttpClient.Builder 11

- `HttpClient build()`

yields an `HttpClient` with the properties configured by this builder.

- `HttpClient.Builder followRedirects(HttpClient.Redirect policy)`

sets the redirect policy to one of the values `ALWAYS`, `NEVER`, or `NORMAL` (only refuse redirects from HTTPS to HTTP) of the `HttpClient.Redirect` enumeration.

- `HttpClient.Builder executor(Executor executor)`

sets the executor for asynchronous requests.

java.net.http.HttpRequest 11

- `HttpRequest.Builder newBuilder()`

http://www.rejoiceblog.com/

yields a builder for building an `HttpRequest`.

java.net.http.HttpRequest.Builder 11

- `HttpRequest build()`

yields an `HttpRequest` with the properties configured by this builder.

- `HttpRequest.Builder uri(URI uri)`

sets the URI for this request.

- `HttpRequest.Builder header(String name, String value)`

sets a request header for this request.

- `HttpRequest.Builder GET()`

- `HttpRequest.Builder DELETE()`

- `HttpRequest.Builder POST(HttpRequest.BodyPublisher bodyPublisher)`

- `HttpRequest.Builder PUT(HttpRequest.BodyPublisher bodyPublisher)`

sets the request method and body for this request.

java.net.http.HttpResponse<T> 11

- `T body()`

yields the body of this response.

- `int statusCode()`

yields the status code for this response.

- `HttpHeaders headers()`

http://www.rejoiceblog.com/

yields the response headers.

java.net.http.HttpHeaders 11

- `Map<String,List<String>> map()`

yields a `Map` of these headers.

- `Optional<String> firstValue(String name)`

the first value with the given name in these headers, if present.

4.5 SENDING E-MAIL

In the past, it was simple to write a program that sends e-mail by making a socket connection to port 25, the SMTP port, of a mail server. The Simple Mail Transport Protocol (SMTP) describes the format for e-mail messages. Once you are connected to the server, send a mail header (in the SMTP format, which is easy to generate), followed by the mail message.

Here are the details:

1. Open a socket to your host.

```
var s = new Socket("mail.yourserver.com", 25); // 25 is SMTP

var out = new PrintWriter(s.getOutputStream(), StandardCharse
```

2. Send the following information to the print stream:

```
HELO sending host

MAIL FROM: sender e-mail address

RCPT TO: recipient e-mail address
```

http://www.rejoiceblog.com/

```
DATA

Subject: subject

(blank line)

mail message (any number of lines)

.

QUIT
```

The SMTP specification (RFC 821) states that lines must be terminated with `\r` followed by `\n`.

It used to be that SMTP servers were often willing to route e-mail from anyone. However, in these days of spam floods, most servers have built-in checks and only accept requests from users or IP address ranges that they trust. Authentication usually happens over secure socket connections.

Implementing these authentication schemes manually would be very tedious. Instead, we will show you how to use the JavaMail API to send e-mail from a Java program.

Download JavaMail from `www.oracle.com/technetwork/java/javamail` and unzip it somewhere on your hard disk.

To use JavaMail, you need to set up some properties that depend on your mail server. For example, with GMail, you use

```
mail.transport.protocol=smtps

mail.smtps.auth=true

mail.smtps.host=smtp.gmail.com

mail.smtps.user=accountname@gmail.com
```

Our sample program reads these from a property file.

http://www.rejoiceblog.com/

For security reasons, we don't put the password into the property file but instead prompt for it.

Read in the property file, then get a mail session like this:

```
Session mailSession = Session.getDefaultInstance(props);
```

Make a message with the desired sender, recipient, subject, and message text:

```
var message = new MimeMessage(mailSession);

message.setFrom(new InternetAddress(from));

message.addRecipient(RecipientType.TO, new InternetAddress(to

message.setSubject(subject);

message.setText(builder.toString());
```

Then send it off:

```
Transport tr = mailSession.getTransport();

tr.connect(null, password);

tr.sendMessage(message, message.getAllRecipients());

tr.close();
```

The program in Listing 4.9 reads the message from a text file of the format

Sender

Recipient

http://www.rejoiceblog.com/

Subject

Message text (any number of lines)

To run the program, download the JavaMail implementation from `https://javaee.github.io/javamail`. You also need the JAR file for the Java Activation Framework—get it from http://www.oracle.com/technetwork/java/javase/jaf-135115.html or search on Maven Central. Then run

```
java -classpath .:javax.mail.jar:activation-1.1.1.jar path/to
```

At the time of this writing, GMail does not check the veracity of the information—you can supply any sender you like. (Keep this in mind the next time you get an e-mail message from president@whitehouse.gov inviting you to a black-tie affair on the front lawn.)

Tip

If you can't figure out why your mail connection isn't working, call

```
mailSession.setDebug(true);
```

and check out the messages. Also, the JavaMail API FAQ has some useful debugging hints.

Listing 4.9 `mail/MailTest.java`

```
package mail;

```

http://www.rejoiceblog.com/

```
3   import  java.io.*;
4   import  java.nio.charset.*;
5   import  java.nio.file.*;
6   import  java.util.*;
7   import  javax.mail.*;
8   import  javax.mail.internet.*;
9   import  javax.mail.internet.MimeMessage.RecipientType;
10
11  /**
12   * This program shows how to use JavaMail to send mail m
13   * @author Cay Horstmann
14   * @version 1.01 2018-03-17
15   */
16  public class MailTest
17  {
18      public static void main(String[] args) throws Messa
19      {
20          var props = new Properties();
21          try (InputStream in = Files.newInputStream(Paths
22          {
```

http://www.rejoiceblog.com/

```
            props.load(in);
        }
        List<String> lines = Files.readAllLines(Paths.ge

        String from = lines.get(0);
        String to = lines.get(1);
        String subject = lines.get(2);

        var builder = new StringBuilder();
        for (int i = 3; i < lines.size(); i++)
        {
            builder.append(lines.get(i));
            builder.append("\n");
        }

        Console console = System.console();
        var password = new String(console.readPassword("

        Session mailSession = Session.getDefaultInstance
```

http://www.rejoiceblog.com/

```
            // mailSession.setDebug(true);
            var message = new MimeMessage(mailSession);
            message.setFrom(new InternetAddress(from));
            message.addRecipient(RecipientType.TO, new Inter
            message.setSubject(subject);
            message.setText(builder.toString());
            Transport tr = mailSession.getTransport();
            try
            {
                tr.connect(null, password);
                tr.sendMessage(message, message.getAllRecipi
            }
            finally
            {
                tr.close();
            }
        }
    }
```

In this chapter, you have seen how to write network clients and servers in Java and how to harvest information from web servers. The next chapter covers database

http://www.rejoiceblog.com/
connectivity. You will learn how to work with relational databases in Java, using the JDBC API.

http://www.rejoiceblog.com/

Chapter 5 Database Programming

In this chapter

- 5.1 The Design of JDBC
- 5.2 The Structured Query Language
- 5.3 JDBC Configuration
- 5.4 Working with JDBC Statements
- 5.5 Query Execution
- 5.6 Scrollable and Updatable Result Sets
- 5.7 Row Sets
- 5.8 Metadata
- 5.9 Transactions
- 5.10 Connection Management in Web and Enterprise Applications

In 1996, Sun released the first version of the JDBC API. This API lets programmers connect to a database to query or update it using the Structured Query Language (SQL). (SQL, usually pronounced “sequel,” is an industry standard for relational database access.) JDBC has since become one of the most commonly used APIs in the Java library.

JDBC has been updated several times. As this book is published, JDBC 4.3, the version included with Java 9, is the most current version.

In this chapter, we will explain the key ideas behind JDBC. We will introduce you to (or refresh your memory of) SQL, the industry-standard Structured

http://www.rejoiceblog.com/

Query Language for relational databases. We will then provide enough details and examples to let you start using JDBC for common programming situations.

Note

According to Oracle, JDBC is a trademarked term and not an acronym for Java Database Connectivity. It was named to be reminiscent of ODBC, a standard database API pioneered by Microsoft and since incorporated into the SQL standard.

5.1 THE DESIGN OF JDBC

From the start, the developers of the Java technology were aware of the potential that Java showed for working with databases. In 1995, they began working on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that a program could talk to any random database using only "pure" Java. It didn't take them long to realize that this is an impossible task: There are simply too many databases out there, using too many protocols. Moreover, although database vendors were all in favor of Java providing a standard network protocol for database access, they were only in favor of it if Java used *their* network protocol.

What all the database vendors and tool vendors *did* agree on was that it would be useful for Java to provide a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases. Database vendors could provide their own drivers to plug in to the driver manager. There would then be a simple mechanism for registering third-party drivers with the driver manager.

This organization follows the very successful model of Microsoft's ODBC which provided a C programming language interface for database access. Both JDBC and ODBC are based on the same idea: Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.

This means the JDBC API is all that most programmers will ever have to deal with.

5.1.1 JDBC Driver Types

The JDBC specification classifies drivers into the following *types*:

http://www.rejoiceblog.com/

• A *type 1 driver* translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Early versions of Java included one such driver, the *JDBC/ODBC bridge*. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, many better drivers are available, and the JDK no longer provides the JDBC/ODBC bridge.

• A *type 2 driver* is written partly in Java and partly in native code; it communicates with the client API of a database. When using such a driver, you must install some platform-specific code onto the client in addition to a Java library.

• A *type 3 driver* is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This simplifies deployment because the platform-specific code is located only on the server.

• A *type 4 driver* is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Note

The JDBC specification is available at `https://jcp.org/aboutJava/communityprocess/mrel/jsr221/index3.html`.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:

• Programmers can write applications in the Java programming language to access any database using standard SQL statements (or even specialized extensions of SQL) while still following Java language conventions.

• Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

http://www.rejoiceblog.com/

Note

If you are curious as to why Java just didn't adopt the ODBC model, the reason, as given at the JavaOne conference in 1996, was this:

• ODBC is hard to learn.

• ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.

• ODBC relies on the use of `void*` pointers and other C features that are not natural in the Java programming language.

• An ODBC-based solution is inherently less safe and harder to deploy than a pure Java solution.

5.1.2 Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (see Figure 5.1). In this model, a JDBC driver is deployed on the client.

Figure 5.1 A traditional client/server application

However, nowadays it is far more common to have a three-tier model where the client application does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of

http://www.rejoiceblog.com/

advantages. It separates *visual presentation* (on the client) from the *business logic* (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java desktop application, a web browser, or a mobile app.

Communication between the client and the middle tier typically occurs through HTTP. JDBC manages the communication between the middle tier and the back-end database. Figure 5.2 shows the basic architecture.

Figure 5.2 A three-tier application

5.2 THE STRUCTURED QUERY LANGUAGE

JDBC lets you communicate with databases using SQL, which is the command language for essentially all modern relational databases. Desktop databases usually have a GUI that lets users manipulate the data directly, but server-based databases are accessed purely through SQL.

The JDBC package can be thought of as nothing more than an API for communicating SQL statements to databases. We will briefly introduce SQL in this section. If you have never seen SQL before, you might not find this material sufficient. If so, turn to one of the many learning resources on the topic; we recommend *Learning SQL* by Alan Beaulieu (O'Reilly, 2009) or the online book *Learn SQL The Hard Way* at `http://sql.learncodethehardway.org/`.

You can think of a database as a bunch of named tables with rows and columns. Each column has a *column name*. Each row contains a set of related data.

As an example database for this book, we use a set of database tables that describe a collection of classic computer science books (see Tables 5.1 through 5.4).

http://www.rejoiceblog.com/

Table 5.1 The `Authors` Table

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
. . .	. . .	. . .

Table 5.2 The `Books` Table

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
. . .	. . .	. . .	. . .

Table 5.3 The `BooksAuthors` Table

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
. . .	. . .	. . .

Table 5.4 The `Publishers` Table

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
. . .	. . .	. . .

Figure 5.3 shows a view of the `Books` table. Figure 5.4 shows the result of *joining* this table with the `Publishers` table. The `Books` and the `Publishers` tables each contain an identifier for the publisher. When we join both tables on the publisher code, we obtain a *query result* made up of values from the joined tables. Each row in the result contains the information about a book, together with the publisher name and web page URL. Note that the publisher names and URLs are duplicated across several rows because we have several rows with the same publisher.

http://www.rejoiceblog.com/

COREJAVA: Books

File Edit View Tools Window Help

Title	ISBN	Publisher_ID	Price
UNIX System Administration Handbook	0-13-020601-6	013	68.00
The C Programming Language	0-13-110362-8	013	42.00
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
Introduction to Automata Theory, Languages, and Computation	0-201-44124-1	0201	105.00
Design Patterns	0-201-63361-2	0201	54.99
The C++ Programming Language	0-201-70073-5	0201	64.99
The Mythical Man-Month	0-201-83595-9	0201	29.95
Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
The Art of Computer Programming vol. 1	0-201-89683-4	0201	59.99
The Art of Computer Programming vol. 2	0-201-89684-2	0201	59.99
The Art of Computer Programming vol. 3	0-201-89685-0	0201	59.99
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
Introduction to Algorithms	0-262-03293-7	0262	80.00
Applied Cryptography	0-471-11709-9	0471	60.00
JavaScript: The Definitive Guide	0-596-00048-0	0596	44.95
The Cathedral and the Bazaar	0-596-00108-8	0596	16.95
The Soul of a New Machine	0-679-60261-5	0679	18.95
The Codebreakers	0-684-83130-9	07434	70.00
Cuckoo's Egg	0-7434-1146-3	07434	13.95
The UNIX Hater"s Handbook	1-56884-203-1	0471	16.95

Record 1 of 20

Figure 5.3 Sample table containing books

Query1

File Edit View Insert Tools Window Help

Title	Publisher_ID	Price	Name	URL
UNIX System Administration Handbook	013	68.00	Prentice H	www.php
The C Programming Language	013	42.00	Prentice H	www.php
A Pattern Language: Towns, Buildings, Construction	019	65.00	Oxford Uni	www.oup
Introduction to Automata Theory, Languages, and Computation	0201	105.00	Addison-W	www.aw
Design Patterns	0201	54.99	Addison-W	www.aw
The C++ Programming Language	0201	64.99	Addison-W	www.aw
The Mythical Man-Month	0201	29.95	Addison-W	www.aw
Computer Graphics: Principles and Practice	0201	79.99	Addison-W	www.aw
The Art of Computer Programming vol. 1	0201	59.99	Addison-W	www.aw
The Art of Computer Programming vol. 2	0201	59.99	Addison-W	www.aw
The Art of Computer Programming vol. 3	0201	59.99	Addison-W	www.aw

Record 1 of 15 *

Books
*
Title
ISBN
Publisher_ID
Price

Publishers
*
Publisher_Id
Name
URL

Field	Title	Publisher_ID	Price	Name	URL		
Alias							
Table	Books	Books	Books	Publishers	Publishers		
Sort							
Visible	☑	☑	☑	☑	☑	☐	☐
Function							
Criterion							
Or							
Or							

Figure 5.4 Two tables joined together

The benefit of joining tables is avoiding unnecessary duplication of data in the database

http://www.rejoiceblog.com/

tables. For example, a naive database design might have had columns for the publisher name and URL right in the `Books` table. But then the database itself, and not just the query result, would have many duplicates of these entries. If a publisher's web address changed, *all* entries would need to be updated. Clearly, this is somewhat error-prone. In the relational model, we distribute data into multiple tables so that no information is unnecessarily duplicated. For example, each publisher's URL is contained only once in the publisher table. If the information needs to be combined, the tables are joined.

In the figures, you can see a graphical tool to inspect and link the tables. Many vendors have tools to express queries in a simple form by connecting column names and filling information into forms. Such tools are often called *query by example* (QBE) tools. In contrast, a query that uses SQL is written out in text, using SQL syntax, for example:

```
SELECT Books, Books.Publisher_Id, Books.Price, Publishers.Nam

FROM Books, Publishers

WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

In the remainder of this section, you will learn how to write such queries. If you are already familiar with SQL, just skip this section.

By convention, SQL keywords are written in capital letters, although this is not necessary.

The `SELECT` statement is quite flexible. You can simply select all rows in the `Books` table with the following query:

```
SELECT * FROM Books
```

The `FROM` clause is required in every SQL `SELECT` statement. It tells the database which tables to examine to find the data.

You can choose the columns that you want:

```
SELECT ISBN, Price, Title
```

http://www.rejoiceblog.com/

```
FROM Books
```

You can restrict the rows in the answer with the `WHERE` clause:

```
SELECT ISBN, Price, Title

FROM Books

WHERE Price <= 29.95
```

Be careful with the "equals" comparison. SQL uses = and <>, rather than == or

`!=` as in the Java programming language, for equality testing.

> **Note**
>
> Some database vendors support the use of `!=` for inequality testing. This is not standard SQL, so we recommend against such use.

The `WHERE` clause can also use pattern matching by means of the `LIKE` operator. The wildcard characters are not the usual `*` and `?`, however. Use a `%` for zero or more characters and an underscore for a single character. For example,

```
SELECT ISBN, Price, Title

FROM Books

WHERE Title NOT LIKE '%n_x%'
```

excludes books with titles that contain words such as Unix or Linux.

Note that strings are enclosed in single quotes, not double quotes. A single quote inside a string is represented by a pair of single quotes. For example,

http://www.rejoiceblog.com/

```
SELECT Title

FROM Books

WHERE Title LIKE '%''%'
```

reports all titles that contain a single quote.

You can select data from multiple tables:

```
SELECT * FROM Books, Publishers
```

Without a `WHERE` clause, this query is not very interesting. It lists *all combinations* of rows from both tables. In our case, where `Books` has 20 rows and `Publishers` has 8 rows, the result is a set of rows with 20 × 8 entries and lots of duplications. We really want to constrain the query to say that we are only interested in *matching* books with their publishers:

```
SELECT * FROM Books, Publishers

WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

This query result has 20 rows, one for each book, because each book has one publisher in the `Publisher` table.

Whenever you have multiple tables in a query, the same column name can occur in two different places. That happened in our example. There is a column called `Publisher_Id` in both the `Books` and the `Publishers` tables. When an ambiguity would otherwise result, you must prefix each column name with the name of the table to which it belongs, such as `Books.Publisher_Id`.

You can use SQL to change the data inside a database as well. For example, suppose you want to reduce by $5.00 the current price of all books that have "C++" in their title:

```
UPDATE Books
```

http://www.rejoiceblog.com/

```
SET Price = Price - 5.00

WHERE Title LIKE '%C++%'
```

Similarly, to delete all C++ books, use a `DELETE` query:

```
DELETE FROM Books

WHERE Title LIKE '%C++%'
```

SQL comes with built-in functions for taking averages, finding maximums and minimums in a column, and so on, which we do not discuss here.

Typically, to insert values into a table, you can use the `INSERT` statement:

```
INSERT INTO Books

VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201
```

You need a separate `INSERT` statement for every row being inserted in the table.

Of course, before you can query, modify, and insert data, you must have a place to store data. Use the `CREATE TABLE` statement to make a new table. Specify the name and data type for each column. For example,

```
CREATE TABLE Books

(

   Title CHAR(60),

   ISBN CHAR(13),

   Publisher_Id CHAR(6),

   Price DECIMAL(10,2)
```

http://www.rejoiceblog.com/

```
)
```

Table 5.5 shows the most common SQL data types.

Table 5.5 Common SQL Data Types

Data Types	Description
INTEGER or INT	Typically, a 32-bit integer
SMALLINT	Typically, a 16-bit integer
NUMERIC(m,n), DECIMAL(m,n) or DEC(m,n)	Fixed-point decimal number with m total digits and n digits after the decimal point
FLOAT(n)	A floating-point number with n binary digits of precision
REAL	Typically, a 32-bit floating-point number
DOUBLE	Typically, a 64-bit floating-point number
CHARACTER(n) or CHAR(n)	Fixed-length string of length n
VARCHAR(n)	Variable-length strings of maximum length n
BOOLEAN	A Boolean value
DATE	Calendar date, implementation-dependent
TIME	Time of day, implementation-dependent
TIMESTAMP	Date and time of day, implementation-dependent
BLOB	A binary large object
CLOB	A character large object

In this book, we do not discuss the additional clauses, such as keys and constraints, that you can use with the `CREATE TABLE` statement.

5.3 JDBC CONFIGURATION

Of course, you need a database program for which a JDBC driver is available. There are many excellent choices, such as IBM DB2, Microsoft SQL Server, MySQL, Oracle, and PostgreSQL.

You must also create a database for your experimental use. We assume you name it `COREJAVA`. Create a new database, or have your database administrator create one with the appropriate permissions. You need to be able to create, update, and drop tables in the database.

http://www.rejoiceblog.com/

If you have never installed a client/server database before, you might find that setting up the database is somewhat complex and that diagnosing the cause for failure can be difficult. It might be best to seek expert help if your setup is not working correctly.

If this is your first experience with databases, we recommend that you use the Apache Derby database, which is available from `http://db.apache.org/derby`.

You need to gather a number of items before you can write your first database program. The following sections cover these items.

5.3.1 Database URLs

When connecting to a database, you must use various database-specific parameters such as host names, port numbers, and database names.

JDBC uses a syntax similar to that of ordinary URLs to describe data sources. Here are examples of the syntax:

```
jdbc:derby://localhost:1527/COREJAVA;create=true

jdbc:postgresql:COREJAVA
```

These JDBC URLs specify a Derby database and a PostgreSQL database named `COREJAVA`.

The general syntax is

```
jdbc:subprotocol:other stuff
```

where a subprotocol selects the specific driver for connecting to the database.

The format for the *other stuff* parameter depends on the subprotocol used. You will need to look up your vendor's documentation for the specific format.

5.3.2 Driver JAR Files

You need to obtain the JAR file in which the driver for your database is located. If you use Derby, you need the file `derbyclient.jar`. With another database, you need to locate the appropriate driver. For example, the PostgreSQL drivers are available at

http://www.rejoiceblog.com/
http://jdbc.postgresql.org.

Include the driver JAR file on the class path when running a program that accesses the database. (You don't need the JAR file for compiling.)

When you launch programs from the command line, simply use the command

```
java -classpath driverPath:. ProgramName
```

On Windows, use a semicolon to separate the current directory (denoted by the `.` character) from the driver JAR location.

5.3.3 Starting the Database

The database server needs to be started before you can connect to it. The details depend on your database.

With the Derby database, follow these steps:

1. Open a command shell and change to a directory that will hold the database files.

2. Locate the file `derbyrun.jar`. With some versions of the JDK, it is contained in the *jdk*`/db/lib` directory. If it's not there, install Apache Derby and locate the JAR file in the installation directory. We will denote the directory containing `lib/derbyrun.jar` with *derby*.

3. Run the command

```
java -jar derby/lib/derbyrun.jar server start
```

4. Double-check that the database is working correctly. Create a file `ij.properties` that contains these lines:

```
ij.driver=org.apache.derby.jdbc.ClientDriver

ij.protocol=jdbc:derby://localhost:1527/

ij.database=COREJAVA;create=true
```

http://www.rejoiceblog.com/

From another command shell, run Derby's interactive scripting tool (called `ij`) by executing

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

Now you can issue SQL commands such as

```
CREATE TABLE Greetings (Message CHAR(20));

INSERT INTO Greetings VALUES ('Hello, World!');

SELECT * FROM Greetings;

DROP TABLE Greetings;
```

Note that each command must be terminated by a semicolon. To exit, type

```
EXIT;
```

5. When you are done using the database, stop the server with the command

```
java -jar derby/lib/derbyrun.jar server shutdown
```

If you use another database, you need to consult the documentation to find out how to start and stop your database server, and how to connect to it and issue SQL commands.

5.3.4 Registering the Driver Class

Many JDBC JAR files (such as the Apache Derby driver) automatically register the driver class. In that case, you can skip the manual registration step that we describe in this section. A JAR file can automatically register the driver class if it contains a file `META-INF/services/java.sql.Driver`. You can simply unzip your driver's JAR file to check.

If your driver's JAR file doesn't support automatic registration, you need to find out the

http://www.rejoiceblog.com/
name of the JDBC driver classes used by your vendor. Typical driver names are

```
org.apache.derby.jdbc.ClientDriver

org.postgresql.Driver
```

There are two ways to register the driver with the `DriverManager`. One way is to load the driver class in your Java program. For example,

```
Class.forName("org.postgresql.Driver"); // force loading of d
```

This statement causes the driver class to be loaded, thereby executing a static initializer that registers the driver.

Alternatively, you can set the `jdbc.drivers` property. You can specify the property with a command-line argument, such as

`java -Djdbc.drivers=org.postgresql.Driver` *ProgramName*

Or, your application can set the system property with a call such as

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

You can also supply multiple drivers; separate them with colons, for example

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

5.3.5 Connecting to the Database

In your Java program, open a database connection like this:

```
String url = "jdbc:postgresql:COREJAVA";

String username = "dbuser";
```

http://www.rejoiceblog.com/

```
String password = "secret";

Connection conn = DriverManager.getConnection(url, username,
```

The driver manager iterates through the registered drivers to find a driver that can use the subprotocol specified in the database URL.

The `getConnection` method returns a `Connection` object. In the following sections, you will see how to use the `Connection` object to execute SQL statements.

To connect to the database, you will need to have a user name and password for your database.

> **Note**
>
> By default, Derby lets you connect with any user name, and it does not check passwords. A separate set of tables is generated for each user. The default user name is `app`.

The test program in Listing 5.1 puts these steps to work. It loads connection parameters from a file named `database.properties` and connects to the database. The `database.properties` file supplied with the sample code contains connection information for the Derby database. If you use a different database, put your database-specific connection information into that file. Here is an example for connecting to a PostgreSQL database:

```
jdbc.drivers=org.postgresql.Driver

jdbc.url=jdbc:postgresql:COREJAVA

jdbc.username=dbuser

jdbc.password=secret
```

http://www.rejoiceblog.com/

After connecting to the database, the test program executes the following SQL statements:

```
CREATE TABLE Greetings (Message CHAR(20))

INSERT INTO Greetings VALUES ('Hello, World!')

SELECT * FROM Greetings
```

The result of the `SELECT` statement is printed, and you should see an output of

```
Hello, World!
```

Then the table is removed by executing the statement

```
DROP TABLE Greetings
```

To run this test, start your database, as described previously, and launch the program as

```
java -classpath .:driverJAR test.TestDB
```

(As always, Windows users need to use `;` instead of `:` to separate the path elements.)

Tip

One way to debug JDBC-related problems is to enable JDBC tracing. Call the `DriverManager.setLogWriter` method to send trace messages to a `PrintWriter`. The trace output contains a detailed listing of the JDBC activity. Most JDBC driver implementations provide additional mechanisms for tracing. For example, with Derby, you can add a `traceFile` option to the JDBC URL: `jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out`.

http://www.rejoiceblog.com/

Listing 5.1 `test/TestDB.java`

```
1  package test;
2
3  import java.nio.file.*;
4  import java.sql.*;
5  import java.io.*;
6  import java.util.*;
7
8  /**
9   * This program tests that the database and the JDBC drive
10  * @version 1.03 2018-05-01
11  * @author Cay Horstmann
12  */
13 public class TestDB
14 {
15    public static void main(String args[]) throws IOExcept
16    {
17       try
18       {
```

http://www.rejoiceblog.com/

```
19            runTest();
20         }
21         catch (SQLException ex)
22         {
23            for (Throwable t : ex)
24               t.printStackTrace();
25         }
26      }
27
28      /**
29      * Runs a test by creating a table, adding a value, sho
30      * the table.
31      */
32      public static void runTest() throws SQLException, IOEx
33      {
34
35         try (Connection conn = getConnection();
36         Statement stat = conn.createStatement())
37      {
38            stat.executeUpdate("CREATE TABLE Greetings (Mess
```

http://www.rejoiceblog.com/

```
39          stat.executeUpdate("INSERT INTO Greetings VALUES
40
41          try (ResultSet result = stat.executeQuery("SELEC
42          {
43             if (result.next())
44                System.out.println(result.getString(1));
45     }
46          stat.executeUpdate("DROP TABLE Greetings");
47     }
48   }
49
50   /**
51   * Gets a connection from the properties specified in the
52   * @return the database connection
53   */
54   public static Connection getConnection() throws SQLExcep
55   {
56     var props = new Properties();
57     try (InputStream in = Files.newInputStream(Paths.get("
```

http://www.rejoiceblog.com/

```
    {
       props.load(in);
    }
    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null) System.setProperty("jdbc.drivers"
    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return DriverManager.getConnection(url, username, pass
  }
}
```

java.sql.DriverManager 1.1

- `static Connection getConnection(String url, String user, String password)`

establishes a connection to the given database and returns a `Connection` object.

5.4 WORKING WITH JDBC STATEMENTS

In the following sections, you will see how to use the JDBC `Statement` to execute SQL statements, obtain results, and deal with errors. Then we show you a simple program for populating a database.

http://www.rejoiceblog.com/

5.4.1 Executing SQL Statements

To execute a SQL statement, you first create a `Statement` object. To create statement objects, use the `Connection` object that you obtained from the call to `DriverManager.getConnection`.

```
Statement stat = conn.createStatement();
```

Next, place the statement that you want to execute into a string, for example

```
String command = "UPDATE Books"
   + " SET Price = Price - 5.00"
   + " WHERE Title NOT LIKE '%Introduction%'";
```

Then call the `executeUpdate` method of the `Statement` interface:

```
stat.executeUpdate(command);
```

The `executeUpdate` method returns a count of the rows that were affected by the SQL statement, or zero for statements that do not return a row count. For example, the call to `executeUpdate` in the preceding example returns the number of rows where the price was lowered by $5.00.

The `executeUpdate` method can execute actions such as `INSERT`, `UPDATE`, and `DELETE`, as well as data definition statements such as `CREATE TABLE` and `DROP TABLE`. However, you need to use the `executeQuery` method to execute `SELECT` queries. There is also a catch-all `execute` statement to execute arbitrary SQL statements. It's commonly used only for queries that a user supplies interactively.

When you execute a query, you are interested in the result. The `executeQuery` object returns an object of type `ResultSet` that you can use to walk through the result one row at a time.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

http://www.rejoiceblog.com/

The basic loop for analyzing a result set looks like this:

```
while (rs.next())
{
    look at a row of the result set
}
```

Caution

The iteration protocol of the `ResultSet` interface is subtly different from the protocol of the `java.util.Iterator` interface. Here, the iterator is initialized to a position *before* the first row. You must call the `next` method once to move the iterator to the first row. Also, there is no `hasNext` method; keep calling `next` until it returns `false`.

The order of the rows in a result set is completely arbitrary. Unless you specifically ordered the result with an `ORDER BY` clause, you should not attach any significance to the row order.

When inspecting an individual row, you will want to know the contents of the fields. A large number of accessor methods give you this information.

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

There are accessors for various *types*, such as `getString` and `getDouble`. Each accessor has two forms: One takes a numeric argument and the other takes a string argument. When you supply a numeric argument, you refer to the column with that number. For example, `rs.getString(1)` returns the value of the first column in the current row.

http://www.rejoiceblog.com/

Caution

Unlike array indexes, database column numbers start at `1`.

When you supply a string argument, you refer to the column in the result set with that name. For example, `rs.getDouble("Price")` returns the value of the column with label `Price`. Using the numeric argument is a bit more efficient, but the string arguments make the code easier to read and maintain.

Each `get` method makes reasonable type conversions when the type of the method doesn't match the type of the column. For example, the call `rs.getString("Price")` converts the floating-point value of the `Price` column to a string.

java.sql.Connection 1.1

- `Statement createStatement()`

creates a `Statement` object that can be used to execute SQL queries and updates without parameters.

- `void close()`

immediately closes the current connection and the JDBC resources that it created.

java.sql.Statement 1.1

- `ResultSet executeQuery(String sqlQuery)`

executes the SQL statement given in the string and returns a `ResultSet` object to view the query result.

- `int executeUpdate(String sqlStatement)`

- `long executeLargeUpdate(String sqlStatement)` 8

http://www.rejoiceblog.com/

executes the SQL `INSERT`, `UPDATE`, or `DELETE` statement specified by the string. Also executes Data Definition Language (DDL) statements such as `CREATE TABLE`. Returns the number of rows affected, or `0` for a statement without an update count.

- `boolean execute(String sqlStatement)`

executes the SQL statement specified by the string. Multiple result sets and update counts may be produced. Returns `true` if the first result is a result set, `false` otherwise. Call `getResultSet` or `getUpdateCount` to retrieve the first result. See Section 5.5.4, "Multiple Results," on p. 321 for details on processing multiple results.

- `ResultSet getResultSet()`

returns the result set of the preceding query statement, or `null` if the preceding statement did not have a result set. Call this method only once per executed statement.

- `int getUpdateCount()`

- `long getLargeUpdateCount()` 8

returns the number of rows affected by the preceding update statement, or `-1` if the preceding statement was a statement without an update count. Call this method only once per executed statement.

- `void close()`

closes this statement object and its associated result set.

- `boolean isClosed()` 6

returns `true` if this statement is closed.

- `void closeOnCompletion()` 7

causes this statement to be closed once all of its result sets have been closed.

java.sql.ResultSet 1.1

- `boolean next()`

http://www.rejoiceblog.com/

makes the current row in the result set move forward by one. Returns `false` after the last row. Note that you must call this method to advance to the first row.

- *Xxx* `getXxx(int columnNumber)`

- *Xxx* `getXxx(String columnLabel)`

(*Xxx* is a type such as `int`, `double`, `String`, `Date`, etc.)

- `<T> T getObject(int columnIndex, Class<T> type)` 7

- `<T> T getObject(String columnLabel, Class<T> type)` 7

- `void updateObject(int columnIndex, Object x, SQLType targetSqlType)` 8

- `void updateObject(String columnLabel, Object x, SQLType targetSqlType)` 8

returns or updates the value of the column with the given column index or label, converted to the specified type. The column label is the label specified in the SQL `AS` clause or the column name if `AS` is not used.

- `int findColumn(String columnName)`

gives the column index associated with a column name.

- `void close()`

immediately closes the current result set.

- `boolean isClosed()` 6

returns `true` if this statement is closed.

5.4.2 Managing Connections, Statements, and Result Sets

Every `Connection` object can create one or more `Statement` objects. You can use the same `Statement` object for multiple unrelated commands and queries. However, a statement has *at most one* open result set. If you issue multiple queries whose results you analyze concurrently, you need multiple `Statement` objects.

http://www.rejoiceblog.com/

Be forewarned, though, that there is a limit to the number of statements per connection. Use the `getMaxStatements` method of the `DatabaseMetaData` interface to find out the number of concurrently open statements that your JDBC driver supports.

In practice, you should probably not fuss with multiple concurrent result sets. If the result sets are related, you should be able to issue a combined query and analyze a single result. It is much more efficient to let the database combine queries than it is for a Java program to iterate through multiple result sets.

Be sure to finish processing any result set before you issuing a new query or update on a `Statement` object. The result sets of prior queries are automatically closed.

When you are done using a `ResultSet`, `Statement`, or `Connection`, it is a good idea to call the `close` method immediately. These objects use large data structures that draw on the finite resources of the database server.

The `close` method of a `Statement` object closes any associated result sets. Similarly, the `close` method of the `Connection` class closes all statements of the connection.

Conversely, you can call the `closeOnCompletion` method on a `Statement`, and it will close automatically as soon as all its result sets have closed.

If your connections are short-lived, you don't have to worry about closing statements and result sets. To make sure that a connection object cannot possibly remain open, use a `try`-with-resources statement:

```
try (Connection conn = . . .)
{
   Statement stat = conn.createStatement();
   ResultSet result = stat.executeQuery(queryString);
   process query result
}
```

http://www.rejoiceblog.com/

5.4.3 Analyzing SQL Exceptions

Each `SQLException` has a chain of `SQLException` objects that are retrieved with the `getNextException` method. This exception chain is in addition to the "cause" chain of `Throwable` objects that every exception has. (See Volume I, Chapter 11 for details about Java exceptions.) One would need two nested loops to fully enumerate all these exceptions. Fortunately, the `SQLException` class has been enhanced to implement the `Iterable<Throwable>` interface. The `iterator()` method yields an `Iterator<Throwable>` that iterates through both chains: starts by going through the cause chain of the first `SQLException`, then moves on to the next `SQLException`, and so on. You can simply use an enhanced `for` loop:

```
for (Throwable t : sqlException)
{
    do something with t
}
```

You can call `getSQLState` and `getErrorCode` on a `SQLException` to analyze it further. The first method yields a string that is standardized by either X/Open or SQL:2003. (Call the `getSQLStateType` method of the `DatabaseMetaData` interface to find out which standard is used by your driver.) The error code is vendor-specific.

The SQL exceptions are organized into an inheritance tree (shown in Figure 5.5). This allows you to catch specific error types in a vendor-independent way.

http://www.rejoiceblog.com/

Figure 5.5 SQL exception types

In addition, the database driver can report nonfatal conditions as warnings. You can retrieve warnings from connections, statements, and result sets. The `SQLWarning` class is a subclass of `SQLException` (even though a `SQLWarning` is not thrown as an exception). Call `getSQLState` and `getErrorCode` to get further information about the warnings. Similar to SQL exceptions, warnings are chained. To retrieve all warnings, use this loop:

```
SQLWarning w = stat.getWarning();

while (w != null)

{

    do something with w

    w = w.nextWarning();

}
```

http://www.rejoiceblog.com/

The `DataTruncation` subclass of `SQLWarning` is used when data are read from the database and unexpectedly truncated. If data truncation happens in an update statement, a `DataTruncation` is thrown as an exception.

java.sql.SQLException 1.1

- `SQLException getNextException()`

gets the next SQL exception chained to this one, or `null` at the end of the chain.

- `Iterator<Throwable> iterator()` 6

gets an iterator that yields the chained SQL exceptions and their causes.

- `String getSQLState()`

gets the "SQL state"—a standardized error code.

- `int getErrorCode()`

gets the vendor-specific error code.

java.sql.SQLWarning 1.1

- `SQLWarning getNextWarning()`

returns the next warning chained to this one, or `null` at the end of the chain.

java.sql.Connection **1.1**
java.sql.Statement 1.1
java.sql.ResultSet 1.1

- `SQLWarning getWarnings()`

returns the first of the pending warnings, or `null` if no warnings are pending.

http://www.rejoiceblog.com/

java.sql.DataTruncation 1.1

- `boolean getParameter()`

returns `true` if the data truncation applies to a parameter, `false` if it applies to a column.

- `int getIndex()`

returns the index of the truncated parameter or column.

- `int getDataSize()`

returns the number of bytes that should have been transferred, or `-1` if the value is unknown.

- `int getTransferSize()`

returns the number of bytes that were actually transferred, or `-1` if the value is unknown.

5.4.4 Populating a Database

We are now ready to write our first real JDBC program. Of course it would be nice to try some of the fancy queries that we discussed earlier, but we have a problem: Right now, there are no data in the database. We need to populate the database, and there is a simple way of doing that with a set of SQL instructions to create tables and insert data into them. Most database programs can process a set of SQL instructions from a text file, but there are pesky differences about statement terminators and other syntactical issues.

For that reason, we will use JDBC to create a simple program that reads a file with SQL instructions, one instruction per line, and executes them.

Specifically, the program reads data from a text file in a format such as

```
CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30),

INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www
```

http://www.rejoiceblog.com/

```
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', '

. . .
```

Listing 5.2 contains the code for the program that reads the SQL statement file and executes the statements. You don't have to read through the code; we merely provide the program so that you can populate your database and run the examples in the remainder of this chapter.

Make sure that your database server is running, and run the program as follows:

```
java -classpath driverPath:. exec.ExecSQL Books.sql

java -classpathdriverPath:. exec.ExecSQL Authors.sql

java -classpath driverPath:. exec.ExecSQL Publishers.sql

java -classpath driverPath:. exec.ExecSQL BooksAuthors.sql
```

Before running the program, check that the file `database.properties` is set up properly for your environment (see Section 5.3.5, "Connecting to the Database," on p. 296).

Note

Your database may also have a utility to read SQL files directly. For example, with Derby, you can run

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

(The `ij.properties` file is described in Section 5.3.3, "Starting the Database," on p. 295.)

In the data format for the `ExecSQL` command, we allow an optional semicolon at the end of each line because most database utilities expect this format.

http://www.rejoiceblog.com/

The following steps briefly describe the `ExecSQL` program:

1. Connect to the database. The `getConnection` method reads the properties in the file `database.properties` and adds the `jdbc.drivers` property to the system properties. The driver manager uses the `jdbc.drivers` property to load the appropriate database driver. The `getConnection` method uses the `jdbc.url`, `jdbc.username`, and `jdbc.password` properties to open the database connection.

2. Open the file with the SQL statements. If no file name was supplied, prompt the user to enter the statements on the console.

3. Execute each statement with the generic `execute` method. If it returns `true`, the statement had a result set. The four SQL files that we provide for the book database all end in a `SELECT *` statement so that you can see that the data were successfully inserted.

4. If there was a result set, print out the result. Since this is a generic result set, we need to use metadata to find out how many columns the result has. For more information, see Section 5.8, "Metadata," on p. 334.

5. If there is any SQL exception, print the exception and any chained exceptions that may be contained in it.

6. Close the connection to the database.

Listing 5.2 shows the code for the program.

Listing 5.2 `exec/ExecSQL.java`

```
package exec;

import   java.io.*;
import   java.nio.charset.*;
```

http://www.rejoiceblog.com/

```
import   java.nio.file.*;
import   java.util.*;
import   java.sql.*;

/**
 * Executes all SQL statements in a file. Call this pro
 * java -classpath driverPath:. ExecSQL commandFile
 *
 * @version 1.33 2018-05-01
 * @author Cay Horstmann
 */
class ExecSQL
{
    public static void main(String args[]) throws IOExc
    {
        try (Scanner in = args.length == 0 ? new Scanner
            : new Scanner(Paths.get(args[0]), StandardCha
        {
            try (Connection conn = getConnection();
                Statement stat = conn.createStatement())
```

http://www.rejoiceblog.com/

```
            {
                while (true)
                {
                    if (args.length == 0) System.out.printl

                 if (!in.hasNextLine()) return;

                 String line = in.nextLine().trim();
                 if (line.equalsIgnoreCase("EXIT")) return
                 if (line.endsWith(";")) // remove trailin
                 {
                     line = line.substring(0, line.length(
                 }
                 try
                 {
                     boolean isResult = stat.execute(line)
                     if (isResult)
                     {
                         try (ResultSet rs = stat.getResult
```

http://www.rejoiceblog.com/

```
44                          {
45                             showResultSet(rs);
46                          }
47                       }
48                       else
49                       {
50                          int updateCount = stat.getUpdateCo
51                          System.out.println(updateCount + "
52                       }
53                    }
54                    catch (SQLException e)
55                    {
56                       for (Throwable t : e)
57                          t.printStackTrace();
58                    }
59                 }
60              }
61           }
62           catch (SQLException e)
63           {
```

http://www.rejoiceblog.com/

```
            for (Throwable t : e)
               t.printStackTrace();
         }
      }

      /**
        * Gets a connection from the properties specified i
        * @return the database connection
        */
      public static Connection getConnection() throws SQLEx
      {
           var props = new Properties();
           try (InputStream in = Files.newInputStream(Paths
           {
              props.load(in);
           }

         String drivers = props.getProperty("jdbc.drivers"
         if (drivers != null) System.setProperty("jdbc.dri
```

http://www.rejoiceblog.com/

```
        String url = props.getProperty("jdbc.url");
        String username = props.getProperty("jdbc.usernam
        String password = props.getProperty("jdbc.passwor

        return DriverManager.getConnection(url, username,
    }

    /**
      * Prints a result set.
      * @param result the result set to be printed
      */
    public static void showResultSet(ResultSet result) th
    {
         ResultSetMetaData metaData = result.getMetaData(
         int columnCount = metaData.getColumnCount();

        for (int i = 1; i <= columnCount; i++)
        {
           if (i > 1) System.out.print(", ");
```

http://www.rejoiceblog.com/

```
            System.out.print(metaData.getColumnLabel(i));
        }
        System.out.println();

          while (result.next())
          {
             for (int i = 1; i <= columnCount; i++)
             {
                if (i > 1) System.out.print(", ");
                System.out.print(result.getString(i));
             }
             System.out.println();
          }
       }
   }
```

5.5 QUERY EXECUTION

In this section, we write a program that executes queries against the `COREJAVA` database. For this program to work, you must have populated the `COREJAVA` database with tables, as described in the preceding section.

When querying the database, you can select the author and the publisher or leave either

http://www.rejoiceblog.com/

of them as `Any`.

You can also change the data in the database. Select a publisher and type an amount. All prices of that publisher are adjusted by the amount you entered, and the program displays how many rows were changed. After a price change, you might want to run a query to verify the new prices.

5.5.1 Prepared Statements

In this program, we use one new feature, *prepared statements*. Consider the query for all books by a particular publisher, independent of the author. The SQL query is

```
SELECT Books.Price, Books

FROM Books, Publishers

WHERE Books.Publisher_Id = Publishers.Publisher_Id

AND Publishers.Name = the name from the list box
```

Instead of building a separate query statement every time the user launches such a query, we can *prepare* a query with a host variable and use it many times, each time filling in a different string for the variable. That technique benefits performance. Whenever the database executes a query, it first computes a strategy of how to do it efficiently. By preparing the query and reusing it, you ensure that the planning step is done only once.

Each host variable in a prepared query is indicated with a `?`. If there is more than one variable, you must keep track of the positions of the `?` when setting the values. For example, our prepared query becomes

```
String publisherQuery =

   "SELECT Books.Price, Books" +

   " FROM Books, Publishers" +

   " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND P
```

http://www.rejoiceblog.com/

```
PreparedStatement stat = conn.prepareStatement(publisherQuery
```

Before executing the prepared statement, you must bind the host variables to actual values with a `set` method. As with the `get` methods of the `ResultSet` interface, there are different `set` methods for the various types. Here, we want to set a string to a publisher name.

```
stat.setString(1, publisher);
```

The first argument is the position number of the host variable that we want to set. The position 1 denotes the first `?`. The second argument is the value that we want to assign to the host variable.

If you reuse a prepared query that you have already executed, all host variables stay bound unless you change them with a `set` method or call the `clearParameters` method. That means you only need to call a `set`*Xxx* method on those host variables that change from one query to the next.

Once all variables have been bound to values, you can execute the prepared statement:

```
ResultSet rs = stat.executeQuery();
```

Tip

Building a query manually, by concatenating strings, is tedious and potentially dangerous. You have to worry about special characters such as quotes, and, if your query involves user input, you have to guard against injection attacks. Therefore, you should use prepared statements whenever your query involves variables.

The price update feature is implemented as an `UPDATE` statement. Note that we call `executeUpdate`, not `executeQuery`, because the `UPDATE` statement does not return

http://www.rejoiceblog.com/
a result set. The return value of `executeUpdate` is the count of changed rows.

```
int r = stat.executeUpdate();

System.out.println(r + " rows updated");
```

Note

A `PreparedStatement` object becomes invalid after the associated `Connection` object is closed. However, many databases automatically *cache* prepared statements. If the same query is prepared twice, the database simply reuses the query strategy. Therefore, don't worry about the overhead of calling `prepareStatement`.

The following list briefly describes the structure of the example program.

• The author and publisher array lists are populated by running two queries that return all author and publisher names in the database.

• The queries involving authors are complex. A book can have multiple authors, so the `BooksAuthors` table stores the correspondence between authors and books. For example, the book with ISBN 0–201–96426–0 has two authors with codes `DATE` and `DARW`. The `BooksAuthors` table has the rows

```
0-201-96426-0, DATE,

1 0-201-96426-0, DARW, 2
```

to indicate this fact. The third column lists the order of the authors. (We can't just use the position of the rows in the table. There is no fixed row ordering in a relational table.) Thus, the query has to join the `Books`, `BooksAuthors`, and `Authors` tables to compare the author name with the one selected by the user.

```
SELECT Books.Price, Books FROM Books, BooksAuthors, Authors,
```

http://www.rejoiceblog.com/

```
WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAut

AND Books.Publisher_Id = Publishers.Publisher_Id AND Authors.
```

Tip

Some Java programmers avoid complex SQL statements such as this one. A surprisingly common, but very inefficient, workaround is to write lots of Java code that iterates through multiple result sets. But the database is *a lot* better at executing query code than a Java program can be—that's the core competency of a database. A rule of thumb: If you can do it in SQL, don't do it in Java.

• The `changePrices` method executes an `UPDATE` statement. Note that the `WHERE` clause of the `UPDATE` statement needs the publisher *code* and we know only the publisher *name*. This problem is solved with a nested subquery:

```
UPDATE Books

SET Price = Price + ?

WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishe
```

Listing 5.3 is the complete program code.

Listing 5.3 `query/QueryTest.java`

```
1   package query;

2

3   import  java.io.*;
```

http://www.rejoiceblog.com/

```
 4   import  java.nio.file.*;
 5   import  java.sql.*;
 6   import  java.util.*;
 7
 8   /**
 9    * This program demonstrates several complex database qu
10    * @version 1.31 2018-05-01
11    * @author Cay Horstmann
12    */
13   public class QueryTest
14   {
15        private static final String allQuery = "SELECT Book
16
17      private   static final String authorPublisherQuery =
18            +   " FROM Books, BooksAuthors, Authors, Publis
19            +   " WHERE Authors.Author_Id = BooksAuthors.Au
20            +   " AND Books.Publisher_Id = Publishers.Publi
21            +   " AND Publishers.Name = ?";
22
```

http://www.rejoiceblog.com/

```
23      private static final String authorQuery
24         = "SELECT Books.Price, Books.Title FROM Books, Boo
25            + " WHERE Authors.Author_Id = BooksAuthors.Auth
26            + " AND Authors.Name = ?";
27
28      private static final String publisherQuery
29         = "SELECT Books.Price, Books.Title FROM Books, Pub
30            + " WHERE Books.Publisher_Id = Publishers.Publi
31
32
33      private static final String priceUpdate = "UPDATE Boo
34            + " WHERE Books.Publisher_Id = (SELECT Publishe
35
36      private static Scanner in;
37      private static ArrayList<String> authors = new ArrayL
38      private static ArrayList<String> publishers = new Arr
39
40      public static void main(String[] args) throws IOExcep
41      {
42         try (Connection conn = getConnection())
```

http://www.rejoiceblog.com/

```
        {
            in = new Scanner(System.in);
            authors.add("Any");
            publishers.add("Any");
            try (Statement stat = conn.createStatement())
            {
                // Fill the authors array list
                var query = "SELECT Name FROM Authors";
                try (ResultSet rs = stat.executeQuery(query)
                {
                    while (rs.next())
                        authors.add(rs.getString(1));
                }

                // Fill the publishers array list
                query = "SELECT Name FROM Publishers";
                try (ResultSet rs = stat.executeQuery(query)
                {
                    while (rs.next())
```

http://www.rejoiceblog.com/

```
                    publishers.add(rs.getString(1));
                }
            }
            var done = false;
            while (!done)
            {
                System.out.print("Q)uery C)hange prices E)xi
                String input = in.next().toUpperCase();
                if (input.equals("Q"))
                    executeQuery(conn);
                else if (input.equals("C"))
                    changePrices(conn);
                else
                    done = true;
            }
        }
        catch (SQLException e)
        {
            for (Throwable t : e)
                System.out.println(t.getMessage());
```

http://www.rejoiceblog.com/

```
        }
    }

    /**
      * Executes the selected query.
      * @param conn the database connection
      */
    private static void executeQuery(Connection conn) thro
    {
        String author = select("Authors:", authors);
        String publisher = select("Publishers:", publishe
        PreparedStatement stat;
        if (!author.equals("Any") && !publisher.equals("A
        {
            stat = conn.prepareStatement(authorPublisherQu
          stat.setString(1, author);
          stat.setString(2, publisher);
        }
        else if (!author.equals("Any") && publisher.equal
```

http://www.rejoiceblog.com/

```
        {
            stat = conn.prepareStatement(authorQuery);
            stat.setString(1, author);
        }
        else if (author.equals("Any") && !publisher.equal
        {
            stat = conn.prepareStatement(publisherQuery)
            stat.setString(1, publisher);
        }
        else
           stat = conn.prepareStatement(allQuery);

        try (ResultSet rs = stat.executeQuery())
        {
           while (rs.next())
              System.out.println(rs.getString(1) + ", " +
        }
    }

    /**
```

http://www.rejoiceblog.com/

```
121       * Executes an update statement to change prices.
122       * @param conn the database connection
123       */
124     public static void changePrices(Connection conn) thro
125     {
126         String publisher = select("Publishers:", publish
127         System.out.print("Change prices by: ");
128         double priceChange = in.nextDouble();
129         PreparedStatement stat = conn.prepareStatement(p
130         stat.setDouble(1, priceChange);
131         stat.setString(2, publisher);
132         int r = stat.executeUpdate();
133         System.out.println(r + " records updated.");
134     }
135
136     /**
137       * Asks the user to select a string.
138       * @param prompt the prompt to display
139       * @param options the options from which the user ca
```

http://www.rejoiceblog.com/

```
140         * @return the option that the user chose
141         */
142       public static String select(String prompt, List<Strin
143       {
144           while (true)
145           {
146               System.out.println(prompt);
147               for (int i = 0; i < options.size(); i++)
148                  System.out.printf("%2d) %s%n", i + 1, op
149               int sel = in.nextInt();
150               if (sel > 0 && sel <= options.size())
151                  return options.get(sel - 1);
152           }
153         }
154
155         /**
156           * Gets a connection from the properties specified
157           * @return the database connection
158           */
159         public static Connection getConnection() throws SQL
```

http://www.rejoiceblog.com/

```
    {
        var props = new Properties();
        try (InputStream in = Files.newInputStream(Pat
        {
            props.load(in);
        }

        String drivers = props.getProperty("jdbc.driver
        if (drivers != null) System.setProperty("jdbc.d
        String url = props.getProperty("jdbc.url");
        String username = props.getProperty("jdbc.usern
        String password = props.getProperty("jdbc.passw

        return DriverManager.getConnection(url, usernam
    }
}
```

java.sql.Connection 1.1

- PreparedStatement prepareStatement(String sql)

http://www.rejoiceblog.com/

returns a `PreparedStatement` object containing the precompiled statement. The string `sql` contains a SQL statement with one or more parameter placeholders denoted by `?` characters.

java.sql.PreparedStatement 1.1

- `void set`*Xxx*`(int n,` *Xxx* `x)`

(*Xxx* is a type such as `int`, `double`, `String`, `Date`, etc.)

sets the value of the `n`th parameter to `x`.

- `void clearParameters()`

clears all current parameters in the prepared statement.

- `ResultSet executeQuery()`

executes a prepared SQL query and returns a `ResultSet` object.

- `int executeUpdate()`

executes the prepared SQL `INSERT`, `UPDATE`, or `DELETE` statement represented by the `PreparedStatement` object. Returns the number of rows affected, or `0` for DDL statements such as `CREATE TABLE`.

5.5.2 Reading and Writing LOBs

In addition to numbers, strings, and dates, many databases can store *large objects* (LOBs) such as images or other data. In SQL, binary large objects are called BLOBs, and character large objects are called CLOBs.

To read a LOB, execute a `SELECT` statement and call the `getBlob` or `getClob` method on the `ResultSet`. You will get an object of type `Blob` or `Clob`. To get the binary data from a `Blob`, call the `getBytes` or `getBinaryStream`. For example, if you have a table with book cover images, you can retrieve an image like this:

```
PreparedStatement stat = conn.prepareStatement("SELECT Cover
```

http://www.rejoiceblog.com/

```
. . .

stat.set(1, isbn);

try (ResultSet result = stat.executeQuery())

{

   if (result.next())

   {

     Blob coverBlob = result.getBlob(1);

     Image coverImage = ImageIO.read(coverBlob.getBinaryStrea

   }

}
```

Similarly, if you retrieve a `Clob` object, you can get character data by calling the `getSubString` or `getCharacterStream` method.

To place a LOB into a database, call `createBlob` or `createClob` on your `Connection` object, get an output stream or writer to the LOB, write the data, and store the object in the database. For example, here is how you store an image:

```
Blob coverBlob = connection.createBlob();

int offset = 0;

OutputStream out = coverBlob.setBinaryStream(offset);

ImageIO.write(coverImage, "PNG", out);

PreparedStatement stat = conn.prepareStatement("INSERT INTO C
```

http://www.rejoiceblog.com/

```
stat.set(1, isbn);

stat.set(2, coverBlob);

stat.executeUpdate();
```

java.sql.ResultSet 1.1

- `Blob getBlob(int columnIndex)` 1.2
- `Blob getBlob(String columnLabel)` 1.2
- `Clob getClob(int columnIndex)` 1.2
- `Clob getClob(String columnLabel)` 1.2

gets the BLOB or CLOB at the given column.

java.sql.Blob 1.2

- `long length()`

gets the length of this BLOB.

- `byte[] getBytes(long startPosition, long length)`

gets the data in the given range from this BLOB.

- `InputStream getBinaryStream()`
- `InputStream getBinaryStream(long startPosition, long length)`

returns a stream to read the data from this BLOB or from the given range.

- `OutputStream setBinaryStream(long startPosition)` 1.4

returns an output stream for writing into this BLOB, starting at the given position.

http://www.rejoiceblog.com/

java.sql.Clob 1.4

- `long length()`

gets the number of characters of this CLOB.

- `String getSubString(long startPosition, long length)`

gets the characters in the given range from this CLOB.

- `Reader getCharacterStream()`
- `Reader getCharacterStream(long startPosition, long length)`

returns a reader (not a stream) to read the characters from this CLOB or from the given range.

- `Writer setCharacterStream(long startPosition)` 1.4

returns a writer (not a stream) for writing into this CLOB, starting at the given position.

java.sql.Connection 1.1

- `Blob createBlob()` 6
- `Clob createClob()` 6

creates an empty BLOB or CLOB.

5.5.3 SQL Escapes

The “escape” syntax features are commonly supported by databases but use database-specific syntax variations. It is the job of the JDBC driver to translate the escape syntax to the syntax of a particular database.

Escapes are provided for the following features:

http://www.rejoiceblog.com/

- Date and time literals
- Calling scalar functions
- Calling stored procedures
- Outer joins
- The escape character in `LIKE` clauses

Date and time literals vary widely among databases. To embed a date or time literal, specify the value in ISO 8601 format (`www.cl.cam.ac.uk/~mgk25/iso-time.html`). The driver will then translate it into the native format. Use `d`, `t`, `ts` for `DATE`, `TIME`, or `TIMESTAMP` values:

```
{d '2008-01-24'}

{t '23:59:59'}

{ts '2008-01-24 23:59:59.999'}
```

A *scalar function* is a function that returns a single value. Many functions are widely available in databases, but with varying names. The JDBC specification provides standard names and translates them into the database-specific names. To call a function, embed the standard function name and arguments like this:

```
{fn left(?, 20)}

{fn user()}
```

You can find a complete list of supported function names in the JDBC specification.

A *stored procedure* is a procedure that executes in the database, written in a database-specific language. To call a stored procedure, use the `call` escape. You need not supply parentheses if the procedure has no parameters. Use = to capture a return value:

```
{call PROC1(?, ?)}
```

http://www.rejoiceblog.com/

```
{call PROC2}

{call ? = PROC3(?)}
```

An *outer join* of two tables does not require that the rows of each table match according to the join condition. For example, the query

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers ON Books.P
```

contains books for which `Publisher_Id` has no match in the `Publishers` table, with `NULL` values to indicate that no match exists. You would need a `RIGHT OUTER JOIN` to include publishers without matching books, or a `FULL OUTER JOIN` to return both. The escape syntax is needed because not all databases use a standard notation for these joins.

Finally, the _ and `%` characters have special meanings in a `LIKE` clause—to match a single character or a sequence of characters. There is no standard way to use them literally. If you want to match all strings containing a _, use this construct:

```
... WHERE ? LIKE %!_% {escape '!'}
```

Here we define `!` as the escape character. The combination `!_` denotes a literal underscore.

5.5.4 Multiple Results

It is possible for a query to return multiple results. This can happen when executing a stored procedure, or with databases that also allow submission of multiple `SELECT` statements in a single query. Here is how you retrieve all result sets:

1. Use the `execute` method to execute the SQL statement.

2. Retrieve the first result or update count.

3. Repeatedly call the `getMoreResults` method to move on to the next result set.

4. Finish when there are no more result sets or update counts.

http://www.rejoiceblog.com/

The `execute` and `getMoreResults` methods return `true` if the next item in the chain is a result set. The `getUpdateCount` method returns `-1` if the next item in the chain is not an update count.

The following loop traverses all results:

```
boolean isResult = stat.execute(command);

boolean done = false;

while (!done)

{

   if (isResult)

   {

     ResultSet result = stat.getResultSet();

     do something with result

   }

   else

   {

     int updateCount = stat.getUpdateCount();

     if (updateCount >= 0)

       do something with updateCount

     else

       done = true;

   }
```

http://www.rejoiceblog.com/

```
      if (!done) isResult = stat.getMoreResults();

   }
```

java.sql.Statement 1.1

- `boolean getMoreResults()`
- `boolean getMoreResults(int current)` 6

gets the next result for this statement. The `current` parameter is one of `CLOSE_CURRENT_RESULT` (default), `KEEP_CURRENT_RESULT`, or `CLOSE_ALL_RESULTS`. Returns `true` if the next result exists and is a result set.

5.5.5 Retrieving Autogenerated Keys

Most databases support some mechanism for autonumbering rows in a database. Unfortunately, the mechanisms differ widely among vendors. These automatic numbers are often used as primary keys. Although JDBC doesn't offer a vendor-independent solution for generating keys, it does provide an efficient way of retrieving them. When you insert a new row into a table and a key is automatically generated, you can retrieve it with the following code:

```
   stat.executeUpdate(insertStatement, Statement.RETURN_GENERATE

   ResultSet rs = stat.getGeneratedKeys();

   if (rs.next())

   {

      int key = rs.getInt(1);

      . . .

   }
```

http://www.rejoiceblog.com/

java.sql.Statement 1.1

- `boolean execute(String statement, int autogenerated)` 1.4
- `int executeUpdate(String statement, int autogenerated)` 1.4

executes the given SQL statement, as previously described. If `autogenerated` is set to `Statement.RETURN_GENERATED_KEYS` and the statement is an `INSERT` statement, the first column contains the autogenerated key.

5.6 SCROLLABLE AND UPDATABLE RESULT SETS

As you have seen, the `next` method of the `ResultSet` interface iterates over the rows in a result set. That is certainly adequate for a program that needs to analyze the data. However, consider a visual data display that shows a table or query results (such as Figure 5.4). You usually want the user to be able to move both forward and backward in the result set. In a *scrollable* result, you can move forward and backward through a result set and even jump to any position.

Furthermore, once users see the contents of a result set displayed, they may be tempted to edit it. In an *updatable* result set, you can programmatically update entries so that the database is automatically updated. We discuss these capabilities in the following sections.

5.6.1 Scrollable Result Sets

By default, result sets are not scrollable or updatable. To obtain scrollable result sets from your queries, you must obtain a different `Statement` object with the method

```
Statement stat = conn.createStatement(type, concurrency);
```

For a prepared statement, use the call

```
PreparedStatement stat = conn.prepareStatement(command, type,
```

http://www.rejoiceblog.com/

The possible values of `type` and `concurrency` are listed in Tables 5.6 and 5.7. You have the following choices:

• Do you want the result set to be scrollable? If not, use `ResultSet.TYPE_FORWARD_ONLY`.

• If the result set is scrollable, do you want it to reflect changes in the database that occurred after the query that yielded it? (In our discussion, we assume the `ResultSet.TYPE_SCROLL_INSENSITIVE` setting for scrollable result sets. This assumes that the result set does not "sense" database changes that occurred after execution of the query.)

• Do you want to be able to update the database by editing the result set? (See the next section for details.)

Table 5.6 `ResultSet` Type Values

Value	Explanation
TYPE_FORWARD_ONLY	The result set is not scrollable (default).
TYPE_SCROLL_INSENSITIVE	The result set is scrollable but not sensitive to database changes.
TYPE_SCROLL_SENSITIVE	The result set is scrollable and sensitive to database changes.

Table 5.7 `ResultSet` Concurrency Values

Value	Explanation
CONCUR_READ_ONLY	The result set cannot be used to update the database (default).
CONCUR_UPDATABLE	The result set can be used to update the database.

For example, if you simply want to be able to scroll through a result set but don't want to edit its data, use

```
Statement stat = conn.createStatement(

      ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ
```

All result sets that are returned by method calls

http://www.rejoiceblog.com/

```
ResultSet rs = stat.executeQuery(query);
```

are now scrollable. A scrollable result set has a *cursor* that indicates the current position.

> **Note**
>
> Not all database drivers support scrollable or updatable result sets. (The `supportsResultSetType` and `supportsResultSetConcurrency` methods of the `DatabaseMetaData` interface will tell you which types and concurrency modes are supported by a particular database using a particular driver.) Even if a database supports all result set modes, a particular query might not be able to yield a result set with all the properties that you requested. (For example, the result set of a complex query might not be updatable.) In that case, the `executeQuery` method returns a `ResultSet` of lesser capabilities and adds a `SQLWarning` to the connection object. (Section 5.4.3, "Analyzing SQL Exceptions," on p. 304 shows how to retrieve the warning.) Alternatively, you can use the `getType` and `getConcurrency` methods of the `ResultSet` interface to find out what mode a result set actually has. If you do not check the result set capabilities and issue an unsupported operation, such as `previous` on a result set that is not scrollable, the operation will throw a `SQLException`.

Scrolling is very simple. Use

```
if (rs.previous()) . . .
```

to scroll backward. The method returns `true` if the cursor is positioned on an actual row, or `false` if it is now positioned before the first row.

You can move the cursor backward or forward by a number of rows with the call

```
rs.relative(n);
```

http://www.rejoiceblog.com/

If *n* is positive, the cursor moves forward. If *n* is negative, it moves backward. If *n* is zero, the call has no effect. If you attempt to move the cursor outside the current set of rows, it is set to point either after the last row or before the first row, depending on the sign of *n*. Then, the method returns `false` and the cursor does not move. The method returns `true` if the cursor is positioned on an actual row.

Alternatively, you can set the cursor to a particular row number:

```
rs.absolute(n);
```

To get the current row number, call

```
int currentRow = rs.getRow();
```

The first row in the result set has number `1`. If the return value is `0`, the cursor is not currently on a row—it is either before the first row or after the last row.

The convenience methods `first`, `last`, `beforeFirst`, and `afterLast` move the cursor to the first, to the last, before the first, or after the last position.

Finally, the methods `isFirst`, `isLast`, `isBeforeFirst`, and `isAfterLast` test whether the cursor is at one of these special positions.

Using a scrollable result set is very simple. The hard work of caching the query data is carried out behind the scenes by the database driver.

5.6.2 Updatable Result Sets

If you want to edit the data in the result set and have the changes automatically reflected in the database, create an updatable result set. Updatable result sets don't have to be scrollable, but if you present data to a user for editing, you usually want to allow scrolling as well.

To obtain updatable result sets, create a statement as follows:

```
Statement stat = conn.createStatement(

 ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABL
```

http://www.rejoiceblog.com/

The result sets returned by a call to `executeQuery` are then updatable.

> **Note**
>
> Not all queries return updatable result sets. If your query is a join that involves multiple tables, the result might not be updatable. However, if your query involves only a single table or if it joins multiple tables by their primary keys, you should expect the result set to be updatable. Call the `getConcurrency` method of the `ResultSet` interface to find out for sure.

For example, suppose you want to raise the prices of some books, but you don't have a simple criterion for issuing an `UPDATE` statement. Then, you can iterate through all books and update prices based on arbitrary conditions.

```
String query = "SELECT * FROM Books";

ResultSet rs = stat.executeQuery(query);

while (rs.next())

{

   if (. . .)

   {

     double increase = . . .

     double price = rs.getDouble("Price");

     rs.updateDouble("Price", price + increase);

     rs.updateRow(); // make sure to call updateRow after upd
```

http://www.rejoiceblog.com/

```
        }

    }
```

There are `updateXxx` methods for all data types that correspond to SQL types, such as `updateDouble`, `updateString`, and so on; specify the name or the number of the column (as with the `getXxx` methods), then the new value for the field.

> **Note**
>
> If you use the `updateXxx` method whose first parameter is the column number, be aware that this is the column number in the *result set*. It could well be different from the column number in the database.

The `updateXxx` method changes only the row values, not the database. When you are done with the field updates in a row, you must call the `updateRow` method. That method sends all updates in the current row to the database. If you move the cursor to another row without calling `updateRow`, this row's updates are discarded from the row set and never communicated to the database. You can also call the `cancelRowUpdates` method to cancel the updates to the current row.

The preceding example shows how to modify an existing row. If you want to add a new row to the database, first use the `moveToInsertRow` method to move the cursor to a special position, called the *insert row*. Then, build up a new row in the insert row position by issuing `updateXxx` instructions. When you are done, call the `insertRow` method to deliver the new row to the database. When you are done inserting, call `moveToCurrentRow` to move the cursor back to the position before the call to `moveToInsertRow`. Here is an example:

```
rs.moveToInsertRow();

rs.updateString("Title", title);

rs.updateString("ISBN", isbn);
```

http://www.rejoiceblog.com/

```
rs.updateString("Publisher_Id", pubid);

rs.updateDouble("Price", price);

rs.insertRow();

rs.moveToCurrentRow();
```

Note that you cannot influence *where* the new data is added in the result set or the database.

If you don't specify a column value in the insert row, it is set to a SQL `NULL`. However, if the column has a `NOT NULL` constraint, an exception is thrown and the row is not inserted.

Finally, you can delete the row under the cursor:

```
rs.deleteRow();
```

The `deleteRow` method immediately removes the row from both the result set and the database.

The `updateRow`, `insertRow`, and `deleteRow` methods of the `ResultSet` interface give you the same power as executing `UPDATE`, `INSERT`, and `DELETE` SQL statements. However, Java programmers might find it more natural to manipulate the database contents through result sets than by constructing SQL statements.

Caution

If you are not careful, you can write staggeringly inefficient code with updatable result sets. It is *much* more efficient to execute an `UPDATE` statement than to make a query and iterate through the result, changing data along the way. Updatable result sets make sense for interactive programs in which a user can make arbitrary changes, but for most programmatic changes, a SQL `UPDATE` is more appropriate.

http://www.rejoiceblog.com/

Note

JDBC 2 delivered further enhancements to result sets, such as the capability to update a result set with the most recent data if the data have been modified by another concurrent database connection. JDBC 3 added yet another refinement, specifying the behavior of result sets when a transaction is committed. However, these advanced features are outside the scope of this introductory chapter. We refer you to the *JDBC™ API Tutorial and Reference, Third Edition*, by Maydene Fisher, Jon Ellis, and Jonathan Bruce (Addison-Wesley, 2003) and the JDBC specification for more information.

java.sql.Connection 1.1

- `Statement createStatement(int type, int concurrency)` **1.2**
- `PreparedStatement prepareStatement(String command, int type, int concurrency)` **1.2**

creates a statement or prepared statement that yields result sets with the given type and concurrency. The `type` parameter is of the constants `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`, and `concurrency` is one of the constants `CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`, all defined in the `ResultSet` interface.

java.sql.ResultSet 1.1

- `int getType()` **1.2**

returns the type of this result set—one of `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`.

- `int getConcurrency()` **1.2**

returns the concurrency setting of this result set—one of `CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`.

http://www.rejoiceblog.com/

- `boolean previous()` **1.2**

moves the cursor to the preceding row. Returns `true` if the cursor is positioned on a row, or `false` if the cursor is positioned before the first row.

- `int getRow()` **1.2**

gets the number of the current row. Rows are numbered starting with `1`.

- `boolean absolute(int r)` **1.2**

moves the cursor to row `r`. Returns `true` if the cursor is positioned on a row.

- `boolean relative(int d)` **1.2**

moves the cursor by `d` rows. If `d` is negative, the cursor is moved backward. Returns `true` if the cursor is positioned on a row.

- `boolean first()` **1.2**

- `boolean last()` **1.2**

moves the cursor to the first or last row. Returns `true` if the cursor is positioned on a row.

- `void beforeFirst()` **1.2**

- `void afterLast()` **1.2**

moves the cursor before the first or after the last row.

- `boolean isFirst()` **1.2**

- `boolean isLast()` **1.2**

tests whether the cursor is at the first or last row.

- `boolean isBeforeFirst()` **1.2**

- `boolean isAfterLast()` **1.2**

tests whether the cursor is before the first or after the last row.

http://www.rejoiceblog.com/

- `void moveToInsertRow()` **1.2**

moves the cursor to the insert row. The insert row is a special row for inserting new data with the `update`*Xxx* and `insertRow` methods.

- `void moveToCurrentRow()` **1.2**

moves the cursor back from the insert row to the row that it occupied when the `moveToInsertRow` method was called.

- `void insertRow()` **1.2**

inserts the contents of the insert row into the database and the result set.

- `void deleteRow()` **1.2**

deletes the current row from the database and the result set.

- `void update`*Xxx*`(int column,` *Xxx* `data)` **1.2**

- `void update`*Xxx*`(String columnName,` *Xxx* `data)` **1.2**

(*Xxx* is a type such as `int`, `double`, `String`, `Date`, etc.)

updates a field in the current row of the result set.

- `void updateRow()` **1.2**

sends the current row updates to the database.

- `void cancelRowUpdates()` **1.2 329**

cancels the current row updates.

java.sql.DatabaseMetaData 1.1

- `boolean supportsResultSetType(int type)` **1.2**

returns `true` if the database can support result sets of the given type; `type` is one of the constants `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE` of the `ResultSet` interface.

http://www.rejoiceblog.com/

• `boolean supportsResultSetConcurrency(int type, int concurrency)` **1.2**

returns `true` if the database can support result sets of the given combination of type and concurrency. The `type` parameter is one of the constants `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`, and `concurrency` is one of the constants `CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`, all defined in the `ResultSet` interface.

5.7 ROW SETS

Scrollable result sets are powerful, but they have a major drawback. You need to keep the database connection open during the entire user interaction. However, a user can walk away from the computer for a long time, leaving the connection occupied. That is not good—database connections are scarce resources. In this situation, use a *row set*. The `RowSet` interface extends the `ResultSet` interface, but row sets don't have to be tied to a database connection.

Row sets are also suitable if you need to move a query result to a different tier of a complex application, or to another device such as a cell phone. You would never want to move a result set—its data structures can be huge, and it is tethered to the database connection.

5.7.1 Constructing Row Sets

The `javax.sql.rowset` package provides the following interfaces that extend the `RowSet` interface:

• A `CachedRowSet` allows disconnected operation. We will discuss cached row sets in the following section.

• A `WebRowSet` is a cached row set that can be saved to an XML file. The XML file can be moved to another tier of a web application where it is opened by another `WebRowSet` object.

• The `FilteredRowSet` and `JoinRowSet` interfaces support lightweight operations on row sets that are equivalent to SQL `SELECT` and `JOIN` operations. These operations are carried out on the data stored in row sets, without having to make a database connection.

http://www.rejoiceblog.com/

• A `JdbcRowSet` is a thin wrapper around a `ResultSet`. It adds useful methods from the `RowSet` interface.

As of Java 7, there is a standard way for obtaining a row set:

```
RowSetFactory factory = RowSetProvider.newFactory();

CachedRowSet crs = factory.createCachedRowSet();
```

There are similar methods for obtaining the other row set types.

5.7.2 Cached Row Sets

A cached row set contains all data from a result set. Since `CachedRowSet` is a subinterface of the `ResultSet` interface, you can use a cached row set exactly as you would use a result set. Cached row sets confer an important benefit: You can close the connection and still use the row set. As you will see in our sample program in Listing 5.4, this greatly simplifies the implementation of interactive applications. Each user command simply opens the database connection, issues a query, puts the result in a cached row set, and then closes the database connection.

It is even possible to modify the data in a cached row set. Of course, the modifications are not immediately reflected in the database; you need to make an explicit request to accept the accumulated changes. The `CachedRowSet` then reconnects to the database and issues SQL statements to write the accumulated changes.

You can populate a `CachedRowSet` from a result set:

```
ResultSet result = . . .;

RowSetFactory factory = RowSetProvider.newFactory();

CachedRowSet crs = factory.createCachedRowSet();

crs.populate(result);

conn.close(); // now OK to close the database connection
```

http://www.rejoiceblog.com/

Alternatively, you can let the `CachedRowSet` object establish a connection automatically. Set up the database parameters:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");

crs.setUsername("dbuser");

crs.setPassword("secret");
```

Then set the query statement and any parameters.

```
crs.setCommand("SELECT * FROM Books WHERE Publisher_ID = ?");

crs.setString(1, publisherId);
```

Finally, populate the row set with the query result:

```
crs.execute();
```

This call establishes a database connection, issues the query, populates the row set, and disconnects.

If your query result is very large, you would not want to put it into the row set in its entirety. After all, your users will probably only look at a few rows. In that case, specify a page size:

```
CachedRowSet crs = . . .;

crs.setCommand(command);

crs.setPageSize(20);

. . .

crs.execute();
```

http://www.rejoiceblog.com/

Now you will only get 20 rows. To get the next batch of rows, call

```
crs.nextPage();
```

You can inspect and modify the row set with the same methods you use for result sets. If you modified the row set contents, you must write it back to the database by calling

```
crs.acceptChanges(conn);
```

or

```
crs.acceptChanges();
```

The second call works only if you configured the row set with the information required to connect to a database (such as URL, user name, and password).

In Section 5.6.2, "Updatable Result Sets," on p. 326, you saw that not all result sets are updatable. Similarly, a row set that contains the result of a complex query will not be able to write its changes back to the database. You should be safe if your row set contains data from a single table.

> **Caution**
>
> If you populated the row set from a result set, the row set does not know the name of the table to update. You need to call `setTableName` to set the table name.

Another complexity arises if the data in the database have changed after you populated the row set. This is clearly a sign of trouble that could lead to inconsistent data. The reference implementation checks whether the original row set values (that is, the values before editing) are identical to the current values in the database. If so, they are replaced with the edited values; otherwise, a `SyncProviderException` is thrown and

http://www.rejoiceblog.com/

none of the changes are written. Other implementations may use other strategies for synchronization.

javax.sql.RowSet 1.4

- `String getURL()`
- `void setURL(String url)`

gets or sets the database URL.

- `String getUsername()`
- `void setUsername(String username)`

gets or sets the user name for connecting to the database.

- `String getPassword()`
- `void setPassword(String password)`

gets or sets the password for connecting to the database.

- `String getCommand()`
- `void setCommand(String command)`

gets or sets the command that is executed to populate this row set.

- `void execute()`

populates this row set by issuing the statement set with `setCommand`. For the driver manager to obtain a connection, the URL, user name, and password must be set.

javax.sql.rowset.CachedRowSet 5.0

- `void execute(Connection conn)`

populates this row set by issuing the statement set with `setCommand`. This method uses the given connection *and closes it*.

http://www.rejoiceblog.com/

- `void populate(ResultSet result)`

populates this cached row set with the data from the given result set.

- `String getTableName()`

- `void setTableName(String tableName)`

gets or sets the name of the table from which this cached row set was populated.

- `int getPageSize()`

- `void setPageSize(int size)`

gets or sets the page size.

- `boolean nextPage()`

- `boolean previousPage()`

loads the next or previous page of rows. Returns `true` if there is a next or previous page.

- `void acceptChanges()`

- `void acceptChanges(Connection conn)`

reconnects to the database and writes the changes that are the result of editing the row set. May throw a `SyncProviderException` if the data cannot be written back because the database data have changed.

javax.sql.rowset.RowSetProvider 7

- `static RowSetFactory newFactory()`

creates a row set factory.

javax.sql.rowset.RowSetFactory 7

http://www.rejoiceblog.com/

- `CachedRowSet createCachedRowSet()`
- `FilteredRowSet createFilteredRowSet()`
- `JdbcRowSet createJdbcRowSet()`
- `JoinRowSet createJoinRowSet()`
- `WebRowSet createWebRowSet()`

creates a row set of the specified type.

5.8 METADATA

In the preceding sections, you saw how to populate, query, and update database tables. However, JDBC can give you additional information about the *structure* of a database and its tables. For example, you can get a list of the tables in a particular database or the column names and types of a table. This information is not useful when you are implementing a business application with a predefined database. After all, if you design the tables, you know their structure. Structural information is, however, extremely useful for programmers who write tools that work with any database.

In SQL, data that describe the database or one of its parts are called *metadata* (to distinguish them from the actual data stored in the database). You can get three kinds of metadata: about a database, about a result set, and about parameters of prepared statements.

To find out more about the database, request an object of type `DatabaseMetaData` from the database connection.

```
DatabaseMetaData meta = conn.getMetaData();
```

Now you are ready to get some metadata. For example, the call

```
ResultSet mrs = meta.getTables(null, null, null, new String[]
```

returns a result set that contains information about all tables in the database. (See the

http://www.rejoiceblog.com/
API note at the end of this section for other parameters to this method.)

Each row in the result set contains information about a table in the database. The third column is the name of the table. (Again, see the API note for the other columns.) The following loop gathers all table names:

```
while (mrs.next())

   tableNames.addItem(mrs.getString(3));
```

There is a second important use for database metadata. Databases are complex, and the SQL standard leaves plenty of room for variability. Well over a hundred methods in the `DatabaseMetaData` interface can inquire about the database, including calls with such exotic names as

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

and

```
meta.nullPlusNonNullIsNull()
```

Clearly, these are geared toward advanced users with special needs—in particular, those who need to write highly portable code that works with multiple databases.

The `DatabaseMetaData` interface gives data about the database. A second metadata interface, `ResultSetMetaData`, reports information about a result set. Whenever you have a result set from a query, you can inquire about the number of columns and each column's name, type, and field width. Here is a typical loop:

```
ResultSet rs = stat.executeQuery("SELECT * FROM " + tableName

ResultSetMetaData meta = rs.getMetaData();

for (int i = 1; i <= meta.getColumnCount(); i++)

{
```

http://www.rejoiceblog.com/

```
    String columnName = meta.getColumnLabel(i);

    int columnWidth = meta.getColumnDisplaySize(i);

    . . .

  }
```

In this section, we will show you how to write such a simple tool. The program in Listing 5.4 uses metadata to let you browse all tables in a database. The program also illustrates the use of a cached row set.

The combo box on top displays all tables in the database. Select one of them, and the center of the frame is filled with the field names of that table and the values of the first row, as shown in Figure 5.6. Click Next and Previous to scroll through the rows in the table. You can also delete a row and edit the row values. Click the Save button to save the changes to the database.

Figure 5.6 The `ViewDB` application

Note

Many databases come with much more sophisticated tools for viewing and editing tables. If your database doesn't, check out DBeaver (`https://dbeaver.io/`) or SQuirreL (`http://squirrel-sql.sourceforge.net`). These programs can view the tables in any JDBC

http://www.rejoiceblog.com/

database. Our example program is not intended as a replacement for these tools, but it shows you how to implement a tool for working with arbitrary tables.

Listing 5.4 `view/ViewDB.java`

```
package view;

import  java.awt.*;
import  java.awt.event.*;
import  java.io.*;
import  java.nio.file.*;
import  java.sql.*;
import  java.util.*;

import javax.sql.*;
import javax.sql.rowset.*;
import javax.swing.*;

/**
* This program uses metadata to display arbitrary tables
* @version 1.34 2018-05-01
```

http://www.rejoiceblog.com/

```
17   * @author Cay Horstmann
18   */
19   public class ViewDB
20   {
21         public static void main(String[] args)
22         {
23            EventQueue.invokeLater(() ->
24               {
25                  var frame = new ViewDBFrame();
26                  frame.setTitle("ViewDB");
27                  frame.setDefaultCloseOperation(JFrame.EXI
28                  frame.setVisible(true);
29               });
30         }
31   }
32
33   /**
34   * The frame that holds the data panel and the navigation
35   */
```

http://www.rejoiceblog.com/

```
36   class ViewDBFrame extends JFrame
37   {
38        private JButton previousButton;
39        private JButton nextButton;
40        private JButton deleteButton;
41        private JButton saveButton;
42        private DataPanel dataPanel;
43        private Component scrollPane;
44        private JComboBox<String> tableNames;
45        private Properties props;
46        private CachedRowSet crs;
47        private Connection conn;
48
49      public ViewDBFrame()
50      {
51         tableNames = new JComboBox<String>();
52
53         try
54         {
55            readDatabaseProperties();
```

http://www.rejoiceblog.com/

```
            conn = getConnection();
            DatabaseMetaData meta = conn.getMetaData();
            try (ResultSet mrs = meta.getTables(null, null,
            {
               while (mrs.next())
                  tableNames.addItem(mrs.getString(3));
            }
         }
         catch (SQLException ex)
         {
             for (Throwable t : ex)
                t.printStackTrace();
         }
         catch (IOException ex)
         {
            ex.printStackTrace();
         }

         tableNames.addActionListener(
```

http://www.rejoiceblog.com/

```
            event -> showTable((String) tableNames.getSelec
        add(tableNames, BorderLayout.NORTH);
        addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent event
                {
                    try
                    {
                        if (conn != null) conn.close();
                    }
                    catch (SQLException ex)
                    {
                        for (Throwable t : ex)
                            t.printStackTrace();
                    }
                }
            });

        var buttonPanel = new JPanel();
        add(buttonPanel, BorderLayout.SOUTH);
```

http://www.rejoiceblog.com/

```
        previousButton = new JButton("Previous");
        previousButton.addActionListener(event -> showPrev
        buttonPanel.add(previousButton);

        nextButton = new JButton("Next");
        nextButton.addActionListener(event -> showNextRow(
        buttonPanel.add(nextButton);

        deleteButton = new JButton("Delete");
        deleteButton.addActionListener(event -> deleteRow(
        buttonPanel.add(deleteButton);

        saveButton = new JButton("Save");
        saveButton.addActionListener(event -> saveChanges(
        buttonPanel.add(saveButton);
        if (tableNames.getItemCount() > 0)
           showTable(tableNames.getItemAt(0), conn);
    }
```

http://www.rejoiceblog.com/

```
114
115     /**
116       * Prepares the text fields for showing a new table,
117       * @param tableName the name of the table to display
118       * @param conn the database connection
119       */
120     public void showTable(String tableName, Connection co
121     {
122         try (Statement stat = conn.createStatement();
123            ResultSet result = stat.executeQuery("SELECT
124         {
125            // get result set
126
127           // copy into cached row set
128           RowSetFactory factory = RowSetProvider.newFact
129           crs = factory.createCachedRowSet();
130           crs.setTableName(tableName);
131           crs.populate(result);
132
133           if (scrollPane != null) remove(scrollPane);
```

http://www.rejoiceblog.com/

```
            dataPanel = new DataPanel(crs);
            scrollPane = new JScrollPane(dataPanel);
            add(scrollPane, BorderLayout.CENTER);
            pack();
            showNextRow();
         }
         catch (SQLException ex)
         {
            for (Throwable t : ex)
               t.printStackTrace();
         }
      }

      /**
        * Moves to the previous table row.
        */
      public void showPreviousRow()
      {
          try
```

http://www.rejoiceblog.com/

```
        {
            if (crs == null || crs.isFirst()) return;
            crs.previous();
            dataPanel.showRow(crs);
        }
        catch (SQLException ex)
        {
            for (Throwable t : ex)
            t.printStackTrace();
        }
    }

    /**
      * Moves to the next table row.
      */
    public void showNextRow()
    {
        try
        {
            if (crs == null || crs.isLast()) return;
```

http://www.rejoiceblog.com/

```
            crs.next();
            dataPanel.showRow(crs);
        }
        catch (SQLException ex)
        {
            for (Throwable t : ex)
               t.printStackTrace();
        }
    }

    /**
      * Deletes current table row.
      */
    public void deleteRow()
    {
        if (crs == null) return;
        new SwingWorker<Void, Void>()
        {
           public Void doInBackground() throws SQLExcept
```

http://www.rejoiceblog.com/

```
            {
               crs.deleteRow();
               crs.acceptChanges(conn);
               if (crs.isAfterLast())
                  if (!crs.last()) crs = null;
               return null;
            }
            public void done()
            {
               dataPanel.showRow(crs);
            }
         }.execute();
    }
    /**
      * Saves all changes.
      */
    public void saveChanges()
      {
          if (crs == null) return;
          new SwingWorker<Void, Void>()
```

http://www.rejoiceblog.com/

```
212            {
213               public Void doInBackground() throws SQLExcep
214               {
215                  dataPanel.setRow(crs);
216                  crs.acceptChanges(conn);
217                  return null;
218               }
219            }.execute();
220         }
221
222         private void readDatabaseProperties() throws IOExce
223         {
224            props = new Properties();
225            try (InputStream in = Files.newInputStream(Paths
226            {
227               props.load(in);
228            }
229            String drivers = props.getProperty("jdbc.drivers
230            if (drivers != null) System.setProperty("jdbc.dr
```

http://www.rejoiceblog.com/

```
231         }
232
233         /**
234           * Gets a connection from the properties specified
235           * @return the database connection
236           */
237         private Connection getConnection() throws SQLExcept
238         {
239             String url = props.getProperty("jdbc.url");
240             String username = props.getProperty("jdbc.user
241             String password = props.getProperty("jdbc.pass
242
243            return DriverManager.getConnection(url, usernam
244         }
245     }
246
247     /**
248       * This panel displays the contents of a result set.
249       */
250     class DataPanel extends JPanel
```

http://www.rejoiceblog.com/

```
251   {
252       private java.util.List<JTextField> fields;
253
254      /**
255       * Constructs the data panel.
256       * @param rs the result set whose contents this pan
257      */
258    public DataPanel(RowSet rs) throws SQLException
259    {
260         fields = new ArrayList<>();
261         setLayout(new GridBagLayout());
262         var gbc = new GridBagConstraints();
263         gbc.gridwidth = 1;
264         gbc.gridheight = 1;
265
266        ResultSetMetaData rsmd = rs.getMetaData();
267        for (int i = 1; i <= rsmd.getColumnCount(); i++)
268        {
269           gbc.gridy = i - 1;
```

http://www.rejoiceblog.com/

```
270
271             String columnName = rsmd.getColumnLabel(i);
272             gbc.gridx = 0;
273             gbc.anchor = GridBagConstraints.EAST;
274             add(new JLabel(columnName), gbc);
275
276             int columnWidth = rsmd.getColumnDisplaySize(i
277             var tb = new JTextField(columnWidth);
278             if (!rsmd.getColumnClassName(i).equals("java.
279                tb.setEditable(false);
280
281             fields.add(tb);
282
283             gbc.gridx = 1;
284             gbc.anchor = GridBagConstraints.WEST;
285             add(tb, gbc);
286         }
287     }
288
289     /**
```

http://www.rejoiceblog.com/

```
     * Shows a database row by populating all text field
     */
    public void showRow(ResultSet rs)
    {
        try
        {
            if (rs == null) return;
            for (int i = 1; i <= fields.size(); i++)
            {
                String field = rs == null ? "" : rs.getSt
                JTextField tb = fields.get(i - 1);
                tb.setText(field);
            }
        }
        catch (SQLException ex)
         {
             for (Throwable t : ex)
                t.printStackTrace();
         }
```

http://www.rejoiceblog.com/

```
309         }
310
311         /**
312           * Updates changed data into the current row of th
313           */
314         public void setRow(RowSet rs) throws SQLException
315         {
316             for (int i = 1; i <= fields.size(); i++)
317             {
318                 String field = rs.getString(i);
319                 JTextField tb = fields.get(i - 1);
320                 if (!field.equals(tb.getText()))
321                     rs.updateString(i, tb.getText());
322             }
323             rs.updateRow();
324         }
325     }
```

java.sql.Connection 1.1

- DatabaseMetaData getMetaData()

http://www.rejoiceblog.com/

returns the metadata for the connection as a `DatabaseMetaData` object.

java.sql.DatabaseMetaData 1.1

- `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`

returns a description of all tables in a catalog that match the schema and table name patterns and the type criteria. (A *schema* describes a group of related tables and access permissions. A *catalog* describes a related group of schemas. These concepts are important for structuring large databases.)

The `catalog` and `schemaPattern` parameters can be `""` to retrieve those tables without a catalog or schema, or `null` to return tables regardless of catalog or schema.

The `types` array contains the names of the table types to include. Typical types are `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, and `SYNONYM`. If `types` is `null`, tables of all types are returned.

The result set has five columns, all of which are of type `String`.

Column	Name	Explanation
1	TABLE_CAT	Table catalog (may be null)
2	TABLE_SCHEM	Table schema (may be null)
3	TABLE_NAME	Table name
4	TABLE_TYPE	Table type
5	REMARKS	Comment on the table

- `int getJDBCMajorVersion()` 1.4

- `int getJDBCMinorVersion()` 1.4

returns the major or minor JDBC version numbers of the driver that established the database connection. For example, a JDBC 3.0 driver has major version number 3 and minor version number 0.

- `int getMaxConnections()`

returns the maximum number of concurrent connections allowed to this database.

http://www.rejoiceblog.com/

- `int getMaxStatements()`

returns the maximum number of concurrently open statements allowed per database connection, or `0` if the number is unlimited or unknown.

java.sql.ResultSet 1.1

- `ResultSetMetaData getMetaData()`

returns the metadata associated with the current `ResultSet` columns.

java.sql.ResultSetMetaData 1.1

- `int getColumnCount()`

returns the number of columns in the current `ResultSet` object.

- `int getColumnDisplaySize(int column)`

returns the maximum width of the column specified by the index parameter.

- `String getColumnLabel(int column)`

returns the suggested title for the column.

- `String getColumnName(int column)`

returns the column name associated with the column index specified.

5.9 TRANSACTIONS

You can group a set of statements to form a *transaction*. The transaction can be *committed* when all has gone well. Or, if an error has occurred in one of them, it can be *rolled back* as if none of the statements had been issued.

The major reason for grouping statements into transactions is *database integrity*. For example, suppose we want to transfer money from one bank account to another. Then, it is important that we simultaneously debit one account and credit another. If the

http://www.rejoiceblog.com/
system fails after debiting the first account but before crediting the other account, the debit needs to be undone.

If you group update statements into a transaction, the transaction either succeeds in its entirety and can be *committed,* or it fails somewhere in the middle. In that case, you can carry out a *rollback* and the database automatically undoes the effect of all updates that occurred since the last committed transaction.

5.9.1 Programming Transactions with JDBC

By default, a database connection is in *autocommit mode,* and each SQL statement is committed to the database as soon as it is executed. Once a statement is committed, you cannot roll it back. Turn off this default so you can use transactions:

```
conn.setAutoCommit(false);
```

Create a statement object in the normal way:

```
Statement stat = conn.createStatement();
```

Call `executeUpdate` any number of times:

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);

stat.executeUpdate(command3);

. . .
```

If all statements have been executed without error, call the `commit` method:

```
conn.commit();
```

However, if an error occurred, call

http://www.rejoiceblog.com/

```
conn.rollback();
```

Then, all statements since the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a `SQLException`.

5.9.2 Save Points

With some databases and drivers, you can gain finer-grained control over the rollback process by using *save points*. Creating a save point marks a point to which you can later return without having to abandon the entire transaction. For example,

```
Statement stat = conn.createStatement(); // start transaction

stat.executeUpdate(command1);

Savepoint svpt = conn.setSavepoint(); // set savepoint; rollb

stat.executeUpdate(command2);



if (. . .) conn.rollback(svpt); // undo effect of command2

. . . conn.commit();
```

When you no longer need a save point, you should release it:

```
conn.releaseSavepoint(svpt);
```

5.9.3 Batch Updates

Suppose a program needs to execute many `INSERT` statements to populate a database table. You can improve the performance of the program by using a *batch update*. In a batch update, a sequence of statements is collected and submitted as a batch.

http://www.rejoiceblog.com/

Note

Use the `supportsBatchUpdates` method of the `DatabaseMetaData` interface to find out if your database supports this feature.

The statements in a batch can be actions such as `INSERT`, `UPDATE`, or `DELETE` as well as data definition statements such as `CREATE TABLE` or `DROP TABLE`. An exception is thrown if you add a `SELECT` statement to a batch. (Conceptually, a `SELECT` statement makes no sense in a batch because it returns a result set without updating the database.)

To execute a batch, first create a `Statement` object in the usual way:

```
Statement stat = conn.createStatement();
```

Now, instead of calling `executeUpdate`, call the `addBatch` method:

```
String command = "CREATE TABLE . . ."

stat. addBatch(command);



while (. . .)

{

   command = "INSERT INTO . . . VALUES (" + . . . + ")";

   stat. addBatch (command);

}
```

Finally, submit the entire batch:

http://www.rejoiceblog.com/

```
int[] counts = stat. executeBatch();
```

The call to `executeBatch` returns an array of the row counts for all submitted statements.

For proper error handling in batch mode, treat the batch execution as a single transaction. If a batch fails in the middle, you want to roll back to the state before the beginning of the batch.

First, turn the autocommit mode off, then collect the batch, execute it, commit it, and finally restore the original autocommit mode:

```
boolean autoCommit = conn.getAutoCommit();

conn.setAutoCommit(false);

Statement stat = conn.getStatement();

. . .

// keep calling stat.addBatch(. . .);

. . .

stat.executeBatch();

conn.commit(); conn.setAutoCommit(autoCommit);
```

java.sql.Connection 1.1

- `boolean getAutoCommit()`
- `void setAutoCommit(boolean b)`

gets or sets the autocommit mode of this connection to `b`. If autocommit is `true`, all statements are committed as soon as their execution is completed.

http://www.rejoiceblog.com/

- `void commit()`

commits all statements that were issued since the last commit.

- `void rollback()`

undoes the effect of all statements that were issued since the last commit.

- `Savepoint setSavepoint()` **1.4**

- `Savepoint setSavepoint(String name)` **1.4**

sets an unnamed or named save point.

- `void rollback(Savepoint svpt)` **1.4**

rolls back until the given save point.

- `void releaseSavepoint(Savepoint svpt)` **1.4**

releases the given save point.

java.sql.Savepoint 1.4

- `int getSavepointId()`

gets the ID of this unnamed save point, or throws a `SQLException` if this is a named save point.

- `String getSavepointName()`

gets the name of this save point, or throws a `SQLException` if this is an unnamed save point.

java.sql.Statement 1.1

- `void addBatch(String command)` **1.2**

adds the command to the current batch of commands for this statement.

http://www.rejoiceblog.com/

- `int[] executeBatch()` **1.2**

- `long[] executeLargeBatch()` **8**

executes all commands in the current batch. Each value in the returned array corresponds to one of the batch statements. If it is non-negative, it is a row count. If it is the value `SUCCESS_NO_INFO`, the statement succeeded, but no row count is available. If it is `EXECUTE_FAILED`, the statement failed.

java.sql.DatabaseMetaData **1.1**

- `boolean supportsBatchUpdates()` **1.2**

returns `true` if the driver supports batch updates.

5.9.4 Advanced SQL Types

Table 5.9 lists the SQL data types supported by JDBC and their equivalents in the Java programming language.

Table 5.8 SQL Data Types and Their Corresponding Java Types

SQL Data Type	Java Data Type
INTEGER or INT	int
SMALLINT	short
NUMERIC(m,n), DECIMAL(m,n) or DEC(m,n)	java.math.BigDecimal
FLOAT(n)	double

http://www.rejoiceblog.com/

REAL	float
DOUBLE	double
CHARACTER(n) or CHAR(n)	String
VARCHAR(n), LONG VARCHAR	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
ROWID	java.sql.RowId
NCHAR(n), NVARCHAR(n), LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

A SQL `ARRAY` is a sequence of values. For example, in a `Student` table, you can have a `Scores` column that is an `ARRAY OF INTEGER`. The `getArray` method returns an object of the interface type `java.sql.Array`. That interface has methods to fetch the array values.

When you get a LOB or an array from a database, the actual contents are fetched from the database only when you request individual values. This is a useful performance enhancement, as the data can be quite voluminous.

Some databases support `ROWID` values that describe the location of a row so that it can be retrieved very rapidly. JDBC 4 introduced an interface `java.sql.RowId` and the methods to supply the row ID in queries and retrieve it from results.

A *national character string* (`NCHAR` and its variants) stores strings in a local character encoding and sorts them using a local sorting convention. JDBC 4 provided methods for converting between Java `String` objects and national character strings in queries and results.

Some databases can store user-defined structured types. JDBC 3 provided a mechanism for automatically mapping structured SQL types to Java objects.

Some databases provide native storage for XML data. JDBC 4 introduced a `SQLXML`

http://www.rejoiceblog.com/

interface that can mediate between the internal XML representation and the DOM `Source/Result` interfaces, as well as binary streams. See the API documentation for the `SQLXML` class for details.

We do not discuss these advanced SQL types any further. You can find more information on these topics in the *JDBC API Tutorial and Reference* and the JDBC specification.

5.10 CONNECTION MANAGEMENT IN WEB AND ENTERPRISE APPLICATIONS

The simplistic database connection setup with a `database.properties` file, as described in the preceding sections, is suitable for small test programs but won't scale for larger applications.

When a JDBC application is deployed in a web or enterprise environment, the management of database connections is integrated with the JNDI. The properties of data sources across the enterprise can be stored in a directory. Using a directory allows for centralized management of user names, passwords, database names, and JDBC URLs.

In such an environment, you can use the following code to establish a database connection:

```
var jndiContext = new InitialContext();

var source = (DataSource) jndiContext.lookup("java:comp/env/j

Connection conn = source.getConnection();
```

Note that the `DriverManager` is no longer involved. Instead, the JNDI service locates a *data source*. A data source is an interface that allows for simple JDBC connections as well as more advanced services, such as executing distributed transactions that involve multiple databases. The `DataSource` interface is defined in the `javax.sql` standard extension package.

http://www.rejoiceblog.com/

Note

In a Java EE container, you don't even have to program the JNDI lookup. Simply use the `Resource` annotation on a `DataSource` field, and the data source reference will be set when your application is loaded:

```
@Resource(name="jdbc/corejava")

private DataSource source;
```

Of course, the data source needs to be configured somewhere. If you write database programs that execute in a servlet container such as Apache Tomcat or in an application server such as GlassFish, place the database configuration (including the JNDI name, JDBC URL, user name, and password) in a configuration file, or set it in an admin GUI.

Management of user names and logins is just one of the issues that require special attention. Another issue involves the cost of establishing database connections. Our sample database programs used two strategies for obtaining a database connection. The `QueryDB` program in Listing 5.3 established a single database connection at the start of the program and closed it at the end of the program. The `ViewDB` program in Listing 5.4 opened a new connection whenever one was needed.

However, neither of these approaches is satisfactory. Database connections are a finite resource. If a user walks away from an application for some time, the connection should not be left open. Conversely, obtaining a connection for each query and closing it afterward is very costly.

The solution is to *pool* connections. This means that database connections are not physically closed but are kept in a queue and reused. Connection pooling is an important service, and the JDBC specification provides hooks for implementors to supply it. However, the JDK itself does not provide any implementation, and database vendors don't usually include one with their JDBC drivers either. Instead, vendors of web containers and application servers supply connection pool implementations.

Using a connection pool is completely transparent to the programmer. Acquire a connection from a source of pooled connections by obtaining a data source and calling

http://www.rejoiceblog.com/
`getConnection`. When you are done using the connection, call `close`. That doesn't close the physical connection but tells the pool that you are done using it. The connection pool typically makes an effort to pool prepared statements as well.

You have now learned about the JDBC fundamentals and know enough to implement simple database applications. However, as we mentioned at the beginning of this chapter, databases are complex and quite a few advanced topics are beyond the scope of this introductory chapter. For an overview of advanced JDBC capabilities, refer to the *JDBC API Tutorial and Reference* or the JDBC specification.

In this chapter, you have learned how to work with relational databases in Java. The next chapter covers the Java 8 date and time library.

http://www.rejoiceblog.com/

Chapter 6 The Date and Time API

In this chapter

- 6.1 The Time Line
- 6.2 Local Dates
- 6.3 Date Adjusters
- 6.4 Local Time
- 6.5 Zoned Time
- 6.6 Formatting and Parsing
- 6.7 Interoperating with Legacy Code

Time flies like an arrow, and we can easily set a starting point and count forward and backward in seconds. So why is it so hard to deal with time? The problem is humans. All would be easy if we could just tell each other: "Meet me at 1523793600, and don't be late!" But we want time to relate to daylight and the seasons. That's where things get complicated. Java 1.0 had a `Date` class that was, in hindsight, naïve, and had most of its methods deprecated in Java 1.1 when a `Calendar` class was introduced. Its API wasn't stellar, its instances were mutable, and it didn't deal with issues such as leap seconds. The third time is a charm, and the `java.time` API introduced in Java 8 has remedied the flaws of the past and should serve us for quite some time. In this chapter, you will learn what makes time computations so vexing, and how the Date and Time API solves these issues.

6.1 THE TIME LINE

Historically, the fundamental time unit—the second—was derived from Earth's rotation around its axis. There are 24 hours or 24 × 60 × 60 = 86400 seconds in a full revolution, so it seems just a question of astronomical measurements to precisely

http://www.rejoiceblog.com/

define a second. Unfortunately, Earth wobbles slightly, and a more precise definition was needed. In 1967, a new precise definition of a second, matching the historical definition, was derived from an intrinsic property of atoms of caesium-133. Since then, a network of atomic clocks keeps the official time.

Ever so often, the official time keepers synchronize the absolute time with the rotation of Earth. At first, the official seconds were slightly adjusted, but starting in 1972, "leap seconds" were occasionally inserted. (In theory, a second might need to be removed once in a while, but that has not yet happened.) There is talk of changing the system again. Clearly, leap seconds are a pain, and many computer systems instead use "smoothing" where time is artificially slowed down or sped up just before the leap second, keeping 86,400 seconds per day. This works because the local time on a computer isn't all that precise, and computers are used to synchronizing themselves with an external time service.

The Java Date and Time API specification requires that Java uses a time scale that:

• Has 86,400 seconds per day.

• Exactly matches the official time at noon each day.

• Closely matches it elsewhere, in a precisely defined way.

The static method call `Instant.now()` gives the current instant. You can compare two instants with the `equals` and `compareTo` methods in the usual way, so you can use instants as timestamps.

To find out the difference between two instants, use the static method `Duration.between`. For example, here is how you can measure the running time of an algorithm:

```
Instant start = Instant.now();

runAlgorithm();

Instant end = Instant.now();

Duration timeElapsed = Duration.between(start, end);

long millis = timeElapsed.toMillis();
```

http://www.rejoiceblog.com/

A `Duration` is the amount of time between two instants. You can get the length of a `Duration` in conventional units by calling `toNanos`, `toMillis`, `toSeconds`, `toMinutes`, `toHours`, or `toDays`.

> **Note**
>
> In Java 8, you had to call `getSeconds` instead of `toSeconds`.

If you need nanosecond precision, be aware of overflow. A `long` value can hold almost 300 years of nanoseconds. If your durations are shorter than that, simply convert them to nanoseconds. You can use longer durations—a `Duration` object stores the number of seconds in a `long`, and the number of nanoseconds in an additional `int`. The `Duration` API has a number of methods, shown at the end of this section, for carrying out arithmetic.

For example, if you want to check whether an algorithm is at least ten times faster than another, you can compute

```
Duration timeElapsed2 = Duration.between(start2, end2);

boolean overTenTimesFaster =

   timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegativ
```

This is just to show the syntax. Since the algorithms aren't going to run for hundreds of years, you can simply use

```
boolean overTenTimesFaster = timeElapsed.toNanos() * 10 < tim
```

> **Note**

http://www.rejoiceblog.com/

> The `Instant` and `Duration` classes are immutable, and all methods, such as `multipliedBy` or `minus`, return a new instance.

In the example program in Listing 6.1, you can see how to use the `Instant` and `Duration` classes for timing two algorithms.

Listing 6.1 `timeline/TimeLine.java`

```
package timeline;

/**
 * @version 1.0 2016-05-10
 * @author Cay Horstmann
 */

import java.time.*;
import java.util.*;
import java.util.stream.*;

public class Timeline
{
   public static void main(String[] args)
   {
```

http://www.rejoiceblog.com/

```
        Instant start = Instant.now();
        runAlgorithm();
        Instant end = Instant.now();
        Duration timeElapsed = Duration.between(start, end)
        long millis = timeElapsed.toMillis();
        System.out.printf("%d milliseconds\n", millis);

        Instant start2 = Instant.now();
        runAlgorithm2();
        Instant end2 = Instant.now();
        Duration timeElapsed2 = Duration.between(start2, en
        System.out.printf("%d milliseconds\n", timeElapsed2
        boolean overTenTimesFaster = timeElapsed.multiplied
              .minus(timeElapsed2).isNegative();
        System.out.printf("The first algorithm is %smore th
              overTenTimesFaster ? "" : "not ");
    }

    public static void runAlgorithm()
```

http://www.rejoiceblog.com/

```
    {
        int size = 10;
        List<Integer> list = new Random().ints().map(i -> i
              .boxed().collect(Collectors.toList());
        Collections.sort(list);
        System.out.println(list);
    }

    public static void runAlgorithm2()
    {
        int size = 10;
        List<Integer> list = new Random().ints().map(i -> i
           .boxed().collect(Collectors.toList());
        while (!IntStream.range(1, list.size()).allMatch(
           i -> list.get(i - 1).compareTo(list.get(i)) <= 0
           Collections.shuffle(list);
        System.out.println(list);
    }
}
```

http://www.rejoiceblog.com/

java.time.Instant 8

- `static Instant now()`

gets the current instant from the best available system clock.

- `Instant plus(TemporalAmount amountToAdd)`

- `Instant minus(TemporalAmount amountToSubtract)`

yields an instant that is the given amount away from this `Instant`. The classes `Duration` and `Period` (see Section 6.2, "Local Dates," on p. 358) implement the `TemporalAmount` interface.

- `Instant (plus|minus)(Nanos|Millis|Seconds)(long number)`

yields an `Instant` that is the given number of nanoseconds, milliseconds, or seconds away from this `Instant`.

java.time.Duration 8

- `static Duration of(Nanos|Millis|Seconds|Minutes|Hours|Days)(long number)`

yields a duration of the given number of time units.

- `static Duration between(Temporal startInclusive, Temporal endExclusive)`

yields a duration between the given points in time. The `Temporal` interface is implemented by the `Instant` class as well as `LocalDate/LocalDateTime/LocalTime` (see Section 6.4, "Local Time," on p. 365) and `ZonedDateTime` (see Section 6.5, "Zoned Time," on p. 366).

- `long toNanos()`

- `long toMillis()`

- `long toSeconds()` 9

http://www.rejoiceblog.com/

- `long toMinutes()`

- `long toHours()`

- `long toSeconds()`

- `long toSeconds()`

- `long toDays()`

gets the number of the time units in the method name for this Duration

- `int to(Nanos|Millis|Seconds|Minutes|Hours)Part()` 9

- `long to(Days|Hours|Minutes|Seconds|Millis|Nanos)Part()` 9

The part of the given time unit in this Duration. For example, in a duration of 100 seconds, the minutes part is 1 and the seconds part is 40.

- `Instant plus(TemporalAmount amountToAdd)`

- `Instant minus(TemporalAmount amountToSubtract)`

yields an instant that is the given amount away from this `Instant`. The classes `Duration` and `Period` (see Section 6.2, "Local Dates," on p. 358) implement the `TemporalAmount` interface.

- `Duration multipliedBy(long multiplicand)`

- `Duration dividedBy(long divisor)`

- `Duration negated()`

yields a duration that is obtained by multiplying or dividing this Duration by the given amount, or by—1.

- `boolean isZero()`

- `boolean isNegative()`

returns true if this `Duration` is zero or negative.

- `Duration (plus|minus)(Nanos|Millis|Seconds|Minutes|Hours|Days)`

http://www.rejoiceblog.com/

```
(long number)
```

yields a Duration obtained by adding or subtracting the given number of time units.

6.2 LOCAL DATES

Now let us turn from absolute time to human time. There are two kinds of human time in the Java API, *local date/time* and *zoned time*. Local date/time has a date and/or time of day, but no associated time zone information. An example of a local date is June 14, 1903 (the day on which Alonzo Church, inventor of the lambda calculus, was born). Since that date has neither a time of day nor time zone information, it does not correspond to a precise instant of time. In contrast, July 16, 1969, 09:32:00 EDT (the launch of Apollo 11) is a zoned date/time, representing a precise instant on the time line.

There are many calculations where time zones are not required, and in some cases they can even be a hindrance. Suppose you schedule a meeting every week at 10:00. If you add 7 days (that is, 7 × 24 × 60 × 60 seconds) to the last zoned time, and you happen to cross the daylight savings time boundary, the meeting will be an hour too early or too late!

For that reason, the API designers recommend that you do not use zoned time unless you really want to represent absolute time instances. Birthdays, holidays, schedule times, and so on are usually best represented as local dates or times.

A `LocalDate` is a date with a year, month, and day of the month. To construct one, you can use the `now` or `of` static methods:

```
LocalDate today = LocalDate.now(); // Today's date

LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);

alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);

   // Uses the Month enumeration
```

Unlike the irregular conventions in UNIX and `java.util.Date`, where months are zero-based and years are counted from 1900, you supply the usual numbers for the

http://www.rejoiceblog.com/

month of year. Alternatively, you can use the `Month` enumeration.

The API notes at the end of this section show the most useful methods for working with `LocalDate` objects.

For example, *Programmer's Day* is the 256th day of the year. Here is how you can easily compute it:

```
LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(

   // September 13, but in a leap year it would be September
```

Recall that the difference between two time instants is a `Duration`. The equivalent for local dates is a `Period`, which expresses a number of elapsed years, months, or days. You can call `birthday.plus(Period.ofYears(1))` to get the birthday next year. Of course, you can also just call `birthday.plusYears(1)`. But `birthday.plus(Duration.ofDays(365))` won't produce the correct result in a leap year.

The `until` method yields the difference between two local dates. For example,

```
independenceDay.until(christmas)
```

yields a period of 5 months and 21 days. That is actually not terribly useful because the number of days per month varies. To find the number of days, use

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 days
```

Caution

Some methods in the `LocalDate` API could potentially create nonexistent dates. For example, adding one month to January 31 should not yield February 31. Instead of throwing an exception, these methods return the last valid day of the month. For example,

http://www.rejoiceblog.com/

```
LocalDate.of(2016, 1, 31).plusMonths(1)
```

and

```
LocalDate.of(2016, 3, 31).minusMonths(1)
```

yield February 29, 2016.

The `getDayOfWeek` yields the weekday, as a value of the `DayOfWeek` enumeration. `DayOfWeek.MONDAY` has the numerical value `1`, and `DayOfWeek.SUNDAY` has the value `7`. For example,

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

yields `1`. The `DayOfWeek` enumeration has convenience methods `plus` and `minus` to compute weekdays modulo `7`. For example, `DayOfWeek.SATURDAY.plus(3)` yields `DayOfWeek.TUESDAY`.

Note

The weekend days actually come at the end of the week. This is different from `java.util.Calendar` where Sunday has value `1` and Saturday value `7`.

Java 9 adds two useful methods `datesUntil` that yield streams of `LocalDate` objects.

```
LocalDate start = LocalDate.of(2000, 1, 1);

LocalDate endExclusive = LocalDate.now();

Stream<LocalDate> allDays = start.datesUntil(endExclusive);
```

http://www.rejoiceblog.com/

```
Stream<LocalDate> firstDaysInMonth = start.datesUntil(endExcl
```

In addition to `LocalDate`, there are also classes `MonthDay`, `YearMonth`, and `Year` to describe partial dates. For example, December 25 (with the year unspecified) can be represented as a `MonthDay`.

The example program in Listing 6.2 shows how to work with the `LocalDate` class.

Listing 6.2 `localdates/LocalDates.java`

```
package localdates;

/**
 * @version 1.0 2016-05-10
 * @author Cay Horstmann
 */
import java.time.*;
import java.time.temporal.*;
import java.util.stream.*;

public class LocalDates
{
   public static void main(String[] args)
```

http://www.rejoiceblog.com/

```
    {
        LocalDate today = LocalDate.now(); // Today's date
        System.out.println("today: " + today);

        LocalDate alonzosBirthday = LocalDate.of(1903, 6, 1
        alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14
        // Uses the Month enumeration
        System.out.println("alonzosBirthday: " + alonzosBir

        LocalDate programmersDay = LocalDate.of(2018, 1, 1)
        // September 13, but in a leap year it would be Sep
        System.out.println("programmersDay: " + programmers

        LocalDate independenceDay = LocalDate.of(2018, Mont
        LocalDate christmas = LocalDate.of(2018, Month.DECE

        System.out
            .println("Until christmas: " + independenceDay.u
        System.out.println("Until christmas: "
            + independenceDay.until(christmas, ChronoUnit.DA
```

http://www.rejoiceblog.com/

```

        System.out.println(LocalDate.of(2016, 1, 31).plusMo
        System.out.println(LocalDate.of(2016, 3, 31).minusM

        DayOfWeek startOfLastMillennium = LocalDate.of(1900
        System.out.println("startOfLastMillennium: " + star
        System.out.println(startOfLastMillennium.getValue()
        System.out.println(DayOfWeek.SATURDAY.plus(3));

        LocalDate start = LocalDate.of(2000, 1, 1);
        LocalDate endExclusive = LocalDate.now();
        Stream<LocalDate> firstDaysInMonth = start.datesUnt
        System.out.println("firstDaysInMonth: " + firstDays
    }
}
```

java.time.LocalDate 8

- `static LocalDate now()`

gets the current `LocalDate`.

http://www.rejoiceblog.com/

- `static LocalDate of(int year, int month, int dayOfMonth)`

- `static LocalDate of(int year, Month month, int dayOfMonth)`

yields a local date with the given year, month (as integer between 1 and 12 or a value of the `Month` enumeration), and day of the month (between 1 and 31).

- `LocalDate (plus|minus)(Days|Weeks|Months|Years)(long number)`

yields a `LocalDate` obtained by adding or subtracting the given number of time units.

- `LocalDate plus(TemporalAmount amountToAdd)`

- `LocalDate minus(TemporalAmount amountToSubtract)`

yields an instant that is the given amount away from this `Instant`. The classes `Duration` and `Period`implement the `TemporalAmount` interface.

- `LocalDate withDayOfMonth(int dayOfMonth)`

- `LocalDate withDayOfYear(int dayOfYear)`

- `LocalDate withMonth(int month)`

- `LocalDate withYear(int year)`

returns a new `LocalDate` with the day of month, day of year, month, or year changed to the given value.

- `int getDayOfMonth()`

gets the day of the month (between 1 and 31).

- `int getDayOfYear()`

gets the day of the year (between 1 and 366).

- `DayOfWeek getDayOfWeek()`

gets the day of the week, returning a value of the `DayOfWeek` enumeration.

- `Month getMonth()`

http://www.rejoiceblog.com/

- `int getMonthValue()`

gets the month as a value of the `Month` enumeration, or as a number between 1 and 12.

- `int getYear()`

gets the year, between –999,999,999 and 999,999,999.

- `Period until(ChronoLocalDate endDateExclusive)`

gets the period until the given end date. The `ChronoLocalDate` interface is implemented by `LocalDate` and date classes for non-Gregorian calendars.

- `boolean isBefore(ChronoLocalDate other)`

- `boolean isAfter(ChronoLocalDate other)`

returns `true` if this date is before or after the given date.

- `boolean isLeapYear()`

returns `true` if the year is a leap year—that is, if it is divisible by 4 but not by 100, or divisible by 400. The algorithm is applied for all past years, even though that is historically inaccurate. (Leap years were invented in the year –46, and the rules involving divisibility by 100 and 400 were introduced in the Gregorian calendar reform of 1582. The reform took over 300 years to become universal.)

- `Stream<LocalDate> datesUntil(LocalDate endExclusive)` 9

- `Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step)` 9

yields a stream of dates from this `LocalDate` until the end, with step size 1 or the given period.

java.time.Period 8

- `static Period of(int years, int months, int days)`

- `Period of(Days|Weeks|Months|Years)(int number)`

http://www.rejoiceblog.com/

yields a `Perid` with the given number of time units.

• `int get(Days|Months|Years)()`

gets the days, months, or years of this `Period`.

• `Period (plus|minus)(Days|Months|Years)(long number)`

yields a `LocalDate` obtained by adding or subtracting the given number of time units.

• `Period plus(TemporalAmount amountToAdd)`

• `Period minus(TemporalAmount amountToSubtract)`

yields an instant that is the given amount away from this `Instant`. The classes `Duration` and `Period`implement the `TemporalAmount` interface.

• `Period with(Days|Months|Years)(int number)`

returns a new `Period` with the days, months, or years changed to the given number.

6.3 DATE ADJUSTERS

For scheduling applications, you often need to compute dates such as "the first Tuesday of every month." The `TemporalAdjusters` class provides a number of static methods for common adjustments. You pass the result of an adjust-

ment method to the `with` method. For example, the first Tuesday of a month can be computed like this:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(

   TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

As always, the `with` method returns a new `LocalDate` object without modifying the original. The API notes at the end of this section show the available adjusters.

You can also make your own adjuster by implementing the `TemporalAdjuster` interface. Here is an adjuster for computing the next weekday.

http://www.rejoiceblog.com/

```
TemporalAdjuster NEXT_WORKDAY = w ->
    {
      var result = (LocalDate) w;
      do
      {
         result = result.plusDays(1);
      }
      while (result.getDayOfWeek().getValue() >= 6);
      return result;
    };
LocalDate backToWork = today.with(NEXT_WORKDAY);
```

Note that the parameter of the lambda expression has type `Temporal`, and it must be cast to `LocalDate`. You can avoid this cast with the `ofDateAdjuster` method that expects a lambda of type `UnaryOperator<LocalDate>`.

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdjus
    {
      LocalDate result = w; // No cast
      do
      {
       result = result.plusDays(1);
```

http://www.rejoiceblog.com/

```
        }

        while (result.getDayOfWeek().getValue() >= 6);

        return result;

      });
```

java.time.LocalDate 9

- `LocalDate with(TemporalAdjuster adjuster)`

Returns the result of adjusting this date by the given adjuster.

java.time.temporal.TemporalAdjusters 9

- `static TemporalAdjuster next(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster previous(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek)`

returns an adjuster that adjusts a date to the given day of the week.

- `static TemporalAdjuster dayOfWeekInMonth(int n, DayOfWeek dayOfWeek)`
- `static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek)`

returns an adjuster that adjusts a date to the `n`th or last given weekday on the month.

- `static TemporalAdjuster firstDayOfMonth()`
- `static TemporalAdjuster firstDayOfNextMonth()`

http://www.rejoiceblog.com/

- `static TemporalAdjuster firstDayOfYear()`
- `static TemporalAdjuster firstDayOfNextYear()`
- `static TemporalAdjuster lastDayOfMonth()`
- `static TemporalAdjuster lastDayOfYear()`

returns an adjuster that adjusts a date to the given day of the month or year.

6.4 LOCAL TIME

A `LocalTime` represents a time of day, such as 15:30:00. You can create an instance with the `now` or `of` methods:

```
LocalTime rightNow = LocalTime.now();

LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(
```

The API notes show common operations with local times. The `plus` and `minus` operations wrap around a 24-hour day. For example,

```
LocalTime wakeup = bedtime.plusHours(8); // wakeup is 6:30:00
```

Note

`LocalTime` doesn't concern itself with AM/PM. That silliness is left to a formatter—see Section 6.6, "Formatting and Parsing," on p. 371.

There is a `LocalDateTime` class representing a date and time. That class is suitable for storing points in time in a fixed time zone—for example, for a schedule of classes or events. However, if you need to make calculations that span the daylight savings time, or if you need to deal with users in different time zones, you should use the

http://www.rejoiceblog.com/

`ZonedDateTime` class that we discuss next.

java.time.LocalTime 8

- `static LocalTime now()`

gets the current `LocalTime`.

- `static LocalTime of(int hour, int minute)`

- `static LocalTime of(int hour, int minute, int second)`

- `static LocalTime of(int hour, int minute, int second, int nanoOfSecond)`

yields a local time with the given hour (between 0 and 23), minute and second (between 0 and 59) and nanosecond (between 0 and 999,999,999).

- `LocalTime (plus|minus)(Hours|Minutes|Seconds|Nanos)(long number)`

yields a `LocalTime` obtained by adding or subtracting the given number of time units.

- `LocalTime plus(TemporalAmount amountToAdd)`

- `LocalTime minus(TemporalAmount amountToSubtract)`

yields an instant that is the given amount away from this `Instant`.

- `LocalTime with(Hour|Minute|Second|Nano)(int value)`

returns a new `LocalTime` with the hour, minute, second, or nanoseconds changed to the given value.

- `int getHour()`

gets the hour (between 0 and 23).

- `int getMinute()`

- `int getSecond()`

gets the minute or second (between 0 and 59).

http://www.rejoiceblog.com/

- `int getNano()`

gets the nanoseconds (between 0 and 999,999,999).

- `int toSecondOfDay()`

- `long toNanoOfDay()`

yields the seconds or nanoseconds since midnight.

- `boolean isBefore(LocalTime other)`

- `boolean isAfter(LocalTime other)`

returns `true` if this date is before or after the given date.

6.5 ZONED TIME

Time zones, perhaps because they are an entirely human creation, are even messier than the complications caused by the earth's irregular rotation. In a rational world, we'd all follow the clock in Greenwich, and some of us would

eat our lunch at 02:00, others at 22:00. Our stomachs would figure it out. This is actually done in China, which spans four conventional time zones. Elsewhere, we have time zones with irregular and shifting boundaries and, to make matters worse, the daylight savings time.

As capricious as the time zones may appear to the enlightened, they are a fact of life. When you implement a calendar application, it needs to work for people who fly from one country to another. When you have a conference call at 10:00 in New York, but happen to be in Berlin, you expect to be alerted at the correct local time.

The Internet Assigned Numbers Authority (IANA) keeps a database of all known time zones around the world (`www.iana.org/time-zones`), which is updated several times per year. The bulk of the updates deals with the changing rules for daylight savings time. Java uses the IANA database.

Each time zone has an ID, such as `America/New_York` or `Europe/Berlin`. To find out all available time zones, call `ZoneId.getAvailableZoneIds`. At the time of this writing, there were almost 600 IDs.

http://www.rejoiceblog.com/

Given a time zone ID, the static method `ZoneId.of(id)` yields a `ZoneId` object. You can use that object to turn a `LocalDateTime` object into a `ZonedDateTime` object by calling `local.atZone(zoneId)`, or you can construct a `ZonedDateTime` by calling the static method `ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId)`. For example,

```
ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16,
   ZoneId.of("America/New_York"));
   // 1969-07-16T09:32-04:00[America/New_York]
```

This is a specific instant in time. Call `apollo11launch.toInstant` to get the `Instant`. Conversely, if you have an instant in time, call `instant.atZone(ZoneId.of("UTC"))` to get the `ZonedDateTime` at the Greenwich Royal Observatory, or use another `ZoneId` to get it elsewhere on the planet.

Note

UTC stands for “Coordinated Universal Time,” and the acronym is a compromise between the aforementioned English and the French “Temps Universel Coordiné,” having the distinction of being incorrect in either language. UTC is the time at the Greenwich Royal Observatory, without daylight savings time.

Many of the methods of `ZonedDateTime` are the same as those of `LocalDateTime` (see the API notes at the end of this section). Most are straightforward, but daylight savings time introduces some complications.

When daylight savings time starts, clocks advance by an hour. What happens when you construct a time that falls into the skipped hour? For example, in 2013, Central Europe switched to daylight savings time on March 31 at 2:00. If you try to construct nonexistent time March 31 2:30, you actually get 3:30.

http://www.rejoiceblog.com/

```
ZonedDateTime skipped = ZonedDateTime.of(
   LocalDate.of(2013, 3, 31),
   LocalTime.of(2, 30),
   ZoneId.of("Europe/Berlin"));
   // Constructs March 31 3:30
```

Conversely, when daylight time ends, clocks are set back by an hour, and there are two instants with the same local time! When you construct a time within that span, you get the earlier of the two.

```
ZonedDateTime ambiguous = ZonedDateTime.of(
   LocalDate.of(2013, 10, 27), // End of daylight savings tim
   LocalTime.of(2, 30),
   ZoneId.of("Europe/Berlin"));
   // 2013-10-27T02:30+02:00[Europe/Berlin]
ZonedDateTime anHourLater = ambiguous.plusHours(1);
   // 2013-10-27T02:30+01:00[Europe/Berlin]
```

An hour later, the time has the same hours and minutes, but the zone offset has changed.

You also need to pay attention when adjusting a date across daylight savings time boundaries. For example, if you set a meeting for next week, don't add a duration of seven days:

```
ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
```

http://www.rejoiceblog.com/

```
// Caution! Won't work with daylight savings time
```

Instead, use the `Period` class.

```
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); /
```

Caution

There is also an `OffsetDateTime` class that represents times with an offset from UTC, but without time zone rules. That class is intended for specialized applications that specifically require the absence of those rules, such as certain network protocols. For human time, use `ZonedDateTime`.

The example program in Listing 6.3 demonstrates the `ZonedDateTime` class.

Listing 6.3 `zonedtimes/ZonedTimes.java`

```
package zonedtimes;

/**
 * @version 1.0 2016-05-10
 * @author Cay Horstmann
 */

import java.time.*;
```

http://www.rejoiceblog.com/

```
 9
10  public class ZonedTimes
11  {
12     public static void main(String[] args)
13     {
14        ZonedDateTime apollo11launch = ZonedDateTime.of(196
15              ZoneId.of("America/New_York"));
16        // 1969-07-16T09:32-04:00[America/New_York]
17        System.out.println("apollo11launch: " + apollo11lau
18
19        Instant instant = apollo11launch.toInstant();
20        System.out.println("instant: " + instant);
21
22        ZonedDateTime zonedDateTime = instant.atZone(ZoneId
23        System.out.println("zonedDateTime: " + zonedDateTim
24
25        ZonedDateTime skipped = ZonedDateTime.of(LocalDate.
26              LocalTime.of(2, 30), ZoneId.of("Europe/Berlin
27        // Constructs March 31 3:30
```

http://www.rejoiceblog.com/

```
        System.out.println("skipped: " + skipped);

        ZonedDateTime ambiguous = ZonedDateTime.of(LocalDat
        // End of daylight savings time
              LocalTime.of(2, 30), ZoneId.of("Europe/Berlin
        // 2013-10-27T02:30+02:00[Europe/Berlin]
        ZonedDateTime anHourLater = ambiguous.plusHours(1);
        // 2013-10-27T02:30+01:00[Europe/Berlin]
        System.out.println("ambiguous: " + ambiguous);
        System.out.println("anHourLater: " + anHourLater);

        ZonedDateTime meeting = ZonedDateTime.of(LocalDate.
              LocalTime.of(14, 30), ZoneId.of("America/Los_
        System.out.println("meeting: " + meeting);
        ZonedDateTime nextMeeting = meeting.plus(Duration.o
        // Caution! Won't work with daylight savings time
        System.out.println("nextMeeting: " + nextMeeting);
        nextMeeting = meeting.plus(Period.ofDays(7)); // OK
        System.out.println("nextMeeting: " + nextMeeting);
    }
```

http://www.rejoiceblog.com/

```
}
```

java.time.ZonedDateTime 8

- `static ZonedDateTime now()`

Gets the current `ZonedDateTime`.

- `static ZonedDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneId zone)`

- `static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)`

- `static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)`

- `static ZonedDateTime ofInstant(Instant instant, ZoneId zone)`

yields a `ZonedDateTime` with the given parameters and time zone.

- `ZonedDateTime (plus|minus) (Days|Weeks|Months|Years|Hours|Minutes|Seconds|Nanos)(long number)`

yields a `ZonedDateTime` obtained by adding or subtracting the given number of time units.

- `ZonedDateTime plus(TemporalAmount amountToAdd)`

- `ZonedDateTime minus(TemporalAmount amountToSubtract)`

yields an instant that is the given amount away from this `Instant`.

- `ZonedDateTime with(DayOfMonth|DayOfYear|Month|Year|Hour|Minute|Second|Nano)(int value)`

http://www.rejoiceblog.com/

returns a new `ZonedDateTime` with the given temporal unit replaced by the given value.

- `ZonedDateTime withZoneSameInstant(ZoneId zone)`

- `ZonedDateTime withZoneSameLocal(ZoneId zone)`

returns a new `ZonedDateTime` in the given time zone, either representing the same instant or the same local time.

- `int getDayOfMonth()`

gets the day of the month (between 1 and 31).

- `int getDayOfYear()`

gets the day of the year (between 1 and 366).

- `DayOfWeek getDayOfWeek()`

gets the day of the week, returning a value of the `DayOfWeek` enumeration.

- `Month getMonth()`

- `int getMonthValue()`

gets the month as a value of the `Month` enumeration, or as a number between 1 and 12.

- `int getYear()`

gets the year, between –999,999,999 and 999,999,999.

- `int getHour()`

gets the hour (between 0 and 23).

- `int getMinute()`

- `int getSecond()`

gets the minute or second (between 0 and 59).

- `int getNano()`

http://www.rejoiceblog.com/

gets the nanoseconds (between 0 and 999,999,999).

- `public ZoneOffset getOffset()`

gets the offset from UTC. Offsets can vary from –12:00 to +14:00. Some time zones have fractional offsets. Offsets change with daylight savings time.

- `LocalDate toLocalDate()`
- `LocalTime toLocalTime()`
- `LocalDateTime toLocalDateTime()`
- `Instant toInstant()`

yields the local date, time, or date/time, or the corresponding instant.

- `boolean isBefore(ChronoZonedDateTime other)`
- `boolean isAfter(ChronoZonedDateTime other)`

returns `true` if this zoned date/time is before or after the given zoned date/time.

6.6 FORMATTING AND PARSING

The `DateTimeFormatter` class provides three kinds of formatters to print a date/time value:

- Predefined standard formatters (see Table 6.1)
- Locale-specific formatters
- Formatters with custom patterns

To use one of the standard formatters, simply call its `format` method:

```
String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME.for

   // 1969-07-16T09:32:00-04:00"
```

http://www.rejoiceblog.com/

Table 6.1 Predefined Formatters

Formatter	Description	Example
BASIC_ISO_DATE	Year, month, day, zone offset without separators	19690716-0500
ISO_LOCAL_DATE, ISO_LOCAL_TIME, ISO_LOCAL_DATE_TIME	Separators -, :, T	1969-07-16, 09:32:00, 1969-07-16T09:32:00
ISO_OFFSET_DATE, ISO_OFFSET_TIME, ISO_OFFSET_DATE_TIME	Like ISO_LOCAL_*XXX*, but with zone offset	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00
ISO_ZONED_DATE_TIME	With zone offset and zone ID	1969-07-16T09:32:00-05:00[America/New_York]
ISO_INSTANT	In UTC, denoted by the Z zone ID	1969-07-16T14:32:00Z
ISO_DATE, ISO_TIME, ISO_DATE_TIME	Like ISO_OFFSET_DATE, ISO_OFFSET_TIME, and ISO_ZONED_DATE_TIME, but the zone information is optional	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00[America/New_York]
ISO_ORDINAL_DATE	The year and day of year, for LocalDate	1969-197
ISO_WEEK_DATE	The year, week, and day of week, for LocalDate	1969-W29-3
RFC_1123_DATE_TIME	The standard for email timestamps, codified in RFC 822 and updated to four digits for the year in RFC 1123	Wed, 16 Jul 1969 09:32:00 -0500

The standard formatters are mostly intended for machine-readable timestamps. To present dates and times to human readers, use a locale-specific formatter. There are four styles, `SHORT`, `MEDIUM`, `LONG`, and `FULL`, for both date and time—see Table 6.2.

The static methods `ofLocalizedDate`, `ofLocalizedTime`, and `ofLocalizedDateTime` create such a formatter. For example:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDa

String formatted = formatter.format(apollo11launch);
```

http://www.rejoiceblog.com/

```
// July 16, 1969 9:32:00 AM EDT
```

These methods use the default locale. To change to a different locale, simply use the `withLocale` method.

```
formatted = formatter.withLocale(Locale.FRENCH).format(apollo

   // 16 juillet 1969 09:32:00 EDT
```

The `DayOfWeek` and `Month` enumerations have methods `getDisplayName` for giving the names of weekdays and months in different locales and formats.

```
for (DayOfWeek w : DayOfWeek.values())

   System.out.print(w.getDisplayName(TextStyle.SHORT, Locale.

   // Prints Mon Tue Wed Thu Fri Sat Sun
```

See Chapter 5 for more information about locales.

Table 6.2 Locale-Specific Formatting Styles

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT

Note

The `java.time.format.DateTimeFormatter` class is intended as a replacement for `java.util.DateFormat`. If you need an instance of the

http://www.rejoiceblog.com/

latter for backward compatibility, call `formatter.toFormat()`.

Finally, you can roll your own date format by specifying a pattern. For example,

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm")
```

formats a date in the form `Wed 1969-07-16 09:32`. Each letter denotes a different time field, and the number of times the letter is repeated selects a particular format, according to rules that are arcane and seem to have organically grown over time. Table 6.3 shows the most useful pattern elements.

Table 6.3 Commonly Used Formatting Symbols for Date/Time Formats

ChronoField or Purpose	Examples
ERA	G: AD, GGGG: Anno Domini, GGGGG: A
YEAR_OF_ERA	yy: 69, yyyy: 1969
MONTH_OF_YEAR	M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J
DAY_OF_MONTH	d: 6, dd: 06
DAY_OF_WEEK	e: 3, E: Wed, EEEE: Wednesday, EEEEE: W
HOUR_OF_DAY	H: 9, HH: 09
CLOCK_HOUR_OF_AM_PM	K: 9, KK: 09
AMPM_OF_DAY	a: AM
MINUTE_OF_HOUR	mm: 02
SECOND_OF_MINUTE	ss: 00
NANO_OF_SECOND	nnnnnn: 000000
Time zone ID	VV: America/New_York
Time zone name	z: EDT, zzzz: Eastern Daylight Time, v: ET, vvvv: Eastern Time
Zone offset	x: -04, xx: -0400, xxx: -04:00, XXX: same, but use Z for zero
Localized zone offset	O: GMT-4, OOOO: GMT-04:00
Modified Julian Day	g: 58243

To parse a date/time value from a string, use one of the static `parse` methods. For example,

http://www.rejoiceblog.com/

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");

ZonedDateTime apollo11launch =

   ZonedDateTime.parse("1969-07-16 03:32:00-0400",

     DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

The first call uses the standard `ISO_LOCAL_DATE` formatter, the second one a custom formatter.

The program in Listing 6.4 shows how to format and parse dates and times.

Listing 6.4 `formatting/Formatting.java`

```
 1  package formatting;

 2

 3  /**

 4   * @version 1.0 2016-05-10

 5   * @author Cay Horstmann

 6   */

 7

 8  import java.time.*;

 9  import java.time.format.*;

10  import java.util.*;

11
```

http://www.rejoiceblog.com/

```
12  public class Formatting
13  {
14     public static void main(String[] args)
15     {
16        ZonedDateTime apollo11launch = ZonedDateTime.of(196
17             ZoneId.of("America/New_York"));
18
19        String formatted = DateTimeFormatter.ISO_OFFSET_DAT
20        // 1969-07-16T09:32:00-04:00
21        System.out.println(formatted);
22
23        DateTimeFormatter formatter = DateTimeFormatter
24             .ofLocalizedDateTime(FormatStyle.LONG);
25        formatted = formatter.format(apollo11launch);
26        // July 16, 1969 9:32:00 AM EDT
27        System.out.println(formatted);
28        formatted = formatter.withLocale(Locale.FRENCH).for
29        // 16 juillet 1969 09:32:00 EDT
30        System.out.println(formatted);
31
```

http://www.rejoiceblog.com/

```
        formatter = DateTimeFormatter.ofPattern("E yyyy-MM-
        formatted = formatter.format(apollo11launch);
        System.out.println(formatted);

        LocalDate churchsBirthday = LocalDate.parse("1903-0
        System.out.println("churchsBirthday: " + churchsBir
        apollo11launch = ZonedDateTime.parse("1969-07-16 03
             DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:
        System.out.println("apollo11launch: " + apollo11lau

        for (DayOfWeek w : DayOfWeek.values())
         System.out.print(w.getDisplayName(TextStyle.SHORT,
               + " ");
    }
}
```

java.time.format.DateTimeFormatter 8

- `String format(TemporalAccessor temporal)`

formats the given value. The classes `Instant`, `LocalDate`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime`, as well as many others, implement the

http://www.rejoiceblog.com/

`TemporalAccessor` interface.

- `static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)`

- `static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)`

- `static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)`

- `static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle timeStyle)`

yields a formatter for the given styles. The `FormatStyle` enumeration has values `SHORT`, `MEDIUM`, `LONG`, and `FULL`.

- `DateTimeFormatter withLocale(Locale locale)`

yields a formatter equal to this formatter, using the given locale.

- `static DateTimeFormatter ofPattern(String pattern)`

- `static DateTimeFormatter ofPattern(String pattern, Locale locale)`

yields a formatter using the given pattern and locale. See Table 6.3 for the pattern syntax.

java.time.LocalDate 8

- `static LocalDate parse(CharSequence text)`

- `static LocalDate parse(CharSequence text, DateTimeFormatter formatter)`

yields a `LocalDate` using the default formatter, or the given formatter.

http://www.rejoiceblog.com/

java.time.ZonedDateTime 8

- `static ZonedDateTime parse(CharSequence text)`
- `static ZonedDateTime parse(CharSequence text, DateTimeFormatter formatter)`

yields a `ZonedDateTime` using the default formatter, or the given formatter.

6.7 INTEROPERATING WITH LEGACY CODE

As a new creation, the Java Date and Time API will have to interoperate with existing classes—in particular, the ubiquitous `java.util.Date`, `java.util.GregorianCalendar`, and `java.sql.Date/Time/Timestamp`.

The `Instant` class is a close analog to `java.util.Date`. In Java 8, that class has two added methods: the `toInstant` method that converts a `Date` to an `Instant`, and the static `from` method that converts in the other direction.

Similarly, `ZonedDateTime` is a close analog to `java.util.GregorianCalendar`, and that class has gained conversion methods in Java 8. The `toZonedDateTime` method converts a `GregorianCalendar` to a `ZonedDateTime`, and the static `from` method does the opposite conversion.

Another set of conversions is available for the date and time classes in the `java.sql` package. You can also pass a `DateTimeFormatter` to legacy code that uses `java.text.Format`. Table 6.4 summarizes these conversions.

Table 6.4 Conversions between `java.time` Classes and Legacy Classes

http://www.rejoiceblog.com/

Classes	To Legacy Class	From Legacy Class
Instant ↔ java.util.Date	Date.from(instant)	date.toInstant()
ZonedDateTime ↔ java.util.GregorianCalendar	GregorianCalendar. from(zonedDateTime)	cal.toZonedDateTime()
Instant ↔ java.sql.Timestamp	TimeStamp.from(instant)	timestamp.toInstant()
LocalDateTime ↔ java.sql.Timestamp	Timestamp.valueOf(localDateTime)	timeStamp.toLocalDateTime()
LocalDate ↔ java.sql.Date	Date.valueOf(localDate)	date.toLocalDate()
LocalTime ↔ java.sql.Time	Time.valueOf(localTime)	time.toLocalTime()
DateTimeFormatter → java.text.DateFormat	formatter.toFormat()	None
java.util.TimeZone → ZoneId	Timezone.getTimeZone(id)	timeZone.toZoneId()
java.nio.file.attribute.FileTime → Instant	FileTime.from(instant)	fileTime.toInstant()

You now know how to use the Java 8 date and time library to work with date and time values around the world. The next chapter takes the discussion of programming for an international audience further. You will see how to format program messages, numbers, and currencies in the way that makes sense for your customers, wherever they may be.

http://www.rejoiceblog.com/

Chapter 7 Internationalization

In this chapter

- 7.1 Locales
- 7.2 Number Formats
- 7.3 Date and Time
- 7.4 Collation and Normalization
- 7.5 Message Formatting
- 7.6 Text Input and Output
- 7.7 Resource Bundles
- 7.8 A Complete Example

There's a big world out there; we hope that lots of its inhabitants will be interested in your software. The Internet, after all, effortlessly spans the barriers between countries. On the other hand, when you pay no attention to an international audience, *you* are putting up a barrier.

The Java programming language was the first language designed from the ground up to support internationalization. From the beginning, it had the one essential feature needed for effective internationalization: It used Unicode for all strings. Unicode support makes it easy to write Java programs that manipulate strings in any one of multiple languages.

Many programmers believe that all they need to do to internationalize their application is to support Unicode and to translate the messages in the user interface. However, as this chapter demonstrates, there is a lot more to internationalizing programs than just Unicode support. Dates, times, currencies, even numbers are formatted differently in

http://www.rejoiceblog.com/
different parts of the world. You need an easy way to configure menu and button names, message strings, and keyboard shortcuts for different languages.

In this chapter, we will show you how to write internationalized Java programs and how to localize dates, times, numbers, text, and GUIs. We will show you the tools that Java offers for writing internationalized programs. We will close this chapter with a complete example—a retirement calculator with a user interface in English, German, and Chinese.

7.1 LOCALES

When you look at an application that is adapted to an international market, the most obvious difference you notice is the language. This observation is actually a bit too limiting for true internationalization, since countries can share a common language, but you might still need to do some work to make computer users of both countries happy. As Oscar Wilde famously said: “We have really everything in common with America nowadays, except, of course, language.”

7.1.1 Why Locales?

When you provide international versions of a program, all program messages will need to be translated to the local language. However, simply translating the user interface text is not sufficient. There are many more subtle differences; for example, numbers are formatted quite differently in English and in German. The number

123,456.78

should be displayed as

123.456,78

for a German user—that is, the roles of the decimal point and the decimal comma separator are reversed. There are similar variations in the display of dates. In the United States, dates are somewhat irrationally displayed as month/day/year. Germany uses the more sensible order of day/month/year, whereas in China, the usage is year/month/day. Thus, the date

3/22/61

should be presented as

22.03.1961

http://www.rejoiceblog.com/
to a German user. Of course, if the month names are written out explicitly, the difference in languages becomes apparent. The English

March 22, 1961

should be presented as

22. März 1961

in German, or

1961 年 3 月 22 日

in Chinese.

A *locale* captures local preferences such as the ones that you have just seen. Whenever you present numbers, dates, currency values, and other items whose formatting varies by language or location, you need to use locale-aware APIs.

7.1.2 Specifying Locales

A locale is made up of up to five components:

1. A language, specified by two or three lowercase letters, such as `en` (English), `de` (German), or `zh` (Chinese). Table 7.1 shows common codes.

2. Optionally, a script, specified by four letters with an initial uppercase, such as `Latn` (Latin), `Cyrl` (Cyrillic), or `Hant` (traditional Chinese characters). This can be useful because some languages, such as Serbian, are written in Latin or Cyrillic, and some Chinese readers prefer the traditional over the simplified characters.

3. Optionally, a country or region, specified by two uppercase letters or three digits, such as `US` (United States) or `CH` (Switzerland). Table 7.2 shows common codes.

4. Optionally, a variant, specifying miscellaneous features such as dialects or spelling rules. Variants are rarely used nowadays. There used to be a “Nynorsk” variant of Norwegian, but it is now expressed with a different language code, `nn`. What used to be variants for the Japanese imperial calendar and Thai numerals are now expressed as extensions (see the next item).

5. Optionally, an extension. Extensions describe local preferences for calendars (such as the Japanese calendar), numbers (Thai instead of Western digits), and so on. The

http://www.rejoiceblog.com/

Unicode standard specifies some of these extensions. Extensions start with `u-` and a two-letter code specifying whether the extension deals with the calendar (`ca`), numbers (`nu`), and so on. For example, the extension `u-nu-thai` denotes the use of Thai numerals. Other extensions are entirely arbitrary and start with `x-`, such as `x-java`.

Rules for locales are formulated in the "Best Current Practices" memo BCP 47 of the Internet Engineering Task Force (`http://tools.ietf.org/html/bcp47`). You can find a more accessible summary at `www.w3.org/International/articles/language-tags`.

Table 7.1 Common ISO 639-1 Language Codes

Language	Code	Language	Code
Chinese	zh	Italian	it
Danish	da	Japanese	ja
Dutch	nl	Korean	ko
English	en	Norwegian	no
French	fr	Portuguese	pt
Finnish	fi	Spanish	es
German	de	Swedish	sv
Greek	el	Turkish	tr

Table 7.2 Common ISO 3166-1 Country Codes

Country	Code	Country	Code
Austria	AT	Japan	JP
Belgium	BE	Korea	KR
Canada	CA	The Netherlands	NL
China	CN	Norway	NO
Denmark	DK	Portugal	PT
Finland	FI	Spain	ES
Germany	DE	Sweden	SE
Great Britain	GB	Switzerland	CH
Greece	GR	Taiwan	TW
Ireland	IE	Turkey	TR
Italy	IT	United States	US

http://www.rejoiceblog.com/

The codes for languages and countries seem a bit random because some of them are derived from local languages. German in German is Deutsch, Chinese in Chinese is zhongwen: hence `de` and `zh`. And Switzerland is `CH`, deriving from the Latin term *Confoederatio Helvetica* for the Swiss confederation.

Locales are described by tags—hyphenated strings of locale elements such as `en-US`.

In Germany, you would use a locale `de-DE`. Switzerland has four official languages (German, French, Italian, and Rhaeto-Romance). A German speaker in Switzerland would want to use a locale `de-CH`. This locale uses the rules for the German language, but currency values are expressed in Swiss francs, not euros.

If you only specify the language, say, `de`, then the locale cannot be used for country-specific issues such as currencies.

You can construct a `Locale` object from a tag string like this:

```
Locale usEnglish = Locale.forLanguageTag("en-US");
```

The `toLanguageTag` method yields the language tag for a given locale. For example, `Locale.US.toLanguageTag()` is the string `"en-US"`

For your convenience, there are predefined locale objects for various countries:

```
Locale.CANADA

Locale.CANADA_FRENCH

Locale.CHINA

Locale.FRANCE

Locale.GERMANY

Locale.ITALY

Locale.JAPAN
```

http://www.rejoiceblog.com/

```
Locale.KOREA

Locale.PRC

Locale.TAIWAN

Locale.UK

Locale.US
```

A number of predefined locales specify just a language without a location:

```
Locale.CHINESE

Locale.ENGLISH

Locale.FRENCH

Locale.GERMAN

Locale.ITALIAN

Locale.JAPANESE

Locale.KOREAN

Locale.SIMPLIFIED_CHINESE

Locale.TRADITIONAL_CHINESE
```

Finally, the static `getAvailableLocales` method returns an array of all locales known to the virtual machine.

Note

You can get all language codes as `Locale.getISOLanguages()` and all

http://www.rejoiceblog.com/

country codes as `Locale.getISOCountries()`.

7.1.3 The Default Locale

The static `getDefault` method of the `Locale` class initially gets the default locale as stored by the local operating system. You can change the default Java locale by calling the `setDefault` method with a different locale.

Some operating systems allow the user to specify different locales for displayed messages and for formatting. For example, a French speaker living in the United States can have French menus but currency values in dollar.

To obtain these preferences, call

```
Locale displayLocale = Locale.getDefault(Locale.Category.DISP

Locale formatLocale = Locale.getDefault(Locale.Category.FORMA
```

Note

In Unix, you can specify separate locales for numbers, currencies, and dates, by setting the `LC_NUMERIC`, `LC_MONETARY`, and `LC_TIME` environment variables. Java does not pay attention to these settings.

Tip

If you want to test a locale that just has language and country settings, you can supply them on the command line when you launch your program. For example, here we set the default locale to `de-CH`:

```
java -Duser.language=de -Duser.region=CH MyProgram
```

http://www.rejoiceblog.com/

7.1.4 Display Names

Once you have a locale, what can you do with it? Initially, not much, as it turns out. The only useful methods in the `Locale` class are those for identifying the language and country codes. The most important one is `getDisplayName`. It returns a string describing the locale. This string does not contain the cryptic two-letter codes, but is in a form that can be presented to a user, such as

```
German (Switzerland)
```

Actually, there is a problem here. The display name is issued in the default locale. That might not be appropriate. If your user already selected German as the preferred language, you probably want to present the string in German. You can do just that by giving the German locale as a parameter. The code

```
var loc = new Locale("de", "CH");

System.out.println(loc.getDisplayName(Locale.GERMAN));
```

prints

```
Deutsch (Schweiz)
```

This example shows why you need `Locale` objects. You feed them to locale-aware methods that produce text that is presented to users in different locations. You will see many examples of this in the following sections.

Caution

Even such mundane operations as turning a string into lowercase or uppercase can be locale-specific. For example, in the Turkish locale, the lowercase of the letter I is a dotless ı. Programs that tried to normalize strings by storing them in lowercase have mysteriously failed for Turkish customers where I and the dotted i don't have the same lowercase version. It is a good idea to always use the variants of `toUpperCase` and `toLowerCase` that take a `Locale` argument. For example, try out:

http://www.rejoiceblog.com/

```
String cmd = "QUIT".toLowerCase(Locale.forLanguageTag("tr"));

// "quıt" with a dotless ı
```

Of course, in Turkey, where `Locale.getDefault()` yields just that locale, `"QUIT".toLowerCase()` is not the same as `"quit"`.

If you want to normalize English language strings to lowercase, you should pass an English locale to the `toLowerCase` method.

Note

You can explicitly set the locale for input and output operations.

• When reading numbers from a `Scanner`, you can set its locale with the `useLocale` method.

• The `String.format` and `PrintWriter.printf` methods optionally take a `Locale` argument.

java.util.Locale 1.1

• `Locale(String language)`

• `Locale(String language, String country)`

• `Locale(String language, String country, String variant)`

constructs a locale with the given language, country, and variant. Don't use variants in new code—use the IETF BCP 47 language tags instead.

• `static Locale forLanguageTag(String languageTag)` 7

constructs a locale corresponding to the given language tag.

• `static Locale getDefault()`

http://www.rejoiceblog.com/

returns the default locale.

- `static void setDefault(Locale loc)`

sets the default locale.

- `String getDisplayName()`

returns a name describing the locale, expressed in the current locale.

- `String getDisplayName(Locale loc)`

returns a name describing the locale, expressed in the given locale.

- `String getLanguage()`

returns the language code, a lowercase two-letter ISO 639 code.

- `String getDisplayLanguage()`

returns the name of the language, expressed in the current locale.

- `String getDisplayLanguage(Locale loc)`

returns the name of the language, expressed in the given locale.

- `String getCountry()`

returns the country code as an uppercase two-letter ISO 3166 code.

- `static String[] getISOCountries()`

- `static` Set<String> getISOCountries(Locale.IsoCountryCode type) 9

Gets all two-letter country codes, or all country codes with 2, 3, or 4 letters. The type is one of the enumeration constants `PART1_ALPHA2`, `PART1_ALPHA3`, and `PART3`.

- `String getDisplayCountry()`

returns the name of the country, expressed in the current locale.

- `String getDisplayCountry(Locale loc)`

http://www.rejoiceblog.com/

returns the name of the country, expressed in the given locale.

- `String toLanguageTag()` 7

returns the IETF BCP 47 language tag for this locale, e.g., `"de-CH"`.

- `String toString()`

returns a description of the locale, with the language and country separated by underscores (e.g., `"de_CH"`). Use this method only for debugging.

7.2 NUMBER FORMATS

We already mentioned how number and currency formatting is highly locale-dependent. The Java library supplies a collection of formatter objects that can format and parse numeric values in the `java.text` package.

7.2.1 Formatting Numeric Values

Go through the following steps to format a number for a particular locale:

1. Get the locale object, as described in the preceding section.

2. Use a “factory method” to obtain a formatter object.

3. Use the formatter object for formatting and parsing.

The factory methods are static methods of the `NumberFormat` class that take a `Locale` argument. There are three factory methods: `getNumberInstance`, `getCurrencyInstance`, and `getPercentInstance`. These methods return objects that can format and parse numbers, currency amounts, and percentages, respectively. For example, here is how you can format a currency value in German:

```
Locale loc = Locale.GERMAN;

NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);

double amt = 123456.78;

String result = currFmt.format(amt);
```

http://www.rejoiceblog.com/

The result is

123.456,78 €

Note that the currency symbol is € and that it is placed at the end of the string. Also, note the reversal of decimal points and decimal commas.

Conversely, to read in a number that was entered or stored with the conventions of a certain locale, use the `parse` method. For example, the following code parses the value that the user typed into a text field. The `parse` method can deal with decimal points and commas, as well as digits in other languages.

```
TextField inputField;

. . .

NumberFormat fmt = NumberFormat.getNumberInstance();

// get the number formatter for default locale

Number input = fmt.parse(inputField.getText().trim());

double x = input.doubleValue();
```

The return type of `parse` is the abstract type `Number`. The returned object is either a `Double` or a `Long` wrapper object, depending on whether the parsed number was a floating-point number. If you don't care about the distinction, you can simply use the `doubleValue` method of the `Number` class to retrieve the wrapped number.

Caution

Objects of type `Number` are not automatically unboxed—you cannot simply assign a `Number` object to a primitive type. Instead, use the `doubleValue` or `intValue` method.

http://www.rejoiceblog.com/

If the text for the number is not in the correct form, the method throws a `ParseException`. For example, leading whitespace in the string is *not* allowed. (Call `trim` to remove it.) However, any characters that follow the number in the string are simply ignored, so no exception is thrown.

Note that the classes returned by the `getXxxInstance` factory methods are not actually of type `NumberFormat`. The `NumberFormat` type is an abstract class, and the actual formatters belong to one of its subclasses. The factory methods merely know how to locate the object that belongs to a particular locale.

You can get a list of the currently supported locales with the static `getAvailableLocales` method. That method returns an array of the locales for which number formatter objects can be obtained.

The sample program for this section lets you experiment with number formatters (see Figure 7.1). The combo box at the top of the figure contains all locales with number formatters. You can choose between number, currency, and percentage formatters. Each time you make another choice, the number in the text field is reformatted. If you go through a few locales, you can get a good impression of the many ways that a number or currency value can be formatted. You can also type a different number and click the Parse button to call the `parse` method, which tries to parse what you entered. If your input is successfully parsed, it is passed to `format` and the result is displayed. If parsing fails, then a "Parse error" message is displayed in the text field.

Figure 7.1 The `NumberFormatTest` program

The code, shown in Listing 7.1, is fairly straightforward. In the constructor, we call `NumberFormat.getAvailableLocales`. For each locale, we call `getDisplayName` and fill a combo box with the strings that the `getDisplayName` method returns. (The strings are not sorted; we tackle this issue in Section 7.4, "Collation and Normalization," on p. 401.) Whenever the user selects another locale or clicks one of the radio buttons, we create a new formatter object and update the text field. When the user clicks the Parse button, we call the `parse` method to do the actual parsing, based

http://www.rejoiceblog.com/

on the locale selected.

Note

You can use a `Scanner` to read localized integers and floating-point numbers. Call the `useLocale` method to set the locale.

Listing 7.1 `numberFormat/NumberFormatTest.java`

```
package numberFormat;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

import javax.swing.*;

/**
 * This program demonstrates formatting numbers under var
 * @version 1.15 2018-05-01
 * @author Cay Horstmann
```

http://www.rejoiceblog.com/

```
 */
public class NumberFormatTest
{
  public static void main(String[] args)
  {
      EventQueue.invokeLater(() ->
        {
          var frame = new NumberFormatFrame();
          frame.setTitle("NumberFormatTest");
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_C
          frame.setVisible(true);
        });
  }
}

/**
 * This frame contains radio buttons to select a number f
 * field to display a formatted number, and a button to p
 */
class NumberFormatFrame extends JFrame
```

http://www.rejoiceblog.com/

```
34  {
35  private Locale[] locales;
36  private double currentNumber;
37  private JComboBox<String> localeCombo = new JComboBox<>()
38  private JButton parseButton = new JButton("Parse");
39  private JTextField numberText = new JTextField(30);
40  private JRadioButton numberRadioButton = new JRadioButton
41  private JRadioButton currencyRadioButton = new JRadioButt
42  private JRadioButton percentRadioButton = new JRadioButto
43  private ButtonGroup rbGroup = new ButtonGroup();
44  private NumberFormat currentNumberFormat;
45
46  public NumberFormatFrame()
47  {
48    setLayout(new GridBagLayout());
49
50    ActionListener listener = event -> updateDisplay();
51
52    var p = new JPanel();
```

http://www.rejoiceblog.com/

```
addRadioButton(p, numberRadioButton, rbGroup, listener)
addRadioButton(p, currencyRadioButton, rbGroup, listene
addRadioButton(p, percentRadioButton, rbGroup, listener

add(new JLabel("Locale:"), new GBC(0, 0).setAnchor(GBC.
add(p, new GBC(1, 1));
add(parseButton, new GBC(0, 2).setInsets(2));
add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
  locales = (Locale[]) NumberFormat.getAvailableLocales
Arrays.sort(locales, Comparator.comparing(Locale::getDi
for (Locale loc : locales)
localeCombo.addItem(loc.getDisplayName());
localeCombo.setSelectedItem(Locale.getDefault().getDisp
currentNumber = 123456.78;
updateDisplay();

localeCombo.addActionListener(listener);

parseButton.addActionListener(event ->
```

http://www.rejoiceblog.com/

```
73      {
74        String s = numberText.getText().trim();
75        try
76        {
77          Number n = currentNumberFormat.parse(s);
78          currentNumber = n.doubleValue();
79          updateDisplay();
80        }
81        catch (ParseException e)
82        {
83            numberText.setText(e.getMessage());
84        }
85      });
86    pack();
87  }
88
89  /**
90   * Adds a radio button to a container.
91   * @param p the container into which to place the button
```

http://www.rejoiceblog.com/

```
 * @param b the button
 * @param g the button group
 * @param listener the button listener
 */
public void addRadioButton(Container p, JRadioButton b, B
{
   b.setSelected(g.getButtonCount() == 0);
   b.addActionListener(listener);
   g.add(b);
   p.add(b);
}

/**
 * Updates the display and formats the number according t
 */
public void updateDisplay()
{
   Locale currentLocale = locales[localeCombo.getSelected
   currentNumberFormat = null;
   if (numberRadioButton.isSelected())
```

http://www.rejoiceblog.com/

```
      currentNumberFormat = NumberFormat.getNumberInstance
    else if (currencyRadioButton.isSelected())
      currentNumberFormat = NumberFormat.getCurrencyInstan
    else if (percentRadioButton.isSelected())
      currentNumberFormat = NumberFormat.getPercentInstanc
    String formatted = currentNumberFormat.format(currentN
    numberText.setText(formatted);
  }
}
```

java.text.NumberFormat 1.1

- `static Locale[] getAvailableLocales()`

returns an array of `Locale` objects for which `NumberFormat` formatters are available.

- `static NumberFormat getNumberInstance()`
- `static NumberFormat getNumberInstance(Locale l)`
- `static NumberFormat getCurrencyInstance()`
- `static NumberFormat getCurrencyInstance(Locale l)`
- `static NumberFormat getPercentInstance()`
- `static NumberFormat getPercentInstance(Locale l)`

returns a formatter for numbers, currency amounts, or percentage values for the

http://www.rejoiceblog.com/

current locale or for the given locale.

- `String format(double x)`

- `String format(long x)`

returns the string resulting from formatting the given floating-point number or integer.

- `Number parse(String s)`

parses the given string and returns the number value, as a `Double` if the input string describes a floating-point number and as a `Long` otherwise. The beginning of the string must contain a number; no leading whitespace is allowed. The number can be followed by other characters, which are ignored. Throws `ParseException` if parsing was not successful.

- `void setParseIntegerOnly(boolean b)`

- `boolean isParseIntegerOnly()`

sets or gets a flag to indicate whether this formatter should parse only integer values.

- `void setGroupingUsed(boolean b)`

- `boolean isGroupingUsed()`

sets or gets a flag to indicate whether this formatter emits and recognizes decimal separators (such as `100,000`).

- `void setMinimumIntegerDigits(int n)`

- `int getMinimumIntegerDigits()`

- `void setMaximumIntegerDigits(int n)`

- `int getMaximumIntegerDigits()`

- `void setMinimumFractionDigits(int n)`

- `int getMinimumFractionDigits()`

- `void setMaximumFractionDigits(int n)`

http://www.rejoiceblog.com/

- `int getMaximumFractionDigits()`

sets or gets the maximum or minimum number of digits allowed in the integer or fractional part of a number.

7.2.2 Currencies

To format a currency value, you can use the `NumberFormat.getCurrencyInstance` method. However, that method is not very flexible—it returns a formatter for a single currency. Suppose you prepare an invoice for an American customer in which some amounts are in dollars and others are in euros. You can't just use two formatters

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInstan

NumberFormat euroFormatter = NumberFormat.getCurrencyInstance
```

Your invoice would look very strange, with some values formatted like $100,000 and others like 100.000 €. (Note that the euro value uses a decimal point, not a comma.)

Instead, use the `Currency` class to control the currency used by the formatters. You can get a `Currency` object by passing a currency identifier to the static `Currency.getInstance` method. Then call the `setCurrency` method for each formatter. Here is how you would set up the euro formatter for your American customer:

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance

euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

The currency identifiers are defined by ISO 4217 (see `www.currency-iso.org/iso_index/iso_tables/iso_tables_a1.htm`). Table 7.3 provides a partial list.

Table 7.3 Currency Identifiers

http://www.rejoiceblog.com/

Currency Value	Identifier	Numeric Code
U.S. Dollar	USD	840
Euro	EUR	978
British Pound	GBP	826
Japanese Yen	JPY	392
Chinese Renminbi (Yuan)	CNY	156
Indian Rupee	INR	356
Russian Ruble	RUB	643

java.util.Currency 1.4

- `static Currency getInstance(String currencyCode)`
- `static Currency getInstance(Locale locale)`

returns the `Currency` instance for the given ISO 4217 currency code or the country of the given locale.

- `String toString()`
- `String getCurrencyCode()`
- `String getNumericCode()` 7
- `String getNumericCodeAsString()` 9

gets the ISO 4217 alphabetic or numeric currency code of this currency.

- `String getSymbol()`
- `String getSymbol(Locale locale)`

gets the formatting symbol of this currency for the default locale or the given locale. For example, the symbol for USD can be `"$"` or `"US$"`, depending on the locale.

- `int getDefaultFractionDigits()`

gets the default number of fraction digits of this currency.

http://www.rejoiceblog.com/

- `static Set<Currency> getAvailableCurrencies()` 7

gets all available currencies.

7.3 DATE AND TIME

When you are formatting date and time, you should be concerned with four locale-dependent issues:

- The names of months and weekdays should be presented in the local language.
- There will be local preferences for the order of year, month, and day.
- The Gregorian calendar might not be the local preference for expressing dates.
- The time zone of the location must be taken into account.

The `DateTimeFormatter` class from the `java.time` package handles these issues. Pick one of the formatting styles shown in Tables 7.4. Then get a formatter:

```
FormatStyle style = . . .; // One of FormatStyle.SHORT, Forma

DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocaliz

DateTimeFormatter timeFormatter = DateTimeFormatter.ofLocaliz

DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofLoc

   // or DateTimeFormatter.ofLocalizedDateTime(style1, style2
```

These formatters use the current locale. To use a different locale, use the `withLocale` method:

```
DateTimeFormatter dateFormatter =

   DateTimeFormatter.ofLocalizedDate(style).withLocale(locale
```

http://www.rejoiceblog.com/

Now you can format a `LocalDate`, `LocalDateTime`, `LocalTime`, or `ZonedDateTime`:

```
ZonedDateTime appointment = . . .;

String formatted = formatter.format(appointment);
```

Table 7.4 Date and Time Formatting Styles

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT in en-US, 9:32:00 MSZ in de-DE (only for ZonedDateTime)
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT in en-US, 9:32 Uhr MSZ in de-DE (only for ZonedDateTime)

Note

Here we use the `DateTimeFormatter` class from the `java.time` package. There is also a legacy `java.text.DateFormatter` class from Java 1.1 that works with `Date` and `Calendar` objects.

You can use one of the static `parse` methods of `LocalDate`, `LocalDateTime`, `LocalTime`, or `ZonedDateTime` to parse a date or time in a string:

```
LocalTime time = LocalTime.parse("9:32 AM", formatter);
```

These methods are not suitable for parsing human input, at least not without preprocessing. For example, the short time formatter for the United States will parse `"9:32 AM"` but not `"9:32AM"` or `"9:32 am"`.

http://www.rejoiceblog.com/

Caution

Date formatters parse nonexistent dates, such as November 31, and adjust them to the last date in the given month.

Sometimes, you need to display just the names of weekdays and months, for example, in a calendar application. Call the `getDisplayName` method of the `DayOfWeek` and `Month` enumerations.

```
for (Month m : Month.values())

   System.out.println(m.getDisplayName(textStyle, locale) + "
```

Tables 7.5 shows the text styles. The `STANDALONE` versions are for display outside a formatted date. For example, in Finnish, January is “tammikuuta” inside a date, but “tammikuu” standalone.

Table 7.5 Values of the `java.time.format.TextStyle` Enumeration

Style	Example
FULL / FULL_STANDALONE	January
SHORT / SHORT_STANDALONE	Jan
NARROW / NARROW_STANDALONE	J

Note

The first day of the week can be Saturday, Sunday, or Monday, depending on the locale. You can obtain it like this:

```
DayOfWeek first = WeekFields.of(locale).getFirstDayOfWeek();
```

http://www.rejoiceblog.com/

Listing 7.2 shows the `DateFormat` class in action. You can select a locale and see how the date and time are formatted in different places around the world.

Figure 7.2 shows the program (after Chinese fonts were installed). As you can see, it correctly displays the output.

Figure 7.2 The `DateFormatTest` program

You can also experiment with parsing. Enter a date, time, or date/time and click the Parse button.

We use a helper class `EnumCombo` to solve a technical problem (see Listing 7.3). We wanted to fill a combo with values such as `Short`, `Medium`, and `Long` and then automatically convert the user's selection to values `FormatStyle.SHORT`, `FormatStyle.MEDIUM`, and `FormatStyle.LONG`. Instead of writing repetitive code, we use reflection: We convert the user's choice to upper case, replace all spaces with underscores, and then find the value of the static field with that name. (See Volume I, Chapter 5 for more details about reflection.)

Listing 7.2 `dateFormat/DateTimeFormatterTest.java`

```
package dateFormat;

import java.awt.*;
import java.awt.event.*;
import java.time.*;
import java.time.format.*;
import java.util.*;
```

http://www.rejoiceblog.com/

```

import javax.swing.*;

/**
 * This program demonstrates formatting dates under vario
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */
public class DateTimeFormatterTest
{
   public static void main(String[] args)
   {
     EventQueue.invokeLater(() ->
       {
         var frame = new DateTimeFormatterFrame();
         frame.setTitle("DateFormatTest");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
         frame.setVisible(true);
       });
```

http://www.rejoiceblog.com/

```
    }
}

/**
 * This frame contains combo boxes to pick a locale, date
 * formatted date and time, buttons to parse the text fie
 */
class DateTimeFormatterFrame extends JFrame
{
 private Locale[] locales;
 private LocalDate currentDate;
 private LocalTime currentTime;
 private ZonedDateTime currentDateTime;
 private DateTimeFormatter currentDateFormat;
 private DateTimeFormatter currentTimeFormat;
 private DateTimeFormatter currentDateTimeFormat;
 private JComboBox<String> localeCombo = new JComboBox<>(
 private JButton dateParseButton = new JButton("Parse");
 private JButton timeParseButton = new JButton("Parse");
 private JButton dateTimeParseButton = new JButton("Parse
```

http://www.rejoiceblog.com/

```
47   private JTextField dateText = new JTextField(30);
48   private JTextField timeText = new JTextField(30);
49   private JTextField dateTimeText = new JTextField(30);
50   private EnumCombo<FormatStyle> dateStyleCombo = new Enum
51       "Short", "Medium", "Long", "Full");
52   private EnumCombo<FormatStyle> timeStyleCombo = new Enum
53       "Short", "Medium");
54   private EnumCombo<FormatStyle> dateTimeStyleCombo = new
55       "Short", "Medium", "Long", "Full");
56
57   public DateTimeFormatterFrame()
58   {
59     setLayout(new GridBagLayout());
60     add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.
61     add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WES
62
63     add(new JLabel("Date"), new GBC(0, 1).setAnchor(GBC.EA
64     add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST))
65     add(dateText, new GBC(2, 1, 2, 1).setFill(GBC.HORIZONT
```

http://www.rejoiceblog.com/

```
    add(dateParseButton, new GBC(4, 1).setAnchor(GBC.WEST)

    add(new JLabel("Time"), new GBC(0, 2).setAnchor(GBC.EA
    add(timeStyleCombo, new GBC(1, 2).setAnchor(GBC.WEST))
    add(timeText, new GBC(2, 2, 2, 1).setFill(GBC.HORIZONT
    add(timeParseButton, new GBC(4, 2).setAnchor(GBC.WEST)

    add(new JLabel("Date and time"), new GBC(0, 3).setAnch
    add(dateTimeStyleCombo, new GBC(1, 3).setAnchor(GBC.WE
    add(dateTimeText, new GBC(2, 3, 2, 1).setFill(GBC.HORI
    add(dateTimeParseButton, new GBC(4, 3).setAnchor(GBC.W

    locales = (Locale[]) Locale.getAvailableLocales().clon
    Arrays.sort(locales, Comparator.comparing(Locale::getD
    for (Locale loc : locales)
      localeCombo.addItem(loc.getDisplayName());
    localeCombo.setSelectedItem(Locale.getDefault().getDis
    currentDate = LocalDate.now();
    currentTime = LocalTime.now();
    currentDateTime = ZonedDateTime.now();
```

http://www.rejoiceblog.com/

```
    updateDisplay();

    ActionListener listener = event -> updateDisplay();
    localeCombo.addActionListener(listener);
    dateStyleCombo.addActionListener(listener);
    timeStyleCombo.addActionListener(listener);
    dateTimeStyleCombo.addActionListener(listener);

    addAction(dateParseButton, () -> {
        currentDate = LocalDate.parse(dateText.getText().t
      });
    addAction(timeParseButton, () -> {
        currentTime = LocalTime.parse(timeText.getText().t
      });
    addAction(dateTimeParseButton, () -> {
        currentDateTime = ZonedDateTime.parse(
          dateTimeText.getText().trim(), currentDateTimeFo
      });

```

http://www.rejoiceblog.com/

```
105    pack();
106  }
107
108  /**
109   * Adds the given action to the button and updates the d
110   * @param button the button to which to add the action
111   * @param action the action to carry out when the button
112   */
113  public void addAction(JButton button, Runnable action)
114  {
115     button.addActionListener(event ->
116       {
117         try
118         {
119           action.run();
120           updateDisplay();
121         }
122         catch (Exception e)
123         {
124           JOptionPane.showMessageDialog(null, e.getMessag
```

http://www.rejoiceblog.com/

```
        }
      });
}

/**
 * Updates the display and formats the date according to
 */
 public void updateDisplay()
{
    Locale currentLocale = locales[localeCombo.getSelect
    FormatStyle dateStyle = dateStyleCombo.getValue();
    currentDateFormat = DateTimeFormatter.ofLocalizedDat
      dateStyle).withLocale(currentLocale);
    dateText.setText(currentDateFormat.format(currentDat
    FormatStyle timeStyle = timeStyleCombo.getValue();
    currentTimeFormat = DateTimeFormatter.ofLocalizedTim
      timeStyle).withLocale(currentLocale);
    timeText.setText(currentTimeFormat.format(currentTim
    FormatStyle dateTimeStyle = dateTimeStyleCombo.getVa
```

http://www.rejoiceblog.com/

```
      currentDateTimeFormat = DateTimeFormatter.ofLocalize
      dateTimeStyle).withLocale(currentLocale);
      dateTimeText.setText(currentDateTimeFormat.format(cu
   }
}
```

Listing 7.3 `dateFormat/EnumCombo.java`

```
package dateFormat;

import java.util.*;
import javax.swing.*;

/**
 * A combo box that lets users choose from among static f
 * values whose names are given in the constructor.
 * @version 1.15 2016-05-06
 * @author Cay Horstmann
 */
public class EnumCombo<T> extends JComboBox<String>
{
```

http://www.rejoiceblog.com/

```
14     private Map<String, T> table = new TreeMap<>();
15
16     /**
17      * Constructs an EnumCombo yielding values of type T.
18      * @param cl a class
19      * @param labels an array of strings describing static
20      * of cl that have type T
21      */
22    public EnumCombo(Class<?> cl, String... labels)
23    {
24       for (String label : labels)
25       {
26         String name = label.toUpperCase().replace(' ', '_'
27         try
28         {
29             java.lang.reflect.Field f = cl.getField(name);
30             @SuppressWarnings("unchecked") T value = (T) f
31             table.put(label, value);
32         }
```

http://www.rejoiceblog.com/

```
        catch (Exception e)
        {
          label = "(" + label + ")";
          table.put(label, null);
        }
        addItem(label);
      }
      setSelectedItem(labels[0]);
  }

  /**
   * Returns the value of the field that the user selected
   * @return the static field value
   */
  public T getValue()
  {
    return table.get(getSelectedItem());
  }
}
```

http://www.rejoiceblog.com/

java.time.format.DateTimeFormatter 8

- `static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)`
- `static DateTimeFormatter ofLocalizedTime(FormatStyle dateStyle)`
- `static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)`
- `static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle, FormatStyle timeStyle)`

returns `DateTimeFormatter` instances that format dates, times, or dates and times with the specified styles.

- `DateTimeFormatter withLocale(Locale locale)`

returns a copy of this formatter with the given locale.

- `String format(TemporalAccessor temporal)`

returns the string resulting from formatting the given date/time.

java.time.LocalDate 8
java.time.LocalTime 8
java.time.LocalDateTime 8
java.time.ZonedDateTime 8

- `static` *Xxx* `parse(CharSequence text, DateTimeFormatter formatter)`

parses the given string and returns the `LocalDate`, `LocalTime`, `LocalDateTime`, or `ZonedDateTime` described in it. Throws a `DateTimeParseException` if parsing was not successful.

7.4 COLLATION AND NORMALIZATION

http://www.rejoiceblog.com/

Most programmers know how to compare strings with the `compareTo` method of the `String` class. Unfortunately, when interacting with human users, this method is not very useful. The `compareTo` method uses the values of the UTF-16 encoding of the string, which leads to absurd results, even in English. For example, the following five strings are ordered according to the `compareTo` method:

```
America

Zulu

able

zebra

Ångström
```

For dictionary ordering, you would want to consider upper case and lower case to be equivalent. To an English speaker, the sample list of words would be ordered as

```
able

America

Ångström

zebra

Zulu
```

However, that order would not be acceptable to a Swedish user. In Swedish, the letter Å is different from the letter A, and it is collated *after* the letter Z! That is, a Swedish user would want the words to be sorted as

```
able

America
```

http://www.rejoiceblog.com/

```
zebra

Zulu

Ångström
```

To obtain a locale-sensitive comparator, call the static `Collator.getInstance` method:

```
Collator coll = Collator.getInstance(locale);

words.sort(coll); // Collator implements Comparator<Object>
```

Since the `Collator` class implements the `Comparator` interface, you can pass a `Collator` object to the `List.sort(Comparator)` method to sort a list of strings.

There are a couple of advanced settings for collators. You can set a collator's *strength* to select how selective it should be. Character differences are classified as *primary*, *secondary*, or *tertiary*. For example, in English, the difference between "A" and "Z" is considered primary, the difference between "A" and "Å" is secondary, and between "A" and "a" is tertiary.

By setting the strength of the collator to `Collator.PRIMARY`, you tell it to pay attention only to primary differences. By setting the strength to `Collator.SECONDARY`, you instruct the collator to take secondary differences into account. That is, two strings will be more likely to be considered different when the strength is set to "secondary" or "tertiary," as shown in Table 7.6.

Table 7.6 Collations with Different Strengths (English Locale)

Primary	Secondary	Tertiary
Angstrom = Ångström	Angstrom ≠ Ångström	Angstrom ≠ Ångström
Able = able	Able = able	Able ≠ able

When the strength has been set to `Collator.IDENTICAL`, no differences are allowed. This setting is mainly useful in conjunction with a rather technical collator setting, the

http://www.rejoiceblog.com/
decomposition mode, which we take up next.

Occasionally, a character or sequence of characters can be described in more than one way in Unicode. For example, an "Å" can be Unicode character U+00C5, or it can be expressed as a plain A (U+0065) followed by a ° ("combining ring above"; U+030A). Perhaps more surprisingly, the letter sequence "ffi" can be described with a single character "Latin small ligature ffi" with code U+FB03. (One could argue that this is a presentation issue that should not have resulted in different Unicode characters, but we don't make the rules.)

The Unicode standard defines four *normalization forms* (D, KD, C, and KC) for strings. See `www.unicode.org/unicode/reports/tr15/tr15-23.html` for the details. In the normalization form C, accented characters are always composed. For example, a sequence of A and a combining ring above ° is combined into a single character Å. In form D, accented characters are always decomposed into their base letters and combining accents: Å is turned into A followed by °. Forms KC and KD also decompose characters such as ligatures or the trademark symbol.

You can choose the degree of normalization that you want a collator to use. The value `Collator.NO_DECOMPOSITION` does not normalize strings at all. This option is faster, but it might not be appropriate for text that expresses characters in multiple forms. The default, `Collator.CANONICAL_DECOMPOSITION`, uses the normalization form D. This is useful for text that contains accents but not ligatures. Finally, "full decomposition" uses normalization form KD. See Table 7.7 for examples.

Table 7.7 Differences between Decomposition Modes

No Decomposition	Canonical Decomposition	Full Decomposition
Å ≠ A°	Å = A°	Å = A°
™ ≠ TM	™ ≠ TM	™ = TM

It is wasteful to have the collator decompose a string many times. If one string is compared many times against other strings, you can save the decomposition in a *collation* key object. The `getCollationKey` method returns a `CollationKey` object that you can use for further, faster comparisons. Here is an example:

```
String a = . . .;

CollationKey aKey = coll.getCollationKey(a);
```

http://www.rejoiceblog.com/

```
if(aKey.compareTo(coll.getCollationKey(b)) == 0) // fast comp

. . .
```

Finally, you might want to convert strings into their normalized forms even when you don't do collation—for example, when storing strings in a database or communicating with another program. The `java.text.Normalizer` class carries out the normalization process. For example,

```
String name = "Ångström";

String normalized = Normalizer.normalize(name, Normalizer.For
```

The normalized string contains ten characters. The "Å" and "ö" are replaced by "Å" and "Å" sequences.

However, that is not usually the best form for storage and transmission. Normalization form C first applies decomposition and then combines the accents back in a standardized order. According to the W3C, this is the recommended mode for transferring data over the Internet.

The program in Listing 7.4 lets you experiment with collation order. Type a word into the text field and click the Add button to add it to the list of words. Each time you add another word, or change the locale, strength, or decomposition mode, the list of words is sorted again. An = sign indicates words that are considered identical (see Figure 7.3).

http://www.rejoiceblog.com/

Figure 7.3 The `CollationTest` program

The locale names in the combo box are displayed in sorted order, using the collator of the default locale. If you run this program with the US English locale, note that "Norwegian (Norway,Nynorsk)" comes before "Norwegian (Norway)", even though the Unicode value of the comma character is greater than the Unicode value of the closing parenthesis.

Listing 7.4 `collation/CollationTest.java`

```
1  package collation;

2

3  import java.awt.*;

4  import java.awt.event.*;

5  import java.text.*;
```

http://www.rejoiceblog.com/

```
import java.util.*;
import java.util.List;

import javax.swing.*;

/**
 * This program demonstrates collating strings under vari
 * @version 1.16 2018-05-01
 * @author Cay Horstmann
 */
public class CollationTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() ->
         {
            var frame = new CollationFrame();
            frame.setTitle("CollationTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
            frame.setVisible(true);
```

http://www.rejoiceblog.com/

```
        });
    }
}

/**
 * This frame contains combo boxes to pick a locale, coll
 * text field and button to add new strings, and a text a
 */
class CollationFrame extends JFrame
{
    private Collator collator = Collator.getInstance(Local
    private List<String> strings = new ArrayList<>();
    private Collator currentCollator;
    private Locale[] locales;
    private JComboBox<String> localeCombo = new JComboBox<
    private JTextField newWord = new JTextField(20);
    private JTextArea sortedWords = new JTextArea(20, 20);
    private JButton addButton = new JButton("Add");
    private EnumCombo<Integer> strengthCombo = new EnumCom
```

http://www.rejoiceblog.com/

```
      "Secondary", "Tertiary", "Identical");
    private EnumCombo<Integer> decompositionCombo = new En
      "Canonical Decomposition", "Full Decomposition", "No

    public CollationFrame()
    {
      setLayout(new GridBagLayout());
      add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GB
      add(new JLabel("Strength"), new GBC(0, 1).setAnchor(
      add(new JLabel("Decomposition"), new GBC(0, 2).setAn
      add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
      add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
      add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST)
      add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.
      add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
      add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1

      locales = (Locale[]) Collator.getAvailableLocales().
      Arrays.sort(locales, (l1, l2) -> collator.compare(l1
      for (Locale loc : locales)
```

http://www.rejoiceblog.com/

```
65          localeCombo.addItem(loc.getDisplayName());
66        localeCombo.setSelectedItem(Locale.getDefault().getD
67
68        strings.add("America");
69        strings.add("able");
70        strings.add("Zulu");
71        strings.add("zebra");
72        strings.add("\u00C5ngstr\u00F6m");
73        strings.add("A\u030angstro\u0308m");
74        strings.add("Angstrom");
75        strings.add("Able");
76        strings.add("office");
77        strings.add("o\uFB03ce");
78        strings.add("Java\u2122");
79        strings.add("JavaTM");
80        updateDisplay();
81
82        addButton.addActionListener(event ->
83          {
```

http://www.rejoiceblog.com/

```
            strings.add(newWord.getText());

          updateDisplay();

          });


        ActionListener listener = event -> updateDisplay();


        localeCombo.addActionListener(listener);

        strengthCombo.addActionListener(listener);

        decompositionCombo.addActionListener(listener);

        pack();

      }


     /**

      * Updates the display and collates the strings accordi

      */

     public void updateDisplay()

     {

     Locale currentLocale = locales[localeCombo.getSelectedI

     localeCombo.setLocale(currentLocale);

```

http://www.rejoiceblog.com/

```
  currentCollator = Collator.getInstance(currentLocale);
  currentCollator.setStrength(strengthCombo.getValue());
  currentCollator.setDecomposition(decompositionCombo.get

  strings.sort(currentCollator);

  sortedWords.setText("");
    for (int i = 0; i < strings.size(); i++)
    {
      String s = strings.get(i);
      if (i > 0 && currentCollator.compare(s, strings.get
        .append("= ");
      sortedWords.append(s + "\n");
    }
    pack();
  }
}
```

java.text.Collator 1.1

http://www.rejoiceblog.com/

- `static Locale[] getAvailableLocales()`

returns an array of `Locale` objects for which `Collator` objects are available.

- `static Collator getInstance()`

- `static Collator getInstance(Locale l)`

returns a collator for the default locale or the given locale.

- `int compare(String a, String b)`

returns a negative value if `a` comes before `b`, `0` if they are considered identical, and a positive value otherwise.

- `boolean equals(String a, String b)`

returns `true` if `a` and `b` are considered identical, `false` otherwise.

- `void setStrength(int strength)`

- `int getStrength()`

sets or gets the strength of the collator. Stronger collators tell more words apart. Strength values are `Collator.PRIMARY`, `Collator.SECONDARY`, and `Collator.TERTIARY`.

- `void setDecomposition(int decomp)`

- `int getDecompositon()`

sets or gets the decomposition mode of the collator. The more a collator decomposes a string, the more strict it will be in deciding whether two strings should be considered identical. Decomposition values are `Collator.NO_DECOMPOSITION`, `Collator.CANONICAL_DECOMPOSITION`, and `Collator.FULL_DECOMPOSITION`.

- `CollationKey getCollationKey(String a)`

returns a collation key that contains a decomposition of the characters in a form that can be quickly compared against another collation key.

http://www.rejoiceblog.com/

java.text.CollationKey 1.1

- `int compareTo(CollationKey b)`

returns a negative value if this key comes before `b`, `0` if they are considered identical, and a positive value otherwise.

java.text.Normalizer 6

- `static String normalize(CharSequence str, Normalizer.Form form)`

returns the normalized form of `str`. The `form` value is one of `ND`, `NKD`, `NC`, or `NKC`.

7.5 MESSAGE FORMATTING

The Java library has a `MessageFormat` class that formats text with variable parts. It is similar to formatting with the `printf` method, but it supports locales and formats for numbers and dates. We will examine that mechanism in the following sections.

7.5.1 Formatting Numbers and Dates

Here is a typical message format string:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage.
```

The numbers in braces are placeholders for actual names and values. The static method `MessageFormat.format` lets you substitute values for the variables. It is a "varargs" method, so you can simply supply the parameters as follows:

```
String msg = MessageFormat.format("On {2}, a {0} destroyed {1
   "hurricane", 99, new GregorianCalendar(1999, 0, 1).getTime
```

http://www.rejoiceblog.com/

In this example, the placeholder `{0}` is replaced with `"hurricane"`, `{1}` is replaced with `99`, and so on.

The result of our example is the string

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and cause
```

That is a start, but it is not perfect. We don't want to display the time "12:00 AM," and we want the damage amount printed as a currency value. The way we do this is by supplying an optional format for some of the placeholders:

```
"On {2,date,long}, a {0} destroyed {1} houses and caused {3,n
```

This example code prints:

```
On January 1, 1999, a hurricane destroyed 99 houses and cause
```

In general, the placeholder index can be followed by a *type* and a *style*. Separate the index, type, and style by commas. The type can be any of

```
number

time

date

choice
```

If the type is `number`, then the style can be

```
integer

currency
```

http://www.rejoiceblog.com/

```
percent
```

or it can be a number format pattern such as `$,##0`. (See the documentation of the `DecimalFormat` class for more information about the possible formats.)

If the type is either `time` or `date`, then the style can be

```
short

medium

long

full
```

or a date format pattern such as `yyyy-MM-dd`. (See the documentation of the `SimpleDateFormat` class for more information about the possible formats.)

Caution

The static `MessageFormat.format` method uses the current locale to format the values. To format with an arbitrary locale, you have to work a bit harder because there is no "varargs" method that you can use. You need to place the values to be formatted into an `Object[]` array, like this:

```
var mf = new MessageFormat(pattern, loc);

String msg = mf.format(new Object[] { values });
```

java.text.MessageFormat 1.1

- `MessageFormat(String pattern)`

http://www.rejoiceblog.com/

- `MessageFormat(String pattern, Locale loc)`

constructs a message format object with the specified pattern and locale.

- `void applyPattern(String pattern)`

sets the pattern of a message format object to the specified pattern.

- `void setLocale(Locale loc)`

- `Locale getLocale()`

sets or gets the locale to be used for the placeholders in the message. The locale is *only* used for subsequent patterns that you set by calling the `applyPattern` method.

- `static String format(String pattern, Object... args)`

formats the pattern string by using `args[i]` as input for placeholder `{i}`.

- `StringBuffer format(Object args, StringBuffer result, FieldPosition pos)`

formats the pattern of this `MessageFormat`. The `args` parameter must be an array of objects. The formatted string is appended to `result`, and `result` is returned. If `pos` equals `new FieldPosition(MessageFormat.Field.ARGUMENT)`, its `beginIndex` and `endIndex` properties are set to the location of the text that replaces the `{1}` placeholder. Supply `null` if you are not interested in position information.

java.text.Format 1.1

- `String format(Object obj)`

formats the given object, according to the rules of this formatter. This method calls `format(obj, new StringBuffer(), new FieldPosition(1)).toString()`.

7.5.2 Choice Formats

Let's look closer at the pattern of the preceding section:

http://www.rejoiceblog.com/

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage.
```

If we replace the disaster placeholder `{0}` with `"earthquake"`, the sentence is not grammatically correct in English:

```
On January 1, 1999, a earthquake destroyed . . .
```

What we really want to do is integrate the article “a” into the placeholder:

```
"On {2}, {0} destroyed {1} houses and caused {3} of damage."
```

The `{0}` would then be replaced with `"a hurricane"` or `"an earthquake"`. That is especially appropriate if this message needs to be translated into a language where the gender of a word affects the article. For example, in German, the pattern would be

```
"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden v
```

The placeholder would then be replaced with the grammatically correct combination of article and noun, such as `"Ein Wirbelsturm"` or `"Eine Naturkatastrophe"`.

Now let us turn to the `{1}` parameter. If the disaster wasn’t all that catastrophic, `{1}` might be replaced with the number 1, and the message would read:

```
On January 1, 1999, a mudslide destroyed 1 houses and . . .
```

Ideally, we would like the message to vary according to the placeholder value, so it would read

```
no houses

one house
```

http://www.rejoiceblog.com/

```
2 houses

. . .
```

depending on the placeholder value. The `choice` formatting option was designed for this purpose.

A choice format is a sequence of pairs, each containing

- A *lower limit*
- A *format string*

The lower limit and format string are separated by a `#` character, and the pairs are separated by `|` characters.

For example,

```
{1,choice,0#no houses|1#one house|2#{1} houses}
```

Table 7.8 shows the effect of this format string for various values of `{1}`.

Table 7.8 String Formatted by Choice Format

{1}	Result	{1}	Result
0	"no houses"	3	"3 houses"
1	"one house"	-1	"no houses"

Why do we use `{1}` twice in the format string? When the message format applies the choice format to the `{1}` placeholder and the value is `2`, the choice format returns `"{1} houses"`. That string is then formatted again by the message format, and the answer is spliced into the result.

Note

This example shows that the designer of the choice format was a bit

http://www.rejoiceblog.com/

muddleheaded. If you have three format strings, you need two limits to separate them. In general, you need *one fewer limit* than you have format strings. As you saw in Table 7.8, the MessageFormat class ignores the first limit.

The syntax would have been a lot clearer if the designer of this class realized that the limits belong *between* the choices, such as

```
no houses|1|one house|2|{1} houses // not the actual format
```

You can use the < symbol to denote that a choice should be selected if the lower bound is strictly less than the value.

You can also use the ≤ symbol (expressed as the Unicode character code `\u2264`) as a synonym for `#`. If you like, you can even specify a lower bound of – ∞ as `-\u221E` for the first value.

For example,

```
-∞<no houses|0<one house|2≤{1} houses
```

or, using Unicode escapes,

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

Let's finish our natural disaster scenario. If we put the choice string inside the original message string, we get the following format instruction:

```
String pattern = "On {2,date,long}, {0} destroyed {1,choice,0

   houses}" + "and caused {3,number,currency} of damage.";
```

Or, in German,

http://www.rejoiceblog.com/

```
String pattern = "{0} zerstörte am {2,date,long} {1,choice,0#

   + "und richtete einen Schaden von {3,number,currency} an."
```

Note that the ordering of the words is different in German, but the array of objects you pass to the `format` method is the *same*. The order of the placeholders in the format string takes care of the changes in the word ordering.

7.6 TEXT INPUT AND OUTPUT

As you know, the Java programming language itself is fully Unicode-based. However, Windows and Mac OS X still support legacy character encodings such as Windows-1252 or Mac Roman in Western European countries, or Big5 in Taiwan. Therefore, communicating with your users through text is not as simple as it should be. The following sections discuss the complications that you may encounter.

7.6.1 Text Files

Nowadays, it is best to use UTF-8 for saving and loading text files. But you may need to work with legacy files. If you know the expected character encoding, you can specify it when writing or reading text files:

```
var out = new PrintWriter(filename, "Windows-1252");
```

For a guess of the best encoding to use, get the “platform encoding” by calling

```
Charset platformEncoding = Charset.defaultCharset();
```

7.6.2 Line Endings

This isn’t an issue of locales but of platforms. On Windows, text files are expected to use `\r\n` at the end of each line, where UNIX-based systems only require a `\n` character. Nowadays, most Windows programs can deal with just a `\n`. The notable exception is Notepad. If it is important to you that users can double-click on a text file that your application produces and view it in Notepad, make sure that the text file has proper line endings.

http://www.rejoiceblog.com/

Any line written with the `println` method will be properly terminated. The only problem is if you print strings that contain `\n` characters. They are not automatically modified to the platform line ending.

Instead of using `\n` in strings, you can use `printf` and the `%n` format specifier to produce platform-dependent line endings. For example,

```
out.printf("Hello%nWorld%n");
```

produces

```
Hello\r\nWorld\r\n
```

on Windows and

```
Hello\nWorld\n
```

everywhere else.

7.6.3 The Console

If you write programs that communicate with the user through `System.in`/`System.out` or `System.console()`, you have to face the possibility that the console may use a character encoding that is different from the platform encoding reported by `Charset.defaultCharset()`. This is particularly noticeable when working with the `cmd` shell on Windows. In the US version, the command shell uses the archaic `IBM437` encoding that originated with IBM personal computers in 1982. There is no official API for revealing that information. The `Charset.defaultCharset()` method will return the `Windows-1252` character set, which is quite different. For example, the euro symbol € is present in `Windows-1252` but not in `IBM437`. When you call

```
System.out.println("100 €");
```

the console will display

http://www.rejoiceblog.com/

```
100 ?
```

You can advise your users to switch the character encoding of the console. In Windows, that is achieved with the `chcp` command. For example,

```
chcp 1252
```

changes the console to the `Windows-1252` code page.

Ideally, of course, your users should switch the console to UTF-8. In Windows, the command is

```
chcp 65001
```

Unfortunately, that is not enough to make Java use UTF-8 in the console. It is also necessary to set the platform encoding with the unofficial `file.encoding` system property:

```
java -Dfile.encoding=UTF-8 MyProg
```

7.6.4 Log Files

When log messages from the `java.util.logging` library are sent to the console, they are written with the console encoding. You saw how to control that in the preceding section. However, log messages in a file use a `FileHandler` which, by default, uses the platform encoding.

To change the encoding to UTF-8, you need to change the log manager settings. In the logging configuration file, set

```
java.util.logging.FileHandler.encoding=UTF-8
```

7.6.5 The UTF-8 Byte Order Mark

As already mentioned, it is a good idea to use UTF-8 for text files when you can. If your

http://www.rejoiceblog.com/

application has to read UTF-8 text files that were created by other programs, you run into another potential problem. It is perfectly legal to add a “byte order mark” character U+FEFF as the first character of a file.

In the UTF-16 encoding, where each code unit is a two-byte quantity, the byte order mark tells a reader whether the file uses “big-endian” or “little-endian” byte ordering. UTF-8 is a single-byte encoding, so there is no need for specifying a byte ordering. But if a file starts with bytes 0xEF 0xBB 0xBF (the UTF-8 encoding of U+FEFF), that is a strong indication that it uses UTF-8. For that reason, the Unicode standard encourages this practice. Any reader is supposed to discard an initial byte order mark.

There is just one fly in the ointment. The Oracle Java implementation stubbornly refuses to follow the Unicode standard, citing potential compatibility issues. That means that you, the programmer, must do what the platform won’t do. When you read a text file and encounter a `U+FEFF` at the beginning, ignore it.

Caution

Unfortunately, the JDK implementors do not follow this advice. When you pass the `javac` compiler a valid UTF-8 source file that starts with a byte order mark, compilation fails with an error message “illegal character: \65279”.

7.6.6 Character Encoding of Source Files

You, the programmer, will need to communicate with the Java compiler—and you do that with tools on your local system. For example, you can use the Chinese version of Notepad to write your Java source code files. The resulting source code files are *not portable* because they use the local character encoding (GB or Big5, depending on which Chinese operating system you use). Only the compiled class files are portable—they will automatically use the “modified UTF-8” encoding for identifiers and strings. That means that when a program is compiling and running, three character encodings are involved:

• Source files: platform encoding

• Class files: modified UTF-8

http://www.rejoiceblog.com/

• Virtual machine: UTF-16

(See Chapter 1 for a definition of the modified UTF-8 and UTF-16 formats.)

Tip

You can specify the character encoding of your source files with the `-encoding` flag, for example,

```
javac -encoding UTF-8 Myfile.java
```

7.7 RESOURCE BUNDLES

When localizing an application, you'll probably have a dauntingly large number of message strings, button labels, and so on, that all need to be translated. To make this task feasible, you'll want to define the message strings in an external location, usually called a *resource*. The person carrying out the translation can then simply edit the resource files without having to touch the source code of the program.

In Java, you can use property files to specify string resources, and you can implement classes for resources of other types.

Note

Java technology resources are not the same as Windows or Macintosh resources. A Macintosh or Windows executable program stores resources, such as menus, dialog boxes, icons, and messages, in a section separate from the program code. A resource editor can inspect and update these resources without affecting the program code.

Note

http://www.rejoiceblog.com/

Volume I, Chapter 13 describes a concept of JAR file resources, whereby data files, sounds, and images can be placed in a JAR file. The `getResource` method of the class `Class` finds the file, opens it, and returns a URL to the resource. By placing the files into the JAR file, you leave the job of finding the files to the class loader, which already knows how to locate items in a JAR file. However, that mechanism has no locale support.

7.7.1 Locating Resource Bundles

When localizing an application, you produce a set of *resource bundles*. Each bundle is a property file or a class that describes locale-specific items (such as messages, labels, and so on). For each bundle, you have to provide versions for all locales that you want to support.

You need to use a specific naming convention for these bundles. For example, resources specific to Germany go into a file *baseName*`_de_DE`, whereas those shared by all German-speaking countries go into *baseName*`_de`. In general, use

baseName_language_country

for all country-specific resources, and use

baseName_language

for all language-specific resources. Finally, as a fallback, you can put defaults into a file without any suffix.

To load a bundle, use the command

```
ResourceBundle currentResources =
ResourceBundle.getBundle(baseName, currentLocale);
```

The `getBundle` method attempts to load the bundle that matches the current locale by language and country. If it is not successful, the country and the language are dropped in turn. Then the same search is applied to the default locale, and finally, the default bundle file is consulted. If even that attempt fails, the method throws a `MissingResourceException`.

That is, the `getBundle` method tries to load the following bundles:

http://www.rejoiceblog.com/

```
baseName_currentLocaleLanguage_currentLocaleCountry baseName_
```

Once the `getBundle` method has located a bundle (say, *baseName*_`de_DE`), it will still keep looking for *baseName*_`de` and *baseName*. If these bundles exist, they become the *parents* of the *baseName*_`de_DE` bundle in a *resource hierarchy*. Later, when looking up a resource, the parents are searched if a lookup was not successful in the current bundle. That is, if a particular resource was not found in *baseName*_`de_DE`, then the *baseName*_`de` and *baseName* will be queried as well.

This is clearly a very useful service—and one that would be tedious to program by hand. The resource bundle mechanism of the Java programming language automatically locates the items that are the best match for a given locale. It is easy to add more and more localizations to an existing program—all you have to do is create additional resource bundles.

Note

We simplified the discussion of resource bundle lookup. If a locale has a script or variant, the lookup is quite a bit more complex. See the documentation of the method `ResourceBundle.Control.getCandidateLocales` for the gory details.

Tip

You need not place all resources for your application into a single bundle. You could have one bundle for button labels, one for error messages, and so on.

7.7.2 Property Files

Internationalizing strings is quite straightforward. You place all your strings into a property file such as `MyProgramStrings.properties`. This is simply a text file with

http://www.rejoiceblog.com/
one key/value pair per line. A typical file would look like this:

```
computeButton=Rechnen

colorName=black

defaultPaperSize=210×297
```

Then you name your property files as described in the preceding section, for example:

```
MyProgramStrings.properties

MyProgramStrings_en.properties

MyProgramStrings_de_DE.properties
```

You can load the bundle simply as

```
ResourceBundle bundle = ResourceBundle.getBundle("MyProgramSt
```

To look up a specific string, call

```
String computeButtonLabel = bundle.getString("computeButton")
```

Caution

Before Java 9, files for storing properties had to be ASCII files. If you are using an old Java version and need to place a Unicode character into a property file, encode it using the `\uxxxx` encoding. For example, to specify `"colorName=Grün"`, use

```
colorName=Gr\u00FCn
```

http://www.rejoiceblog.com/

You can use the `native2ascii` tool to generate these encodings.

7.7.3 Bundle Classes

To provide resources that are not strings, define classes that extend the `ResourceBundle` class. Use the standard naming convention to name your classes, for example

```
MyProgramResources.java

MyProgramResources_en.java

MyProgramResources_de_DE.java
```

Load the class with the same `getBundle` method that you use to load a property file:

```
ResourceBundle bundle = ResourceBundle.getBundle("MyProgramRe
```

Caution

When searching for bundles, a bundle in a class is given preference over a property file when the two bundles have the same base names.

Each resource bundle class implements a lookup table. You need to provide a key string for each setting you want to localize, and use that key string to retrieve the setting. For example,

```
var backgroundColor = (Color) bundle.getObject("backgroundCol

double[] paperSize = (double[]) bundle.getObject("defaultPape
```

http://www.rejoiceblog.com/

The simplest way to implement resource bundle classes is to extend the `ListResourceBundle` class. The `ListResourceBundle` lets you place all your resources into an object array and then does the lookup for you. Follow this code outline:

```
public class baseName_language_country extends ListResourceBundle

{

   private static final Object[][] contents =

   {

      { key1, value2},

      { key2, value2},

      . . .

   }

   public Object[][] getContents() { return contents; }

}
```

For example,

```
public class ProgramResources_de extends ListResourceBundle

{

   private static final Object[][] contents =

   {

      { "backgroundColor", Color.black },
```

http://www.rejoiceblog.com/

```
      { "defaultPaperSize", new double[] { 210, 297 } }

    }

    public Object[][] getContents() { return contents; }

  }

  public class ProgramResources_en_US extends ListResourceBundl

  {

    private static final Object[][] contents =

    {

      { "backgroundColor", Color.blue },

      { "defaultPaperSize", new double[] { 216, 279 } }

    }

    public Object[][] getContents() { return contents; }

  }
```

Note

The paper sizes are given in millimeters. Everyone on the planet, with the exception of the United States and Canada, uses ISO 216 paper sizes. For more information, see `www.cl.cam.ac.uk/~mgk25/iso-paper.html`.

Alternatively, your resource bundle classes can extend the `ResourceBundle` class. Then you need to implement two methods, to enumerate all keys and to look up the value for a given key:

http://www.rejoiceblog.com/

```
Enumeration<String> getKeys()

Object handleGetObject(String key)
```

The `getObject` method of the `ResourceBundle` class calls the `handleGetObject` method that you supply.

java.util.ResourceBundle 1.1

- `static ResourceBundle getBundle(String baseName, Locale loc)`
- `static ResourceBundle getBundle(String baseName)`

loads the resource bundle class with the given name, for the given locale or the default locale, and its parent classes. If the resource bundle classes are located in a package, the base name must contain the full package name, such as `"intl.ProgramResources"`. The resource bundle classes must be `public` so that the `getBundle` method can access them.

- `Object getObject(String name)`

looks up an object from the resource bundle or its parents.

- `String getString(String name)`

looks up an object from the resource bundle or its parents and casts it as a string.

- `String[] getStringArray(String name)`

looks up an object from the resource bundle or its parents and casts it as a string array.

- `Enumeration<String> getKeys()`

returns an enumeration object to enumerate the keys of this resource bundle. It enumerates the keys in the parent bundles as well.

- `Object handleGetObject(String key)`

should be overridden to look up the resource value associated with the given key if you

http://www.rejoiceblog.com/

define your own resource lookup mechanism.

7.8 A COMPLETE EXAMPLE

In this section, we apply the material of this chapter to localize a retirement calculator. The program calculates whether or not you are saving enough money for your retirement. You enter your age, how much money you save every month, and so on (see Figure 7.4).

Figure 7.4 The retirement calculator in English

The text area and the graph show the balance of the retirement account for every year. If the numbers turn negative toward the later part of your life and the bars in the graph appear below the *x* axis, you need to do something—for example, save more money, postpone your retirement, die earlier, or be younger.

The retirement calculator works in three locales (English, German, and Chinese). Here are some of the highlights of the internationalization:

• The labels, buttons, and messages are translated into German and Chinese. You can find them in the classes `RetireResources_de` and `RetireResources_zh`. English is used as the fallback—see the `RetireResources` file.

• Whenever the locale changes, we reset the labels and reformat the contents of the text fields.

http://www.rejoiceblog.com/

• The text fields handle numbers, currency amounts, and percentages in the local format.

• The computation field uses a `MessageFormat`. The format string is stored in the resource bundle of each language.

• Just to show that it can be done, we use different colors for the bar graph, depending on the language chosen by the user.

Listings 7.5 through 7.8 show the code. Listings 7.9 through 7.11 are the property files for the localized strings. Figures 7.5 and 7.6 show the outputs in German and Chinese, respectively. To see Chinese characters, be sure you have Chinese fonts installed and configured with your Java runtime. Otherwise, Chinese characters will show up as "missing character" icons.

Figure 7.5 The retirement calculator in German

http://www.rejoiceblog.com/

Figure 7.6 The retirement calculator in Chinese

Listing 7.5 `retire/Retire.java`

```
package retire;

import java.awt.*;
import java.awt.geom.*;
import java.text.*;
import java.util.*;

import javax.swing.*;

/**
```

http://www.rejoiceblog.com/

```
 * This program shows a retirement calculator. The UI is
 * @version 1.25 2018-05-01
 * @author Cay Horstmann
 */
public class Retire
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() ->
         {
            var frame = new RetireFrame();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_
            frame.setVisible(true);
         });
   }
}

class RetireFrame extends JFrame
{
   private JTextField savingsField = new JTextField(10);
```

http://www.rejoiceblog.com/

```
31      private JTextField contribField = new JTextField(10);
32      private JTextField incomeField = new JTextField(10);
33      private JTextField currentAgeField = new JTextField(4)
34      private JTextField retireAgeField = new JTextField(4);
35      private JTextField deathAgeField = new JTextField(4);
36      private JTextField inflationPercentField = new JTextFi
37      private JTextField investPercentField = new JTextField
38      private JTextArea retireText = new JTextArea(10, 25);
39      private RetireComponent retireCanvas = new RetireCompo
40      private JButton computeButton = new JButton();
41      private JLabel languageLabel = new JLabel();
42      private JLabel savingsLabel = new JLabel();
43      private JLabel contribLabel = new JLabel();
44      private JLabel incomeLabel = new JLabel();
45      private JLabel currentAgeLabel = new JLabel();
46      private JLabel retireAgeLabel = new JLabel();
47      private JLabel deathAgeLabel = new JLabel();
48      private JLabel inflationPercentLabel = new JLabel();
49      private JLabel investPercentLabel = new JLabel();
```

http://www.rejoiceblog.com/

```
    private RetireInfo info = new RetireInfo();
    private Locale[] locales = { Locale.US, Locale.CHINA, 
    private Locale currentLocale;
    private JComboBox<Locale> localeCombo = new LocaleComb
    private ResourceBundle res;
    private ResourceBundle resStrings;
    private NumberFormat currencyFmt;
    private NumberFormat numberFmt;
    private NumberFormat percentFmt;

    public RetireFrame()
    {
      setLayout(new GridBagLayout());
      add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST)
      add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST))
      add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST))
      add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
      add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAS
      add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST
      add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST)
```

http://www.rejoiceblog.com/

```
      add(inflationPercentLabel, new GBC(0, 3).setAnchor(G
      add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.
      add(localeCombo, new GBC(1, 0, 3, 1));
      add(savingsField, new GBC(1, 1).setWeight(100, 0).se
      add(contribField, new GBC(3, 1).setWeight(100, 0).se
      add(incomeField, new GBC(5, 1).setWeight(100, 0).set
      add(currentAgeField, new GBC(1, 2).setWeight(100, 0)
      add(retireAgeField, new GBC(3, 2).setWeight(100, 0).
      add(deathAgeField, new GBC(5, 2).setWeight(100, 0).s
      add(inflationPercentField, new GBC(1, 3).setWeight(1
      add(investPercentField, new GBC(3, 3).setWeight(100,
      add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100,
      add(new JScrollPane(retireText), new GBC(4, 4, 2, 1)

      computeButton.setName("computeButton");
      computeButton.addActionListener(event ->
        {
          getInfo();
          updateData();
```

http://www.rejoiceblog.com/

```
            updateGraph();
          });
        add(computeButton, new GBC(5, 3));

        retireText.setEditable(false);
        retireText.setFont(new Font("Monospaced", Font.PLAIN

        info.setSavings(0);
        info.setContrib(9000);
        info.setIncome(60000);
        info.setCurrentAge(35);
        info.setRetireAge(65);
        info.setDeathAge(85);
        info.setInvestPercent(0.1);
        info.setInflationPercent(0.05);

        int localeIndex = 0; // US locale is default selecti
        for (int i = 0; i < locales.length; i++)
          // if current locale one of the choices, select it
          if (getLocale().equals(locales[i])) localeIndex =
```

http://www.rejoiceblog.com/

```
    setCurrentLocale(locales[localeIndex]);

    localeCombo.addActionListener(event ->
      {
        setCurrentLocale((Locale) localeCombo.getSelecte
        validate();
      });
    pack();
  }

  /**
   * Sets the current locale.
   * @param locale the desired locale
   */
  public void setCurrentLocale(Locale locale)
  {
    currentLocale = locale;
    localeCombo.setLocale(currentLocale);
    localeCombo.setSelectedItem(currentLocale);
```

http://www.rejoiceblog.com/

```
128
129     res = ResourceBundle.getBundle("retire.RetireResource
130     resStrings = ResourceBundle.getBundle("retire.RetireS
131     currencyFmt = NumberFormat.getCurrencyInstance(curren
132     numberFmt = NumberFormat.getNumberInstance(currentLoc
133     percentFmt = NumberFormat.getPercentInstance(currentL
134
135     updateDisplay();
136     updateInfo();
137     updateData();
138     updateGraph();
139   }
140
141   /**
142    * Updates all labels in the display.
143    */
144   public void updateDisplay()
145   {
146     languageLabel.setText(resStrings.getString("language"
147     savingsLabel.setText(resStrings.getString("savings"))
```

http://www.rejoiceblog.com/

```
    contribLabel.setText(resStrings.getString("contrib"))
    incomeLabel.setText(resStrings.getString("income"));
    currentAgeLabel.setText(resStrings.getString("current
    retireAgeLabel.setText(resStrings.getString("retireAg
    deathAgeLabel.setText(resStrings.getString("deathAge"
    inflationPercentLabel.setText(resStrings.getString("i
    investPercentLabel.setText(resStrings.getString("inve
    computeButton.setText(resStrings.getString("computeBu
  }

  /**
   * Updates the information in the text fields.
   */
  public void updateInfo()
  {
    savingsField.setText(currencyFmt.format(info.getSavin
    contribField.setText(currencyFmt.format(info.getContr
    incomeField.setText(currencyFmt.format(info.getIncome
    currentAgeField.setText(numberFmt.format(info.getCurr
```

http://www.rejoiceblog.com/

```
    retireAgeField.setText(numberFmt.format(info.getRetir
    deathAgeField.setText(numberFmt.format(info.getDeathA
    investPercentField.setText(percentFmt.format(info.get
    inflationPercentField.setText(percentFmt.format(info.
  }

  /**
   * Updates the data displayed in the text area.
   */
  public void updateData()
  {
    retireText.setText("");
    var retireMsg = new MessageFormat("");
    retireMsg.setLocale(currentLocale);
    retireMsg.applyPattern(resStrings.getString("retire")

    for (int i = info.getCurrentAge(); i <= info.getDeath
    {
      Object[] args = { i, info.getBalance(i) };
      retireText.append(retireMsg.format(args) + "\n");
```

http://www.rejoiceblog.com/

```
    }
  }

  /**
   * Updates the graph.
   */
  public void updateGraph()
  {
    retireCanvas.setColorPre((Color) res.getObject("color
    retireCanvas.setColorGain((Color) res.getObject("colo
    retireCanvas.setColorLoss((Color) res.getObject("colo
    retireCanvas.setInfo(info);
    repaint();
  }

  /**
   * Reads the user input from the text fields.
   */
  public void getInfo()
```

http://www.rejoiceblog.com/

```
  {
    try
    {
      info.setSavings(currencyFmt.parse(savingsField.getT
      info.setContrib(currencyFmt.parse(contribField.getT
      info.setIncome(currencyFmt.parse(incomeField.getTex
      info.setCurrentAge(numberFmt.parse(currentAgeField.
      info.setRetireAge(numberFmt.parse(retireAgeField.ge
      info.setDeathAge(numberFmt.parse(deathAgeField.getT
      info.setInvestPercent(percentFmt.parse(investPercen
      info.setInflationPercent(percentFmt.parse(inflation
    }
    catch (ParseException ex)
    {
      ex.printStackTrace();
    }
  }
}

/**
```

http://www.rejoiceblog.com/

```
 * The information required to compute retirement income
 */
class RetireInfo
{
    private double savings;
    private double contrib;
    private double income;
    private int currentAge;
    private int retireAge;
    private int deathAge;
    private double inflationPercent;
    private double investPercent;
    private int age;
    private double balance;

/**
 * Gets the available balance for a given year.
 * @param year the year for which to compute the balance
 * @return the amount of money available (or required) i
```

http://www.rejoiceblog.com/

```
 */
public double getBalance(int year)
{
    if (year < currentAge) return 0;
    else if (year == currentAge)
    {
      age = year;
      balance = savings;
      return balance;
    }
    else if (year == age) return balance;
    if (year != age + 1) getBalance(year - 1);
    age = year;
    if (age < retireAge) balance += contrib;
    else balance -= income;
    balance = balance * (1 + (investPercent - inflationPe
    return balance;
}

/**
```

http://www.rejoiceblog.com/

```
 * Gets the amount of prior savings.
 * @return the savings amount
 */
public double getSavings()
{
   return savings;
}

/**
 * Sets the amount of prior savings.
 * @param newValue the savings amount
 */
public void setSavings(double newValue)
{
   savings = newValue;
}

/**
 * Gets the annual contribution to the retirement accoun
```

http://www.rejoiceblog.com/

```
 * @return the contribution amount
 */
public double getContrib()
{
    return contrib;
}

/**
 * Sets the annual contribution to the retirement accoun
 * @param newValue the contribution amount
 */
public void setContrib(double newValue)
{
    contrib = newValue;
}

/**
 * Gets the annual income.
 * @return the income amount
 */
```

http://www.rejoiceblog.com/

```
304  public double getIncome()
305  {
306     return income;
307  }
308
309  /**
310   * Sets the annual income.
311   * @param newValue the income amount
312   */
313  public void setIncome(double newValue)
314  {
315     income = newValue;
316  }
317
318  /**
319   * Gets the current age.
320   * @return the age
321   */
322  public int getCurrentAge()
```

http://www.rejoiceblog.com/

```
{
    return currentAge;
}

/**
 * Sets the current age.
 * @param newValue the age
*/
public void setCurrentAge(int newValue)
{
    currentAge = newValue;
}

/**
 * Gets the desired retirement age.
 * @return the age
 */
public int getRetireAge()
{
    return retireAge;
```

http://www.rejoiceblog.com/

```
343  }
344
345  /**
346   * Sets the desired retirement age.
347   * @param newValue the age
348   */
349  public void setRetireAge(int newValue)
350  {
351     retireAge = newValue;
352  }
353
354  /**
355   * Gets the expected age of death.
356   * @return the age
357   */
358  public int getDeathAge()
359  {
360     return deathAge;
361  }
```

http://www.rejoiceblog.com/

```
362
363  /**
364   * Sets the expected age of death.
365   * @param newValue the age
366   */
367  public void setDeathAge(int newValue)
368  {
369     deathAge = newValue;
370  }
371
372  /**
373   * Gets the estimated percentage of inflation.
374   * @return the percentage
375   */
376  public double getInflationPercent()
377  {
378     return inflationPercent;
379  }
380
381  /**
```

http://www.rejoiceblog.com/

```
 * Sets the estimated percentage of inflation.
 * @param newValue the percentage
 */
public void setInflationPercent(double newValue)
{
    inflationPercent = newValue;
}

/**
 * Gets the estimated yield of the investment.
 * @return the percentage
 */
public double getInvestPercent()
{
    return investPercent;
}

/**
 * Sets the estimated yield of the investment.
```

http://www.rejoiceblog.com/

```
 * @param newValue the percentage
 */
public void setInvestPercent(double newValue)
{
    investPercent = newValue;
}
}

/**
 * This component draws a graph of the investment result.
 */
class RetireComponent extends JComponent
{
  private static final int PANEL_WIDTH = 400;
  private static final int PANEL_HEIGHT = 200;
  private static final Dimension PREFERRED_SIZE = new Dim
  private RetireInfo info = null;
  private Color colorPre;
  private Color colorGain;
  private Color colorLoss;
```

http://www.rejoiceblog.com/

```
421
422   public RetireComponent()
423   {
424     setSize(PANEL_WIDTH, PANEL_HEIGHT);
425   }
426
427  /**
428   * Sets the retirement information to be plotted.
429   * @param newInfo the new retirement info.
430   */
431  public void setInfo(RetireInfo newInfo)
432  {
433     info = newInfo;
434     repaint();
435  }
436
437  public void paintComponent(Graphics g)
438  {
439     var g2 = (Graphics2D) g;
```

http://www.rejoiceblog.com/

```
    if (info == null) return;

    double minValue = 0;
    double maxValue = 0;
    int i;
    for (i = info.getCurrentAge(); i <= info.getDeathAge(
    {
      double v = info.getBalance(i);
      if (minValue > v) minValue = v;
      if (maxValue < v) maxValue = v;
    }
    if (maxValue == minValue) return;

    int barWidth = getWidth() / (info.getDeathAge() - inf
    double scale = getHeight() / (maxValue - minValue);

    for (i = info.getCurrentAge(); i <= info.getDeathAge(
    {
      int x1 = (i - info.getCurrentAge()) * barWidth + 1;
      int y1;
```

http://www.rejoiceblog.com/

```
460         double v = info.getBalance(i);
461         int height;
462         int yOrigin = (int) (maxValue * scale);
463
464         if (v >= 0)
465         {
466           y1 = (int) ((maxValue - v) * scale);
467           height = yOrigin - y1;
468         }
469         else
470         {
471           y1 = yOrigin;
472           height = (int) (-v * scale);
473         }
474
475         if (i < info.getRetireAge()) g2.setPaint(colorPre);
476         else if (v >= 0) g2.setPaint(colorGain);
477         else g2.setPaint(colorLoss);
478         var bar = new Rectangle2D.Double(x1, y1, barWidth -
```

http://www.rejoiceblog.com/

```
      g2.fill(bar);

      g2.setPaint(Color.black);

      g2.draw(bar);

    }

  }

/**

 * Sets the color to be used before retirement.

 * @param color the desired color

 */

public void setColorPre(Color color)

{

    colorPre = color;

    repaint();

}

/**

 * Sets the color to be used after retirement while the

 * @param color the desired color

 */
```

http://www.rejoiceblog.com/

```
public void setColorGain(Color color)
{
    colorGain = color;
    repaint();
}

/**
 * Sets the color to be used after retirement when the a
 * @param color the desired color
 */
public void setColorLoss(Color color)
{
    colorLoss = color;
    repaint();
}

 public Dimension getPreferredSize() { return PREFERRED_
}
```

Listing 7.6 `retire/RetireResources.java`

http://www.rejoiceblog.com/

```
1   package retire;
2
3   import java.awt.*;
4
5   /**
6    * These are the English non-string resources for the ret
7    * @version 1.21 2001-08-27
8    * @author Cay Horstmann
9    */
10  public class RetireResources extends java.util.ListResour
11  {
12     private static final Object[][] contents = {
13     // BEGIN LOCALIZE
14           { "colorPre", Color.blue }, { "colorGain", Color
15     // END LOCALIZE
16     };
17
18     public Object[][] getContents()
19     {
```

http://www.rejoiceblog.com/

```
20        return contents;

21      }

22  }
```

Listing 7.7 `retire/RetireResources_de.java`

```
1   package retire;

2

3   import java.awt.*;

4

5   /**

6    * These are the German non-string resources for the reti

7    * @version 1.21 2001-08-27

8    * @author Cay Horstmann

9    */

10  public class RetireResources_de extends java.util.ListRes

11  {

12      private static final Object[][] contents = {

13      // BEGIN LOCALIZE

14            { "colorPre", Color.yellow }, { "colorGain", Col

15      // END LOCALIZE
```

http://www.rejoiceblog.com/

```
    };

    public Object[][] getContents()
    {
      return contents;
    }
}
```

Listing 7.8 `retire/RetireResources_zh.java`

```
package retire;

import java.awt.*;

/**
 * These are the Chinese non-string resources for the ret
 * @version 1.21 2001-08-27
 * @author Cay Horstmann
 */
public class RetireResources_zh extends java.util.ListRes
```

http://www.rejoiceblog.com/

```
{
    private static final Object[][] contents = {
    // BEGIN LOCALIZE
          { "colorPre", Color.red }, { "colorGain", Color.
    // END LOCALIZE
    };

    public Object[][] getContents()
    {
        return contents;
    }
}
```

Listing 7.9 `retire/RetireStrings.properties`

```
language=Language
computeButton=Compute
savings=Prior Savings
contrib=Annual Contribution
income=Retirement Income
currentAge=Current Age
```

http://www.rejoiceblog.com/

```
retireAge=Retirement Age
deathAge=Life Expectancy
inflationPercent=Inflation
investPercent=Investment Return
retire=Age: {0,number} Balance: {1,number,currency}
```

Listing 7.10 `retire/RetireStrings_de.properties`

```
language=Sprache
computeButton=Rechnen
savings=Vorherige Ersparnisse
contrib=Jährliche Einzahlung
income=Einkommen nach Ruhestand
currentAge=Jetziges Alter
retireAge=Ruhestandsalter
deathAge=Lebenserwartung
inflationPercent=Inflation
investPercent=Investitionsgewinn
retire=Alter: {0,number} Guthaben: {1,number,currency}
```

Listing 7.11 `retire/RetireStrings_zh.properties`

http://www.rejoiceblog.com/

```
language=
computeButton=
savings=
contrib=
income=
currentAge=
retireAge=
deathAge=
inflationPercent=
investPercent=
retire= : {0,number} : {1,number,currency}
```

You have seen how to use the internationalization features of the Java language. You can now use resource bundles to provide translations into multiple languages, and use formatters and collators for locale-specific text processing.

In the next chapter, we will delve into scripting, compiling, and annotation processing.

http://www.rejoiceblog.com/

Chapter 8 Scripting, Compiling, and Annotation Processing

In this chapter

• 8.1 Scripting for the Java Platform

• 8.2 The Compiler API

• 8.3 Using Annotations

• 8.4 Annotation Syntax

• 8.5 Standard Annotations

• 8.6 Source-Level Annotation Processing

• 8.7 Bytecode Engineering

This chapter introduces three techniques for processing code. The scripting API lets you invoke code in a scripting language such as JavaScript or Groovy. You can use the compiler API when you want to compile Java code inside your application. Annotation processors operate on Java source or class files that contain annotations. As you will see, there are many applications for annotation processing, ranging from simple diagnostics to "bytecode engineering"—the insertion of bytecodes into class files or even running programs.

8.1 SCRIPTING FOR THE JAVA PLATFORM

A scripting language is a language that avoids the usual edit/compile/link/run cycle by interpreting the program text at runtime. Scripting languages have a number of advantages:

• Rapid turnaround, encouraging experimentation

• Changing the behavior of a running program

http://www.rejoiceblog.com/

• Enabling customization by program users

On the other hand, most scripting languages lack features that are beneficial for programming complex applications, such as strong typing, encapsulation, and modularity.

It is therefore tempting to combine the advantages of scripting and traditional languages. The scripting API lets you do just that for the Java platform. It enables you to invoke scripts written in JavaScript, Groovy, Ruby, and even exotic languages such as Scheme and Haskell, from a Java program. For example, the Renjin project (`www.renjin.org`) provides a Java implementation of the R programming language, which is commonly used for statistical programming, together with an "engine" of the scripting API.

In the following sections, we'll show you how to select an engine for a particular language, how to execute scripts, and how to make use of advanced features that some scripting engines offer.

8.1.1 Getting a Scripting Engine

A scripting engine is a library that can execute scripts in a particular language. When the virtual machine starts, it discovers the available scripting engines. To enumerate them, construct a `ScriptEngineManager` and invoke the `getEngineFactories` method. You can ask each engine factory for the supported engine names, MIME types, and file extensions. Table 8.1 shows typical values.

Table 8.1 Properties of Scripting Engine Factories

Engine	Names	MIME types	Extensions
Nashorn (included with the JDK)	nashorn, Nashorn, js, JS, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript	js
Groovy	groovy	None	groovy
Renjin	Renjin	text/x-R	R, r, S, s
SISC Scheme	sisc	None	scheme, sisc

Usually, you know which engine you need, and you can simply request it by name, MIME type, or extension. For example:

http://www.rejoiceblog.com/

```
ScriptEngine engine = manager.getEngineByName("nashorn");
```

Java 8 introduced Nashorn, a JavaScript interpreter developed by Oracle. You can add more languages by providing the necessary JAR files on the class path.

javax.script.ScriptEngineManager 6

- `List<ScriptEngineFactory> getEngineFactories()`

gets a list of all discovered engine factories.

- `ScriptEngine getEngineByName(String name)`
- `ScriptEngine getEngineByExtension(String extension)`
- `ScriptEngine getEngineByMimeType(String mimeType)`

gets the script engine with the given name, script file extension, or MIME type.

javax.script.ScriptEngineFactory 6

- `List<String> getNames()`
- `List<String> getExtensions()`
- `List<String> getMimeTypes()`

gets the names, script file extensions, and MIME types under which this factory is known.

8.1.2 Script Evaluation and Bindings

Once you have an engine, you can call a script simply by invoking

```
Object result = engine.eval(scriptString);
```

http://www.rejoiceblog.com/

If the script is stored in a file, open a `Reader` and call

```
Object result = engine.eval(reader);
```

You can invoke multiple scripts on the same engine. If one script defines variables, functions, or classes, most scripting engines retain the definitions for later use. For example,

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

will return `1729`.

Note

To find out whether it is safe to concurrently execute scripts in multiple threads, call

```
Object param = factory.getParameter("THREADING");
```

The returned value is one of the following:

- `null`: Concurrent execution is not safe.
- `"MULTITHREADED"`: Concurrent execution is safe. Effects from one thread might be visible from another thread.
- `"THREAD-ISOLATED"`: In addition to `"MULTITHREADED"`, different variable bindings are maintained for each thread.
- `"STATELESS"`: In addition to `"THREAD-ISOLATED"`, scripts do not alter variable bindings.

You will often want to add variable bindings to the engine. A binding consists of a name and an associated Java object. For example, consider these statements:

http://www.rejoiceblog.com/

```
engine.put("k", 1728);

Object result = engine.eval("k + 1");
```

The script code reads the definition of `k` from the bindings in the "engine scope." This is particularly important because most scripting languages can access Java objects, often with a syntax that is simpler than the Java syntax. For example,

```
engine.put("b", new JButton());

engine.eval("b.text = 'Ok'");
```

Conversely, you can retrieve variables that were bound by scripting statements:

```
engine.eval("n = 1728");

Object result = engine.get("n");
```

In addition to the engine scope, there is also a global scope. Any bindings that you add to the `ScriptEngineManager` are visible to all engines.

Instead of adding bindings to the engine or global scope, you can collect them in an object of type `Bindings` and pass it to the `eval` method:

```
Bindings scope = engine.createBindings();

scope.put("b", new JButton());

engine.eval(scriptString, scope);
```

This is useful if a set of bindings should not persist for future calls to the `eval` method.

Note

http://www.rejoiceblog.com/

You might want to have scopes other than the engine and global scopes. For example, a web container might need request and session scopes. However, then you are on your own. You will need to write a class that implements the `ScriptContext` interface, managing a collection of scopes. Each scope is identified by an integer number, and scopes with lower numbers should be searched first. (The standard library provides a `SimpleScriptContext` class, but it only holds global and engine scopes.)

javax.script.ScriptEngine 6

- `Object eval(String script)`
- `Object eval(Reader reader)`
- `Object eval(String script, Bindings bindings)`
- `Object eval(Reader reader, Bindings bindings)`

evaluates the script given by the string or reader, subject to the given bindings.

- `Object get(String key)`
- `void put(String key, Object value)`

gets or puts a binding in the engine scope.

- `Bindings createBindings()`

creates an empty `Bindings` object suitable for this engine.

javax.script.ScriptEngineManager 6

- `Object get(String key)`
- `void put(String key, Object value)`

gets or puts a binding in the global scope.

http://www.rejoiceblog.com/

javax.script.Bindings 6

- `Object get(String key)`
- `void put(String key, Object value)`

gets or puts a binding into the scope represented by this `Bindings` object.

8.1.3 Redirecting Input and Output

You can redirect the standard input and output of a script by calling the `setReader` and `setWriter` methods of the script context. For example,

```
var writer = new StringWriter();

engine.getContext().setWriter(new PrintWriter(writer, true));
```

Any output written with the JavaScript `print` or `println` functions is sent to `writer`.

The `setReader` and `setWriter` methods only affect the scripting engine's standard input and output sources. For example, if you execute the JavaScript code

```
println("Hello");

java.lang.System.out.println("World");
```

only the first output is redirected.

The Nashorn engine does not have the notion of a standard input source. Calling `setReader` has no effect.

javax.script.ScriptEngine 6

- `ScriptContext getContext()`

http://www.rejoiceblog.com/

gets the default script context for this engine.

javax.script.ScriptContext 6

- `Reader getReader()`
- `void setReader(Reader reader)`
- `Writer getWriter()`
- `void setWriter(Writer writer)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer writer)`

gets or sets the reader for input or writer for normal or error output.

8.1.4 Calling Scripting Functions and Methods

With many script engines, you can invoke a function in the scripting language without having to evaluate the actual script code. This is useful if you allow users to implement a service in a scripting language of their choice.

The script engines that offer this functionality implement the `Invocable` interface. In particular, the Nashorn engine implements `Invocable`.

To call a function, call the `invokeFunction` method with the function name, followed by the function parameters:

```
// Define greet function in JavaScript

engine.eval("function greet(how, whom) { return how + ', ' + 
```

```
// Call the function with arguments "Hello", "World"
```

http://www.rejoiceblog.com/

```
result = ((Invocable) engine).invokeFunction("greet", "Hello"
```

If the scripting language is object-oriented, call `invokeMethod`:

```
// Define Greeter class in JavaScript

engine.eval("function Greeter(how) { this.how = how }");

engine.eval("Greeter.prototype.welcome = "

   + " function(whom) { return this.how + ', ' + whom + '!' }
```

```
// Construct an instance

Object yo = engine.eval("new Greeter('Yo')");
```

```
// Call the welcome method on the instance

result = ((Invocable) engine).invokeMethod(yo, "welcome", "Wo
```

Note

For more information on how to define classes in JavaScript, see *JavaScript—The Good Parts* by Douglas Crockford (O'Reilly, 2008).

Note

If the script engine does not implement the `Invocable` interface, you might still be able to call a method in a language-independent way. The

http://www.rejoiceblog.com/

> `getMethodCallSyntax` method of the `ScriptEngineFactory` interface produces a string that you can pass to the `eval` method. However, all method parameters must be bound to names, whereas `invokeMethod` can be called with arbitrary values.

You can go a step further and ask the scripting engine to implement a Java interface. Then you can call scripting functions and methods with the Java method call syntax.

The details depend on the scripting engine, but typically you need to supply a function for each method of the interface. For example, consider a Java interface

```
public interface Greeter

{

   String welcome(String whom);

}
```

If you define a global function with the same name in Nashorn, you can call it through this interface.

```
// Define welcome function in JavaScript

engine.eval("function welcome(whom) { return 'Hello, ' + whom
```

```
// Get a Java object and call a Java method

Greeter g = ((Invocable) engine).getInterface(Greeter.class);

result = g.welcome("World");
```

In an object-oriented scripting language, you can access a script class through a matching Java interface. For example, here is how to call an object of the JavaScript

http://www.rejoiceblog.com/

`SimpleGreeter` class with Java syntax:

```
Greeter g = ((Invocable) engine).getInterface(yo, Greeter.cla

result = g.welcome("World");
```

In summary, the `Invocable` interface is useful if you want to call scripting code from Java without worrying about the scripting language syntax.

javax.script.Invocable 6

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`

invokes the function or method with the given name, passing the given parameters.

- `<T> T getInterface(Class<T> iface)`

returns an implementation of the given interface, implementing the methods with functions in the scripting engine.

- `<T> T getInterface(Object implicitParameter, Class<T> iface)`

returns an implementation of the given interface, implementing the methods with the methods of the given object.

8.1.5 Compiling a Script

Some scripting engines can compile scripting code into an intermediate form for efficient execution. Those engines implement the `Compilable` interface. The following example shows how to compile and evaluate code contained in a script file:

```
var reader = new FileReader("myscript.js");

CompiledScript script = null;
```

http://www.rejoiceblog.com/

```
if (engine implements Compilable)
   script = ((Compilable) engine).compile(reader);
```

Once the script is compiled, you can execute it. The following code executes the compiled script if compilation was successful, or the original script if the engine didn't support compilation:

```
if (script != null)
   script.eval();
else
   engine.eval(reader);
```

Of course, it only makes sense to compile a script if you need to execute it repeatedly.

javax.script.Compilable 6
• `CompiledScript compile(String script)` • `CompiledScript compile(Reader reader)` compiles the script given by a string or reader.

javax.script.CompiledScript 6
• `Object eval()` • `Object eval(Bindings bindings)` evaluates this script.

http://www.rejoiceblog.com/

8.1.6 An Example: Scripting GUI Events

To illustrate the scripting API, we will write a sample program that allows users to specify event handlers in a scripting language of their choice.

Have a look at the program in Listing 8.1 that adds scripting to an arbitrary frame class. By default it reads the `ButtonFrame` class in Listing 8.2, which is similar to the event handling demo in Volume I, with two differences:

- Each component has its `name` property set.
- There are no event handlers.

The event handlers are defined in a property file. Each property definition has the form

```
componentName.eventName = scriptCode
```

For example, if you choose to use JavaScript, supply the event handlers in a file `js.properties`, like this:

```
yellowButton.action=panel.background = java.awt.Color.YELLOW

blueButton.action=panel.background = java.awt.Color.BLUE

redButton.action=panel.background = java.awt.Color.RED
```

The companion code also has files for Groovy, R, and SISC Scheme.

The program starts by loading an engine for the language specified on the command line. If no language is specified, we use JavaScript.

We then process a script `init.`*language* if it is present. This is useful for the R and Scheme languages, which need some initializations that we did not want to include in every event handler script.

Next, we recursively traverse all child components and add the bindings (*name*, *object*) into a map of components. Then we add the bindings to the engine.

Next, we read the file *language*`.properties`. For each property, we synthesize an

http://www.rejoiceblog.com/
event handler proxy that causes the script code to be executed. The details are a bit technical. You might want to read the section on proxies in Volume I, Chapter 6, if you want to follow the implementation in detail. The essential part, however, is that each event handler calls

```
engine.eval(scriptCode);
```

Let us look at the `yellowButton` in more detail. When the line

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

is processed, we find the `JButton` component with the name `"yellowButton"`. We then attach an `ActionListener` with an `actionPerformed` method that executes the script

```
panel.background = java.awt.Color.YELLOW
```

if the scripting is done with Nashorn.

The engine contains a binding that binds the name `"panel"` to the `JPanel` object. When the event occurs, the `setBackground` method of the panel is executed, and the color changes.

You can run this program with the JavaScript event handlers simply by executing

```
java ScriptTest
```

For the Groovy handlers, use

```
java -classpath .:groovy/lib/\* ScriptTest groovy
```

Here, *groovy* is the directory into which you installed Groovy.

For the Renjin implementation of R, include the JAR files for Renjin Studio and the

http://www.rejoiceblog.com/

Renjin script engine on the classpath. Both are available at `www.renjin.org/downloads.html`.

To try out Scheme, download SISC Scheme from `http://sisc-scheme.org` and run

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.
```

where *sisc* is the installation directory for SISC Scheme and *jsr223-engines* is the directory that contains the engine adapters from `http://java.net/projects/scripting`.

This application demonstrates how to use scripting for Java GUI programming. One could go a step further and describe the GUI with an XML file, as you have seen in Chapter 3. Then our program would become an interpreter for GUIs that have visual presentation defined by XML and behavior defined by a scripting language. Note the similarity to a dynamic HTML page or a dynamic server-side scripting environment.

Listing 8.1 `script/ScriptTest.java`

```
package script;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import javax.script.*;
import javax.swing.*;
```

http://www.rejoiceblog.com/

```
10
11  /**
12   * @version 1.03 2018-05-01
13   * @author Cay Horstmann
14   */
15  public class ScriptTest
16  {
17     public static void main(String[] args)
18     {
19       EventQueue.invokeLater(() ->
20       {
21         try
22         {
23           var manager = new ScriptEngineManager();
24           String language;
25           if (args.length == 0)
26           {
27             System.out.println("Available factories: ");
28             for (ScriptEngineFactory factory : manager.get
29               System.out.println(factory.getEngineName());
```

http://www.rejoiceblog.com/

```
30
31            language = "nashorn";
32          }
33          else language = args[0];
34
35          final ScriptEngine engine = manager.getEngineByN
36          if (engine == null)
37          {
38            System.err.println("No engine for " + language
39            System.exit(1);
40          }
41
42          final String frameClassName = args.length < 2 ?
43
44          var frame = (JFrame) Class.forName(frameClassNam
45          InputStream in = frame.getClass().getResourceAsS
46          if (in != null) engine.eval(new InputStreamReade
47          var components = new HashMap<String, Component>(
48          getComponentBindings(frame, components);
```

http://www.rejoiceblog.com/

```
49          components.forEach((name, c) -> engine.put(name,
50
51          var events = new Properties();
52          in = frame.getClass().getResourceAsStream(langua
53          events.load(in);
54
55          for (Object e : events.keySet())
56          {
57            String[] s = ((String) e).split("\\.");
58            addListener(s[0], s[1], (String) events.get(e)
59          }
60          frame.setTitle("ScriptTest");
61          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
62          frame.setVisible(true);
63        }
64        catch (ReflectiveOperationException | IOException
65          | ScriptException | IntrospectionException ex)
66        {
67          ex.printStackTrace();
68        }
```

http://www.rejoiceblog.com/

```
      });
  }

/**
 * Gathers all named components in a container.
 * @param c the component
 * @param namedComponents a map into which to enter the c
 */
private static void getComponentBindings(Component c, Map
 {
   String name = c.getName();
   if (name != null) { namedComponents.put(name, c); }
   if (c instanceof Container)
   {
     for (Component child : ((Container) c).getComponents
       getComponentBindings(child, namedComponents);
   }
 }

```

http://www.rejoiceblog.com/

```
88  /**
89   * Adds a listener to an object whose listener method exe
90   * @param beanName the name of the bean to which the list
91   * @param eventName the name of the listener type, such a
92   * @param scriptCode the script code to be executed
93   * @param engine the engine that executes the code
94   * @param bindings the bindings for the execution
95   * @throws IntrospectionException
96   */
97  private static void addListener(String beanName, String e
98    final ScriptEngine engine, Map<String, Component> compo
99    throws ReflectiveOperationException, IntrospectionExcep
100  {
101     Object bean = components.get(beanName);
102     EventSetDescriptor descriptor = getEventSetDescriptor
103     if (descriptor == null) return;
104     descriptor.getAddListenerMethod().invoke(bean,
105       Proxy.newProxyInstance(null, new Class[] { descript
106         (proxy, method, args) ->
107               {
```

http://www.rejoiceblog.com/

```
                    engine.eval(scriptCode);
                    return null;
                }
            ));
    }

    private static EventSetDescriptor getEventSetDescriptor
        throws IntrospectionException
        {
            for (EventSetDescriptor descriptor : Introspector.
                .getEventSetDescriptors())
              if (descriptor.getName().equals(eventName)) retu
            return null;
        }
    }
```

Listing 8.2 `buttons1/ButtonFrame.java`

```
package buttons1;

```

http://www.rejoiceblog.com/

```
 3  import javax.swing.*;
 4
 5  /**
 6   * A frame with a button panel.
 7   * @version 1.00 2007-11-02
 8   * @author Cay Horstmann
 9   */
10  public class ButtonFrame extends JFrame
11  {
12     private static final int DEFAULT_WIDTH = 300;
13     private static final int DEFAULT_HEIGHT = 200;
14
15     private JPanel panel;
16     private JButton yellowButton;
17     private JButton blueButton;
18     private JButton redButton;
19
20     public ButtonFrame()
21     {
22         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

http://www.rejoiceblog.com/

```

        panel = new JPanel();
        panel.setName("panel");
        add(panel);

        yellowButton = new JButton("Yellow");
        yellowButton.setName("yellowButton");
        blueButton = new JButton("Blue");
        blueButton.setName("blueButton");
        redButton = new JButton("Red");
        redButton.setName("redButton");

        panel.add(yellowButton);
        panel.add(blueButton);
        panel.add(redButton);
    }
}
```

8.2 THE COMPILER API

There are quite a few tools that need to compile Java code. Obviously, development environments and programs that teach Java programming are among them, as well as

http://www.rejoiceblog.com/
testing and build automation tools. Another example is the processing of JavaServer Pages—web pages with embedded Java statements.

8.2.1 Invoking the Compiler

It is very easy to invoke the compiler. Here is a sample call:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

OutputStream outStream = . . .;

OutputStream errStream = . . .;

int result = compiler.run(null, outStream, errStream,

   "-sourcepath", "src", "Test.java");
```

A `result` value of `0` indicates successful compilation.

The compiler sends output and error messages to the provided streams. You can set these parameters to `null`, in which case `System.out` and `System.err` are used. The first parameter of the `run` method is an input stream. As the compiler takes no console input, you can always leave it as `null`. (The `run` method is inherited from a generic `Tool` interface, which allows for tools that read input.)

The remaining parameters of the `run` method are the arguments that you would pass to `javac` if you invoked it on the command line. These can be options or file names.

8.2.2 Launching a Compilation Task

You can have more control over the compilation process with a `CompilationTask` object. This can be useful if you want to supply source from string, capture class files in memory, or process the error and warning messages.

To obtain a `CompilationTask`, start with a `compiler` object as in the preceding section. Then call

```
JavaCompiler.CompilationTask task = compiler.getTask(
```

http://www.rejoiceblog.com/

```
    errorWriter, // Uses System.err if null
    fileManager, // Uses the standard file manager if null
    diagnostics, // Uses System.err if null
    options, // null if no options
    classes, // For annotation processing; null if none
    sources);
```

The last three arguments are `Iterable` instances. For example, a sequence of options might be specified as

```
Iterable<String> options = List.of("-d", "bin");
```

The `sources` parameter is an `Iterable` of `JavaFileObject` instances. If you want to compile disk files, get a `StandardJavaFileManager` and call its `getJavaFileObjects` method:

```
StandardJavaFileManager fileManager =
    compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> sources =
    fileManager.getJavaFileObjectsFromStrings(List.of("File1.j
JavaCompiler.CompilationTask task = compiler.getTask(
    null, null, null, options, null, sources);
```

http://www.rejoiceblog.com/

Note

The `classes` parameter is only used for annotation processing. In that case, you also need to call `task.processors(annotationProcessors)` with a list of `Processor` objects. See Chapter 11 for an example of annotation processing.

The `getTask` method returns the task object but does not yet start the compilation process. The `CompilationTask` class extends `Callable<Boolean>`. You can pass it to an `ExecutorService` for concurrent execution, or you can just make a synchronous call:

```
Boolean success = task.call();
```

8.2.3 Capturing Diagnostics

To listen to error messages, install a `DiagnosticListener`. The listener receives a `Diagnostic` object whenever the compiler reports a warning or error message. The `DiagnosticCollector` class implements this interface. It simply collects all diagnostics so that you can iterate through them after the compilation is complete.

```
DiagnosticCollector<JavaFileObject> collector = new Diagnosti

compiler.getTask(null, fileManager,

   collector, null, null, sources).call();

for (Diagnostic<? extends JavaFileObject> d : collector.getDi

   System.out.println(d);

}
```

A `Diagnostic` object contains information about the problem location (including file

http://www.rejoiceblog.com/
name, line number, and column number) as well as a human-readable description.

You can also install a `DiagnosticListener` to the standard file manager, in case you want to trap messages about missing files:

```
StandardJavaFileManager fileManager

   = compiler.getStandardFileManager(diagnostics, null, null)
```

8.2.4 Reading Source Files from Memory

If you generate source code on the fly, you can have it compiled from memory, without having to save files to disk. Use this class to hold the code:

```
public class StringSource extends SimpleJavaFileObject

{

   private String code;



   StringSource(String name, String code)

   {

     super(URI.create("string:///" + name.replace('.','/') +

        Kind.SOURCE);

     this.code = code;

   }



  public CharSequence getCharContent(boolean ignoreEncodingEr
```

http://www.rejoiceblog.com/

```
    {

     return code;

    }

  }
```

Then generate the code for your classes and give the compiler a list of `StringSource` objects.

```
List<StringSource> sources = List.of(

   new StringSource(className1, class1CodeString), . . .);

task = compiler.getTask(null, fileManager, diagnostics, null,
```

8.2.5 Writing Byte Codes to Memory

If you compile classes on the fly, there is no need to save the class files to disk. You can save them to memory and load them right away.

First, here is a class for holding the bytes:

```
public class ByteArrayClass extends SimpleJavaFileObject

{

   private ByteArrayOutputStream out;

   ByteArrayClass(String name) {

     super(URI.create("bytes:///" + name.replace('.','/') + "

         Kind.CLASS);
```

http://www.rejoiceblog.com/

```
    }



    public byte[] getCode() {

      return out.toByteArray();

    }



    public OutputStream openOutputStream() throws IOException

      out = new ByteArrayOutputStream();

      return out;

    }

  }
```

Next, you need to configure the file manager to use these classes for output:

```
  List<ByteArrayClass> classes = new ArrayList<>();

  StandardJavaFileManager stdFileManager

     = compiler.getStandardFileManager(null, null, null);

  JavaFileManager fileManager

     = new ForwardingJavaFileManager<JavaFileManager>(stdFileMa

        {

          public JavaFileObject getJavaFileForOutput(Location l
```

http://www.rejoiceblog.com/

```
            String className, Kind kind, FileObject sibling)

              throws IOException

          {

            if (kind == Kind.CLASS)

            {

             ByteArrayClass outfile = new ByteArrayClass(classN

             classes.add(outfile);

             return outfile;

            }

            else

               return super.getJavaFileForOutput(

                 location, className, kind, sibling);

          }

        };
```

To load the classes, you need a class loader (see Chapter 10):

```
public class ByteArrayClassLoader extends ClassLoader

{

   private Iterable<ByteArrayClass> classes;
```

http://www.rejoiceblog.com/

```
    public ByteArrayClassLoader(Iterable<ByteArrayClass> class

    {

      this.classes = classes;

    }



    public Class<?> findClass(String name) throws ClassNotFoun

    {

      for (ByteArrayClass cl : classes)

      {

         if (cl.getName().equals("/" + name.replace('.','/') +

         {

            byte[] bytes = cl.getCode();

            return defineClass(name, bytes, 0, bytes.length);

         }

      }

      throw new ClassNotFoundException(name);

    }

  }
```

http://www.rejoiceblog.com/

After compilation has finished, call the `Class.forName` method with that class loader:

```
ByteArrayClassLoader loader = new ByteArrayClassLoader(classe

Class<?> cl = Class.forName(className, true, loader);
```

8.2.6 An Example: Dynamic Java Code Generation

In the JSP technology for dynamic web pages, you can mix HTML with snippets of Java code, for example

```
<p>The current date and time is <b><%= new java.util.Date() %
```

The JSP engine dynamically compiles the Java code into a servlet. In our sample application, we use a simpler example and generate dynamic Swing code instead. The idea is that you use a GUI builder to lay out the components in a frame and specify the behavior of the components in an external file. Listing 8.4 shows a very simple example of a frame class, and Listing 8.5 shows the code for the button actions. Note that the constructor of the frame class calls an abstract method `addEventHandlers`. Our code generator will produce a subclass that implements the `addEventHandlers` method, adding an action listener for each line in the `action.properties` file. (We leave it as the proverbial exercise to the reader to extend the code generation to other event types.)

We place the subclass into a package with the name `x`, which we hope is not used anywhere else in the program. The generated code has the form

```
package x;

public class Frame extends SuperclassName

{

   protected void addEventHandlers()

   {
```

http://www.rejoiceblog.com/

```
        componentName1.addActionListener(event -> {
            code for event handler1
          });
        // repeat for the other event handlers . . .
      }
  }
```

The `buildSource` method in the program of Listing 8.3 builds up this code and places it into a `StringSource` object. That object is passed to the Java compiler.

As described in the preceding section, we use a `ForwardingJavaFileManager` that constructs a `ByteArrayClass` object for every compiled class. These objects capture the class files generated when the `x.Frame` class is compiled. The method adds each file object to a list before returning it so that we can locate the bytecodes later.

After compilation, we use the class loader from the preceding section to loads the classes stored in this list. Then, we construct and display the application's frame class.

```
var loader = new ByteArrayClassLoader(classFileObjects);
var frame = (JFrame) loader.loadClass("x.Frame").getConstruct
frame.setVisible(true);
```

When you click the buttons, the background color changes in the usual way. To see that the actions are dynamically compiled, change one of the lines in `action.properties`, for example, like this:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW); yell
```

http://www.rejoiceblog.com/

Run the program again. Now the Yellow button is disabled after you click it. Also, have a look at the code directories. You will not find any source or class files for the classes in the `x` package. This example demonstrates how you can use dynamic compilation with in-memory source and class files.

Listing 8.3 `compiler/CompilerTest.java`

```
package compiler;

import java.awt.*;
import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.List;

import javax.swing.*;
import javax.tools.*;
import javax.tools.JavaFileObject.*;

/**
 * @version 1.10 2018-05-01
 * @author Cay Horstmann
 */
```

http://www.rejoiceblog.com/

```
17  public class CompilerTest
18  {
19     public static void main(final String[] args)
20           throws IOException, ReflectiveOperationException
21     {
22       JavaCompiler compiler = ToolProvider.getSystemJavaCo
23
24       var classFileObjects = new ArrayList<ByteArrayClass>
25
26       var diagnostics = new DiagnosticCollector<JavaFileOb
27
28       JavaFileManager fileManager = compiler.getStandardFi
29       fileManager = new ForwardingJavaFileManager<JavaFile
30         {
31           public JavaFileObject getJavaFileForOutput(Locat
32               Kind kind, FileObject sibling) throws IOExce
33           {
34             if (kind == Kind.CLASS)
35             {
36               var fileObject = new ByteArrayClass(classNam
```

http://www.rejoiceblog.com/

```
                classFileObjects.add(fileObject);
                return fileObject;
            }
            else return super.getJavaFileForOutput(locatio
        }
    };


String frameClassName = args.length == 0 ? "buttons2.But
//compiler.run(null, null, null, frameClassName.replace(

StandardJavaFileManager fileManager2 =
      compiler.getStandardFileManager(null, null, null);
var sources = new ArrayList<JavaFileObject>();
for (JavaFileObject o : fileManager2.getJavaFileObjectsF
    sources.add(o);

JavaFileObject source = buildSource(frameClassName);
JavaCompiler.CompilationTask task = compiler.getTask(nul
```

http://www.rejoiceblog.com/

```
        null, List.of(source));
Boolean result = task.call();

for (Diagnostic<? extends JavaFileObject> d : diagnostic
  System.out.println(d.getKind() + ": " + d.getMessage(n
fileManager.close();
if (!result)
{
   System.out.println("Compilation failed.");
     System.exit(1);
}

var loader = new ByteArrayClassLoader(classFileObjects);
var frame = (JFrame) loader.loadClass("x.Frame").getCons

EventQueue.invokeLater(() ->
  {
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
    frame.setTitle("CompilerTest");
    frame.setVisible(true);
```

http://www.rejoiceblog.com/

```
    });
}

/*
 * Builds the source for the subclass that implements the
 * @return a file object containing the source in a strin
 */
static JavaFileObject buildSource(String superclassName)
    throws IOException, ClassNotFoundException
{
    var builder = new StringBuilder();
    builder.append("package x;\n\n");
    builder.append("public class Frame extends " + supercl
    builder.append("protected void addEventHandlers() {\n"
    var props = new Properties();
    props.load(Files.newInputStream(
          Paths.get(superclassName.replace(".", "/")).getP
    for (Map.Entry<Object, Object> e : props.entrySet())
    {
```

http://www.rejoiceblog.com/

```
      var beanName = (String) e.getKey();
      var eventCode = (String) e.getValue();
      builder.append(beanName + ".addActionListener(event
      builder.append(eventCode);
      builder.append("\n} );\n");
    }
    builder.append("} }\n");
    return new StringSource("x.Frame", builder.toString())
  }
}
```

Listing 8.4 `buttons2/ButtonFrame.java`

```
package buttons2;
import javax.swing.*;

/**
 * A frame with a button panel.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
```

http://www.rejoiceblog.com/

```
 9  public abstract class ButtonFrame extends JFrame
10  {
11     public static final int DEFAULT_WIDTH = 300;
12     public static final int DEFAULT_HEIGHT = 200;
13
14     protected JPanel panel;
15     protected JButton yellowButton;
16     protected JButton blueButton;
17     protected JButton redButton;
18
19     protected abstract void addEventHandlers();
20
21     public ButtonFrame()
22     {
23       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24
25       panel = new JPanel();
26       add(panel);
27
```

http://www.rejoiceblog.com/

```
        yellowButton = new JButton("Yellow");

        blueButton = new JButton("Blue");

        redButton = new JButton("Red");


        panel.add(yellowButton);

        panel.add(blueButton);

        panel.add(redButton);


        addEventHandlers();

    }

}
```

Listing 8.5 `buttons2/action.properties`

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);

 blueButton=panel.setBackground(java.awt.Color.BLUE);
```

javax.tools.Tool 6

- `int run(InputStream in, OutputStream out, OutputStream err, String... arguments)`

runs the tool with the given input, output, and error streams and the given arguments. Returns `0` for success, a nonzero value for failure.

http://www.rejoiceblog.com/

javax.tools.JavaCompiler 6

- `StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)`

gets the standard file manager for this compiler. You can supply `null` for default error reporting, locale, and character set.

- `JavaCompiler.CompilationTask getTask(Writer out, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)`

gets a compilation task that, when called, will compile the given source files. See the discussion in the preceding section for details.

javax.tools.StandardJavaFileManager 6

- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)`

- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)`

translates a sequence of file names or files into a sequence of `JavaFileObject` instances.

javax.tools.JavaCompiler.CompilationTask 6

- `Boolean call()`

performs the compilation task.

http://www.rejoiceblog.com/

javax.tools.DiagnosticCollector<S> 6

- `DiagnosticCollector()`

constructs an empty collector.

- `List<Diagnostic<? extends S>> getDiagnostics()`

gets the collected diagnostics.

javax.tools.Diagnostic<S> 6

- `S getSource()`

gets the source object associated with this diagnostic.

- `Diagnostic.Kind getKind()`

gets the type of this diagnostic—one of `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE`, or `OTHER`.

- `String getMessage(Locale locale)`

gets the message describing the issue raised in this diagnostic. Pass `null` for the default locale.

- `long getLineNumber()`

- `long getColumnNumber()`

gets the position of the issue raised in this diagnostic.

javax.tools.SimpleJavaFileObject 6

- `CharSequence getCharContent(boolean ignoreEncodingErrors)`

Override this method for a file object that represents a source file and produces the source code.

http://www.rejoiceblog.com/

• `OutputStream openOutputStream()`

Override this method for a file object that represents a class file and produces a stream to which the bytecodes can be written.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

• `protected ForwardingJavaFileManager(M fileManager)`

constructs a `JavaFileManager` that delegates all calls to the given file manager.

• `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`

intercept this call if you want to substitute a file object for writing class files; `kind` is one of `SOURCE`, `CLASS`, `HTML`, or `OTHER`.

8.3 USING ANNOTATIONS

Annotations are tags that you insert into your source code so that some tool can process them. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.

Annotations do not change the way in which your programs are compiled. The Java compiler generates the same virtual machine instructions with or without the annotations.

To benefit from annotations, you need to select a *processing tool*. Use annotations that your processing tool understands, then apply the processing tool to your code.

There is a wide range of uses for annotations, and that generality can be confusing at first. Here are some uses for annotations:

• Automatic generation of auxiliary files, such as deployment descriptors or bean information classes

• Automatic generation of code for testing, logging, transaction semantics, and so on

http://www.rejoiceblog.com/

8.3.1 An Introduction into Annotations

We'll start our discussion of annotations with the basic concepts and put them to use in a concrete example: We will mark methods as event listeners for AWT components, and show you an annotation processor that analyzes the annotations and hooks up the listeners. We'll then discuss the syntax rules in detail and finish the chapter with two advanced examples of annotation processing. One of them processes source-level annotations, the other uses the Apache Bytecode Engineering Library to process class files, injecting additional bytecodes into annotated methods.

Here is an example of a simple annotation:

```
public class MyClass

{

   . . .

   @Test public void checkRandomInsertions()

}
```

The annotation `@Test` annotates the `checkRandomInsertions` method.

In Java, an annotation is used like a *modifier* and is placed before the annotated item *without a semicolon*. (A modifier is a keyword such as `public` or `static`.) The name of each annotation is preceded by an `@` symbol, similar to Javadoc comments. However, Javadoc comments occur inside `/** . . . */` delimiters, whereas annotations are part of the code.

By itself, the `@Test` annotation does not do anything. It needs a tool to be useful. For example, the JUnit 4 testing tool (available at `http://junit.org`) calls all methods that are labeled `@Test` when testing a class. Another tool might remove all test methods from a class file so they are not shipped with the program after it has been tested.

Annotations can be defined to have *elements*, such as

```
@Test(timeout="10000")
```

http://www.rejoiceblog.com/

These elements can be processed by the tools that read the annotations. Other forms of elements are possible; we'll discuss them later in this chapter.

Besides methods, you can annotate classes, fields, and local variables—an annotation can be anywhere you could put a modifier such as `public` or `static`. In addition, as you will see in Section 8.4, "Annotation Syntax," on p. 471, you can annotate packages, parameter variables, type parameters, and type uses.

Each annotation must be defined by an *annotation interface*. The methods of the interface correspond to the elements of the annotation. For example, the JUnit `Test` annotation is defined by the following interface:

```
@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Test

{

   long timeout() default 0L;

   . . .

}
```

The `@interface` declaration creates an actual Java interface. Tools that process annotations receive objects that implement the annotation interface. A tool would call the `timeout` method to retrieve the `timeout` element of a particular `Test` annotation.

The `Target` and `Retention` annotations are *meta-annotations*. They annotate the `Test` annotation, marking it as an annotation that can be applied to methods only and is retained when the class file is loaded into the virtual machine. We'll discuss these in detail in Section 8.5.3, "Meta-Annotations," on p. 481.

You have now seen the basic concepts of program metadata and annotations. In the next section, we'll walk through a concrete example of annotation processing.

http://www.rejoiceblog.com/

Note

For a compelling use of annotations, check out `JCommander` (`http://jcommander.org`) and picocli (`http://picocli.info`). These libraries use annotations for the processing of command-line parameters.

8.3.2 An Example: Annotating Event Handlers

One of the more boring tasks in user interface programming is the wiring of listeners to event sources. Many listeners are of the form

```
myButton.addActionListener(() -> doSomething());
```

In this section, we'll design an annotation to reverse the wiring. The annotation, defined in Listing 8.8, is used as follows:

```
@ActionListenerFor(source="myButton") void doSomething() { .
```

The programmer no longer has to make calls to `addActionListener`. Instead, each method is tagged with an annotation. Listing 8.7 shows the `ButtonFrame` class from Volume I, Chapter 11, reimplemented with these annotations.

We also need to define an annotation interface. The code is in Listing 8.8.

Of course, the annotations don't do anything by themselves. They sit in the source file. The compiler places them in the class file, and the virtual machine loads them. We now need a mechanism to analyze them and install action listeners. That is the job of the `ActionListenerInstaller` class. The `ButtonFrame` constructor calls

```
ActionListenerInstaller.processAnnotations(this);
```

The static `processAnnotations` method enumerates all methods of the object it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

http://www.rejoiceblog.com/

```
Class<?> cl = obj.getClass();

for (Method m : cl.getDeclaredMethods())

{

   ActionListenerFor a = m.getAnnotation(ActionListenerFor.cl

   if (a != null) . . .

}
```

Here, we use the `getAnnotation` method defined in the `AnnotatedElement` interface. The classes `Method`, `Constructor`, `Field`, `Class`, and `Package` implement this interface.

The name of the source field is stored in the annotation object. We retrieve it by calling the `source` method, and then look up the matching field.

```
String fieldName = a.source();

Field f = cl.getDeclaredField(fieldName);
```

This shows a limitation of our annotation. The source element must be the name of a field. It cannot be a local variable.

The remainder of the code is rather technical. For each annotated method, we construct a proxy object, implementing the `ActionListener` interface, with an

`actionPerformed` method that calls the annotated method. (For more information about proxies, see Volume I, Chapter 6.) The details are not important. The key observation is that the functionality of the annotations was established by the `processAnnotations` method.

Figure 8.1 shows how annotations are handled in this example.

http://www.rejoiceblog.com/

Figure 8.1 Processing annotations at runtime

In this example, the annotations were processed at runtime. It is also possible to process them at the source level: A source code generator would then produce the code for adding the listeners. Alternatively, the annotations can be processed at the bytecode level: A bytecode editor could inject the calls to `addActionListener` into the frame constructor. This sounds complex, but libraries are available to make this task relatively straightforward. You can see an example in Section 8.7, "Bytecode Engineering," on p. 490.

Our example was not intended as a serious tool for user interface programmers. A utility method for adding a listener could be just as convenient for the programmer as the annotation. (In fact, the `java.beans.EventHandler` class tries to do just that. You could make the class truly useful by supplying a method that adds the event handler instead of just constructing it.)

However, this example shows the mechanics of annotating a program and of analyzing the annotations. Having seen a concrete example, you are now more prepared (we hope) for the following sections that describe the annotation syntax in complete detail.

Listing 8.6 `runtimeAnnotations/ActionListenerInstaller.java`

```
package runtimeAnnotations;

import java.awt.event.*;
import java.lang.reflect.*;
```

http://www.rejoiceblog.com/

```
 5
 6  /**
 7   * @version 1.00 2004-08-17
 8   * @author Cay Horstmann
 9   */
10  public class ActionListenerInstaller
11  {
12  /**
13   * Processes all ActionListenerFor annotations in the giv
14   * @param obj an object whose methods may have ActionList
15   */
16  public static void processAnnotations(Object obj)
17  {
18     try
19     {
20       Class<?> cl = obj.getClass();
21       for (Method m : cl.getDeclaredMethods())
22       {
23         ActionListenerFor a = m.getAnnotation(ActionListen
```

http://www.rejoiceblog.com/

```
        if (a != null)
        {
            Field f = cl.getDeclaredField(a.source());
            f.setAccessible(true);
            addListener(f.get(obj), obj, m);
        }
      }
    }
    catch (ReflectiveOperationException e)
    {
      e.printStackTrace();
    }
}

/**
 * Adds an action listener that calls a given method.
 * @param source the event source to which an action list
 * @param param the implicit parameter of the method that
 * @param m the method that the listener calls
 */
```

http://www.rejoiceblog.com/

```
public static void addListener(Object source, final Objec
    throws ReflectiveOperationException
{
  var handler = new InvocationHandler()
      {
        public Object invoke(Object proxy, Method mm, Obje
        {
            return m.invoke(param);
        }
      };

  Object listener = Proxy.newProxyInstance(null,
      new Class[] { java.awt.event.ActionListener.class }
  Method adder = source.getClass().getMethod("addActionLi
  adder.invoke(source, listener);
}
}
```

Listing 8.7 `buttons3/ButtonFrame.java`

http://www.rejoiceblog.com/

```
1  package buttons3;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import runtimeAnnotations.*;
6
7  /**
8   * A frame with a button panel.
9   * @version 1.00 2004-08-17
10  * @author Cay Horstmann
11  */
12 public class ButtonFrame extends JFrame
13 {
14    private static final int DEFAULT_WIDTH = 300;
15    private static final int DEFAULT_HEIGHT = 200;
16
17    private JPanel panel;
18    private JButton yellowButton;
19    private JButton blueButton;
```

http://www.rejoiceblog.com/

```
20      private JButton redButton;
21
22      public ButtonFrame()
23      {
24        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
25
26        panel = new JPanel();
27        add(panel);
28
29        yellowButton = new JButton("Yellow");
30        blueButton = new JButton("Blue");
31        redButton = new JButton("Red");
32
33        panel.add(yellowButton);
34        panel.add(blueButton);
35        panel.add(redButton);
36
37        ActionListenerInstaller.processAnnotations(this);
38      }
39
```

http://www.rejoiceblog.com/

```
    @ActionListenerFor(source = "yellowButton")
    public void yellowBackground()
    {
      panel.setBackground(Color.YELLOW);
    }

    @ActionListenerFor(source = "blueButton")
    public void blueBackground()
    {
      panel.setBackground(Color.BLUE);
    }

    @ActionListenerFor(source = "redButton")
    public void redBackground()
    {
      panel.setBackground(Color.RED);
    }
}
```

Listing 8.8 `runtimeAnnotations/ActionListenerFor.java`

http://www.rejoiceblog.com/

```
1  package runtimeAnnotations;
2
3  import java.lang.annotation.*;
4
5  /**
6   * @version 1.00 2004-08-17
7   * @author Cay Horstmann
8   */
9
10 @Target(ElementType.METHOD)
11 @Retention(RetentionPolicy.RUNTIME)
12 public @interface ActionListenerFor
13 {
14    String source();
15 }
```

java.lang.reflect.AnnotatedElement 5.0

- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`

returns `true` if this item has an annotation of the given type.

http://www.rejoiceblog.com/

- `<T extends Annotation> T getAnnotation(Class<T> annotationType)`

gets the annotation of the given type, or `null` if this item has no such annotation.

- `<T extends Annotation> T[] getAnnotationsByType(Class<T> annotationType)` **8**

gets all annotations of a repeatable annotation type (see Section 8.5.3, "Meta-Annotations," on p. 481), or an array of length 0.

- `Annotation[] getAnnotations()`

gets all annotations present for this item, including inherited annotations. If no annotations are present, an array of length 0 is returned.

- `Annotation[] getDeclaredAnnotations()`

gets all annotations declared for this item, excluding inherited annotations. If no annotations are present, an array of length 0 is returned.

8.4 ANNOTATION SYNTAX

In the following sections, we cover everything you need to know about the annotation syntax.

8.4.1 Annotation Interfaces

An annotation is defined by an annotation interface:

```
modifiers @interface AnnotationName
{
    elementDeclaration1
    elementDeclaration2
    . . .
```

http://www.rejoiceblog.com/

```
}
```

Each element declaration has the form

type elementName`()`*;*

or

type elementName`() default` *value;*

For example, the following annotation has two elements, `assignedTo` and `severity`:

```
public @interface BugReport

{

   String assignedTo() default "[none]";

   int severity();

}
```

All annotation interfaces implicitly extend the `java.lang.annotation.Annotation` interface. That interface is a regular interface, *not* an annotation interface. See the API notes at the end of this section for the methods provided by this interface. You cannot extend annotation interfaces. In other words, all annotation interfaces directly extend `java.lang.annotation.Annotation`. You never supply classes that implement annotation interfaces.

The methods of an annotation interface have no parameters and no `throws` clauses. They cannot be `default` or `static` methods, and they cannot have type parameters.

The type of an annotation element is one of the following:

- A primitive type (`int`, `short`, `long`, `byte`, `char`, `double`, `float`, or `boolean`)

- `String`

http://www.rejoiceblog.com/

- `Class` (with an optional type parameter such as `Class<? extends MyClass>`)
- An `enum` type
- An annotation type
- An array of the preceding types (an array of arrays is not a legal element type)

Here are examples of valid element declarations:

```
public @interface BugReport
{
   enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
   boolean showStopper() default false;
   String assignedTo() default "[none]";
   Class<?> testCase() default Void.class;
   Status status() default Status.UNCONFIRMED;
   Reference ref() default @Reference(); // an annotation typ
   String[] reportedBy();
}
```

java.lang.annotation.Annotation 5.0

- `Class<? extends Annotation> annotationType()`

returns the `Class` object that represents the annotation interface of this annotation object. Note that calling `getClass` on an annotation object would return the actual class, not the interface.

http://www.rejoiceblog.com/

- `boolean equals(Object other)`

returns `true` if `other` is an object that implements the same annotation interface as this annotation object and if all elements of this object and `other` are equal.

- `int hashCode()`

returns a hash code, compatible with the `equals` method, derived from the name of the annotation interface and the element values.

- `String toString()`

returns a string representation that contains the annotation interface name and the element values; for example, `@BugReport(assignedTo=[none], severity=0)`.

8.4.2 Annotations

Each annotation has the format

```
@AnnotationName(elementName1=value1, elementName2=value2, . . .)
```

For example,

```
@BugReport(assignedTo="Harry", severity=10)
```

The order of the elements does not matter. The annotation

```
@BugReport(severity=10, assignedTo="Harry")
```

is identical to the preceding one.

The default value of the declaration is used if an element value is not specified. For example, consider the annotation

```
@BugReport(severity=10)
```

http://www.rejoiceblog.com/

The value of the `assignedTo` element is the string `"[none]"`.

> **Caution**
>
> Defaults are not stored with the annotation; instead, they are dynamically computed. For example, if you change the default for the `assignedTo` element to `"[]"` and recompile the `BugReport` interface, the annotation `@BugReport(severity=10)` will use the new default, even in class files that have been compiled before the default changed.

Two special shortcuts can simplify annotations.

If no elements are specified, either because the annotation doesn't have any or because all of them use the default value, you don't need to use parentheses. For example,

```
@BugReport
```

is the same as

```
@BugReport(assignedTo="[none]", severity=0)
```

Such an annotation is called a *marker annotation*.

The other shortcut is the *single value annotation*. If an element has the special name `value` and no other element is specified, you can omit the element name and the = symbol. For example, had we defined the `ActionListenerFor` annotation interface of the preceding section as

```
public @interface ActionListenerFor

{

   String value();
```

http://www.rejoiceblog.com/

```
}
```

then the annotations could be written as

```
@ActionListenerFor("yellowButton")
```

instead of

```
@ActionListenerFor(value="yellowButton")
```

An item can have multiple annotations:

```
@Test

@BugReport(showStopper=true, reportedBy="Joe")

public void checkRandomInsertions()
```

If the author of an annotation declared it to be repeatable, you can repeat the same annotation multiple times:

```
@BugReport(showStopper=true, reportedBy="Joe")

@BugReport(reportedBy={"Harry", "Carl"})

public void checkRandomInsertions()
```

Note

Since annotations are evaluated by the compiler, all element values must be compile-time constants. For example,

```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTest
```

http://www.rejoiceblog.com/

```
status=BugReport.Status.CONFIRMED, . . .)
```

Caution

An annotation element can never be set to `null`. Not even a `default` of `null` is permissible. This can be rather inconvenient in practice. You will need to find other defaults, such as `""` or `Void.class`.

If an element value is an array, enclose its values in braces:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

You can omit the braces if the element has a single value:

```
@BugReport(. . ., reportedBy="Joe") // OK, same as {"Joe"}
```

Since an annotation element can be another annotation, you can build arbitrarily complex annotations. For example,

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```

Note

It is an error to introduce circular dependencies in annotations. For example, `BugReport` has an element of the annotation type `Reference`, therefore `Reference` cannot have an element of type `BugReport`.

http://www.rejoiceblog.com/

8.4.3 Annotating Declarations

There are many places where annotations can occur. They fall into two categories: *declarations* and *type uses*. Declaration annotations can appear at the declarations of

- Packages
- Classes (including `enum`)
- Interfaces (including annotation interfaces)
- Methods
- Constructors
- Instance fields (including `enum` constants)
- Local variables
- Parameter variables
- Type parameters

For classes and interfaces, put the annotations before the `class` or `interface` keyword:

```
@Entity public class User { . . . }
```

For variables, put them before the type:

```
@SuppressWarnings("unchecked") List<User> users = . . .;

public User getUser(@Param("id") String userId)
```

A type parameter in a generic class or method can be annotated like this:

```
public class Cache<@Immutable V> { . . . }
```

A package is annotated in a file `package-info.java` that contains only the package

http://www.rejoiceblog.com/

statement preceded by annotations.

```
/**
    Package-level Javadoc
*/
@GPL(version="3")
package com.horstmann.corejava;
import org.gnu.GPL;
```

Note

Annotations for local variables can only be processed at the source level. Class files do not describe local variables. Therefore, all local variable annotations are discarded when a class is compiled. Similarly, annotations for packages are not retained beyond the source level.

8.4.4 Annotating Type Uses

A declaration annotation provides some information about the item being declared. For example, in the declaration

```
public User getUser(@NonNull String userId)
```

it is asserted that the `userId` parameter is not null.

Note

The `@NonNull` annotation is a part of the Checker Framework

http://www.rejoiceblog.com/

(http://types.cs.washington.edu/checker-framework). With that framework, you can include assertions in your program, such that a parameter is non-null or that a `String` contains a regular expression. A static analysis tool then checks whether the assertions are valid in a given body of source code.

Now suppose we have a parameter of type `List<String>`, and we want to express that all of the strings are non-`null`. That is where type use annotations come in. Place the annotation before the type argument: `List<@NonNull String>`.

Type use annotations can appear in the following places:

• With generic type arguments: `List<@NonNull String>`, `Comparator.<@NonNull String> reverseOrder()`.

• In any position of an array: `@NonNull String[][] words` (`words[i][j]` is not `null`), `String @NonNull [][] words` (`words` is not `null`), `String[] @NonNull [] words` (`words[i]` is not `null`).

• With superclasses and implemented interfaces: `class Warning extends @Localized Message`.

• With constructor invocations: `new @Localized String(. . .)`.

• With casts and `instanceof` checks: `(@Localized String) text`, `if (text instanceof @Localized String)`. (The annotations are only for use by external tools. They have no effect on the behavior of a cast or an `instanceof` check.)

• With exception specifications: `public String read() throws @Localized IOException`.

• With wildcards and type bounds: `List<@Localized ? extends Message>`, `List<? extends @Localized Message>`.

• With method and constructor references: `@Localized Message::getText`.

There are a few type positions that cannot be annotated:

```
@NonNull String.class // ERROR: Cannot annotate class literal
```

http://www.rejoiceblog.com/

```
import java.lang.@NonNull String; // ERROR: Cannot annotate i
```

You can place annotations before or after other modifiers such as `private` and `static`. It is customary (but not required) to put type use annotations after other modifiers, and declaration annotations before other modifiers. For example,

```
private @NonNull String text; // Annotates the type use

@Id private String userId; // Annotates the variable
```

Note

An annotation author needs to specify where a particular annotation can appear. If an annotation is permissible both for a variable and a type use, and it is used in a variable declaration, then both the variable and the type use are annotated. For example, consider

```
public User getUser(@NonNull String userId)
```

If `@NonNull` can apply both to parameters and to type uses, the `userId` parameter is annotated, and the parameter type is `@NonNull String`.

8.4.5 Annotating this

Suppose you want to annotate parameters that are not being mutated by a method.

```
public class Point

{

   public boolean equals(@ReadOnly Object other) { . . . }

}
```

http://www.rejoiceblog.com/

Then a tool that processes this annotation would, upon seeing a call

```
p.equals(q)
```

reason that `q` has not been changed.

But what about `p`?

When the method is called, the `this` variable is bound to `p`. But `this` is never declared, so you cannot annotate it.

Actually, you can declare it, with a rarely used syntax variant, just so that you can add an annotation:

```
public class Point

{

   public boolean equals(@ReadOnly Point this, @ReadOnly Obje

}
```

The first parameter is called the *receiver parameter*. It must be named `this`. Its type is the class that is being constructed.

Note

You can provide a receiver parameter only for methods, not for constructors. Conceptually, the `this` reference in a constructor is not an object of the given type until the constructor has completed. Instead, an annotation placed on the constructor describes a property of the constructed object.

http://www.rejoiceblog.com/

A different hidden parameter is passed to the constructor of an inner class, namely the reference to the enclosing class object. You can make that parameter explicit as well:

```
public class Sequence
{
   private int from;
   private int to;


   class Iterator implements java.util.Iterator<Integer>
   {
     private int current;
       public Iterator(@ReadOnly Sequence Sequence.this)
     {
        this.current = Sequence.this.from;
     }
     . . .
   }
   . . .
}
```

The parameter must be named just like when you refer to it, *EnclosingClass*`.this`, and its type is the enclosing class.

http://www.rejoiceblog.com/

8.5 STANDARD ANNOTATIONS

A number of annotation interfaces are defined in the `java.lang`, `java.lang.annotation`, and `javax.annotation` packages. Four of them are meta-annotations that describe the behavior of annotation interfaces. The others are regular annotations that you can use to annotate items in your source code. Table 8.2 shows these annotations. We'll discuss them in detail in the following two sections.

Table 8.2 The Standard Annotations

Annotation Interface	Applicable To	Purpose
`Deprecated`	All	Marks item as deprecated.
`SuppressWarnings`	All but packages and annotations	Suppresses warnings of the given type.
`SafeVarargs`	Methods and constructors	Asserts that the varargs parameter is safe to use.
`Override`	Methods	Checks that this method overrides a superclass method.
`FunctionalInterface`	Interfaces	Marks an interface as functional (with a single abstract method).
`PostConstruct` `PreDestroy`	Methods	The marked method should be invoked immediately after construction or before removal.
`Resource`	Classes, interfaces, methods, fields	On a class or interface, marks it as a resource to be used elsewhere. On a method or field, marks it for "injection."
`Resources`	Classes, interfaces	Specifies an array of resources.
`Generated`	All	Marks an item as source code that has been generated by a tool.
`Target`	Annotations	Specifies the items to which this annotation can be applied.
`Retention`	Annotations	Specifies how long this annotation is retained.
`Documented`	Annotations	Specifies that this annotation should be included in the documentation of annotated items.
`Inherited`	Annotations	Specifies that this annotation, when applied to a class, is automatically inherited by its subclasses.
`Repeatable`	Annotations	Specifies that this annotation can be applied multiple times to the same item.

http://www.rejoiceblog.com/

8.5.1 Annotations for Compilation

The `@Deprecated` annotation can be attached to any items whose use is no longer encouraged. The compiler will warn when you use a deprecated item. This annotation has the same role as the `@deprecated` Javadoc tag. However, the annotation persists until runtime.

> **Note**
>
> The `jdeprscan` utility that is part of the JDK can scan a set of JAR files for deprecated elements.

The `@SuppressWarnings` annotation tells the compiler to suppress warnings of a particular type, for example,

```
@SuppressWarnings("unchecked")
```

The `@Override` annotation applies only to methods. The compiler checks that a method with this annotation really overrides a method from the superclass. For example, if you declare

```
public MyClass

{

   @Override public boolean equals(MyClass other);

   . . .

}
```

then the compiler will report an error. After all, the `equals` method does *not* override the `equals` method of the `Object` class because that method has a parameter of type `Object`, not `MyClass`.

http://www.rejoiceblog.com/

The `@Generated` annotation is intended for use by code generator tools. Any generated source code can be annotated to differentiate it from programmer-provided code. For example, a code editor can hide the generated code, or a code generator can remove older versions of generated code. Each annotation must contain a unique identifier for the code generator. A date string (in ISO 8601 format) and a comment string are optional. For example,

```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56
```

8.5.2 Annotations for Managing Resources

The `@PostConstruct` and `@PreDestroy` annotations are used in environments that control the lifecycle of objects, such as web containers and application servers. Methods tagged with these annotations should be invoked immediately after an object has been constructed or immediately before it is being removed.

The `@Resource` annotation is intended for resource injection. For example, consider a web application that accesses a database. Of course, the database access information should not be hardwired into the web application. Instead, the web container has some user interface for setting connection parameters and a JNDI name for a data source. In the web application, you can reference the data source like this:

```
@Resource(name="jdbc/mydb")

private DataSource source;
```

When an object containing this field is constructed, the container “injects” a reference to the data source.

8.5.3 Meta-Annotations

The `@Target` meta-annotation is applied to an annotation, restricting the items to which the annotation applies. For example,

```
@Target({ElementType.TYPE, ElementType.METHOD})

public @interface BugReport
```

http://www.rejoiceblog.com/

Table 8.3 shows all possible values. They belong to the enumerated type `ElementType`. You can specify any number of element types, enclosed in braces.

Table 8.3 Element Types for the `@Target` Annotation

Element Type	Annotation Applies To
ANNOTATION_TYPE	Annotation type declarations
PACKAGE	Packages
TYPE	Classes (including `enum`) and interfaces (including annotation types)
METHOD	Methods
CONSTRUCTOR	Constructors
FIELD	Fields (including `enum` constants)
PARAMETER	Method or constructor parameters
LOCAL_VARIABLE	Local variables
TYPE_PARAMETER	Type parameters
TYPE_USE	Uses of a type

An annotation without an `@Target` restriction can be applied to any item. The compiler checks that you apply an annotation only to a permitted item. For example, if you apply `@BugReport` to a field, a compile-time error results.

The `@Retention` meta-annotation specifies how long an annotation is retained. You can specify at most one of the values in Table 8.4. The default is `RetentionPolicy.CLASS`.

Table 8.4 Retention Policies for the `@Retention` Annotation

Retention Policy	Description
SOURCE	Annotations are not included in class files.
CLASS	Annotations are included in class files, but the virtual machine need not load them.
RUNTIME	Annotations are included in class files and loaded by the virtual machine. They are available through the reflection API.

In Listing 8.8, the `@ActionListenerFor` annotation was declared with `RetentionPolicy.RUNTIME` because we used reflection to process annotations. In

http://www.rejoiceblog.com/

the following two sections, you will see examples of processing annotations at the source and class file levels.

The `@Documented` meta-annotation gives a hint to documentation tools such as Javadoc. Documented annotations should be treated just like other modifiers, such as `protected` or `static`, for documentation purposes. The use of other annotations is not included in the documentation. For example, suppose we declare `@ActionListenerFor` as a documented annotation:

```
@Documented

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface ActionListenerFor
```

Now the documentation of each annotated method contains the annotation, as shown in Figure 8.2.

Figure 8.2 Documented annotations

http://www.rejoiceblog.com/

If an annotation is transient (such as `@BugReport`), you should probably not document its use.

> **Note**
>
> It is legal to apply an annotation to itself. For example, the `@Documented` annotation is itself annotated as `@Documented`. Therefore, the Javadoc documentation for annotations shows whether they are documented.

The `@Inherited` meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. This makes it easy to create annotations that work as marker interfaces, such as `Serializable`.

In fact, an annotation `@Serializable` would be more appropriate than the `Serializable` marker interface with no methods. A class is serializable because there is runtime support for reading and writing its fields, not because of any principles of object-oriented design. An annotation describes this fact better than does interface inheritance. Of course, the `Serializable` interface was created in JDK 1.1, long before annotations existed.

Suppose you define an inherited annotation `@Persistent` to indicate that objects of a class can be saved in a database. Then the subclasses of persistent classes are automatically annotated as persistent.

```
@Inherited @interface Persistent { }

@Persistent class Employee { . . . }

class Manager extends Employee { . . . } // also @Persistent
```

When the persistence mechanism searches for objects to store in the database, it will detect both `Employee` and `Manager` objects.

As of Java 8, it is legal to apply the same annotation type multiple times to an item. For

http://www.rejoiceblog.com/

backward compatibility, the implementor of a repeatable annotation needs to provide a *container annotation* that holds the repeated annotations in an array.

Here is how to define the `@TestCase` annotation and its container:

```
@Repeatable(TestCases.class)

@interface TestCase

{

   String params();

   String expected();

}



@interface TestCases

{

   TestCase[] value();

}
```

Whenever the user supplies two or more `@TestCase` annotations, they are automatically wrapped into a `@TestCases` annotation.

Caution

You have to be careful when processing repeatable annotations. If you call `getAnnotation` to look up a repeatable annotation, and the annotation was actually repeated, then you get `null`. That is because the repeated annotations were wrapped into the container annotation.

http://www.rejoiceblog.com/

In that case, you should call `getAnnotationsByType`. That call "looks through" the container and gives you an array of the repeated annotations. If there was just one annotation, you get it in an array of length 1. With this method, you don't have to worry about the container annotation.

8.6 SOURCE-LEVEL ANNOTATION PROCESSING

In the preceding section, you saw how to analyze annotations in a running program. Another use for annotation is the automatic processing of source files to produce more source code, configuration files, scripts, or whatever else one might want to generate.

8.6.1 Annotation Processors

Annotation processing is integrated into the Java compiler. During compilation, you can *invoke annotation processors* by running

```
javac -processor ProcessorClassName1, ProcessorClassName2,. . . sourc
```

The compiler locates the annotations of the source files. Each annotation processor is executed in turn and given the annotations in which it expressed an interest. If an annotation processor creates a new source file, the process is repeated. Once a processing round yields no further source files, all source files are compiled.

Note

An annotation processor can only generate new source files. It cannot modify an existing source file.

An annotation processor implements the `Processor` interface, generally by extending the `AbstractProcessor` class. You need to specify which annotations your processor supports. In our case:

```
@SupportedAnnotationTypes("com.horstmann.annotations.ToString
```

http://www.rejoiceblog.com/

```
@SupportedSourceVersion(SourceVersion.RELEASE_8)

public class ToStringAnnotationProcessor extends AbstractProc

{

   public boolean process(Set<? extends TypeElement> annotati

      RoundEnvironment currentRound)

   {

      . . .

   }

}
```

A processor can claim specific annotation types, wildcards such as `"com.horstmann.*"` (all annotations in the `com.horstmann` package or any subpackage), or even `"*"` (all annotations).

The `process` method is called once for each round, with the set of all annotations that were found in any files during this round, and a `RoundEnvironment` reference that contains information about the current processing round.

8.6.2 The Language Model API

Use the *language model* API for analyzing source-level annotations. Unlike the reflection API, which presents the virtual machine representation of classes and methods, the language model API lets you analyze a Java program according to the rules of the Java language.

The compiler produces a tree whose nodes are instances of classes that implement the `javax.lang.model.element.Element` interface and its subinterfaces: `TypeElement`, `VariableElement`, `ExecutableElement`, and so on. These are the compile-time analogs to the `Class`, `Field/Parameter`, `Method/Constructor` reflection classes.

http://www.rejoiceblog.com/

I do not want to cover the API in detail, but here are the highlights that you need to know for processing annotations:

• The `RoundEnvironment` gives you a set of all elements annotated with a particular annotation, by calling the method

```
Set<? extends Element> getElementsAnnotatedWith(Class<? exten
```

• The source-level equivalent of the `AnnotateElement` interface is `AnnotatedConstruct`. Use the methods

```
A getAnnotation(Class<A> annotationType)

A[] getAnnotationsByType(Class<A> annotationType)
```

to get the annotation or repeated annotations for a given annotation class.

• A `TypeElement` represents a class or interface. The `getEnclosedElements` method yields a list of its fields and methods.

• Calling `getSimpleName` on an `Element` or `getQualifiedName` on a `TypeElement` yields a `Name` object that can be converted to a string with `toString`.

8.6.3 Using Annotations to Generate Source Code

As an example, we will use annotations to reduce the tedium of implementing `toString` methods. We can't put these methods into the original classes—annotation processors can only produce new classes, not modify existing ones.

Therefore, we'll add all methods into a utility class `ToStrings`:

```
public class ToStrings

{

   public static String toString(Point obj)
```

http://www.rejoiceblog.com/

```
    {
      Generated code
    }
    public static String toString(Rectangle obj)
    {
      Generated code
    }
    . . .
    public static String toString(Object obj)
    {
      return Objects.toString(obj);
    }
}
```

We don't want to use reflection, so we annotate accessor methods, not fields:

```
@ToString
public class Rectangle
{
    . . .
    @ToString(includeName=false) public Point getTopLeft() { r
```

http://www.rejoiceblog.com/

```
    @ToString public int getWidth() { return width; }

    @ToString public int getHeight() { return height; }

  }
```

The annotation processor should then generate the following source code:

```
  public static String toString(Rectangle obj)

    {

      var result = new StringBuilder();

      result.append("Rectangle");

      result.append("[");

      result.append(toString(obj.getTopLeft()));

      result.append(",");

      result.append("width=");

      result.append(toString(obj.getWidth()));

      result.append(",");

      result.append("height=");

      result.append(toString(obj.getHeight()));

      result.append("]");

      return result.toString();

    }
```

http://www.rejoiceblog.com/

The “boilerplate” code is in gray. Here is an outline of the method that produces the `toString` method for a class with given `TypeElement`:

```
private void writeToStringMethod(PrintWriter out, TypeElement

{

    String className = te.getQualifiedName().toString();

    Print method header and declaration of string builder

    ToString ann = te.getAnnotation(ToString.class);

    if (ann.includeName())

      Print code to add class name

    for (Element c : te.getEnclosedElements())

    {

      ann = c.getAnnotation(ToString.class);

      if (ann != null)

      {

          if (ann.includeName()) Print code to add field name

          Print code to append toString(obj.methodName())

      }

    }

    Print code to return string

}
```

http://www.rejoiceblog.com/

And here is an outline of the `process` method of the annotation processor. It creates a source file for the helper class and writes the class header and one method for each annotated class.

```
public boolean process(Set<? extends TypeElement> annotations

     RoundEnvironment currentRound)

{

   if (annotations.size() == 0) return true;

   try

   {

     JavaFileObject sourceFile = processingEnv.getFiler().cre

         "com.horstmann.annotations.ToStrings");

     try (var out = new PrintWriter(sourceFile.openWriter()))

     {

         Print code for package and class

         for (Element e : currentRound.getElementsAnnotatedWit

         {

           if (e instanceof TypeElement)

           {

              TypeElement te = (TypeElement) e;

              writeToStringMethod(out, te);
```

http://www.rejoiceblog.com/

```
            }

        }

        Print code for toString(Object)

    }

    catch (IOException ex)

    {

        processingEnv.getMessager().printMessage(

            Kind.ERROR, ex.getMessage());

    }

  }

  return true;

}
```

For the tedious details, check the book's companion code.

Note that the `process` method is called in subsequent rounds with an empty list of annotations. It then returns immediately so it doesn't create the source file twice.

First compile the annotation processor, and then compile and run the test program as follows:

```
javac sourceAnnotations/ToStringAnnotationProcessor.java

javac -processor sourceAnnotations.ToStringAnnotationProcesso

java rect.SourceLevelAnnotationDemo
```

http://www.rejoiceblog.com/

Tip

To see the rounds, run the `javac` command with the `-XprintRounds` flag:

```
Round 1:
  input files: {rect.Point, rect.Rectangle,
     rect.SourceLevelAnnotationDemo}
  annotations: [sourceAnnotations.ToString]
  last round: false
Round 2:
  input files: {sourceAnnotations.ToStrings}
  annotations: []
  last round: false
Round 3:
  input files: {}
  annotations: []
  last round: true
```

This example demonstrates how tools can harvest source file annotations to produce other files. The generated files don't have to be source files. Annotation processors may choose to generate XML descriptors, property files, shell scripts, HTML documentation, and so on.

http://www.rejoiceblog.com/

Note

Some people have suggested using annotations to remove an even bigger drudgery. Wouldn't it be nice if trivial getters and setters were generated automatically? For example, the annotation

```
@Property private String title;
```

could produce the methods

```
public String getTitle() { return title; }

public void setTitle(String title) { this = title; }
```

However, those methods need to be added to the *same class*. This requires editing a source file, not just generating another file, and is beyond the capabilities of annotation processors. It would be possible to build another tool for this purpose, but such a tool would go beyond the mission of annotations. An annotation is intended as a description *about* a code item, not a directive for adding or changing code.

8.7 BYTECODE ENGINEERING

You have seen how annotations can be processed at runtime or at the source code level. There is a third possibility: processing at the bytecode level. Unless annotations are removed at the source level, they are present in the class files. The class file format is documented (see `http://docs.oracle.com/javase/specs/jvms/se10/html`). The format is rather complex, and it would be challenging to process class files without special libraries. One such library is the ASM library, available at `http://asm.ow2.org`.

8.7.1 Modifying Class Files

In this section, we use ASM to add logging messages to annotated methods. If a method is annotated with

```
@LogEntry(logger=loggerName)
```

http://www.rejoiceblog.com/

then we add the bytecodes for the following statement at the beginning of the method:

```
Logger.getLogger(loggerName).entering(className, methodName);
```

For example, if you annotate the `hashCode` method of the `Item` class as

```
@LogEntry(logger="global") public int hashCode()
```

then a message similar to the following is printed whenever the method is called:

```
May 17, 2016 10:57:59 AM Item hashCode

FINER: ENTRY
```

To achieve this, we do the following:

1. Load the bytecodes in the class file.

2. Locate all methods.

3. For each method, check whether it has a `LogEntry` annotation.

4. If it does, add the bytecodes for the following instructions at the beginning of the method:

```
ldc loggerName

invokestatic

   java/util/logging/Logger.getLogger:(Ljava/lang/String;)Lja

ldc className

ldc methodName

invokevirtual java/util/logging/Logger.entering:(Ljava/lang/S
```

http://www.rejoiceblog.com/

Inserting these bytecodes sounds tricky, but ASM makes it fairly straightforward. We don't describe the process of analyzing and inserting bytecodes in detail. The important point is that the program in Listing 8.9 edits a class file and inserts a logging call at the beginning of the methods annotated with the `LogEntry` annotation.

For example, here is how you add the logging instructions to `Item.java` in Listing 8.10, where *asm* is the directory into which you installed the ASM library:

```
javac set/Item.java

javac -classpath .:asm/lib/\* bytecodeAnnotations/EntryLogger.

java -classpath .:asm/lib/\* bytecodeAnnotations.EntryLogger s
```

Try running

```
javap -c set.Item
```

before and after modifying the `Item` class file. You can see the inserted instructions at the beginning of the `hashCode`, `equals`, and `compareTo` methods.

```
public int hashCode();

   Code:

   0: ldc #85; // String global

   2:  invokestatic  #80;

      // Method java/util/logging/Logger.getLogger:(Ljava/lan

   5: ldc   #86; //String Item

   7: ldc   #88; //String hashCode
```

http://www.rejoiceblog.com/

```
      9: invokevirtual  #84;

         // Method java/util/logging/Logger.entering:(Ljava/lang

      12: bipush  13

      14: aload_0

      15: getfield      #2; // Field description:Ljava/lang/Stri

      18: invokevirtual #15; // Method java/lang/String.hashCode

      21: imul

      22: bipush 17

      24: aload_0

      25: getfield      #3; // Field partNumber:I

      28: imul

      29: iadd

      30: ireturn
```

The `SetTest` program in Listing 8.11 inserts `Item` objects into a hash set. When you run it with the modified class file, you will see the logging messages.

```
   May 17, 2016 10:57:59 AM Item hashCode

   FINER: ENTRY

   May 17, 2016 10:57:59 AM Item hashCode

   FINER: ENTRY
```

http://www.rejoiceblog.com/

```
May 17, 2016 10:57:59 AM Item hashCode

FINER: ENTRY

May 17, 2016 10:57:59 AM Item equals

FINER: ENTRY

[[description=Toaster, partNumber=1729], [description=Microwa
```

Note the call to `equals` when we insert the same item twice.

This example shows the power of bytecode engineering. Annotations are used to add directives to a program, and a bytecode editing tool picks up the directives and modifies the virtual machine instructions.

Listing 8.9 `bytecodeAnnotations/EntryLogger.java`

```
package bytecodeAnnotations;

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

import org.objectweb.asm.*;
import org.objectweb.asm.commons.*;

/**
 * Adds "entering" logs to all methods of a class that ha
```

http://www.rejoiceblog.com/

```
11   * @version 1.21 2018-05-01
12   * @author Cay Horstmann
13   */
14  public class EntryLogger extends ClassVisitor
15  {
16     private String className;
17
18    /**
19     * Constructs an EntryLogger that inserts logging into
20     * @param cg the class
21     */
22    public EntryLogger(ClassWriter writer, String className
23    {
24       super(Opcodes.ASM5, writer);
25       this.className = className;
26    }
27
28    public MethodVisitor visitMethod(int access, String met
29          String signature, String[] exceptions)
30    {
```

http://www.rejoiceblog.com/

```
31        MethodVisitor mv = cv.visitMethod(access, methodName
32        return new AdviceAdapter(Opcodes.ASM5, mv, access, m
33        {
34           private String loggerName;
35
36           public AnnotationVisitor visitAnnotation(String d
37           {
38           return new AnnotationVisitor(Opcodes.ASM5)
39             {
40               public void visit(String name, Object value)
41               {
42                  if (desc.equals("LbytecodeAnnotations/LogE
43                     loggerName = value.toString();
44               }
45             };
46        }
47
48        public void onMethodEnter()
49        {
```

http://www.rejoiceblog.com/

```
        if (loggerName != null)
        {
          visitLdcInsn(loggerName);
          visitMethodInsn(INVOKESTATIC, "java/util/logging
            "(Ljava/lang/String;)Ljava/util/logging/Logger
          visitLdcInsn(className);
          visitLdcInsn(methodName);
          visitMethodInsn(INVOKEVIRTUAL, "java/util/loggin
            "(Ljava/lang/String;Ljava/lang/String;)V", fal
          loggerName = null;
        }
      }
    };
}

/**
 * Adds entry logging code to the given class.
 * @param args the name of the class file to patch
 */
public static void main(String[] args) throws IOException
```

http://www.rejoiceblog.com/

```
  {
    if (args.length == 0)
    {
      System.out.println("USAGE: java bytecodeAnnotations.
      System.exit(1);
    }
    Path path = Paths.get(args[0]);
    var reader = new ClassReader(Files.newInputStream(path
    var writer = new ClassWriter(
      ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAME
    var entryLogger = new EntryLogger(writer,
      path.toString().replace(".class", "").replaceAll("[/
    reader.accept(entryLogger, ClassReader.EXPAND_FRAMES);
    Files.write(Paths.get(args[0]), writer.toByteArray());
  }
}
```

Listing 8.10 `set/Item.java`

```
package set;
```

http://www.rejoiceblog.com/

```
import java.util.*;
import bytecodeAnnotations.*;

/**
  * An item with a description and a part number.
  * @version 1.01 2012-01-26
  * @author Cay Horstmann
  */
public class Item
{
      private String description;
      private int partNumber;

    /**
      * Constructs an item.
      * @param aDescription the item's description
      * @param aPartNumber the item's part number
      */
    public Item(String aDescription, int aPartNumber)
```

http://www.rejoiceblog.com/

```
22      {
23          description = aDescription;
24          partNumber = aPartNumber;
25      }
26
27      /**
28        * Gets the description of this item.
29        * @return the description
30        */
31      public String getDescription()
32      {
33          return description;
34      }
35
36      public String toString()
37      {
38         return "[description=" + description + ", partNumb
39      }
40
```

http://www.rejoiceblog.com/

```
    @LogEntry(logger = "com.horstmann")
    public boolean equals(Object otherObject)
    {
       if (this == otherObject) return true;
       if (otherObject == null) return false;
       if (getClass() != otherObject.getClass()) return f
       var other = (Item) otherObject;
       return Objects.equals(description, other.descripti
     }

     @LogEntry(logger = "com.horstmann")
     public int hashCode()
     {
        return Objects.hash(description, partNumber);
     }
}
```

Listing 8.11 set/SetTest.java

```
package set;

```

http://www.rejoiceblog.com/

```
 3   import java.util.*;
 4   import java.util.logging.*;
 5
 6   /**
 7     * @version 1.03 2018-05-01
 8     * @author Cay Horstmann
 9     */
10   public class SetTest
11   {
12       public static void main(String[] args)
13       {
14          Logger.getLogger("com.horstmann").setLevel(Level.
15          var handler = new ConsoleHandler();
16          handler.setLevel(Level.FINEST);
17          Logger.getLogger("com.horstmann").addHandler(hand
18
19           var parts = new HashSet<Item>();
20           parts.add(new Item("Toaster", 1279));
21           parts.add(new Item("Microwave", 4104));
```

http://www.rejoiceblog.com/

```
            parts.add(new Item("Toaster", 1279));

            System.out.println(parts);

        }

    }
```

8.7.2 Modifying Bytecodes at Load Time

In the preceding section, you saw a tool that edits class files. However, it can be cumbersome to add yet another tool into the build process. An attractive alternative is to defer the bytecode engineering until *load time*, when the class loader loads the class.

The *instrumentation API* has a hook for installing a bytecode transformer. The transformer must be installed before the `main` method of the program is called.

You can meet this requirement by defining an *agent*, a library that is loaded to monitor a program in some way. The agent code can carry out initializations in a `premain` method.

Here are the steps required to build an agent:

1. Implement a class with a method

```
public static void premain(String arg, Instrumentation instr)
```

This method is called when the agent is loaded. The agent can get a single command-line argument, which is passed in the `arg` parameter. The `instr` parameter can be used to install various hooks.

2. Make a manifest file `EntryLoggingAgent.mf` that sets the `Premain-Class` attribute, for example:

```
Premain-Class: bytecodeAnnotations.EntryLoggingAgent
```

3. Package the agent code and the manifest into a JAR file:

http://www.rejoiceblog.com/

```
javac -classpath .:asm/lib/\* bytecodeAnnotations/EntryLoggin

jar cvfm EntryLoggingAgent.jar bytecodeAnnotations/EntryLoggi

   bytecodeAnnotations/Entry*.class
```

To launch a Java program together with the agent, use the following command-line options:

```
java -javaagent:AgentJARFile=agentArgument . . .
```

For example, to run the `SetTest` program with the entry logging agent, call

```
javac set/SetTest.java

java -javaagent:EntryLoggingAgent.jar=set.Item -classpath .:as
```

The `Item` argument is the name of the class that the agent should modify.

Listing 8.12 shows the agent code. The agent installs a class file transformer. The transformer first checks whether the class name matches the agent argument. If so, it uses the `EntryLogger` class from the preceding section to modify the bytecodes. However, the modified bytecodes are not saved to a file. Instead, the transformer returns them for loading into the virtual machine (see Figure 8.3). In other words, this technique carries out "just in time" modification of the bytecodes.

http://www.rejoiceblog.com/

Figure 8.3 Modifying classes at load time

Listing 8.12 `bytecodeAnnotations/EntryLoggingAgent.java`

```
package bytecodeAnnotations;

import java.lang.instrument.*;

import org.objectweb.asm.*;

/**
 * @version 1.11 2018-05-01
 * @author Cay Horstmann
 */
public class EntryLoggingAgent
{
      public static void premain(final String arg, Instru
      {
         instr.addTransformer((loader, className, cl, pd,
            {
               if (!className.replace("/", ".").equals(ar
               var reader = new ClassReader(data);
```

http://www.rejoiceblog.com/

```
                var writer = new ClassWriter(
                   ClassWriter.COMPUTE_MAXS | ClassWriter.
                var el = new EntryLogger(writer, className
                reader.accept(el, ClassReader.EXPAND_FRAME
                return writer.toByteArray();
            });
        }
    }
```

In this chapter, you have learned how to

• Add annotations to Java programs

• Design your own annotation interfaces

• Implement tools that make use of the annotations

You have seen three technologies for processing code: scripting, compiling Java programs, and processing annotations. The first two were quite straightforward. On the other hand, building annotation tools is undeniably complex and not something that most developers will need to tackle. This chapter gave you the background for understanding the inner workings of the annotation tools you will encounter, and perhaps piqued your interest in developing your own tools.

The following chapter discusses the Java Platform Module System, the key feature of Java 9 that is important for moving the Java platform forward.

http://www.rejoiceblog.com/

Chapter 9 The Java Platform Module System

In this chapter

- 9.1 The Module Concept
- 9.2 Naming Modules
- 9.3 The Modular “Hello, World!” Program
- 9.4 Requiring Modules
- 9.5 Exporting Packages
- 9.6 Modular JARs
- 9.7 Modules and Reflective Access
- 9.8 Automatic Modules
- 9.9 The Unnamed Module
- 9.10 Command-Line Flags For Migration
- 9.11 Transitive and Static Requirements
- 9.12 Qualified Exporting and Opening
- 9.13 Service Loading
- 9.14 Tools for Working With Modules

An important characteristic of object-oriented programming is encapsulation. A class declaration consists of a public interface and a private implementation. A class can evolve by changing the implementation without affecting its users. A module system provides the same benefits for programming in the large. A module can make classes

http://www.rejoiceblog.com/
and packages selectively available so that its evolution can be controlled.

Several existing Java module systems rely on class loaders to isolate classes. However, Java 9 introduced a new system, called the Java Platform Module System, that is supported by the Java compiler and virtual machine. It was designed to modularize the large code base of the Java platform. You can, if you choose, use this system to modularize your own applications.

Whether or not you use Java platform modules in your own applications, you may be impacted by the modularized Java platform. This chapter shows you how to declare and use Java platform modules. You will also learn how to migrate your applications to work with the modularized Java platform and third-party modules.

9.1 THE MODULE CONCEPT

In object-oriented programming, the fundamental building block is the class. Classes provide encapsulation. Private features can only be accessed by code that has explicit permission, namely the methods of the class. This makes it possible to reason about access. If a private variable has changed, you can produce a set of all possible culprits. If you need to modify the private representation, you know which methods are affected.

In Java, packages provide the next larger organizational grouping. A package is a collection of classes. Packages also provide a level of encapsulation. Any feature with package access (neither public nor private) is accessible only from methods in the same package.

However, in large systems, this level of access control is not enough. Any public feature (that is, a feature that is accessible outside a package) is accessible everywhere. Suppose you want to modify or drop a rarely-used feature. Once it is public, there is no way to reason about the impact of that change.

This is the situation that the Java platform designers faced. Over twenty years, the JDK grows by leaps and bounds, but clearly some features are essentially obsolete. Everyone's favorite example is CORBA. When is the last time you used it? Yet, the `org.omg.corba` package was shipped with every JDK until Java 10. As of Java 11, those few who still need it must add the required JAR files to their projects.

What about `java.awt`? It shouldn't be required in a server-side application, right? Except that the class `java.awt.DataFlavor` is used in the implementation of SOAP, an XML-based web services protocol.

http://www.rejoiceblog.com/

The Java platform designers, faced with a giant hairball of code, decided that they needed a structuring mechanism that provides more control. They looked at existing module systems (such as OSGi) and found them unsuitable for their problem. Instead, they designed a new system, called the *Java Platform Module System*, that is now a part of the Java language and virtual machine. That system has been used successfully to modularize the Java API, and you can, if you so choose, use it with your own applications.

A Java platform module consists of:

• a collection of packages

• optionally, resource files and other files such as native libraries

• a list of the accessible packages in the module

• a list of all modules on which this module depends

The Java platform enforces encapsulation and dependencies, both at compile time and in the virtual machine.

Why should you consider using the Java Platform Module System for your own programs instead of following the traditional approach of using JAR files on the class path? There are two advantages.

1. Strong encapsulation: You can control which of your packages are accessible, and you don't have to worry about maintaining code that you didn't intend for public consumption.

2. Reliable configuration: You avoid common class path problems such as duplicate or missing classes.

There are some issues that the Java Platform Module System does not address, such as versioning of modules. There is no support for specifying which version of a module is required, or for using multiple versions of a module in the same program. These can be desirable features, but you must use mechanisms other than the Java Platform Module System if you need them.

9.2 NAMING MODULES

A module is a collection of packages. The package names in the module need not be related. For example, the module `java.sql` contains packages `java.sql`,

http://www.rejoiceblog.com/

`javax.sql`, and `javax.transaction.xa`. Also, as you can see from this example, it is perfectly acceptable for the module name to be the same as a package name.

Just like a path name, a module name is made up of letters, digits, underscores, and periods. Also, just as with path names, there is no hierarchical relationship between modules. If you had a module `com.horstmann` and another module `com.horstmann.corejava`, they would be unrelated, as far as the module system is concerned.

When creating a module for use by others, it is important that its name is globally unique. It is expected that most module names will follow the "reverse domain name" convention, just like package names.

The easiest approach is to name a module after the top-level package that the module provides. For example, the SLF4J logging façade has a module `org.slf4j` with packages `org.slf4j`, `org.slf4j.spi`, `org.slf4j.event`, and `org.slf4j.helpers`.

This convention prevents package name conflicts in modules. Any given package can only be placed in one module. If your module names are unique and your package names start with the module name, then your package names will also be unique.

You can use shorter module names for modules that are not meant to be used by other programmers, such as a module containing an application program. Just to show that it can be done, I will do the same in this chapter. Modules with what could plausibly be library code will have names such as `com.horstmann.util`, and modules containing programs (with a class that has a `main` method) will have catchy names such as `v2ch09.hellomod`.

Note

You only use module names in module declarations. In the source files for your Java classes, you never refer to module names. Instead, you use package names in the same way that they have always been used.

9.3 THE MODULAR "HELLO, WORLD!" PROGRAM

Let us put the traditional "Hello, World!" program into a module. First we need to put

http://www.rejoiceblog.com/
the class into a package—the “unnamed package” cannot be contained in a module. Here it is:

```
package com.horstmann.hello;



public class HelloWorld

{

   public static void main(String[] args)

   {

     System.out.println("Hello, Modular World!");

   }

}
```

So far, nothing has changed. To make a module `v2ch09.hellomod` containing this package, you need to add a module declaration. You place it in a file named `module-info.java`, located in the base directory (that is, the same directory that contains the `com` directory). By convention, the name of the base directory is the same as the module name.

```
v2ch09.hellomod/
└ module-info.java
  com/
  └ horstmann/
    └ hello/
      └ HelloWorld.java
```

The `module-info.java` file contains the module declaration:

```
module v2ch09.hellomod

{
```

http://www.rejoiceblog.com/

```
}
```

This module declaration is empty because the module has nothing to offer to anyone, nor does it need anything.

Now you compile as usual:

```
javac v2ch09.hellomod/module-info.java v2ch09.hellomod/com/ho
```

The `module-info.java` file doesn't look like a Java source file, and of course there can't be a class with the name `module-info`, since class names cannot contain hyphens. The `module` keyword, as well as keywords `requires`, `exports`, and so on, that you will see in the following sections, are "restricted keywords" that have a special meaning only in module declarations. The file is compiled into a class file `module-info.class` that contains the module definition in binary form.

To run this program as a modular application, you specify the *module path*, which is similar to the class path, but it contains modules. You also specify the main class in the format *modulename/classname*:

```
java --module-path v2ch09.hellomod --module v2ch09.hellomod/c
```

Instead of `--module-path` and `--module`, you can use the single-letter options `-p` and `-m`:

```
java -p v2ch09.hellomod -m v2ch09.hellomod/com.horstmann.hell
```

Either way, the `"Hello, Modular World"` greeting will appear, demonstrating that you have successfully modularized your first application.

Note

http://www.rejoiceblog.com/

When you compile this module, you get a warning:

```
warning: [module] module name component v2ch09 should avoid termina
```

This warning is intended to discourage programmers from adding version numbers to module names. You can ignore the warning, or suppress it with an annotation:

```
@SuppressWarnings("module")

module v2ch09.hellomod

{

}
```

In this one respect, the `module` declaration is just like a class declaration: You can annotate it. (The annotation type must have target `ElementType.MODULE`.)

9.4 REQUIRING MODULES

Let us make a new module `v2ch09.requiremod` in which a program uses a `JOptionPane` to show the "Hello, Modular World" message:

```
package com.horstmann.hello;



import javax.swing.JOptionPane;



public class HelloWorld

{
```

http://www.rejoiceblog.com/

```
    public static void main(String[] args)

    {

      JOptionPane.showMessageDialog(null, "Hello, Modular Worl

    }

  }
```

Now compilation fails with this message:

```
  error: package javax.swing is not visible

    (package javax.swing is declared in module java.desktop, b
```

The JDK has been modularized, and the `javax.swing` package is now contained in the `java.desktop` module. Our module needs to declare that it relies on that module:

```
  module v2ch09.requiremod

  {

    requires java.desktop;

  }
```

It is a design goal of the module system that modules are explicit about their requirements, so that the virtual machine can ensure that all requirements are fulfilled before starting a program.

In the preceding section, the need for explicit requireents did not arise because we only used the `java.lang` and `java.io` packages. These packages are included in the `java.base` module which is required by default.

http://www.rejoiceblog.com/

Note that our `v2ch09.requiremod` module lists only its own module requirements. It requires the `java.desktop` module so that it can use the `javax.swing` package. The `java.desktop` module itself declares that it requires three other modules, namely `java.datatransfer`, `java.prefs`, and `java.xml`.

Figure 9.1 shows the *module graph* whose nodes are modules. The edges of the graph (that is, the arrows joining nodes) are either declared requirements or the implied requirement on `java.base` when none is declared.

Figure 9.1 The module graph of the Swing "Hello, Modular World" application

You cannot have cycles in the module graph. That is, a module cannot directly or indirectly require itself.

A module does not automatically pass on access rights to other modules. In our example, the `java.desktop` module declares that it requires `java.prefs`, and the `java.prefs` module declares that it requires `java.xml`. That does not give `java.desktop` the right to use packages from the `java.xml` module. It needs to explicitly declare that requirement. In mathematical terms, the `requires` relationship is not "transitive". Generally, this behavior is desirable because it makes requirements explicit, but as you will see in Section 9.11, "Transitive and Static

http://www.rejoiceblog.com/
Requirements," on p. 520, you can relax it in some cases.

> **Note**
>
> The error message at the beginning of this section stated that our `v2ch09.requiremod` module did not "read" the `java.desktop` module. In the parlance of the Java Platform Module System, module *M reads* module *N* in the following cases:
>
> 1. *M* requires *N*
>
> 2. *M* requires a module that transitively requires *N* (see Section 9.11, "Transitive and Static Requirements," on p. 520)
>
> 3. *N* is *M* or `java.base`

9.5 EXPORTING PACKAGES

In the preceding section, you saw that a module must require another module if it wants to use its packages. However, that does not automatically make all packages in the required module available. A module states which of its packages are accessible, using the `exports` keyword. For example, here is a part of the module declaration for the `java.xml` module:

```
module java.xml

{

   exports javax.xml;

   exports javax.xml.catalog;

   exports javax.xml.datatype;

   exports javax.xml.namespace;

   exports javax.xml.parsers;
```

http://www.rejoiceblog.com/

```
    ...

}
```

This module makes many packages available, but hides others (such as `jdk.xml.internal`) by not exporting them.

When a package is exported, its `public` and `protected` classes and interfaces, and their `public` and `protected` members, are accessible outside the module. (As always, `protected` types and members are accessible only in subclasses.)

However, a package that is not exported is not accessible outside its own module. This is quite different from Java before modules. In the past, you were able to use public classes from any package, even if it was not part of the public API. For example, it was commonly recommended to use classes such as `sun.misc.BASE64Encoder` or `com.sun.rowset.CachedRowSetImpl` when the public API did not provide the appropriate functionality.

Nowadays, you can no longer access unexported packages from the Java platform API since all of them are contained inside modules. As a result, some programs will no longer run with Java 9. Of course, nobody ever committed to keeping non-public APIs available, so this should not come as a shock.

Let us put exports to use in a simple situation. We will prepare a module `com.horstmann.greet` that exports a package, also called `com.horstmann.greet`, following the convention that a module that provides code for others should be named after the top-level package inside it. There is also a package `com.horstmann.greet.internal` that we don't export.

A public `Greeter` interface is in the first package.

```
package com.horstmann.greet;



public interface Greeter

{
```

http://www.rejoiceblog.com/

```
    static Greeter newInstance()

    {

      return new com.horstmann.greet.internal.GreeterImpl();

    }

    String greet(String subject);

  }
```

The second package has a class that implements the interface. The class is public since it is accessed in the first package.

```
  package com.horstmann.greet.internal;



  import com.horstmann.greet.Greeter;



  public class GreeterImpl implements Greeter

  {

    public String greet(String subject)

    {

      return "Hello, " + subject + "!";

    }

  }
```

http://www.rejoiceblog.com/

The `com.horstmann.greet` module contains both packages but only exports the first:

```
module com.horstmann.greet

{

   exports com.horstmann.greet;

}
```

The second package is inaccessible outside the module.

We put our application into a second module, which will require the first module:

```
module v2ch09.exportedpkg

{

   requires com.horstmann.greet;

}
```

Note

The `exports` statement is followed by a package name, whereas `requires` is followed by a module name.

Our application now uses a `Greeter` to obtain a greeting:

```
package com.horstmann.hello;
```

http://www.rejoiceblog.com/

```
import com.horstmann.greet.Greeter;



public class HelloWorld

{

   public static void main(String[] args)

   {

     Greeter greeter = Greeter.newInstance();

     System.out.println(greeter.greet("Modular World"));

   }

}
```

Here is the source file structure for these two modules:

```
com.horstmann.greet
├ module-info.java
└ com
  └ horstmann
    └ greet
      ├ Greeter.java
      └ internal
        └ GreeterImpl.java
v2ch09.exportedpkg
├ module-info.java
└ com
└ horstmann
  └ hello
    └ HelloWorld.java
```

To build this application, first compile the `com.horstmann.greet` module:

```
javac com.horstmann.greet/module-info.java \
```

http://www.rejoiceblog.com/

```
    com.horstmann.greet/com/horstmann/greet/Greeter.java \

    com.horstmann.greet/com/horstmann/greet/internal/GreeterIm
```

Then compile the application module with the first module on the module path:

```
javac -p com.horstmann.greet v2ch09.exportedpkg/module-info.j

    v2ch09.exportedpkg/com/horstmann/hello/HelloWorld.java
```

Finally, run the program with both modules on the module path:

```
java -p v2ch09.exportedpkg:com.horstmann.greet \

    -m v2ch09.exportedpkg/com.horstmann.hello.HelloWorld
```

Tip

To build this application with Eclipse, make a separate project for each module. In the `v2ch09.exportedpkg` project, edit the project properties. In the Projects tab, add the `com.horstmann.greet` module to the module path —see Figure 9.2.

http://www.rejoiceblog.com/

Figure 9.2 Adding a dependent module to an Eclipse project

You have now seen the `requires` and `exports` statements that form the backbone of the Java Platform Module System. As you can see, the module system is conceptually simple. Modules specify what modules they need, and which packages they offer to other modules. Section 9.12, "Qualified Exporting and Opening," on p. 521 shows a minor variation of the `exports` statement.

Caution

A module does not provide a scope. You cannot have two packages with the same name in different modules. This is true even for hidden packages (that is, packages that are not exported.)

9.6 MODULAR JARS

So far, we have simply compiled modules into the directory tree of the source code. Clearly, that is not satisfactory for deployment. Instead, a module can be deployed by placing all its classes in a JAR file, with a `module-info.class` in the root. Such a JAR file is called a *modular* JAR.

http://www.rejoiceblog.com/

To create a modular JAR file, you use the `jar` tool in the usual way. If you have multiple packages, it is best to compile with the `-d` option which places class files into a separate directory. The directory is created if it doesn't already exists. Then use the `-C` option of the `jar` command to change to that directory when collecting files.

```
javac -d modules/com.horstmann.greet $(find com.horstmann.gre

jar -cvf com.horstmann.greet.jar -C modules/com.horstmann.gre
```

If you use a build tool such as Maven, Ant, or Gradle, just keep building your JAR file as you always do. As long as `module-info.class` is included, you get a modular JAR.

Then you can include the modular JAR in the module path, and the module will be loaded.

> **Caution**
>
> In the past, the classes of a package were sometimes distributed over multiple JAR files. (Such a package is called a "split package".) This was probably never a good idea, and it is not possible with modules.

As with regular JAR files, you can specify a main class in a modular JAR:

```
javac -p com.horstmann.greet.jar -d modules/v2ch09.exportedpk

jar -c -v -f v2ch09.exportedpkg.jar -e com.horstmann.hello.He
```

When you launch the program, you specify the module containing the main class:

```
java -p com.horstmann.greet.jar:v2ch09.exportedpkg.jar -m v2c
```

http://www.rejoiceblog.com/

When creating a JAR file, you can optionally specify a version number. Use the `--module-version` parameter, and also add `@` and the version number to the JAR file name:

```
jar -c -v -f com.horstmann.greet@1.0.jar --module-version 1.0
```

As already discussed, the version number is not used by the Java Platform Module System for resolving modules, but it can be queried by other tools and frameworks.

Note

You can find out the version number through the reflection API. In our example:

```
Optional<String> version = Greeter.class.getModule().getDescriptor().rav
```

yields an `Optional` containing the version string `"1.0"`.

Note

The module equivalent to a class loader is a *layer*. The Java Platform Module System loads the JDK modules and application modules into the *boot layer*. A program can load other modules, using the layer API (which is not covered in this book). Such a program may choose to take module versions into account. Is is expected that developers of programs such as Java EE application servers make use of the layer API to provide support for modules.

Tip

http://www.rejoiceblog.com/

If you want to load a module into JShell, include the JAR on the module path and use the `--add-modules` option:

```
jshell --module-path com.horstmann.greet@1.0.jar --add-modules com.h
```

9.7 MODULES AND REFLECTIVE ACCESS

In the preceding sections, you saw that the module system enforces encapsulation. A module can only access explicitly exported packages from another module. In the past, it was always possible to overcome pesky access restrictions by using reflection. As you have seen in Chapter 5 of Volume I, reflection can access private members of any class.

However, in the modular world, that is no longer true. If a class is inside a module, reflective access to non-public members will fail. Specifically, recall how we accessed private fields:

```
Field f = obj.getClass().getDeclaredField("salary");

f.setAccessible(true);

double value = f.getDouble(obj);

f.setDouble(obj, value * 1.1);
```

The call `f.setAccessible(true)` succeeds unless a security manager disallows private field access. However, it is not common to run Java applications with security managers, and there are many libraries that use reflective access. Typical examples are object-relational mappers such as JPA that automatically persist objects in databases, and libraries that convert between objects and XML or JSON, such as JAXB and JSON-B.

If you use such a library, and you also want to use modules, you have to be careful. To demonstrate this issue, let us place the `ObjectAnalyzer` class from Chapter 5 of Volume I into a module `com.horstmann.util`. That class has a `toString` method that prints the fields of an object, using reflection.

http://www.rejoiceblog.com/

A separate `v2ch09.openpkg` module contains a simple `Country` class:

```
package com.horstmann.places;



public class Country

{

   private String name;

   private double area;



   public Country(String name, double area)

   {

     this.name = name;

     this.area = area;

   }

   // ...

}
```

A short program demonstrates how to analyze a `Country` object:

```
package com.horstmann.places;



import com.horstmann.util.*;
```

http://www.rejoiceblog.com/

```
public class Demo

{

   public static void main(String[] args) throws ReflectiveOp

   {

     var belgium = new Country("Belgium", 30510);

     var analyzer = new ObjectAnalyzer();

     System.out.println(analyzer.toString(belgium));

   }

}
```

Now compile both modules and the `Demo` program:

```
javac com.horstmann.util/module-info.java \

   com.horstmann.util/com/horstmann/util/ObjectAnalyzer.java

javac -p com.horstmann.util v2ch09.openpkg/module-info.java \

   v2ch09.openpkg/com/horstmann/places/*.java

java -p v2ch09.openpkg:com.horstmann.util \

   -m v2ch09.openpkg/com.horstmann.places.Demo
```

The program will fail with an exception:

http://www.rejoiceblog.com/

```
Exception in thread "main" java.lang.reflect.InaccessibleObje

   make field private java.lang.String com.horstmann.places.C

   v2ch09.openpkg does not "opens com.horstmann.places" to mo
```

Of course, in pristine theory, it is wrong to violate encapsulation and poke around in the private members of an object. But mechanisms such as object-relational mapping or XML/JSON binding are so common that the module system must accommodate them.

Using the `opens` keyword, a module can *open* a package, which enables reflective access to all instances of classes in the given package. Here is what our module has to do:

```
module v2ch09.openpkg

{

   requires com.horstmann.util;

   opens com.horstmann.places;

}
```

With this change, the `ObjectAnalyzer` will work correctly.

A module can be declared as `open`, such as

```
open module v2ch09.openpkg

{

   requires com.horstmann.util;

}
```

http://www.rejoiceblog.com/

An open module grants runtime access to all of its packages, as if all packages had been declared with `exports` and `opens`. However, only explicitly exported packages are accessible at compile-time. Open modules combine the compile-time safety of the module system with the classic permissive run-time behavior.

Recall from Chapter 5 of Volume I that JAR files can contain, in addition to class files and a manifest, *file resources* which can be loaded with the method `Class.getResourceAsStream`, and now also with `Module.getResourceAsStream`. If a resource is stored in a directory that matches a package in a module, then the package must be opened to the caller. Resources in other directories, as well as the class files and manifest, can be read by anyone.

Note

For a more realistic example, we can convert the `Country` object to XML or JSON. Java 9 and 10 included the `java.xml.bind` module for conversion to XML. That module has been removed from Java 11 (together with the modules `java.activation`, `java.corba`, `java.transaction`, `java.xml.ws`, and `java.xml.ws.annotation`). These modules contain packages that are a also part of the Jakarta EE (formerly Java EE) specification, where the APIs are more encompassing than with Java SE. Enterprise application servers can not be modularized if the JDK has conflicting packages. Unfortunately, as this book is written, there is no modularized replacement for XML binding.

However, the JSON-B implementation provides modular JAR files if one builds it from source. Hopefully, these JAR files will make it to Maven Central by the time you read this. Place the JAR files on the module path and run the `com.horstmann.places.Demo2` program. When the `com.horstmann.places` package is opened, conversion to JSON succeeds.

Note

It is possible that future libraries will use *variable handles* instead of reflection for reading and writing fields. A `VarHandle` is similar to a `Field`. You can use it to read or write a specific field of any instance of a specific class. However, to obtain a `VarHandle`, the library code needs a `Lookup` object:

http://www.rejoiceblog.com/

```
public Object getFieldValue(Object obj, String fieldName, Lookup lookup)
  throws NoSuchFieldException, IllegalAccessException
{
  Class<?> cl = obj.getClass();
  Field field = cl.getDeclaredField(fieldName);
  VarHandle handle = MethodHandles.privateLookupIn(cl, lookup)
    .unreflectVarHandle(field);
  return handle.get(obj);
}
```

This works provided the `Lookup` object is generated in the module that has the permission to access the field. Some method in the module simply calls `MethodHandles.lookup()`, which yields an object encapsulating the access rights of the caller. In this way, one module can give permission for accessing private members to another module. The practical issue is how those permissions can be given with a minimum of hassle.

9.8 AUTOMATIC MODULES

You now know to put the Java Platform Module System to use. If you start with a brand-new project in which you write all the code yourself, you can design modules, declare module dependencies, and package your application into modular JAR files.

However, that is an extremely uncommon scenario. Almost all projects rely on third party libraries. Of course, you can wait until the providers of all libraries have turned them into modules, and then modularize your own code.

But what if you don't want to wait? The Java Platform Module System provides two mechanisms for crossing the chasm that separates today's pre-modular world and fully modular applications: automatic modules and the unnamed module.

For migration purposes, you can turn any JAR file into a module, simply by placing it

http://www.rejoiceblog.com/

onto a directory in the module path instead of the class path. A JAR without a `module-info.class` on the module path is called an *automatic module*. An automatic module has the following properties:

1. The module implicitly has a `requires` clause for all other modules.

2. All of its packages are exported and opened.

3. If there is an entry with key `Automatic-Module-Name` in the JAR file manifest `META-INF/MANIFEST.MF`, the value becomes the module name.

4. Otherwise the module name is obtained from the JAR file name, dropping any trailing version number and replacing sequences of non-alphanumeric characters with a dot.

The first two rules imply that the packages in the automatic module act as if they were on the class path. The reason for using the module path is for the benefit of other modules, allowing them to express dependencies on this module.

Suppose, for example, that you are implementing a module that processes CSV files and uses the Apache Commons CSV library. You would like to express in your `module-info.java` file that your module depends on Apache Commons CSV.

If you add `commons-csv-1.5.jar` onto the module path, then your modules can reference the module. Its name is `commons.csv` since the trailing version number `-1.5` is removed and the non-alphanumeric character – is replaced by a dot.

This name might be an acceptable module name because Commons CSV is well known and it is unlikely that someone else will try to use the same name for a different module. But it would be better if the maintainers of this JAR file could quickly agree to reserve a reverse DNS name, preferably the top-level package name `org.apache.commons.csv` as the module name. They just need to add a line

```
Automatic-Module-Name: org.apache.commons.csv
```

to the `META-INF/MANIFEST.MF` file inside the JAR. Eventually, hopefully, they will turn the JAR file into a true module by adding `module-info.java` with the reserved module name, and every other module that refers to the CSV module with that name will just continue to work.

http://www.rejoiceblog.com/

Note

The migration plan to modules is a great social experiment, and nobody knows whether it will end well. Before you put third party JARs on the module path, check whether they are modular, and if not, whether their manifest has a module name. If not, you can still turn the JAR into an automatic module, but be prepared to update the module name later.

As this book is written, version 1.5 of the Commons CSV JAR file does not have a module descriptor or an automatic module name. Nevertheless, it will work fine on the module path. You can download the library from `https://commons.apache.org/proper/commons-csv/`. Uncompress and place the `commons-csv-1.5.jar` file into the directory of the `v2ch9.automod` module. That module contains a simple program that reads a CSV file with country data:

```
package com.horstmann.places;



import java.io.*;

import org.apache.commons.csv.*;



public class CSVDemo

{

   public static void main(String[] args) throws IOException

   {

     var in = new FileReader("countries.csv");

     Iterable<CSVRecord> records = CSVFormat.EXCEL.withDelimi
```

http://www.rejoiceblog.com/

```
            .withHeader().parse(in);

        for (CSVRecord record : records)

        {

          String name = record.get("Name");

          double area = Double.parseDouble(record.get("Area"));

          System.out.println(name + " has area " + area);

        }

      }

    }
```

Since we will use `commons-csv-1.5.jar` as an automatic module, we need to require it:

```
@SuppressWarnings("module")

module v2ch09.automod

{

   requires commons.csv;

}
```

Here are the commands for compiling and running the program:

```
javac -p v2ch09.automod:commons-csv-1.5.jar \
```

http://www.rejoiceblog.com/

```
    v2ch09.automod/com/horstmann/places/CSVDemo.java \

    v2ch09.automod/module-info.java

java -p v2ch09.automod:commons-csv-1.5.jar \

    -m v2ch09.automod/com.horstmann.places.CSVDemo
```

9.9 THE UNNAMED MODULE

Any class that is not on the module path is part of an *unnamed module*. Technically, there may be more than one unnamed module, but all of them together act as if they were a single module, which is called *the* unnamed module. As with automatic modules, the unnamed module can access all other modules, and all of its packages are exported and opened.

However, *no explicit module* can access the unnamed module. (An explicit module is a module that is neither automatic nor unnamed; that is, a module with a `module-info.class` on the module path.) In other words, explicit modules are always free from "class path hell".

Consider for example the program of the preceding section. Suppose you put `commons-csv-1.5.jar` onto the class path instead of the module path:

```
java --module-path v2ch09.automod \

    --class-path commons-csv-1.5.jar \

    -m v2ch09.automod/com.horstmann.places.CSVDemo
```

Now the program won't start:

```
Error occurred during initialization of boot layer

java.lang.module.FindException: Module commons.csv not found,
```

http://www.rejoiceblog.com/

Therefore, migration to the Java Platform Module System is necessarily a bottom-up process:

1. The Java platform itself is modularized.

2. Next, libraries are modularized, either by using automatic modules or by turning them explicit modules.

3. Once all libraries used by your application are modularized, you can turn the code of your application into a module.

Note

Automatic modules *can* read the unnamed module, so their dependencies can go onto the class path.

9.10 COMMAND-LINE FLAGS FOR MIGRATION

Even if your programs do not use modules, you cannot escape the modular world when running with Java 9 and beyond. Even if the application code resides on the class path in an unnamed module and all packages are exported and opened, it interacts with the Java platform, which is modularized.

Up to Java 11, the default behavior is to permit illegal module access but to display a warning on the console for the first instance of each offense. In a future version of Java, the default behavior will change, and illegal access with be denied. In order to give you time to prepare for that change, you should test your applications with the `--illegal-access` flag. There are four possible settings:

1. `--illegal-access=permit` is the Java 9 default behavior, printing a message for the first instance of illegal access

2. `--illegal-access=warn` prints a message for each illegal access

3. `--illegal-access=debug` prints a message and stack trace for each illegal access

4. `--illegal-access=deny` is the future default behavior, denying all illegal access

http://www.rejoiceblog.com/

Now is the time to test with `--illegal-access=deny` so that you can be ready when that behavior becomes the default.

Consider an application that uses an internal API that is no longer accessible, such as `com.sun.rowset.CachedRowSetImpl`. The best remedy is to change the implementation. (As of Java 7, you can get a cached row set from a `RowSetProvider`.) But suppose you don't have access to the source code.

In that case, start the application with the `--add-exports` flag. Specify the module and the package that you want to export, and the module to which you want to export the package, which in our case is the unnamed module.

```
java --illegal-access=deny --add-exports java.sql.rowset/com.
```

Now suppose your application uses reflection to access private fields or methods. Reflection inside the unnamed module is ok, but it is no longer possible to reflectively access non-public members of the Java platform classes. For example, some libraries that dynamically generate Java classes call the protected `ClassLoader.defineClass` method through reflection. If an application uses such a library, add the flag

```
--add-opens java.base/java.lang=ALL-UNNAMED
```

When adding all those command line options to get a legacy app to work, you may well end up with the command line from hell. To better manage a multitude of options, you can put options in one or more files which you specify with an `@` prefix. For example,

```
java @options1 @options2 -jar MyProg.java
```

where the files `options1` and `options2` contain options for the `java` command.

There are a few syntax rules for the options files:

• Separate options with spaces, tabs, or newlines.

• Use double quotes around arguments that include spaces, such as `"Program`

http://www.rejoiceblog.com/

`Files"`.

• A line ending in a `\` is merged with the next line.

• Backslashes must be escaped, such as `C:\\Users\\Fred`.

• Comment lines start with `#`.

9.11 TRANSITIVE AND STATIC REQUIREMENTS

In Section 9.4, "Requiring Modules," on p. 504, you have seen the basic form of the `requires` statement. In this section, you will see two variants that are occasionally useful.

In some situation, it can be tedious for a user of a given module to declare all required modules. Consider for example the `javafx.controls` module that contains JavaFX user interface elements such as buttons. The `javafx.controls` requires the `javafx.base` module, and everyone using `javafx.controls` will also need `javafx.base`. (You wouldn't be able to do much with a user interface control such as a `Button` if you didn't have packages from the `javafx.base` module available.) For that reason, the `javafx.controls` module declares the requirement with the `transitive` modifier:

```
module javafx.controls
{
   requires transitive javafx.base;
   ...
}
```

Any module that declares a requirement on `javafx.controls` now automatically requires `javafx.base`.

Note

http://www.rejoiceblog.com/

Some programmers recommend that you should always use `requires transitive` when a package from another module is used in the public API. But that is not a rule of the Java language. Consider for example, the `java.sql` module:

```
module java.sql

{

  requires transitive java.logging;

  ...

}
```

There is a single use of a package from the `java.logging` module in the entire `java.sql` API, namely the `java.sql.Driver.parentLogger` method that returns a `java.util.logging.Logger`. It would have been perfectly acceptable to not declare this module requirement as transitive. Then those modules, and only those, who actually use that method would need to declare that they require `java.logging`.

One compelling use of the `requires transitive` statement is an *aggregator* module—a module with no packages and only transitive requirements. One such module is the `java.se` module, declared like this:

```
module java.se

{

   requires transitive java.compiler;

   requires transitive java.datatransfer;

   requires transitive java.desktop;

   ...
```

http://www.rejoiceblog.com/

```
    requires transitive java.sql;

    requires transitive java.sql.rowset;

    requires transitive java.xml;

    requires transitive java.xml.crypto;

}
```

A programmer who isn't interested in fine-grained module dependencies can simply require `java.se` and get all modules of the Java SE platform.

Finally, there is an uncommon `requires static` variant that states that a module must be present at compile-time but is optional at runtime. There are two use cases:

1. To access an annotation that is processed at compile-time and declared in a different module.

2. To use a class in a different module if it is available, and otherwise do something else, such as:

```
try {

    new oracle.jdbc.driver.OracleDriver();

    ...

} catch (NoClassDefFoundError er) {

    // Do something else

}
```

9.12 QUALIFIED EXPORTING AND OPENING

In this section, you will see a variant of the `exports` and `opens` statement that narrows their scope to a specified set of modules. For example, the `javafx.base`

http://www.rejoiceblog.com/

module contains a statement

```
exports com.sun.javafx.collections to

   javafx.controls, javafx.graphics, javafx.fxml, javafx.swin
```

Such a statement is called a *qualified export*. The listed modules can access the package, but other modules cannot.

Excessive use of qualified exports can indicate a poor modular structure. Nevertheless, they can arise when modularizing an existing code base. Here, the Java platform designers distributed the code for JavaFX into multiple modules, which is a good idea because not all JavaFX applications need FXML

or Swing interoperability. However, the JavaFX implementors liberally used internal classes such as `com.sun.javafx.collections.ListListenerHelper` in their code. In a greenfield project, one can instead design a more robust public API.

Similarly, you can restrict the `opens` statement to specific modules. For example, in Section 9.7, "Modules and Reflective Access," on p. 512 we could have used a qualified `opens` statement, like this:

```
module v2ch09.openpkg

{

   requires com.horstmann.util;

   opens com.horstmann.places to com.horstmann.util;

}
```

Now the `com.horstmann.places` package is only opened to the `com.horstmann.util` module.

9.13 SERVICE LOADING

http://www.rejoiceblog.com/

The `ServiceLoader` class (see Chapter 6 of Volume I) provides a lightweight mechanism for matching up service interfaces with implementations. The Java Platform Module System makes it easier to use this mechanism.

Here is a quick reminder of service loading. A service has an interface and one or more possible implementations. Here is a simple example of a simple interface.

```
public interface GreeterService

{

   String greet(String subject);

   Locale getLocale();

}
```

One or more modules provide implementations, such as

```
public class FrenchGreeter implements GreeterService

{

   public String greet(String subject) { return "Bonjour " +

   public Locale getLocale() { return Locale.FRENCH; }

}
```

The service consumer must pick an implementation among all offered implementations, based on whatever criteria it deems appropriate.

```
ServiceLoader<GreeterService> greeterLoader = ServiceLoader.l

GreeterService chosenGreeter;
```

http://www.rejoiceblog.com/

```
for (GreeterService greeter : greeterLoader)

{

    if (...) {

        chosenGreeter = greeter;

    }

}
```

In the past, implementations were offered by placing text files into the `META-INF/services` directory of the JAR file containing the implementation classes. The module system provides a better approach. Instead of text files, you add statements to the module descriptors.

A module providing an implementation of a service adds a `provides` statement that lists the service interface (which may be defined in any module) and the implementing class (which must be a part of this module). Here is an example from the `jdk.security.auth` module:

```
module jdk.security.auth

{

    ...

    provides javax.security.auth.spi.LoginModule with

        com.sun.security.auth.module.Krb5LoginModule,

        com.sun.security.auth.module.UnixLoginModule,

        com.sun.security.auth.module.JndiLoginModule,

        com.sun.security.auth.module.KeyStoreLoginModule,
```

http://www.rejoiceblog.com/

```
        com.sun.security.auth.module.LdapLoginModule,

        com.sun.security.auth.module.NTLoginModule;

    }
```

This is the equivalent of the `META-INF/services` file.

The consuming modules contain a `uses` statement.

```
    module java.base

    {

        ...

        uses javax.security.auth.spi.LoginModule;

    }
```

When code in a consuming module calls `ServiceLoader.load(`*ServiceInterface*`.class)`, the matching provider classes will be loaded, even though they may not be in accessible packages.

In our code example, we provide implementations for a German and French greeter in the package `com.horstmann.greetsvc.internal`. The service module exports the `com.horstmann.greetsvc` package, but not the package with the implementations. The `provides` statement declares the service and its implementing classes in the unexported package:

```
    module com.horstmann.greetsvc

    {

        exports com.horstmann.greetsvc;
```

http://www.rejoiceblog.com/

```
    provides com.horstmann.greetsvc.GreeterService with

      com.horstmann.greetsvc.internal.FrenchGreeter,

      com.horstmann.greetsvc.internal.GermanGreeterFactory;

  }
```

The `v2ch09.useservice` module consumes the service. Using the `ServiceLoader` facility, we iterate over the provided services and pick the one matching the desired language:

```
  package com.horstmann.hello;



  import java.util.*;

  import com.horstmann.greetsvc.*;



  public class HelloWorld

  {

     public static void main(String[] args)

     {

       ServiceLoader<GreeterService> greeterLoader

          = ServiceLoader.load(GreeterService.class);

       String desiredLanguage = args.length > 0 ? args[0] : "de

       GreeterService chosenGreeter = null;
```

http://www.rejoiceblog.com/

```
        for (GreeterService greeter : greeterLoader)
        {
            if (greeter.getLocale().getLanguage().equals(desiredL
                chosenGreeter = greeter;
        }
        if (chosenGreeter == null)
          System.out.println("No suitable greeter.");
        else
          System.out.println(chosenGreeter.greet("Modular World"
      }
    }
```

The module declaration requires the service module and declares that the `GreeterService` is being used.

```
module v2ch09.useservice
{
    requires com.horstmann.greetsvc;
    uses com.horstmann.greetsvc.GreeterService;
}
```

As a result of the `provides` and `uses` declarations, the module that consumes the

http://www.rejoiceblog.com/
service is allowed access to the private implementation classes.

To build and run the program, first compile the service:

```
javac com.horstmann.greetsvc/module-info.java \
   com.horstmann.greetsvc/com/horstmann/greetsvc/GreeterServi
   com.horstmann.greetsvc/com/horstmann/greetsvc/internal/*.j
```

Then compile and run the consuming module:

```
javac -p com.horstmann.greetsvc \
   v2ch09.useservice/com/horstmann/hello/HelloWorld.java \
   v2ch09.useservice/module-info.java
java -p com.horstmann.greetsvc:v2ch09.useservice \
   -m v2ch09.useservice/com.horstmann.hello.HelloWorld
```

9.14 TOOLS FOR WORKING WITH MODULES

The `jdeps` tool analyzes the dependencies of a given set of JAR files. Suppose, for example, you want to modularize JUnit 4. Run

```
jdeps -s junit-4.12.jar hamcrest-core-1.3.jar
```

The `-s` flag generates a summary output:

```
hamcrest-core-1.3.jar -> java.base
junit-4.12.jar -> hamcrest-core-1.3.jar
```

http://www.rejoiceblog.com/

```
junit-4.12.jar -> java.base

junit-4.12.jar -> java.management
```

That tells you the module graph:

If you omit the `-s` flag, you get the module summary followed by a mapping from packages to required packages and modules. If you add the `-v` flag, the listing maps classes to required packages and modules.

The `--generate-module-info` option produces `module-info` files for each analyzed module:

```
jdeps --generate-module-info /tmp/junit junit-4.12.jar hamcre
```

Note

There is also an option to generate graphical output in the “dot” language for describing graphs. Assuming you have the dot tool installed, run these commands:

```
jdeps -s -dotoutput /tmp/junit junit-4.12.jar hamcrest-core-1.3.jar

dot -Tpng /tmp/junit/summary.dot > /tmp/junit/summary.png
```

Then you get this `summary.png` image:

http://www.rejoiceblog.com/

You use the `jlink` tool to produce an application that executes without a separate Java runtime. The resulting image is much smaller than the entire JDK. You specify the modules that you want to have included and an output directory.

```
jlink --module-path com.horstmann.greet.jar:v2ch09.exportedpk

   --add-modules v2ch09.exportedpkg --output /tmp/hello
```

The output directory has a subdirectory `bin` with a `java` executable. If you run

```
bin/java -m v2ch09.exportedpkg
```

the `main` method of the module's main class is invoked.

The point of `jlink` is that it bundles up the minimal set of modules that is required to run the application. You can list them all:

```
bin/java --list-modules
```

In this example, the output is

```
v2ch09.exportedpkg

com.horstmann.greet

java.base@9
```

http://www.rejoiceblog.com/

All modules are included in a *runtime image* file `lib/modules`. On my computer, that file is 23MB, whereas the runtime image of all JDK modules take up 181MB. The entire application takes up 45MB, less than 10% of the JDK which is 486MB.

This can be the basis of a useful tool for packaging an application. You would still need to produce file sets for multiple platforms and launch scripts for the application.

Note

You can inspect the runtime image with the `jimage` command. However, the format is internal to the JVM, and runtime images are not meant to be generated or used by other tools.

Finally, the `jmod` tool builds and inspects the module files that are included with the JDK. When you look into the `jmods` directory inside the JDK, you will find a file with extension `jmod` for each module. There is no longer a `rt.jar` file.

Like JAR files, these files contain class files. In addition, they can hold native code libraries, commands, header files, configuration files, and legal notices. The JMOD files use the ZIP format. You can inspect their contents with any ZIP tool.

Unlike JAR files, JMOD files are only useful for linking; that is, for producing runtime images. There is no need for you to produce JMOD files unless you also want to bundle binary files such as native code libraries with your modules.

Note

Since the `rt.jar` and `tools.jar` files are no longer included with Java 9, you need to update any references to them. For example, if you referred to `tools.jar` in a security policy file, change it to a reference to the module:

```
grant codeBase "jrt:/jdk.compiler" {

  permission java.security.AllPermission;
```

http://www.rejoiceblog.com/

```
};
```

The `jrt:` syntax denotes the Java runtime file.

This brings us to the end of the chapter on the Java Platform Module System. The following chapter covers another important topic: security. Security has always been a core feature of the Java platform. As the world in which we live and compute gets more dangerous, a thorough understanding of Java security will be of increasing importance for many developers.

http://www.rejoiceblog.com/

Chapter 10 Security

In this chapter

• 10.1 Class Loaders

• 10.2 Security Managers and Permissions

• 10.3 User Authentication

• 10.4 Digital Signatures

• 10.5 Encryption

When Java technology first appeared on the scene, the excitement was not about a well-crafted programming language but about the possibility of safely executing applets delivered over the Internet. Obviously, serving executable applets is only practical when the recipients can be sure the code won't wreak havoc on their machines. Security therefore was and is a major concern of both the designers and the users of Java technology. This means that unlike other languages and systems, where security was implemented as an afterthought or as a reaction to break-ins, security mechanisms are an integral part of Java technology.

Three mechanisms help ensure safety:

• Language design features (bounds checking on arrays, no unchecked type conversions, no pointer arithmetic, and so on).

• An access control mechanism that controls what the code can do (file access, network access, and so on).

• Code signing, whereby code authors can use standard cryptographic algorithms to authenticate Java code. Then, the users of the code can determine exactly who created the code and whether the code has been altered after it was signed.

We will first discuss *class loaders* that check class files for integrity when they are loaded into the virtual machine. We will demonstrate how that mechanism can detect tampering with class files.

For maximum security, both the default mechanism for loading a class and a custom class loader need to work with a *security manager* class that controls what actions code can perform. You'll see in detail how to configure Java platform security.

Finally, you'll see the cryptographic algorithms supplied in the `java.security` package,

http://www.rejoiceblog.com/

which allow for code signing and user authentication.

As always, we'll focus on those topics that are of greatest interest to application programmers. For an in-depth view, we recommend the book *Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation, Second Edition*, by Li Gong, Gary Ellison, and Mary Dageforde (Prentice Hall, 2003).

10.1 CLASS LOADERS

A Java compiler converts source instructions into code for the Java virtual machine. The virtual machine code is stored in a class file with a `.class` extension. Each class file contains the definition and implementation code for one class or interface. In the following section, you will see how the virtual machine loads these class files.

10.1.1 The Class Loading Process

The virtual machine loads only those class files that are needed for the execution of a program. For example, suppose program execution starts with `MyProgram.class`. Here are the steps that the virtual machine carries out:

1. The virtual machine has a mechanism for loading class files—for example, by reading the files from disk or by requesting them from the Web; it uses this mechanism to load the contents of the `MyProgram` class file.

2. If the `MyProgram` class has fields or superclasses of another class type, their class files are loaded as well. (The process of loading all the classes that a given class depends on is called *resolving* the class.)

3. The virtual machine then executes the `main` method in `MyProgram` (which is static, so no instance of a class needs to be created).

4. If the `main` method or a method that `main` calls requires additional classes, these are loaded next.

The class loading mechanism doesn't just use a single class loader, however. Every Java program has at least three class loaders:

• The bootstrap class loader

• The platform class loader

• The system class loader (sometimes called the application class loader)

The bootstrap class loader loads the platform classes contained in the modules

```
java.base

java.datatransfer
```

http://www.rejoiceblog.com/

```
java.desktop

java.instrument

java.logging

java.management

java.management.rmi

java.naming

java.prefs

java.rmi

java.security.sasl

java.xml
```

as well as a number of JDK-internal modules.

There is no `ClassLoader` object corresponding to the bootstrap class loader. For example,

```
StringBuilder.class.getClassLoader()
```

returns `null`.

Prior to Java 9, the Java platform classes were located in a file `rt.jar`. Nowadays, the Java platform is modular, and each platform module is contained in a JMOD file (see Chapter 9). The platform class loader loads all classes of the Java platform that are not loaded by the bootstrap class loader.

The system class loader loads application classes from the module path and class path.

Note

Prior to Java 9, an “extension class loader” loaded “standard extensions” from the `jre/lib/ext` directory, and an “endorsed standards override” mechanism provided a way of overriding certain platform classes (including the CORBA and XML implementations) with newer versions. Both of these mechanisms have been removed.

http://www.rejoiceblog.com/

10.1.2 The Class Loader Hierarchy

Class loaders have a *parent/child* relationship. Every class loader except for the bootstrap one has a parent class loader. A class loader is supposed to give its parent a chance to load any given class and to only load it if the parent has failed. For example, when the system class loader is asked to load a system class (say, `java.lang.StringBuilder`), it first asks the platform class loader. That class loader first asks the bootstrap class loader. The bootstrap class loader finds and loads the class, so neither of the other class loaders searches any further.

Some programs have a plugin architecture in which certain parts of the code are packaged as optional plugins. If the plugins are packaged as JAR files, you can simply load the plugin classes with an instance of `URLClassLoader`.

```
var url = new URL("file:///path/to/plugin.jar");

var pluginLoader = new URLClassLoader(new URL[] { url });

Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

Since no parent was specified in the `URLClassLoader` constructor, the parent of the `pluginLoader` is the system class loader. Figure 10.1 shows the hierarchy.

Caution

Prior to Java 9, the system class loader was an instance of `URLClassLoader`. Some programmers used a cast to access the `getURLs` method, or added JAR files to the class path by calling the protected `addURLs` method through reflection. This is no longer possible.

Most of the time, you don't have to worry about the class loader hierarchy. Generally, classes are loaded because they are required by other classes, and that process is transparent to you.

Occasionally, however, you need to intervene and specify a class loader. Consider this example:

- Your application code contains a helper method that calls `Class.forName(classNameString)`.
- That method is called from a plugin class.
- The `classNameString` specifies a class that is contained in the plugin JAR.

The author of the plugin wants the class to be loaded. However, the helper method's class was

http://www.rejoiceblog.com/

loaded by the system class loader, and that is the class loader used by `Class.forName`. The classes in the plugin JAR are not visible. This phenomenon is called *classloader inversion*.

Figure 10.1 The class loader hierarchy

To overcome this problem, the helper method needs to use the correct class loader. It can require the class loader as a parameter. Alternatively, it can require that the correct class loader is set as the *context class loader* of the current thread. This strategy is used by many frameworks (such as JAXP and JNDI).

Each thread has a reference to a class loader, called the context class loader. The main thread's context class loader is the system class loader. When a new thread is created, its context class loader is set to the creating thread's context class loader. Thus, if you don't do anything, all threads will have their context class loaders set to the system class loader.

However, you can set any class loader by calling

```
Thread t = Thread.currentThread();

t.setContextClassLoader(loader);
```

http://www.rejoiceblog.com/

The helper method can then retrieve the context class loader:

```
Thread t = Thread.currentThread();

ClassLoader loader = t.getContextClassLoader();

Class<?> cl = loader.loadClass(className);
```

Tip

If you write a method that loads a class by name, it is a good idea to offer the caller the choice between passing an explicit class loader and using the context class loader. Don't simply use the class loader of the method's class.

10.1.3 Using Class Loaders as Namespaces

Every Java programmer knows that package names are used to eliminate name conflicts. There are two classes called `Date` in the standard library, but of course their real names are `java.util.Date` and `java.sql.Date`. The simple name is only a programmer convenience and requires the inclusion of appropriate `import` statements. In a running program, all class names contain their package names.

It might surprise you, however, that you can have two classes in the same virtual machine that have the same class *and package* name. A class is determined by its full name *and* the class loader. This technique is useful for loading code from multiple sources. For example, an application server uses separate class loaders for each application. This allows the virtual machine to separate classes from different applications, no matter what they are named. Figure 10.2 shows an example. Suppose an application server loads two different applications, and each has a class called `Util`. Since each class is loaded by a separate class loader, these classes are entirely distinct and do not conflict with each other.

http://www.rejoiceblog.com/

Figure 10.2 Two class loaders load different classes with the same name.

10.1.4 Writing Your Own Class Loader

You can write your own class loader for specialized purposes. That lets you carry out custom checks before you pass the bytecodes to the virtual machine. For example, your class loader can refuse to load a class that has not been marked as "paid for."

To write your own class loader, simply extend the `ClassLoader` class and override the method

```
findClass(String className)
```

The `loadClass` method of the `ClassLoader` superclass takes care of the delegation to the parent and calls `findClass` only if the class hasn't already been loaded and if the parent class loader was unable to load the class.

http://www.rejoiceblog.com/

Your implementation of this method must do the following:

1. Load the bytecodes for the class from the local file system or some other source.

2. Call the `defineClass` method of the `ClassLoader` superclass to present the bytecodes to the virtual machine.

In the program of Listing 10.1, we implement a class loader that loads encrypted class files. The program asks the user for the name of the first class to load (that is, the class containing `main`) and the decryption key. It then uses a special class loader to load the specified class and calls the `main` method. The class loader decrypts the specified class and all nonsystem classes that are referenced by it. Finally, the program calls the `main` method of the loaded class (see Figure 10.3).

Figure 10.3 The `ClassLoaderTest` program

For simplicity, we ignore the 2,000 years of progress in the field of cryptography and use the venerable Caesar cipher for encrypting the class files.

> **Note**
>
> David Kahn's wonderful book *The Codebreakers* (Macmillan, 1967, p. 84) refers to Suetonius as a historical source for the Caesar cipher. Caesar shifted the 24 letters of the Roman alphabet by 3 letters, which at the time baffled his adversaries.
>
> When this chapter was first written, the U.S. government restricted the export of strong encryption methods. Therefore, we used Caesar's method for our example because it was clearly legal for export.

Our version of the Caesar cipher has as a key a number between 1 and 255. To decrypt, simply add that key to every byte and reduce modulo 256. The `Caesar.java` program of Listing 10.2 carries out the encryption.

To not confuse the regular class loader, we use a different extension, `.caesar`, for the

http://www.rejoiceblog.com/

encrypted class files.

To decrypt, the class loader simply subtracts the key from every byte. In the companion code for this book, you will find four class files, encrypted with a key value of 3—the traditional choice. To run the encrypted program, you'll need the custom class loader defined in our `ClassLoaderTest` program.

Encrypting class files has a number of practical uses (provided, of course, that you use something stronger than the Caesar cipher). Without the decryption key, the class files are useless. They can neither be executed by a standard virtual machine nor readily disassembled.

This means that you can use a custom class loader to authenticate the user of the class or to ensure that a program has been paid for before it will be allowed to run. Of course, encryption is only one application of a custom class loader. You can use other types of class loaders to solve other problems—for example, storing class files in a database.

Listing 10.1 `classLoader/ClassLoaderTest.java`

```
package classLoader;

import   java.io.*;
import   java.lang.reflect.*;
import   java.nio.file.*;
import   java.awt.*;
import   java.awt.event.*;
import   javax.swing.*;

/**
 * This program demonstrates a custom class loader that decry
 * @version 1.25 2018-05-01
 * @author Cay Horstmann
 */
```

http://www.rejoiceblog.com/

```
15  public class ClassLoaderTest
16  {
17      public static void main(String[] args)
18      {
19         EventQueue.invokeLater(() ->
20            {
21               var frame = new ClassLoaderFrame();
22               frame.setTitle("ClassLoaderTest");
23               frame.setDefaultCloseOperation(JFrame.EXIT_ON_
24               frame.setVisible(true);
25            });
26      }
27  }
28
29  /**
30   * This frame contains two text fields for the name of the cl
31   */
32  class ClassLoaderFrame extends JFrame
33  {
34      private JTextField keyField = new JTextField("3", 4);
35      private JTextField nameField = new JTextField("Calculato
36      private static final int DEFAULT_WIDTH = 300;
37      private static final int DEFAULT_HEIGHT = 200;
```

http://www.rejoiceblog.com/

```
38
39      public ClassLoaderFrame()
40      {
41         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
42         setLayout(new GridBagLayout());
43         add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EA
44         add(nameField, new GBC(1, 0).setWeight(100, 0).setAncho
45         add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST
46         add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor
47         var loadButton = new JButton("Load");
48         add(loadButton, new GBC(0, 2, 2, 1));
49         loadButton.addActionListener(event -> runClass(nameFiel
50         pack();
51      }
52
53      /**
54       * Runs the main method of a given class.
55       * @param name the class name
56       * @param key the decryption key for the class files
57       */
58      public void runClass(String name, String key)
59      {
60          try
```

http://www.rejoiceblog.com/

```
            {
                var loader = new CryptoClassLoader(Integer.parse
                Class<?> c = loader.loadClass(name);
                Method m = c.getMethod("main", String[].class);
                m.invoke(null, (Object) new String[] {});
            }
            catch (Throwable t)
            {
               JOptionPane.showMessageDialog(this, t);
            }
        }

    }

    /**
     * This class loader loads encrypted class files.
     */
    class CryptoClassLoader extends ClassLoader
    {
         private int key;

        /**
         * Constructs a crypto class loader.
```

http://www.rejoiceblog.com/

```
84          * @param k the decryption key
85          */
86         public CryptoClassLoader(int k)
87         {
88              key = k;
89         }
90
91         protected Class<?> findClass(String name) throws ClassNo
92         {
93            try
94            {
95                byte[] classBytes = null;
96                classBytes = loadClassBytes(name);
97                Class<?> cl = defineClass(name, classBytes, 0, cl
98                if (cl == null) throw new ClassNotFoundException
99                return cl;
100           }
101           catch (IOException e)
102           {
103               throw new ClassNotFoundException(name);
104           }
105        }
106
```

http://www.rejoiceblog.com/

```
    /**
     * Loads and decrypt the class file bytes.
     * @param name the class name
     * @return an array with the class file bytes
     */
    private byte[] loadClassBytes(String name) throws IOExce
    {
        String cname = name.replace('.', '/') + ".caesar";
        byte[] bytes = Files.readAllBytes(Paths.get(cname))
        for (int i = 0; i < bytes.length; i++)
           bytes[i] = (byte) (bytes[i] - key);
        return bytes;
    }
}
```

Listing 10.2 classLoader/Caesar.java

```
package classLoader;

import java.io.*;

/**
 * Encrypts a file using the Caesar cipher.
 * @version 1.02 2018-05-01
```

http://www.rejoiceblog.com/

```
 8     * @author Cay Horstmann
 9     */
10    public class Caesar
11    {
12        public static void main(String[] args) throws Exception
13        {
14           if (args.length != 3)
15           {
16              System.out.println("USAGE: java classLoader.Caesa
17              return;
18           }
19
20            try(var in = new FileInputStream(args[0]);
21               var out = new FileOutputStream(args[1]))
22            {
23               int key = Integer.parseInt(args[2]);
24               int ch;
25               while ((ch = in.read()) != -1)
26               {
27                  byte c = (byte) (ch + key);
28                  out.write(c);
29               }
30            }
```

http://www.rejoiceblog.com/

```
        }
    }
```

java.lang.Class 1.0

- `ClassLoader getClassLoader()`

gets the class loader that loaded this class.

java.lang.ClassLoader 1.0

- `ClassLoader getParent()` **1.2**

returns the parent class loader, or `null` if the parent class loader is the bootstrap class loader.

- `static ClassLoader getSystemClassLoader()` **1.2**

gets the system class loader—that is, the class loader that was used to load the first application class.

- `protected Class findClass(String name)` **1.2**

should be overridden by a class loader to find the bytecodes for a class and present them to the virtual machine by calling the `defineClass` method. In the name of the class, use `.` as package name separator, and don't use a `.class` suffix.

- `Class defineClass(String name, byte[] byteCodeData, int offset, int length)`

adds a new class to the virtual machine whose bytecodes are provided in the given data range.

java.net.URLClassLoader 1.2

- `URLClassLoader(URL[] urls)`

- `URLClassLoader(URL[] urls, ClassLoader parent)`

constructs a class loader that loads classes from the given URLs. If a URL ends in a `/`, it is assumed to be a directory, otherwise it is assumed to be a JAR file.

http://www.rejoiceblog.com/

java.lang.Thread 1.0

- `ClassLoader getContextClassLoader()` 1.2

gets the class loader that the creator of this thread has designated as the most reasonable class loader to use when executing this thread.

- `void setContextClassLoader(ClassLoader loader)` 1.2

sets a class loader for code in this thread to retrieve for loading classes. If no context class loader is set explicitly when a thread is started, the parent's context class loader is used.

10.1.5 Bytecode Verification

When a class loader presents the bytecodes of a newly loaded Java platform class to the virtual machine, these bytecodes are first inspected by a *verifier*. The verifier checks that the instructions cannot perform actions that are obviously damaging. All classes except for system classes are verified.

Here are some of the checks that the verifier carries out:

- Variables are initialized before they are used.
- Method calls match the types of object references.
- Rules for accessing private data and methods are not violated.
- Local variable accesses fall within the runtime stack.
- The runtime stack does not overflow.

If any of these checks fails, the class is considered corrupted and will not be loaded.

Note

If you are familiar with Gödel's theorem, you might wonder how the verifier can prove that a class file is free from type mismatches, uninitialized variables, and stack overflows. Gödel's theorem states that it is impossible to design algorithms that process program files and decide whether the input programs have a particular property (such as being free from stack overflows). Is this a conflict between the public relations department at Oracle and the laws of logic? No—in fact, the verifier is *not* a decision algorithm in the sense of Gödel. If the verifier accepts a program, it is indeed safe. However, the verifier might reject virtual machine instructions even though they would actually be safe. (You might have run into this issue when you were forced to initialize a variable with a dummy value because the verifier couldn't

http://www.rejoiceblog.com/

see that it was going to be properly initialized.)

This strict verification is an important security consideration. Accidental errors, such as uninitialized variables, can easily wreak havoc if they are not caught. More importantly, in the wide open world of the Internet, you must be protected against malicious programmers who create evil effects on purpose. For example, by modifying values on the runtime stack or by writing to the private data fields of system objects, a program can break through the security system of a browser.

You might wonder, however, why a special verifier checks all these features. After all, the compiler would never allow you to generate a class file in which an uninitialized variable is used or in which a private data field is accessed from another class. Indeed, a class file generated by a compiler for the Java programming language always passes verification. However, the bytecode format used in the class files is well documented, and it is an easy matter for someone with experience in assembly programming and a hex editor to manually produce a class file containing valid but unsafe instructions for the Java virtual machine. The verifier is always guarding against maliciously altered class files, not just checking the class files produced by a compiler.

Here's an example of how to construct such an altered class file. We start with the program `VerifierTest.java` of Listing 10.3. This is a simple program that calls a method and displays the method's result. The program can be run both as a console program and as an applet. The `fun` method itself just computes 1 + 2.

```
static int fun()

{

   int m;

   int n;

   m = 1;

   n = 2;

   int r = m + n;

   return r;

}
```

As an experiment, try to compile the following modification of this program:

http://www.rejoiceblog.com/

```
static int fun()

{

   int m = 1;

   int n;

   m = 1;

   m = 2;

   int r = m + n;

   return r;

}
```

In this case, `n` is not initialized, and it could have any random value. Of course, the compiler detects that problem and refuses to compile the program. To create a bad class file, we have to work a little harder. First, run the `javap` program to find out how the compiler translates the `fun` method. The command

```
javap -c verifier.VerifierTest
```

shows the bytecodes in the class file in mnemonic form.

```
Method int fun()

   0 iconst_1

   1 istore_0

   2 iconst_2

   3 istore_1

   4 iload_0

   5 iload_1

   6 iadd

   7 istore_2
```

http://www.rejoiceblog.com/

```
8 iload_2

9 ireturn
```

Use a hex editor to change instruction 3 from `istore_1` to `istore_0`. That is, local variable 0 (which is `m`) is initialized twice, and local variable 1 (which is `n`) is not initialized at all. We need to know the hexadecimal values for these instructions; these values are readily available from *The Java™ Virtual Machine Specification, Second Edition*, by Tim Lindholm and Frank Yellin (Prentice Hall, 1999).

```
0 iconst_1 04

1 istore_0 3B

2 iconst_2 05

3 istore_1 3C

4 iload_0 1A

5 iload_1 1B

6 iadd 60

7 istore_2 3D

8 iload_2 1C

9 ireturn AC
```

You can use any hex editor to carry out the modification. In Figure 10.4, you see the class file `VerifierTest.class` loaded into the Gnome hex editor, with the bytecodes of the `fun` method highlighted.

http://www.rejoiceblog.com/

Figure 10.4 Modifying bytecodes with a hex editor

Change `3C` to `3B` and save the class file. Then try running the `VerifierTest` pro-gram. You get an error message:

```
Exception in thread "main" java.lang.VerifyError: (class: Verifie

()I) Accessing value from uninitialized register 1
```

That is good—the virtual machine detected our modification.

Now run the program with the `-noverify` (or `-Xverify:none`) option.

```
java -noverify verifier.VerifierTest
```

The `fun` method returns a seemingly random value. This is actually `2` plus the value that happened to be stored in the variable `n`, which never was initialized. Here is a typical printout:

```
1 + 2 == 15102330
```

Listing 10.3 `verifier/VerifierTest.java`

http://www.rejoiceblog.com/

```
1   package verifier;
2
3   import java.awt.*;
4
5   /**
6    * This application demonstrates the bytecode verifier of the
7    * editor to modify the class file, then the virtual machine
8    * @version 1.10 2018-05-05
9    * @author Cay Horstmann
10   */
11  public class VerifierTest
12  {
13       public static void main(String[] args)
14       {
15          System.out.println("1 + 2 == " + fun());
16       }
17
18      /**
19       * A function that computes 1 + 2.
20       * @return 3, if the code has not been corrupted
21       */
22      public static int fun()
23      {
```

http://www.rejoiceblog.com/

```
        int m;
        int n;
        m = 1;
        n = 2;
        // use hex editor to change to "m = 2" in class file
        int r = m + n;
        return r;
    }
}
```

10.2 SECURITY MANAGERS AND PERMISSIONS

Once a class has been loaded into the virtual machine and checked by the verifier, the second security mechanism of the Java platform springs into action: the *security manager*. This is the topic of the following sections.

10.2.1 Permission Checking

The security manager controls whether a specific operation is permitted. Operations checked by the security manager include the following:

• Creating a new class loader

• Exiting the virtual machine

• Accessing a field of another class by using reflection

• Accessing a file

• Opening a socket connection

• Starting a print job

• Accessing the system clipboard

• Accessing the AWT event queue

• Bringing up a top-level window

http://www.rejoiceblog.com/

There are many other checks throughout the Java library.

The default behavior when running Java applications is to install *no* security manager, so all these operations are permitted. In contrast, applets relied on a very restrictive security policy. Stricter security also makes sense in other situations.

For example, suppose you run a Tomcat instance and allow collaborators or students to install servlets. You would not want any of them to call `System.exit` since this would terminate the Tomcat instance. You can set a security policy that causes calls to `System.exit` to throw a security exception instead of actually closing down the virtual machine. Here is what happens in detail. The `exit` method of the `Runtime` class calls the `checkExit` method of the security manager. Here is the entire code of the `exit` method:

```
public void exit(int status)

{

   SecurityManager security = System.getSecurityManager();

   if (security != null)

     security.checkExit(status);

   exitInternal(status);

}
```

The security manager now checks if the exit request came from the browser or an individual applet. If the security manager agrees with the exit request, the `checkExit` method simply returns and normal processing continues. However, if the security manager doesn't want to grant the request, the `checkExit` method throws a `SecurityException`.

The `exit` method continues only if no exception occurred. It then calls the *private native* `exitInternal` method that actually terminates the virtual machine. There is no other way to terminate the virtual machine, and since the `exitInternal` method is private, it cannot be called from any other class. Thus, any code that attempts to exit the virtual machine must go through the `exit` method and thus through the `checkExit` security check without triggering a security exception.

Clearly, the integrity of the security policy depends on careful coding. The providers of system services in the standard library must always consult the security manager before attempting any sensitive operation.

The security manager of the Java platform allows both programmers and system administrators fine-grained control over individual security permissions. We will describe these features in the following section. First, we'll summarize the Java 2 platform security

http://www.rejoiceblog.com/

model. We'll then show how you can control permissions with *policy files*. Finally, we'll explain how you can define your own permission types.

10.2.2 Java Platform Security

JDK 1.0 had a very simple security model: Local classes had full permissions, and remote classes were confined to the *sandbox*. Just like a child that can only play in a sandbox, remote code was only allowed to paint on the screen and interact with the user. The applet security manager denied all access to local resources. JDK 1.1 implemented a slight modification: Remote code that was signed by a trusted entity was granted the same permissions as local classes. However, both versions of the JDK used an all-or-nothing approach. Programs either had full access or they had to play in the sandbox.

Starting with Java 1.2, the Java platform has a much more flexible mechanism. A *security policy* maps *code sources to permission sets* (see Figure 10.5).

Figure 10.5 A security policy

A *code source* is specified by a *code base* and a set of *certificates*. The code base specifies the origin of the code. For example, the code base of remote applet code is the HTTP URL from which the applet was loaded. The code base of code in a JAR file is the file's URL. A certificate, if present, is an assurance by some party that the code has not been tampered with. We'll cover certificates later in this chapter.

A *permission* is any property that is checked by a security manager. The Java platform supports a number of permission classes, each encapsulating the details of a particular permission. For example, the following instance of the `FilePermission` class states that it is okay to read and write any file in the `/tmp` directory:

http://www.rejoiceblog.com/

```
var p = new FilePermission("/tmp/*", "read,write");
```

More importantly, the default implementation of the `Policy` class reads permissions from a *permission file*. Inside a permission file, the same read permission is expressed as

```
permission java.io.FilePermission "/tmp/*", "read,write";
```

We'll discuss permission files in the next section.

Figure 10.6 shows the hierarchy of the permission classes that were supplied with Java 1.2. Many more permission classes have been added in subsequent Java releases.

Figure 10.6 A part of the hierarchy of permission classes

In the preceding section, you saw that the `SecurityManager` class has security check methods such as `checkExit`. These methods exist only for the convenience of the programmer and for backward compatibility. They all map into standard permission checks. For example, here is the source code for the `checkExit` method:

```
public void checkExit()

{

   checkPermission(new RuntimePermission("exitVM"));

}
```

http://www.rejoiceblog.com/

Each class has a *protection domain*—an object that encapsulates both the code source and the collection of permissions of the class. When the `SecurityManager` needs to check a permission, it looks at the classes of all methods currently on the call stack. It then gets the protection domains of all classes and asks each protection domain if its permission collection allows the operation currently being checked. If all domains agree, the check passes. Otherwise, a `SecurityException` is thrown.

Why do all methods on the call stack need to allow a particular operation? Let us work through an example. Suppose the `init` method of a servlet wants to open a file. It might call

```
var in = new FileReader(name);
```

The `FileReader` constructor calls the `FileInputStream` constructor, which calls the `checkRead` method of the security manager, which finally calls `checkPermission` with a `FilePermission(name, "read")` object. Table 10.1 shows the call stack.

Table 10.1 Call Stack During Permission Checking

Class	Method	Code Source	Permissions
SecurityManager	checkPermission	null	AllPermission
SecurityManager	checkRead	null	AllPermission
FileInputStream	Constructor	null	AllPermission
FileReader	Constructor	null	AllPermission
Servlet	init	Servlet code source	Tomcat web app permissions

The `FileInputStream` and `SecurityManager` classes are *system classes* for which `CodeSource` is `null` and the permissions consist of an instance of the `AllPermission` class, which allows all operations. Clearly, their permissions alone can't determine the outcome of the check. As you can see, the `checkPermission` method must take into account the restricted permissions of the applet class. By checking the entire call stack, the security mechanism ensures that one class can never ask another class to carry out a sensitive operation on its behalf.

Note

This brief discussion of permission checking explains the basic concepts. However, we omit a number of technical details here. With security, the devil lies in the details, and we encourage you to read the book by Li Gong for more information. For a more critical view of the Java platform's security model, see the book *Securing Java: Getting Down to Business with Mobile Code, Second Edition*, by Gary McGraw and Ed W. Felten (Wiley, 1999). You can find an online version of that book at `www.securingjava.com`.

http://www.rejoiceblog.com/

java.lang.SecurityManager 1.0

- `void checkPermission(Permission p)` 1.2

checks whether this security manager grants the given permission. The method throws a `SecurityException` if the permission is not granted.

java.lang.Class 1.0

- `ProtectionDomain getProtectionDomain()` 1.2

gets the protection domain for this class, or `null` if this class was loaded without a protection domain.

java.security.ProtectionDomain 1.2

- `ProtectionDomain(CodeSource source, PermissionCollection permissions)`

constructs a protection domain with the given code source and permissions.

- `CodeSource getCodeSource()`

gets the code source of this protection domain.

- `boolean implies(Permission p)`

returns `true` if the given permission is allowed by this protection domain.

java.security.CodeSource 1.2

- `Certificate[] getCertificates()`

gets the certificate chain for class file signatures associated with this code source.

- `URL getLocation()`

gets the code base of class files associated with this code source.

10.2.3 Security Policy Files

http://www.rejoiceblog.com/

The *policy manager* reads *policy files* that contain instructions for mapping code sources to permissions. Here is a typical policy file:

```
grant codeBase "http://www.horstmann.com/classes"

{

   permission java.io.FilePermission "/tmp/*", "read,write";

};
```

This file grants permission to read and write files in the `/tmp` directory to all code that was downloaded from `www.horstmann.com/classes`.

You can install policy files in standard locations. By default, there are two locations:

• The file `java.policy` in the Java platform's home directory

• The file `.java.policy` (notice the period at the beginning of the file name) in the user's home directory

Note

You can change the locations of these files in the `java.security` configuration file in the `jdk/conf/security` directory. The defaults are specified as

```
policy.url.1=file:${java.home}/lib/security/java.policy

policy.url.2=file:${user.home}/.java.policy
```

A system administrator can modify the `java.security` file and specify policy URLs that reside on another server and cannot be edited by users. There can be any number of policy URLs (with consecutive numbers) in the policy file. The permissions of all files are combined.

If you want to store policies outside the file system, you can implement a subclass of the `Policy` class that gathers the permissions. Then change the line

```
policy.provider=sun.security.provider.PolicyFile
```

in the `java.security` configuration file.

During testing, we don't like to constantly modify the standard policy files. Therefore, we

http://www.rejoiceblog.com/

prefer to explicitly name the policy file required for each application. Place the permissions into a separate file—say, `MyApp.policy`. To apply the policy, you have two choices. You can set a system property inside your applications' `main` method:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

Alternatively, you can start the virtual machine as

```
java -Djava.security.policy=MyApp.policy MyApp
```

In these examples, the `MyApp.policy` file is added to the other policies in effect. If you add a second equal sign, such as

```
java -Djava.security.policy==MyApp.policy MyApp
```

then your application will use *only* the specified policy file, and the standard policy files will be ignored.

Caution

An easy mistake during testing is to accidentally leave a `.java.policy` file that grants a lot of permissions, perhaps even `AllPermission`, in the home directory. If you find that your application doesn't seem to pay attention to the restrictions in your policy file, check for a left-behind `.java.policy` file in your home directory. If you use a UNIX system, this is a particularly easy mistake to make because files with names that start with a period are not displayed by default.

As you saw previously, Java applications by default do not install a security manager. Therefore, you won't see the effect of policy files until you install one. You can, of course, add a line

```
System.setSecurityManager(new SecurityManager());
```

into your `main` method. Or you can add the command-line option `-Djava.security.manager` when starting the virtual machine.

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy M
```

http://www.rejoiceblog.com/

In the remainder of this section, we'll show you in detail how to describe permissions in the policy file. We'll describe the entire policy file format, except for code certificates which we cover later in this chapter.

A policy file contains a sequence of `grant` entries. Each entry has the following form:

```
grant codesource
{
    permission1;
    permission2;
    . . .
};
```

The code source contains a code base (which can be omitted if the entry applies to code from all sources) and the names of trusted principals and certificate signers (which can be omitted if signatures are not required for this entry).

The code base is specified as

```
codeBase "url"
```

If the URL ends in a `/`, it refers to a directory. Otherwise, it is taken to be the name of a JAR file. For example,

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . };
```

The code base is a URL and should always contain forward slashes as file separators, even for file URLs in Windows. For example,

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```

Note

http://www.rejoiceblog.com/

Everyone knows that `http` URLs start with two slashes (`http://`). But there is enough confusion about `file` URLs, so the policy file reader accepts two forms of file URLs—namely, `file://localFile` and `file:localFile`. Furthermore, a slash before a Windows drive letter is optional. That is, all of the following are acceptable:

```
file:C:/dir/filename.ext

file:/C:/dir/filename.ext

file://C:/dir/filename.ext

file:///C:/dir/filename.ext
```

Actually, in our tests, the `file:////C:/dir/filename.ext` is acceptable as well, and we have no explanation for that.

Note

Consider an application that compiles some Java code, which requires a large number of permissions. Prior to JDK 9, you could have granted all permissions to the code in the `tools.jar` file. This JAR file doesn't exist any more. Instead, give permissions to the appropriate module, lile this:

```
grant codeBase "jrt:/jdk.compiler" {

   permission java.security.AllPermission;

};
```

The permissions have the following structure:

```
permission className targetName, actionList;
```

The *className* is the fully qualified class name of the permission class (such as `java.io.FilePermission`). The *targetName* is a permission-specific value—for example, a file or directory name for the file permission, or a host and port for a socket permission. The *actionList* is also permission-specific. It is a list of actions, such as `read` or `connect`, separated by commas. Some permission classes don't need target names and action lists. Table 10.2 lists the commonly used permission classes and their actions.

Table 10.2 Permissions and Their Associated Targets and Actions

http://www.rejoiceblog.com/

Permission	Target	Action
java.io.FilePermission	File target (see text)	read, write, execute, delete
java.net.SocketPermission	Socket target (see text)	accept, connect, listen, resolve
java.util.PropertyPermission	Property target (see text)	read, write
java.lang.RuntimePermission	createClassLoader getClassLoader setContextClassLoader enableContextClassLoaderOverride createSecurityManager setSecurityManager exitVM getenv.variableName shutdownHooks setFactory setIO modifyThread stopThread modifyThreadGroup getProtectionDomain readFileDescriptor writeFileDescriptor loadLibrary.libraryName accessClassInPackage.packageName defineClassInPackage.packageName accessDeclaredMembers.className queuePrintJob getStackTrace setDefaultUncaughtExceptionHandler preferences usePolicy	None

http://www.rejoiceblog.com/

java.awt.AWTPermission	showWindowWithoutWarningBanner accessClipboard accessEventQueue createRobot fullScreenExclusive listenToAllAWTEvents readDisplayPixels replaceKeyboardFocusManager watchMousePointer setWindowAlwaysOnTop setAppletStub	None
java.net.NetPermission	setDefaultAuthenticator specifyStreamHandler requestPasswordAuthentication setProxySelector getProxySelector setCookieHandler getCookieHandler setResponseCache getResponseCache	None
java.lang.reflect.ReflectPermission	suppressAccessChecks	None
java.io.SerializablePermission	enableSubclassImplementation enableSubstitution	None

http://www.rejoiceblog.com/

java.security.SecurityPermission	createAccessControlContext getDomainCombiner getPolicy setPolicy getProperty.keyName setProperty.keyName insertProvider.providerName removeProvider.providerName setSystemScope setIdentityPublicKey setIdentityInfo addIdentityCertificate removeIdentityCertificate printIdentity clearProviderProperties.providerName putProviderProperty.providerName removeProviderProperty.providerName getSignerPrivateKey setSignerKeyPair	None
java.security.AllPermission	None	None
javax.audio.AudioPermission	Play record	None
javax.security.auth.AuthPermission	doAs doAsPrivileged getSubject getSubjectFromDomainCombiner setReadOnly modifyPrincipals modifyPublicCredentials modifyPrivateCredentials refreshCredential destroyCredential createLoginContext.contextName getLoginConfiguration setLoginConfiguration refreshLoginConfiguration	None
java.util.logging.LoggingPermission	control	None
java.sql.SQLPermission	setLog	None

As you can see from Table 10.2, most permissions simply permit a particular operation. You can think of the operation as the target with an implied action `"permit"`. These permission classes all extend the `BasicPermission` class (see Figure 10.6). However, the targets for the file, socket, and property permissions are more complex, and we need to investigate them in detail.

File permission targets can have the following form:

http://www.rejoiceblog.com/

file	A file
directory/	A directory
*directory/**	All files in the directory
*	All files in the current directory
directory/-	All files in the directory or one of its subdirectories
-	All files in the current directory or one of its subdirectories
<<ALL FILES>>	All files in the file system

For example, the following permission entry gives access to all files in the directory `/myapp` and any of its subdirectories:

```
permission java.io.FilePermission "/myapp/-", "read,write,delete";
```

You must use the `\\` escape sequence to denote a backslash in a Windows file name.

```
permission java.io.FilePermission "c:\\myapp\\-", "read,write,dele
```

Socket permission targets consist of a host and a port range. Host specifications have the following form:

hostname or *IPaddress*	A single host
`localhost` or the empty string	The local host
*.*domainSuffix*	Any host whose domain ends with the given suffix
*	All hosts

Port ranges are optional and have the form:

:*n*	A single port
:*n*-	All ports numbered *n* and above
:-*n*	All ports numbered *n* and below
:*n*1-*n*2	All ports in the given range

Here is an example:

```
permission java.net.SocketPermission "*.horstmann.com:8000-8999",
```

Finally, property permission targets can have one of two forms:

property	A specific property
propertyPrefix.*	All properties with the given prefix

http://www.rejoiceblog.com/

Examples are `"java.home"` and `"java.vm.*"`.

For example, the following permission entry allows a program to read all properties that start with `java.vm`:

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

You can use system properties in policy files. The token `${property}` is replaced by the property value. For example, `${user.home}` is replaced by the home directory of the user. Here is a typical use of this system property in a permission entry:

```
permission java.io.FilePermission "${user.home}", "read,write";
```

To create platform-independent policy files, it is a good idea to use the `file.separator` property instead of explicit `/` or `\\` separators. To make this simpler, the special notation `${/}` is a shortcut for `${file.separator}`. For example,

```
permission java.io.FilePermission "${user.home}${/}-", "read,write
```

is a portable entry for granting permission to read and write in the user's home directory and any of its subdirectories.

10.2.4 Custom Permissions

In this section, you'll see how you can supply your own permission class that users can refer to in their policy files.

To implement your permission class, extend the `Permission` class and supply the following methods:

- A constructor with two `String` parameters, for the target and the action list
- `String getActions()`
- `boolean equals(Object other)`
- `int hashCode()`
- `boolean implies(Permission other)`

The last method is the most important. Permissions have an *ordering*, in which more general permissions *imply* more specific ones. Consider the file permission

http://www.rejoiceblog.com/

```
p1 = new FilePermission("/tmp/-", "read, write");
```

This permission allows reading and writing of any file in the `/tmp` directory and any of its subdirectories.

This permission implies other, more specific permissions:

```
p2 = new FilePermission("/tmp/-", "read");

p3 = new FilePermission("/tmp/aFile", "read, write");

p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

In other words, a file permission `p1` implies another file permission `p2` if

1. The target file set of `p1` contains the target file set of `p2`.

2. The action set of `p1` contains the action set of `p2`.

Consider the following example of the use of the `implies` method. When the `FileInputStream` constructor wants to open a file for reading, it checks whether it has permission to do so. For that check, a *specific* file permission object is passed to the `checkPermission` method:

```
checkPermission(new FilePermission(fileName, "read"));
```

The security manager now asks all applicable permissions whether they imply this permission. If any one of them implies it, the check passes.

In particular, the `AllPermission` implies all other permissions.

If you define your own permission classes, you need to define a suitable notion of implication for your permission objects. Suppose, for example, that you define a `TVPermission` for a set-top box powered by Java technology. A permission

```
new TVPermission("Tommy:2-12:1900-2200", "watch,record")
```

might allow Tommy to watch and record television channels 2–12 between 19:00 and 22:00. You need to implement the `implies` method so that this permission implies a more specific one, such as

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

http://www.rejoiceblog.com/

10.2.5 Implementation of a Permission Class

In the next sample program, we implement a new permission for monitoring the insertion of text into a text area. The program ensures that you cannot add bad words such as *sex*, *drugs*, and *C++* into a text area. We use a custom permission class so that the list of bad words can be supplied in a policy file.

The following subclass of `JTextArea` asks the security manager whether it is okay to add new text:

```
class WordCheckTextArea extends JTextArea

{

   public void append(String text)

   {

     var p = new WordCheckPermission(text, "insert");

     SecurityManager manager = System.getSecurityManager();

     if (manager != null) manager.checkPermission(p);

     super.append(text);

   }

}
```

If the security manager grants the `WordCheckPermission`, the text is appended. Otherwise, the `checkPermission` method throws an exception.

Word check permissions have two possible actions: `insert` (the permission to insert a specific text) and `avoid` (the permission to add any text that avoids certain bad words). You should run this program with the following policy file:

```
grant

{

   permission permissions.WordCheckPermission "sex,drugs,C++", "av

};
```

http://www.rejoiceblog.com/

This policy file grants the permission to insert any text that avoids the bad words *sex*, *drugs*, and *C++*.

When designing the `WordCheckPermission` class, we must pay particular attention to the `implies` method. Here are the rules that control whether permission `p1` implies permission `p2`:

• If `p1` has action `avoid` and `p2` has action `insert`, then the target of `p2` must avoid all words in `p1`. For example, the permission

```
permissions.WordCheckPermission "sex,drugs,C++", "avoid"
```

implies the permission

```
permissions.WordCheckPermission "Mary had a little lamb", "insert'
```

• If `p1` and `p2` both have action `avoid`, then the word set of `p2` must contain all words in the word set of `p1`. For example, the permission

```
permissions.WordCheckPermission "sex,drugs", "avoid"
```

implies the permission

```
permissions.WordCheckPermission "sex,drugs,C++", "avoid"
```

• If `p1` and `p2` both have action `insert`, then the text of `p1` must contain the text of `p2`. For example, the permission

```
permissions.WordCheckPermission "Mary had a little lamb", "insert'
```

implies the permission

```
permissions.WordCheckPermission "a little lamb", "insert"
```

You can find the implementation of this class in Listing 10.4.

Note that to retrieve the permission target, you need to use the confusingly named `getName` method of the `Permission` class.

http://www.rejoiceblog.com/

Since permissions are described by a pair of strings in policy files, permission classes need to be prepared to parse these strings. In particular, we use the following method to transform the comma-separated list of bad words of an `avoid` permission into a genuine `Set`:

```
public Set<String> badWordSet()
{
   var set = new HashSet<String>();
   set.addAll(List.of(getName().split(",")));
   return set;
}
```

This code allows us to use the `equals` and `containsAll` methods to compare sets. As you saw in Chapter 2, the `equals` method of a set class finds two sets to be equal if they contain the same elements in any order. For example, the sets resulting from `"sex,drugs,C++"` and `"C++,drugs,sex"` are equal.

Caution

Make sure that your permission class is a public class. The policy file loader cannot load classes with package visibility, and it silently ignores any classes that it cannot find.

The program in Listing 10.5 shows how the `WordCheckPermission` class works. Type any text into the text field and click the Insert button. If the security check passes, the text is appended to the text area. If not, an error message is displayed (see Figure 10.7).

http://www.rejoiceblog.com/

Figure 10.7 The `PermissionTest` program

You have now seen how to configure Java platform security. Most commonly, you will simply tweak the standard permissions. For additional control, you can define custom permissions that can be configured in the same way as the standard permissions.

Listing 10.4 `permissions/WordCheckPermission.java`

```
package permissions;

import java.security.*;
import java.util.*;

/**
 * A permission that checks for bad words.
 */
public class WordCheckPermission extends Permission
{
    private String action;

```

http://www.rejoiceblog.com/

```
    /**
     * Constructs a word check permission.
     * @param target a comma separated word list
     * @param anAction "insert" or "avoid"
     */
    public WordCheckPermission(String target, String anAction)
    {
        super(target);
        action = anAction;
    }

    public String getActions()
    {
       return action;
    }

    public boolean equals(Object other)
    {
       if (other == null) return false;
       if (!getClass().equals(other.getClass())) return false;
       var b = (WordCheckPermission) other;
       if (!Objects.equals(action, b.action)) return false;
       if ("insert".equals(action)) return Objects.equals(getNa
```

http://www.rejoiceblog.com/

```
        else if ("avoid".equals(action)) return badWordSet().equ
        else return false;
    }

    public int hashCode()
    {
        return Objects.hash(getName(), action);
    }

    public boolean implies(Permission other)
    {
        if (!(other instanceof WordCheckPermission)) return fals
        var b = (WordCheckPermission) other;
        if (action.equals("insert"))
        {
            return b.action.equals("insert") && getName().indexOf
        }
        else if (action.equals("avoid"))
        {
            if (b.action.equals("avoid")) return b.badWordSet().c
            else if (b.action.equals("insert"))
            {
                for (String badWord : badWordSet())
```

http://www.rejoiceblog.com/

```
                if (b.getName().indexOf(badWord) >= 0) return t
            return true;
        }
        else return false;
    }
    else return false;
  }

  /**
   * Gets the bad words that this permission rule describes.
   * @return a set of the bad words
   */
  public Set<String> badWordSet()
  {
    var set = new HashSet<String>();
    set.addAll(List.of(getName().split(",")));
    return set;
  }
}
```

Listing 10.5 `permissions/PermissionTest.java`

```
package permissions;

```

http://www.rejoiceblog.com/

```
import java.awt.*;

import javax.swing.*;

/**
 * This class demonstrates the custom WordCheckPermission.
 * @version 1.05 2018-05-01
 * @author Cay Horstmann
 */
public class PermissionTest
{
    public static void main(String[] args)
    {
        System.setProperty("java.security.policy", "permissio
        System.setSecurityManager(new SecurityManager());
        EventQueue.invokeLater(() ->
          {
            var frame = new PermissionTestFrame();
            frame.setTitle("PermissionTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_
            frame.setVisible(true);
          });
    }
```

http://www.rejoiceblog.com/

```
}

/**
 * This frame contains a text field for inserting words into
 * words".
 */
class PermissionTestFrame extends JFrame
{
     private JTextField textField;
    private WordCheckTextArea textArea;
    private static final int TEXT_ROWS = 20;
    private static final int TEXT_COLUMNS = 60;

    public PermissionTestFrame()
    {
       textField = new JTextField(20);
       var panel = new JPanel();
       panel.add(textField);
       var openButton = new JButton("Insert");
       panel.add(openButton);
       openButton.addActionListener(event -> insertWords(text

        add(panel, BorderLayout.NORTH);
```

http://www.rejoiceblog.com/

```
49
50          textArea = new WordCheckTextArea();
51          textArea.setRows(TEXT_ROWS);
52          textArea.setColumns(TEXT_COLUMNS);
53          add(new JScrollPane(textArea), BorderLayout.CENTER);
54          pack();
55      }
56
57      /**
58       * Tries to insert words into the text area. Displays a c
59       * @param words the words to insert
60       */
61      public void insertWords(String words)
62      {
63           try
64           {
65               textArea.append(words + "\n");
66           }
67           catch (SecurityException ex)
68           {
69               JOptionPane.showMessageDialog(this, "I am sorry,
70               ex.printStackTrace();
71           }
```

http://www.rejoiceblog.com/

```
    }
}

/**
 * A text area whose append method makes a security check to
 */
class WordCheckTextArea extends JTextArea
{
    public void append(String text)
    {
        var p = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager()
        if (manager != null) manager.checkPermission(p);
        super.append(text);
    }
}
```

java.security.Permission 1.2

- `Permission(String name)`

constructs a permission with the given target name.

- `String getName()`

returns the target name of this permission.

- `boolean implies(Permission other)`

http://www.rejoiceblog.com/

checks whether this permission implies the other permission. That is the case if the other permission describes a more specific condition that is a consequence of the condition described by this permission.

10.3 USER AUTHENTICATION

The Java API provides a framework, called the Java Authentication and Authorization Service (JAAS), that integrates platform-provided authentication with permission management. We'll discuss the JAAS framework in the following sections.

10.3.1 The JAAS Framework

As you can tell from its name, the JAAS framework has two components. The "authentication" part is concerned with ascertaining the identity of a program user. The "authorization" part maps users to permissions.

JAAS is a "pluggable" API that isolates Java applications from the particular technology used to implement authentication. It supports, among others, UNIX logins, NT logins, Kerberos authentication, and certificate-based authentication.

Once a user has been authenticated, you can attach a set of permissions. For example, here we grant Harry a particular set of permissions that other users do not have:

```
grant principal com.sun.security.auth.UnixPrincipal "harry"

{

   permission java.util.PropertyPermission "user.*", "read";

   . . .

};
```

The `com.sun.security.auth.UnixPrincipal` class checks the name of the UNIX user who is running this program. Its `getName` method returns the UNIX login name, and we check whether that name equals `"harry"`.

Use a `LoginContext` to allow the security manager to check such a grant statement. Here is the basic outline of the login code:

```
try

{

   System.setSecurityManager(new SecurityManager());
```

http://www.rejoiceblog.com/

```
    var context = new LoginContext("Login1"); // defined in JAAS co

    context.login();

    // get the authenticated Subject

    Subject subject = context.getSubject();

    . . .

    context.logout();

}

catch (LoginException exception) // thrown if login was not succes

{

    exception.printStackTrace();

}
```

Now the `subject` denotes the individual who has been authenticated.

The string parameter `"Login1"` in the `LoginContext` constructor refers to an entry with the same name in the JAAS configuration file. Here is a sample configuration file:

```
Login1

{

    com.sun.security.auth.module.UnixLoginModule required;

    com.whizzbang.auth.module.RetinaScanModule sufficient;

};
```

```
Login2

{

    . . .
```

http://www.rejoiceblog.com/

```
};
```

Of course, the JDK contains no biometric login modules. The following modules are supplied in the `com.sun.security.auth.module` package:

```
UnixLoginModule

NTLoginModule

Krb5LoginModule

JndiLoginModule

KeyStoreLoginModule
```

A login policy consists of a sequence of login modules, each of which is labeled `required`, `sufficient`, `requisite`, or `optional`. The meaning of these keywords is given by the following algorithm.

A login authenticates a *subject*, which can have multiple *principals*. A principal describes some property of the subject, such as the user name, group ID, or role. As you saw in the `grant` statement, principals govern permissions. The `com.sun.security.auth.UnixPrincipal` describes the UNIX login name, and the `UnixNumericGroupPrincipal` can test for membership in a UNIX group.

A `grant` clause can test for a principal, with the syntax

```
grant principalClass "principalName"
```

For example:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

When a user has logged in, you then run, in a separate access control context, the code that requires checking of principals. Use the static `doAs` or `doAsPrivileged` method to start a new `PrivilegedAction` whose `run` method executes the code.

Both of those methods execute an action by calling the `run` method of an object that implements the `PrivilegedAction` interface, using the permissions of the subject's principals:

http://www.rejoiceblog.com/

```
PrivilegedAction<T> action = () ->

   {

      // run with permissions of subject principals

      . . .

   };

T result = Subject.doAs(subject, action); // or Subject.doAsPrivil
```

If the actions can throw checked exceptions, you need to implement the `PrivilegedExceptionAction` interface instead.

The difference between the `doAs` and `doAsPrivileged` methods is subtle. The `doAs` method starts out with the current access control context, whereas the `doAsPrivileged` method starts out with a new context. The latter method allows you to separate the permissions for the login code and the "business logic." In our example application, the login code has permissions

```
permission javax.security.auth.AuthPermission "createLoginContext.

permission javax.security.auth.AuthPermission "doAsPrivileged";
```

The authenticated user has a permission

```
permission java.util.PropertyPermission "user.*", "read";
```

If we had used `doAs` instead of `doAsPrivileged`, then the login code would have also needed that permission!

The program in Listings 10.6 and 10.7 demonstrates how to restrict permissions to certain users. The `AuthTest` program authenticates a user and runs a simple action that retrieves a system property.

To make this example work, package the code for the login and the action into two separate JAR files:

```
javac auth/*.java

jar cvf login.jar auth/AuthTest.class
```

http://www.rejoiceblog.com/

```
jar cvf action.jar auth/SysPropAction.class
```

If you look at the policy file in Listing 10.8, you will see that the UNIX user with the name `harry` has the permission to read all files. Change `harry` to your login name. Then run the command

```
java -classpath login.jar:action.jar

   -Djava.security.policy=auth/AuthTest.policy

   -Djava.security.auth.login.config=auth/jaas.config

   auth.AuthTest
```

Listing 10.9 shows the login configuration.

On Windows, change `UnixPrincipal` to `NTUserPrincipal` in `AuthTest.policy` and `UnixLoginModule` to `NTLoginModule` in `jaas.config`. When running the program, use a semicolon to separate the JAR files:

```
java -classpath login.jar;action.jar . . .
```

The `AuthTest` program should now display the value of the `user.home` property. However, if you log in with a different name, a security exception should be thrown because you no longer have the required permission.

Caution

Be careful to follow these instructions *exactly*. It is very easy to get the setup wrong by making seemingly innocuous changes.

Listing 10.6 `auth/AuthTest.java`

```
1   package auth;

2

3   import java.security.*;
```

http://www.rejoiceblog.com/

```
import javax.security.auth.*;
import javax.security.auth.login.*;

/**
 * This program authenticates a user via a custom login and t
 * user's privileges.
 * @version 1.02 2018-05-01
 * @author Cay Horstmann
 */
public class AuthTest
{
    public static void main(final String[] args)
    {
        System.setSecurityManager(new SecurityManager());
        try
        {
            var context = new LoginContext("Login1");
            context.login();
            System.out.println("Authentication successful.");
            Subject subject = context.getSubject();
            System.out.println("subject=" + subject);
            var action = new SysPropAction("user.home");
            String result = Subject.doAsPrivileged(subject, a
```

http://www.rejoiceblog.com/

```
            System.out.println(result);
            context.logout();
        }
        catch (LoginException e)
        {
            e.printStackTrace();
        }
    }
}
```

Listing 10.7 `auth/SysPropAction.java`

```
package auth;

import java.security.*;

/**
   This action looks up a system property.
 * @version 1.01 2007-10-06
 * @author Cay Horstmann
 */
public class SysPropAction implements PrivilegedAction<String
{
    private String propertyName;
```

http://www.rejoiceblog.com/

```

    /**
         Constructs an action for looking up a given property.
         @param propertyName the property name (such as "user.
    */
     public SysPropAction(String propertyName) { this.property

     public String run()
     {
        return System.getProperty(propertyName);
     }
}
```

Listing 10.8 `auth/AuthTest.policy`

```
grant codebase "file:login.jar"
{
   permission javax.security.auth.AuthPermission "createLogin
   permission javax.security.auth.AuthPermission "doAsPrivile
};

grant principal com.sun.security.auth.UnixPrincipal "harry"
{
   permission java.util.PropertyPermission "user.*", "read";
```

http://www.rejoiceblog.com/

```
};
```

Listing 10.9 `auth/jaas.config`

```
Login1
{
   com.sun.security.auth.module.UnixLoginModule required;
};
```

javax.security.auth.login.LoginContext 1.4

- `LoginContext(String name)`

constructs a login context. The `name` corresponds to the login descriptor in the JAAS configuration file.

- `void login()`

establishes a login or throws `LoginException` if the login failed. Invokes the `login` method on the managers in the JAAS configuration file.

- `void logout()`

logs out the subject. Invokes the `logout` method on the managers in the JAAS configuration file.

- `Subject getSubject()`

returns the authenticated subject.

javax.security.auth.Subject 1.4

- `Set<Principal> getPrincipals()`

gets the principals of this subject.

- `static Object doAs(Subject subject, PrivilegedAction action)`
- `static Object doAs(Subject subject, PrivilegedExceptionAction action)`

http://www.rejoiceblog.com/

- `static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)`

- `static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action, AccessControlContext context)`

executes the privileged action on behalf of the subject. Returns the return value of the `run` method. The `doAsPrivileged` methods execute the action in the given access control context. You can supply a "context snapshot" that you obtained earlier by calling the static method `AccessController.getContext()`, or you can supply `null` to execute the code in a new context.

java.security.PrivilegedAction 1.4

- `Object run()`

You must define this method to execute the code that you want to have executed on behalf of a subject.

java.security.PrivilegedExceptionAction 1.4

- `Object run()`

You must define this method to execute the code that you want to have executed on behalf of a subject. This method may throw any checked exceptions.

java.security.Principal 1.1

- `String getName()`

returns the identifying name of this principal.

10.3.2 JAAS Login Modules

In this section, we'll look at a JAAS example that shows you

- How to implement your own login module

- How to implement *role-based* authentication

Supplying your own login module is useful if you store login information in a database. Even if you are happy with the default module, studying a custom module will help you understand the JAAS configuration file options.

http://www.rejoiceblog.com/

Role-based authentication is essential if you manage a large number of users. It would be impractical to put the names of all legitimate users into a policy file. Instead, the login module should map users to roles such as “admin” or “HR”, and the permissions should be based on these roles.

One job of the login module is to populate the principal set of the subject that is being authenticated. If a login module supports roles, it adds `Principal` objects that describe roles. The Java library does not provide a class for this purpose, so we wrote our own (see Listing 10.10). The class simply stores a description/value pair, such as `role=admin`. Its `getName` method returns that pair, so we can add role-based permissions into a policy file:

```
grant principal SimplePrincipal "role=admin" { . . . }
```

Our login module looks up users, passwords, and roles in a text file that contains lines like this:

```
harry|secret|admin

carl|guessme|HR
```

Of course, in a realistic login module, you would store this information in a database or directory.

You can find the code for the `SimpleLoginModule` in Listing 10.11. The `checkLogin` method checks whether the user name and password match a record in the password file. If so, we add two `SimplePrincipal` objects to the subject’s principal set:

```
Set<Principal> principals = subject.getPrincipals();

principals.add(new SimplePrincipal("username", username));

principals.add(new SimplePrincipal("role", role));
```

The remainder of `SimpleLoginModule` is straightforward plumbing. The `initialize` method receives

- The `Subject` that is being authenticated
- A handler to retrieve login information
- A `sharedState` map that can be used for communication between login modules
- An `options` map that contains the name/value pairs that are set in the login configuration

http://www.rejoiceblog.com/

For example, we configure our module as follows:

```
SimpleLoginModule required pwfile="password.txt";
```

The login module retrieves the `pwfile` settings from the `options` map.

The login module does not gather the user name and password; that is the job of a separate handler. This separation allows you to use the same login module without worrying whether the login information comes from a GUI dialog box, a console prompt, or a configuration file.

The handler is specified when you construct the `LoginContext`, for example:

```
var context = new LoginContext("Login1",

   new com.sun.security.auth.callback.DialogCallbackHandler());
```

The `DialogCallbackHandler` pops up a simple GUI dialog box to retrieve the user name and password. The `com.sun.security.auth.callback.TextCallbackHandler` class gets the information from the console.

However, in our application, we have our own GUI for collecting the user name and password (see Figure 10.8). We produce a simple handler that merely stores and returns that information (see Listing 10.12).

Figure 10.8 A custom login module

The handler has a single method, `handle`, that processes an array of `Callback` objects. A number of predefined classes, such as `NameCallback` and `PasswordCallback`, implement the `Callback` interface. You could also add your own class, such as `RetinaScanCallback`. The handler code is a bit unsightly because it needs to analyze the types of the callback objects:

```
public void handle(Callback[] callbacks)

{

   for (Callback callback : callbacks)
```

http://www.rejoiceblog.com/

```
    {
      if (callback instanceof NameCallback) . . .
      else if (callback instanceof PasswordCallback) . . .
      else . . .
    }
  }
```

The login module prepares an array of the callbacks that it needs for authentication:

```
var nameCall = new NameCallback("username: ");
var passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

Then it retrieves the information from the callbacks.

The program in Listing 10.13 displays a form for entering the login information and the name of a system property. If the user is authenticated, the property value is retrieved in a `PrivilegedAction`. As you can see from the policy file in Listing 10.14, only users with the `admin` role have permission to read properties.

As in the preceding section, you must separate the login and action code. Create two JAR files:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

Then run the program as

```
java -classpath login.jar:action.jar
   -Djava.security.policy=JAASTest.policy
```

http://www.rejoiceblog.com/

```
-Djava.security.auth.login.config=jaas.config

JAASTest
```

Listing 10.15 shows the policy file.

Note

It is possible to support a more complex two-phase protocol, whereby a login is *committed* if all modules in the login configuration were successful. For more information, see the login module developer's guide at `http://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASLMDevGuide`

Listing 10.10 `jaas/SimplePrincipal.java`

```
package jaas;

import java.security.*;
import java.util.*;

/**
 * A principal with a named value (such as "role=HR" or "use
 */
public class SimplePrincipal implements Principal
{
    private String descr;
    private String value;

    /**
```

http://www.rejoiceblog.com/

```
15         * Constructs a SimplePrincipal to hold a description anc
16         * @param descr the description
17         * @param value the associated value
18         */
19        public SimplePrincipal(String descr, String value)
20        {
21             this.descr = descr;
22             this.value = value;
23        }
24
25        /**
26         * Returns the role name of this principal.
27         * @return the role name
28         */
29        public String getName()
30        {
31             return descr + "=" + value;
32        }
33
34        public boolean equals(Object otherObject)
35        {
36           if (this == otherObject) return true;
37           if (otherObject == null) return false;
```

http://www.rejoiceblog.com/

```
        if (getClass() != otherObject.getClass()) return false
        var other = (SimplePrincipal) otherObject;
        return Objects.equals(getName(), other.getName());
    }

    public int hashCode()
    {
        return Objects.hashCode(getName());
    }
}
```

Listing 10.11 `jaas/SimpleLoginModule.java`

```
package jaas;

import   java.io.*;
import   java.nio.charset.*;
import   java.nio.file.*;
import   java.security.*;
import   java.util.*;
import   javax.security.auth.*;
import   javax.security.auth.callback.*;
import   javax.security.auth.login.*;
import   javax.security.auth.spi.*;
```

http://www.rejoiceblog.com/

```
12
13  /**
14   * This login module authenticates users by reading usernames
15   * file.
16   */
17  public class SimpleLoginModule implements LoginModule
18  {
19      private Subject subject;
20      private CallbackHandler callbackHandler;
21      private Map<String, ?> options;
22
23    public void initialize(Subject subject, CallbackHandler ca
24          Map<String, ?> sharedState, Map<String, ?> options)
25    {
26        this.subject = subject;
27        this.callbackHandler = callbackHandler;
28        this.options = options;
29    }
30
31    public boolean login() throws LoginException
32    {
33        if (callbackHandler == null) throw new LoginException("
34
```

http://www.rejoiceblog.com/

```
        var nameCall = new NameCallback("username: ");
        var passCall = new PasswordCallback("password: ", false
        try
        {
            callbackHandler.handle(new Callback[] { nameCall, p
        }
        catch (UnsupportedCallbackException e)
        {
            var e2 = new LoginException("Unsupported callback")
            e2.initCause(e);
            throw e2;
        }
        catch (IOException e)
        {
            var e2 = new LoginException("I/O exception in callb
            e2.initCause(e);
            throw e2;
        }

        try
        {
            return checkLogin(nameCall.getName(), passCall.getP
         }
```

http://www.rejoiceblog.com/

```
        catch (IOException ex)
        {
            var ex2 = new LoginException();
            ex2.initCause(ex);
            throw ex2;
        }
    }

    /**
     * Checks whether the authentication information is valid.
     * principals for the user name and role.
     * @param username the user name
     * @param password a character array containing the passwo
     * @return true if the authentication information is valic
     */
    private boolean checkLogin(String username, char[] passwor
    {
         try (var in = new Scanner(Paths.get("" + options.get
         {
            while (in.hasNextLine())
            {
               String[] inputs = in.nextLine().split("\\|");
               if (inputs[0].equals(username) && Arrays.equals
```

http://www.rejoiceblog.com/

```
                {
                   String role = inputs[2];
                   Set<Principal> principals = subject.getPrinc
                   principals.add(new SimplePrincipal("username
                   principals.add(new SimplePrincipal("role", r
                   return true;
                }
             }
             return false;
          }
      }

      public boolean logout()
      {
         return true;
      }

      public boolean abort()
      {
         return true;
      }

      public boolean commit()
```

http://www.rejoiceblog.com/

```
    {
        return true;
    }
}
```

Listing 10.12 `jaas/SimpleCallbackHandler.java`

```
package jaas;

import javax.security.auth.callback.*;

/**
 * This simple callback handler presents the given user name
 */
public class SimpleCallbackHandler implements CallbackHandle
{
    private String username;
    private char[] password;

    /**
     * Constructs the callback handler.
     * @param username the user name
     * @param password a character array containing the pass
     */
```

http://www.rejoiceblog.com/

```
        public SimpleCallbackHandler(String username, char[] pas
        {
            this.username = username;
            this.password = password;
        }

        public void handle(Callback[] callbacks)
        {
           for (Callback callback : callbacks)
           {
              if (callback instanceof NameCallback)
              {
                 ((NameCallback) callback).setName(username);
              }
              else if (callback instanceof PasswordCallback)
              {
                 ((PasswordCallback) callback).setPassword(passw
              }
           }
        }
    }
```

Listing 10.13 `jaas/JAASTest.java`

http://www.rejoiceblog.com/

```
1   package jaas;
2
3   import java.awt.*;
4   import javax.swing.*;
5
6   /**
7    * This program authenticates a user via a custom login and t
8    * user's privileges.
9    * @version 1.03 2018-05-01
10   * @author Cay Horstmann
11   */
12  public class JAASTest
13  {
14      public static void main(final String[] args)
15      {
16          System.setSecurityManager(new SecurityManager());
17          EventQueue.invokeLater(() ->
18              {
19                  var frame = new JAASFrame();
20                  frame.setDefaultCloseOperation(JFrame.EXIT_ON_
21                  frame.setTitle("JAASTest");
22                  frame.setVisible(true);
23              });
```

http://www.rejoiceblog.com/

```
        }
    }
```

Listing 10.14 `jaas/JAASTest.policy`

```
grant codebase "file:login.jar"
{
   permission java.awt.AWTPermission "showWindowWithoutWarnir
   permission java.awt.AWTPermission "accessEventQueue";
   permission javax.security.auth.AuthPermission "createLogir
   permission javax.security.auth.AuthPermission "doAsPrivile
   permission javax.security.auth.AuthPermission "modifyPrinc
   permission java.io.FilePermission "jaas/password.txt", "re
};

grant principal jaas.SimplePrincipal "role=admin"
{
   permission java.util.PropertyPermission "*", "read";
};
```

Listing 10.15 `jaas/jaas.config`

```
Login1
{
   jaas.SimpleLoginModule required pwfile="jaas/password.txt"
```

http://www.rejoiceblog.com/

```
};
```

javax.security.auth.callback.CallbackHandler 1.4

- `void handle(Callback[] callbacks)`

handles the given callbacks, interacting with the user if desired, and stores the security information in the callback objects.

javax.security.auth.callback.NameCallback 1.4

- `NameCallback(String prompt)`
- `NameCallback(String prompt, String defaultName)`

constructs a `NameCallback` with the given prompt and default name.

- `String getName()`
- `void setName(String name)`

gets or sets the name gathered by this callback.

- `String getPrompt()`

gets the prompt to use when querying this name.

- `String getDefaultName()`

gets the default name to use when querying this name.

javax.security.auth.callback.PasswordCallback 1.4

- `PasswordCallback(String prompt, boolean echoOn)`

constructs a `PasswordCallback` with the given prompt and echo flag.

- `char[] getPassword()`
- `void setPassword(char[] password)`

gets or sets the password gathered by this callback.

- `String getPrompt()`

http://www.rejoiceblog.com/

gets the prompt to use when querying this password.

- `boolean isEchoOn()`

gets the echo flag to use when querying this password.

javax.security.auth.spi.LoginModule 1.4

- `void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)`

initializes this `LoginModule` for authenticating the given `subject`. During login processing, uses the given handler to gather login information. Use the `sharedState` map for communicating with other login modules. The `options` map contains the name/value pairs specified in the login configuration for this module instance.

- `boolean login()`

carries out the authentication process and populates the subject's principals. Returns `true` if the login was successful.

- `boolean commit()`

is called after all login modules were successful, for login scenarios that require a two-phase commit. Returns `true` if the operation was successful.

- `boolean abort()`

is called if the failure of another login module caused the login process to abort. Returns `true` if the operation was successful.

- `boolean logout()`

logs out this subject. Returns `true` if the operation was successful.

10.4 DIGITAL SIGNATURES

As we said earlier, applets were what started the Java craze. In practice, people discovered that although they could write animated applets (like the famous "nervous text"), applets could not do a whole lot of useful stuff in the JDK 1.0 security model. For example, since applets under JDK 1.0 were so closely supervised, they couldn't do much good on a corporate intranet, even though relatively little risk attaches to executing an applet from your company's secure intranet. It quickly became clear to Sun that for applets to become truly useful, users need to be able to assign *different* levels of security, depending on where the applet originated. If an applet comes from a trusted supplier and has not been tampered with, the user of that applet can decide whether to give the applet more privileges.

http://www.rejoiceblog.com/

To give more trust to an applet, we need to know two things:

• Where did the applet come from?

• Was the code corrupted in transit?

In the past 50 years, mathematicians and computer scientists have developed sophisticated algorithms for ensuring the integrity of data and for electronic signatures. The `java.security` package contains implementations of many of these algorithms. Fortunately, you don't need to understand the underlying mathematics to use the algorithms in the `java.security` package. In the next sections, we'll show you how message digests can detect changes in data files and how digital signatures can prove the identity of the signer.

10.4.1 Message Digests

A message digest is a digital fingerprint of a block of data. For example, the so-called SHA-1 (Secure Hash Algorithm #1) condenses any data block, no matter how long, into a sequence of 160 bits (20 bytes). As with real finger-prints, one hopes that no two messages have the same SHA-1 fingerprint. Of course, that cannot be true—there are only 2^{160} SHA-1 fingerprints, so there must be some messages with the same fingerprint. But 2^{160} is so large that the probability of duplication occurring is negligible. How negligible? According to James Walsh in *True Odds: How Risks Affect Your Everyday Life* (Merritt Publishing, 1996), the chance that you will die from being struck by lightning is about one in 30,000. Now, think of nine other people—for example, your nine least favorite managers or professors. The chance that you and *all of them* will die from lightning strikes is higher than that of a forged message having the same SHA-1 fingerprint as the original. (Of course, more than ten people, none of whom you are likely to know, *will* die from lightning strikes. However, we are talking about the far slimmer chance that *your particular choice* of people will be wiped out.)

A message digest has two essential properties:

• If one bit or several bits of the data are changed, the message digest also changes.

• A forger who is in possession of a given message cannot construct a fake message that has the same message digest as the original.

The second property is, again, a matter of probabilities. Consider the following message by the billionaire father:

"Upon my death, my property shall be divided equally among my children; however, my son George shall receive nothing."

That message (with a final newline) has an SHA-1 fingerprint of

```
12 5F 09 03 E7 31 30 19 2E A6 E7 E4 90 43 84 B4 38 99 8F 67
```

http://www.rejoiceblog.com/

The distrustful father has deposited the message with one attorney and the fingerprint with another. Now, suppose George bribes the lawyer holding the message. He wants to change the message so that Bill gets nothing. Of course, that changes the fingerprint to a completely different bit pattern:

```
7D F6 AB 08 EB 40 EC CD AB 74 ED E9 86 F9 ED 99 D1 45 B1 57
```

Can George find some other wording that matches the fingerprint? If he had been the proud owner of a billion computers from the time the Earth was formed, each computing a million messages a second, he would not yet have found a message he could substitute.

A number of algorithms have been designed to compute such message digests. Among them are SHA-1, the secure hash algorithm developed by the National Institute of Standards and Technology, and MD5, an algorithm invented by Ronald Rivest of MIT. Both algorithms scramble the bits of a message in ingenious ways. For details about these algorithms, see, for example, *Cryptography and Network Security, Fifth Edition*, by William Stallings (Prentice Hall, 2011). However, subtle regularities have been discovered in both algorithms, and NIST recommends to switch to stronger alternatives. Java supports the SHA-2 and SHA-3 sets of algorithms.

The `MessageDigest` class is a *factory* for creating objects that encapsulate the fingerprinting algorithms. It has a static method, called `getInstance`, that returns an object of a class that extends the `MessageDigest` class. This means the `MessageDigest` class serves double duty:

- As a factory class
- As the superclass for all message digest algorithms

For example, here is how you obtain an object that can compute SHA fingerprints:

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

After you have obtained a `MessageDigest` object, feed it all the bytes in the message by repeatedly calling the `update` method. For example, the following code passes all bytes in a file to the `alg` object just created to do the finger-printing:

```
InputStream in = . . .

int ch;

while ((ch = in.read()) != -1)

   alg.update((byte) ch);
```

http://www.rejoiceblog.com/

Alternatively, if you have the bytes in an array, you can update the entire array at once:

```
byte[] bytes = . . .;

alg.update(bytes);
```

When you are done, call the `digest` method. This method pads the input as required by the fingerprinting algorithm, does the computation, and returns the digest as an array of bytes.

```
byte[] hash = alg.digest();
```

The program in Listing 10.16 computes a message digest. You can specify the file and algorithm on the command line:

```
java hash.Digest hash/input.txt SHA-1
```

If you do not supply command line arguments, you will be prompted for the file and algorithm name.

Listing 10.16 `hash/Digest.java`

```
package hash;

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

/**
 * This program computes the message digest of a file.
 * @version 1.21 2018-04-10
 * @author Cay Horstmann
```

http://www.rejoiceblog.com/

```
 */
public class Digest
{
    /**
     * @param args args[0] is the filename, args[1] is optior
     * (SHA-1, SHA-256, or MD5)
     */
     public static void main(String[] args) throws IOExceptic
     {
          var in = new Scanner(System.in);
          String filename;
          if (args.length >= 1)
             filename = args[0];
          else
          {
             System.out.print("File name: ");
             filename = in.nextLine();
          }
          String algname;
          if (args.length >= 2)
             algname = args[1];
          else
        {
```

http://www.rejoiceblog.com/

```
            System.out.println("Select one of the following a
            for (Provider p : Security.getProviders())
               for (Provider.Service s : p.getServices())
                  if (s.getType().equals("MessageDigest"))
                     System.out.println(s.getAlgorithm());
            System.out.print("Algorithm: ");
            algname = in.nextLine();
         }
         MessageDigest alg = MessageDigest.getInstance(algname
         byte[] input = Files.readAllBytes(Paths.get(filename)
         byte[] hash = alg.digest(input);
         for (int i = 0; i < hash.length; i++)
            System.out.printf("%02X ", hash[i] & 0xFF);
         System.out.println();
      }
   }
```

java.security.MessageDigest 1.1

- `static MessageDigest getInstance(String algorithmName)`

returns a `MessageDigest` object that implements the specified algorithm. Throws `NoSuchAlgorithmException` if the algorithm is not provided.

- `void update(byte input)`
- `void update(byte[] input)`
- `void update(byte[] input, int offset, int len)`

http://www.rejoiceblog.com/

updates the digest, using the specified bytes.

- `byte[] digest()`

completes the hash computation, returns the computed digest, and resets the algorithm object.

- `void reset()`

resets the digest.

10.4.2 Message Signing

In the last section, you saw how to compute a message digest—a fingerprint for the original message. If the message is altered, the fingerprint of the altered message will not match the fingerprint of the original. If the message and its fingerprint are delivered separately, the recipient can check whether the message has been tampered with. However, if both the message and the fingerprint were intercepted, it is an easy matter to modify the message and then recompute the fingerprint. After all, the message digest algorithms are publicly known, and they don't require secret keys. In that case, the recipient of the forged message and the recomputed fingerprint would never know that the message has been altered. Digital signatures solve this problem.

To help you understand how digital signatures work, we'll explain a few concepts from the field called *public key cryptography*. Public key cryptography is based on the notion of a *public* key and *private* key. The idea is that you tell everyone in the world your public key. However, only you hold the private key, and it is important that you safeguard it and don't release it to anyone else. The keys are matched by mathematical relationships, though the exact nature of these relationships is not important to us. (If you are interested, look it up in *The Handbook of Applied Cryptography* at `www.cacr.math.uwaterloo.ca/hac`.)

The keys are quite long and complex. For example, here is a matching pair of public and private Digital Signature Algorithm (DSA) keys.

Public key:

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617a

9bcd132acd50d99151bdc43ee737592e17
```

```
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
```

```
g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9
```

http://www.rejoiceblog.com/

```
0e1c2062354d0da20a6c416e50be794ca4
```

```
y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a812

7281ddb22cb9bc4df596d7de4d1b977d50
```

Private key:

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617a

9bcd132acd50d99151bdc43ee737592e17
```

```
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
```

```
g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9

0e1c2062354d0da20a6c416e50be794ca4
```

```
x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a
```

It is believed to be practically impossible to compute one key from the other. That is, even though everyone knows your public key, they can't compute your private key in your lifetime, no matter how many computing resources they have available.

It might seem difficult to believe that you can't compute the private key from the public key, but nobody has ever found an algorithm to do this for the

encryption algorithms in common use today. If the keys are sufficiently long, brute force—simply trying all possible keys—would require more computers than can be built from all the atoms in the solar system, crunching away for thousands of years. Of course it is possible that someone could come up with algorithms for computing keys that are much more clever than brute force. For example, the RSA algorithm (the encryption algorithm invented by Rivest, Shamir, and Adleman) depends on the difficulty of factoring large numbers. For the last 20 years, many of the best mathematicians have tried to come up with good factoring algorithms, but so far with no success. For that reason, most cryptographers believe that keys with a "modulus" of 2,000 bits or more are currently completely safe from any attack. DSA is believed to be similarly secure.

http://www.rejoiceblog.com/

Figure 10.9 illustrates how the process works in practice.

Figure 10.9 Public key signature exchange with DSA

Suppose Alice wants to send Bob a message, and Bob wants to know this message came from Alice and not an impostor. Alice writes the message and *signs* the message digest with her private key. Bob gets a copy of her public key. Bob then applies the public key to *verify* the signature. If the verification passes, Bob can be assured of two facts:

• The original message has not been altered.

• The message was signed by Alice, the holder of the private key that matches the public key that Bob used for verification.

You can see why the security of private keys is so important. If someone steals Alice's private key, or if a government can require her to turn it over, then she is in trouble. The thief or a government agent can now impersonate her by sending messages, such as money transfer instructions, that others will believe came from Alice.

10.4.3 Verifying a Signature

The JDK comes with the `keytool` program, which is a command-line tool to generate and manage a set of certificates. We expect that ultimately the functionality of this tool will be embedded in other, more user-friendly programs. But right now, we'll use `keytool` to show

http://www.rejoiceblog.com/

how Alice can sign a document and send it to Bob, and how Bob can verify that the document really was signed by Alice and not an impostor.

The `keytool` program manages *keystores*, databases of certificates and private/public key pairs. Each entry in the keystore has an *alias*. Here is how Alice creates a keystore, `alice.certs`, and generates a key pair with alias `alice`:

```
keytool -genkeypair -keystore alice.certs -alias alice
```

When creating or opening a keystore, you are prompted for a keystore password. For this example, just use `secret`. If you were to use the `keytool`-generated keystore for any serious purpose, you would need to choose a good password and safeguard this file.

When generating a key, you are prompted for the following information:

```
Enter keystore password: secret

Reenter new password: secret

What is your first and last name?

   [Unknown]: Alice Lee

What is the name of your organizational unit?

   [Unknown]: Engineering

What is the name of your organization?

   [Unknown]: ACME Software

What is the name of your City or Locality?

   [Unknown]: San Francisco

What is the name of your State or Province?

   [Unknown]: CA

What is the two-letter country code for this unit?

   [Unknown]: US

Is <CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Francisc
```

http://www.rejoiceblog.com/

```
    [no]: yes
```

The `keytool` uses names in the X.500 format, whose components are Common Name (CN), Organizational Unit (OU), Organization (O), Location (L), State (ST), and Country (C), to identify key owners and certificate issuers.

Finally, specify a key password, or press Enter to use the keystore password as the key password.

Suppose Alice wants to give her public key to Bob. She needs to export a certificate file:

```
keytool -exportcert -keystore alice.certs -alias alice -file alice
```

Now Alice can send the certificate to Bob. When Bob receives the certificate, he can print it:

```
keytool -printcert -file alice.cer
```

The printout looks like this:

```
Owner: CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Franci

Issuer: CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Franc

Serial number: 470835ce

Valid from: Sat Oct 06 18:26:38 PDT 2007 until: Fri Jan 04 17:26:3

Certificate fingerprints:

     MD5: BC:18:15:27:85:69:48:B1:5A:C3:0B:1C:C6:11:B7:81

     SHA1: 31:0A:A0:B8:C2:8B:3B:B6:85:7C:EF:C0:57:E5:94:95:61:47:6

     Signature algorithm name: SHA1withDSA

     Version: 3
```

If Bob wants to check that he got the right certificate, he can call Alice and verify the certificate fingerprint over the phone.

http://www.rejoiceblog.com/

Note

Some certificate issuers publish certificate fingerprints on their web sites. For example, to check the VeriSign certificate in the keystore `jre/lib/security/cacerts` directory, use the `-list` option:

```
keytool -list -v -keystore jre/lib/security/cacerts
```

The password for this keystore is `changeit`. One of the certificates in this keystore is

```
Owner: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authoriz

OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.", C=U

Issuer: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorize

use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign,

C=US

Serial number: 4cc7eaaa983e71d39310f83d3a899192

Valid from: Sun May 17 17:00:00 PDT 1998 until: Tue Aug 01 16:59:59 PDT 20

Certificate fingerprints:

    MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83

    SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
```

You can check that your certificate is valid by visiting the web site `www.verisign.com/repository/root.html`.

Once Bob trusts the certificate, he can import it into his keystore.

```
keytool -importcert -keystore bob.certs -alias alice -file alice.c
```

Caution

Never import into a keystore a certificate that you don't fully trust. Once a certificate

http://www.rejoiceblog.com/

is added to the keystore, any program that uses the keystore assumes that the certificate can be used to verify signatures.

Now Alice can start sending signed documents to Bob. The `jarsigner` tool signs and verifies JAR files. Alice simply adds the document to be signed into a JAR file.

```
jar cvf document.jar document.txt
```

She then uses the `jarsigner` tool to add the signature to the file. She needs to specify the keystore, the JAR file, and the alias of the key to use.

```
jarsigner -keystore alice.certs document.jar alice
```

When Bob receives the file, he uses the `-verify` option of the `jarsigner` program.

```
jarsigner -verify -keystore bob.certs document.jar
```

Bob does not need to specify the key alias. The `jarsigner` program finds the X.500 name of the key owner in the digital signature and looks for a matching certificate in the keystore.

If the JAR file is not corrupted and the signature matches, the `jarsigner` program prints

```
jar verified.
```

Otherwise, the program displays an error message.

10.4.4 The Authentication Problem

Suppose you get a message from your friend Alice, signed with her private key, using the method we just showed you. You might already have her public key, or you can easily get it by asking her for a copy or by getting it from her web page. Then, you can verify that the message was in fact authored by Alice and has not been tampered with. Now, suppose you get a message from a stranger who claims to represent a famous software company, urging you to run a program attached to the message. The stranger even sends you a copy of his public key so you can verify that he authored the message. You check that the signature is valid. This proves that the message was signed with the matching private key and has not been corrupted.

Be careful: *You still have no idea who wrote the message*. Anyone could have generated a pair of public and private keys, signed the message with the private key, and sent the signed message and the public key to you. The problem of determining the identity of the sender is

http://www.rejoiceblog.com/

called the *authentication problem*.

The usual way to solve the authentication problem is simple. Suppose the stranger and you have a common acquaintance you both trust. Suppose the stranger meets your acquaintance in person and hands over a disk with the public key. Your acquaintance later meets you, assures you that he met the stranger and that the stranger indeed works for the famous software company, and then gives you the disk (see Figure 10.10). That way, your acquaintance vouches for the authenticity of the stranger.

Figure 10.10 Authentication through a trusted intermediary

In fact, your acquaintance does not actually need to meet you. Instead, he can use his private key to sign the stranger's public key file (see Figure 10.11).

http://www.rejoiceblog.com/

Figure 10.11 Authentication through a trusted intermediary's signature

When you get the public key file, you verify the signature of your friend, and because you trust him, you are confident that he did check the stranger's credentials before applying his signature.

However, you might not have a common acquaintance. Some trust models assume that there is always a "chain of trust"—a chain of mutual acquaintances—so that you trust every member of that chain. In practice, of course, that isn't always true. You might trust your friend, Alice, and you know that Alice trusts Bob, but you don't know Bob and aren't sure that you trust him. Other trust models assume that there is a benevolent big brother—a company in which we all trust. The best known of such companies is VeriSign, Inc. (`www.verisign.com`).

You will often encounter digital signatures signed by one or more entities who will vouch for the authenticity, and you will need to evaluate to what degree you trust the authenticators. You might place a great deal of trust in VeriSign, perhaps because you saw their logo on many web pages or because you heard that they require multiple people with black attaché cases to come together into a secure chamber whenever new master keys are to be minted.

http://www.rejoiceblog.com/

However, you should have realistic expectations about what is actually being authenticated. The CEO of VeriSign does not personally meet every individual or company representative when authenticating a public key. You can get a "class 1" ID simply by filling out a web form and paying a small fee. The key is mailed to the e-mail address included in the certificate. Thus, you can be reasonably assured that the e-mail address is genuine, but the requestor could have filled in *any* name and organization. There are more stringent classes of IDs. For example, with a "class 3" ID, VeriSign will require an individual requestor to appear before a notary public, and it will check the financial rating of a corporate requestor. Other authenticators will have different procedures. Thus, when you receive an authenticated message, it is important that you understand what, in fact, is being authenticated.

10.4.5 Certificate Signing

In Section 10.4.3, "Verifying a Signature," on p. 591 you saw how Alice used a self-signed certificate to distribute a public key to Bob. However, Bob needed to ensure that the certificate was valid by verifying the fingerprint with Alice.

Suppose Alice wants to send her colleague Cindy a signed message, but Cindy doesn't want to bother with verifying lots of signature fingerprints. Now suppose there is an entity that Cindy trusts to verify signatures. In this example, Cindy trusts the Information Resources Department at ACME Software.

That department operates a *certificate authority* (CA). Everyone at ACME has the CA's public key in their keystore, installed by a system administrator who carefully checked the key fingerprint. The CA signs the keys of ACME employees. When they install each other's keys, the keystore will trust them implicitly because they are signed by a trusted key.

Here is how you can simulate this process. Create a keystore `acmesoft.certs`. Generate a key pair and export the public key:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot

keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file
```

The public key is exported into a "self-signed" certificate. Then add it to every employee's keystore.

```
keytool -importcert -keystore cindy.certs -alias acmeroot -file ac
```

For Alice to send messages to Cindy and to everyone else at ACME Software, she needs to bring her certificate to the Information Resources Department and have it signed. Unfortunately, this functionality is missing in the `keytool` program. In the book's companion code, we supply a `CertificateSigner` class to fill the gap. An authorized staff member at ACME Software would verify Alice's identity and generate a signed certificate as

http://www.rejoiceblog.com/

follows:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot
   -infile alice.cer -outfile alice_signedby_acmeroot.cer
```

The certificate signer program must have access to the ACME Software key-store, and the staff member must know the keystore password. Clearly, this is a sensitive operation.

Alice gives the file `alice_signedby_acmeroot.cer` file to Cindy and to anyone else in ACME Software. Alternatively, ACME Software can simply store the file in a company directory. Remember, this file contains Alice's public key and an assertion by ACME Software that this key really belongs to Alice.

Now Cindy imports the signed certificate into her keystore:

```
keytool -importcert -keystore cindy.certs -alias alice -file alice
```

The keystore verifies that the key was signed by a trusted root key that is already present in the keystore. Cindy is *not* asked to verify the certificate fingerprint.

Once Cindy has added the root certificate and the certificates of the people who regularly send her documents, she never has to worry about the keystore again.

10.4.6 Certificate Requests

In the preceding section, we simulated a CA with a keystore and the `CertificateSigner` tool. However, most CAs run more sophisticated software to manage certificates, and they use slightly different formats for certificates. This section shows the added steps required to interact with those software packages.

We will use the OpenSSL software package as an example. The software is preinstalled on many Linux systems and Mac OS X, and a Cygwin port is also available. You can also download the software at `www.openssl.org`.

To create a CA, run the `CA` script. The exact location depends on your operating system. On Ubuntu, run

```
/usr/lib/ssl/misc/CA.pl -newca
```

This script creates a subdirectory called `demoCA` in the current directory. The directory contains a root key pair and storage for certificates and certificate revocation lists.

http://www.rejoiceblog.com/

You will want to import the public key into the Java keystores of all employees, but it is in the Privacy Enhanced Mail (PEM) format, not the DER format that the keystore accepts easily. Copy the file `demoCA/cacert.pem` to a file `acmeroot.pem` and open that file in a text editor. Remove everything before the line

```
-----BEGIN CERTIFICATE-----
```

and after the line

```
-----END CERTIFICATE-----
```

Now you can import `acmeroot.pem` into each keystore in the usual way:

```
keytool -importcert -keystore cindy.certs -alias alice -file acme
```

It seems quite incredible that the keytool cannot carry out this editing operation itself.

To sign Alice's public key, you start by generating a *certificate request* that contains the certificate in the PEM format:

```
keytool -certreq -keystore alice.store -alias alice -file alice.pe
```

To sign the certificate, run

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

As before, cut out everything outside the `BEGIN CERTIFICATE/END CERTIFICATE` markers from `alice_signedby_acmeroot.pem`. Then import it into the keystore:

```
keytool -importcert -keystore cindy.certs -alias alice -file alice
```

You can use the same steps to have a certificate signed by a public certificate authority such as VeriSign.

10.4.7 Code Signing

A common use of authentication technology is signing executable programs. If you download a program, you are naturally concerned about the damage it can do. For example, the program could have been infected by a virus. If you know where the code comes from *and*

http://www.rejoiceblog.com/

that it has not been tampered with since it left its origin, your comfort level will be a lot higher than without this knowledge.

In this section, we'll show you how to sign JAR files, and how you can configure Java to verify the signature. This capability was designed for applets

and Java Web Start applications. These are no longer commonly used technologies, but you may still need to support them in legacy products.

When Java was first released, applets ran in the "sandbox," with limited permissions, as soon as they were loaded. If users wanted to use applets that can access the local file system, make network connections, and so on, they had to explicitly agree. To ensure that the applet code was not tampered with in transit, it had to be digitally signed.

Here is a specific example. Suppose that while surfing the Internet, you encounter a web site that offers to run an applet from an unfamiliar vendor, provided you grant it the permission to do so (see Figure 10.12). Such a program is signed with a *software developer* certificate issued by a certificate authority that the Java runtime trusts. The pop-up dialog box identifies the software developer and the certificate issuer. Now you need to decide whether to authorize the program.

Figure 10.12 Launching a signed applet

What facts do you have at your disposal that might influence your decision? Here is what you know:

• Thawte sold a certificate to the software developer.

http://www.rejoiceblog.com/

• The program really was signed with that certificate, and it hasn't been modified in transit.

• That certificate really was signed by Thawte—it was verified by the public key in the local `cacerts` file.

Of course, none of this tells you whether the code is safe to run. Can you trust a vendor if all you know is the vendor's name and the fact that Thawte sold them a software developer certificate? This approach never made much sense.

For intranet deployment, certificates are more plausible. Administrators can install policy files and certificates on local machines so that no user interaction is required for launching trusted code. Whenever the Java Plug-in tool loads signed code, it consults the policy file for the permissions and the keystore for signatures.

For the remainder of this section, we will describe how you can build policy files that grant specific permissions to trusted code.

Suppose ACME Software wants its employees to run certain programs that require local file access, and it wants to deploy these programs through a browser as Web Start applications.

As you saw earlier in this chapter, ACME could identify the programs by their code base. But that means ACME would need to update the policy files each time the programs are moved to a different web server. Instead, ACME decides to *sign* the JAR files that contain the program code.

First, ACME generates a root certificate:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

Of course, the keystore containing the private root key must be kept in a safe place. Therefore, we create a second keystore `client.certs` for the public certificates and add the public `acmeroot` certificate into it.

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file

keytool -importcert -keystore client.certs -alias acmeroot -file a
```

A trusted person at ACME runs the `jarsigner` tool to sign any application that they wish to sign, specifying the JAR file and the alias of the private key:

```
jarsigner -keystore acmesoft.certs ACMEApp.jar acmeroot
```

The signed Web Start application is now ready to be deployed on a web server.

http://www.rejoiceblog.com/

Next, let us turn to the client machine configuration. A policy file must be distributed to each client machine.

To reference a keystore, a policy file starts with the line

```
keystore "keystoreURL", "keystoreType";
```

The URL can be absolute or relative. Relative URLs are relative to the location of the policy file. The type is `JKS` if the keystore was generated by `keytool`. For example,

```
keystore "client.certs", "JKS";
```

Then `grant` clauses can have suffixes `signedBy "alias"`, such as this one:

```
grant signedBy "acmeroot"

{

    . . .

};
```

Any signed code that can be verified with the public key associated with the alias is now granted the permissions inside the `grant` clause.

10.5 ENCRYPTION

So far, we have discussed one important cryptographic technique implemented in the Java security API—namely, authentication through digital signatures. A second important aspect of security is *encryption*. Even when authenticated, the information itself is plainly visible. The digital signature merely verifies that the information has not been changed. In contrast, when information is encrypted, it is not visible. It can only be decrypted with a matching key.

Authentication is sufficient for code signing—there is no need to hide the code. However, encryption is necessary when applets or applications transfer confidential information, such as credit card numbers and other personal data.

In the past, patents and export controls prevented many companies from offering strong encryption. Fortunately, export controls are now much less stringent, and the patents for important algorithms have expired. Nowadays, Java provides excellent encryption support as a part of the standard library.

10.5.1 Symmetric Ciphers

http://www.rejoiceblog.com/

The Java cryptographic extensions contain a class `Cipher` that is the superclass of all encryption algorithms. To get a cipher object, call the `getInstance` method:

```
Cipher cipher = Cipher.getInstance(algorithName);
```

or

```
Cipher cipher = Cipher.getInstance(algorithName, providerName);
```

The JDK comes with ciphers by the provider named `"SunJCE"`. It is the default provider used if you don't specify another provider name. You might want another provider if you need specialized algorithms that Oracle does not support.

The algorithm name is a string such as `"AES"` or `"DES/CBC/PKCS5Padding"`.

The Data Encryption Standard (DES) is a venerable block cipher with a key length of 56 bits. Nowadays, the DES algorithm is considered obsolete because it can be cracked with brute force (see, for example, `http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker`). A far better alternative is its successor, the Advanced Encryption Standard (AES). See `www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf` for a detailed description of the AES algorithm. We use AES for our example.

Once you have a cipher object, initialize it by setting the mode and the key:

```
int mode = . . .;

Key key = . . .;

cipher.init(mode, key);
```

The mode is one of

```
Cipher.ENCRYPT_MODE

Cipher.DECRYPT_MODE

Cipher.WRAP_MODE

Cipher.UNWRAP_MODE
```

The wrap and unwrap modes encrypt one key with another—see the next section for an

http://www.rejoiceblog.com/

example.

Now you can repeatedly call the `update` method to encrypt blocks of data:

```
int blockSize = cipher.getBlockSize();

var inBytes = new byte[blockSize];

. . . // read inBytes

int outputSize= cipher.getOutputSize(blockSize);

var outBytes = new byte[outputSize];

int outLength = cipher.update(inBytes, 0, outputSize, outBytes);

. . . // write outBytes
```

When you are done, you must call the `doFinal` method once. If a final block of input data is available (with fewer than `blockSize` bytes), call

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

If all input data have been encrypted, instead call

```
outBytes = cipher.doFinal();
```

The call to `doFinal` is necessary to carry out *padding* of the final block. Consider the DES cipher. It has a block size of eight bytes. Suppose the last block of the input data has fewer than eight bytes. Of course, we can fill the remaining bytes with `0`, to obtain one final block of eight bytes, and encrypt it. But when the blocks are decrypted, the result will have several trailing `0` bytes appended to it, and therefore will be slightly different from the original input file. To avoid this problem, we need a *padding scheme*. A commonly used padding scheme is the one described in the Public Key Cryptography Standard (PKCS) #5 by RSA Security, Inc. (`https://tools.ietf.org/html/rfc2898`).

In this scheme, the last block is not padded with a pad value of zero, but with a pad value that equals the number of pad bytes. In other words, if `L` is the last (incomplete) block, it is padded as follows:

http://www.rejoiceblog.com/

```
L 01                          if length(L) = 7
L 02 02                       if length(L) = 6
L 03 03 03                    if length(L) = 5
. . .
L 07 07 07 07 07 07 07        if length(L) = 1
```

Finally, if the length of the input is actually divisible by 8, then one block

```
08 08 08 08 08 08 08 08
```

is appended to the input and encrypted. After decryption, the very last byte of the plaintext is a count of the padding characters to discard.

10.5.2 Key Generation

To encrypt, you need to generate a key. Each cipher has a different format for keys, and you need to make sure that the key generation is random. Follow these steps:

1. Get a `KeyGenerator` for your algorithm.

2. Initialize the generator with a source for randomness. If the block length of the cipher is variable, also specify the desired block length.

3. Call the `generateKey` method.

For example, here is how you generate an AES key:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");

var random = new SecureRandom(); // see below

keygen.init(random);

Key key = keygen.generateKey();
```

Alternatively, you can produce a key from a fixed set of raw data (perhaps derived from a password or the timing of keystrokes). Construct a `SecretKeySpec` (which implements the `SecretKey` interface) like this:

```
byte[] keyData = . . .; // 16 bytes for AES

var key = new SecretKeySpec(keyData, "AES");
```

When generating keys, make sure you use *truly random* numbers. For example, the regular random number generator in the `Random` class, seeded by the current date and time, is not

http://www.rejoiceblog.com/

random enough. Suppose the computer clock is accurate to 1/10 of a second. Then there are at most 864,000 seeds per day. If an attacker knows the day a key was issued (which can often be deduced from a message date or certificate expiration date), it is an easy matter to generate all possible seeds for that day.

The `SecureRandom` class generates random numbers that are far more secure than those produced by the `Random` class. You still need to provide a seed to start the number sequence at a random spot. The best method for doing this is to obtain random input from a hardware device such as a white-noise generator. Another reasonable source for random input is to ask the user to type away aimlessly on the keyboard, with each keystroke contributing only one or two bits to the random seed. Once you gather such random bits in an array of bytes, pass it to the `setSeed` method:

```
var secrand = new SecureRandom();

var b = new byte[20];

// fill with truly random bits

secrand.setSeed(b);
```

If you don't seed the random number generator, it will compute its own 20-byte seed by launching threads, putting them to sleep, and measuring the exact time when they are awakened.

> **Note**
>
> This algorithm is *not* known to be safe. In the past, algorithms that relied on the timing of some components of the computer, such as hard disk access time, were shown not to be completely random.

The sample program at the end of this section puts the AES cipher to work (see Listing 10.17). The `crypt` utility method in Listing 10.18 will be reused in other examples. To use the program, you first need to generate a secret key. Run

```
java aes.AESTest -genkey secret.key
```

The secret key is saved in the file `secret.key`.

Now you can encrypt with the command

http://www.rejoiceblog.com/

```
java aes.AESTest -encrypt plaintextFile encryptedFile secret.key
```

Decrypt with the command

```
java aes.AESTest -decrypt encryptedFile decryptedFile secret.key
```

The program is straightforward. The `-genkey` option produces a new secret key and serializes it in the given file. That operation takes a long time because the initialization of the secure random generator is time-consuming. The `-encrypt` and `-decrypt` options both call into the same `crypt` method that calls the `update` and `doFinal` methods of the cipher. Note how the `update` method is called as long as the input blocks have the full length, and the `doFinal` method is either called with a partial input block (which is then padded) or with no additional data (to generate one pad block).

Listing 10.17 `aes/AESTest.java`

```
package aes;

import java.io.*;
import java.security.*;
import javax.crypto.*;

/**
 * This program tests the AES cipher. Usage:<br>
 * java aes.AESTest -genkey keyfile<br>
 * java aes.AESTest -encrypt plaintext encrypted keyfile<br>
 * java aes.AESTest -decrypt encrypted decrypted keyfile<br>
 * @author Cay Horstmann
 * @version 1.02 2018-05-01
```

http://www.rejoiceblog.com/

```
 */
public class AESTest
{
    public static void main(String[] args)
        throws IOException, GeneralSecurityException, ClassNo
    {
        if (args[0].equals("-genkey"))
        {
            KeyGenerator keygen = KeyGenerator.getInstance("AE
            var random = new SecureRandom();
             keygen.init(random);
             SecretKey key = keygen.generateKey();
             try (var out = new ObjectOutputStream(new FileOut
             {
                out.writeObject(key);
             }
        }
        else
        {
            int mode;
            if (args[0].equals("-encrypt")) mode = Cipher.ENCR
            else mode = Cipher.DECRYPT_MODE;

```

http://www.rejoiceblog.com/

```
            try (var keyIn = new ObjectInputStream(new FileIn
               var in = new FileInputStream(args[1]);
               var out = new FileOutputStream(args[2]))
            {
               var key = (Key) keyIn.readObject();
               Cipher cipher = Cipher.getInstance("AES");
               cipher.init(mode, key);
               Util.crypt(in, out, cipher);
            }
         }
      }
   }

```

Listing 10.18 `aes/Util.java`

```
package aes;

import java.io.*;
import java.security.*;
import javax.crypto.*;

public class Util
{
```

http://www.rejoiceblog.com/

```
    /**
     * Uses a cipher to transform the bytes in an input strear
     * output stream.
     * @param in the input stream
     * @param out the output stream
     * @param cipher the cipher that transforms the bytes
     */
    public static void crypt(InputStream in, OutputStream out,
            GeneralSecurityException
    {
        int   blockSize = cipher.getBlockSize();
        int   outputSize = cipher.getOutputSize(blockSize);
        var   inBytes = new byte[blockSize];
        var   outBytes = new byte[outputSize];

        int inLength = 0;
        boolean more = true;
        while (more)
        {
            inLength = in.read(inBytes);
            if (inLength == blockSize)
            {
                int outLength = cipher.update(inBytes, 0, bloc
```

http://www.rejoiceblog.com/

```
                out.write(outBytes, 0, outLength);
            }
            else more = false;
        }
        if (inLength > 0) outBytes = cipher.doFinal(inBytes,
        else outBytes = cipher.doFinal();
        out.write(outBytes);
    }
}
```

javax.crypto.Cipher 1.4

- `static Cipher getInstance(String algorithmName)`
- `static Cipher getInstance(String algorithmName, String providerName)`

returns a `Cipher` object that implements the specified algorithm. Throws a `NoSuchAlgorithmException` if the algorithm is not provided.

- `int getBlockSize()`

returns the size (in bytes) of a cipher block, or `0` if the cipher is not a block cipher.

- `int getOutputSize(int inputLength)`

returns the size of an output buffer that is needed if the next input has the given number of bytes. This method takes into account any buffered bytes in the cipher object.

- `void init(int mode, Key key)`

initializes the cipher algorithm object. The mode is one of `ENCRYPT_MODE`, `DECRYPT_MODE`, `WRAP_MODE`, or `UNWRAP_MODE`.

- `byte[] update(byte[] in)`
- `byte[] update(byte[] in, int offset, int length)`

http://www.rejoiceblog.com/

- `int update(byte[] in, int offset, int length, byte[] out)`

transforms one block of input data. The first two methods return the output. The third method returns the number of bytes placed into `out`.

- `byte[] doFinal()`

- `byte[] doFinal(byte[] in)`

- `byte[] doFinal(byte[] in, int offset, int length)`

- `int doFinal(byte[] in, int offset, int length, byte[] out)`

transforms the last block of input data and flushes the buffer of this algorithm object. The first three methods return the output. The fourth method returns the number of bytes placed into `out`.

javax.crypto.KeyGenerator 1.4

- `static KeyGenerator getInstance(String algorithmName)`

returns a `KeyGenerator` object that implements the specified algorithm. Throws a `NoSuchAlgorithmException` if the algorithm is not provided.

- `void init(SecureRandom random)`

- `void init(int keySize, SecureRandom random)`

initializes the key generator.

- `SecretKey generateKey()`

generates a new key.

javax.crypto.spec.SecretKeySpec 1.4

- `SecretKeySpec(byte[] key, String algorithmName)`

constructs a key specification.

10.5.3 Cipher Streams

The JCE library provides a convenient set of stream classes that automatically encrypt or decrypt stream data. For example, here is how you can encrypt data to a file:

http://www.rejoiceblog.com/

```
Cipher cipher = . . .;

cipher.init(Cipher.ENCRYPT_MODE, key);

var out = new CipherOutputStream(new FileOutputStream(outputFileNa

var bytes = new byte[BLOCKSIZE];

int inLength = getData(bytes); // get data from data source

while (inLength != -1)

{

   out.write(bytes, 0, inLength);

   inLength = getData(bytes); // get more data from data source

}

out.flush();
```

Similarly, you can use a `CipherInputStream` to read and decrypt data from a file:

```
Cipher cipher = . . .;

cipher.init(Cipher.DECRYPT_MODE, key);

var in = new CipherInputStream(new FileInputStream(inputFileName),

var bytes = new byte[BLOCKSIZE];

int inLength = in.read(bytes);

while (inLength != -1)

{

   putData(bytes, inLength); // put data to destination

   inLength = in.read(bytes);

}
```

http://www.rejoiceblog.com/

The cipher stream classes transparently handle the calls to `update` and `doFinal`, which is clearly a convenience.

javax.crypto.CipherInputStream 1.4

- `CipherInputStream(InputStream in, Cipher cipher)`

constructs an input stream that reads data from `in` and decrypts or encrypts them by using the given cipher.

- `int read()`
- `int read(byte[] b, int off, int len)`

reads data from the input stream, which is automatically decrypted or encrypted.

javax.crypto.CipherOutputStream 1.4

- `CipherOutputStream(OutputStream out, Cipher cipher)`

constructs an output stream that writes data to `out` and encrypts or decrypts them using the given cipher.

- `void write(int ch)`
- `void write(byte[] b, int off, int len)`

writes data to the output stream, which is automatically encrypted or decrypted.

- `void flush()`

flushes the cipher buffer and carries out padding if necessary.

10.5.4 Public Key Ciphers

The AES cipher that you have seen in the preceding section is a *symmetric* cipher. The same key is used for both encryption and decryption. The Achilles heel of symmetric ciphers is key distribution. If Alice sends Bob an encrypted method, Bob needs the same key that Alice used. If Alice changes the key, she needs to send Bob both the message and, through a secure channel, the new key. But perhaps she has no secure channel to Bob—which is why she encrypts her messages to him in the first place.

Public key cryptography solves that problem. In a public key cipher, Bob has a key pair consisting of a public key and a matching private key. Bob can publish the public key

http://www.rejoiceblog.com/

anywhere, but he must closely guard the private key. Alice simply uses the public key to encrypt her messages to Bob.

Actually, it's not quite that simple. All known public key algorithms are *much* slower than symmetric key algorithms such as DES or AES. It would not be practical to use a public key algorithm to encrypt large amounts of information. However, that problem can easily be overcome by combining a public key cipher with a fast symmetric cipher, like this:

1. Alice generates a random symmetric encryption key. She uses it to encrypt her plaintext.

2. Alice encrypts the symmetric key with Bob's public key.

3. Alice sends Bob both the encrypted symmetric key and the encrypted plaintext.

4. Bob uses his private key to decrypt the symmetric key.

5. Bob uses the decrypted symmetric key to decrypt the message.

Nobody but Bob can decrypt the symmetric key because only Bob has the private key for decryption. Thus, the expensive public key encryption is only applied to a small amount of key data.

The most commonly used public key algorithm is the RSA algorithm invented by Rivest, Shamir, and Adleman. Until October 2000, the algorithm was protected by a patent assigned to RSA Security, Inc. Licenses were not cheap—typically a 3% royalty, with a minimum payment of $50,000 per year. Now the algorithm is in the public domain.

To use the RSA algorithm, you need a public/private key pair. Use a `KeyPairGenerator` like this:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");

var random = new SecureRandom();

pairgen.initialize(KEYSIZE, random);

KeyPair keyPair = pairgen.generateKeyPair();

Key publicKey = keyPair.getPublic();

Key privateKey = keyPair.getPrivate();
```

The program in Listing 10.19 has three options. The `-genkey` option produces a key pair. The `-encrypt` option generates an AES key and *wraps* it with the public key.

```
Key key = . . .; // an AES key
```

http://www.rejoiceblog.com/

```
Key publicKey = . . .; // a public RSA key

Cipher cipher = Cipher.getInstance("RSA");

cipher.init(Cipher.WRAP_MODE, publicKey);

byte[] wrappedKey = cipher.wrap(key);
```

It then produces a file that contains

- The length of the wrapped key
- The wrapped key bytes
- The plaintext encrypted with the AES key

The `-decrypt` option decrypts such a file. To try the program, first generate the RSA keys:

```
java rsa.RSATest -genkey public.key private.key
```

Then encrypt a file:

```
java rsa.RSATest -encrypt plaintextFile encryptedFile public.key
```

Finally, decrypt it and verify that the decrypted file matches the plaintext:

```
java rsa.RSATest -decrypt encryptedFile decryptedFile private.key
```

Listing 10.19 `rsa/RSATest.java`

```
package rsa;

import java.io.*;
import java.security.*;
import javax.crypto.*;
```

http://www.rejoiceblog.com/

```
 6
 7  /**
 8   * This program tests the RSA cipher. Usage:<br>
 9   * java rsa.RSATest -genkey public private<br>
10   * java rsa.RSATest -encrypt plaintext encrypted public<br>
11   * java rsa.RSATest -decrypt encrypted decrypted private<br>
12   * @author Cay Horstmann
13   * @version 1.02 2018-05-01
14   */
15  public class RSATest
16  {
17    private static final int KEYSIZE = 512;
18
19    public static void main(String[] args)
20       throws IOException, GeneralSecurityException, ClassNotFo
21    {
22       if (args[0].equals("-genkey"))
23       {
24          KeyPairGenerator pairgen = KeyPairGenerator.getInstan
25          var random = new SecureRandom();
26          pairgen.initialize(KEYSIZE, random);
27          KeyPair keyPair = pairgen.generateKeyPair();
28          try (var out = new ObjectOutputStream(new FileOutputS
```

http://www.rejoiceblog.com/

```
29          {
30             out.writeObject(keyPair.getPublic());
31          }
32          try (var out = new ObjectOutputStream(new FileOutputS
33          {
34             out.writeObject(keyPair.getPrivate());
35          }
36       }
37       else if (args[0].equals("-encrypt"))
38       {
39          KeyGenerator keygen = KeyGenerator.getInstance("AES")
40          var random = new SecureRandom();
41          keygen.init(random);
42          SecretKey key = keygen.generateKey();
43
44          // wrap with RSA public key
45          try (var keyIn = new ObjectInputStream(new FileInputS
46             var out = new DataOutputStream(new FileOutputStrea
47             var in = new FileInputStream(args[1]) )
48          {
49             var publicKey = (Key) keyIn.readObject();
50             Cipher cipher = Cipher.getInstance("RSA");
51             cipher.init(Cipher.WRAP_MODE, publicKey);
```

http://www.rejoiceblog.com/

```
            byte[] wrappedKey = cipher.wrap(key);
            out.writeInt(wrappedKey.length);
            out.write(wrappedKey);

            cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.ENCRYPT_MODE, key);
            Util.crypt(in, out, cipher);
        }
    }
    else
    {
        try (var in = new DataInputStream(new FileInputStrear
            var keyIn = new ObjectInputStream(new FileInputSt
                var out = new FileOutputStream(args[2]))
             {
                int length = in.readInt();
                var wrappedKey = new byte[length];
                in.read(wrappedKey, 0, length);

                // unwrap with RSA private key
                var privateKey = (Key) keyIn.readObject();

                Cipher cipher = Cipher.getInstance("RSA");
```

http://www.rejoiceblog.com/

```
                    cipher.init(Cipher.UNWRAP_MODE, privateKey);
                    Key key = cipher.unwrap(wrappedKey, "AES", Ci

                    cipher = Cipher.getInstance("AES");
                    cipher.init(Cipher.DECRYPT_MODE, key);

                    Util.crypt(in, out, cipher);
                }
            }
        }
    }
```

You have now seen how the Java security model allows controlled execution of code, which is a unique and increasingly important aspect of the Java platform. You have also seen the services for authentication and encryption that the Java library provides.

In the next chapter, we will delve into advanced Swing and graphics programming.

http://www.rejoiceblog.com/

Chapter 11 Advanced Swing and Graphics

In this chapter

https://avxhm.se/blogs/hill0

- 11.1 Tables
- 11.2 Trees
- 11.3 Advanced AWT
- 11.4 Raster Images
- 11.5 Printing

In this chapter, we continue our discussion of the Swing user interface toolkit and AWT graphics from Volume I. We focus on techniques that are applicable to both client-side user interfaces and server-side generation of graphics and images. Swing has sophisticated components for rendering tables and trees. With the 2D graphics API, you can produce vector art of arbitrary complexity. The ImageIO API lets you manipulate raster images. Finally, you can use the printing API to generate printouts and PostScript files.

11.1 TABLES

The `JTable` component displays a two-dimensional grid of objects. Tables are common in user interfaces, and the Swing team has put a lot of effort into the table control. Tables are inherently complex, but—perhaps more successfully than other Swing classes—the `JTable` component hides much of that complexity. You can produce fully functional tables with rich behavior by writing a few lines of code. You can also write more code and customize the display and behavior for your specific applications.

In the following sections, we will explain how to make simple tables, how the user interacts with them, and how to make some of the most common adjustments. As with the other complex Swing controls, it is impossible to cover all aspects in complete detail. For more information, look in *Graphic Java™, Third Edition*, by David M. Geary

http://www.rejoiceblog.com/
(Prentice Hall, 1999), or *Core Swing* by Kim Topley (Prentice Hall, 1999).

11.1.1 A Simple Table

A `JTable` does not store its own data but obtains them from a *table model*. The `JTable` class has a constructor that wraps a two-dimensional array of objects into a default model. That is the strategy that we use in our first example; later in this chapter, we will turn to table models.

Figure 11.1 shows a typical table, describing the properties of the planets of the solar system. (A planet is *gaseous* if it consists mostly of hydrogen and helium. You should take the "Color" entries with a grain of salt—that column was added because it will be useful in later code examples.)

Figure 11.1 A simple table

As you can see from the code in Listing 11.1, the data of the table is stored as a two-dimensional array of `Object` values:

```
Object[][] cells =

{

   { "Mercury", 2440.0, 0, false, Color.YELLOW },

   { "Venus", 6052.0, 0, false, Color.YELLOW },

   . . .
```

http://www.rejoiceblog.com/

```
}
```

Note

Here, we take advantage of autoboxing. The entries in the second, third, and fourth columns are automatically converted into objects of type `Double`, `Integer`, and `Boolean`.

The table simply invokes the `toString` method on each object to display it. That's why the colors show up as `java.awt.Color[r=. . .,g=. . .,b=. . .]`.

Supply the column names in a separate array of strings:

```
String[] columnNames = { "Planet", "Radius", "Moons", "Gaseou
```

Then, construct a table from the cell and column name arrays:

```
var table = new JTable(cells, columnNames);
```

You can add scroll bars in the usual way—by wrapping the table in a `JScrollPane`:

```
var pane = new JScrollPane(table);
```

When you scroll the table, the table header doesn't scroll out of view.

Next, click on one of the column headers and drag it to the left or right. See how the entire column becomes detached (see Figure 11.2). You can drop it in a different location. This rearranges the columns *in the view only*. The data model is not affected.

http://www.rejoiceblog.com/

Figure 11.2 Moving a column

To *resize* columns, simply place the cursor between two columns until the cursor shape changes to an arrow. Then, drag the column boundary to the desired place (see Figure 11.3).

Figure 11.3 Resizing columns

Users can select rows by clicking anywhere in a row. The selected rows are highlighted; you will see later how to get selection events. Users can also edit the table entries by clicking on a cell and typing into it. However, in this code example, the edits do not change the underlying data. In your programs, you should either make cells uneditable or handle cell editing events and update your model. We will discuss those topics later in this section.

Finally, click on a column header. The rows are automatically sorted. Click again, and the sort order is reversed. This behavior is activated by the call

http://www.rejoiceblog.com/

```
table.setAutoCreateRowSorter(true);
```

You can print a table with the call

```
table.print();
```

Caution

If you don't wrap a table into a scroll pane, you need to explicitly add the header:

```
add(table.getTableHeader(), BorderLayout.NORTH);
```

Listing 11.1 `table/TableTest.java`

```
package table;

import java.awt.*;
import java.awt.print.*;

import javax.swing.*;

/**
 * This program demonstrates how to show a simple table.
```

http://www.rejoiceblog.com/

```
10    * @version 1.14 2018-05-01
11    * @author Cay Horstmann
12    */
13   public class TableTest
14   {
15      public static void main(String[] args)
16      {
17         EventQueue.invokeLater(() ->
18            {
19               var frame = new PlanetTableFrame();
20               frame.setTitle("TableTest");
21               frame.setDefaultCloseOperation(JFrame.EXIT_
22               frame.setVisible(true);
23            });
24      }
25   }
26
27   /**
28    * This frame contains a table of planet data.
```

http://www.rejoiceblog.com/

```
 */
class PlanetTableFrame extends JFrame
{
   private String[] columnNames = { "Planet", "Radius",
   private Object[][] cells = { { "Mercury", 2440.0, 0,
         { "Venus", 6052.0, 0, false, Color.YELLOW }, {
         { "Mars", 3397.0, 2, false, Color.RED }, { "Jup
         { "Saturn", 60268.0, 18, true, Color.ORANGE },
         { "Uranus", 25559.0, 17, true, Color.BLUE }, {
         { "Pluto", 1137.0, 1, false, Color.BLACK } };

   public PlanetTableFrame()
   {
      var table = new JTable(cells, columnNames);
      table.setAutoCreateRowSorter(true);
      add(new JScrollPane(table), BorderLayout.CENTER);
      var printButton = new JButton("Print");
      printButton.addActionListener(event ->
         {
            try { table.print(); }
```

http://www.rejoiceblog.com/

```
                catch (SecurityException | PrinterException
            });
        var buttonPanel = new JPanel();
        buttonPanel.add(printButton);
        add(buttonPanel, BorderLayout.SOUTH);
        pack();
    }
}
```

javax.swing.JTable 1.2

- `JTable(Object[][] entries, Object[] columnNames)`

constructs a table with a default table model.

- `void print()` 5.0

displays a print dialog box and prints the table.

- `boolean getAutoCreateRowSorter()` 6

- `void setAutoCreateRowSorter(boolean newValue)` 6

gets or sets the `autoCreateRowSorter` property. The default is `false`. When set, a default row sorter is automatically set whenever the model changes.

- `boolean getFillsViewportHeight()` 6

- `void setFillsViewportHeight(boolean newValue)` 6

gets or sets the `fillsViewportHeight` property. The default is `false`. When set,

http://www.rejoiceblog.com/

the table always fills the enclosing viewport.

11.1.2 Table Models

In the preceding example, the table data were stored in a two-dimensional array. However, you should generally not use that strategy in your own code. Instead of dumping data into an array to display it as a table, consider implementing your own table model.

Table models are particularly simple to implement because you can take advantage of the `AbstractTableModel` class that implements most of the required methods. You only need to supply three methods:

```
public int getRowCount();

public int getColumnCount();

public Object getValueAt(int row, int column);
```

There are many ways of implementing the `getValueAt` method. For example, if you want to display the contents of a `RowSet` that contains the result of a database query, simply provide this method:

```
public Object getValueAt(int r, int c)

{

  try

  {

     rowSet.absolute(r + 1);

     return rowSet.getObject(c + 1);

  }
```

http://www.rejoiceblog.com/

```
    catch (SQLException e)

    {

        e.printStackTrace();

        return null;

    }

  }
```

Our sample program is even simpler. We construct a table that shows some computed values—namely, the growth of an investment under different interest rate scenarios (see Figure 11.4).

InvestmentTable

5%	6%	7%	8%	9%	10%
100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
105000.00	106000.00	107000.00	108000.00	109000.00	110000.00
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00
121550.63	126247.70	131079.60	136048.90	141158.16	146410.00
127628.16	133822.56	140255.17	146932.81	153862.40	161051.00
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84
188564.91	213292.83	240984.50	271962.37	306580.46	345227.12
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82

Figure 11.4 Growth of an investment

The `getValueAt` method computes the appropriate value and formats it:

```
  public Object getValueAt(int r, int c)

  {

      double rate = (c + minRate) / 100.0;
```

http://www.rejoiceblog.com/

```
        int nperiods = r;

        double futureBalance = INITIAL_BALANCE * Math.pow(1 + rat

        return String.format("%.2f", futureBalance);

    }
```

The `getRowCount` and `getColumnCount` methods simply return the number of rows and columns:

```
    public int getRowCount() { return years; }

    public int getColumnCount() { return maxRate - minRate + 1; }
```

If you don't supply column names, the `getColumnName` method of the `AbstractTableModel` names the columns A, B, C, and so on. To change the default column names, override the `getColumnName` method. In this example, we simply label each column with the interest rate.

```
    public String getColumnName(int c) { return (c + minRate) + "
```

You can find the complete source code in Listing 11.2.

Listing 11.2 `tableModel/InvestmentTable.java`

```
1   package tableModel;

2

3   import java.awt.*;

4
```

http://www.rejoiceblog.com/

```
import javax.swing.*;
import javax.swing.table.*;

/**
 * This program shows how to build a table from a table r
 * @version 1.04 2018-05-01
 * @author Cay Horstmann
 */
public class InvestmentTable
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() ->
         {
            var frame = new InvestmentTableFrame();
            frame.setTitle("InvestmentTable");
            frame.setDefaultCloseOperation(JFrame.EXIT_
            frame.setVisible(true);
         });
   }
```

http://www.rejoiceblog.com/

```
25   }
26
27   /**
28    * This frame contains the investment table.
29    */
30   class InvestmentTableFrame extends JFrame
31   {
32      public InvestmentTableFrame()
33      {
34         var model = new InvestmentTableModel(30, 5, 10);
35         var table = new JTable(model);
36         add(new JScrollPane(table));
37         pack();
38      }
39
40   }
41
42   /**
43    * This table model computes the cell entries each time
```

http://www.rejoiceblog.com/

```
 * the growth of an investment for a number of years und
 */
class InvestmentTableModel extends AbstractTableModel
{
   private static double INITIAL_BALANCE = 100000.0;

  private int years;
  private int minRate;
  private int maxRate;

  /**
   * Constructs an investment table model.
   * @param y the number of years
   * @param r1 the lowest interest rate to tabulate
   * @param r2 the highest interest rate to tabulate
   */
  public InvestmentTableModel(int y, int r1, int r2)
  {
     years = y;
     minRate = r1;
```

http://www.rejoiceblog.com/

```
        maxRate = r2;
    }

    public int getRowCount()
    {
        return years;
    }

    public int getColumnCount()
    {
        return maxRate - minRate + 1;
    }

    public Object getValueAt(int r, int c)
    {
        double rate = (c + minRate) / 100.0;
        int nperiods = r;
        double futureBalance = INITIAL_BALANCE * Math.pow(1
        return String.format("%.2f", futureBalance);
```

http://www.rejoiceblog.com/

```
    }

    public String getColumnName(int c)
    {
        return (c + minRate) + "%";
    }
  }
```

javax.swing.table.TableModel 1.2

- `int getRowCount()`
- `int getColumnCount()`

gets the number of rows and columns in the table model.

- `Object getValueAt(int row, int column)`

gets the value at the given row and column.

- `void setValueAt(Object newValue, int row, int column)`

sets a new value at the given row and column.

- `boolean isCellEditable(int row, int column)`

returns `true` if the cell at the given row and column is editable.

- `String getColumnName(int column)`

gets the column title.

http://www.rejoiceblog.com/

11.1.3 Working with Rows and Columns

In this subsection, you will see how to manipulate the rows and columns in a table. As you read through this material, keep in mind that a Swing table is quite asymmetric—the operations that you can carry out on rows and columns are different. The table component was optimized to display rows of information with the same structure, such as the result of a database query, not an arbitrary two-dimensional grid of objects. You will see this asymmetry throughout this subsection.

11.1.3.1 Column Classes

In the next example, we again display our planet data, but this time we want to give the table more information about the column types. This is achieved by defining the method

```
Class<?> getColumnClass(int columnIndex)
```

of the table model to return the class that describes the column type.

The `JTable` class uses this information to pick an appropriate renderer for the class. Table 11.1 shows the default rendering actions.

Table 11.1 Default Rendering Actions

Type	Rendered As
`Boolean`	Checkbox
`Icon`	Image
`Object`	String

You can see the checkboxes and images in Figure 11.5. (Thanks to Jim Evins, `www.snaught.com/JimsCoolIcons/Planets`, for providing the planet images!)

http://www.rejoiceblog.com/

Figure 11.5 A table with planet data

To render other types, you can install a custom renderer—see Section 11.1.4, "Cell Rendering and Editing," on p. 641.

11.1.3.2 Accessing Table Columns

The `JTable` class stores information about table columns in objects of type `TableColumn`. A `TableColumnModel` object manages the columns. (Figure 11.6 shows the relationships among the most important table classes.) If you don't want to insert or remove columns dynamically, you won't use the column model much. The most common use for the column model is simply to get a `TableColumn` object:

```
int columnIndex = . . .;
```

```
TableColumn column = table.getColumnModel().getColumn(columnI
```

http://www.rejoiceblog.com/

Figure 11.6 Relationship between table classes

11.1.3.3 Resizing Columns

The `TableColumn` class gives you control over the resizing behavior of columns. You can set the preferred, minimum, and maximum width with the methods

```
void setPreferredWidth(int width)

void setMinWidth(int width)

void setMaxWidth(int width)
```

This information is used by the table component to lay out the columns.

Use the method

http://www.rejoiceblog.com/

```
void setResizable(boolean resizable)
```

to control whether the user is allowed to resize the column.

You can programmatically resize a column with the method

```
void setWidth(int width)
```

When a column is resized, the default is to leave the total size of the table unchanged. Of course, the width increase or decrease of the resized column must then be distributed over other columns. The default behavior is to change the size of all columns to the right of the resized column. That's a good default because it allows a user to adjust all columns to a desired width, moving from left to right.

You can set another behavior from Table 11.2 by using the method

```
void setAutoResizeMode(int mode)
```

of the `JTable` class.

Table 11.2 Resize Modes

Mode	Behavior
AUTO_RESIZE_OFF	Don't resize other columns; change the table width.
AUTO_RESIZE_NEXT_COLUMN	Resize the next column only.
AUTO_RESIZE_SUBSEQUENT_COLUMNS	Resize all subsequent columns equally; this is the default behavior.
AUTO_RESIZE_LAST_COLUMN	Resize the last column only.
AUTO_RESIZE_ALL_COLUMNS	Resize all columns in the table; this is not a good choice because it prevents the user from adjusting multiple columns to a desired size.

11.1.3.4 Resizing Rows

Row heights are managed directly by the `JTable` class. If your cells are taller than the default, you may want to set the row height:

http://www.rejoiceblog.com/

```
table.setRowHeight(height);
```

By default, all rows of the table have the same height. You can set the heights of individual rows with the call

```
table.setRowHeight(row, height);
```

The actual row height equals the row height set with these methods, reduced by the row margin. The default row margin is 1 pixel, but you can change it with the call

```
table.setRowMargin(margin);
```

11.1.3.5 Selecting Rows, Columns, and Cells

Depending on the selection mode, the user can select rows, columns, or individual cells in the table. By default, row selection is enabled. Clicking inside a cell selects the entire row (see Figure 11.5). Call

```
table.setRowSelectionAllowed(false)
```

to disable row selection.

When row selection is enabled, you can control whether the user is allowed to select a single row, a contiguous set of rows, or any set of rows. You need to retrieve the *selection model* and use its `setSelectionMode` method:

```
table.getSelectionModel().setSelectionMode(mode);
```

Here, `mode` is one of the three values:

```
ListSelectionModel.SINGLE_SELECTION

ListSelectionModel.SINGLE_INTERVAL_SELECTION
```

http://www.rejoiceblog.com/

```
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

Column selection is disabled by default. You can turn it on with the call

```
table.setColumnSelectionAllowed(true)
```

Enabling both row and column selection is equivalent to enabling cell selection. The user then selects ranges of cells (see Figure 11.7). You can also enable that setting with the call

```
table.setCellSelectionEnabled(true)
```

Figure 11.7 Selecting a range of cells

Run the program in Listing 11.3 to watch cell selection in action. Enable row, column, or cell selection in the Selection menu and watch how the selection behavior changes.

http://www.rejoiceblog.com/

You can find out which rows and columns are selected by calling the `getSelectedRows` and `getSelectedColumns` methods. Both return an `int[]` array of the indexes of the selected items. Note that the index values are those of the table view, not the underlying table model. Try selecting rows and columns, then drag columns to different places and sort the rows by clicking on column headers. Use the Print Selection menu item to see which rows and columns are reported as selected.

If you need to translate the table index values to table model index values, use the `JTable` methods `convertRowIndexToModel` and `convertColumnIndexToModel`.

11.1.3.6 Sorting Rows

As you have seen in our first table example, it is easy to add row sorting to a `JTable` simply by calling the `setAutoCreateRowSorter` method. However, to have finer-grained control over the sorting behavior, install a `TableRowSorter<M>` object into a `JTable` and customize it. The type parameter `M` denotes the table model; it needs to be a subtype of the `TableModel` interface.

```
var sorter = new TableRowSorter<TableModel>(model);

table.setRowSorter(sorter);
```

Some columns should not be sortable, such as the image column in our planet data. Turn sorting off by calling

```
sorter.setSortable(IMAGE_COLUMN, false);
```

You can install a custom comparator for each column. In our example, we will sort the colors in the Color column by preferring blue and green over red. When you click on the Color column, you will see that the blue planets go to the bottom of the table. This is achieved with the following call:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()

   {

      public int compare(Color c1, Color c2)
```

http://www.rejoiceblog.com/

```
        {
            int d = c1.getBlue() - c2.getBlue();
            if (d != 0) return d;
            d = c1.getGreen() - c2.getGreen();
            if (d != 0) return d;
            return c1.getRed() - c2.getRed();
        }
    });
```

If you do not specify a comparator for a column, the sort order is determined as follows:

1. If the column class is `String`, use the default collator returned by `Collator.getInstance()`. It sorts strings in a way that is appropriate for the current locale. (See Chapter 5 for more information about locales and collators.)

2. If the column class implements `Comparable`, use its `compareTo` method.

3. If a `TableStringConverter` has been set for the sorter, sort the strings returned by the converter's `toString` method with the default collator. If you want to use this approach, define a converter as follows:

```
sorter.setStringConverter(new TableStringConverter()
    {
        public String toString(TableModel model, int row, int co
        {
            Object value = model.getValueAt(row, column);
```

http://www.rejoiceblog.com/

```
            convert value to a string and return it

        }

    });
```

4. Otherwise, call the `toString` method on the cell values and sort them with the default collator.

11.1.3.7 Filtering Rows

In addition to sorting rows, the `TableRowSorter` can also selectively hide rows—a process called *filtering*. To activate filtering, set a `RowFilter`. For example, to include all rows that contain at least one moon, call

```
sorter.setRowFilter(RowFilter.numberFilter(ComparisonType.NOT_
```

Here, we use a predefined number filter. To construct a number filter, supply

• The comparison type (one of `EQUAL`, `NOT_EQUAL`, `AFTER`, or `BEFORE`).

• An object of a subclass of `Number` (such as an `Integer` or `Double`). Only objects that have the same class as the given `Number` object are considered.

• Zero or more column index values. If no index values are supplied, all columns are searched.

The static `RowFilter.dateFilter` method constructs a date filter in the same way; you need to supply a `Date` object instead of the `Number` object.

Finally, the static `RowFilter.regexFilter` method constructs a filter that looks for strings matching a regular expression. For example,

```
sorter.setRowFilter(RowFilter.regexFilter(".*[^s]$", PLANET_C
```

http://www.rejoiceblog.com/
only displays those planets whose name doesn't end with an "s". (See Chapter 1 for more information on regular expressions.)

You can also combine filters with the `andFilter`, `orFilter`, and `notFilter` methods. To filter for planets not ending in an "s" with at least one moon, you can use this filter combination:

```
sorter.setRowFilter(RowFilter.andFilter(List.of(

   RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN),

   RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_
```

To implement your own filter, provide a subclass of `RowFilter` and implement an `include` method to indicate which rows should be displayed. This is easy to do, but the glorious generality of the `RowFilter` class makes it a bit scary.

The `RowFilter<M, I>` class has two type parameters—the types for the model and for the row identifier. When dealing with tables, the model is always a subtype of `TableModel` and the identifier type is `Integer`. (At some point in the future, other components might also support row filtering. For example, to filter rows in a `JTree`, one might use a `RowFilter<TreeModel, TreePath>`.)

A row filter must implement the method

```
public boolean include(RowFilter.Entry<? extends M, ? extends
```

The `RowFilter.Entry` class supplies methods to obtain the model, the row identifier, and the value at a given index. Therefore, you can filter both by row identifier and by the contents of the row.

For example, this filter displays every other row:

```
var filter = new RowFilter<TableModel, Integer>()

{
```

http://www.rejoiceblog.com/

```
    public boolean include(Entry<? extends TableModel, ? exten
    {
        return entry.getIdentifier() % 2 == 0;
    }
};
```

If you wanted to include only those planets with an even number of moons, you would instead test for

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

In our sample program, we allow the user to hide arbitrary rows. We store the hidden row indexes in a set. The row filter includes all rows whose indexes are not in that set.

The filtering mechanism wasn't designed for filters with criteria changing over time. In our sample program, we keep calling

```
sorter.setRowFilter(filter);
```

whenever the set of hidden rows changes. Setting a filter causes it to be applied immediately.

11.1.3.8 Hiding and Displaying Columns

As you saw in the preceding section, you can filter table rows by either their contents or their row identifier. Hiding table columns uses a completely different mechanism.

The `removeColumn` method of the `JTable` class removes a column from the table view. The column data are not actually removed from the model—they are just hidden from view. The `removeColumn` method takes a `TableColumn` argument. If you have the column number (for example, from a call to `getSelectedColumns`), you need to ask the table model for the actual table column object:

http://www.rejoiceblog.com/

```
TableColumnModel columnModel = table.getColumnModel();

TableColumn column = columnModel.getColumn(i);

table.removeColumn(column);
```

If you remember the column, you can later add it back in:

```
table.addColumn(column);
```

This method adds the column to the end. If you want it to appear elsewhere, call the `moveColumn` method.

You can also add a new column that corresponds to a column index in the table model, by adding a new `TableColumn` object:

```
table.addColumn(new TableColumn(modelColumnIndex));
```

You can have multiple table columns that view the same column of the model.

The program in Listing 11.3 demonstrates selection and filtering of rows and columns.

Listing 11.3 `tableRowColumn/PlanetTableFrame.java`

```
package tableRowColumn;

import java.awt.*;
import java.util.*;

import javax.swing.*;
```

http://www.rejoiceblog.com/

```
import javax.swing.table.*;

/**
 * This frame contains a table of planet data.
 */
public class PlanetTableFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 600;
   private static final int DEFAULT_HEIGHT = 500;

   public static final int COLOR_COLUMN = 4;
   public static final int IMAGE_COLUMN = 5;

   private   JTable table;
   private   HashSet<Integer> removedRowIndices;
   private   ArrayList<TableColumn> removedColumns;
   private   JCheckBoxMenuItem rowsItem;
   private   JCheckBoxMenuItem columnsItem;
   private   JCheckBoxMenuItem cellsItem;

```

http://www.rejoiceblog.com/

```
    private String[] columnNames = { "Planet", "Radius",

    private Object[][] cells = {
            { "Mercury", 2440.0, 0, false, Color.YELLOW,
              new ImageIcon(getClass().getResource("Mercur
            { "Venus", 6052.0, 0, false, Color.YELLOW,
              new ImageIcon(getClass().getResource("Venus.
            { "Earth", 6378.0, 1, false, Color.BLUE,
              new ImageIcon(getClass().getResource("Earth.
            { "Mars", 3397.0, 2, false, Color.RED,
              new ImageIcon(getClass().getResource("Mars.g
            { "Jupiter", 71492.0, 16, true, Color.ORANGE,
              new ImageIcon(getClass().getResource("Jupite
            { "Saturn", 60268.0, 18, true, Color.ORANGE,
              new ImageIcon(getClass().getResource("Saturn
            { "Uranus", 25559.0, 17, true, Color.BLUE,
              new ImageIcon(getClass().getResource("Uranus
            { "Neptune", 24766.0, 8, true, Color.BLUE,
              new ImageIcon(getClass().getResource("Neptun
```

http://www.rejoiceblog.com/

```
            { "Pluto", 1137.0, 1, false, Color.BLACK,
              new ImageIcon(getClass().getResource("Pluto.

    public PlanetTableFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        var model = new DefaultTableModel(cells, columnName
            {
                public Class<?> getColumnClass(int c)
                {
                    return cells[0][c].getClass();
                }
            };

        table = new JTable(model);

        table.setRowHeight(100);
        table.getColumnModel().getColumn(COLOR_COLUMN).setM
        table.getColumnModel().getColumn(IMAGE_COLUMN).setM
```

http://www.rejoiceblog.com/

```
        var sorter = new TableRowSorter<TableModel>(model);
        table.setRowSorter(sorter);
        sorter.setComparator(COLOR_COLUMN, Comparator.compa
           .thenComparing(Color::getGreen).thenComparing(Co
        sorter.setSortable(IMAGE_COLUMN, false);
        add(new JScrollPane(table), BorderLayout.CENTER);

        removedRowIndices = new HashSet<>();
        removedColumns = new ArrayList<>();

        var filter = new RowFilter<TableModel, Integer>()
        {
              public boolean include(Entry<? extends TableM
              {
                 return !removedRowIndices.contains(entry.g
              }
          };

```

http://www.rejoiceblog.com/

```
        // create menu

        var menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        var selectionMenu = new JMenu("Selection");
        menuBar.add(selectionMenu);

        rowsItem = new JCheckBoxMenuItem("Rows");
        columnsItem = new JCheckBoxMenuItem("Columns");
        cellsItem = new JCheckBoxMenuItem("Cells");

        rowsItem.setSelected(table.getRowSelectionAllowed())
        columnsItem.setSelected(table.getColumnSelectionAllo
        cellsItem.setSelected(table.getCellSelectionEnabled(

        rowsItem.addActionListener(event ->
           {
               table.clearSelection();
               table.setRowSelectionAllowed(rowsItem.isSelec
```

http://www.rejoiceblog.com/

```
            updateCheckboxMenuItems();
        });
    selectionMenu.add(rowsItem);

    columnsItem.addActionListener(event ->
        {
            table.clearSelection();
            table.setColumnSelectionAllowed(columnsItem.i
            updateCheckboxMenuItems();
        });
    selectionMenu.add(columnsItem);

    cellsItem.addActionListener(event ->
        {
            table.clearSelection();
            table.setCellSelectionEnabled(cellsItem.isSel
            updateCheckboxMenuItems();
        });
    selectionMenu.add(cellsItem);
```

http://www.rejoiceblog.com/

```
124
125     var tableMenu = new JMenu("Edit");
126     menuBar.add(tableMenu);
127
128     var hideColumnsItem = new JMenuItem("Hide Columns");
129     hideColumnsItem.addActionListener(event ->
130        {
131           int[] selected = table.getSelectedColumns();
132           TableColumnModel columnModel = table.getColumnI
133
134           // remove columns from view, starting at the l
135           // index so that column numbers aren't affecte
136
137           for (int i = selected.length - 1; i >= 0; i--)
138           {
139              TableColumn column = columnModel.getColumn(
140              table.removeColumn(column);
141
142              // store removed columns for "show columns"
143
```

http://www.rejoiceblog.com/

```
                removedColumns.add(column);
            }
        });
    tableMenu.add(hideColumnsItem);

    var showColumnsItem = new JMenuItem("Show Columns");
    showColumnsItem.addActionListener(event ->
        {
            // restore all removed columns
            for (TableColumn tc : removedColumns)
                table.addColumn(tc);
            removedColumns.clear();
        });
    tableMenu.add(showColumnsItem);

    var hideRowsItem = new JMenuItem("Hide Rows");
    hideRowsItem.addActionListener(event ->
        {
            int[] selected = table.getSelectedRows();
```

http://www.rejoiceblog.com/

```
            for (int i : selected)
                removedRowIndices.add(table.convertRowIndex
            sorter.setRowFilter(filter);
        });
    tableMenu.add(hideRowsItem);

    var showRowsItem = new JMenuItem("Show Rows");
    showRowsItem.addActionListener(event ->
        {
            removedRowIndices.clear();
            sorter.setRowFilter(filter);
        });
    tableMenu.add(showRowsItem);

    var printSelectionItem = new JMenuItem("Print Select
    printSelectionItem.addActionListener(event ->
        {
            int[] selected = table.getSelectedRows();
            System.out.println("Selected rows: " + Arrays.
            selected = table.getSelectedColumns();
```

http://www.rejoiceblog.com/

```
            System.out.println("Selected columns: " + Arra
         });
      tableMenu.add(printSelectionItem);
   }

   private void updateCheckboxMenuItems()
   {
      rowsItem.setSelected(table.getRowSelectionAllowed()
      columnsItem.setSelected(table.getColumnSelectionAll
      cellsItem.setSelected(table.getCellSelectionEnabled
   }
}
```

javax.swing.table.TableModel 1.2

- `Class getColumnClass(int columnIndex)`

gets the class for the values in this column. This information is used for sorting and rendering.

javax.swing.JTable 1.2

- `TableColumnModel getColumnModel()`

http://www.rejoiceblog.com/

gets the "column model" that describes the arrangement of the table columns.

- `void setAutoResizeMode(int mode)`

sets the mode for automatic resizing of table columns.

Parameters:	`mode`	One of `AUTO_RESIZE_OFF`, `AUTO_RESIZE_NEXT_COLUMN`, `AUTO_RESIZE_SUBSEQUENT_COLUMNS`, `AUTO_RESIZE_LAST_COLUMN`, and `AUTO_RESIZE_ALL_COLUMNS`

- `int getRowMargin()`

- `void setRowMargin(int margin)`

gets or sets the amount of empty space between cells in adjacent rows.

- `int getRowHeight()`

- `void setRowHeight(int height)`

gets or sets the default height of all rows of the table.

- `int getRowHeight(int row)`

- `void setRowHeight(int row, int height)`

gets or sets the height of the given row of the table.

- `ListSelectionModel getSelectionModel()`

returns the list selection model. You need that model to choose between row, column, and cell selection.

- `boolean getRowSelectionAllowed()`

- `void setRowSelectionAllowed(boolean b)`

gets or sets the `rowSelectionAllowed` property. If `true`, rows are selected when the user clicks on cells.

- `boolean getColumnSelectionAllowed()`

- `void setColumnSelectionAllowed(boolean b)`

http://www.rejoiceblog.com/

gets or sets the `columnSelectionAllowed` property. If `true`, columns are selected when the user clicks on cells.

- `boolean getCellSelectionEnabled()`

returns `true` if both `rowSelectionAllowed` and `columnSelectionAllowed` are `true`.

- `void setCellSelectionEnabled(boolean b)`

sets both `rowSelectionAllowed` and `columnSelectionAllowed` to `b`.

- `void addColumn(TableColumn column)`

adds a column as the last column of the table view.

- `void moveColumn(int from, int to)`

moves the column whose table index is `from` so that its index becomes `to`. Only the view is affected.

- `void removeColumn(TableColumn column)`

removes the given column from the view.

- `int convertRowIndexToModel(int index)` 6

- `int convertColumnIndexToModel(int index)`

returns the model index of the row or column with the given index. This value is different from `index` when rows are sorted or filtered, or when columns are moved or removed.

- `void setRowSorter(RowSorter<? extends TableModel> sorter)`

sets the row sorter.

javax.swing.table.TableColumnModel 1.2

- `TableColumn getColumn(int index)`

http://www.rejoiceblog.com/

gets the table column object that describes the column with the given view index.

javax.swing.table.TableColumn 1.2

- `TableColumn(int modelColumnIndex)`

constructs a table column for viewing the model column with the given index.

- `void setPreferredWidth(int width)`
- `void setMinWidth(int width)`
- `void setMaxWidth(int width)`

sets the preferred, minimum, and maximum width of this table column to `width`.

- `void setWidth(int width)`

sets the actual width of this column to `width`.

- `void setResizable(boolean b)`

If `b` is `true`, this column is resizable.

javax.swing.ListSelectionModel 1.2

- `void setSelectionMode(int mode)`

Parameters:	`mode`	One of `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION`, and `MULTIPLE_INTERVAL_SELECTION`

javax.swing.DefaultRowSorter<M, I> 6

- `void setComparator(int column, Comparator<?> comparator)`

sets the comparator to be used with the given column.

- `void setSortable(int column, boolean enabled)`

http://www.rejoiceblog.com/

enables or disables sorting for the given column.

- `void setRowFilter(RowFilter<? super M,? super I> filter)`

sets the row filter.

javax.swing.table.TableRowSorter<M extends TableModel> 6

- `void setStringConverter(TableStringConverter stringConverter)`

sets the string converter used for sorting and filtering.

javax.swing.table.TableStringConverter 6

- `abstract String toString(TableModel model, int row, int column)`

converts the model value at the given location to a string; you can override this method.

javax.swing.RowFilter<M, I> 6

- `boolean include(RowFilter.Entry<? extends M,? extends I> entry)`

specifies the rows that are retained; you can override this method.

- `static <M,I> RowFilter<M,I> numberFilter(RowFilter.ComparisonType type, Number number, int... indices)`

- `static <M,I> RowFilter<M,I> dateFilter(RowFilter.ComparisonType type, Date date, int... indices)`

returns a filter that includes rows containing values that match the given comparison

http://www.rejoiceblog.com/

to the given number or date. The comparison type is one of `EQUAL`, `NOT_EQUAL`, `AFTER`, or `BEFORE`. If any column model indexes are given, only those columns are searched; otherwise, all columns are searched. For the number filter, the class of the cell value must match the class of `number`.

- `static <M,I> RowFilter<M,I> regexFilter(String regex, int... indices)`

returns a filter that includes rows that have a string value matching the given regular expression. If any column model indexes are given, only those columns are searched; otherwise, all columns are searched. Note that the string returned by the `getStringValue` method of `RowFilter.Entry` is matched.

- `static <M,I> RowFilter<M,I> andFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)`

- `static <M,I> RowFilter<M,I> orFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)`

returns a filter that includes the entries included by all filters or at least one of the filters.

- `static <M,I> RowFilter<M,I> notFilter(RowFilter<M,I> filter)`

returns a filter that includes the entries not included by the given filter.

javax.swing.RowFilter.Entry<M, I> 6

- `I getIdentifier()`

returns the identifier of this row entry.

- `M getModel()`

returns the model of this row entry.

- `Object getValue(int index)`

returns the value stored at the given index of this row.

http://www.rejoiceblog.com/

- `int getValueCount()`

returns the number of values stored in this row.

- `String getStringValue()`

returns the value stored at the given index of this row, converted to a string. The `TableRowSorter` produces entries whose `getStringValue` calls the sorter's string converter.

11.1.4 Cell Rendering and Editing

As you saw in Section 11.5.5.2, "Accessing Table Columns," on p. , the column type determines how the cells are rendered. There are default renderers for the types `Boolean` and `Icon` that render a checkbox or icon. For all other types, you need to install a custom renderer.

11.1.4.1 Rendering Cells

Table cell renderers are similar to the list cell renderers that you saw earlier. They implement the `TableCellRenderer` interface which has a single method:

```
Component getTableCellRendererComponent(JTable table, Object

      boolean hasFocus, int row, int column)
```

That method is called when the table needs to draw a cell. You return a component whose `paint` method is then invoked to fill the cell area.

The table in Figure 11.8 contains cells of type `Color`. The renderer simply returns a panel with a background color that is the color object stored in the cell. The color is passed as the `value` parameter.

```
class ColorTableCellRenderer extends JPanel implements TableC

{

   public Component getTableCellRendererComponent(JTable tabl
```

http://www.rejoiceblog.com/

```
            boolean isSelected, boolean hasFocus, int row, int c

        {

            setBackground((Color) value);

            if (hasFocus)

                setBorder(UIManager.getBorder("Table.focusCellHighli

            else

                setBorder(null);

        return this;

        }

    }
```

Figure 11.8 A table with cell renderers

http://www.rejoiceblog.com/

As you can see, the renderer draws a border when the cell has focus. (We ask the `UIManager` for the correct border. To find the lookup key, we peeked into the source code of the `DefaultTableCellRenderer` class.)

> **Tip**
>
> If your renderer simply draws a text string or an icon, you can extend the `DefaultTableCellRenderer` class. It takes care of rendering the focus and selection status for you.

You need to tell the table to use this renderer with all objects of type `Color`. The `setDefaultRenderer` method of the `JTable` class lets you establish this association. Supply a `Class` object and the renderer:

```
table.setDefaultRenderer(Color.class, new ColorTableCellRende
```

That renderer is now used for all objects of the given type in this table.

If you want to select a renderer based on some other criterion, you need to subclass the `JTable` class and override the `getCellRenderer` method.

11.1.4.2 Rendering the Header

To display an icon in the header, set the header value:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

However, the table header isn't smart enough to choose an appropriate renderer for the header value. You have to install the renderer manually. For example, to show an image icon in a column header, call

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIc
```

http://www.rejoiceblog.com/

11.1.4.3 Editing Cells

To enable cell editing, the table model must indicate which cells are editable by defining the `isCellEditable` method. Most commonly, you will want to make certain columns editable. In the example program, we allow editing in four columns.

```
public boolean isCellEditable(int r, int c)

{

   return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GAS

      || c == COLOR_COLUMN;

}
```

Note

The `AbstractTableModel` defines the `isCellEditable` method to always return `false`. The `DefaultTableModel` overrides the method to always return `true`.

If you run the program (Listings 11.4 to 11.7), note that you can click the checkboxes in the Gaseous column and turn the check marks on and off. If you click a cell in the Moons column, a combo box appears (see Figure 11.9). You will shortly see how to install such a combo box as a cell editor.

http://www.rejoiceblog.com/

Figure 11.9 A cell editor

Finally, click a cell in the first column. The cell gains focus. You can start typing, and the cell contents change.

What you just saw in action are the three variations of the `DefaultCellEditor` class. A `DefaultCellEditor` can be constructed with a `JTextField`, a `JCheckBox`, or a `JComboBox`. The `JTable` class automatically installs a checkbox editor for `Boolean` cells and a text field editor for all editable cells that don't supply their own renderer. The text fields let the user edit the strings that result from applying `toString` to the return value of the `getValueAt` method of the table model.

When the edit is complete, the edited value is retrieved by calling the `getCellEditorValue` method of your editor. That method should return a value of the correct type (that is, the type returned by the `getColumnType` method of the model).

To get a combo box editor, set a cell editor manually—the `JTable` component has no idea what values might be appropriate for a particular type. For the Moons column, we wanted to enable the user to pick any value between 0 and 20. Here is the code for initializing the combo box:

```
var moonCombo = new JComboBox();
```

http://www.rejoiceblog.com/

```
for (int i = 0; i <= 20; i++)

  moonCombo.addItem(i);
```

To construct a `DefaultCellEditor`, supply the combo box in the constructor:

```
var moonEditor = new DefaultCellEditor(moonCombo);
```

Next, we need to install the editor. Unlike the color cell renderer, this editor does not depend on the object *type*—we don't necessarily want to use it for all objects of type `Integer`. Instead, we need to install it into a particular column:

```
moonColumn.setCellEditor(moonEditor);
```

11.1.4.4 Custom Editors

Run the example program again and click a color. A *color chooser* pops up and lets you pick a new color for the planet. Select a color and click OK. The cell color is updated (see Figure 11.10).

http://www.rejoiceblog.com/

Figure 11.10 Editing the cell color with a color chooser

The color cell editor is not a standard table cell editor but a custom implementation. To create a custom cell editor, implement the `TableCellEditor` inter-face. That interface is a bit tedious, and as of Java SE 1.3, an `AbstractCellEditor` class is provided to take care of the event handling details.

The `getTableCellEditorComponent` method of the `TableCellEditor` interface requests a component to render the cell. It is exactly the same as the `getTableCellRendererComponent` method of the `TableCellRenderer` interface, except that there is no `focus` parameter. When the cell is being edited, it is presumed to have focus. The editor component temporarily *replaces* the renderer when the editing is in progress. In our example, we return a blank panel that is not colored. This is an indication to the user that the cell is currently being edited.

Next, you want to have your editor pop up when the user clicks on the cell.

The `JTable` class calls your editor with an event (such as a mouse click) to find out if

http://www.rejoiceblog.com/

that event is acceptable to initiate the editing process. The `AbstractCellEditor` class defines the method to accept all events.

```
public boolean isCellEditable(EventObject anEvent)
{
   return true;
}
```

However, if you override this method to return `false`, the table would not go through the trouble of inserting the editor component.

Once the editor component is installed, the `shouldSelectCell` method is called, presumably with the same event. You should initiate editing in this method—for example, by popping up an external edit dialog box.

```
public boolean shouldSelectCell(EventObject anEvent)
{
   colorDialog.setVisible(true);
   return true;
}
```

If the user cancels the edit, the table calls the `cancelCellEditing` method. If the user has clicked on another table cell, the table calls the `stopCellEditing` method. In both cases, you should hide the dialog box. When your `stopCellEditing` method is called, the table would like to use the partially edited value. You should return `true` if the current value is valid. In the color chooser, any value is valid. But if you edit other data, you can ensure that only valid data are retrieved from the editor.

Also, you should call the superclass methods that take care of event firing—otherwise, the editing won't be properly canceled.

http://www.rejoiceblog.com/

```
public void cancelCellEditing()

{

    colorDialog.setVisible(false);

    super.cancelCellEditing();

}
```

Finally, you need a method that yields the value that the user supplied in the editing process:

```
public Object getCellEditorValue()

{

    return colorChooser.getColor();

}
```

To summarize, your custom editor should do the following:

1. Extend the `AbstractCellEditor` class and implement the `TableCellEditor` interface.

2. Define the `getTableCellEditorComponent` method to supply a component. This can either be a dummy component (if you pop up a dialog box) or a component for in-place editing such as a combo box or text field.

3. Define the `shouldSelectCell`, `stopCellEditing`, and `cancelCellEditing` methods to handle the start, completion, and cancellation of the editing process. The `stopCellEditing` and `cancelCellEditing` methods should call the superclass methods to ensure that listeners are notified.

4. Define the `getCellEditorValue` method to return the value that is the result of the editing process.

http://www.rejoiceblog.com/

Finally, indicate when the user is finished editing by calling the `stopCellEditing` and `cancelCellEditing` methods. When constructing the color dialog box, we install the accept and cancel callbacks that fire these events.

```
colorDialog = JColorChooser.createDialog(null, "Planet Color"

   EventHandler.create(ActionListener.class, this, "stopCellE

   EventHandler.create(ActionListener.class, this, "cancelCel
```

This completes the implementation of the custom editor.

You now know how to make a cell editable and how to install an editor. There is one remaining issue—how to update the model with the value that the user edited. When editing is complete, the `JTable` class calls the following method of the table model:

```
void setValueAt(Object value, int r, int c)
```

You need to override the method to store the new value. The `value` parameter is the object that was returned by the cell editor. If you implemented the cell editor, you know the type of the object you return from the `getCellEditorValue` method. In the case of the `DefaultCellEditor`, there are three possibilities for that value. It is a `Boolean` if the cell editor is a checkbox, a string if it is a text field, and, if the value comes from a combo box, it is the object that the user selected.

If the `value` object does not have the appropriate type, you need to convert it. That happens most commonly when a number is edited in a text field. In our example, we populated the combo box with `Integer` objects so that no conversion is necessary.

Listing 11.4 `tableCellRender/TableCellRenderFrame.java`

```
1  package tableCellRender;

2

3  import java.awt.*;
```

http://www.rejoiceblog.com/

```
 4  import javax.swing.*;
 5  import javax.swing.table.*;
 6
 7  /**
 8   * This frame contains a table of planet data.
 9   */
10  public class TableCellRenderFrame extends JFrame
11  {
12    private static final int DEFAULT_WIDTH = 600;
13    private static final int DEFAULT_HEIGHT = 400;
14
15    public TableCellRenderFrame()
16    {
17       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19       var model = new PlanetTableModel();
20       var table = new JTable(model);
21       table.setRowSelectionAllowed(false);
22
```

http://www.rejoiceblog.com/

```
23        // set up renderers and editors
24
25        table.setDefaultRenderer(Color.class, new ColorTable
26        table.setDefaultEditor(Color.class, new ColorTableCe
27
28        var moonCombo = new JComboBox<Integer>();
29        for (int i = 0; i <= 20; i++)
30          moonCombo.addItem(i);
31
32        TableColumnModel columnModel = table.getColumnModel(
33        TableColumn moonColumn = columnModel.getColumn(Plane
34        moonColumn.setCellEditor(new DefaultCellEditor(moonC
35        moonColumn.setHeaderRenderer(table.getDefaultRendere
36        moonColumn.setHeaderValue(new ImageIcon(getClass().g
37
38        // show table
39
40        table.setRowHeight(100);
41        add(new JScrollPane(table), BorderLayout.CENTER);
42      }
```

http://www.rejoiceblog.com/

```
}

```

Listing 11.5 `tableCellRender/PlanetTableModel.java`

```
package tableCellRender;

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * The planet table model specifies the values, rendering
 * data.
 */
public class PlanetTableModel extends AbstractTableModel
{
    public static final int PLANET_COLUMN = 0;
    public static final int MOONS_COLUMN = 2;
    public static final int GASEOUS_COLUMN = 3;
```

http://www.rejoiceblog.com/

```
    public static final int COLOR_COLUMN = 4;

    private Object[][] cells = {
            { "Mercury", 2440.0, 0, false, Color.YELLOW,
              new ImageIcon(getClass().getResource("Mercury.
            { "Venus", 6052.0, 0, false, Color.YELLOW,
              new ImageIcon(getClass().getResource("Venus.gi
            { "Earth", 6378.0, 1, false, Color.BLUE,
              new ImageIcon(getClass().getResource("Earth.gi
            { "Mars", 3397.0, 2, false, Color.RED,
              new ImageIcon(getClass().getResource("Mars.gif
            { "Jupiter", 71492.0, 16, true, Color.ORANGE,
              new ImageIcon(getClass().getResource("Jupiter.
            { "Saturn", 60268.0, 18, true, Color.ORANGE,
              new ImageIcon(getClass().getResource("Saturn.g
            { "Uranus", 25559.0, 17, true, Color.BLUE,
              new ImageIcon(getClass().getResource("Uranus.g
            { "Neptune", 24766.0, 8, true, Color.BLUE,
              new ImageIcon(getClass().getResource("Neptune.
            { "Pluto", 1137.0, 1, false, Color.BLACK,
```

http://www.rejoiceblog.com/

```
            new ImageIcon(getClass().getResource("Pluto.gi

    private String[] columnNames = { "Planet", "Radius", "

    public String getColumnName(int c)
    {
      return columnNames[c];
    }

    public Class<?> getColumnClass(int c)
    {
      return cells[0][c].getClass();
    }

    public int getColumnCount()
    {
      return cells[0].length;
    }

```

http://www.rejoiceblog.com/

```
    public int getRowCount()
    {
      return cells.length;
    }

    public Object getValueAt(int r, int c)
    {
      return cells[r][c];
    }

    public void setValueAt(Object obj, int r, int c)
    {
      cells[r][c] = obj;
    }

    public boolean isCellEditable(int r, int c)
    {
      return c == PLANET_COLUMN || c == MOONS_COLUMN || c
    }
}
```

http://www.rejoiceblog.com/

Listing 11.6 `tableCellRender/ColorTableCellRenderer.java`

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.table.*;
6
7  /**
8   * This renderer renders a color value as a panel with th
9   */
10 public class ColorTableCellRenderer extends JPanel implem
11 {
12    public Component getTableCellRendererComponent(JTable
13          boolean hasFocus, int row, int column)
14    {
15      setBackground((Color) value);
16      if (hasFocus) setBorder(UIManager.getBorder("Table.f
17      else setBorder(null);
18      return this;
```

http://www.rejoiceblog.com/

```
    }
}
```

Listing 11.7 `tableCellRender/ColorTableCellEditor.java`

```
package tableCellRender;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * This editor pops up a color dialog to edit a cell valu
 */
public class ColorTableCellEditor extends AbstractCellEdi
{
   private JColorChooser colorChooser;
```

http://www.rejoiceblog.com/

```
    private JDialog colorDialog;

    private JPanel panel;

    public ColorTableCellEditor()
    {
      panel = new JPanel();
      // prepare color dialog

      colorChooser = new JColorChooser();
      colorDialog = JColorChooser.createDialog(null, "Plan
         EventHandler.create(ActionListener.class, this, "
         EventHandler.create(ActionListener.class, this, "
    }

    public Component getTableCellEditorComponent(JTable ta
          int row, int column)
    {
       // this is where we get the current Color value. We
       // starts editing
       colorChooser.setColor((Color) value);
```

http://www.rejoiceblog.com/

```
        return panel;
    }

    public boolean shouldSelectCell(EventObject anEvent)
    {
        // start editing
        colorDialog.setVisible(true);

        // tell caller it is ok to select this cell
        return true;
    }

    public void cancelCellEditing()
    {
        // editing is canceled--hide dialog
        colorDialog.setVisible(false);
        super.cancelCellEditing();
    }

```

http://www.rejoiceblog.com/

```
    public boolean stopCellEditing()
    {
      // editing is complete--hide dialog
      colorDialog.setVisible(false);
      super.stopCellEditing();

      // tell caller is is ok to use color value
      return true;
    }

    public Object getCellEditorValue()
    {
      return colorChooser.getColor();
    }
  }
```

javax.swing.JTable 1.2

- `TableCellRenderer getDefaultRenderer(Class<?> type)`

gets the default renderer for the given type.

- `TableCellEditor getDefaultEditor(Class<?> type)`

http://www.rejoiceblog.com/

gets the default editor for the given type.

javax.swing.table.TableCellRenderer 1.2

- `Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)`

returns a component whose `paint` method is invoked to render a table cell.

Parameters:	`table`	The table containing the cell to be rendered
	`value`	The cell to be rendered
	`selected`	`true` if the cell is currently selected
	`hasFocus`	`true` if the cell currently has focus
	`row, column`	The row and column of the cell

javax.swing.table.TableColumn 1.2

- `void setCellEditor(TableCellEditor editor)`

- `void setCellRenderer(TableCellRenderer renderer)`

sets the cell editor or renderer for all cells in this column.

- `void setHeaderRenderer(TableCellRenderer renderer)`

sets the cell renderer for the header cell in this column.

- `void setHeaderValue(Object value)`

sets the value to be displayed for the header in this column.

javax.swing.DefaultCellEditor 1.2

- `DefaultCellEditor(JComboBox comboBox)`

constructs a cell editor that presents the combo box for selecting cell values.

http://www.rejoiceblog.com/

javax.swing.table.TableCellEditor 1.2

- `Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)`

returns a component whose `paint` method renders a table cell.

Parameters:	`table`	The table containing the cell to be rendered
	`value`	The cell to be rendered
	`selected`	`true` if the cell is currently selected
	`row, column`	The row and column of the cell

javax.swing.CellEditor 1.2

- `boolean isCellEditable(EventObject event)`

returns `true` if the event is suitable for initiating the editing process for this cell.

- `boolean shouldSelectCell(EventObject anEvent)`

starts the editing process. Returns `true` if the edited cell should be *selected*. Normally, you want to return `true`, but you can return `false` if you don't want the editing process to change the cell selection.

- `void cancelCellEditing()`

cancels the editing process. You can abandon partial edits.

- `boolean stopCellEditing()`

stops the editing process, with the intent of using the result. Returns `true` if the edited value is in a proper state for retrieval.

- `Object getCellEditorValue()`

returns the edited result.

- `void addCellEditorListener(CellEditorListener l)`

http://www.rejoiceblog.com/

- `void removeCellEditorListener(CellEditorListener l)`

adds or removes the obligatory cell editor listener.

11.2 TREES

Every computer user who has worked with a hierarchical file system has seen tree displays. Of course, directories and files form only one of the many examples of tree-like organizations. Many tree structures arise in everyday life, such as the hierarchy of countries, states, and cities shown in Figure 11.11.

Figure 11.11 A hierarchy of countries, states, and cities

As programmers, we often need to display tree structures. Fortunately, the Swing library has a `JTree` class for this purpose. The `JTree` class (together with its helper classes) takes care of laying out the tree and processing user requests for expanding and collapsing nodes. In this section, you will learn how to put the `JTree` class to use.

As with the other complex Swing components, we must focus on the common and useful cases and cannot cover every nuance. If you want to achieve something unusual, we recommend that you consult *Graphic Java* ™*, Third Edition*, by David M. Geary or *Core Swing* by Kim Topley.

Before going any further, let's settle on some terminology (see Figure 11.12). A *tree* is composed of *nodes*. Every node is either a *leaf* or it has *child nodes*. Every node, with

http://www.rejoiceblog.com/

the exception of the root node, has exactly one *parent*. A tree has exactly one root node. Sometimes you have a collection of trees, each with its own root node. Such a collection is called a *forest*.

Figure 11.12 Tree terminology

11.2.1 Simple Trees

In our first example program, we will simply display a tree with a few nodes (see Figure 11.14). As with many other Swing components, you need to provide a model of the data, and the component displays it for you. To construct a `JTree`, supply the tree model in the constructor:

```
TreeModel model = . . .;

var tree = new JTree(model);
```

Note

There are also constructors that construct trees out of a collection of

http://www.rejoiceblog.com/

elements:

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)

JTree(Hashtable<?, ?> nodes) // the values become the nodes
```

These constructors are not very useful. They merely build a forest of trees, each with a single node. The third constructor seems particularly useless because the nodes appear in the seemingly random order determined by the hash codes of the keys.

How do you obtain a tree model? You can construct your own model by creating a class that implements the `TreeModel` interface. You will see later in this chapter how to do that. For now, we will stick with the `DefaultTreeModel` that the Swing library supplies.

To construct a default tree model, you must supply a root node.

```
TreeNode root = . . .;

var model = new DefaultTreeModel(root);
```

`TreeNode` is another interface. Populate the default tree model with objects of any class that implements the interface. For now, we will use the concrete node class that Swing supplies—namely, `DefaultMutableTreeNode`. This class implements the `MutableTreeNode` interface, a subinterface of `TreeNode` (see Figure 11.13).

http://www.rejoiceblog.com/

Figure 11.13 Tree classes

A default mutable tree node holds an object—the *user object*. The tree renders the user objects for all nodes. Unless you specify a renderer, the tree displays the string that is the result of the `toString` method.

In our first example, we use strings as user objects. In practice, you would usually populate a tree with more expressive user objects. For example, when displaying a directory tree, it makes sense to use `File` objects for the nodes.

You can specify the user object in the constructor, or you can set it later with the `setUserObject` method.

```
var node = new DefaultMutableTreeNode("Texas");

. . .
```

http://www.rejoiceblog.com/

```
node.setUserObject("California");
```

Next, you need to establish the parent/child relationships between the nodes. Start with the root node and use the `add` method to add the children:

```
var root = new DefaultMutableTreeNode("World");

var country = new DefaultMutableTreeNode("USA");

root.add(country);

var state = new DefaultMutableTreeNode("California");

country.add(state);
```

Figure 11.14 illustrates how the tree will look.

Figure 11.14 A simple tree

Link up all nodes in this fashion. Then, construct a `DefaultTreeModel` with the root node. Finally, construct a `JTree` with the tree model.

```
var treeModel = new DefaultTreeModel(root);

var tree = new JTree(treeModel);
```

http://www.rejoiceblog.com/

Or, as a shortcut, you can simply pass the root node to the `JTree` constructor. Then the tree automatically constructs a default tree model:

```
var tree = new JTree(root);
```

Listing 11.8 contains the complete code.

Listing 11.8 `tree/SimpleTreeFrame.java`

```
package tree;

import javax.swing.*;
import javax.swing.tree.*;

/**
 * This frame contains a simple tree that displays a manu
 */
public class SimpleTreeFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public SimpleTreeFrame()
```

http://www.rejoiceblog.com/

```
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // set up tree model data

        var root = new DefaultMutableTreeNode("World");
        var country = new DefaultMutableTreeNode("USA");
        root.add(country);
        var state = new DefaultMutableTreeNode("California")
        country.add(state);
        var city = new DefaultMutableTreeNode("San Jose");
        state.add(city);
        city = new DefaultMutableTreeNode("Cupertino");
        state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
        country.add(state);
        city = new DefaultMutableTreeNode("Ann Arbor");
        state.add(city);
        country = new DefaultMutableTreeNode("Germany");
        root.add(country);
```

http://www.rejoiceblog.com/

```
        state = new DefaultMutableTreeNode("Schleswig-Holste
        country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
        state.add(city);

        // construct tree and put it in a scroll pane

        var tree = new JTree(root);
        add(new JScrollPane(tree));
    }
}
```

When you run the program, the tree first looks as in Figure 11.15. Only the root node and its children are visible. Click on the circle icons (the *handles*) to open up the subtrees. The line sticking out from the handle icon points to the right when the subtree is collapsed and down when the subtree is expanded (see Figure 11.16). We don't know what the designers of the Metal look-and-feel had in mind, but we think of the icon as a door handle. You push down on the handle to open the subtree.

http://www.rejoiceblog.com/

Figure 11.15 The initial tree display

Figure 11.16 Collapsed and expanded subtrees

Note

Of course, the display of the tree depends on the selected look-and-feel. We just described the Metal look-and-feel. In the Windows look-and-feel, the handles have the more familiar look—a "–" or "+" in a box (see Figure 11.17).

http://www.rejoiceblog.com/

Figure 11.17 A tree with the Windows look-and-feel

You can use the following magic incantation to turn off the lines joining parents and children (see Figure 11.18):

```
tree.putClientProperty("JTree.lineStyle", "None");
```

Figure 11.18 A tree with no connecting lines

Conversely, to make sure that the lines are shown, use

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

Another line style, `"Horizontal"`, is shown in Figure 11.19. The tree is displayed with

http://www.rejoiceblog.com/

horizontal lines separating only the children of the root. We aren't quite sure what it is good for.

Figure 11.19 A tree with the horizontal line style

By default, there is no handle for collapsing the root of the tree. If you like, you can add one with the call

```
tree.setShowsRootHandles(true);
```

Figure 11.20 shows the result. Now you can collapse the entire tree into the root node.

Figure 11.20 A tree with a root handle

Conversely, you can hide the root altogether. You will thus display a *forest*—a set of trees, each with its own root. You still must join all trees in the forest to a common root; then, hide the root with the instruction

http://www.rejoiceblog.com/

```
tree.setRootVisible(false);
```

Look at Figure 11.21. There appear to be two roots, labeled “USA” and “Germany.” The actual root that joins the two is made invisible.

Figure 11.21 A forest

Let’s turn from the root to the leaves of the tree. Note that the leaves have an icon different from the other nodes (see Figure 11.22).

Figure 11.22 Leaf and folder icons

When the tree is displayed, each node is drawn with an icon. There are actually three kinds of icons: a leaf icon, an opened nonleaf icon, and a closed nonleaf icon. For simplicity, we refer to the last two as folder icons.

The node renderer needs to know which icon to use for each node. By default, the

http://www.rejoiceblog.com/

decision process works like this: If the `isLeaf` method of a node returns `true`, then the leaf icon is used; otherwise, a folder icon is used.

The `isLeaf` method of the `DefaultMutableTreeNode` class returns `true` if the node has no children. Thus, nodes with children get folder icons, and nodes without children get leaf icons.

Sometimes, that behavior is not appropriate. Suppose we added a node "Montana" to our sample tree, but we're at a loss as to what cities to add. We would not want the state node to get a leaf icon because conceptually only the cities are leaves.

The `JTree` class has no idea which nodes should be leaves. It asks the tree model. If a childless node isn't automatically a conceptual leaf, you can ask the tree model to use a different criterion for leafiness—namely, to query the "allows children" node property.

For those nodes that should not have children, call

```
node.setAllowsChildren(false);
```

Then, tell the tree model to ask the value of the "allows children" property to determine whether a node should be displayed with a leaf icon. Use the `setAsksAllowsChildren` method of the `DefaultTreeModel` class to set this behavior:

```
model.setAsksAllowsChildren(true);
```

With this decision criterion, nodes that allow children get folder icons, and nodes that don't allow children get leaf icons.

Alternatively, if you construct the tree from the root node, supply the setting for the "asks allows children" property in the constructor.

```
var tree = new JTree(root, true); // nodes that don't allow c
```

javax.swing.JTree 1.2

http://www.rejoiceblog.com/

- `JTree(TreeModel model)`

constructs a tree from a tree model.

- `JTree(TreeNode root)`

- `JTree(TreeNode root, boolean asksAllowChildren)`

constructs a tree with a default tree model that displays the root and its children.

Parameters:	`root`	The root node
	`asksAllowsChildren`	`true` to use the "allows children" node property for determining whether a node is a leaf

- `void setShowsRootHandles(boolean b)`

if `b` is `true`, the root node has a handle for collapsing or expanding its children.

- `void setRootVisible(boolean b)`

if `b` is true, then the root node is displayed. Otherwise, it is hidden.

javax.swing.tree.TreeNode 1.2

- `boolean isLeaf()`

returns `true` if this node is conceptually a leaf.

- `boolean getAllowsChildren()`

returns `true` if this node can have child nodes.

javax.swing.tree.MutableTreeNode 1.2

- `void setUserObject(Object userObject)`

sets the "user object" that the tree node uses for rendering.

http://www.rejoiceblog.com/

javax.swing.tree.TreeModel 1.2

- `boolean isLeaf(Object node)`

returns `true` if `node` should be displayed as a leaf node.

javax.swing.tree.DefaultTreeModel 1.2

- `void setAsksAllowsChildren(boolean b)`

if `b` is `true`, nodes are displayed as leaves when their `getAllowsChildren` method returns `false`. Otherwise, they are displayed as leaves when their `isLeaf` method returns `true`.

javax.swing.tree.DefaultMutableTreeNode 1.2

- `DefaultMutableTreeNode(Object userObject)`

constructs a mutable tree node with the given user object.

- `void add(MutableTreeNode child)`

adds a node as the last child of this node.

- `void setAllowsChildren(boolean b)`

if `b` is `true`, children can be added to this node.

javax.swing.JComponent 1.2

- `void putClientProperty(Object key, Object value)`

adds a key/value pair to a small table that each component manages. This is an "escape hatch" mechanism that some Swing components use for storing properties specific to a look-and-feel.

http://www.rejoiceblog.com/

11.2.1.1 Editing Trees and Tree Paths

In the next example program, you will see how to edit a tree. Figure 11.23 shows the user interface. If you click the Add Sibling or Add Child button, the program adds a new node (with title New) to the tree. If you click the Delete button, the program deletes the currently selected node.

Figure 11.23 Editing a tree

To implement this behavior, you need to find out which tree node is currently selected. The `JTree` class has a surprising way of identifying nodes in a tree. It does not deal with tree nodes but with *paths of objects*, called *tree paths*. A tree path starts at the root and consists of a sequence of child nodes (see Figure 11.24).

Figure 11.24 A tree path

You might wonder why the `JTree` class needs the whole path. Couldn't it just get a

http://www.rejoiceblog.com/

`TreeNode` and keep calling the `getParent` method? In fact, the `JTree` class knows nothing about the `TreeNode` interface. That interface is never used by the `TreeModel` interface; it is only used by the `DefaultTreeModel` implementation. You can have other tree models in which the nodes do not implement the `TreeNode` interface at all. If you use a tree model that manages other types of objects, those objects might not have `getParent` and `getChild` methods. They would of course need to have some other connection to each other. It is the job of the tree model to link nodes together. The `JTree` class itself has no clue about the nature of their linkage. For that reason, the `JTree` class always needs to work with complete paths.

The `TreePath` class manages a sequence of `Object` (not `TreeNode`!) references. A number of `JTree` methods return `TreePath` objects. When you have a tree path, you usually just need to know the terminal node, which you can get with the `getLastPathComponent` method. For example, to find out the currently selected node in a tree, use the `getSelectionPath` method of the `JTree` class. You will get a `TreePath` object back, from which you can retrieve the actual node.

```
TreePath selectionPath = tree.getSelectionPath();

var selectedNode = (DefaultMutableTreeNode) selectionPath.get
```

Actually, since this particular query is so common, there is a convenience method that gives the selected node immediately:

```
var selectedNode = (DefaultMutableTreeNode) tree.getLastSelec
```

This method is not called `getSelectedNode` because the tree does not know that it contains nodes—its tree model deals only with paths of objects.

Note

Tree paths are one of the two ways in which the `JTree` class describes nodes. Quite a few `JTree` methods take or return an integer index—the *row position*. A row position is simply the row number (starting with 0) of the node

http://www.rejoiceblog.com/

in the tree display. Only visible nodes have row numbers, and the row number of a node changes if other nodes before it are expanded, collapsed, or modified. For that reason, you should avoid row positions. All `JTree` methods that use rows have equivalents that use tree paths instead.

Once you have the selected node, you can edit it. However, do not simply add children to a tree node:

```
selectedNode.add(newNode); // No!
```

If you change the structure of the nodes, you change the model but the associated view is not notified. You could send out a notification yourself, but if you use the `insertNodeInto` method of the `DefaultTreeModel` class, the model class takes care of that. For example, the following call appends a new node as the last child of the selected node and notifies the tree view:

```
model.insertNodeInto(newNode, selectedNode, selectedNode.getC
```

The analogous call `removeNodeFromParent` removes a node and notifies the view:

```
model.removeNodeFromParent(selectedNode);
```

If you keep the node structure in place but change the user object, you should call the following method:

```
model.nodeChanged(changedNode);
```

The automatic notification is a major advantage of using the `DefaultTreeModel`. If you supply your own tree model, you have to implement automatic notification by hand. (See *Core Swing* by Kim Topley for details.)

http://www.rejoiceblog.com/

Caution

The `DefaultTreeModel` class has a `reload` method that reloads the entire model. However, don't call `reload` simply to update the tree after making a few changes. When the tree is regenerated, all nodes beyond the root's children are collapsed again. It will be quite disconcerting to your users if they have to keep expanding the tree after every change.

When the view is notified of a change in the node structure, it updates the display but does not automatically expand a node to show newly added children. In particular, if a user in our sample program adds a new child node to a node for which children are currently collapsed, the new node is silently added to the collapsed subtree. This gives the user no feedback that the command was actually carried out. In such a case, you should make a special effort to expand all parent nodes so that the newly added node becomes visible. Use the `makeVisible` method of the `JTree` class for this purpose. The `makeVisible` method expects a tree path leading to the node that should become visible.

Thus, you need to construct a tree path from the root to the newly inserted node. To get a tree path, first call the `getPathToRoot` method of the `DefaultTreeModel` class. It returns a `TreeNode[]` array of all nodes from a node to the root node. Pass that array to a `TreePath` constructor.

For example, here is how you make the new node visible:

```
TreeNode[] nodes = model.getPathToRoot(newNode);

var path = new TreePath(nodes);

tree.makeVisible(path);
```

Note

It is curious that the `DefaultTreeModel` class feigns almost complete

http://www.rejoiceblog.com/

ignorance of the `TreePath` class, even though its job is to communicate with a `JTree`. The `JTree` class uses tree paths a lot, and it never uses arrays of node objects.

But now suppose your tree is contained inside a scroll pane. After the tree node expansion, the new node might still not be visible because it falls outside the viewport. To overcome that problem, call

```
tree.scrollPathToVisible(path);
```

instead of calling `makeVisible`. This call expands all nodes along the path and tells the ambient scroll pane to scroll the node at the end of the path into view (see Figure 11.25).

Figure 11.25 The scroll pane scrolls to display a new node.

By default, tree nodes cannot be edited. However, if you call

```
tree.setEditable(true);
```

the user can edit a node simply by double-clicking, editing the string, and pressing the Enter key. Double-clicking invokes the *default cell editor*, which is implemented by the `DefaultCellEditor` class (see Figure 11.26). It is possible to install other cell editors, using the same process that you have seen in our discussion of table cell editors.

http://www.rejoiceblog.com/

Figure 11.26 The default cell editor

Listing 11.9 shows the complete source code of the tree editing program. Run the program, add a few nodes, and edit them by double-clicking them. Observe how collapsed nodes expand to show added children and how the scroll pane keeps added nodes in the viewport.

Listing 11.9 `treeEdit/TreeEditFrame.java`

```
package treeEdit;

import java.awt.*;

import javax.swing.*;
import javax.swing.tree.*;

/**
 * A frame with a tree and buttons to edit the tree.
 */
```

http://www.rejoiceblog.com/

```
11  public class TreeEditFrame extends JFrame
12  {
13     private static final int DEFAULT_WIDTH = 400;
14     private static final int DEFAULT_HEIGHT = 200;
15
16     private DefaultTreeModel model;
17     private JTree tree;
18
19     public TreeEditFrame()
20     {
21        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22
23        // construct tree
24
25        TreeNode root = makeSampleTree();
26        model = new DefaultTreeModel(root);
27        tree = new JTree(model);
28        tree.setEditable(true);
29
```

http://www.rejoiceblog.com/

```
        // add scroll pane with tree

        var scrollPane = new JScrollPane(tree);
        add(scrollPane, BorderLayout.CENTER);

        makeButtons();
    }

    public TreeNode makeSampleTree()
    {
        var root = new DefaultMutableTreeNode("World");
        var country = new DefaultMutableTreeNode("USA");
        root.add(country);
        var state = new DefaultMutableTreeNode("California"
        country.add(state);
        var city = new DefaultMutableTreeNode("San Jose");
        state.add(city);
        city = new DefaultMutableTreeNode("San Diego");
        state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
```

http://www.rejoiceblog.com/

```
        country.add(state);
        city = new DefaultMutableTreeNode("Ann Arbor");
        state.add(city);
        country = new DefaultMutableTreeNode("Germany");
        root.add(country);
        state = new DefaultMutableTreeNode("Schleswig-Holst
        country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
        state.add(city);
        return root;
    }

    /**
     * Makes the buttons to add a sibling, add a child, and
     */
    public void makeButtons()
    {
      var panel = new JPanel();
      var addSiblingButton = new JButton("Add Sibling");
```

http://www.rejoiceblog.com/

```
    addSiblingButton.addActionListener(event ->
        {
            var selectedNode = (DefaultMutableTreeNode) tree

            if (selectedNode == null) return;

            var parent = (DefaultMutableTreeNode) selectedNo

            if (parent == null) return;

            var newNode = new DefaultMutableTreeNode("New");

            int selectedIndex = parent.getIndex(selectedNode
            model.insertNodeInto(newNode, parent, selectedIn

            // now display new node

            TreeNode[] nodes = model.getPathToRoot(newNode);
            var path = new TreePath(nodes);
            tree.scrollPathToVisible(path);
```

http://www.rejoiceblog.com/

```
        });
        panel.add(addSiblingButton);

        var addChildButton = new JButton("Add Child");
        addChildButton.addActionListener(event ->
            {
                var selectedNode = (DefaultMutableTreeNode) tr

                if (selectedNode == null) return;

                var newNode = new DefaultMutableTreeNode("New"
                model.insertNodeInto(newNode, selectedNode, se

                // now display new node

                TreeNode[] nodes = model.getPathToRoot(newNode
                var path = new TreePath(nodes);
                tree.scrollPathToVisible(path);
            });
```

http://www.rejoiceblog.com/

```
      panel.add(addChildButton);

      var deleteButton = new JButton("Delete");
      deleteButton.addActionListener(event ->
         {
           var selectedNode = (DefaultMutableTreeNode) tre

           if (selectedNode != null && selectedNode.getPar
                  .removeNodeFromParent(selectedNode);
         });
      panel.add(deleteButton);
      add(panel, BorderLayout.SOUTH);
    }
  }
```

javax.swing.JTree 1.2

- `TreePath getSelectionPath()`

gets the path to the currently selected node, or the path to the first selected node if multiple nodes are selected. Returns `null` if no node is selected.

- `Object getLastSelectedPathComponent()`

http://www.rejoiceblog.com/

gets the node object that represents the currently selected node, or the first node if multiple nodes are selected. Returns `null` if no node is selected.

- `void makeVisible(TreePath path)`

expands all nodes along the path.

- `void scrollPathToVisible(TreePath path)`

expands all nodes along the path and, if the tree is contained in a scroll pane, scrolls to ensure that the last node on the path is visible.

javax.swing.tree.TreePath 1.2

- `Object getLastPathComponent()`

gets the last object on this path—that is, the node object that the path represents.

javax.swing.tree.TreeNode 1.2

- `TreeNode getParent()`

returns the parent node of this node.

- `TreeNode getChildAt(int index)`

looks up the child node at the given index. The index must be between `0` and `getChildCount() - 1`.

- `int getChildCount()`

returns the number of children of this node.

- `Enumeration children()`

returns an enumeration object that iterates through all children of this node.

http://www.rejoiceblog.com/

javax.swing.tree.DefaultTreeModel 1.2

- `void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)`

inserts `newChild` as a new child node of `parent` at the given index and notifies the tree model listeners.

- `void removeNodeFromParent(MutableTreeNode node)`

removes `node` from this model and notifies the tree model listeners.

- `void nodeChanged(TreeNode node)`

notifies the tree model listeners that `node` has changed.

- `void nodesChanged(TreeNode parent, int[] changedChildIndexes)`

notifies the tree model listeners that all child nodes of `parent` with the given indexes have changed.

- `void reload()`

reloads all nodes into the model. This is a drastic operation that you should use only if the nodes have changed completely because of some outside influence.

11.2.2 Node Enumeration

Sometimes you need to find a node in a tree by starting at the root and visiting all children until you have found a match. The `DefaultMutableTreeNode` class has several convenience methods for iterating through nodes.

The `breadthFirstEnumeration` and `depthFirstEnumeration` methods return enumeration objects whose `nextElement` method visits all children of the current node, using either a breadth-first or depth-first traversal. Figure 11.27 shows the traversals for a sample tree—the node labels indicate the order in which the nodes are traversed.

http://www.rejoiceblog.com/

Figure 11.27 Tree traversal orders

Breadth-first enumeration is the easiest to visualize. The tree is traversed in layers. The root is visited first, followed by all of its children, then the grandchildren, and so on.

To visualize depth-first enumeration, imagine a rat trapped in a tree-shaped maze. It rushes along the first path until it comes to a leaf. Then, it backtracks and turns around to the next path, and so on.

Computer scientists also call this *postorder traversal* because the search process visits the children before visiting the parents. The `postOrderEnumeration` method is a synonym for `depthFirstEnumeration`. For completeness, there is also a `preOrderEnumeration`, a depth-first search that enumerates parents before the children.

Here is the typical usage pattern:

```
Enumeration breadthFirst = node.breadthFirstEnumeration();

while (breadthFirst.hasMoreElements())

    do something with breadthFirst.nextElement();
```

Finally, a related method, `pathFromAncestorEnumeration`, finds a path from an ancestor to a given node and enumerates the nodes along that path. That's no big deal—

http://www.rejoiceblog.com/

it just keeps calling `getParent` until the ancestor is found and then presents the path in reverse order.

In our next example program, we put node enumeration to work. The program displays inheritance trees of classes. Type the name of a class into the text field on the bottom of the frame. The class and all of its superclasses are added to the tree (see Figure 11.28).

Figure 11.28 An inheritance tree

In this example, we take advantage of the fact that the user objects of the tree nodes can be objects of any type. Since our nodes describe classes, we store `Class` objects in the nodes.

We don't want to add the same class object twice, so we need to check whether a class already exists in the tree. The following method finds the node with a given user object if it exists in the tree.

```
public DefaultMutableTreeNode findUserObject(Object obj)

{

   Enumeration e = root.breadthFirstEnumeration();

   while (e.hasMoreElements())

   {
```

http://www.rejoiceblog.com/

```
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)

        if (node.getUserObject().equals(obj))

            return node;

    }

    return null;

}
```

11.2.3 Rendering Nodes

In your applications, you will often need to change the way a tree component draws the nodes. The most common change is, of course, to choose different icons for nodes and leaves. Other changes might involve changing the font of the node labels or drawing images at the nodes. All these changes are possible by installing a new *tree cell renderer* into the tree. By default, the `JTree` class uses `DefaultTreeCellRenderer` objects to draw each node. The `DefaultTreeCellRenderer` class extends the `JLabel` class. The label contains the node icon and the node label.

Note

The cell renderer does not draw the “handles” for expanding and collapsing subtrees. The handles are part of the look-and-feel, and it is recommended that you do not change them.

You can customize the display in three ways.

• You can change the icons, font, and background color used by a `DefaultTreeCellRenderer`. These settings are used for all nodes in the tree.

• You can install a renderer that extends the `DefaultTreeCellRenderer` class and vary the icons, fonts, and background color for each node.

http://www.rejoiceblog.com/

• You can install a renderer that implements the `TreeCellRenderer` interface to draw a custom image for each node.

Let us look at these possibilities one by one. The easiest customization is to construct a `DefaultTreeCellRenderer` object, change the icons, and install it into the tree:

```
var renderer = new DefaultTreeCellRenderer();

renderer.setLeafIcon(new ImageIcon("blue-ball.gif")); // used

renderer.setClosedIcon(new ImageIcon("red-ball.gif")); // use

renderer.setOpenIcon(new ImageIcon("yellow-ball.gif")); // us

tree.setCellRenderer(renderer);
```

You can see the effect in Figure 11.28. We just use the "ball" icons as placeholders—presumably your user interface designer would supply you with appropriate icons to use for your applications.

We don't recommend that you change the font or background color for an entire tree—that is really the job of the look-and-feel.

However, it can be useful to change the font of some nodes in a tree to highlight them. If you look carefully at Figure 11.28, you will notice that the *abstract* classes are set in italics.

To change the appearance of individual nodes, install a tree cell renderer. Tree cell renderers are very similar to the list cell renderers we discussed earlier in this chapter. The `TreeCellRenderer` interface has a single method:

```
Component getTreeCellRendererComponent(JTree tree, Object val

      boolean expanded, boolean leaf, int row, boolean hasFoc
```

The `getTreeCellRendererComponent` method of the

http://www.rejoiceblog.com/

`DefaultTreeCellRenderer` class returns `this`—in other words, a label. (The `DefaultTreeCellRenderer` class extends the `JLabel` class.) To customize the component, extend the `DefaultTreeCellRenderer` class. Override the `getTreeCellRendererComponent` method as follows: Call the superclass method so it can prepare the label data, customize the label properties, and finally return `this`.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer

{

   public Component getTreeCellRendererComponent(JTree tree, 

        boolean expanded, boolean leaf, int row, boolean hasF

   {

     Component comp = super.getTreeCellRendererComponent(tree

           expanded, leaf, row, hasFocus);

     DefaultMutableTreeNode node = (DefaultMutableTreeNode) v

     look at node.getUserObject();

     Font font = appropriate font;

     comp.setFont(font);

     return comp;

   }

};
```

Caution

http://www.rejoiceblog.com/

The `value` parameter of the `getTreeCellRendererComponent` method is the *node* object, *not* the user object! Recall that the user object is a feature of the `DefaultMutableTreeNode`, and that a `JTree` can contain nodes of an arbitrary type. If your tree uses `DefaultMutableTreeNode` nodes, you must retrieve the user object in a second step, as we did in the preceding code sample.

Caution

The `DefaultTreeCellRenderer` uses the *same* label object for all nodes, only changing the label text for each node. If you change the font for a particular node, you must set it back to its default value when the method is called again. Otherwise, all subsequent nodes will be drawn in the changed font! Look at the code in Listing 11.10 to see how to restore the font to the default.

The `ClassNameTreeCellRenderer` in Listing 11.10 sets the class name in either the normal or italic font, depending on the `ABSTRACT` modifier of the `Class` object. We don't want to set a particular font because we don't want to change whatever font the look-and-feel normally uses for labels. For that reason, we use the font from the label and *derive* an italic font from it. Recall that only a single shared `JLabel` object is returned by all calls. We need to hang on to the original font and restore it in the next call to the `getTreeCellRendererComponent` method.

Also, note how we change the node icons in the `ClassTreeFrame` constructor.

javax.swing.tree.DefaultMutableTreeNode 1.2

- `Enumeration breadthFirstEnumeration()`
- `Enumeration depthFirstEnumeration()`
- `Enumeration preOrderEnumeration()`
- `Enumeration postOrderEnumeration()`

http://www.rejoiceblog.com/

returns enumeration objects for visiting all nodes of the tree model in a particular order. In breadth-first traversal, children that are closer to the root are visited before those that are farther away. In depth-first traversal, all children of a node are completely enumerated before its siblings are visited. The `postOrderEnumeration` method is a synonym for `depthFirstEnumeration`. The preorder traversal is identical to the postorder traversal except that parents are enumerated before their children.

javax.swing.tree.TreeCellRenderer 1.2

- `Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)`

returns a component whose `paint` method is invoked to render a tree cell.

Parameters:	`tree`	The tree containing the node to be rendered
	`value`	The node to be rendered
	`selected`	`true` if the node is currently selected
	`expanded`	`true` if the children of the node are visible
	`leaf`	`true` if the node needs to be displayed as a leaf
	`row`	The display row containing the node
	`hasFocus`	`true` if the node currently has input focus

javax.swing.tree.DefaultTreeCellRenderer 1.2

- `void setLeafIcon(Icon icon)`
- `void setOpenIcon(Icon icon)`
- `void setClosedIcon(Icon icon)`

sets the icon to show for a leaf node, an expanded node, and a collapsed node.

11.2.4 Listening to Tree Events

Most commonly, a tree component is paired with some other component. When the

http://www.rejoiceblog.com/

user selects tree nodes, some information shows up in another window. See Figure 11.29 for an example. When the user selects a class, the instance and static variables of that class are displayed in the text area to the right.

Figure 11.29 A class browser

To obtain this behavior, you need to install a *tree selection listener*. The listener must implement the `TreeSelectionListener` interface—an interface with a single method:

```
void valueChanged(TreeSelectionEvent event)
```

That method is called whenever the user selects or deselects tree nodes.

Add the listener to the tree in the normal way:

```
tree.addTreeSelectionListener(listener);
```

You can specify whether the user is allowed to select a single node, a contiguous range of nodes, or an arbitrary, potentially discontiguous, set of nodes. The `JTree` class uses a `TreeSelectionModel` to manage node selection. You need to retrieve the model to set the selection state to one of `SINGLE_TREE_SELECTION`, `CONTIGUOUS_TREE_SELECTION`, or `DISCONTIGUOUS_TREE_SELECTION`. (Discontiguous selection

http://www.rejoiceblog.com/

mode is the default.) For example, in our class browser, we want to allow selection of only a single class:

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;

tree.getSelectionModel().setSelectionMode(mode);
```

Apart from setting the selection mode, you need not worry about the tree selection model.

> **Note**
>
> How the user selects multiple items depends on the look-and-feel. In the Metal look-and-feel, hold down the Ctrl key while clicking an item to add it to the selection, or to remove it if it is currently selected. Hold down the Shift key while clicking an item to select a *range* of items, extending from the previously selected item to the new item.

To find out the current selection, query the tree with the `getSelectionPaths` method:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

If you restricted the user to single-item selection, you can use the convenience method `getSelectionPath` which returns the first selected path or `null` if no path was selected.

> **Caution**
>
> The `TreeSelectionEvent` class has a `getPaths` method that returns an array of `TreePath` objects, but that array describes *selection changes*, not the current selection.

http://www.rejoiceblog.com/

Listing 11.10 shows the frame class for the class tree program. The program displays inheritance hierarchies and customizes the display to show abstract classes in italics. (See Listing 11.11 for the cell renderer.) You can type the name of any class into the text field at the bottom of the frame. Press the Enter key or click the Add button to add the class and its superclasses to the tree. You must enter the full package name, such as `java.util.ArrayList`.

This program is a bit tricky because it uses reflection to construct the class tree. This work is done inside the `addClass` method. (The details are not that important. We use the class tree in this example because inheritance yields a nice supply of trees without laborious coding. When you display trees in your applications, you will have your own source of hierarchical data.) The method uses the breadth-first search algorithm to find whether the current class is already in the tree by calling the `findUserObject` method that we implemented in the preceding section. If the class is not already in the tree, we add the superclasses to the tree, then make the new class node a child and make that node visible.

When you select a tree node, the text area to the right is filled with the fields of the selected class. In the frame constructor, we restrict the user to single-item selection and add a tree selection listener. When the `valueChanged` method is called, we ignore its event parameter and simply ask the tree for the current selection path. As always, we need to get the last node of the path and look up its user object. We then call the `getFieldDescription` method which uses reflection to assemble a string with all fields of the selected class.

Listing 11.10 `treeRender/ClassTreeFrame.java`

```
package treeRender;

import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
```

http://www.rejoiceblog.com/

```
import javax.swing.*;
import javax.swing.tree.*;

/**
 * This frame displays the class tree, a text field, and
 * into the tree.
 */
public class ClassTreeFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;

   private DefaultMutableTreeNode root;
   private DefaultTreeModel model;
   private JTree tree;
   private JTextField textField;
   private JTextArea textArea;

   public ClassTreeFrame()
```

http://www.rejoiceblog.com/

```
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // the root of the class tree is Object
        root = new DefaultMutableTreeNode(java.lang.Object
        model = new DefaultTreeModel(root);
        tree = new JTree(model);

        // add this class to populate the tree with some d
        addClass(getClass());

        // set up node icons
        var renderer = new ClassNameTreeCellRenderer();
        renderer.setClosedIcon(new ImageIcon(getClass().ge
        renderer.setOpenIcon(new ImageIcon(getClass().getR
        renderer.setLeafIcon(new ImageIcon(getClass().getR
        tree.setCellRenderer(renderer);

        // set up selection mode
```

http://www.rejoiceblog.com/

```
        tree.addTreeSelectionListener(event ->
           {
             // the user selected a different node--update
             TreePath path = tree.getSelectionPath();
             if (path == null) return;
             var selectedNode = (DefaultMutableTreeNode) p
             Class<?> c = (Class<?>) selectedNode.getUserO
             String description = getFieldDescription(c);
             textArea.setText(description);
           });
        int mode = TreeSelectionModel.SINGLE_TREE_SELECTIO
        tree.getSelectionModel().setSelectionMode(mode);

        // this text area holds the class description
        textArea = new JTextArea();

        // add tree and text area
        var panel = new JPanel();
        panel.setLayout(new GridLayout(1, 2));
        panel.add(new JScrollPane(tree));
```

http://www.rejoiceblog.com/

```
        panel.add(new JScrollPane(textArea));

        add(panel, BorderLayout.CENTER);

        addTextField();
    }

    /**
     * Add the text field and "Add" button to add a new cl
     */
    public void addTextField()
    {
       var panel = new JPanel();

       ActionListener addListener = event ->
          {
            // add the class whose name is in the text fie
            try
                 {
```

http://www.rejoiceblog.com/

```
                    String text = textField.getText();
                    addClass(Class.forName(text)); // clea
                    textField.setText("");
                }
                catch (ClassNotFoundException e)
                {
                    JOptionPane.showMessageDialog(null, "C
                }
            };

        // new class names are typed into this text field
        textField = new JTextField(20);
        textField.addActionListener(addListener);
        panel.add(textField);

        var addButton = new JButton("Add");
        addButton.addActionListener(addListener);
        panel.add(addButton);

        add(panel, BorderLayout.SOUTH);
```

http://www.rejoiceblog.com/

```
    }

    /**
     * Finds an object in the tree.
     * @param obj the object to find
     * @return the node containing the object or null if
     */
    public DefaultMutableTreeNode findUserObject(Object o
    {
        // find the node containing a user object
        var e = (Enumeration<TreeNode>) root.breadthFirstE
        while (e.hasMoreElements())
        {
          var node = (DefaultMutableTreeNode) e.nextElemen
          if (node.getUserObject().equals(obj)) return nod
        }
        return null;
    }

```

http://www.rejoiceblog.com/

```
    /**
     * Adds a new class and any parent classes that aren't
     * @param c the class to add
     * @return the newly added node.
     */
    public DefaultMutableTreeNode addClass(Class<?> c)
    {
      // add a new class to the tree

      // skip non-class types
      if (c.isInterface() || c.isPrimitive()) return null

      // if the class is already in the tree, return its
      DefaultMutableTreeNode node = findUserObject(c);
      if (node != null) return node;

      // class isn't present--first add class parent recu

      Class<?> s = c.getSuperclass();

```

http://www.rejoiceblog.com/

```
        DefaultMutableTreeNode parent;
        if (s == null) parent = root;
        else parent = addClass(s);

        // add the class as a child to the parent
        var newNode = new DefaultMutableTreeNode(c);
        model.insertNodeInto(newNode, parent, parent.getChi

        // make node visible
        var path = new TreePath(model.getPathToRoot(newNode
        tree.makeVisible(path);

        return newNode;
    }

    /**
     * Returns a description of the fields of a class.
     * @param the class to be described
     * @return a string containing all field types and na
```

http://www.rejoiceblog.com/

```
     */
    public static String getFieldDescription(Class<?> c)
    {
      // use reflection to find types and names of fields
      var r = new StringBuilder();
      Field[] fields = c.getDeclaredFields();
      for (int i = 0; i < fields.length; i++)
      {
         Field f = fields[i];
         if ((f.getModifiers() & Modifier.STATIC) != 0) r
         r.append(f.getType().getName());
         r.append(" ");
         r.append(f.getName());
         r.append("\n");
      }
      return r.toString();
    }
}
```

Listing 11.11 `treeRender/ClassNameTreeCellRenderer.java`

http://www.rejoiceblog.com/

```
1  package treeRender;
2
3  import java.awt.*;
4  import java.lang.reflect.*;
5  import javax.swing.*;
6  import javax.swing.tree.*;
7
8  /**
9   * This class renders a class name either in plain or ita
10  */
11 public class ClassNameTreeCellRenderer extends DefaultTre
12 {
13   private Font plainFont = null;
14   private Font italicFont = null;
15
16   public Component getTreeCellRendererComponent(JTree tre
17         boolean expanded, boolean leaf, int row, boolean h
18   {
19     super.getTreeCellRendererComponent(tree, value, selec
20     // get the user object
```

http://www.rejoiceblog.com/

```
21      var node = (DefaultMutableTreeNode) value;
22      Class<?> c = (Class<?>) node.getUserObject();
23
24      // the first time, derive italic font from plain font
25      if (plainFont == null)
26      {
27         plainFont = getFont();
28         // the tree cell renderer is sometimes called with
29         if (plainFont != null) italicFont = plainFont.deri
30      }
31
32      // set font to italic if the class is abstract, plain
33      if ((c.getModifiers() & Modifier.ABSTRACT) == 0) setF
34      else setFont(italicFont);
35      return this;
36   }
37 }
```

javax.swing.JTree 1.2

http://www.rejoiceblog.com/

- `TreePath getSelectionPath()`
- `TreePath[] getSelectionPaths()`

returns the first selected path, or an array of paths to all selected nodes. If no paths are selected, both methods return `null`.

javax.swing.event.TreeSelectionListener 1.2

- `void valueChanged(TreeSelectionEvent event)`

is called whenever nodes are selected or deselected.

javax.swing.event.TreeSelectionEvent 1.2

- `TreePath getPath()`
- `TreePath[] getPaths()`

gets the first path or all paths that have *changed* in this selection event. If you want to know the current selection, not the selection change, call `JTree.getSelectionPaths` instead.

11.2.5 Custom Tree Models

In the final example, we implement a program that inspects the contents of an object, just like a debugger does (see Figure 11.30).

http://www.rejoiceblog.com/

Figure 11.30 An object inspection tree

Before going further, compile and run the example program. Each node corresponds to an instance field. If the field is an object, expand it to see *its* instance fields. The program inspects the contents of the frame window. If you poke around a few of the instance fields, you should be able to find some familiar classes. You'll also gain some respect for how complex the Swing user interface components are under the hood.

What's remarkable about the program is that the tree does not use the `DefaultTreeModel`. If you already have data that are hierarchically organized, you might not want to build a duplicate tree and worry about keeping both trees synchronized. That is the situation in our case—the inspected objects are already linked to each other through the object references, so there is no need to replicate the linking structure.

The `TreeModel` interface has only a handful of methods. The first group of methods enables the `JTree` to find the tree nodes by first getting the root, then the children. The `JTree` class calls these methods only when the user actually expands a node.

```
Object getRoot()

int getChildCount(Object parent)

Object getChild(Object parent, int index)
```

http://www.rejoiceblog.com/

This example shows why the `TreeModel` interface, like the `JTree` class itself, does not need an explicit notion of nodes. The root and its children can be any objects. The `TreeModel` is responsible for telling the `JTree` how they are connected.

The next method of the `TreeModel` interface is the reverse of `getChild`:

```
int getIndexOfChild(Object parent, Object child)
```

Actually, this method can be implemented in terms of the first three—see the code in Listing 11.12.

The tree model tells the `JTree` which nodes should be displayed as leaves:

```
boolean isLeaf(Object node)
```

If your code changes the tree model, the tree needs to be notified so that it can redraw itself. The tree adds itself as a `TreeModelListener` to the model. Thus, the model must support the usual listener management methods:

```
void addTreeModelListener(TreeModelListener l)

void removeTreeModelListener(TreeModelListener l)
```

You can see the implementations for these methods in Listing 11.13.

When the model modifies the tree contents, it calls one of the four methods of the `TreeModelListener` interface:

```
void treeNodesChanged(TreeModelEvent e)

void treeNodesInserted(TreeModelEvent e)

void treeNodesRemoved(TreeModelEvent e)

void treeStructureChanged(TreeModelEvent e)
```

http://www.rejoiceblog.com/

The `TreeModelEvent` object describes the location of the change. The details of assembling a tree model event that describes an insertion or removal event are quite technical. You only need to worry about firing these events if your tree can actually have nodes added and removed. In Listing 11.12, we show how to fire one event by replacing the root with a new object.

Tip

To simplify the code for event firing, use the `javax.swing.EventListenerList` convenience class that collects listeners. The last three methods of Listing 11.13 show how to use the class.

Finally, if the user edits a tree node, your model is called with the change:

```
void valueForPathChanged(TreePath path, Object newValue)
```

If you don't allow editing, this method is never called.

If you don't need to support editing, constructing a tree model is easily done. Implement the three methods

```
Object getRoot()

int getChildCount(Object parent)

Object getChild(Object parent, int index)
```

These methods describe the structure of the tree. Supply routine implementations of the other five methods, as in Listing 11.12. You are then ready to display your tree.

Now let's turn to the implementation of the example program. Our tree will contain objects of type `Variable`.

http://www.rejoiceblog.com/

Note

Had we used the `DefaultTreeModel`, our nodes would have been objects of type `DefaultMutableTreeNode` with *user objects* of type `Variable`.

For example, suppose you inspect the variable

```
Employee joe;
```

That variable has a *type* `Employee.class`, a *name* `"joe"`, and a *value*—the value of the object reference `joe`. In Listing 11.14, we define a class `Variable` that describes a variable in a program:

```
var v = new Variable(Employee.class, "joe", joe);
```

If the type of the variable is a primitive type, you must use an object wrapper for the value.

```
new Variable(double.class, "salary", new Double(salary));
```

If the type of the variable is a class, the variable has *fields*. Using reflection, we enumerate all fields and collect them in an `ArrayList`. Since the `getFields` method of the `Class` class does not return the fields of the superclass, we need to call `getFields` on all superclasses as well. You can find the code in the `Variable` constructor. The `getFields` method of our `Variable` class returns the array of fields. Finally, the `toString` method of the `Variable` class formats the node label. The label always contains the variable type and name. If the variable is not a class, the label also contains the value.

Note

If the type is an array, we do not display the elements of the array. This

http://www.rejoiceblog.com/

> would not be difficult to do; we leave it as the proverbial “exercise for the reader.”

Let’s move on to the tree model. The first two methods are simple.

```
public Object getRoot()
{
    return root;
}
public int getChildCount(Object parent)
{
    return ((Variable) parent).getFields().size();
}
```

The `getChild` method returns a new `Variable` object that describes the field with the given index. The `getType` and `getName` methods of the `Field` class yield the field type and name. By using reflection, you can read the field value as `f.get(parentValue)`. That method can throw an `IllegalAccessException`. However, we made all fields accessible in the `Variable` constructor, so this won’t happen in practice.

Here is the complete code of the `getChild` method:

```
public Object getChild(Object parent, int index)
{
   ArrayList fields = ((Variable) parent).getFields();
```

http://www.rejoiceblog.com/

```
      var f = (Field) fields.get(index);

      Object parentValue = ((Variable) parent).getValue();

      try

      {

        return new Variable(f.getType(), f.getName(), f.get(pare

      }

      catch (IllegalAccessException e)

      {

        return null;

      }

   }
```

These three methods reveal the structure of the object tree to the `JTree` component. The remaining methods are routine—see the source code in Listing 11.13.

There is one remarkable fact about this tree model: It actually describes an *infinite* tree. You can verify this by following one of the `WeakReference` objects. Click on the variable named `referent`. It leads you right back to the original object. You get an identical subtree, and you can open its `WeakReference` object again, ad infinitum. Of course, you cannot *store* an infinite set of nodes; the tree model simply generates the nodes on demand as the user expands the parents. Listing 11.12 shows the frame class of the sample program.

Listing 11.12 `treeModel/ObjectInspectorFrame.java`

```
1  package treeModel;
```

http://www.rejoiceblog.com/

```
 2
 3  import java.awt.*;
 4  import javax.swing.*;
 5
 6  /**
 7   * This frame holds the object tree.
 8   */
 9  public class ObjectInspectorFrame extends JFrame
10  {
11     private JTree tree;
12     private static final int DEFAULT_WIDTH = 400;
13     private static final int DEFAULT_HEIGHT = 300;
14
15     public ObjectInspectorFrame()
16     {
17       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19       // we inspect this frame object
20
21       var v = new Variable(getClass(), "this", this);
```

http://www.rejoiceblog.com/

```
22        var model = new ObjectTreeModel();
23        model.setRoot(v);
24
25        // construct and show tree
26
27        tree = new JTree(model);
28        add(new JScrollPane(tree), BorderLayout.CENTER);
29     }
30  }
```

Listing 11.13 `treeModel/ObjectTreeModel.java`

```
1  package treeModel;
2
3  import java.lang.reflect.*;
4  import java.util.*;
5  import javax.swing.event.*;
6  import javax.swing.tree.*;
7
8  /**
9   * This tree model describes the tree structure of a Java
```

http://www.rejoiceblog.com/

```
 * stored in instance variables.
 */
public class ObjectTreeModel implements TreeModel
{
    private Variable root;
    private EventListenerList listenerList = new EventList

    /**
     * Constructs an empty tree.
     */
    public ObjectTreeModel()
    {
      root = null;
    }

    /**
     * Sets the root to a given variable.
     * @param v the variable that is being described by th
     */
```

http://www.rejoiceblog.com/

```
    public void setRoot(Variable v)
    {
      Variable oldRoot = v;
      root = v;
      fireTreeStructureChanged(oldRoot);
    }

    public Object getRoot()
    {
      return root;
    }

    public int getChildCount(Object parent)
    {
      return ((Variable) parent).getFields().size();
    }

    public Object getChild(Object parent, int index)
    {
      ArrayList<Field> fields = ((Variable) parent).getFie
```

http://www.rejoiceblog.com/

```
      var f = (Field) fields.get(index);
      Object parentValue = ((Variable) parent).getValue();
      try
      {
        return new Variable(f.getType(), f.getName(), f.ge
      }
      catch (IllegalAccessException e)
      {
        return null;
      }
    }

    public int getIndexOfChild(Object parent, Object child
    {
      int n = getChildCount(parent);
      for (int i = 0; i < n; i++)
        if (getChild(parent, i).equals(child)) return i;
      return -1;
    }
```

http://www.rejoiceblog.com/

```
68
69      public boolean isLeaf(Object node)
70      {
71        return getChildCount(node) == 0;
72      }
73
74      public void valueForPathChanged(TreePath path, Object
75      {
76      }
77
78      public void addTreeModelListener(TreeModelListener l)
79      {
80        listenerList.add(TreeModelListener.class, l);
81      }
82
83      public void removeTreeModelListener(TreeModelListener
84      {
85        listenerList.remove(TreeModelListener.class, l);
86      }
87
```

http://www.rejoiceblog.com/

```
    protected void fireTreeStructureChanged(Object oldRoot
    {
      var event = new TreeModelEvent(this, new Object[] {
      for (TreeModelListener l : listenerList.getListeners
        l.treeStructureChanged(event);
    }
}
```

Listing 11.14 `treeModel/Variable.java`

```
package treeModel;

import java.lang.reflect.*;
import java.util.*;

/**
 * A variable with a type, name, and value.
 */
public class Variable
{
```

http://www.rejoiceblog.com/

```
    private Class<?> type;

    private String name;

    private Object value;

    private ArrayList<Field> fields;

    /**
     * Construct a variable.
     * @param aType the type
     * @param aName the name
     * @param aValue the value
     */
    public Variable(Class<?> aType, String aName, Object a
    {
      type = aType;
      name = aName;
      value = aValue;
      fields = new ArrayList<>();

      // find all fields if we have a class type except we

```

http://www.rejoiceblog.com/

```
31        if (!type.isPrimitive() && !type.isArray() && !type.
32        {
33           // get fields from the class and all superclasses
34           for (Class<?> c = value.getClass(); c != null; c
35           {
36              Field[] fs = c.getDeclaredFields();
37              AccessibleObject.setAccessible(fs, true);
38
39           // get all nonstatic fields
40           for (Field f : fs)
41              if ((f.getModifiers() & Modifier.STATIC) == 0)
42           }
43        }
44     }
45
46   /**
47    * Gets the value of this variable.
48    * @return the value
49    */
```

http://www.rejoiceblog.com/

```
50   public Object getValue()
51   {
52     return value;
53   }
54
55   /**
56    * Gets all nonstatic fields of this variable.
57    * @return an array list of variables describing the fie
58    */
59   public ArrayList<Field> getFields()
60   {
61     return fields;
62   }
63
64   public String toString()
65   {
66     String r = type + " " + name;
67     if (type.isPrimitive()) r += "=" + value;
68     else if (type.equals(String.class)) r += "=" + value;
69     else if (value == null) r += "=null";
```

http://www.rejoiceblog.com/

```
    return r;
  }
}
```

javax.swing.tree.TreeModel 1.2

- `Object getRoot()`

returns the root node.

- `int getChildCount(Object parent)`

gets the number of children of the `parent` node.

- `Object getChild(Object parent, int index)`

gets the child node of the `parent` node at the given index.

- `int getIndexOfChild(Object parent, Object child)`

gets the index of the `child` node in the `parent` node, or `-1` if `child` is not a child of `parent` in this tree model.

- `boolean isLeaf(Object node)`

returns `true` if `node` is conceptually a leaf of the tree.

- `void addTreeModelListener(TreeModelListener l)`
- `void removeTreeModelListener(TreeModelListener l)`

adds or removes listeners that are notified when the information in the tree model changes.

- `void valueForPathChanged(TreePath path, Object newValue)`

is called when a cell editor has modified the value of a node.

http://www.rejoiceblog.com/

Parameters:	`path`	The path to the node that has been edited
	`newValue`	The replacement value returned by the editor

javax.swing.event.TreeModelListener 1.2

- `void treeNodesChanged(TreeModelEvent e)`
- `void treeNodesInserted(TreeModelEvent e)`
- `void treeNodesRemoved(TreeModelEvent e)`
- `void treeStructureChanged(TreeModelEvent e)`

is called by the tree model when the tree has been modified.

javax.swing.event.TreeModelEvent 1.2

- `TreeModelEvent(Object eventSource, TreePath node)`

constructs a tree model event.

Parameters:	`eventSource`	The tree model generating this event
	`node`	The path to the node that is being changed

11.3 ADVANCED AWT

You can use the methods of the `Graphics` class to create simple drawings. Those methods are sufficient for simple applications, but they fall short when you need to create complex shapes or require complete control over the appearance of the graphics. The Java 2D API is a more sophisticated class library that you can use to produce high-quality drawings. In the following sections, we will give you an overview of that API.

11.3.1 The Rendering Pipeline

The original JDK 1.0 had a very simple mechanism for drawing shapes. You selected color and paint mode, and called methods of the `Graphics` class such as `drawRect` or `fillOval`. The Java 2D API supports many more options.

http://www.rejoiceblog.com/

• You can easily produce a wide variety of *shapes*.

• You have control over the *stroke*—the pen that traces shape boundaries.

• You can *fill* shapes with solid colors, varying hues, and repeating patterns.

• You can use *transformations* to move, scale, rotate, or stretch shapes.

• You can *clip* shapes to restrict them to arbitrary areas.

• You can select *composition rules* to describe how to combine the pixels of a new shape with existing pixels.

To draw a shape, go through the following steps:

1. Obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. Ever since Java SE 1.2, methods such as `paint` and `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g)
{
   var g2 = (Graphics2D) g;
   . . .
}
```

2. Use the `setRenderingHints` method to set *rendering hints*—trade-offs between speed and drawing quality.

```
RenderingHints hints = . . .;
g2.setRenderingHints(hints);
```

3. Use the `setStroke` method to set the *stroke*. The stroke draws the outline of the shape. You can select the thickness and choose among solid and dotted lines.

http://www.rejoiceblog.com/

```
Stroke stroke = . . .;

g2.setStroke(stroke);
```

4. Use the `setPaint` method to set the *paint*. The paint fills areas such as the stroke path or the interior of a shape. You can create solid color paint, paint with changing hues, or tiled fill patterns.

```
Paint paint = . . .;

g2.setPaint(paint);
```

5. Use the `clip` method to set the *clipping region*.

```
Shape clip = . . .;

g2.clip(clip);
```

6. Use the `transform` method to set a *transformation* from user space to device space. Use transformations if it is easier for you to define your shapes in a custom coordinate system than by using pixel coordinates.

```
AffineTransform transform = . . .;

g2.transform(transform);
```

7. Use the `setComposite` method to set a *composition rule* that describes how to combine the new pixels with the existing pixels.

```
Composite composite = . . .;

g2.setComposite(composite);
```

8. Create a shape. The Java 2D API supplies many shape objects and methods to

http://www.rejoiceblog.com/

combine shapes.

```
Shape shape = . . .;
```

9. Draw or fill the shape. If you draw the shape, its outline is stroked. If you fill the shape, the interior is painted.

```
g2.draw(shape);

g2.fill(shape);
```

Of course, in many practical circumstances, you don't need all these steps. There are reasonable defaults for the settings of the 2D graphics context; change the settings only if you want to deviate from the defaults.

In the following sections, you will see how to describe shapes, strokes, paints, transformations, and composition rules.

The various `set` methods simply set the state of the 2D graphics context. They don't cause any drawing. Similarly, when you construct `Shape` objects, no drawing takes place. A shape is only rendered when you call `draw` or `fill`. At that time, the new shape is computed in a *rendering pipeline* (see Figure 11.31).

Figure 11.31 The rendering pipeline

In the rendering pipeline, the following steps take place to render a shape:

1. The path of the shape is stroked.

2. The shape is transformed.

http://www.rejoiceblog.com/

3. The shape is clipped. If there is no intersection between the shape and the clipping area, the process stops.

4. The remainder of the shape after clipping is filled.

5. The pixels of the filled shape are composed with the existing pixels. (In Figure 11.31, the circle is part of the existing pixels, and the cup shape is superimposed over it.)

In the next section, you will see how to define shapes. Then, we will turn to the 2D graphics context settings.

java.awt.Graphics2D 1.2
• `void draw(Shape s)` draws the outline of the given shape with the current paint. • `void fill(Shape s)` fills the interior of the given shape with the current paint.

11.3.2 Shapes

Here are some of the methods in the `Graphics` class to draw shapes:

```
drawLine

drawRectangle

drawRoundRect

draw3DRect

drawPolygon

drawPolyline

drawOval
```

http://www.rejoiceblog.com/

```
drawArc
```

There are also corresponding `fill` methods. These methods have been in the `Graphics` class ever since JDK 1.0. The Java 2D API uses a completely different, object-oriented approach. Instead of methods, there are classes:

```
Line2D

Rectangle2D

RoundRectangle2D

Ellipse2D

Arc2D

QuadCurve2D

CubicCurve2D

GeneralPath
```

These classes all implement the `Shape` interface, which we will examine in the following sections.

11.3.2.1 The Shape Class Hierarchy

The `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D`, and `Arc2D` classes correspond to the `drawLine`, `drawRectangle`, `drawRoundRect`, `drawOval`, and `drawArc` methods. (The concept of a "3D rectangle" has died the death it so richly deserved—there is no analog to the `draw3DRect` method.) The Java 2D API supplies two additional classes, quadratic and cubic curves, that we will discuss in this section. There is no `Polygon2D` class; instead, the `GeneralPath` class describes paths made up from lines, quadratic and cubic curves. You can use a `GeneralPath` to describe a polygon; we'll show you how later in this section.

To draw a shape, first create an object of a class that implements the `Shape` interface and then call the `draw` method of the `Graphics2D` class.

http://www.rejoiceblog.com/

The classes

```
Rectangle2D

RoundRectangle2D

Ellipse2D

Arc2D
```

all inherit from a common superclass `RectangularShape`. Admittedly, ellipses and arcs are not rectangular, but they have a *bounding rectangle* (see Figure 11.32).

Figure 11.32 The bounding rectangle of an ellipse and an arc

Each of the classes with a name ending in "2D" has two subclasses for specifying coordinates as `float` or `double` quantities. In Volume I, you already encountered `Rectangle2D.Float` and `Rectangle2D.Double`.

The same scheme is used for the other classes, such as `Arc2D.Float` and `Arc2D.Double`.

Internally, all graphics classes use `float` coordinates because `float` numbers use less storage space but have sufficient precision for geometric computations. However, the Java programming language makes it a bit more tedious to manipulate `float` numbers. For that reason, most methods of the graphics classes use `double` parameters and return values. Only when constructing a 2D object you need to choose between the constructors with `float` and `double` co-ordinates. For example,

```
var floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
```

http://www.rejoiceblog.com/

```
var doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

The *Xxx*`2D.Float` and *Xxx*`2D.Double` classes are subclasses of the *Xxx*`2D` classes. After object construction, essentially no benefit accrues from remembering the subclass, and you can just store the constructed object in a superclass variable as in the code example above.

As you can see from the curious names, the *Xxx*`2D.Float` and *Xxx*`2D.Double` classes are also inner classes of the *Xxx*`2D` classes. That is just a minor syntactical convenience to avoid inflation of outer class names.

Finally, the `Point2D` class describes a point with an *x* and a *y* coordinate. Points are used to define shapes, but they aren't themselves shapes.

Figure 11.33 shows the relationships between the shape classes. However, the `Double` and `Float` subclasses are omitted. Legacy classes from the pre-2D library are marked with a gray fill.

http://www.rejoiceblog.com/

Figure 11.33 Relationships between the shape classes

11.3.2.2 Using the Shape Classes

You already saw how to use the `Rectangle2D`, `Ellipse2D`, and `Line2D` classes in Volume I, Chapter 10. In this section, you will learn how to work with the remaining 2D shapes.

For the `RoundRectangle2D` shape, specify the top left corner, width, height, and the *x* and *y* dimensions of the corner area that should be rounded (see Figure 11.34). For example, the call

```
var r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20
```

http://www.rejoiceblog.com/
produces a rounded rectangle with circles of radius 20 at each of the corners.

Figure 11.34 Constructing a `RoundRectangle2D`

To construct an arc, specify the bounding box, the start angle, the angle swept out by the arc (see Figure 11.35), and the closure type—one of `Arc2D.OPEN`, `Arc2D.PIE`, or `Arc2D.CHORD`.

```
var a = new Arc2D(x, y, width, height, startAngle, arcAngle,
```

http://www.rejoiceblog.com/

Figure 11.35 Constructing an elliptical arc

Figure 11.36 illustrates the arc types.

http://www.rejoiceblog.com/

Figure 11.36 Arc types

Caution

If the arc is elliptical, the computation of the arc angles is not at all straightforward. The API documentation states: “The angles are specified relative to the nonsquare framing rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the framing rectangle. As a result, if the framing rectangle is noticeably longer along one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the frame.” Unfortunately, the documentation is silent on how to compute this “skew.” Here are the details:

Suppose the center of the arc is the origin and the point (x, y) lies on the arc.

http://www.rejoiceblog.com/

You can get a skewed angle with the following formula:

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * width));
```

The result is a value between `-180` and `180`. Compute the skewed start and end angles in this way. Then, compute the difference between the two skewed angles. If the start angle or the difference is negative, add `360` to the start angle. Then, supply the start angle and the difference to the arc constructor.

If you run the example program at the end of this section, you can visually check that this calculation yields the correct values for the arc constructor (see Figure 11.39).

The Java 2D API supports *quadratic* and *cubic* curves. In this chapter, we do not get into the mathematics of these curves. We suggest you get a feel for how the curves look by running the program in Listing 11.15. As you can see in Figures 11.37 and 11.38, quadratic and cubic curves are specified by two *end points* and one or two *control points*. Moving the control points changes the shape of the curves.

Figure 11.37 A quadratic curve

http://www.rejoiceblog.com/

Figure 11.38 A cubic curve

To construct quadratic and cubic curves, give the coordinates of the end points and the control points. For example,

```
var q = new QuadCurve2D.Double(startX, startY, controlX, cont

var c = new CubicCurve2D.Double(startX, startY, control1X, co

      control2X, control2Y, endX, endY);
```

Quadratic curves are not very flexible, and they are not commonly used in practice. Cubic curves (such as the Bézier curves drawn by the `CubicCurve2D` class) are, however, very common. By combining many cubic curves so that the slopes at the connection points match, you can create complex, smooth-looking curved shapes. For more information, we refer you to *Computer Graphics: Principles and Practice, Second Edition in C*, by James D. Foley, Andries van Dam, Steven K. Feiner, et al. (Addison-Wesley, 1995).

You can build arbitrary sequences of line segments, quadratic curves, and cubic curves, and store them in a `GeneralPath` object. Specify the first coordinate of the path with the `moveTo` method, for example:

```
var path = new GeneralPath();

path.moveTo(10, 20);
```

http://www.rejoiceblog.com/

You can then extend the path by calling one of the methods `lineTo`, `quadTo`, or `curveTo`. These methods extend the path by a line, a quadratic curve, or a cubic curve. To call `lineTo`, supply the end point. For the two curve methods, supply the control points, then the end point. For example,

```
path.lineTo(20, 30);

path.curveTo(control1X, control1Y, control2X, control2Y, endX
```

Close the path by calling the `closePath` method. It draws a line back to the starting point of the path.

To make a polygon, simply call `moveTo` to go to the first corner point, followed by repeated calls to `lineTo` to visit the other corner points. Finally, call `closePath` to close the polygon. The program in Listing 11.15 shows this in more detail.

A general path does not have to be connected. You can call `moveTo` at any time to start a new path segment.

Finally, you can use the `append` method to add arbitrary `Shape` objects to a general path. The outline of the shape is added to the end to the path. The second parameter of the `append` method is `true` if the new shape should be connected to the last point on the path, `false` otherwise. For example, the call

```
Rectangle2D r = . . .;

path.append(r, false);
```

appends the outline of a rectangle to the path without connecting it to the existing path. But

```
path.append(r, true);
```

adds a straight line from the end point of the path to the starting point of the rectangle, and then adds the rectangle outline to the path.

http://www.rejoiceblog.com/

The program in Listing 11.15 lets you create sample paths. Figures 11.37 and 11.38 show sample runs of the program. You can pick a shape maker from the combo box. The program contains shape makers for

• Straight lines

• Rectangles, rounded rectangles, and ellipses

• Arcs (showing lines for the bounding rectangle and the start and end angles, in addition to the arc itself)

• Polygons (using a `GeneralPath`)

• Quadratic and cubic curves

Use the mouse to adjust the control points. As you move them, the shape continuously repaints itself.

The program is a bit complex because it handles multiple shapes and supports dragging of the control points.

An abstract superclass `ShapeMaker` encapsulates the commonality of the shape maker classes. Each shape has a fixed number of control points that the user can move around. The `getPointCount` method returns that value. The abstract method

```
Shape makeShape(Point2D[] points)
```

computes the actual shape, given the current positions of the control points. The `toString` method returns the class name so that the `ShapeMaker` objects can simply be dumped into a `JComboBox`.

To enable dragging of the control points, the `ShapePanel` class handles both mouse and mouse motion events. If the mouse is pressed on top of a rectangle, subsequent mouse drags move the rectangle.

The majority of the shape maker classes are simple—their `makeShape` methods just construct and return the requested shapes. However, the `ArcMaker` class needs to compute the distorted start and end angles. Furthermore, to demonstrate that the computation is indeed correct, the returned shape is a `GeneralPath` containing the arc itself, the bounding rectangle, and the lines from the center of the arc to the angle

http://www.rejoiceblog.com/
control points (see Figure 11.39).

Figure 11.39 The `ShapeTest` program

Listing 11.15 `shape/ShapeTest.java`

```
package shape;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
```

http://www.rejoiceblog.com/

```
 * This program demonstrates the various 2D shapes.
 * @version 1.04 2018-05-01
 * @author Cay Horstmann
 */
public class ShapeTest
{
  public static void main(String[] args)
  {
      EventQueue.invokeLater(() ->
         {
            var frame = new ShapeTestFrame();
            frame.setTitle("ShapeTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_O
            frame.setVisible(true);
         });
   }
}

/**
 * This frame contains a combo box to select a shape and
```

http://www.rejoiceblog.com/

```
 */
class ShapeTestFrame extends JFrame
{
   public ShapeTestFrame()
   {
      var comp = new ShapeComponent();
      add(comp, BorderLayout.CENTER);
      var comboBox = new JComboBox<ShapeMaker>();
      comboBox.addItem(new LineMaker());
      comboBox.addItem(new RectangleMaker());
      comboBox.addItem(new RoundRectangleMaker());
      comboBox.addItem(new EllipseMaker());
      comboBox.addItem(new ArcMaker());
      comboBox.addItem(new PolygonMaker());
      comboBox.addItem(new QuadCurveMaker());
      comboBox.addItem(new CubicCurveMaker());
      comboBox.addActionListener(event ->
         {
            ShapeMaker shapeMaker = comboBox.getItemAt(c
```

http://www.rejoiceblog.com/

```
            comp.setShapeMaker(shapeMaker);
        });
        add(comboBox, BorderLayout.NORTH);
        comp.setShapeMaker((ShapeMaker) comboBox.getItemAt
        pack();
    }
}

/**
 * This component draws a shape and allows the user to m
 */
class ShapeComponent extends JComponent
{
    private static final Dimension PREFERRED_SIZE = new D
    private Point2D[] points;
    private static Random generator = new Random();
    private static int SIZE = 10;
    private int current;
    private ShapeMaker shapeMaker;

```

http://www.rejoiceblog.com/

```
    public ShapeComponent()
    {
        addMouseListener(new MouseAdapter()
            {
                public void mousePressed(MouseEvent event)
                {
                    Point p = event.getPoint();
                    for (int i = 0; i < points.length; i++)
                    {
                        double x = points[i].getX() - SIZE / 2
                        double y = points[i].getY() - SIZE / 2
                        var r = new Rectangle2D.Double(x, y, S
                        if (r.contains(p))
                        {
                          current = i;
                          return;
                        }
                    }
                }
```

http://www.rejoiceblog.com/

```
88
89                public void mouseReleased(MouseEvent event)
90                {
91                  current = -1;
92                }
93            });
94         addMouseMotionListener(new MouseMotionAdapter()
95            {
96                public void mouseDragged(MouseEvent event)
97                {
98                   if (current == -1) return;
99                   points[current] = event.getPoint();
100                  repaint();
101               }
102           });
103        current = -1;
104     }
105
106     /**
107      * Set a shape maker and initialize it with a random p
```

http://www.rejoiceblog.com/

```
     * @param aShapeMaker a shape maker that defines a sh
     */
    public void setShapeMaker(ShapeMaker aShapeMaker)
    {
        shapeMaker = aShapeMaker;
        int n = shapeMaker.getPointCount();
        points = new Point2D[n];
        for (int i = 0; i < n; i++)
        {
          double x = generator.nextDouble() * getWidth();
          double y = generator.nextDouble() * getHeight()
          points[i] = new Point2D.Double(x, y);
        }
        repaint();
    }

    public void paintComponent(Graphics g)
    {
       if (points == null) return;
```

http://www.rejoiceblog.com/

```
        var g2 = (Graphics2D) g;

        for (int i = 0; i < points.length; i++)

        {

           double x = points[i].getX() - SIZE / 2;

           double y = points[i].getY() - SIZE / 2;

           g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE

        }


        g2.draw(shapeMaker.makeShape(points));

      }


      public Dimension getPreferredSize() { return PREFERR

   }


   /**

    * A shape maker can make a shape from a point set. Con

    * makeShape method.

    */

   abstract class ShapeMaker

   {
```

http://www.rejoiceblog.com/

```
    private int pointCount;

    /**
     * Constructs a shape maker.
     * @param ointCount the number of points needed to d
     */
    public ShapeMaker(int pointCount)
    {
      this.pointCount = pointCount;
    }

    /**
     * Gets the number of points needed to define this s
     * @return the point count
     */
    public int getPointCount()
    {
      return pointCount;
    }
```

http://www.rejoiceblog.com/

```
    /**
     * Makes a shape out of the given point set.
     * @param p the points that define the shape
     * @return the shape defined by the points
     */
    public abstract Shape makeShape(Point2D[] p);

    public String toString()
    {
        return getClass().getName();
    }
}

/**
 * Makes a line that joins two given points.
 */
class LineMaker extends ShapeMaker
{
  public LineMaker()
```

http://www.rejoiceblog.com/

```
    {
      super(2);
    }

      public Shape makeShape(Point2D[] p)
      {
         return new Line2D.Double(p[0], p[1]);
      }
  }

  /**
   * Makes a rectangle that joins two given corner points
   */
  class RectangleMaker extends ShapeMaker
  {
     public RectangleMaker()
     {
        super(2);
     }
```

http://www.rejoiceblog.com/

```

    public Shape makeShape(Point2D[] p)
    {
        var s = new Rectangle2D.Double();
        s.setFrameFromDiagonal(p[0], p[1]);
        return s;
    }
}

/**
 * Makes a round rectangle that joins two given corner p
 */
class RoundRectangleMaker extends ShapeMaker
{
    public RoundRectangleMaker()
    {
        super(2);
    }

    public Shape makeShape(Point2D[] p)
```

http://www.rejoiceblog.com/

```
    {
        var s = new RoundRectangle2D.Double(0, 0, 0, 0, 2
        s.setFrameFromDiagonal(p[0], p[1]);
        return s;
    }
}

/**
 * Makes an ellipse contained in a bounding box with tw
 */
class EllipseMaker extends ShapeMaker
{
    public EllipseMaker()
    {
        super(2);
    }

    public Shape makeShape(Point2D[] p)
    {
```

http://www.rejoiceblog.com/

```
        var s = new Ellipse2D.Double();
        s.setFrameFromDiagonal(p[0], p[1]);
        return s;
    }
}

/**
 * Makes an arc contained in a bounding box with two gi
 * ending angles given by lines emanating from the cent
 * given points. To show the correctness of the angle c
 * arc, the bounding box, and the lines.
 */
class ArcMaker extends ShapeMaker
{
    public ArcMaker()
    {
        super(4);
    }

    public Shape makeShape(Point2D[] p)
```

http://www.rejoiceblog.com/

```
    {
        double centerX = (p[0].getX() + p[1].getX()) / 2;
        double centerY = (p[0].getY() + p[1].getY()) / 2;
        double width = Math.abs(p[1].getX() - p[0].getX()
        double height = Math.abs(p[1].getY() - p[0].getY()

        double skewedStartAngle = Math.toDegrees(Math.atan
            (p[2].getX() - centerX) * height));
        double skewedEndAngle = Math.toDegrees(Math.atan2(
            (p[3].getX() - centerX) * height));
        double skewedAngleDifference = skewedEndAngle - sk
        if (skewedStartAngle < 0) skewedStartAngle += 360;
        if (skewedAngleDifference < 0) skewedAngleDifferen

        var s = new Arc2D.Double(0, 0, 0, 0, skewedStartAn
        s.setFrameFromDiagonal(p[0], p[1]);

        var g = new GeneralPath();
        g.append(s, false);
```

http://www.rejoiceblog.com/

```
        var r = new Rectangle2D.Double();
        r.setFrameFromDiagonal(p[0], p[1]);
        g.append(r, false);
        var center = new Point2D.Double(centerX, centerY);
        g.append(new Line2D.Double(center, p[2]), false);
        g.append(new Line2D.Double(center, p[3]), false);
        return g;
    }
}

/**
 * Makes a polygon defined by six corner points.
 */
class PolygonMaker extends ShapeMaker
{
    public PolygonMaker()
    {
        super(6);
    }

```

http://www.rejoiceblog.com/

```
    public Shape makeShape(Point2D[] p)
    {
        var s = new GeneralPath();
        s.moveTo((float) p[0].getX(), (float) p[0].getY()
        for (int i = 1; i < p.length; i++)
            s.lineTo((float) p[i].getX(), (float) p[i].get
        s.closePath();
        return s;
    }
}

/**
 * Makes a quad curve defined by two end points and a c
 */
class QuadCurveMaker extends ShapeMaker
{
    public QuadCurveMaker()
    {
        super(3);
```

http://www.rejoiceblog.com/

```
    }

    public Shape makeShape(Point2D[] p)
    {
        return new QuadCurve2D.Double(p[0].getX(), p[0].g
          p[2].getX(), p[2].getY());
    }
}

/**
 * Makes a cubic curve defined by two end points and tw
 */
class CubicCurveMaker extends ShapeMaker
{
  public CubicCurveMaker()
  {
      super(4);
  }

  public Shape makeShape(Point2D[] p)
```

http://www.rejoiceblog.com/

```
    {
        return new CubicCurve2D.Double(p[0].getX(), p[0].g
                .getX(), p[2].getY(), p[3].getX(), p[3].getY
    }
}
```

java.awt.geom.RoundRectangle2D.Double 1.2

- `RoundRectangle2D.Double(double x, double y, double width, double height, double arcWidth, double arcHeight)`

constructs a rounded rectangle with the given bounding rectangle and arc dimensions. See Figure 11.34 for an explanation of the `arcWidth` and `arcHeight` parameters.

java.awt.geom.Arc2D.Double 1.2

- `Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type)`

constructs an arc with the given bounding rectangle, start and arc angle, and arc type. The `startAngle` and `arcAngle` are explained on p. 704. The type is one of `Arc2D.OPEN`, `Arc2D.PIE`, and `Arc2D.CHORD`.

java.awt.geom.QuadCurve2D.Double 1.2

- `QuadCurve2D.Double(double x1, double y1, double ctrlx, double ctrly, double x2, double y2)`

constructs a quadratic curve from a start point, a control point, and an end point.

http://www.rejoiceblog.com/

java.awt.geom.CubicCurve2D.Double 1.2

- `CubicCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)`

constructs a cubic curve from a start point, two control points, and an end point.

java.awt.geom.GeneralPath 1.2

- `GeneralPath()`

constructs an empty general path.

java.awt.geom.Path2D.Float 6

- `void moveTo(float x, float y)`

makes (`x`, `y`) the *current point*—that is, the starting point of the next segment.

- `void lineTo(float x, float y)`

- `void quadTo(float ctrlx, float ctrly, float x, float y)`

- `void curveTo(float ctrl1x, float ctrl1y, float ctrl2x, float ctrl2y, float x, float y)`

draws a line, quadratic curve, or cubic curve from the current point to the end point (`x`, `y`), and makes that end point the current point.

java.awt.geom.Path2D 6

- `void append(Shape s, boolean connect)`

adds the outline of the given shape to the general path. If `connect` is `true`, the

http://www.rejoiceblog.com/

current point of the general path is connected to the starting point of the added shape by a straight line.

• `void closePath()`

closes the path by drawing a straight line from the current point to the first point in the path.

11.3.3 Areas

In the preceding section, you saw how you can specify complex shapes by constructing general paths composed of lines and curves. By using a sufficient number of lines and curves, you can draw essentially any shape. For example, the shapes of characters in the fonts that you see on the screen and on your printouts are all made up of lines and cubic curves.

Occasionally, it is easier to describe a shape by composing it from *areas,* such as rectangles, polygons, or ellipses. The Java 2D API supports four *constructive area geometry* operations that combine two areas into a new area.

• `add`: The combined area contains all points that are in the first or the second area.

• `subtract`: The combined area contains all points that are in the first but not the second area.

• `intersect`: The combined area contains all points that are in the first and the second area.

• `exclusiveOr`: The combined area contains all points that are in either the first or the second area, but not in both.

Figure 11.40 shows these operations.

http://www.rejoiceblog.com/

Figure 11.40 Constructive area geometry operations

To construct a complex area, start with a default area object.

```
var a = new Area();
```

Then, combine the area with any shape.

```
a.add(new Rectangle2D.Double(. . .));

a.subtract(path);

. . .
```

The `Area` class implements the `Shape` interface. You can stroke the boundary of the area with the `draw` method or paint the interior with the `fill` method of the `Graphics2D` class.

http://www.rejoiceblog.com/

java.awt.geom.Area

- `void add(Area other)`
- `void subtract(Area other)`
- `void intersect(Area other)`
- `void exclusiveOr(Area other)`

carries out the constructive area geometry operation with this area and the other area and sets this area to the result.

11.3.4 Strokes

The `draw` operation of the `Graphics2D` class draws the boundary of a shape by using the currently selected *stroke*. By default, the stroke is a solid line that is 1 pixel wide. You can select a different stroke by calling the `setStroke` method and supplying an object of a class that implements the `Stroke` interface. The Java 2D API defines only one such class, called `BasicStroke`. In this section, we'll look at the capabilities of the `BasicStroke` class.

You can construct strokes of arbitrary thickness. For example, here is how to draw lines that are 10 pixels wide:

```
g2.setStroke(new BasicStroke(10.0F));

g2.draw(new Line2D.Double(. . .));
```

When a stroke is more than a pixel thick, the *end* of the stroke can have different styles. Figure 11.41 shows these so-called end cap styles. You have three choices:

- A *butt cap* simply ends the stroke at its end point.
- A *round cap* adds a half-circle to the end of the stroke.
- A *square cap* adds a half-square to the end of the stroke.

http://www.rejoiceblog.com/

Figure 11.41 End cap styles

When two thick strokes meet, there are three choices for the *join style* (see Figure 11.42).

• A *bevel join* joins the strokes with a straight line that is perpendicular to the bisector of the angle between the two strokes.

• A *round join* extends each stroke to have a round cap.

• A *miter join* extends both strokes by adding a "spike."

http://www.rejoiceblog.com/

Figure 11.42 Join styles

The miter join is not suitable for lines that meet at small angles. If two lines join with an angle that is less than the *miter limit,* a bevel join is used instead, which prevents extremely long spikes. By default, the miter limit is 10 degrees.

You can specify these choices in the `BasicStroke` constructor, for example:

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND, Ba

g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, Bas

     15.0F /* miter limit */));
```

Finally, you can create dashed lines by setting a *dash pattern*. In the program in Listing 11.16, you can select a dash pattern that spells out SOS in Morse code. The dash pattern is a `float[]` array that contains the lengths of the “on” and “off” intervals (see Figure 11.43).

http://www.rejoiceblog.com/

Figure 11.43 A dash pattern

You can specify the dash pattern and a *dash phase* when constructing the `BasicStroke`. The dash phase indicates where in the dash pattern each line should start. Normally, you set this value to `0`.

```
float[] dashPattern = { 10, 10, 10, 10, 10, 10, 30, 10, 30, .

g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, Bas

   10.0F /* miter limit */, dashPattern, 0 /* dash phase */))
```

Note

End cap styles are applied to the ends of *each dash* in a dash pattern.

The program in Listing 11.16 lets you specify end cap styles, join styles, and dashed lines (see Figure 11.44). You can move the ends of the line segments to test the miter limit: Select the miter join, then move the line segment to form a very acute angle. You will see the miter join turn into a bevel join.

http://www.rejoiceblog.com/

Figure 11.44 The `StrokeTest` program

The program is similar to the program in Listing 11.15. The mouse listener remembers your click on the end point of a line segment, and the mouse motion listener monitors the dragging of the end point. A set of radio buttons signal the user choices for the end cap style, join style, and solid or dashed line. The `paintComponent` method of the `StrokePanel` class constructs a `GeneralPath` consisting of the two line segments that join the three points which the user can move with the mouse. It then constructs a `BasicStroke`, according to the selections the user made, and finally draws the path.

Listing 11.16 `stroke/StrokeTest.java`

```
package stroke;

import java.awt.*;
import java.awt.event.*;
```

http://www.rejoiceblog.com/

```
import java.awt.geom.*;
import javax.swing.*;

/**
 * This program demonstrates different stroke types.
 * @version 1.05 2018-05-01
 * @author Cay Horstmann
 */
public class StrokeTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() ->
         {
            var frame = new StrokeTestFrame();
            frame.setTitle("StrokeTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_
            frame.setVisible(true);
         });
```

http://www.rejoiceblog.com/

```
    }
}

/**
 * This frame lets the user choose the cap, join, and li
 */
class StrokeTestFrame extends JFrame
{
  private StrokeComponent canvas;
  private JPanel buttonPanel;

  public StrokeTestFrame()
  {
     canvas = new StrokeComponent();
     add(canvas, BorderLayout.CENTER);

     buttonPanel = new JPanel();
     buttonPanel.setLayout(new GridLayout(3, 3));
     add(buttonPanel, BorderLayout.NORTH);

```

http://www.rejoiceblog.com/

```
        var group1 = new ButtonGroup();
        makeCapButton("Butt Cap", BasicStroke.CAP_BUTT, gro
        makeCapButton("Round Cap", BasicStroke.CAP_ROUND, g
        makeCapButton("Square Cap", BasicStroke.CAP_SQUARE,

        var group2 = new ButtonGroup();
        makeJoinButton("Miter Join", BasicStroke.JOIN_MITER
        makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL
        makeJoinButton("Round Join", BasicStroke.JOIN_ROUND

        var group3 = new ButtonGroup();
        makeDashButton("Solid Line", false, group3);
        makeDashButton("Dashed Line", true, group3);
    }

    /**
     * Makes a radio button to change the cap style.
     * @param label the button label
     * @param style the cap style
```

http://www.rejoiceblog.com/

```
     * @param group the radio button group
     */
    private void makeCapButton(String label, final int sty
    {
        // select first button in group
        boolean selected = group.getButtonCount() == 0;
        var button = new JRadioButton(label, selected);
        buttonPanel.add(button);
        group.add(button);
        button.addActionListener(event -> canvas.setCap(sty
        pack();
    }

    /**
     * Makes a radio button to change the join style.
     * @param label the button label
     * @param style the join style
     * @param group the radio button group
     */
    private void makeJoinButton(String label, final int st
```

http://www.rejoiceblog.com/

```
    {
        // select first button in group
        boolean selected = group.getButtonCount() == 0;
        var button = new JRadioButton(label, selected);
        buttonPanel.add(button);
        group.add(button);
        button.addActionListener(event -> canvas.setJoin(s
    }

    /**
     * Makes a radio button to set solid or dashed lines
     * @param label the button label
     * @param style false for solid, true for dashed lines
     * @param group the radio button group
     */
    private void makeDashButton(String label, final boolea
    {
        // select first button in group
        boolean selected = group.getButtonCount() == 0;
```

http://www.rejoiceblog.com/

```
      var button = new JRadioButton(label, selected);
      buttonPanel.add(button);
      group.add(button);
      button.addActionListener(event -> canvas.setDash(st
   }
}

/**
 * This component draws two joined lines, using differen
 * drag the three points defining the lines.
 */
class StrokeComponent extends JComponent
{
   private static final Dimension PREFERRED_SIZE = new D
   private static int SIZE = 10;

   private   Point2D[] points;
   private   int current;
   private   float width;
   private   int cap;
```

http://www.rejoiceblog.com/

```
    private   int join;
    private   boolean dash;

    public StrokeComponent()
    {
       addMouseListener(new MouseAdapter()
          {
             public void mousePressed(MouseEvent event)
             {
                Point p = event.getPoint();
                for (int i = 0; i < points.length; i++)
                {
                   double x = points[i].getX() - SIZE / 2
                   double y = points[i].getY() - SIZE / 2
                   var r = new Rectangle2D.Double(x, y, S
                   if (r.contains(p))
                   {
                      current = i;
                      return;
```

http://www.rejoiceblog.com/

```
                    }
                }
            }

            public void mouseReleased(MouseEvent event)
            {
                current = -1;
            }
        });

        addMouseMotionListener(new MouseMotionAdapter()
            {
              public void mouseDragged(MouseEvent event)
              {
                  if (current == -1) return;
                  points[current] = event.getPoint();
                  repaint();
              }
            });

```

http://www.rejoiceblog.com/

```
        points = new Point2D[3];
        points[0] = new Point2D.Double(200, 100);
        points[1] = new Point2D.Double(100, 200);
        points[2] = new Point2D.Double(200, 200);
        current = -1;
        width = 8.0F;
    }

    public void paintComponent(Graphics g)
    {
        var g2 = (Graphics2D) g;
        var path = new GeneralPath();
        path.moveTo((float) points[0].getX(), (float) poin
        for (int i = 1; i < points.length; i++)
            path.lineTo((float) points[i].getX(), (float) p
        BasicStroke stroke;
        if (dash)
        {
            float miterLimit = 10.0F;
```

http://www.rejoiceblog.com/

```
            float[] dashPattern = { 10F, 10F, 10F, 10F, 10F
                 10F, 10F, 10F, 10F, 30F };
            float dashPhase = 0;
            stroke = new BasicStroke(width, cap, join, mite
        }
        else stroke = new BasicStroke(width, cap, join);
        g2.setStroke(stroke);
        g2.draw(path);
    }

    /**
     * Sets the join style.
     * @param j the join style
     */
    public void setJoin(int j)
    {
      join = j;
      repaint();
    }

```

http://www.rejoiceblog.com/

```
    /**
     * Sets the cap style.
     * @param c the cap style
     */
    public void setCap(int c)
    {
      cap = c;
      repaint();
    }

    /**
     * Sets solid or dashed lines.
     * @param d false for solid, true for dashed lines
     */
    public void setDash(boolean d)
    {
       dash = d;
       repaint();
    }
```

http://www.rejoiceblog.com/

```
    public Dimension getPreferredSize() { return PREFERRE
  }
```

java.awt.Graphics2D 1.2

- `void setStroke(Stroke s)`

sets the stroke of this graphics context to the given object that implements the `Stroke` interface.

java.awt.BasicStroke 1.2

- `BasicStroke(float width)`
- `BasicStroke(float width, int cap, int join)`
- `BasicStroke(float width, int cap, int join, float miterlimit)`
- `BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashPhase)`

constructs a stroke object with the given attributes.

http://www.rejoiceblog.com/

Parameters:	width	The width of the pen
	cap	The end cap style—one of CAP_BUTT, CAP_ROUND, and CAP_SQUARE
	join	The join style—one of JOIN_BEVEL, JOIN_MITER, and JOIN_ROUND
	miterlimit	The angle, in degrees, below which a miter join is rendered as a bevel join
	dash	An array of the lengths of the alternating filled and blank portions of a dashed stroke
	dashPhase	The "phase" of the dash pattern; a segment of this length, preceding the starting point of the stroke, is assumed to have the dash pattern already applied.

11.3.5 Paint

When you fill a shape, its inside is covered with *paint*. Use the `setPaint` method to set the paint style to an object whose class implements the `Paint` interface. The Java 2D API provides three such classes:

• The `Color` class implements the `Paint` interface. To fill shapes with a solid color, simply call `setPaint` with a `Color` object, such as

```
g2.setPaint(Color.red);
```

• The `GradientPaint` class varies colors by interpolating between two given color values (see Figure 11.45).

Figure 11.45 Gradient paint

• The `TexturePaint` class fills an area with repetitions of an image (see Figure 11.46).

http://www.rejoiceblog.com/

Figure 11.46 Texture paint

You can construct a `GradientPaint` object by specifying two points and the colors that you want at these two points.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW
```

Colors are interpolated along the line joining the two points. Colors are constant along lines perpendicular to that joining line. Points beyond an end point of the line are given the color at the end point.

Alternatively, if you call the `GradientPaint` constructor with `true` for the `cyclic` parameter,

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW
```

then the color variation *cycles* and keeps varying beyond the end points.

To construct a `TexturePaint` object, specify a `BufferedImage` and an *anchor* rectangle.

```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle))
```

We will introduce the `BufferedImage` class later in this chapter when we discuss images in detail. The simplest way of obtaining a buffered image is to read an image file:

http://www.rejoiceblog.com/

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

The anchor rectangle is extended indefinitely in *x* and *y* directions to tile the entire coordinate plane. The image is scaled to fit into the anchor and then replicated into each tile.

java.awt.Graphics2D 1.2

- `void setPaint(Paint s)`

sets the paint of this graphics context to the given object that implements the `Paint` interface.

java.awt.GradientPaint 1.2

- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2)`
- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cyclic)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2, boolean cyclic)`

constructs a gradient paint object that fills shapes with color such that the start point is colored with `color1`, the end point is colored with `color2`, and the colors in between are linearly interpolated. Colors are constant along lines perpendicular to the line joining the start and the end point. By default, the gradient paint is not cyclic—that is, points beyond the start and end points are colored with the same color as the start and end point. If the gradient paint is *cyclic,* then colors continue to be interpolated, first returning to the starting point color and then repeating indefinitely in both directions.

http://www.rejoiceblog.com/

java.awt.TexturePaint 1.2

- `TexturePaint(BufferedImage texture, Rectangle2D anchor)`

creates a texture paint object. The anchor rectangle defines the tiling of the space to be painted; it is repeated indefinitely in *x* and *y* directions, and the texture image is scaled to fill each tile.

11.3.6 Coordinate Transformations

Suppose you need to draw an object, such as an automobile. You know, from the manufacturer's specifications, the height, wheelbase, and total length. You could, of course, figure out all pixel positions, assuming some number of pixels per meter. However, there is an easier way: You can ask the graphics context to carry out the conversion for you.

```
g2.scale(pixelsPerMeter, pixelsPerMeter);

g2.draw(new Line2D.Double(coordinates in meters)); // converts to pi
```

The `scale` method of the `Graphics2D` class sets the *coordinate transformation* of the graphics context to a scaling transformation. That transformation changes *user coordinates* (user-specified units) to *device coordinates* (pixels). Figure 11.47 shows how the transformation works.

http://www.rejoiceblog.com/

Figure 11.47 User and device coordinates

Coordinate transformations are very useful in practice. They allow you to work with convenient coordinate values. The graphics context takes care of the dirty work of transforming them to pixels.

There are four fundamental transformations.

• Scaling: blowing up, or shrinking, all distances from a fixed point

• Rotation: rotating all points around a fixed center

• Translation: moving all points by a fixed amount

• Shear: leaving one line fixed and “sliding” the lines parallel to it by an amount that is proportional to the distance from the fixed line

Figure 11.48 shows how these four fundamental transformations act on a unit square.

http://www.rejoiceblog.com/

Figure 11.48 The fundamental transformations

The `scale`, `rotate`, `translate`, and `shear` methods of the `Graphics2D` class set the coordinate transformation of the graphics context to one of these fundamental transformations.

You can *compose* the transformations. For example, you might want to rotate shapes *and* double their size; supply both a rotation and a scaling transformation:

```
g2.rotate(angle);

g2.scale(2, 2);

g2.draw(. . .);
```

In this case, it does not matter in which order you supply the transformations. However, with most transformations, order does matter. For example, if you want to rotate and shear, then it makes a difference which of the transformations you supply first. You need to figure out what your intention is. The graphics context will apply the transformations in the order opposite to that in which you supplied them—that is, the last transformation you supply is applied first.

You can supply as many transformations as you like. For example, consider the following sequence of transformations:

```
g2.translate(x, y);

g2.rotate(a);

g2.translate(-x, -y);
```

The last transformation (which is applied first) moves the point (`x`, `y`) to the origin. The second transformation rotates with an angle `a` around the origin. The final transformation moves the origin back to (`x`, `y`). The overall effect is a rotation with center point (`x`, `y`)—see Figure 11.49. Since rotating about a point other than the origin is such a common operation, there is a shortcut:

http://www.rejoiceblog.com/

```
g2.rotate(a, x, y);
```

Figure 11.49 Composing transformations

If you know some matrix theory, you are probably aware that all rotations, translations, scalings, shears, and their compositions can be expressed by transformation matrices of the form:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Such a transformation is called an *affine transformation*. In the Java 2D API, the `AffineTransform` class describes such a transformation. If you know the

http://www.rejoiceblog.com/
components of a particular transformation matrix, you can construct it directly as

```
var t = new AffineTransform(a, b, c, d, e, f);
```

Additionally, the factory methods `getRotateInstance`, `getScaleInstance`, `getTranslateInstance`, and `getShearInstance` construct the matrices that represent these transformation types. For example, the call

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F);
```

returns a transformation that corresponds to the matrix

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, the instance methods `setToRotation`, `setToScale`, `setToTranslation`, and `setToShear` set a transformation object to a new type. Here is an example:

```
t.setToRotation(angle); // sets t to a rotation
```

You can set the coordinate transformation of the graphics context to an `AffineTransform` object.

```
g2.setTransform(t); // replaces current transformation
```

However, in practice, you shouldn't call the `setTransform` operation, as it replaces any existing transformation that the graphics context may have. For example, a graphics context for printing in landscape mode already contains a 90-degree rotation transformation. If you call `setTransform`, you obliterate that rotation. Instead, call the `transform` method.

```
g2.transform(t); // composes current transformation with t
```

http://www.rejoiceblog.com/

It composes the existing transformation with the new `AffineTransform` object.

If you just want to apply a transformation temporarily, first get the old transformation, compose it with your new transformation, and finally restore the old transformation when you are done.

```
AffineTransform oldTransform = g2.getTransform(); // save old

g2.transform(t); // apply temporary transform

draw on g2

g2.setTransform(oldTransform); // restore old transform
```

java.awt.geom.AffineTransform 1.2

- `AffineTransform(double a, double b, double c, double d, double e, double f)`
- `AffineTransform(float a, float b, float c, float d, float e, float f)`

constructs the affine transform with matrix

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- AffineTransform(double[] m)
- AffineTransform(float[] m)

constructs the affine transform with matrix

$$\begin{bmatrix} m[0] & m[2] & m[4] \\ m[1] & m[3] & m[5] \\ 0 & 0 & 1 \end{bmatrix}$$

http://www.rejoiceblog.com/

• static AffineTransform getRotateInstance(double a)

creates a rotation around the origin by the angle a (in radians). The transformation matrix is

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If a is between 0 and π / 2, the rotation moves the positive x axis toward the positive y axis.

• static AffineTransform getRotateInstance(double a, double x, double y)

creates a rotation around the point (x,y) by the angle a (in radians).

• static AffineTransform getScaleInstance(double sx, double sy)

creates a scaling transformation that scales the x axis by sx and the y axis by sy. The transformation matrix is

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

• static AffineTransform getShearInstance(double shx, double shy)

creates a shear transformation that shears the x axis by shx and the y axis by shy. The transformation matrix is

$$\begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

• static AffineTransform getTranslateInstance(double tx, double ty)

creates a translation that moves the x axis by tx and the y axis by ty. The transformation matrix is

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

http://www.rejoiceblog.com/

- `void setToRotation(double a)`
- `void setToRotation(double a, double x, double y)`
- `void setToScale(double sx, double sy)`
- `void setToShear(double sx, double sy)`
- `void setToTranslation(double tx, double ty)`

sets this affine transformation to a basic transformation with the given parameters. See the `get`*Xxx*`Instance` methods for an explanation of the basic transformations and their parameters.

java.awt.Graphics2D 1.2

- `void setTransform(AffineTransform t)`

replaces the existing coordinate transformation of this graphics context with `t`.

- `void transform(AffineTransform t)`

composes the existing coordinate transformation of this graphics context with `t`.

- `void rotate(double a)`
- `void rotate(double a, double x, double y)`
- `void scale(double sx, double sy)`
- `void shear(double sx, double sy)`
- `void translate(double tx, double ty)`

composes the existing coordinate transformation of this graphics context with a basic transformation with the given parameters. See the `AffineTransform.get`*Xxx*`Instance` methods for an explanation of the basic transformations and their parameters.

http://www.rejoiceblog.com/

11.3.7 Clipping

By setting a *clipping shape* in the graphics context, you constrain all drawing operations to the interior of that clipping shape.

```
g2.setClip(clipShape); // but see below

g2.draw(shape); // draws only the part that falls inside the
```

However, in practice, you don't want to call the `setClip` operation because it replaces any existing clipping shape that the graphics context might have. For example, as you will see later in this chapter, a graphics context for printing comes with a clip rectangle that ensures that you don't draw on the margins. Instead, call the `clip` method.

```
g2.clip(clipShape); // better
```

The `clip` method intersects the existing clipping shape with the new one that you supply.

If you just want to apply a clipping area temporarily, you should first get the old clip, add your new clip, and finally restore the old clip when you are done:

```
Shape oldClip = g2.getClip(); // save old clip

g2.clip(clipShape); // apply temporary clip

draw on g2

g2.setClip(oldClip); // restore old clip
```

In Figure 11.50, we show off the clipping capability with a rather dramatic drawing of a line pattern clipped by a complex shape—namely, the outline of a set of letters.

http://www.rejoiceblog.com/

Figure 11.50 Using letter shapes to clip a line pattern

To obtain the character outlines, you need a *font render context*. Use the `getFontRenderContext` method of the `Graphics2D` class.

```
FontRenderContext context = g2.getFontRenderContext();
```

Next, using a string, a font, and the font render context, create a `TextLayout` object:

```
var layout = new TextLayout("Hello", font, context);
```

This text layout object describes the layout of a sequence of characters, as rendered by a particular font render context. The layout depends on the font render context—the same characters will look different on a screen and on a printer.

More important for our application, the `getOutline` method returns a `Shape` object that describes the shape of the outline of the characters in the text layout. The outline shape starts at the origin (0, 0), which might not be what you want. In that case, supply an affine transform to the `getOutline` operation to specify where you would like the outline to appear.

```
AffineTransform transform = AffineTransform.getTranslateInsta

Shape outline = layout.getOutline(transform);
```

http://www.rejoiceblog.com/

Then, append the outline to the clipping shape.

```
var clipShape = new GeneralPath();

clipShape.append(outline, false);
```

Finally, set the clipping shape and draw a set of lines. The lines appear only inside the character boundaries.

```
g2.setClip(clipShape);

var p = new Point2D.Double(0, 0);

for (int i = 0; i < NLINES; i++)

{

   double x = . . .;

   double y = . . .;

   var q = new Point2D.Double(x, y);

   g2.draw(new Line2D.Double(p, q)); // lines are clipped

}
```

java.awt.Graphics 1.0

- `void setClip(Shape s)` **1.2**

sets the current clipping shape to the shape `s`.

- `Shape getClip()` **1.2**

returns the current clipping shape.

http://www.rejoiceblog.com/

java.awt.Graphics2D 1.2

- `void clip(Shape s)`

intersects the current clipping shape with the shape `s`.

- `FontRenderContext getFontRenderContext()`

returns a font render context that is necessary for constructing `TextLayout` objects.

java.awt.font.TextLayout 1.2

- `TextLayout(String s, Font f, FontRenderContext context)`

constructs a text layout object from a given string and font, using the font render context to obtain font properties for a particular device.

- `float getAdvance()`

returns the width of this text layout.

- `float getAscent()`

- `float getDescent()`

returns the height of this text layout above and below the baseline.

- `float getLeading()`

returns the distance between successive lines in the font used by this text layout.

11.3.8 Transparency and Composition

In the standard RGB color model, every color is described by its red, green, and blue components. However, it is also convenient to describe areas of an image that are *transparent* or partially transparent. When you superimpose an image onto an existing drawing, the transparent pixels do not obscure the pixels under them at all, whereas partially transparent pixels are mixed with the pixels under them. Figure 11.51 shows

http://www.rejoiceblog.com/
the effect of overlaying a partially transparent rectangle on an image. You can still see the details of the image shine through from under the rectangle.

Figure 11.51 Overlaying a partially transparent rectangle on an image

In the Java 2D API, transparency is described by an *alpha channel*. Each pixel has, in addition to its red, green, and blue color components, an alpha value between 0 (fully transparent) and 1 (fully opaque). For example, the rectangle in Figure 11.51 was filled with a pale yellow color with 50% transparency:

```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```

Now let us look at what happens if you superimpose two shapes. You need to blend or *compose* the colors and alpha values of the source and destination pixels. Porter and Duff, two researchers in the field of computer graphics, have formulated 12 possible *composition rules* for this blending process. The Java 2D API implements all of these rules. Before going any further, we'd like to point out that only two of these rules have practical significance. If you find the rules arcane or confusing, just use the `SRC_OVER` rule. It is the default rule for a `Graphics2D` object, and it gives the most intuitive results.

Here is the theory behind the rules. Suppose you have a *source pixel* with alpha value a_S. In the image, there is already a *destination pixel* with alpha value a_D. You want to compose the two. The diagram in Figure 11.52 shows how to design a composition rule.

http://www.rejoiceblog.com/

Figure 11.52 Designing a composition rule

Porter and Duff consider the alpha value as the probability that the pixel color should be used. From the perspective of the source, there is a probability a_S that it wants to use the source color and a probability of $1 - a_S$ that it doesn't care. The same holds for the destination. When composing the colors, let us assume that the probabilities are independent. Then there are four cases, as shown in Figure 11.52. If the source wants to use the source color and the destination doesn't care, then it seems reasonable to let the source have its way. That's why the upper right corner of the diagram is labeled "S". The probability for that event is $a_S \cdot (1 - a_D)$. Similarly, the lower left corner is labeled "D". What should one do if both destination and source would like to select their color? That's where the Porter–Duff rules come in. If we decide that the source is more important, we label the lower right corner with an "S" as well. That rule is called `SRC_OVER`. In that rule, you combine the source colors with a weight of a_S and the destination colors with a weight of $(1 - a_S) \cdot a_D$.

The visual effect is a blending of the source and destination, with preference given to the source. In particular, if a_S is 1, then the destination color is not taken into account at all. If a_S is 0, then the source pixel is completely transparent and the destination color is unchanged.

The other rules depend on what letters you put in the boxes of the probability diagram. Table 11.3 and Figure 11.53 show all rules that are supported by the Java 2D API. The images in the figure show the results of the rules when a rectangular source region with an alpha of 0.75 is combined with an elliptical destination region with an alpha of 1.0.

http://www.rejoiceblog.com/

Table 11.3 The Porter–Duff Composition Rules

Rule	Explanation
CLEAR	Source clears destination.
SRC	Source overwrites destination and empty pixels.
DST	Source does not affect destination.
SRC_OVER	Source blends with destination and overwrites empty pixels.
DST_OVER	Source does not affect destination and overwrites empty pixels.
SRC_IN	Source overwrites destination.
SRC_OUT	Source clears destination and overwrites empty pixels.
DST_IN	Source alpha modifies destination.
DST_OUT	Source alpha complement modifies destination.
SRC_ATOP	Source blends with destination.
DST_ATOP	Source alpha modifies destination. Source overwrites empty pixels.
XOR	Source alpha complement modifies destination. Source overwrites empty pixels.

Figure 11.53 Porter–Duff composition rules

As you can see, most of the rules aren't very useful. Consider, as an extreme case, the

http://www.rejoiceblog.com/

DST_IN rule. It doesn't take the source color into account at all, but it uses the alpha of the source to affect the destination. The SRC rule is potentially useful—it forces the source color to be used, turning off blending with the destination.

For more information on the Porter–Duff rules, see, for example, *Computer Graphics: Principles and Practice, Second Edition in C*, by James D. Foley, Andries van Dam, Steven K. Feiner, et al.

Use the setComposite method of the Graphics2D class to install an object of a class that implements the Composite interface. The Java 2D API supplies one such class, AlphaComposite, that implements all the Porter–Duff rules in Figure 11.53.

The factory method getInstance of the AlphaComposite class yields an AlphaComposite object. You supply the rule and the alpha value to be used for source pixels. For example, consider the following code:

```
int rule = AlphaComposite.SRC_OVER;

float alpha = 0.5f;

g2.setComposite(AlphaComposite.getInstance(rule, alpha));

g2.setPaint(Color.blue);

g2.fill(rectangle);
```

The rectangle is then painted with blue color and an alpha value of 0.5. Since the composition rule is SRC_OVER, it is transparently overlaid on the existing image.

The program in Listing 11.17 lets you explore these composition rules. Pick a rule from the combo box and use the slider to set the alpha value of the AlphaComposite object.

Furthermore, the program displays a verbal description of each rule. Note that the descriptions are computed from the composition rule diagrams. For example, a "DS" in the second row stands for "blends with destination."

The program has one important twist. There is no guarantee that the graphics context that corresponds to the screen has an alpha channel. (In fact, it generally does not.) When pixels are deposited to a destination without an alpha channel, the pixel colors

http://www.rejoiceblog.com/
are multiplied with the alpha value and the alpha value is discarded. Now, several of the Porter–Duff rules use the alpha values of the destination, which means a destination alpha channel is important. For that reason, we use a buffered image with the ARGB color model to compose the shapes. After the images have been composed, we draw the resulting image to the screen.

```
var image = new BufferedImage(getWidth(), getHeight(), Buffer

Graphics2D gImage = image.createGraphics();

// now draw to gImage

g2.drawImage(image, null, 0, 0);
```

Listings 11.17 and 11.18 show the frame and component class. The `Rule` class in Listing 11.19 provides a brief explanation for each rule—see Figure 11.54. As you run the program, move the alpha slider from left to right to see the effect on the composed shapes. In particular, note that the only difference between the `DST_IN` and `DST_OUT` rules is how the destination (!) color changes when you change the source alpha.

http://www.rejoiceblog.com/

Figure 11.54 The `CompositeTest` program

Listing 11.17 `composite/CompositeTestFrame.java`

```
package composite;

import java.awt.*;

import javax.swing.*;

/**
 * This frame contains a combo box to choose a compositio
```

http://www.rejoiceblog.com/

```
 * channel, and a component that shows the composition.
 */
class CompositeTestFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;

   private CompositeComponent canvas;
   private JComboBox<Rule> ruleCombo;
   private JSlider alphaSlider;
   private JTextField explanation;

   public CompositeTestFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      canvas = new CompositeComponent();
      add(canvas, BorderLayout.CENTER);

      ruleCombo = new JComboBox<>(new Rule[] { new Rule("C
```

http://www.rejoiceblog.com/

```
        new Rule("SRC", " S", " S"), new Rule("DST", " ",
        new Rule("SRC_OVER", " S", "DS"), new Rule("DST_OV
        new Rule("SRC_IN", " ", " S"), new Rule("SRC_OUT",
        new Rule("DST_IN", " ", " D"), new Rule("DST_OUT",
        new Rule("SRC_ATOP", " ", "DS"), new Rule("DST_ATO
        new Rule("XOR", " S", "D "), });
      ruleCombo.addActionListener(event ->
        {
          var r = (Rule) ruleCombo.getSelectedItem();
          canvas.setRule(r.getValue());
          explanation.setText(r.getExplanation());
        });

      alphaSlider = new JSlider(0, 100, 75);
      alphaSlider.addChangeListener(event -> canvas.setAlp
      var panel = new JPanel();
      panel.add(ruleCombo);
      panel.add(new JLabel("Alpha"));
      panel.add(alphaSlider);
```

http://www.rejoiceblog.com/

```
        add(panel, BorderLayout.NORTH);

        explanation = new JTextField();
        add(explanation, BorderLayout.SOUTH);

        canvas.setAlpha(alphaSlider.getValue());
        Rule r = ruleCombo.getItemAt(ruleCombo.getSelectedIn
        canvas.setRule(r.getValue());
        explanation.setText(r.getExplanation());
    }
}
```

Listing 11.18 `composite/CompositeComponent.java`

```
package composite;

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import javax.swing.*;

```

http://www.rejoiceblog.com/

```
 8  /**
 9   * This component draws two shapes, composed with a compo
10   */
11  class CompositeComponent extends JComponent
12  {
13     private int rule;
14     private Shape shape1;
15     private Shape shape2;
16     private float alpha;
17
18     public CompositeComponent()
19     {
20       shape1 = new Ellipse2D.Double(100, 100, 150, 100);
21       shape2 = new Rectangle2D.Double(150, 150, 150, 100);
22     }
23
24     public void paintComponent(Graphics g)
25     {
26       var g2 = (Graphics2D) g;
```

http://www.rejoiceblog.com/

```

      var image = new BufferedImage(getWidth(), getHeight(
      Graphics2D gImage = image.createGraphics();
      gImage.setPaint(Color.red);
      gImage.fill(shape1);
      AlphaComposite composite = AlphaComposite.getInstanc
      gImage.setComposite(composite);
      gImage.setPaint(Color.blue);
      gImage.fill(shape2);
      g2.drawImage(image, null, 0, 0);
    }

    /**
     * Sets the composition rule.
     * @param r the rule (as an AlphaComposite constant)
     */
    public void setRule(int r)
    {
      rule = r;
      repaint();
```

http://www.rejoiceblog.com/

```
    }

    /**
     * Sets the alpha of the source.
     * @param a the alpha value between 0 and 100
     */
    public void setAlpha(int a)
    {
      alpha = (float) a / 100.0F;
      repaint();
    }
}
```

Listing 11.19 `composite/Rule.java`

```
package composite;

import java.awt.*;

/**
```

http://www.rejoiceblog.com/

```
 * This class describes a Porter-Duff rule.
 */
class Rule
{
   private String name;
   private String porterDuff1;
   private String porterDuff2;

   /**
    * Constructs a Porter-Duff rule.
    * @param n the rule name
    * @param pd1 the first row of the Porter-Duff square
    * @param pd2 the second row of the Porter-Duff square
    */
   public Rule(String n, String pd1, String pd2)
   {
      name = n;
      porterDuff1 = pd1;
      porterDuff2 = pd2;
   }
```

http://www.rejoiceblog.com/

```
26
27      /**
28       * Gets an explanation of the behavior of this rule.
29       * @return the explanation
30       */
31      public String getExplanation()
32      {
33        var r = new StringBuilder("Source ");
34        if (porterDuff2.equals(" ")) r.append("clears");
35        if (porterDuff2.equals(" S")) r.append("overwrites")
36        if (porterDuff2.equals("DS")) r.append("blends with"
37        if (porterDuff2.equals(" D")) r.append("alpha modifi
38        if (porterDuff2.equals("D ")) r.append("alpha comple
39        if (porterDuff2.equals("DD")) r.append("does not aff
40        r.append(" destination");
41        if (porterDuff1.equals(" S")) r.append(" and overwri
42        r.append(".");
43        return r.toString();
44      }
```

http://www.rejoiceblog.com/

```
45
46      public String toString()
47      {
48        return name;
49      }
50
51      /**
52       * Gets the value of this rule in the AlphaComposite c
53       * @return the AlphaComposite constant value, or -1 if
54       */
55      public int getValue()
56      {
57        try
58        {
59          return (Integer) AlphaComposite.class.getField(nam
60        }
61        catch (Exception e)
62        {
63          return -1;
64        }
```

http://www.rejoiceblog.com/

```
    }
}
```

java.awt.Graphics2D 1.2

- `void setComposite(Composite s)`

sets the composite of this graphics context to the given object that implements the `Composite` interface.

java.awt.AlphaComposite 1.2

- `static AlphaComposite getInstance(int rule)`
- `static AlphaComposite getInstance(int rule, float sourceAlpha)`

constructs an alpha composite object. The rule is one of `CLEAR`, `SRC`, `SRC_OVER`, `DST_OVER`, `SRC_IN`, `SRC_OUT`, `DST_IN`, `DST_OUT`, `DST`, `DST_ATOP`, `SRC_ATOP`, `XOR`.

http://www.rejoiceblog.com/

11.4 RASTER IMAGES

The Java2D API lets you create drawings that are made up of lines, curves, and areas. It is a "vector" API because you specify the mathematical properties of the shapes. However, for processing images that are made up of pixels, you want to work with a "raster" of color data. The following sections show you how to process raster images in Java.

11.4.1 Readers and Writers for Images

The `javax.imageio` package contains "out of the box" support for reading and writing several common file formats, as well as a framework that enables third parties to add readers and writers for other formats. The GIF, JPEG, PNG, BMP (Windows bitmap), and WBMP (wireless bitmap) file formats are supported.

The basics of the library are extremely straightforward. To load an image, use the static `read` method of the `ImageIO` class:

```
File f = . . .;

BufferedImage image = ImageIO.read(f);
```

The `ImageIO` class picks an appropriate reader, based on the file type. It may consult the file extension and the "magic number" at the beginning of the file for that purpose. If no suitable reader can be found or the reader can't decode the file contents, the `read` method returns `null`.

Writing an image to a file is just as simple:

```
File f = . . .;

String format = . . .;
```

http://www.rejoiceblog.com/

```
ImageIO.write(image, format, f);
```

Here the format string is a string identifying the image format, such as `"JPEG"` or `"PNG"`. The `ImageIO` class picks an appropriate writer and saves the file.

11.4.1.1 Obtaining Readers and Writers for Image File Types

For more advanced image reading and writing operations that go beyond the static `read` and `write` methods of the `ImageIO` class, you first need to get the appropriate `ImageReader` and `ImageWriter` objects. The `ImageIO` class enumerates readers and writers that match one of the following:

- An image format (such as `"JPEG"`)
- A file suffix (such as `"jpg"`)
- A MIME type (such as `"image/jpeg"`)

> **Note**
>
> MIME is the Multipurpose Internet Mail Extensions standard. The MIME standard defines common data formats such as `"image/jpeg"` and `"application/pdf"`.

For example, you can obtain a reader that reads JPEG files as follows:

```
ImageReader reader = null;

Iterator<ImageReader> iter = ImageIO.getImageReadersByFormatN

if (iter.hasNext()) reader = iter.next();
```

The `getImageReadersBySuffix` and `getImageReadersByMIMEType` methods enumerate readers that match a file extension or MIME type.

http://www.rejoiceblog.com/

It is possible that the `ImageIO` class can locate multiple readers that can all read a particular image type. In that case, you have to pick one of them, but it isn't clear how you can decide which one is the best. To find out more information about a reader, obtain its *service provider interface*:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Then you can get the vendor name and version number:

```
String vendor = spi.getVendor();

String version = spi.getVersion();
```

Perhaps that information can help you decide among the choices, or you might just present a list of readers to your program users and let them choose. For now, we assume that the first enumerated reader is adequate.

In the sample program in Listing 11.20, we want to find all file suffixes of all available readers so that we can use them in a file filter. Use the static `ImageIO.getReaderFileSuffixes` method for this purpose:

```
String[] extensions = ImageIO.getWriterFileSuffixes();

chooser.setFileFilter(new FileNameExtensionFilter("Image file
```

For saving files, we have to work harder. We'd like to present the user with a menu of all supported image types. Unfortunately, the `getWriterFormatNames` of the `ImageIO` class returns a rather curious list with redundant names, such as

```
jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gi
```

That's not something one would want to present in a menu. What is needed is a list of "preferred" format names. We supply a helper method `getWriterFormats` for this purpose (see Listing 11.20). We look up the first writer associated with each format

http://www.rejoiceblog.com/

name. Then we ask it what its format names are, in the hope that it will list the most popular one first. Indeed, for the JPEG writer, this works fine—it lists `"JPEG"` before the other options. (The PNG writer, on the other hand, lists `"png"` in lower case before `"PNG"`. We hope this behavior will be addressed at some point in the future. For now, we force all-lowercase names to upper case.) Once we pick a preferred name, we remove all alternate names from the original set. We keep going until all format names are handled.

11.4.1.2 Reading and Writing Files with Multiple Images

Some files—in particular, animated GIF files—contain multiple images. The `read` method of the `ImageIO` class reads a single image. To read multiple images, turn the input source (for example, an input stream or file) into an `ImageInputStream`.

```
InputStream in = . . .;

ImageInputStream imageIn = ImageIO.createImageInputStream(in)
```

Then, attach the image input stream to the reader:

```
reader.setInput(imageIn, true);
```

The second parameter indicates that the input is in “seek forward only” mode. Otherwise, random access is used, either by buffering stream input as it is read or by using random file access. Random access is required for certain operations. For example, to find out the number of images in a GIF file, you need to read the entire file. If you then want to fetch an image, the input must be read again.

This consideration is only important if you read from a stream, if the input contains multiple images, and if the image format doesn’t have the information that you request (such as the image count) in the header. If you read from a file, simply use

```
File f = . . .;

ImageInputStream imageIn = ImageIO.createImageInputStream(f);

reader.setInput(imageIn);
```

http://www.rejoiceblog.com/

Once you have a reader, you can read the images in the input by calling

```
BufferedImage image = reader.read(index);
```

where `index` is the image index, starting with `0`.

If the input is in the "seek forward only" mode, you keep reading images until the `read` method throws an `IndexOutOfBoundsException`. Otherwise, you can call the `getNumImages` method:

```
int n = reader.getNumImages(true);
```

Here, the parameter indicates that you allow a search of the input to determine the number of images. That method throws an `IllegalStateException` if the input is in the "seek forward only" mode. Alternatively, you can set the "allow search" parameter to `false`. Then the `getNumImages` method returns `-1` if it can't determine the number of images without a search. In that case, you'll have to switch to Plan B and keep reading images until you get an `IndexOutOfBoundsException`.

Some files contain thumbnails—smaller versions of an image for preview purposes. You can get the number of thumbnails of an image with the call

```
int count = reader.getNumThumbnails(index);
```

Then you get a particular index as

```
BufferedImage thumbnail = reader.getThumbnail(index, thumbnai
```

Sometimes you may want to get the image size before actually getting the image—in particular, if the image is huge or comes from a slow network connection. Use the calls

```
int width = reader.getWidth(index);
```

http://www.rejoiceblog.com/

```
int height = reader.getHeight(index);
```

to get the dimensions of an image with a given index.

To write a file with multiple images, you first need an `ImageWriter`. The `ImageIO` class can enumerate the writers capable of writing a particular image format:

```
String format = . . .;

ImageWriter writer = null;

Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatN

if (iter.hasNext()) writer = iter.next();
```

Next, turn an output stream or file into an `ImageOutputStream` and attach it to the writer. For example,

```
File f = . . .;

ImageOutputStream imageOut = ImageIO.createImageOutputStream(

writer.setOutput(imageOut);
```

You must wrap each image into an `IIOImage` object. You can optionally supply a list of thumbnails and image metadata (such as compression algorithms and color information). In this example, we just use `null` for both; see the API documentation for additional information.

```
var iioImage = new IIOImage(images[i], null, null);
```

To write out the *first* image, use the `write` method:

http://www.rejoiceblog.com/

```
writer.write(new IIOImage(images[0], null, null));
```

For subsequent images, use

```
if (writer.canInsertImage(i))

   writer.writeInsert(i, iioImage, null);
```

The third parameter can contain an `ImageWriteParam` object to set image writing details such as tiling and compression; use `null` for default values.

Not all file formats can handle multiple images. In that case, the `canInsertImage` method returns `false` for `i` > 0, and only a single image is saved.

The program in Listing 11.20 lets you load and save files in the formats for which the Java library supplies readers and writers. The program displays multiple images (see Figure 11.55), but not thumbnails.

http://www.rejoiceblog.com/

Figure 11.55 An animated GIF image

Listing 11.20 `imageIO/ImageIOFrame.java`

```
package imageIO;

import java.awt.image.*;
import java.io.*;
import java.util.*;

import javax.imageio.*;
```

http://www.rejoiceblog.com/

```
 8  import javax.imageio.stream.*;
 9  import javax.swing.*;
10  import javax.swing.filechooser.*;
11
12  /**
13   * This frame displays the loaded images. The menu has it
14   */
15  public class ImageIOFrame extends JFrame
16  {
17     private static final int DEFAULT_WIDTH = 400;
18     private static final int DEFAULT_HEIGHT = 400;
19
20     private static Set<String> writerFormats = getWriterFo
21
22     private BufferedImage[] images;
23
24     public ImageIOFrame()
25     {
26       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

http://www.rejoiceblog.com/

```
27
28        var fileMenu = new JMenu("File");
29        var openItem = new JMenuItem("Open");
30        openItem.addActionListener(event -> openFile());
31        fileMenu.add(openItem);
32
33        var saveMenu = new JMenu("Save");
34        fileMenu.add(saveMenu);
35        Iterator<String> iter = writerFormats.iterator();
36        while (iter.hasNext())
37        {
38          final String formatName = iter.next();
39          var formatItem = new JMenuItem(formatName);
40          saveMenu.add(formatItem);
41          formatItem.addActionListener(event -> saveFile(for
42        }
43
44        var exitItem = new JMenuItem("Exit");
45        exitItem.addActionListener(event -> System.exit(0));
46        fileMenu.add(exitItem);
```

http://www.rejoiceblog.com/

```
47
48        var menuBar = new JMenuBar();
49        menuBar.add(fileMenu);
50        setJMenuBar(menuBar);
51     }
52
53     /**
54      * Open a file and load the images.
55      */
56     public void openFile()
57     {
58        var chooser = new JFileChooser();
59        chooser.setCurrentDirectory(new File("."));
60        String[] extensions = ImageIO.getReaderFileSuffixes(
61        chooser.setFileFilter(new FileNameExtensionFilter("I
62        int r = chooser.showOpenDialog(this);
63        if (r != JFileChooser.APPROVE_OPTION) return;
64        File f = chooser.getSelectedFile();
65        Box box = Box.createVerticalBox();
```

http://www.rejoiceblog.com/

```
66        try
67        {
68          String name = f.getName();
69          String suffix = name.substring(name.lastIndexOf('.
70          Iterator<ImageReader> iter = ImageIO.getImageReade
71          ImageReader reader = iter.next();
72          ImageInputStream imageIn = ImageIO.createImageInpu
73          reader.setInput(imageIn);
74          int count = reader.getNumImages(true);
75          images = new BufferedImage[count];
76          for (int i = 0; i < count; i++)
77          {
78            images[i] = reader.read(i);
79            box.add(new JLabel(new ImageIcon(images[i])));
80          }
81        }
82        catch (IOException e)
83        {
84          JOptionPane.showMessageDialog(this, e);
85        }
```

http://www.rejoiceblog.com/

```
86        setContentPane(new JScrollPane(box));
87        validate();
88      }
89
90      /**
91       * Save the current image in a file.
92       * @param formatName the file format
93       */
94      public void saveFile(final String formatName)
95      {
96        if (images == null) return;
97        Iterator<ImageWriter> iter = ImageIO.getImageWriters
98        ImageWriter writer = iter.next();
99        var chooser = new JFileChooser();
100       chooser.setCurrentDirectory(new File("."));
101       String[] extensions = writer.getOriginatingProvider(
102       chooser.setFileFilter(new FileNameExtensionFilter("I
103
104       int r = chooser.showSaveDialog(this);
```

http://www.rejoiceblog.com/

```
105      if (r != JFileChooser.APPROVE_OPTION) return;
106      File f = chooser.getSelectedFile();
107      try
108      {
109        ImageOutputStream imageOut = ImageIO.createImageOu
110        writer.setOutput(imageOut);
111
112        writer.write(new IIOImage(images[0], null, null));
113        for (int i = 1; i < images.length; i++)
114        {
115          var iioImage = new IIOImage(images[i], null, nul
116          if (writer.canInsertImage(i)) writer.writeInsert
117        }
118      }
119      catch (IOException e)
120      {
121        JOptionPane.showMessageDialog(this, e);
122     }
123   }
124
```

http://www.rejoiceblog.com/

```
/**
 * Gets a set of "preferred" format names of all image
 * first format name that a writer specifies.
 * @return the format name set
 */
public static Set<String> getWriterFormats()
{
  var writerFormats = new TreeSet<>();
  var formatNames = new TreeSet<>(List.of(ImageIO.getWr
  while (formatNames.size() > 0)
  {
      String name = formatNames.iterator().next();
      Iterator<ImageWriter> iter = ImageIO.getImageWrit
      ImageWriter writer = iter.next();
      String[] names = writer.getOriginatingProvider().
      String format = names[0];
      if (format.equals(format.toLowerCase())) format =
      writerFormats.add(format);
      formatNames.removeAll(List.of(names));
```

http://www.rejoiceblog.com/

```
    }

    return writerFormats;

  }

}
```

javax.imageio.ImageIO 1.4

- `static BufferedImage read(File input)`
- `static BufferedImage read(InputStream input)`
- `static BufferedImage read(URL input)`

reads an image from `input`.

- `static boolean write(RenderedImage image, String formatName, File output)`
- `static boolean write(RenderedImage image, String formatName, OutputStream output)`

writes an image in the given format to `output`. Returns `false` if no appropriate writer was found.

- `static Iterator<ImageReader> getImageReadersByFormatName(String formatName)`
- `static Iterator<ImageReader> getImageReadersBySuffix(String fileSuffix)`
- `static Iterator<ImageReader> getImageReadersByMIMEType(String mimeType)`
- `static Iterator<ImageWriter> getImageWritersByFormatName(String formatName)`

http://www.rejoiceblog.com/

- `static Iterator<ImageWriter> getImageWritersBySuffix(String fileSuffix)`

- `static Iterator<ImageWriter> getImageWritersByMIMEType(String mimeType)`

gets all readers and writers that are able to handle the given format (e.g., `"JPEG"`), file suffix (e.g., `"jpg"`), or MIME type (e.g., `"image/jpeg"`).

- `static String[] getReaderFormatNames()`

- `static String[] getReaderMIMETypes()`

- `static String[] getWriterFormatNames()`

- `static String[] getWriterMIMETypes()`

- `static String[] getReaderFileSuffixes()` 6

- `static String[] getWriterFileSuffixes()` 6

gets all format names, MIME type names, and file suffixes supported by readers and writers.

- `ImageInputStream createImageInputStream(Object input)`

- `ImageOutputStream createImageOutputStream(Object output)`

creates an image input or image output stream from the given object. The object can be a file, a stream, a `RandomAccessFile`, or another object for which a service provider exists. Returns `null` if no registered service provider can handle the object.

javax.imageio.ImageReader 1.4

- `void setInput(Object input)`

- `void setInput(Object input, boolean seekForwardOnly)`

sets the input source of the reader.

http://www.rejoiceblog.com/

Parameters:	input	An ImageInputStream object or another object that this reader can accept.
	seekForwardOnly	true if the reader should read forward only. By default, the reader uses random access and, if necessary, buffers image data.

- `BufferedImage read(int index)`

reads the image with the given image index (starting at `0`). Throws an `IndexOutOfBoundsException` if no such image is available.

- `int getNumImages(boolean allowSearch)`

gets the number of images in this reader. If `allowSearch` is `false` and the number of images cannot be determined without reading forward, then `-1` is returned. If `allowSearch` is `true` and the reader is in the "seek forward only" mode, then an `IllegalStateException` is thrown.

- `int getNumThumbnails(int index)`

gets the number of thumbnails of the image with the given `index`.

- `BufferedImage readThumbnail(int index, int thumbnailIndex)`

gets the thumbnail with index `thumbnailIndex` of the image with the given `index`.

- `int getWidth(int index)`

- `int getHeight(int index)`

gets the image width and height. Throws an `IndexOutOfBoundsException` if no such image is available.

- `ImageReaderSpi getOriginatingProvider()`

gets the service provider that constructed this reader.

javax.imageio.spi.IIOServiceProvider 1.4

- `String getVendorName()`

http://www.rejoiceblog.com/

- `String getVersion()`

gets the vendor name and version of this service provider.

javax.imageio.spi.ImageReaderWriterSpi 1.4

- `String[] getFormatNames()`
- `String[] getFileSuffixes()`
- `String[] getMIMETypes()`

gets the format names, file suffixes, and MIME types supported by the readers or writers that this service provider creates.

javax.imageio.ImageWriter 1.4

- `void setOutput(Object output)`

sets the output target of this writer.

Parameters:	`output`	An `ImageOutputStream` object or another object that this writer can accept

- `void write(IIOImage image)`
- `void write(RenderedImage image)`

writes a single image to the output.

- `void writeInsert(int index, IIOImage image, ImageWriteParam param)`

writes an image into a multi-image file.

- `boolean canInsertImage(int index)`

returns `true` if it is possible to insert an image at the given index.

http://www.rejoiceblog.com/

- `ImageWriterSpi getOriginatingProvider()`

gets the service provider that constructed this writer.

javax.imageio.IIOImage 1.4

- `IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)`

constructs an `IIOImage` from an image, optional thumbnails, and optional metadata.

11.4.2 Image Manipulation

Suppose you have an image and you would like to improve its appearance. You then need to access the individual pixels of the image and replace them with other pixels. Or perhaps you want to compute the pixels of an image from scratch—for example, to show the result of physical measurements or a mathematical computation. The `BufferedImage` class gives you control over the pixels in an image, and the classes that implement the `BufferedImageOp` interface let you transform images.

Note

JDK 1.0 had a completely different, and far more complex, imaging framework that was optimized for *incremental rendering* of images downloaded from the Web, a scan line at a time. However, it was difficult to manipulate those images. We do not discuss that framework in this book.

11.4.2.1 Constructing Raster Images

Most of the images that you manipulate are simply read in from an image file—they were either produced by a device such as a digital camera or scanner, or constructed by a drawing program. In this section, we'll show you a differ-

ent technique for constructing an image—namely, building it up a pixel at a time.

To create an image, construct a `BufferedImage` object in the usual way.

http://www.rejoiceblog.com/

```
image = new BufferedImage(width, height, BufferedImage.TYPE_I
```

Now, call the `getRaster` method to obtain an object of type `WritableRaster`. You will use this object to access and modify the pixels of the image.

```
WritableRaster raster = image.getRaster();
```

The `setPixel` method lets you set an individual pixel. The complexity here is that you can't simply set the pixel to a `Color` value. You must know how the buffered image specifies color values. That depends on the *type* of the image. If your image has a type of `TYPE_INT_ARGB`, then each pixel is described by four values—red, green, blue, and alpha, each between `0` and `255`. You have to supply them in an array of four integers:

```
int[] black = { 0, 0, 0, 255 };

raster.setPixel(i, j, black);
```

In the lingo of the Java 2D API, these values are called the *sample values* of the pixel.

Caution

There are also `setPixel` methods that take array parameters of types `float[]` and `double[]`. However, the values that you need to place into these arrays are *not* normalized color values between `0.0` and `1.0`.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };

raster.setPixel(i, j, red); // ERROR
```

You need to supply values between `0` and `255`, no matter what the type of the array is.

http://www.rejoiceblog.com/

You can supply batches of pixels with the `setPixels` method. Specify the starting pixel position and the width and height of the rectangle that you want to set. Then, supply an array that contains the sample values for all pixels. For example, if your buffered image has a type of `TYPE_INT_ARGB`, supply the red, green, blue, and alpha values of the first pixel, then the red, green, blue, and alpha values for the second pixel, and so on.

```
var pixels = new int[4 * width * height];

pixels[0] = . . . // red value for first pixel

pixels[1] = . . . // green value for first pixel

pixels[2] = . . . // blue value for first pixel

pixels[3] = . . . // alpha value for first pixel

. . .

raster.setPixels(x, y, width, height, pixels);
```

Conversely, to read a pixel, use the `getPixel` method. Supply an array of four integers to hold the sample values.

```
var sample = new int[4];

raster.getPixel(x, y, sample);

var color = new Color(sample[0], sample[1], sample[2], sample
```

You can read multiple pixels with the `getPixels` method.

```
raster.getPixels(x, y, width, height, samples);
```

If you use an image type other than `TYPE_INT_ARGB` and you know how that type

http://www.rejoiceblog.com/

represents pixel values, you can still use the `getPixel/setPixel` methods. However, you have to know the encoding of the sample values in the particular image type.

If you need to manipulate an image with an arbitrary, unknown image type, then you have to work a bit harder. Every image type has a *color model* that can translate between sample value arrays and the standard RGB color model.

Note

The RGB color model isn't as standard as you might think. The exact look of a color value depends on the characteristics of the imaging device. Digital cameras, scanners, monitors, and LCD displays all have their own idiosyncrasies. As a result, the same RGB value can look quite different on different devices. The International Color Consortium (`www.color.org`) recommends that all color data be accompanied by an *ICC profile* that specifies how the colors map to a standard form such as the 1931 CIE XYZ color specification. That specification was designed by the Commission Internationale de l'Eclairage, or CIE (`www.cie.co.at`), the international organization in charge of providing technical guidance in all matters of illumination and color. The specification is a standard method for representing any color that the human eye can perceive as a triplet of coordinates called X, Y, Z. (See, for example, *Computer Graphics: Principles and Practice, Second Edition in C*, by James D. Foley, Andries van Dam, Steven K. Feiner, et al., Chapter 13, for more information on the 1931 CIE XYZ specification.)

ICC profiles are complex, however. A simpler proposed standard, called sRGB (`www.w3.org/Images/Color/sRGB.html`), specifies an exact mapping between RGB values and the 1931 CIE XYZ values that was designed to work well with typical color monitors. The Java 2D API uses that mapping when converting between RGB and other color spaces.

The `getColorModel` method returns the color model:

```
ColorModel model = image.getColorModel();
```

To find the color value of a pixel, call the `getDataElements` method of the `Raster` class. That call returns an `Object` that contains a color-model-specific description of

http://www.rejoiceblog.com/

the color value.

```
Object data = raster.getDataElements(x, y, null);
```

Note

The object that is returned by the `getDataElements` method is actually an array of sample values. You don't need to know this to process the object, but it explains why the method is called `getDataElements`.

The color model can translate the object to standard ARGB values. The `getRGB` method returns an `int` value that has the alpha, red, green, and blue values packed in four blocks of eight bits each. You can construct a `Color` value out of that integer with the `Color(int argb, boolean hasAlpha)` constructor:

```
int argb = model.getRGB(data);

var color = new Color(argb, true);
```

To set a pixel to a particular color, reverse these steps. The `getRGB` method of the `Color` class yields an `int` value with the alpha, red, green, and blue values. Supply that value to the `getDataElements` method of the `ColorModel` class. The return value is an `Object` that contains the color-model-specific description of the color value. Pass the object to the `setDataElements` method of the `WritableRaster` class.

```
int argb = color.getRGB();

Object data = model.getDataElements(argb, null);

raster.setDataElements(x, y, data);
```

To illustrate how to use these methods to build an image from individual pixels, we bow to tradition and draw a Mandelbrot set, as shown in Figure 11.56.

http://www.rejoiceblog.com/

Figure 11.56 A Mandelbrot set

The idea of the Mandelbrot set is that each point of the plane is associated with a sequence of numbers. If that sequence stays bounded, you color the point. If it "escapes to infinity," you leave it transparent.

Here is how you can construct the simplest Mandelbrot set. For each point (a, b), look at sequences that start with $(x, y) = (0, 0)$ and iterate:

$$x_{new} = x^2 - y^2 + a$$

$$y_{new} = 2 \cdot x \cdot y + b$$

It turns out that if x or y ever gets larger than 2, then the sequence escapes to infinity. Only the pixels that correspond to points (a, b) leading to a bounded sequence are colored. (The formulas for the number sequences come ultimately from the mathematics of complex numbers; we'll just take them for granted. For more on the mathematics of fractals, see, for example, `http://classes.yale.edu/fractals`.)

Listing 11.21 shows the code. In this program, we demonstrate how to use the `ColorModel` class for translating `Color` values into pixel data. That process is

http://www.rejoiceblog.com/

independent of the image type. Just for fun, change the color type of the buffered image to `TYPE_BYTE_GRAY`. You don't need to change any other code—the color model of the image automatically takes care of the conversion from colors to sample values.

Listing 11.21 `rasterImage/RasterImageFrame.java`

```
package rasterImage;

import java.awt.*;
import java.awt.image.*;
import javax.swing.*;

/**
 * This frame shows an image with a Mandelbrot set.
 */
public class RasterImageFrame extends JFrame
{
   private static final double XMIN = -2;
   private static final double XMAX = 2;
   private static final double YMIN = -2;
   private static final double YMAX = 2;
   private static final int MAX_ITERATIONS = 16;
   private static final int IMAGE_WIDTH = 400;
```

http://www.rejoiceblog.com/

```
18      private static final int IMAGE_HEIGHT = 400;
19
20      public RasterImageFrame()
21      {
22        BufferedImage image = makeMandelbrot(IMAGE_WIDTH, IM
23        add(new JLabel(new ImageIcon(image)));
24        pack();
25      }
26
27      /**
28       * Makes the Mandelbrot image.
29       * @param width the width
30       * @parah height the height
31       * @return the image
32       */
33      public BufferedImage makeMandelbrot(int width, int hei
34      {
35        var image = new BufferedImage(width, height, Buffere
36        WritableRaster raster = image.getRaster();
```

http://www.rejoiceblog.com/

```
      ColorModel model = image.getColorModel();

      Color fractalColor = Color.RED;
      int argb = fractalColor.getRGB();
      Object colorData = model.getDataElements(argb, null)

      for (int i = 0; i < width; i++)
        for (int j = 0; j < height; j++)
        {
          double a = XMIN + i * (XMAX - XMIN) / width;
          double b = YMIN + j * (YMAX - YMIN) / height;
          if (!escapesToInfinity(a, b)) raster.setDataElem
        }
      return image;
    }

    private boolean escapesToInfinity(double a, double b)
    {
      double x = 0.0;
      double y = 0.0;
```

http://www.rejoiceblog.com/

```
      int iterations = 0;
      while (x <= 2 && y <= 2 && iterations < MAX_ITERATIO
      {
        double xnew = x * x - y * y + a;
        double ynew = 2 * x * y + b;
        x = xnew;
        y = ynew;
        iterations++;
      }
      return x > 2 || y > 2;
    }
  }
```

java.awt.image.BufferedImage 1.2

- `BufferedImage(int width, int height, int imageType)`

constructs a buffered image object.

Parameters:	`width, height`	The image dimensions
	`imageType`	The image type. The most common types are `TYPE_INT_RGB`, `TYPE_INT_ARGB`, `TYPE_BYTE_GRAY`, and `TYPE_BYTE_INDEXED`.

- `ColorModel getColorModel()`

http://www.rejoiceblog.com/

returns the color model of this buffered image.

- `WritableRaster getRaster()`

gets the raster for accessing and modifying pixels of this buffered image.

java.awt.image.Raster 1.2

- `Object getDataElements(int x, int y, Object data)`

returns the sample data for a raster point, in an array whose element type and length depend on the color model. If `data` is not `null`, it is assumed to be an array that is appropriate for holding sample data, and it is filled. If `data` is `null`, a new array is allocated. Its element type and length depend on the color model.

- `int[] getPixel(int x, int y, int[] sampleValues)`

- `float[] getPixel(int x, int y, float[] sampleValues)`

- `double[] getPixel(int x, int y, double[] sampleValues)`

- `int[] getPixels(int x, int y, int width, int height, int[] sampleValues)`

- `float[] getPixels(int x, int y, int width, int height, float[] sampleValues)`

- `double[] getPixels(int x, int y, int width, int height, double[] sampleValues)`

returns the sample values for a raster point, or a rectangle of raster points, in an array whose length depends on the color model. If `sampleValues` is not `null`, it is assumed to be sufficiently long for holding the sample values, and it is filled. If `sampleValues` is `null`, a new array is allocated. These methods are only useful if you know the meaning of the sample values for a color model.

java.awt.image.WritableRaster 1.2

http://www.rejoiceblog.com/

- `void setDataElements(int x, int y, Object data)`

sets the sample data for a raster point. `data` is an array filled with the sample data for a pixel. Its element type and length depend on the color model.

- `void setPixel(int x, int y, int[] sampleValues)`

- `void setPixel(int x, int y, float[] sampleValues)`

- `void setPixel(int x, int y, double[] sampleValues)`

- `void setPixels(int x, int y, int width, int height, int[] sampleValues)`

- `void setPixels(int x, int y, int width, int height, float[] sampleValues)`

- `void setPixels(int x, int y, int width, int height, double[] sampleValues)`

sets the sample values for a raster point or a rectangle of raster points. These methods are only useful if you know the encoding of the sample values for a color model.

java.awt.image.ColorModel 1.2

- `int getRGB(Object data)`

returns the ARGB value that corresponds to the sample data passed in the `data` array. Its element type and length depend on the color model.

- `Object getDataElements(int argb, Object data);`

returns the sample data for a color value. If `data` is not `null`, it is assumed to be an array that is appropriate for holding sample data, and it is filled. If `data` is `null`, a new array is allocated. `data` is an array filled with the sample data for a pixel. Its element type and length depend on the color model.

java.awt.Color 1.0

http://www.rejoiceblog.com/

- `Color(int argb, boolean hasAlpha)` **1.2**

creates a color with the specified combined ARGB value if `hasAlpha` is `true`, or the specified RGB value if `hasAlpha` is `false`.

- `int getRGB()`

returns the ARGB color value corresponding to this color.

11.4.2.2 Filtering Images

In the preceding section, you saw how to build up an image from scratch. However, often you want to access image data for a different reason: You already have an image and you want to improve it in some way.

Of course, you can use the `getPixel/getDataElements` methods that you saw in the preceding section to read the image data, manipulate them, and write them back. But fortunately, the Java 2D API already supplies a number of *filters* that carry out common image processing operations for you.

The image manipulations all implement the `BufferedImageOp` interface. After you construct the operation, you simply call the `filter` method to transform an image into another.

```
BufferedImageOp op = . . .;

BufferedImage filteredImage =

     new BufferedImage(image.getWidth(), image.getHeight(), i

op.filter(image, filteredImage);
```

Some operations can transform an image in place (`op.filter(image, image)`), but most can't.

Five classes implement the `BufferedImageOp` interface:

```
AffineTransformOp
```

http://www.rejoiceblog.com/

```
RescaleOp

LookupOp

ColorConvertOp

ConvolveOp
```

The `AffineTransformOp` carries out an affine transformation on the pixels. For example, here is how you can rotate an image about its center:

```
AffineTransform transform = AffineTransform.getRotateInstance

     image.getWidth() / 2, image.getHeight() / 2);

var op = new AffineTransformOp(transform, interpolation);

op.filter(image, filteredImage);
```

The `AffineTransformOp` constructor requires an affine transform and an *interpolation* strategy. Interpolation is necessary to determine pixels in the target image if the source pixels are transformed somewhere between target pixels. For example, if you rotate source pixels, then they will generally not fall exactly onto target pixels. There are two interpolation strategies: `AffineTransformOp.TYPE_BILINEAR` and `AffineTransformOp.TYPE_NEAREST_NEIGHBOR`. Bilinear interpolation takes a bit longer but looks better.

The program in Listing 11.22 lets you rotate an image by 5 degrees (see Figure 11.57).

http://www.rejoiceblog.com/

Figure 11.57 A rotated image

The `RescaleOp` carries out a rescaling operation

$$x_{\text{new}} = a \cdot x + b$$

for each of the color components in the image. (Alpha components are not affected.) The effect of rescaling with $a > 1$ is to brighten the image. Construct the `RescaleOp` by specifying the scaling parameters and optional rendering hints. In Listing 11.22, we use:

```
float a = 1.1f;

float b = 20.0f;

var op = new RescaleOp(a, b, null);
```

You can also supply separate scaling values for each color component—see the API notes.

The `LookupOp` operation lets you specify an arbitrary mapping of sample values. Supply a table that specifies how each value should be mapped. In the example program, we compute the *negative* of all colors, changing the color c to $255 - c$.

The `LookupOp` constructor requires an object of type `LookupTable` and a map of optional hints. The `LookupTable` class is abstract, with two concrete subclasses:

http://www.rejoiceblog.com/

`ByteLookupTable` and `ShortLookupTable`. Since RGB color values are bytes, a `ByteLookupTable` should suffice. However, because of the bug described in `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6183251`, we will use a `ShortLookupTable` instead. Here is how we construct the `LookupOp` for the example program:

```
var negative = new short[256];

for (int i = 0; i < 256; i++) negative[i] = (short) (255 - i)

var table = new ShortLookupTable(0, negative);

var op = new LookupOp(table, null);
```

The lookup is applied to each color component separately, but not to the alpha component. You can also supply different lookup tables for each color component—see the API notes.

Note

You cannot apply a `LookupOp` to an image with an indexed color model. (In those images, each sample value is an offset into a color palette.)

The `ColorConvertOp` is useful for color space conversions. We do not discuss it here.

The most powerful of the transformations is the `ConvolveOp`, which carries out a mathematical *convolution*. We won't get too deeply into the mathematical details, but the basic idea is simple. Consider, for example, the *blur filter* (see Figure 11.58).

http://www.rejoiceblog.com/

Figure 11.58 Blurring an image

The blurring is achieved by replacing each pixel with the *average* value from the pixel and its eight neighbors. Intuitively, it makes sense why this operation would blur out the picture. Mathematically, the averaging can be expressed as a convolution operation with the following *kernel:*

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

The kernel of a convolution is a matrix that tells what weights should be applied to the neighboring values. The kernel above produces a blurred image. A different kernel carries out *edge detection,* locating the areas of color changes:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Edge detection is an important technique for analyzing photographic images (see Figure 11.59).

http://www.rejoiceblog.com/

Figure 11.59 Edge detection and inversion

To construct a convolution operation, first set up an array of the values for the kernel and construct a `Kernel` object. Then, construct a `ConvolveOp` object from the kernel and use it for filtering.

```
float[] elements =
   {
      0.0f, -1.0f, 0.0f,
      -1.0f, 4.f, -1.0f,
      0.0f, -1.0f, 0.0f
   };
var kernel = new Kernel(3, 3, elements);
var op = new ConvolveOp(kernel);
op.filter(image, filteredImage);
```

The program in Listing 11.22 allows a user to load in a GIF or JPEG image and carry out

http://www.rejoiceblog.com/

the image manipulations that we discussed. Thanks to the power of the operations provided by Java 2D API, the program is very simple.

Listing 11.22 `imageProcessing/ImageProcessingFrame.java`

```
1  package imageProcessing;
2
3  import java.awt.*;
4  import java.awt.geom.*;
5  import java.awt.image.*;
6  import java.io.*;
7
8  import javax.imageio.*;
9  import javax.swing.*;
10 import javax.swing.filechooser.*;
11
12 /**
13  * This frame has a menu to load an image and to specify
14  * show the resulting image.
15  */
16 public class ImageProcessingFrame extends JFrame
17 {
```

http://www.rejoiceblog.com/

```
    private static final int DEFAULT_WIDTH = 400;

    private static final int DEFAULT_HEIGHT = 400;

    private BufferedImage image;

    public ImageProcessingFrame()
    {
        setTitle("ImageProcessingTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        add(new JComponent()
          {
            public void paintComponent(Graphics g)
            {
              if (image != null) g.drawImage(image, 0, 0, 
            }
          });

        var fileMenu = new JMenu("File");
        var openItem = new JMenuItem("Open");
```

http://www.rejoiceblog.com/

```
        openItem.addActionListener(event -> openFile());
        fileMenu.add(openItem);

        var exitItem = new JMenuItem("Exit");
        exitItem.addActionListener(event -> System.exit(0)
        fileMenu.add(exitItem);

        var editMenu = new JMenu("Edit");
        var blurItem = new JMenuItem("Blur");
        blurItem.addActionListener(event ->
           {
              float weight = 1.0f / 9.0f;
              float[] elements = new float[9];
              for (int i = 0; i < 9; i++)
              elements[i] = weight;
              convolve(elements);
           });
        editMenu.add(blurItem);

```

http://www.rejoiceblog.com/

```
        var sharpenItem = new JMenuItem("Sharpen");
        sharpenItem.addActionListener(event ->
        {
          float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 5
          convolve(elements);
        });
    editMenu.add(sharpenItem);

    var brightenItem = new JMenuItem("Brighten");
    brightenItem.addActionListener(event ->
      {
        float a = 1.1f;
        float b = 20.0f;
        var op = new RescaleOp(a, b, null);
        filter(op);
      });
    editMenu.add(brightenItem);

    var edgeDetectItem = new JMenuItem("Edge detect");
    edgeDetectItem.addActionListener(event ->
```

http://www.rejoiceblog.com/

```
      {
        float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 4.f
        convolve(elements);
      });
    editMenu.add(edgeDetectItem);

    var negativeItem = new JMenuItem("Negative");
    negativeItem.addActionListener(event ->
      {
        short[] negative = new short[256 * 1];
        for (int i = 0; i < 256; i++)
          negative[i] = (short) (255 - i);
        var table = new ShortLookupTable(0, negative);
        var op = new LookupOp(table, null);
        filter(op);
      });
    editMenu.add(negativeItem);

    var rotateItem = new JMenuItem("Rotate");
```

http://www.rejoiceblog.com/

```
    rotateItem.addActionListener(event ->
      {
        if (image == null) return;
        var transform = AffineTransform.getRotateInstance(
            image.getWidth() / 2, image.getHeight() / 2);
        var op = new AffineTransformOp(transform,
            AffineTransformOp.TYPE_BICUBIC);
        filter(op);
      });
    editMenu.add(rotateItem);

    var menuBar = new JMenuBar();
    menuBar.add(fileMenu);
    menuBar.add(editMenu);
    setJMenuBar(menuBar);
  }

  /**
   * Open a file and load the image.
   */
```

http://www.rejoiceblog.com/

```
public void openFile()
{
  var chooser = new JFileChooser(".");
  chooser.setCurrentDirectory(new File(getClass().getPa
  String[] extensions = ImageIO.getReaderFileSuffixes()
  chooser.setFileFilter(new FileNameExtensionFilter("Im
  int r = chooser.showOpenDialog(this);
  if (r != JFileChooser.APPROVE_OPTION) return;

  try
  {
    Image img = ImageIO.read(chooser.getSelectedFile())
    image = new BufferedImage(img.getWidth(null), img.g
         BufferedImage.TYPE_INT_RGB);
    image.getGraphics().drawImage(img, 0, 0, null);
  }
  catch (IOException e)
  {
    JOptionPane.showMessageDialog(this, e);
```

http://www.rejoiceblog.com/

```
    }
    repaint();
  }

  /**
   * Apply a filter and repaint.
   * @param op the image operation to apply
   */
  private void filter(BufferedImageOp op)
  {
    if (image == null) return;
    image = op.filter(image, null);
    repaint();
  }

  /**
   * Apply a convolution and repaint.
   * @param elements the convolution kernel (an array of
   */
  private void convolve(float[] elements)
```

http://www.rejoiceblog.com/

```
  {
    var kernel = new Kernel(3, 3, elements);
    var op = new ConvolveOp(kernel);
    filter(op);
  }
}
```

java.awt.image.BufferedImageOp 1.2

- `BufferedImage filter(BufferedImage source, BufferedImage dest)`

applies the image operation to the source image and stores the result in the destination image. If `dest` is `null`, a new destination image is created. The destination image is returned.

java.awt.image.AffineTransformOp 1.2

- `AffineTransformOp(AffineTransform t, int interpolationType)`

constructs an affine transform operator. The interpolation type is one of `TYPE_BILINEAR`, `TYPE_BICUBIC`, or `TYPE_NEAREST_NEIGHBOR`.

java.awt.image.RescaleOp 1.2

- `RescaleOp(float a, float b, RenderingHints hints)`
- `RescaleOp(float[] as, float[] bs, RenderingHints hints)`

constructs a rescale operator that carries out the scaling operation $x_{\text{new}} = a \cdot x + b$.

http://www.rejoiceblog.com/

When using the first constructor, all color components (but not the alpha component) are scaled with the same coefficients. When using the second constructor, you supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

java.awt.image.LookupOp 1.2

- `LookupOp(LookupTable table, RenderingHints hints)`

constructs a lookup operator for the given lookup table.

java.awt.image.ByteLookupTable 1.2

- `ByteLookupTable(int offset, byte[] data)`
- `ByteLookupTable(int offset, byte[][] data)`

constructs a lookup table for converting `byte` values. The offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, you supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

java.awt.image.ShortLookupTable 1.2

- `ShortLookupTable(int offset, short[] data)`
- `ShortLookupTable(int offset, short[][] data)`

constructs a lookup table for converting `short` values. The offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, you supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

http://www.rejoiceblog.com/

java.awt.image.ConvolveOp 1.2

- `ConvolveOp(Kernel kernel)`
- `ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)`

constructs a convolution operator. The edge condition specified is one of `EDGE_NO_OP` and `EDGE_ZERO_FILL`. Edge values need to be treated specially because they don't have sufficient neighboring values to compute the convolution. The default is `EDGE_ZERO_FILL`.

java.awt.image.Kernel 1.2

- `Kernel(int width, int height, float[] matrixElements)`

constructs a kernel for the given matrix.

11.5 PRINTING

In the following sections, we will show you how you can easily print a drawing on a single sheet of paper, how you can manage a multipage printout, and how you can save a printout as a PostScript file.

11.5.1 Graphics Printing

In this section, we will tackle what is probably the most common printing situation: printing a 2D graphic. Of course, the graphic can contain text in various fonts or even consist entirely of text.

To generate a printout, you have to take care of these two tasks:

- Supply an object that implements the `Printable` interface
- Start a print job

The `Printable` interface has a single method:

http://www.rejoiceblog.com/

```
int print(Graphics g, PageFormat format, int page)
```

That method is called whenever the print engine needs to have a page formatted for printing. Your code draws the text and the images to be printed onto the graphics context. The page format tells you the paper size and the print margins. The page number tells you which page to render.

To start a print job, use the `PrinterJob` class. First, call the static `getPrinterJob` method to get a print job object. Then set the `Printable` object that you want to print.

```
Printable canvas = . . .;

PrinterJob job = PrinterJob.getPrinterJob();

job.setPrintable(canvas);
```

Caution

The class `PrintJob` handles JDK 1.1-style printing. That class is now obsolete. Do not confuse it with the `PrinterJob` class.

Before starting the print job, you should call the `printDialog` method to display a print dialog box (see Figure 11.60). That dialog box gives the user a chance to select the printer to be used (in case multiple printers are available), the page range that should be printed, and various printer settings.

http://www.rejoiceblog.com/

Figure 11.60 A cross-platform print dialog box

Collect printer settings in an object of a class that implements the `PrintRequestAttributeSet` interface, such as the `HashPrintRequestAttributeSet` class.

```
var attributes = new HashPrintRequestAttributeSet();
```

Add attribute settings and pass the `attributes` object to the `printDialog` method.

The `printDialog` method returns `true` if the user clicked OK and `false` if the user canceled the dialog box. If the user accepted, call the `print` method of the `PrinterJob` class to start the printing process. The `print` method might throw a `PrinterException`. Here is the outline of the printing code:

```
if (job.printDialog(attributes))

{
```

http://www.rejoiceblog.com/

```
    try
    {
      job.print(attributes);
    }
     catch (PrinterException exception)
    {
      . . .
    }
}
```

Note

Prior to JDK 1.4, the printing system used the native print and page setup dialog boxes of the host platform. To show a native print dialog box, call the `printDialog` method with no parameters. (There is no way to collect user settings in an attribute set.)

During printing, the `print` method of the `PrinterJob` class makes repeated calls to the `print` method of the `Printable` object associated with the job.

Since the job does not know how many pages you want to print, it simply keeps calling the `print` method. As long as the `print` method returns the value `Printable.PAGE_EXISTS`, the print job keeps producing pages. When the `print` method returns `Printable.NO_SUCH_PAGE`, the print job stops.

http://www.rejoiceblog.com/

Caution

The page numbers that the print job passes to the `print` method start with page 0.

Therefore, the print job doesn't have an accurate page count until after the printout is complete. For that reason, the print dialog box can't display the correct page range—instead it displays "Pages 1 to 1." You will see in the next section how to avoid this blemish by supplying a `Book` object to the print job.

During the printing process, the print job repeatedly calls the `print` method of the `Printable` object. The print job is allowed to make multiple calls *for the same page*. You should therefore not count pages inside the `print` method but always rely on the page number parameter. There is a good reason why the print job might call the `print` method repeatedly for the same page. Some printers, in particular dot-matrix and inkjet printers, use *banding*. They print one band at a time, advance the paper, and then print the next band. The print job might use banding even for laser printers that print a full page at a time—it gives the print job a way of managing the size of the spool file.

If the print job needs the `Printable` object to print a band, it sets the clip area of the graphics context to the requested band and calls the `print` method. Its drawing operations are clipped against the band rectangle, and only those drawing elements that show up in the band are rendered. Your `print` method need not be aware of that process, with one caveat: It should *not* interfere with the clip area.

Caution

The `Graphics` object that your `print` method gets is also clipped against the page margins. If you replace the clip area, you can draw outside the margins. Especially in a printer graphics context, the clipping area must be respected. Call `clip`, not `setClip`, to further restrict the clipping area. If you must remove a clip area, make sure to call `getClip` at the beginning of your `print` method and restore that clip area.

http://www.rejoiceblog.com/

The `PageFormat` parameter of the `print` method contains information about the printed page. The methods `getWidth` and `getHeight` return the paper size, measured in *points*. One point is 1/72 of an inch. (An inch equals 25.4 millimeters.) For example, A4 paper is approximately 595 × 842 points, and US Letter paper is 612 × 792 points.

Points are a common measurement in the printing trade in the United States. Much to the chagrin of the rest of the world, the printing package uses point units. There are two purposes for that: paper sizes and paper margins are measured in points, and points are the default unit for all print graphics contexts. You can verify that in the example program at the end of this section. The program prints two lines of text that are 72 units apart. Run the example program and measure the distance between the baselines; they are exactly 1 inch or 25.4 millimeters apart.

The `getWidth` and `getHeight` methods of the `PageFormat` class give you the complete paper size. Not all of the paper area is printable. Users typically select margins, and even if they don't, printers need to somehow grip the sheets of paper on which they print and therefore have a small unprintable area around the edges.

The methods `getImageableWidth` and `getImageableHeight` tell you the dimensions of the area that you can actually fill. However, the margins need not be symmetrical, so you must also know the top left corner of the imageable area (see Figure 11.61), which you obtain by the methods `getImageableX` and `getImageableY`.

http://www.rejoiceblog.com/

Figure 11.61 Page format measurements

Tip

The graphics context that you receive in the `print` method is clipped to exclude the margins, but the origin of the coordinate system is nevertheless the top left corner of the paper. It makes sense to translate the coordinate system to the top left corner of the imageable area. Simply start your `print` method with

```
g.translate(pageFormat.getImageableX(), pageFormat.getImageableY()
```

If you want your users to choose the settings for the page margins or to switch between portrait and landscape orientation without setting other printing attributes, call the `pageDialog` method of the `PrinterJob` class:

http://www.rejoiceblog.com/

```
PageFormat format = job.pageDialog(attributes);
```

Note

One of the tabs of the print dialog box contains the page setup dialog (see Figure 11.62). You might still want to give users an option to set the page format before printing, especially if your program presents a "what you see is what you get" display of the pages to be printed. The `pageDialog` method returns a `PageFormat` object with the user settings.

Figure 11.62 A cross-platform page setup dialog

The program in Listings 11.23 and 11.24 shows how to render the same set of shapes on the screen and on the printed page. A subclass of `JPanel` implements the `Printable` interface. Both the `paintComponent` and the `print` methods call the same method to carry out the actual drawing.

http://www.rejoiceblog.com/

```
class PrintPanel extends JPanel implements Printable

{

   public void paintComponent(Graphics g)

   {

      super.paintComponent(g);

      var g2 = (Graphics2D) g;

      drawPage(g2);

   }



   public int print(Graphics g, PageFormat pf, int page)

     throws PrinterException

   {

     if (page >= 1) return Printable.NO_SUCH_PAGE;

     var g2 = (Graphics2D) g;

     g2.translate(pf.getImageableX(), pf.getImageableY());

     drawPage(g2);

     return Printable.PAGE_EXISTS;

   }



   public void drawPage(Graphics2D g2)
```

http://www.rejoiceblog.com/

```
    {

      // shared drawing code goes here

      . . .

    }

    . . .

  }
```

This example displays and prints the image shown in Figure 11.50—namely, the outline of the message “Hello, World” used as a clipping area for a pattern of lines.

Click the Print button to start printing, or click the Page setup button to open the page setup dialog box. Listing 11.23 shows the code.

Note

To show a native page setup dialog box, pass a default `PageFormat` object to the `pageDialog` method. The method clones that object, modifies it according to the user selections in the dialog box, and returns the cloned object.

```
PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat = printJob.pageDialog(defaultFormat);
```

Listing 11.23 `print/PrintTestFrame.java`

```
package print;

```

http://www.rejoiceblog.com/

```
 3  import java.awt.*;
 4  import java.awt.print.*;
 5
 6  import javax.print.attribute.*;
 7  import javax.swing.*;
 8
 9  /**
10   * This frame shows a panel with 2D graphics and buttons
11   * page format.
12   */
13  public class PrintTestFrame extends JFrame
14  {
15     private PrintComponent canvas;
16     private PrintRequestAttributeSet attributes;
17
18     public PrintTestFrame()
19     {
20        canvas = new PrintComponent();
21        add(canvas, BorderLayout.CENTER);
```

http://www.rejoiceblog.com/

```
22
23        attributes = new HashPrintRequestAttributeSet();
24
25        var buttonPanel = new JPanel();
26        var printButton = new JButton("Print");
27        buttonPanel.add(printButton);
28        printButton.addActionListener(event ->
29            {
30              try
31              {
32                PrinterJob job = PrinterJob.getPrinterJob();
33                job.setPrintable(canvas);
34                if (job.printDialog(attributes)) job.print(a
35              }
36              catch (PrinterException ex)
37              {
38                JOptionPane.showMessageDialog(PrintTestFrame
39              }
40            });
41
```

http://www.rejoiceblog.com/

```
      var pageSetupButton = new JButton("Page setup");
      buttonPanel.add(pageSetupButton);
      pageSetupButton.addActionListener(event ->
         {
            PrinterJob job = PrinterJob.getPrinterJob();
            job.pageDialog(attributes);
         });

      add(buttonPanel, BorderLayout.NORTH);
      pack();
   }
}
```

Listing 11.24 `print/PrintComponent.java`

```
package print;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
```

http://www.rejoiceblog.com/

```
 6  import java.awt.print.*;
 7  import javax.swing.*;
 8
 9  /**
10   * This component generates a 2D graphics image for scree
11   */
12  public class PrintComponent extends JComponent implements
13  {
14     private static final Dimension PREFERRED_SIZE = new Di
15     public void paintComponent(Graphics g)
16     {
17       var g2 = (Graphics2D) g;
18       drawPage(g2);
19     }
20
21     public int print(Graphics g, PageFormat pf, int page)
22     {
23       if (page >= 1) return Printable.NO_SUCH_PAGE;
24       var g2 = (Graphics2D) g;
25       g2.translate(pf.getImageableX(), pf.getImageableY())
```

http://www.rejoiceblog.com/

```
26        g2.draw(new Rectangle2D.Double(0, 0, pf.getImageable
27
28        drawPage(g2);
29        return Printable.PAGE_EXISTS;
30      }
31
32      /**
33       * This method draws the page both on the screen and t
34       * @param g2 the graphics context
35       */
36      public void drawPage(Graphics2D g2)
37      {
38        FontRenderContext context = g2.getFontRenderContext(
39        var f = new Font("Serif", Font.PLAIN, 72);
40        var clipShape = new GeneralPath();
41
42        var layout = new TextLayout("Hello", f, context);
43        AffineTransform transform = AffineTransform.getTrans
44        Shape outline = layout.getOutline(transform);
```

http://www.rejoiceblog.com/

```
      clipShape.append(outline, false);

      layout = new TextLayout("World", f, context);
      transform = AffineTransform.getTranslateInstance(0,
      outline = layout.getOutline(transform);
      clipShape.append(outline, false);

      g2.draw(clipShape);
      g2.clip(clipShape);

      final int NLINES = 50;
      var p = new Point2D.Double(0, 0);
      for (int i = 0; i < NLINES; i++)
      {
        double x = (2 * getWidth() * i) / NLINES;
        double y = (2 * getHeight() * (NLINES - 1 - i)) /
        var q = new Point2D.Double(x, y);
        g2.draw(new Line2D.Double(p, q));
      }
    }
```

http://www.rejoiceblog.com/

```
    public Dimension getPreferredSize() { return PREFERRED
}
```

java.awt.print.Printable 1.2

- `int print(Graphics g, PageFormat format, int pageNumber)`

renders a page and returns `PAGE_EXISTS`, or returns `NO_SUCH_PAGE`.

Parameters:	`g`	The graphics context onto which the page is rendered
	`format`	The format of the page to draw on
	`pageNumber`	The number of the requested page

java.awt.print.PrinterJob 1.2

- `static PrinterJob getPrinterJob()`

returns a printer job object.

- `PageFormat defaultPage()`

returns the default page format for this printer.

- `boolean printDialog(PrintRequestAttributeSet attributes)`

- `boolean printDialog()`

opens a print dialog box to allow a user to select the pages to be printed and to change print settings. The first method displays a cross-platform dialog box, the second a native dialog box. The first method modifies the `attributes` object to reflect the user settings. Both methods return `true` if the user accepts the dialog box.

http://www.rejoiceblog.com/

- `PageFormat pageDialog(PrintRequestAttributeSet attributes)`

- `PageFormat pageDialog(PageFormat defaults)`

displays a page setup dialog box. The first method displays a cross-platform dialog box, the second a native dialog box. Both methods return a `PageFormat` object with the format that the user requested in the dialog box. The first method modifies the `attributes` object to reflect the user settings. The second method does not modify the `defaults` object.

- `void setPrintable(Printable p)`

- `void setPrintable(Printable p, PageFormat format)`

sets the `Printable` of this print job and an optional page format.

- `void print()`

- `void print(PrintRequestAttributeSet attributes)`

prints the current `Printable` by repeatedly calling its `print` method and sending the rendered pages to the printer, until no more pages are available.

java.awt.print.PageFormat 1.2

- `double getWidth()`

- `double getHeight()`

returns the width and height of the page.

- `double getImageableWidth()`

- `double getImageableHeight()`

returns the width and height of the imageable area of the page.

- `double getImageableX()`

- `double getImageableY()`

http://www.rejoiceblog.com/

returns the position of the top left corner of the imageable area.

- `int getOrientation()`

returns one of `PORTRAIT`, `LANDSCAPE`, or `REVERSE_LANDSCAPE`. Page orientation is transparent to programmers because the page format and graphics context settings automatically reflect the page orientation.

11.5.2 Multiple-Page Printing

In practice, you usually shouldn't pass a raw `Printable` object to a print job. Instead, you should obtain an object of a class that implements the `Pageable` interface. The Java platform supplies one such class, called `Book`. A book is made up of sections, each of which is a `Printable` object. To make a book, add `Printable` objects and their page counts.

```
var book = new Book();

Printable coverPage = . . .;

Printable bodyPages = . . .;

book.append(coverPage, pageFormat); // append 1 page

book.append(bodyPages, pageFormat, pageCount);
```

Then, use the `setPageable` method to pass the `Book` object to the print job.

```
printJob.setPageable(book);
```

Now the print job knows exactly how many pages to print, so the print dialog box displays an accurate page range and the user can select the entire range or subranges.

Caution

http://www.rejoiceblog.com/

When the print job calls the `print` methods of the `Printable` sections, it passes the current page number of the *book*, and not of each *section*, as the current page number. That is a huge pain—each section must know the page counts of the preceding sections to make sense of the page number parameter.

From your perspective as a programmer, the biggest challenge of using the `Book` class is that you must know how many pages each section will have when you print it. Your `Printable` class needs a *layout algorithm* that computes the layout of the material on the printed pages. Before printing starts, invoke that algorithm to compute the page breaks and the page count. You can retain the layout information so you have it handy during the printing process.

You must guard against the possibility that the user has changed the page format. If that happens, you must recompute the layout, even if the information that you want to print has not changed.

Listing 11.26 shows how to produce a multipage printout. This program prints a message in very large characters on a number of pages (see Figure 11.63). You can then trim the margins and tape the pages together to form a banner.

Figure 11.63 A banner

The `layoutPages` method of the `Banner` class computes the layout. We first lay out the message string in a 72-point font. We then compute the height of the resulting string and compare it with the imageable height of the page. We derive a scale factor from these two measurements. When printing the string, we magnify it by that scale factor.

http://www.rejoiceblog.com/

Caution

To lay out your information precisely, you usually need access to the printer graphics context. Unfortunately, there is no way to obtain that graphics context before printing actually starts. In our example program, we make do with the screen graphics context and hope that the font metrics of the screen and printer match.

The `getPageCount` method of the `Banner` class first calls the layout method. Then it scales up the width of the string and divides it by the imageable width of each page. The quotient, rounded up to the next integer, is the page count.

It sounds like it might be difficult to print the banner because characters can be broken across multiple pages. However, thanks to the power of the Java 2D API, this turns out not to be a problem at all. When a particular page is requested, we simply use the `translate` method of the `Graphics2D` class to shift the top left corner of the string to the left. Then, we set a clip rectangle that equals the current page (see Figure 11.64). Finally, we scale the graphics context with the scale factor that the layout method computed.

Figure 11.64 Printing a page of a banner

This example shows the power of transformations. The drawing code is kept simple, and the transformation does all the work of placing the drawing in the appropriate

http://www.rejoiceblog.com/
place. Finally, the clip cuts away the part of the image that falls outside the page. This program shows another compelling use of transformations—to display a print preview.

Listing 11.25 `book/BookTestFrame.java`

```
package book;

import java.awt.*;
import java.awt.print.*;

import javax.print.attribute.*;
import javax.swing.*;

/**
 * This frame has a text field for the banner text and bu
 * preview.
 */
public class BookTestFrame extends JFrame
{
   private JTextField text;
   private PageFormat pageFormat;
   private PrintRequestAttributeSet attributes;
```

http://www.rejoiceblog.com/

```
18
19      public BookTestFrame()
20      {
21        text = new JTextField();
22        add(text, BorderLayout.NORTH);
23
24        attributes = new HashPrintRequestAttributeSet();
25
26        var buttonPanel = new JPanel();
27
28        var printButton = new JButton("Print");
29        buttonPanel.add(printButton);
30        printButton.addActionListener(event ->
31          {
32            try
33            {
34              PrinterJob job = PrinterJob.getPrinterJob();
35              job.setPageable(makeBook());
36              if (job.printDialog(attributes))
37              {
```

http://www.rejoiceblog.com/

```
38                job.print(attributes);
39              }
40            }
41            catch (PrinterException e)
42            {
43              JOptionPane.showMessageDialog(BookTestFrame.th
44            }
45          });
46
47      var pageSetupButton = new JButton("Page setup");
48      buttonPanel.add(pageSetupButton);
49      pageSetupButton.addActionListener(event ->
50        {
51          PrinterJob job = PrinterJob.getPrinterJob();
52          pageFormat = job.pageDialog(attributes);
53        });
54
55      var printPreviewButton = new JButton("Print preview");
56      buttonPanel.add(printPreviewButton);
```

http://www.rejoiceblog.com/

```
    printPreviewButton.addActionListener(event ->
      {
        var dialog = new PrintPreviewDialog(makeBook());
        dialog.setVisible(true);
      });

    add(buttonPanel, BorderLayout.SOUTH);
    pack();
  }

/**
 * Makes a book that contains a cover page and the pages
 */
public Book makeBook()
{
  if (pageFormat == null)
  {
    PrinterJob job = PrinterJob.getPrinterJob();
    pageFormat = job.defaultPage();
  }
```

http://www.rejoiceblog.com/

```
    var book = new Book();
    String message = text.getText();
    var banner = new Banner(message);
    int pageCount = banner.getPageCount((Graphics2D) getGra
    book.append(new CoverPage(message + " (" + pageCount +
    book.append(banner, pageFormat, pageCount);
    return book;
  }
}
```

Listing 11.26 `book/Banner.java`

```
package book;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.print.*;

/**
```

http://www.rejoiceblog.com/

```
 * A banner that prints a text string on multiple pages.
 */
public class Banner implements Printable
{
   private String message;
   private double scale;

   /**
    * Constructs a banner.
    * @param m the message string
    */
   public Banner(String m)
   {
      message = m;
   }

   /**
    * Gets the page count of this section.
    * @param g2 the graphics context
    * @param pf the page format
```

http://www.rejoiceblog.com/

```
29       * @return the number of pages needed
30       */
31      public int getPageCount(Graphics2D g2, PageFormat pf)
32      {
33         if (message.equals("")) return 0;
34         FontRenderContext context = g2.getFontRenderContex
35         var f = new Font("Serif", Font.PLAIN, 72);
36         Rectangle2D bounds = f.getStringBounds(message, co
37         scale = pf.getImageableHeight() / bounds.getHeight
38         double width = scale * bounds.getWidth();
39         int pages = (int) Math.ceil(width / pf.getImageabl
40         return pages;
41      }
42
43      public int print(Graphics g, PageFormat pf, int page)
44      {
45         var g2 = (Graphics2D) g;
46         if (page > getPageCount(g2, pf)) return Printable.
47         g2.translate(pf.getImageableX(), pf.getImageableY(
```

http://www.rejoiceblog.com/

```
        drawPage(g2, pf, page);
        return Printable.PAGE_EXISTS;
    }

    public void drawPage(Graphics2D g2, PageFormat pf, in
    {
        if (message.equals("")) return;
        page--; // account for cover page

        drawCropMarks(g2, pf);
        g2.clip(new Rectangle2D.Double(0, 0, pf.getImageab
        g2.translate(-page * pf.getImageableWidth(), 0);
        g2.scale(scale, scale);
        FontRenderContext context = g2.getFontRenderContex
        var f = new Font("Serif", Font.PLAIN, 72);
        var layout = new TextLayout(message, f, context);
        AffineTransform transform = AffineTransform.getTra
        Shape outline = layout.getOutline(transform);
        g2.draw(outline);
```

http://www.rejoiceblog.com/

```
    }

   /**
    * Draws 1/2" crop marks in the corners of the page.
    * @param g2 the graphics context
    * @param pf the page format
    */
   public void drawCropMarks(Graphics2D g2, PageFormat pf
   {
     final double C = 36; // crop mark length = 1/2 inch
     double w = pf.getImageableWidth();
     double h = pf.getImageableHeight();
     g2.draw(new Line2D.Double(0, 0, 0, C));
     g2.draw(new Line2D.Double(0, 0, C, 0));
     g2.draw(new Line2D.Double(w, 0, w, C));
     g2.draw(new Line2D.Double(w, 0, w - C, 0));
     g2.draw(new Line2D.Double(0, h, 0, h - C));
     g2.draw(new Line2D.Double(0, h, C, h));
     g2.draw(new Line2D.Double(w, h, w, h - C));
```

http://www.rejoiceblog.com/

```
      g2.draw(new Line2D.Double(w, h, w - C, h));
    }
  }

  /**
   * This class prints a cover page with a title.
   */
  class CoverPage implements Printable
  {
    private String title;

    /**
     * Constructs a cover page.
     * @param t the title
     */
    public CoverPage(String t)
    {
      title = t;
    }

```

http://www.rejoiceblog.com/

```
    public int print(Graphics g, PageFormat pf, int page)
    {
        if (page >= 1) return Printable.NO_SUCH_PAGE;
        var g2 = (Graphics2D) g;
        g2.setPaint(Color.black);
        g2.translate(pf.getImageableX(), pf.getImageableY(
        FontRenderContext context = g2.getFontRenderContex
        Font f = g2.getFont();
        var layout = new TextLayout(title, f, context);
        float ascent = layout.getAscent();
        g2.drawString(title, 0, ascent);
        return Printable.PAGE_EXISTS;
    }
}
```

Listing 11.27 book/PrintPreviewDialog.java

```
package book;

import java.awt.*;
```

http://www.rejoiceblog.com/

```
 4  import java.awt.print.*;
 5
 6  import javax.swing.*;
 7
 8  /**
 9   * This class implements a generic print preview dialog.
10   */
11  public class PrintPreviewDialog extends JDialog
12  {
13     private static final int DEFAULT_WIDTH = 300;
14     private static final int DEFAULT_HEIGHT = 300;
15
16     private PrintPreviewCanvas canvas;
17
18     /**
19      * Constructs a print preview dialog.
20      * @param p a Printable
21      * @param pf the page format
22      * @param pages the number of pages in p
23      */
```

http://www.rejoiceblog.com/

```
24      public PrintPreviewDialog(Printable p, PageFormat pf,
25      {
26        var book = new Book();
27        book.append(p, pf, pages);
28        layoutUI(book);
29      }
30
31      /**
32       * Constructs a print preview dialog.
33       * @param b a Book
34       */
35      public PrintPreviewDialog(Book b)
36      {
37        layoutUI(b);
38      }
39
40      /**
41       * Lays out the UI of the dialog.
42       * @param book the book to be previewed
```

http://www.rejoiceblog.com/

```
43      */
44     public void layoutUI(Book book)
45     {
46       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
47
48       canvas = new PrintPreviewCanvas(book);
49       add(canvas, BorderLayout.CENTER);
50
51       var buttonPanel = new JPanel();
52
53       var nextButton = new JButton("Next");
54       buttonPanel.add(nextButton);
55       nextButton.addActionListener(event -> canvas.flipPag
56
57       var previousButton = new JButton("Previous");
58       buttonPanel.add(previousButton);
59       previousButton.addActionListener(event -> canvas.fli
60
61       var closeButton = new JButton("Close");
62       buttonPanel.add(closeButton);
```

http://www.rejoiceblog.com/

```
        closeButton.addActionListener(event -> setVisible(fa

        add(buttonPanel, BorderLayout.SOUTH);
    }
}
```

Listing 11.28 `book/PrintPreviewCanvas.java`

```
package book;

import java.awt.*;
import java.awt.geom.*;
import java.awt.print.*;
import javax.swing.*;

/**
 * The canvas for displaying the print preview.
 */
class PrintPreviewCanvas extends JComponent
{
```

http://www.rejoiceblog.com/

```
13      private Book book;
14      private int currentPage;
15
16      /**
17       * Constructs a print preview canvas.
18       * @param b the book to be previewed
19       */
20      public PrintPreviewCanvas(Book b)
21      {
22        book = b;
23        currentPage = 0;
24      }
25
26      public void paintComponent(Graphics g)
27      {
28        var g2 = (Graphics2D) g;
29        PageFormat pageFormat = book.getPageFormat(currentPa
30
31        double xoff; // x offset of page start in window
32        double yoff; // y offset of page start in window
```

http://www.rejoiceblog.com/

```
33        double scale; // scale factor to fit page in window
34        double px = pageFormat.getWidth();
35        double py = pageFormat.getHeight();
36        double sx = getWidth() - 1;
37        double sy = getHeight() - 1;
38        if (px / py < sx / sy) // center horizontally
39        {
40          scale = sy / py;
41          xoff = 0.5 * (sx - scale * px);
42          yoff = 0;
43        }
44        else
45        // center vertically
46        {
47          scale = sx / px;
48          xoff = 0;
49          yoff = 0.5 * (sy - scale * py);
50        }
51        g2.translate((float) xoff, (float) yoff);
```

http://www.rejoiceblog.com/

```
52        g2.scale((float) scale, (float) scale);
53
54        // draw page outline (ignoring margins)
55        var page = new Rectangle2D.Double(0, 0, px, py);
56        g2.setPaint(Color.white);
57        g2.fill(page);
58        g2.setPaint(Color.black);
59        g2.draw(page);
60
61        Printable printable = book.getPrintable(currentPage)
62        try
63        {
64          printable.print(g2, pageFormat, currentPage);
65        }
66        catch (PrinterException e)
67        {
68          g2.draw(new Line2D.Double(0, 0, px, py));
69          g2.draw(new Line2D.Double(px, 0, 0, py));
70        }
71      }
```

http://www.rejoiceblog.com/

```

    /**
     * Flip the book by the given number of pages.
     * @param by the number of pages to flip by. Negative
     */
    public void flipPage(int by)
    {
      int newPage = currentPage + by;
      if (0 <= newPage && newPage < book.getNumberOfPages(
      {
        currentPage = newPage;
        repaint();
      }
    }
}
```

11.5.3 Print Services

So far, you have seen how to print 2D graphics. However, the printing API introduced in Java SE 1.4 affords far greater flexibility. The API defines a number of data types and lets you find print services that are able to print them. Among the data types are

- Images in GIF, JPEG, or PNG format

http://www.rejoiceblog.com/

- Documents in text, HTML, PostScript, or PDF format
- Raw printer code data
- Objects of a class that implements `Printable`, `Pageable`, or `RenderableImage`

The data themselves can be stored in a source of bytes or characters such as an input stream, a URL, or an array. A *document flavor* describes the combination of a data source and a data type. The `DocFlavor` class defines a number of inner classes for the various data sources. Each of the inner classes defines constants to specify the flavors. For example, the constant

```
DocFlavor.INPUT_STREAM.GIF
```

describes a GIF image that is read from an input stream. Table 11.4 lists the combinations.

Table 11.4 Document Flavors for Print Services

http://www.rejoiceblog.com/

Data Source	Data Type	MIME Type
INPUT_STREAM	GIF	image/gif
URL	JPEG	image/jpeg
BYTE_ARRAY	PNG	image/png
	POSTSCRIPT	application/postscript
	PDF	application/pdf
	TEXT_HML_HOST	text/html (using host encoding)
	TEXT_HTML_US_ASCII	text/html; charset=us-ascii
	TEXT_HTML_UTF_8	text/html; charset=utf-8
	TEXT_HTML_UTF_16	text/html; charset=utf-16
	TEXT_HTML_UTF_16LE	text/html; charset=utf-16le (little-endian)
	TEXT_HTML_UTF_16BE	text/html; charset=utf-16be (big-endian)
	TEXT_PLAIN_HOST	text/plain (using host encoding)
	TEXT_PLAIN_US_ASCII	text/plain; charset=us-ascii
	TEXT_PLAIN_UTF_8	text/plain; charset=utf-8
	TEXT_PLAIN_UTF_16	text/plain; charset=utf-16
	TEXT_PLAIN_UTF_16LE	text/plain; charset=utf-16le (little-endian)
	TEXT_PLAIN_UTF_16BE	text/plain; charset=utf-16be (big-endian)
	PCL	application/vnd.hp-PCL (Hewlett Packard Printer Control Language)
	AUTOSENSE	application/octet-stream (raw printer data)
READER	TEXT_HTML	text/html; charset=utf-16
STRING	TEXT_PLAIN	text/plain; charset=utf-16
CHAR_ARRAY		
SERVICE_FORMATTED	PRINTABLE	N/A
	PAGEABLE	N/A
	RENDERABLE_IMAGE	N/A

Suppose you want to print a GIF image located in a file. First, find out whether there is a *print service* that is capable of handling the task. The static `lookupPrintServices` method of the `PrintServiceLookup` class returns an array of `PrintService` objects that can handle the given document flavor.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
```

http://www.rejoiceblog.com/

```
PrintService[] services = PrintServiceLookup.lookupPrintServi
```

The second parameter of the `lookupPrintServices` method is `null` to indicate that we don't want to constrain the search by specifying printer attributes. We'll cover attributes in the next section.

If the lookup yields an array with more than one element, select from the listed print services. You can call the `getName` method of the `PrintService` class to get the printer names and let the user choose.

Next, get a document print job from the service:

```
DocPrintJob job = services[i].createPrintJob();
```

For printing, you need an object that implements the `Doc` interface. The Java library supplies a class `SimpleDoc` for that purpose. The `SimpleDoc` constructor requires the data source object, the document flavor, and an optional attribute set. For example,

```
var in = new FileInputStream(fileName);

var doc = new SimpleDoc(in, flavor, null);
```

Finally, you are ready to print:

```
job.print(doc, null);
```

As before, the `null` parameter can be replaced by an attribute set.

Note that this printing process is quite different from that of the preceding section. There is no user interaction through print dialog boxes. For example, you can implement a server-side printing mechanism in which users submit print jobs through a web form.

javax.print.PrintServiceLookup 1.4

http://www.rejoiceblog.com/

- `PrintService[] lookupPrintServices(DocFlavor flavor, AttributeSet attributes)`

looks up the print services that can handle the given document flavor and attributes.

Parameters:	`flavor`	The document flavor
	`attributes`	The required printing attributes, or `null` if attributes should not be considered

javax.print.PrintService 1.4

- `DocPrintJob createPrintJob()`

creates a print job for printing an object of a class that implements the `Doc` interface, such as a `SimpleDoc`.

javax.print.DocPrintJob 1.4

- `void print(Doc doc, PrintRequestAttributeSet attributes)`

prints the given document with the given attributes.

Parameters:	`doc`	The `Doc` to be printed
	`attributes`	The required printing attributes, or `null` if no printing attributes are required

javax.print.SimpleDoc 1.4

- `SimpleDoc(Object data, DocFlavor flavor, DocAttributeSet attributes)`

constructs a `SimpleDoc` object that can be printed with a `DocPrintJob`.

http://www.rejoiceblog.com/

Parameters:	data	The object with the print data, such as an input stream or a Printable
	flavor	The document flavor of the print data
	attributes	Document attributes, or null if attributes are not required

11.5.4 Stream Print Services

A print service sends print data to a printer. A stream print service generates the same print data but instead sends them to a stream, perhaps for delayed printing or because the print data format can be interpreted by other programs. In particular, if the print data format is PostScript, it may be useful to save the print data to a file because many programs can process PostScript files. The Java platform includes a stream print service that can produce PostScript output from images and 2D graphics. You can use that service on all systems, even if there are no local printers.

Enumerating stream print services is a bit more tedious than locating regular print services. You need both the `DocFlavor` of the object to be printed and the MIME type of the stream output. You then get a `StreamPrintServiceFactory` array of factories.

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;

String mimeType = "application/postscript";

StreamPrintServiceFactory[] factories

    = StreamPrintServiceFactory.lookupStreamPrintServiceFacto
```

The `StreamPrintServiceFactory` class has no methods that would help us distinguish any one factory from another, so we just take `factories[0]`. We call the `getPrintService` method with an output stream parameter to get a `StreamPrintService` object.

```
var out = new FileOutputStream(fileName);

StreamPrintService service = factories[0].getPrintService(out
```

http://www.rejoiceblog.com/

The `StreamPrintService` class is a subclass of `PrintService`. To produce a printout, simply follow the steps of the preceding section.

javax.print.StreamPrintServiceFactory 1.4

- `StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor flavor, String mimeType)`

looks up the stream print service factories that can print the given document flavor and produce an output stream of the given MIME type.

- `StreamPrintService getPrintService(OutputStream out)`

gets a print service that sends the printing output to the given output stream.

The program in Listing 11.29 demonstrates how to use a stream print service to print Java 2D shapes to a PostScript file. You can replace the sample drawing code with code that generates any Java 2D shapes and have the shapes converted to PostScript. Then you can easily convert the result to PDF or EPS, using an external tool. (Unfortunately, Java does not support printing to PDF directly.)

Note

In this example, we call a `draw` method that draws Java 2D shapes onto a `Graphics2D` object. If you want to draw the surface of a component (for example, a table or tree), use the following code:

```
private static int IMAGE_WIDTH = component.getWidth();

private static int IMAGE_HEIGHT = component.getHeight();

public static void draw(Graphics2D g2) { component.paint(g2)};
```

http://www.rejoiceblog.com/

Listing 11.29 `printService/PrintServiceTest.java`

```
package printService;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.print.*;
import java.io.*;
import javax.print.*;
import javax.print.attribute.*;

/**
 * This program demonstrates the use of stream print serv
 * Java 2D shapes to a PostScript file. If you don't supp
 * line, the output is saved to out.ps.
 * @version 1.0 2018-06-01
 * @author Cay Horstmann
 */
public class PrintServiceTest
```

http://www.rejoiceblog.com/

```
{
    // Set your image dimensions here
    private static int IMAGE_WIDTH = 300;
    private static int IMAGE_HEIGHT = 300;

    public static void draw(Graphics2D g2) {
      // Your drawing instructions go here
      FontRenderContext context = g2.getFontRenderContext(
      var f = new Font("Serif", Font.PLAIN, 72);
      var clipShape = new GeneralPath();

      var layout = new TextLayout("Hello", f, context);
      AffineTransform transform = AffineTransform.getTrans
      Shape outline = layout.getOutline(transform);
      clipShape.append(outline, false);

      layout = new TextLayout("World", f, context);
      transform = AffineTransform.getTranslateInstance(0,
      outline = layout.getOutline(transform);
      clipShape.append(outline, false);
```

http://www.rejoiceblog.com/

```
39
40        g2.draw(clipShape);
41        g2.clip(clipShape);
42
43        final int NLINES = 50;
44        var p = new Point2D.Double(0, 0);
45        for (int i = 0; i < NLINES; i++)
46        {
47          double x = (2 * IMAGE_WIDTH * i) / NLINES;
48          double y = (2 * IMAGE_HEIGHT * (NLINES - 1 - i)) /
49          var q = new Point2D.Double(x, y);
50          g2.draw(new Line2D.Double(p, q));
51        }
52      }
53
54      public static void main(String[] args) throws IOExcept
55      {
56        String fileName = args.length > 0 ? args[0] : "out.p
57        DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINT
```

http://www.rejoiceblog.com/

```
58        var mimeType = "application/postscript";
59        StreamPrintServiceFactory[] factories
60           = StreamPrintServiceFactory.lookupStreamPrintServ
61        var out = new FileOutputStream(fileName);
62        if (factories.length > 0) {
63          PrintService service = factories[0].getPrintServic
64          var doc = new SimpleDoc(new Printable() {
65            public int print(Graphics g, PageFormat pf, int
66              if (page >= 1) return Printable.NO_SUCH_PAGE;
67              else {
68                double sf1 = pf.getImageableWidth() / (IMAGE
69                double sf2 = pf.getImageableHeight() / (IMAG
70                double s = Math.min(sf1, sf2);
71                var g2 = (Graphics2D) g;
72                g2.translate((pf.getWidth() - pf.getImageabl
73                  (pf.getHeight() - pf.getImageableHeight())
74                g2.scale(s, s);
75
76                draw(g2);
77                return Printable.PAGE_EXISTS;
```

http://www.rejoiceblog.com/

```
            }
          }
        }, flavor, null);
      DocPrintJob job = service.createPrintJob();
      var attributes = new HashPrintRequestAttributeSet();
      job.print(doc, attributes);
    }
    else
      System.out.println("No factories for " + mimeType);
  }
}
```

11.5.5 Printing Attributes

The print service API contains a complex set of interfaces and classes to specify various kinds of attributes. There are four important groups of attributes. The first two specify requests to the printer.

• *Print request attributes* request particular features for all `doc` objects in a print job, such as two-sided printing or the paper size.

• *Doc attributes* are request attributes that apply only to a single `doc` object.

The other two attributes contain information about the printer and job status.

• *Print service attributes* give information about the print service, such as the printer make and model or whether the printer is currently accepting jobs.

http://www.rejoiceblog.com/

• *Print job attributes* give information about the status of a particular print job, such as whether the job is already completed.

To describe the various attributes, there is an interface `Attribute` with subinter-faces:

```
PrintRequestAttribute

DocAttribute

PrintServiceAttribute

PrintJobAttribute

SupportedValuesAttribute
```

Individual attribute classes implement one or more of these interfaces. For example, objects of the `Copies` class describe the number of copies of a printout. That class implements both the `PrintRequestAttribute` and the `PrintJobAttribute` interfaces. Clearly, a print request can contain a request for multiple copies. Conversely, an attribute of the print job might be how many of these copies were actually printed. That number might be lower, perhaps because of printer limitations or because the printer ran out of paper.

The `SupportedValuesAttribute` interface indicates that an attribute value does not reflect actual request or status data but rather the capability of a service. For example, the `CopiesSupported` class implements the `SupportedValuesAttribute` interface.

An object of that class might describe that a printer supports 1 through 99 copies of a printout.

Figure 11.65 shows a class diagram of the attribute hierarchy.

http://www.rejoiceblog.com/

Figure 11.65 The attribute hierarchy

In addition to the interfaces and classes for individual attributes, the print service API defines interfaces and classes for attribute sets. A superinterface, `AttributeSet`, has four subinterfaces:

```
PrintRequestAttributeSet

DocAttributeSet

PrintServiceAttributeSet

PrintJobAttributeSet
```

Each of these interfaces has an implementing class, yielding the five classes:

```
HashAttributeSet

HashPrintRequestAttributeSet

HashDocAttributeSet

HashPrintServiceAttributeSet
```

http://www.rejoiceblog.com/

```
HashPrintJobAttributeSet
```

Figure 11.66 shows a class diagram of the attribute set hierarchy.

Figure 11.66 The attribute set hierarchy

For example, you can construct a print request attribute set like this:

```
var attributes = new HashPrintRequestAttributeSet();
```

After constructing the set, you are freed from worrying about the `Hash` prefix.

Why have all these interfaces? They make it possible to check for correct attribute usage. For example, a `DocAttributeSet` accepts only objects that implement the `DocAttribute` interface. Any attempt to add another attribute results in a runtime error.

An attribute set is a specialized kind of map where the keys are of type `Class` and the values belong to a class that implements the `Attribute` interface. For example, if you insert an object

http://www.rejoiceblog.com/

```
new Copies(10)
```

into an attribute set, then its key is the `Class` object `Copies.class`. That key is called the *category* of the attribute. The `Attribute` interface declares a method

```
Class getCategory()
```

that returns the category of an attribute. The `Copies` class defines the method to return the object `Copies.class`, but it isn't a requirement that the category be the same as the class of the attribute.

When an attribute is added to an attribute set, the category is extracted automatically. Just add the attribute value:

```
attributes.add(new Copies(10));
```

If you subsequently add another attribute with the same category, it overwrites the first one.

To retrieve an attribute, you need to use the category as the key, for example:

```
AttributeSet attributes = job.getAttributes();

var copies = (Copies) attribute.get(Copies.class);
```

Finally, attributes are organized by the values they can have. The `Copies` at-tribute can have any integer value. The `Copies` class extends the `IntegerSyntax` class that takes care of all integer-valued attributes. The `getValue` method returns the integer value of the attribute, for example:

```
int n = copies.getValue();
```

The classes

http://www.rejoiceblog.com/

```
TextSyntax

DateTimeSyntax

URISyntax
```

encapsulate a string, a date and time value, or a URI.

Finally, many attributes can take a finite number of values. For example, the `PrintQuality` attribute has three settings: draft, normal, and high. They are represented by three constants:

```
PrintQuality.DRAFT

PrintQuality.NORMAL

PrintQuality.HIGH
```

Attribute classes with a finite number of values extend the `EnumSyntax` class, which provides a number of convenience methods to set up these enumerations in a typesafe manner. You need not worry about the mechanism when using such an attribute. Simply add the named values to attribute sets:

```
attributes.add(PrintQuality.HIGH);
```

Here is how you check the value of an attribute:

```
if (attributes.get(PrintQuality.class) == PrintQuality.HIGH)

   . . .
```

Table 11.5 lists the printing attributes. The second column lists the superclass of the attribute class (for example, `IntegerSyntax` for the `Copies` attribute) or the set of enumeration values for the attributes with a finite set of values. The last four columns

http://www.rejoiceblog.com/

indicate whether the attribute class implements the `DocAttribute` (DA), `PrintJobAttribute` (PJA), `PrintRequestAttribute` (PRA), and `PrintServiceAttribute` (PSA) interfaces.

Table 11.5 Printing Attributes

Attribute	Superclass or Enumeration Constants	DA	PJA	PRA	PSA
Chromaticity	MONOCHROME, COLOR	✓	✓	✓	
ColorSupported	SUPPORTED, NOT_SUPPORTED				✓
Compression	COMPRESS, DEFLATE, GZIP, NONE	✓			
Copies	IntegerSyntax		✓	✓	
DateTimeAtCompleted	DateTimeSyntax		✓		
DateTimeAtCreation	DateTimeSyntax		✓		
DateTimeAtProcessing	DateTimeSyntax		✓		
Destination	URISyntax		✓	✓	
DocumentName	TextSyntax	✓			
Fidelity	FIDELITY_TRUE, FIDELITY_FALSE		✓	✓	
Finishings	NONE, STAPLE, EDGE_STITCH, BIND, SADDLE_STITCH, COVER, . . .	✓	✓	✓	
JobHoldUntil	DateTimeSyntax		✓	✓	
JobImpressions	IntegerSyntax		✓	✓	
JobImpressionsCompleted	IntegerSyntax		✓		
JobKOctets	IntegerSyntax		✓	✓	
JobKOctetsProcessed	IntegerSyntax		✓		
JobMediaSheets	IntegerSyntax		✓	✓	
JobMediaSheetsCompleted	IntegerSyntax		✓		
JobMessageFromOperator	TextSyntax		✓		
JobName	TextSyntax		✓	✓	
JobOriginatingUserName	TextSyntax		✓		

http://www.rejoiceblog.com/

JobPriority	IntegerSyntax		✓	✓	
JobSheets	STANDARD, NONE		✓	✓	
JobState	ABORTED, CANCELED, COMPLETED, PENDING, PENDING_HELD, PROCESSING, PROCESSING_STOPPED		✓		
JobStateReason	ABORTED_BY_SYSTEM, DOCUMENT_FORMAT_ERROR, many others				
JobStateReasons	HashSet		✓		
MediaName	ISO_A4_WHITE, ISO_A4_TRANSPARENT, NA_LETTER_WHITE, NA_LETTER_TRANSPARENT	✓	✓	✓	
MediaSize	ISO.A0–ISO.A10, ISO.B0–ISO.B10, ISO.C0–ISO.C10, NA.LETTER, NA.LEGAL, various other paper and envelope sizes				
MediaSizeName	ISO_A0–ISO_A10, ISO_B0–ISO_B10, ISO_C0–ISO_C10, NA_LETTER, NA_LEGAL, various other paper and envelope size names	✓	✓	✓	
MediaTray	TOP, MIDDLE, BOTTOM, SIDE, ENVELOPE, LARGE_CAPACITY, MAIN, MANUAL	✓	✓	✓	
MultipleDocumentHandling	SINGLE_DOCUMENT, SINGLE_DOCUMENT_NEW_SHEET, SEPARATE_DOCUMENTS_COLLATED_COPIES, SEPARATE_DOCUMENTS_UNCOLLATED_COPIES		✓	✓	
NumberOfDocuments	IntegerSyntax		✓		
NumberOfInterveningJobs	IntegerSyntax		✓		
NumberUp	IntegerSyntax	✓	✓	✓	
OrientationRequested	PORTRAIT, LANDSCAPE, REVERSE_PORTRAIT, REVERSE_LANDSCAPE	✓	✓	✓	
OutputDeviceAssigned	TextSyntax		✓		
PageRanges	SetOfInteger	✓	✓	✓	
PagesPerMinute	IntegerSyntax				✓
PagesPerMinuteColor	IntegerSyntax				

http://www.rejoiceblog.com/

PDLOverrideSupported	ATTEMPTED, NOT_ATTEMPTED				✓
PresentationDirection	TORIGHT_TOBOTTOM, TORIGHT_TOTOP, TOBOTTOM_TORIGHT, TOBOTTOM_TOLEFT, TOLEFT_TOBOTTOM, TOLEFT_TOTOP, TOTOP_TORIGHT, TOTOP_TOLEFT		✓	✓	
PrinterInfo	TextSyntax				✓
PrinterIsAcceptingJobs	ACCEPTING_JOBS, NOT_ACCEPTING_JOBS				✓
PrinterLocation	TextSyntax				✓
PrinterMakeAndModel	TextSyntax				✓
PrinterMessageFromOperator	TextSyntax				✓
PrinterMoreInfo	URISyntax				✓
PrinterMoreInfoManufacturer	URISyntax				✓
PrinterName	TextSyntax				✓
PrinterResolution	ResolutionSyntax	✓	✓	✓	
PrinterState	PROCESSING, IDLE, STOPPED, UNKNOWN				✓
PrinterStateReason	COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM, and many others				
PrinterStateReasons	HashMap				
PrinterURI	URISyntax				✓
PrintQuality	DRAFT, NORMAL, HIGH	✓	✓	✓	
QueuedJobCount	IntegerSyntax				✓
ReferenceUriSchemesSupported	FILE, FTP, GOPHER, HTTP, HTTPS, NEWS, NNTP, WAIS				
RequestingUserName	TextSyntax			✓	
Severity	ERROR, REPORT, WARNING				
SheetCollate	COLLATED, UNCOLLATED	✓	✓	✓	
Sides	ONE_SIDED, DUPLEX (= TWO_SIDED_LONG_EDGE), TUMBLE (= TWO_SIDED_SHORT_EDGE)	✓	✓	✓	

Note

As you can see, there are lots of attributes, many of which are quite specialized. The source for most of the attributes is the Internet Printing Protocol 1.1 (RFC 2911).

http://www.rejoiceblog.com/

Note

An earlier version of the printing API introduced the `JobAttributes` and `PageAttributes` classes, whose purpose was similar to the printing attributes covered in this section. These classes are now obsolete.

javax.print.attribute.Attribute 1.4

- `Class getCategory()`

gets the category of this attribute.

- `String getName()`

gets the name of this attribute.

javax.print.attribute.AttributeSet 1.4

- `boolean add(Attribute attr)`

adds an attribute to this set. If the set has another attribute with the same category, that attribute is replaced by the given attribute. Returns `true` if the set changed as a result of this operation.

- `Attribute get(Class category)`

retrieves the attribute with the given category key, or `null` if no such attribute exists.

- `boolean remove(Attribute attr)`

- `boolean remove(Class category)`

removes the given attribute, or the attribute with the given category, from the set. Returns `true` if the set changed as a result of this operation.

- `Attribute[] toArray()`

http://www.rejoiceblog.com/

returns an array with all attributes in this set.

javax.print.PrintService 1.4

- `PrintServiceAttributeSet getAttributes()`

gets the attributes of this print service.

javax.print.DocPrintJob 1.4

- `PrintJobAttributeSet getAttributes()`

gets the attributes of this print job.

You have now reached the end of this long chapter covering advanced Swing and AWT features. In the final chapter, we will turn to a different aspect of Java programming: interacting, on the same machine, with “native” code in a different programming language.

http://www.rejoiceblog.com/

Chapter 12 Native Methods

In this chapter

- 12.1 Calling a C Function from a Java Program
- 12.2 Numeric Parameters and Return Values
- 12.3 String Parameters
- 12.4 Accessing Fields
- 12.5 Encoding Signatures
- 12.6 Calling Java Methods
- 12.7 Accessing Array Elements
- 12.8 Handling Errors
- 12.9 Using the Invocation API
- 12.10 A Complete Example: Accessing the Windows Registry

While a “100% Pure Java” solution is nice in principle, there are situations in which you will want to write (or use) code written in another language. (Such code is usually called *native* code.)

Particularly in the early days of Java, many people assumed that it would be a good idea to use C or C++ to speed up critical parts of a Java application. However, in practice, this was rarely useful. A presentation at the 1996 JavaOne conference showed this clearly. The implementors of the cryptography library at Sun Microsystems reported that a pure Java platform implementation of their cryptographic functions was more than adequate. It was true that the code was not as fast as a C implementation would have been, but it turned

http://www.rejoiceblog.com/

out not to matter. The Java platform implementation was far faster than the network I/O. This turned out to be the real bottleneck.

Of course, there are drawbacks to going native. If a part of your application is written in another language, you must supply a separate native library for every platform you want to support. Code written in C or C++ offers no protection against overwriting memory through invalid pointer usage. It is easy to write native methods that corrupt your program or infect the operating system.

Thus, we suggest using native code only when you need to. In particular, there are three reasons why native code might be the right choice:

• Your application requires access to system features or devices that are not accessible through the Java platform.

• You have substantial amounts of tested and debugged code in another language, and you know how to port it to all desired target platforms.

• You have found, through benchmarking, that the Java code is much slower than the equivalent code in another language.

The Java platform has an API for interoperating with native C code called the Java Native Interface (JNI). We'll discuss JNI programming in this chapter.

C++ Note

You can also use C++ instead of C to write native methods. There are a few advantages—type checking is slightly stricter, and accessing the JNI functions is a bit more convenient. However, JNI does not support any mapping between Java and C++ classes.

12.1 CALLING A C FUNCTION FROM A JAVA PROGRAM

Suppose you have a C function that does something you like and, for one reason or another, you don't want to bother reimplementing it in Java. For the sake of illustration, we'll start with a simple C function that prints a greeting.

The Java programming language uses the keyword `native` for a native method, and

http://www.rejoiceblog.com/

you will obviously need to place a method in a class. The result is shown in Listing 12.1.

The `native` keyword alerts the compiler that the method will be defined externally. Of course, native methods will contain no code in the Java programming language, and the method header is followed immediately by a terminating semicolon. Therefore, native method declarations look similar to abstract method declarations.

Listing 12.1 `helloNative/HelloNative.java`

```
/**
  * @version 1.11 2007-10-26
  * @author Cay Horstmann
  */
class HelloNative
{
    public static native void greeting();
}
```

Note

As in the previous chapter, we do not use packages here to keep examples simple.

In this particular example, the native method is also declared as `static`. Native methods can be both static and nonstatic. We'll start with a static method because we do not yet want to deal with parameter passing.

You can actually compile this class, but if you try to use it in a program, the virtual

http://www.rejoiceblog.com/

machine will tell you it doesn't know how to find `greeting`—reporting an `UnsatisfiedLinkError`. To implement the native code, write a corresponding C function. You must name that function *exactly* the way the Java virtual machine expects. Here are the rules:

1. Use the full Java method name, such as `HelloNative.greeting`. If the class is in a package, prepend the package name, such as `com.horstmann.HelloNative.greeting`.

2. Replace every period with an underscore, and append the prefix `Java_`. For example, `Java_HelloNative_greeting` or `Java_com_horstmann_HelloNative_greeting`.

3. If the class name contains characters that are not ASCII letters or digits—that is, `'_'`, `'$'`, or Unicode characters with codes greater than `'\u007F'`—replace them with `_0`*xxxx*, where *xxxx* is the sequence of four hexadecimal digits of the character's Unicode value.

Note

If you *overload* native methods—that is, if you provide multiple native methods with the same name—you must append a double underscore followed by the encoded argument types. (We'll describe the encoding of the argument types later in this chapter.) For example, if you have a native method `greeting` and another native method `greeting(int repeat)`, then the first one is called `Java_HelloNative_greeting__`, and the second, `Java_HelloNative_greeting__I`.

Actually, nobody does this by hand; instead, run `javac` with the `-h` flag. providing the directory in which the header files should be placed:

```
javac -h . HelloNative.java
```

This command creates a header file `HelloNative.h` in the current directory, as shown in Listing 12.2.

http://www.rejoiceblog.com/

Listing 12.2 `helloNative/HelloNative.h`

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloNative */

#ifndef _Included_HelloNative
#define _Included_HelloNative
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      HelloNative
 * Method:     greeting
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloNative_greeting
   (JNIEnv *, jclass);

#ifdef __cplusplus
```

http://www.rejoiceblog.com/

```
}

#endif

#endif
```

As you can see, this file contains the declaration of a function `Java_HelloNative_greeting`. (The macros `JNIEXPORT` and `JNICALL` are defined in the header file `jni.h`. They denote compiler-dependent specifiers for exported functions that come from a dynamically loaded library.)

Now, simply copy the function prototype from the header file into a source file and give the implementation code for the function, as shown in Listing 12.3.

Listing 12.3 `helloNative/HelloNative.c`

```
/*
      @version 1.10 1997-07-01
      @author Cay Horstmann
*/

#include "HelloNative.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv*
{
   printf("Hello Native World!\n");
```

http://www.rejoiceblog.com/

```
}
```

In this simple function, ignore the `env` and `cl` arguments. You'll see their use later.

C++ Note

You can use C++ to implement native methods. However, you must then declare the functions that implement the native methods as `extern "C"`. (This stops the C++ compiler from "mangling" the method name.) For example,

```
extern "C"

JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jcla

  cout << "Hello, Native World!" << endl;

}
```

Compile the native C code into a dynamically loaded library. The details depend on your compiler.

For example, with the GNU C compiler on Linux, use these commands:

```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared -o libHel
```

With the Microsoft compiler under Windows, the command is

```
cl -I jdk\include -I jdk\include\win32 -LD HelloNative.c -FeHel
```

Here, *jdk* is the directory that contains the JDK.

http://www.rejoiceblog.com/

Tip

If you use the Microsoft compiler from a command shell, first run a batch file such as `vsvars32.bat` or `vcvarsall.bat`. That batch file sets up the path and the environment variables needed by the compiler. You can find it in the directory `c:\Program Files\Microsoft Visual Studio 14.0\Common7\Tools`, or a similar monstrosity. Check the Visual Studio documentation for details.

You can also use the freely available Cygwin programming environment from `www.cygwin.com`. It contains the GNU C compiler and libraries for UNIX-style programming on Windows. With Cygwin, use the command

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/ -I jdk/i

   -shared -Wl,--add-stdcall-alias -o HelloNative.dll HelloNa
```

Type the entire command on a single line.

Note

The Windows version of the header file `jni_md.h` contains the type declaration

```
typedef __int64 jlong;
```

which is specific to the Microsoft compiler. If you use the GNU compiler, you might want to edit that file, for example,

```
#ifdef __GNUC__

  typedef long long jlong;

#else
```

http://www.rejoiceblog.com/

```
    typedef __int64 jlong;

#endif
```

Alternatively, compile with `-D __int64="long long"`, as shown in the sample compiler invocation.

Finally, add a call to the `System.loadLibrary` method in your program. To ensure that the virtual machine will load the library before the first use of the class, use a static initialization block, as in Listing 12.4.

Figure 12.1 gives a summary of the native code processing.

http://www.rejoiceblog.com/

Figure 12.1 Processing native code

Listing 12.4 `helloNative/HelloNativeTest.java`

```
/**
 * @version 1.11 2007-10-26
 * @author Cay Horstmann
 */
```

http://www.rejoiceblog.com/

```
class HelloNativeTest
{
    public static void main(String[] args)
    {
        HelloNative.greeting();
    }

    static
    {
        System.loadLibrary("HelloNative");
    }
}
```

After you compile and run this program, the message “Hello, Native World!” is displayed in a terminal window.

Note

If you run Linux, you must add the current directory to the library path. Either set the `LD_LIBRARY_PATH` environment variable:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

or set the `java.library.path` system property:

http://www.rejoiceblog.com/

```
java -Djava.library.path=. HelloNativeTest
```

Of course, this is not particularly impressive by itself. However, keep in mind that this message is generated by the C `printf` command and not by any Java programming language code. We have taken the first step toward bridging the gap between the two languages!

In summary, follow these steps to link a native method to a Java program:

1. Declare a native method in a Java class.

2. Run `javah` to get a header file with a C declaration for the method.

3. Implement the native method in C.

4. Place the code in a shared library.

5. Load that library in your Java program.

java.lang.System 1.0

- `void loadLibrary(String libname)`

loads the library with the given name. The library is located in the library search path. The exact method for locating the library depends on the operating system.

Note

Some shared libraries for native code must execute certain initializations. You can place any initialization code into a `JNI_OnLoad` method. Similarly, when the virtual machine (VM) shuts down, it will call the `JNI_OnUnload` method if you provide it. The prototypes are

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
```

http://www.rejoiceblog.com/

```
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

The `JNI_OnLoad` method needs to return the minimum version of the VM it requires, such as `JNI_VERSION_1_2`.

12.2 NUMERIC PARAMETERS AND RETURN VALUES

When passing numbers between C and Java, you should understand which types correspond to each other. For example, although C does have data types called `int` and `long`, their implementation is platform-dependent. On some platforms, an `int` is a 16-bit quantity, on others it is a 32-bit quantity. In the Java platform, of course, an `int` is *always* a 32-bit integer. For that reason, JNI defines types `jint`, `jlong`, and so on.

Table 12.1 shows the correspondence between Java types and C types.

Table 12.1 Java Types and C Types

Java Programming Language	C Programming Language	Bytes
boolean	jboolean	1
byte	jbyte	1
char	jchar	2
short	jshort	2
int	jint	4
long	jlong	8
float	jfloat	4
double	jdouble	8

In the header file `jni.h`, these types are declared with `typedef` statements as the equivalent types on the target platform. That header file also defines the constants `JNI_FALSE = 0` and `JNI_TRUE = 1`.

Until Java 5, Java had no direct analog of the C `printf` function. In the following examples, we will suppose you are stuck with an ancient JDK release and decide to implement the same functionality by calling the C `printf` function in a native method.

Listing 12.5 shows a class called `Printf1` that uses a native method to print a floating-point number with a given field width and precision.

http://www.rejoiceblog.com/

Listing 12.5 `printf1/Printf1.java`

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf1
{
      public static native int print(int width, int preci

      static
      {
         System.loadLibrary("Printf1");
      }
}
```

Notice that when the method is implemented in C, all `int` and `double` parameters are changed to `jint` and `jdouble`, as shown in Listing 12.6.

Listing 12.6 `printf1/Printf1.c`

```
/**
      @version 1.10 1997-07-01
```

http://www.rejoiceblog.com/

```
     @author Cay Horstmann
*/

#include "Printf1.h"
#include <stdio.h>

JNIEXPORT jint JNICALL Java_Printf1_print(JNIEnv* env, j
   jint width, jint precision, jdouble x)
{
   char fmt[30];
   jint ret;
   sprintf(fmt, "%%%d.%df", width, precision);
   ret = printf(fmt, x);
   fflush(stdout);
   return ret;
}

```

The function simply assembles a format string `"%w.pf"` in the variable `fmt`, then calls `printf`. It returns the number of characters printed.

Listing 12.7 shows the test program that demonstrates the `Printf1` class.

http://www.rejoiceblog.com/

Listing 12.7 `printf1/Printf1Test.java`

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf1Test
{
      public static void main(String[] args)
      {
         int count = Printf1.print(8, 4, 3.14);
         count += Printf1.print(8, 4, count);
         System.out.println();
         for (int i = 0; i < count; i++)
            System.out.print("-");
         System.out.println();
      }
}
```

12.3 STRING PARAMETERS

Next, we want to consider how to transfer strings to and from native methods. Strings are quite different in the two languages: In Java they are sequences of UTF-16 code

http://www.rejoiceblog.com/

points whereas C strings are null-terminated sequences of bytes. JNI has two sets of functions for manipulating strings: One converts Java strings to "modified UTF-8" byte sequences and another converts them to arrays of UTF-16 values—that is, to `jchar` arrays. (The UTF-8, "modified UTF-8," and UTF-16 formats were discussed in Chapter 1. Recall that the UTF-8 and "modified UTF-8" encodings leave ASCII characters unchanged, but all other Unicode characters are encoded as multibyte sequences.)

Note

The standard UTF-8 encoding and the "modified UTF-8" encoding differ only for "supplementary" characters with codes higher than 0xFFFF. In the standard UTF-8 encoding, these characters are encoded as 4-byte sequences. In the "modified" encoding, each such character is first encoded as a pair of "surrogates" in the UTF-16 encoding, and then each surrogate is encoded with UTF-8, yielding a total of 6 bytes. This is clumsy, but it is a historical accident—the JVM specification was written when Unicode was still limited to 16 bits.

If your C code already uses Unicode, you'll want to use the second set of conversion functions. On the other hand, if all your strings are restricted to ASCII characters, you can use the "modified UTF-8" conversion functions.

A native method with a `String` parameter actually receives a value of an opaque type called `jstring`. A native method with a return value of type `String` must return a value of type `jstring`. JNI functions read and construct these `jstring` objects. For example, the `NewStringUTF` function makes a new `jstring` object out of a `char` array that contains ASCII characters or, more generally, "modified UTF-8"-encoded byte sequences.

JNI functions have a somewhat odd calling convention. Here is a call to the `NewStringUTF` function:

```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting(JNIEnv

{

    jstring jstr;
```

http://www.rejoiceblog.com/

```
    char greeting[] = "Hello, Native World\n";

    jstr = (*env)->NewStringUTF(env, greeting);

    return jstr;

}
```

Note

Unless explicitly mentioned otherwise, all code in this chapter is C code.

All calls to JNI functions use the `env` pointer that is the first argument of every native method. The `env` pointer is a pointer to a table of function pointers (see Figure 12.2). Therefore, you must prefix every JNI call with `(*env)->` to actually dereference the function pointer. Furthermore, `env` is the first parameter of every JNI function.

Figure 12.2 The `env` pointer

http://www.rejoiceblog.com/

C++ Note

It is simpler to access JNI functions in C++. The C++ version of the `JNIEnv` class has inline member functions that take care of the function pointer lookup for you. For example, you can call the `NewStringUTF` function as

```
jstr = env->NewStringUTF(greeting);
```

Note that you omit the `JNIEnv` pointer from the parameter list of the call.

The `NewStringUTF` function lets you construct a new `jstring`. To read the contents of an existing `jstring` object, use the `GetStringUTFChars` function. This function returns a `const jbyte*` pointer to the "modified UTF-8" characters that describe the character string. Note that a specific virtual machine is free to choose this character encoding for its internal string representation, so you might get a character pointer into the actual Java string. Since Java strings are meant to be immutable, it is *very* important that you treat the `const` seriously and do not try to write into this character array. On the other hand, if the virtual machine uses UTF-16 or UTF-32 characters for its internal string representation, this function call allocates a new memory block that will be filled with the "modified UTF-8" equivalents.

The virtual machine must know when you are finished using the string so that it can garbage-collect it. (The garbage collector runs in a separate thread, and it can interrupt the execution of native methods.) For that reason, you must call the `ReleaseStringUTFChars` function.

Alternatively, you can supply your own buffer to hold the string characters by calling the `GetStringRegion` or `GetStringUTFRegion` methods.

Finally, the `GetStringUTFLength` function returns the number of characters needed for the "modified UTF-8" encoding of the string.

Note

You can find the JNI API at

http://www.rejoiceblog.com/

http://docs.oracle.com/javase/7/docs/technotes/guides/jni.

Accessing Java Strings from C Code

- `jstring NewStringUTF(JNIEnv* env, const char bytes[])`

returns a new Java string object from a zero byte-terminated "modified UTF-8" byte sequence, or `NULL` if the string cannot be constructed.

- `jsize GetStringUTFLength(JNIEnv* env, jstring string)`

returns the number of bytes required for the "modified UTF-8" encoding (not counting the zero byte terminator).

- `const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`

returns a pointer to the "modified UTF-8" encoding of a string, or `NULL` if the character array cannot be constructed. The pointer is valid until `ReleaseStringUTFChars` is called. `isCopy` points to a `jboolean` filled with `JNI_TRUE` if a copy is made, with `JNI_FALSE` otherwise.

- `void ReleaseStringUTFChars(JNIEnv* env, jstring string, const jbyte bytes[])`

informs the virtual machine that the native code no longer needs access to the Java string through `bytes` (a pointer returned by `GetStringUTFChars`).

- `void GetStringRegion(JNIEnv *env, jstring string, jsize start, jsize length, jchar *buffer)`

copies a sequence of UTF-16 double bytes from a string to a user-supplied buffer of size at least 2 × `length`.

- `void GetStringUTFRegion(JNIEnv *env, jstring string, jsize start, jsize length, jbyte *buffer)`

copies a sequence of "modified UTF-8" bytes from a string to a user-supplied buffer.

http://www.rejoiceblog.com/

The buffer must be long enough to hold the bytes. In the worst case, 3 × `length` bytes are copied.

- `jstring NewString(JNIEnv* env, const jchar chars[], jsize length)`

returns a new Java string object from a Unicode string, or `NULL` if the string cannot be constructed.

- `jsize GetStringLength(JNIEnv* env, jstring string)`

returns the number of characters in the string.

- `const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)`

returns a pointer to the Unicode encoding of a string, or `NULL` if the character array cannot be constructed. The pointer is valid until `ReleaseStringChars` is called. `isCopy` is either `NULL` or points to a `jboolean` filled with `JNI_TRUE` if a copy is made, with `JNI_FALSE` otherwise.

- `void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])`

informs the virtual machine that the native code no longer needs access to the Java string through `chars` (a pointer returned by `GetStringChars`).

Let us put these functions to work and write a class that calls the C function `sprintf`. We would like to call the function as shown in Listing 12.8.

Listing 12.8 `printf2/Printf2Test.java`

```
/**
   * @version 1.10 1997-07-01
   * @author Cay Horstmann
   */
```

http://www.rejoiceblog.com/

```
class Printf2Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
        double amountDue = price * (1 + tax / 100);

        String s = Printf2.sprint("Amount due = %8.2f",
        System.out.println(s);
    }
}
```

Listing 12.9 shows the class with the native `sprint` method.

Listing 12.9 `printf2/Printf2.java`

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
```

http://www.rejoiceblog.com/

```
class Printf2
{
        public static native String sprint(String format, 

       static
       {
          System.loadLibrary("Printf2");
       }
}
```

Therefore, the C function that formats a floating-point number has the prototype

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jc
```

Listing 12.10 shows the code for the C implementation. Note the calls to `GetStringUTFChars` to read the format argument, `NewStringUTF` to generate the return value, and `ReleaseStringUTFChars` to inform the virtual machine that access to the string is no longer required.

Listing 12.10 `printf2/Printf2.c`

```
/**
        @version 1.10 1997-07-01
        @author Cay Horstmann
*/
```

http://www.rejoiceblog.com/

```

#include    "Printf2.h"
#include    <string.h>
#include    <stdlib.h>
#include    <float.h>

/**
      @param format a string containing a printf format s
      (such as "%8.2f"). Substrings "%%" are skipped.
      @return a pointer to the format specifier (skipping
      or NULL if there wasn't a unique format specifier
*/
char* find_format(const char format[])
{
    char* p;
    char* q;

      p = strchr(format, '%');
      while (p != NULL && *(p + 1) == '%') /* skip %% */
```

http://www.rejoiceblog.com/

```
24          p = strchr(p + 2, '%');
25       if (p == NULL) return NULL;
26       /* now check that % is unique */
27       p++;
28      q = strchr(p, '%');
29      while (q != NULL && *(q + 1) == '%') /* skip %% */
30         q = strchr(q + 2, '%');
31      if (q != NULL) return NULL; /* % not unique */
32      q = p + strspn(p, " -0+#"); /* skip past flags */
33      q += strspn(q, "0123456789"); /* skip past field wid
34      if (*q == '.') { q++; q += strspn(q, "0123456789");
35         /* skip past precision */
36      if (strchr("eEfFgG", *q) == NULL) return NULL;
37         /* not a floating-point format */
38      return p;
39   }
40
41   JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* en
42      jstring format, jdouble x)
43   {
```

http://www.rejoiceblog.com/

```
44      const char* cformat;
45      char* fmt;
46      jstring ret;
47
48       cformat = (*env)->GetStringUTFChars(env, format, NUL
49       fmt = find_format(cformat);
50       if (fmt == NULL)
51          ret = format;
52       else
53       {
54          char* cret;
55          int width = atoi(fmt);
56          if (width == 0) width = DBL_DIG + 10;
57          cret = (char*) malloc(strlen(cformat) + width);
58          sprintf(cret, cformat, x);
59          ret = (*env)->NewStringUTF(env, cret);
60          free(cret);
61       }
62       (*env)->ReleaseStringUTFChars(env, format, cformat);
```

http://www.rejoiceblog.com/

```
        return ret;

    }

```

In this function, we chose to keep error handling simple. If the format code to print a floating-point number is not of the form `%w.pc`, where `c` is one of the characters `e`, `E`, `f`, `g`, or `G`, then we simply do not format the number. We'll show you later how to make a native method throw an exception.

12.4 ACCESSING FIELDS

All the native methods you saw so far were static methods with number and string parameters. We'll now consider native methods that operate on objects. As an exercise, we will reimplement as native a method of the `Employee` class that was introduced in Volume I, Chapter 4. Again, this is not something you would normally want to do, but it does illustrate how to access fields from a native method when you need to do so.

12.4.1 Accessing Instance Fields

To see how to access instance fields from a native method, we will reimplement the `raiseSalary` method. Here is the code in Java:

```
public void raiseSalary(double byPercent)

{

   salary *= 1 + byPercent / 100;

}
```

Let us rewrite this as a native method. Unlike the previous examples of native methods, this is not a static method. Running `javac -h` gives the following prototype:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv *, jo
```

http://www.rejoiceblog.com/

Note the second argument. It is no longer of type `jclass` but of type `jobject`. In fact, it is an equivalent of the `this` reference. Static methods obtain a reference to the class, whereas nonstatic methods obtain a reference to the implicit `this` argument object.

Now we access the `salary` field of the implicit argument. In the "raw" Java-toC binding of Java 1.0, this was easy—a programmer could directly access object data fields. However, direct access requires all virtual machines to expose their internal data layout. For that reason, the JNI requires programmers to get and set the values of data fields by calling special JNI functions.

In our case, we need to use the `GetDoubleField` and `SetDoubleField` functions because the type of `salary` is `double`. There are other functions —`GetIntField/SetIntField`, `GetObjectField/SetObjectField`, and so on for other field types. The general syntax is:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);

(*env)->SetXxxField(env, this_obj, fieldID, x);
```

Here, `fieldID` is a value of a special type, `jfieldID`, that identifies a field in a structure, and *Xxx* represents a Java data type (`Object`, `Boolean`, `Byte`, and so on). To obtain the `fieldID`, you must first get a value representing the class, which you can do in one of two ways. The `GetObjectClass` function returns the class of any object. For example:

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj)
```

The `FindClass` function lets you specify the class name as a string (curiously, with `/` characters instead of periods as package name separators).

```
jclass class_String = (*env)->FindClass(env, "java/lang/Strin
```

Use the `GetFieldID` function to obtain the `fieldID`. You must supply the name of the field and its *signature*, an encoding of its type. For example, here is the code to obtain the field ID of the `salary` field:

http://www.rejoiceblog.com/

```
jfieldID id_salary = (*env)->GetFieldID(env, class_Employee,
```

The string `"D"` denotes the type `double`. You'll learn the complete rules for encoding signatures in the next section.

You might be thinking that accessing a data field is quite convoluted. The designers of the JNI did not want to expose the data fields directly, so they had to supply functions for getting and setting field values. To minimize the cost of these functions, computing the field ID from the field name—which is the most expensive step—is factored out into a separate step. That is, if you repeatedly get and set the value of a particular field, you can incur the cost of computing the field identifier only once.

Let us put all the pieces together. The following code reimplements the `raiseSalary` method as a native method:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env,

{

   /* get the class */

   jclass class_Employee = (*env)->GetObjectClass(env, this_o

   /* get the field ID */

   jfieldID id_salary = (*env)->GetFieldID(env, class_Employe

   /* get the field value */

   jdouble salary = (*env)->GetDoubleField(env, this_obj, id_
```

http://www.rejoiceblog.com/

```
    salary *= 1 + byPercent / 100;



    /* set the field value */

    (*env)->SetDoubleField(env, this_obj, id_salary, salary);

}
```

Caution

Class references are only valid until the native method returns. You cannot cache the return values of `GetObjectClass` in your code. Do *not* store away a class reference for reuse in a later method call. You must call `GetObjectClass` every time the native method executes. If this is intolerable, you can lock the reference with a call to `NewGlobalRef`:

```
static jclass class_X = 0;

static jfieldID id_a;

. . .

if (class_X == 0)

{

  jclass cx = (*env)->GetObjectClass(env, obj);

  class_X = (*env)->NewGlobalRef(env, cx);

  id_a = (*env)->GetFieldID(env, cls, "a", ". . .");

}
```

Now you can use the class reference and field IDs in subsequent calls. When you are done using the class, make sure to call

http://www.rejoiceblog.com/

```
(*env)->DeleteGlobalRef(env, class_X);
```

Listings 12.11 and 12.12 show the Java code for a test program and the `Employee` class. Listing 12.13 contains the C code for the native `raiseSalary` method.

Listing 12.11 `employee/EmployeeTest.java`

```
/**
 * @version 1.11 2018-05-01
 * @author Cay Horstmann
 */

public class EmployeeTest
{
   public static void main(String[] args)
   {
      var staff = new Employee[3];

         staff[0] = new Employee("Harry Hacker", 35000);
         staff[1] = new Employee("Carl Cracker", 75000);
         staff[2] = new Employee("Tony Tester", 38000);

```

http://www.rejoiceblog.com/

```
            for (Employee e : staff)
               e.raiseSalary(5);
            for (Employee e : staff)
               e.print();
         }
   }
```

Listing 12.12 `employee/Employee.java`

```
/**
 * @version 1.10 1999-11-13
 * @author Cay Horstmann
 */

public class Employee
{
   private String name;
   private double salary;

      public native void raiseSalary(double byPercent);
```

http://www.rejoiceblog.com/

```

        public Employee(String n, double s)
        {
            name = n;
            salary = s;
        }

        public void print()
        {
            System.out.println(name + " " + salary);
        }

        static
        {
            System.loadLibrary("Employee");
        }
    }
```

Listing 12.13 `employee/Employee.c`

```
/**
```

http://www.rejoiceblog.com/

```
 2        @version 1.10 1999-11-13
 3        @author Cay Horstmann
 4   */
 5
 6   #include "Employee.h"
 7
 8   #include <stdio.h>
 9
10   JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv*
11   {
12      /* get the class */
13      jclass class_Employee = (*env)->GetObjectClass(env, t
14
15        /* get the field ID */
16        jfieldID id_salary = (*env)->GetFieldID(env, class_
17
18       /* get the field value */
19       jdouble salary = (*env)->GetDoubleField(env, this_ob
20
```

http://www.rejoiceblog.com/

```
        salary *= 1 + byPercent / 100;

        /* set the field value */
        (*env)->SetDoubleField(env, this_obj, id_salary, sal
    }

```

12.4.2 Accessing Static Fields

Accessing static fields is similar to accessing nonstatic fields. Use the `GetStaticFieldID` and `GetStatic`*Xxx*`Field/SetStatic`*Xxx*`Field` functions that work almost identically to their nonstatic counterparts, with two differences:

• As you have no object, you must use `FindClass` instead of `GetObjectClass` to obtain the class reference.

• You have to supply the class, not the instance object, when accessing the field.

For example, here is how you can get a reference to `System.out`:

```
/* get the class */
jclass class_System = (*env)->FindClass(env, "java/lang/Syste

/* get the field ID */
jfieldID id_out = (*env)->GetStaticFieldID(env, class_System,
     "Ljava/io/PrintStream;");
```

http://www.rejoiceblog.com/

```
/* get the field value */

jobject obj_out = (*env)->GetStaticObjectField(env, class_Sys
```

Accessing Fields

- `jfieldID GetFieldID(JNIEnv *env, jclass cl, const char name[], const char fieldSignature[])`

returns the identifier of a field in a class.

- *Xxx* `Get`*Xxx*`Field(JNIEnv *env, jobject obj, jfieldID id)`

returns the value of a field. The field type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

- `void Set`*Xxx*`Field(JNIEnv *env, jobject obj, jfieldID id,` *Xxx* `value)`

sets a field to a new value. The field type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

- `jfieldID GetStaticFieldID(JNIEnv *env, jclass cl, const char name[], const char fieldSignature[])`

returns the identifier of a static field in a class.

- *Xxx* `GetStatic`*Xxx*`Field(JNIEnv *env, jclass cl, jfieldID id)`

returns the value of a static field. The field type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

- `void SetStatic`*Xxx*`Field(JNIEnv *env, jclass cl, jfieldID id,` *Xxx* `value)`

sets a static field to a new value. The field type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

http://www.rejoiceblog.com/

12.5 ENCODING SIGNATURES

To access instance fields and call methods defined in the Java programming language, you need to learn the rules for "mangling" the names of data types and method signatures. (A method signature describes the parameters and return type of the method.) Here is the encoding scheme:

B	byte
C	char
D	double
F	float
I	int
J	long
L*classname*;	a class type
S	short
V	void
Z	boolean

To describe an array type, use a `[`. For example, an array of strings is

```
[Ljava/lang/String;
```

A `float[][]` is mangled into

```
[[F
```

For the complete signature of a method, list the parameter types inside a pair of parentheses and then list the return type. For example, a method receiving two integers and returning an integer is encoded as

```
(II)I
```

The `sprint` method in Section 12.3, "String Parameters," on p. 825 has a mangled signature of

```
(Ljava/lang/String;D)Ljava/lang/String;
```

http://www.rejoiceblog.com/

That is, the method receives a `String` and a `double` and returns a `String`.

Note that the semicolon at the end of the `L` expression is the terminator of the type expression, not a separator between parameters. For example, the constructor

```
Employee(java.lang.String, double, java.util.Date)
```

has a signature

```
"(Ljava/lang/String;DLjava/util/Date;)V"
```

Note that there is no separator between the `D` and `Ljava/util/Date;`. Also note that in this encoding scheme, you must use `/` instead of `.` to separate the package and class names. The `V` at the end denotes a return type of `void`. Even though you don't specify a return type for constructors in Java, you need to add a `V` to the virtual machine signature.

Tip

You can use the `javap` command with option `-s` to generate the method signatures from class files. For example, run

```
javap -s -private Employee
```

You will get the following output, displaying the signatures of all fields and methods:

```
Compiled from "Employee.java"

public class Employee extends java.lang.Object{

private java.lang.String name;

  Signature: Ljava/lang/String;
```

http://www.rejoiceblog.com/

```
private double salary;

  Signature: D

public Employee(java.lang.String, double);

  Signature: (Ljava/lang/String;D)V

public native void raiseSalary(double);

  Signature: (D)V

public void print();

  Signature: ()V

static {};

  Signature: ()V

}
```

Note

There is no rationale whatsoever for forcing programmers to use this mangling scheme for signatures. The designers of the native calling mechanism could have just as easily written a function that reads signatures in the Java programming language style, such as `void(int,java.lang.String)`, and encodes them into whatever internal representation they prefer. Then again, using the mangled signatures lets you partake in the mystique of programming close to the virtual machine.

12.6 CALLING JAVA METHODS

Of course, Java programming language functions can call C functions—that is what native methods are for. Can we go the other way? Why would we want to do this anyway? It often happens that a native method needs to request a service from an object that was passed to it. We'll first show you how to do it for instance methods, then

http://www.rejoiceblog.com/

for static methods.

12.6.1 Instance Methods

As an example of calling a Java method from native code, let's enhance the `Printf` class and add a method that works similarly to the C function `fprintf`. That is, it should be able to print a string on an arbitrary `PrintWriter` object. Here is the definition of the method in Java:

```
class Printf3

{

   public native static void fprint(PrintWriter out, String s

   . . .

}
```

We'll first assemble the string to be printed into a `String` object `str`, as in the `sprint` method that we already implemented. Then, from the C function that implements the native method, we'll call the `print` method of the `PrintWriter` class.

You can call any Java method from C by using the function call

```
(*env)->CallXxxMethod(env, implicit parameter, methodID, explicit paramete
```

Replace *Xxx* with `Void`, `Int`, `Object`, and so on, depending on the return type of the method. Just as you need a `fieldID` to access a field of an object, you need a method ID to call a method. To obtain a method ID, call the JNI function `GetMethodID` and supply the class, the name of the method, and the method signature.

In our example, we want to obtain the ID of the `print` method of the `PrintWriter` class. The `PrintWriter` class has several overloaded methods called `print`. For that reason, you must also supply a string describing the parameters and the return value of the specific function that you want to use. For example, we want to use `void`

http://www.rejoiceblog.com/

`print(java.lang.String)`. As described in the preceding section, we must now "mangle" the signature into the string "`(Ljava/lang/String;)V`".

Here is the complete code to make the method call, by

1. Obtaining the class of the implicit parameter

2. Obtaining the method ID

3. Making the call

```
/* get the class */

class_PrintWriter = (*env)->GetObjectClass(env, out);



/* get the method ID */

id_print = (*env)->GetMethodID(env, class_PrintWriter, "print



/* call the method */

(*env)->CallVoidMethod(env, out, id_print, str);
```

Listings 12.14 and 12.15 show the Java code for a test program and the `Printf3` class. Listing 12.16 contains the C code for the native `fprint` method.

Note

The numerical method IDs and field IDs are conceptually similar to `Method` and `Field` objects in the reflection API. You can convert between them with the following functions:

```
jobject ToReflectedMethod(JNIEnv* env, jclass class, jmethodID methodI
```

http://www.rejoiceblog.com/

```
    // returns Method object

methodID FromReflectedMethod(JNIEnv* env, jobject method);

jobject ToReflectedField(JNIEnv* env, jclass class, jfieldID fieldID);

    // returns Field object

fieldID FromReflectedField(JNIEnv* env, jobject field);
```

12.6.2 Static Methods

Calling static methods from native methods is similar to calling instance methods. There are two differences:

• Use the `GetStaticMethodID` and `CallStatic`*Xxx*`Method` functions

• Supply a class object, not an implicit parameter object, when invoking the method

As an example of this, let's make the call to the static method

```
System.getProperty("java.class.path")
```

from a native method. The return value of this call is a string that gives the current class path.

First, we have to find the class to use. As we have no object of the class `System`

readily available, we use `FindClass` rather than `GetObjectClass`.

```
jclass class_System = (*env)->FindClass(env, "java/lang/Syste
```

Next, we need the ID of the static `getProperty` method. The encoded signature of that method is

http://www.rejoiceblog.com/

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

because both the parameter and the return value are strings. Hence, we obtain the method ID as follows:

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env, cla

     "(Ljava/lang/String;)Ljava/lang/String;");
```

Finally, we can make the call. Note that the class object is passed to the `CallStaticObjectMethod` function.

```
jobject obj_ret = (*env)->CallStaticObjectMethod(env, class_S

     (*env)->NewStringUTF(env, "java.class.path"));
```

The return value of this method is of type `jobject`. If we want to manipulate it as a string, we must cast it to `jstring`:

```
jstring str_ret = (jstring) obj_ret;
```

C++ Note

In C, the types `jstring` and `jclass`, as well as the array types to be introduced later, are all type-equivalent to `jobject`. The cast of the preceding example is therefore not strictly necessary in C. But in C++, these types are defined as pointers to “dummy classes” that have the correct inheritance hierarchy. For example, assigning a `jstring` to a `jobject` is legal without a cast in C++, but an assignment from a `jobject` to a `jstring` requires a cast.

http://www.rejoiceblog.com/

12.6.3 Constructors

A native method can create a new Java object by invoking its constructor. Invoke the constructor by calling the `NewObject` function.

```
jobject obj_new = (*env)->NewObject(env, class, methodID, construct
```

You can obtain the method ID needed for this call from the `GetMethodID` function by specifying the method name as `"<init>"` and the encoded signature of the constructor (with return type `void`). For example, here is how a native method can create a `FileOutputStream` object:

```
const char[] fileName = ". . .";

jstring str_fileName = (*env)->NewStringUTF(env, fileName);

jclass class_FileOutputStream = (*env)->FindClass(env, "java/

jmethodID id_FileOutputStream

   = (*env)->GetMethodID(env, class_FileOutputStream, "<init>

jobject obj_stream

   = (*env)->NewObject(env, class_FileOutputStream, id_FileOu
```

Note that the signature of the constructor takes a parameter of type `java.lang.String` and has a return type of `void`.

12.6.4 Alternative Method Invocations

Several variants of the JNI functions can be used to call a Java method from native code. These are not as important as the functions that we already discussed, but they are occasionally useful.

The `CallNonvirtual`*Xxx*`Method` functions receive an implicit argument, a method ID, a class object (which must correspond to a superclass of the implicit argument), and

http://www.rejoiceblog.com/

explicit arguments. The function calls the version of the method in the specified class, bypassing the normal dynamic dispatch mechanism.

All call functions have versions with suffixes “A” and “V” that receive the explicit parameters in an array or a `va_list` (as defined in the C header `stdarg.h`).

Listing 12.14 `printf3/Printf3Test.java`

```
import java.io.*;

/**
  * @version 1.11 2018-05-01
  * @author Cay Horstmann
  */
class Printf3Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
        double amountDue = price * (1 + tax / 100);
        var out = new PrintWriter(System.out);
        Printf3.fprint(out, "Amount due = %8.2f\n", amoun
        out.flush();
```

http://www.rejoiceblog.com/

```
    }
}
```

Listing 12.15 `printf3/Printf3.java`

```
import java.io.*;

/**
  * @version 1.10 1997-07-01
  * @author Cay Horstmann
  */
class Printf3
{
    public static native void fprint(PrintWriter out, S

    static
    {
        System.loadLibrary("Printf3");
    }
}
```

http://www.rejoiceblog.com/

Listing 12.16 `printf3/Printf3.c`

```
/**
        @version 1.10 1997-07-01
        @author Cay Horstmann
*/

#include    "Printf3.h"
#include    <string.h>
#include    <stdlib.h>
#include    <float.h>

/**
      @param format a string containing a printf format s
      (such as "%8.2f"). Substrings "%%" are skipped.
      @return a pointer to the format specifier (skipping
      or NULL if there wasn't a unique format specifier
*/
char* find_format(const char format[])
{
```

http://www.rejoiceblog.com/

```
    char* p;
    char* q;

    p = strchr(format, '%');
    while (p != NULL && *(p + 1) == '%') /* skip %% */
       p = strchr(p + 2, '%');
    if (p == NULL) return NULL;
    /* now check that % is unique */
    p++;
    q = strchr(p, '%');
    while (q != NULL && *(q + 1) == '%') /* skip %% */
      q = strchr(q + 2, '%');
    if (q != NULL) return NULL; /* % not unique */
    q = p + strspn(p, " -0+#"); /* skip past flags */
    q += strspn(q, "0123456789"); /* skip past field widt
    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
       /* skip past precision */
    if (strchr("eEfFgG", *q) == NULL) return NULL;
       /* not a floating-point format */
    return p;
```

http://www.rejoiceblog.com/

```
}

JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env,
    jobject out, jstring format, jdouble x)
{
    const char* cformat;
    char* fmt;
    jstring str;
    jclass class_PrintWriter;
    jmethodID id_print;

    cformat = (*env)->GetStringUTFChars(env, format, NULL
    fmt = find_format(cformat);
    if (fmt == NULL)
        str = format;
    else
    {
        char* cstr;
        int width = atoi(fmt);
```

http://www.rejoiceblog.com/

```
        if (width == 0) width = DBL_DIG + 10;
        cstr = (char*) malloc(strlen(cformat) + width);
        sprintf(cstr, cformat, x);
        str = (*env)->NewStringUTF(env, cstr);
        free(cstr);
    }
    (*env)->ReleaseStringUTFChars(env, format, cformat);

    /* now call ps.print(str) */

    /* get the class */
    class_PrintWriter = (*env)->GetObjectClass(env, out);

    /* get the method ID */
    id_print = (*env)->GetMethodID(env, class_PrintWriter

    /* call the method */
    (*env)->CallVoidMethod(env, out, id_print, str);
}
```

http://www.rejoiceblog.com/

Executing Java Methods

- `jmethodID GetMethodID(JNIEnv *env, jclass cl, const char name[], const char methodSignature[])`

returns the identifier of a method in a class.

- *Xxx* `Call`*Xxx*`Method(JNIEnv *env, jobject obj, jmethodID id, args)`

- *Xxx* `Call`*Xxx*`MethodA(JNIEnv *env, jobject obj, jmethodID id, jvalue args[])`

- *Xxx* `Call`*Xxx*`MethodV(JNIEnv *env, jobject obj, jmethodID id, va_list args)`

calls a method. The return type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of `jvalue`, where `jvalue` is a union defined as

```
typedef union jvalue

{

    jboolean z;

    jbyte b;

    jchar c;

    jshort s;

    jint i;

    jlong j;

    jfloat f;
```

http://www.rejoiceblog.com/

```
    jdouble d;

    jobject l;

  } jvalue;
```

The third function receives the method parameters in a `va_list`, as defined in the C header `stdarg.h`.

- *Xxx* `CallNonvirtual`*Xxx*`Method(JNIEnv *env, jobject obj, jclass cl, jmethodID id, args)`

- *Xxx* `CallNonvirtual`*Xxx*`MethodA(JNIEnv *env, jobject obj, jclass cl, jmethodID id, jvalue args[])`

- *Xxx* `CallNonvirtual`*Xxx*`MethodV(JNIEnv *env, jobject obj, jclass cl, jmethodID id, va_list args)`

calls a method, bypassing dynamic dispatch. The return type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of `jvalue`. The third function receives the method parameters in a `va_list`, as defined in the C header `stdarg.h`.

- `jmethodID GetStaticMethodID(JNIEnv *env, jclass cl, const char name[], const char methodSignature[])`

returns the identifier of a static method in a class.

- *Xxx* `CallStatic`*Xxx*`Method(JNIEnv *env, jclass cl, jmethodID id, args)`

- *Xxx* `CallStatic`*Xxx*`MethodA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])`

- *Xxx* `CallStatic`*Xxx*`MethodV(JNIEnv *env, jclass cl, jmethodID id, va_list args)`

calls a static method. The return type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`,

http://www.rejoiceblog.com/

Short, Int, Long, Float, or Double. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of jvalue. The third function receives the method parameters in a va_list, as defined in the C header stdarg.h.

- jobject NewObject(JNIEnv *env, jclass cl, jmethodID id, args)
- jobject NewObjectA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])
- jobject NewObjectV(JNIEnv *env, jclass cl, jmethodID id, va_list args)

calls a constructor. The method ID is obtained from GetMethodID with a method name of "<init>" and a return type of void. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of jvalue. The third function receives the method parameters in a va_list, as defined in the C header stdarg.h.

12.7 ACCESSING ARRAY ELEMENTS

All array types of the Java programming language have corresponding C types, as shown in Table 12.2.

Table 12.2 Correspondence between Java Array Types and C Types

Java Type	C Type	Java Type	C Type
boolean[]	jbooleanArray	long[]	jlongArray
byte[]	jbyteArray	float[]	jfloatArray
char[]	jcharArray	double[]	jdoubleArray
int[]	jintArray	Object[]	jobjectArray
short[]	jshortArray		

C++ Note

In C, all these array types are actually type synonyms of jobject. In C++,

http://www.rejoiceblog.com/

however, they are arranged in the inheritance hierarchy shown in Figure 12.3. The type `jarray` denotes a generic array.

http://www.rejoiceblog.com/

Figure 12.3 Inheritance hierarchy of array types

The `GetArrayLength` function returns the length of an array.

```
jarray array = . . .; jsize length = (*env)->GetArrayLength(e
```

How you access elements in an array depends on whether the array stores objects or values of a primitive type (`bool`, `char`, or a numeric type). To access elements in an object array, use the `GetObjectArrayElement` and `SetObjectArrayElement` methods.

```
jobjectArray array = . . .;

int i, j;

jobject x = (*env)->GetObjectArrayElement(env, array, i);

(*env)->SetObjectArrayElement(env, array, j, x);
```

Although simple, this approach is also clearly inefficient; you want to be able to access array elements directly, especially when doing vector and matrix computations.

The `Get`*Xxx*`ArrayElements` function returns a C pointer to the starting element of an array. As with ordinary strings, you must remember to call the corresponding `Release`*Xxx*`ArrayElements` function to tell the virtual machine when you no longer need that pointer. Here, the type *Xxx* must be a primitive type—that is, not `Object`. You can then read and write the array elements directly. However, since the pointer *might point to a copy*, any changes that you make are guaranteed to be reflected in the original array only after you call the corresponding `Release`*Xxx*`ArrayElements` function!

Note

You can find out if an array is a copy by passing a pointer to a `jboolean` variable as the third parameter to a `GetXxxArrayElements` method. The variable is filled with `JNI_TRUE` if the array is a copy. If you aren't interested

http://www.rejoiceblog.com/

in that information, just pass a `NULL` pointer.

Here is an example that multiplies all elements in an array of `double` values by a constant. We obtain a C pointer `a` into the Java array and then access individual elements as `a[i]`.

```
jdoubleArray array_a = . . .;

double scaleFactor = . . .;

double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL

for (i = 0; i < (*env)->GetArrayLength(env, array_a); i++)

   a[i] = a[i] * scaleFactor;

(*env)->ReleaseDoubleArrayElements(env, array_a, a, 0);
```

Whether the virtual machine actually copies the array depends on how it allocates arrays and does its garbage collection. Some “copying” garbage collectors routinely move objects around and update object references. That strategy is not compatible with “pinning” an array to a particular location, because the collector cannot update the pointer values in native code.

Note

In the Oracle JVM implementation, `boolean` arrays are represented as packed arrays of 32-bit words. The `GetBooleanArrayElements` method copies them into unpacked arrays of `jboolean` values.

To access just a few elements of a large array, use the `Get`*Xxx*`ArrayRegion` and `Set`*Xxx*`ArrayRegion` methods that copy a range of elements from the Java array into a C array and back.

http://www.rejoiceblog.com/

You can create new Java arrays in native methods with the `New*Xxx*Array` function. To create a new array of objects, specify the length, the type of the array elements, and an initial element for all entries (typically, `NULL`). Here is an example:

```
jclass class_Employee = (*env)->FindClass(env, "Employee");

jobjectArray array_e = (*env)->NewObjectArray(env, 100, class
```

Arrays of primitive types are simpler: just supply the length of the array.

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

The array is then filled with zeroes.

Note

The following methods are used for working with “direct buffers”:

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address, jlong capacity)

void* GetDirectBufferAddress(JNIEnv* env, jobject buf)

jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf)
```

Direct buffers are used in the `java.nio` package to support more efficient input/output operations and to minimize the copying of data between native and Java arrays.

Manipulating Java Arrays
• `jsize GetArrayLength(JNIEnv *env, jarray array)`

http://www.rejoiceblog.com/

returns the number of elements in the array.

- `jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index)`

returns the value of an array element.

- `void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value)`

sets an array element to a new value.

- *Xxx*`* Get`*Xxx*`ArrayElements(JNIEnv *env, jarray array, jboolean* isCopy)`

yields a C pointer to the elements of a Java array. The field type *Xxx* is one of `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`. The pointer must be passed to `Release`*Xxx*`ArrayElements` when it is no longer needed.`isCopy` is either `NULL` or points to a `jboolean` that is filled with `JNI_TRUE` if a copy is made, with `JNI_FALSE` otherwise.

- `void Release`*Xxx*`ArrayElements(JNIEnv *env, jarray array,` *Xxx* `elems[], jint mode)`

notifies the virtual machine that a pointer obtained by `Get`*Xxx*`ArrayElements` is no longer needed. `mode` is one of `0` (free the `elems` buffer after updating the array elements), `JNI_COMMIT` (do not free the `elems` buffer after updating the array elements), or `JNI_ABORT` (free the `elems` buffer without updating the array elements).

- `void Get`*Xxx*`ArrayRegion(JNIEnv *env, jarray array, jint start, jint length,` *Xxx* `elems[])`

copies elements from a Java array to a C array. The field type *Xxx* is one of `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

- `void Set`*Xxx*`ArrayRegion(JNIEnv *env, jarray array, jint start, jint length,` *Xxx* `elems[])`

copies elements from a C array to a Java array. The field type *Xxx* is one of `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

http://www.rejoiceblog.com/

12.8 HANDLING ERRORS

Native methods are a significant security risk to Java programs. The C runtime system has no protection against array bounds errors, indirection through bad pointers, and so on. It is particularly important that programmers of native methods handle all error conditions to preserve the integrity of the Java platform. In particular, when your native method diagnoses a problem it cannot handle, it should report this problem to the Java virtual machine.

Normally, you would throw an exception in this situation. However, C has no exceptions. Instead, you must call the `Throw` or `ThrowNew` function to create a new exception object. When the native method exits, the Java virtual machine throws that exception.

To use the `Throw` function, call `NewObject` to create an object of a subtype of `Throwable`. For example, here we allocate an `EOFException` object and throw it:

```
jclass class_EOFException = (*env)->FindClass(env, "java/io/E

jmethodID id_EOFException = (*env)->GetMethodID(env, class_EO

   /* ID of no-argument constructor */

jthrowable obj_exc = (*env)->NewObject(env, class_EOFExceptio

(*env)->Throw(env, obj_exc);
```

It is usually more convenient to call `ThrowNew`, which constructs an exception object, given a class and a “modified UTF-8” byte sequence.

```
(*env)->ThrowNew(env, (*env)->FindClass(env, "java/io/EOFExce
```

Both `Throw` and `ThrowNew` merely *post* the exception; they do not interrupt the control flow of the native method. Only when the method returns does the Java virtual machine throw the exception. Therefore, every call to `Throw` and `ThrowNew` should

http://www.rejoiceblog.com/
always be immediately followed by a `return` statement.

> **C++ Note**
>
> If you implement native methods in C++, you cannot throw a Java exception object in your C++ code. In a C++ binding, it would be possible to implement a translation between exceptions in C++ and Java; however, this is not currently done. Use `Throw` or `ThrowNew` to throw a Java exception in a native C++ method, and make sure your native methods throw no C++ exceptions.

Normally, native code need not be concerned with catching Java exceptions. However, when a native method calls a Java method, that method might throw an exception. Moreover, a number of the JNI functions throw exceptions as well. For example, `SetObjectArrayElement` throws an `ArrayIndexOutOfBoundsException` if the index is out of bounds, and an `ArrayStoreException` if the class of the stored object is not a subclass of the element class of the array. In situations like these, a native method should call the `ExceptionOccurred` method to determine whether an exception has been thrown. The call

```
jthrowable obj_exc = (*env)->ExceptionOccurred(env);
```

returns `NULL` if no exception is pending, or a reference to the current exception object. If you just want to check whether an exception has been thrown, without obtaining a reference to the exception object, use

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

Normally, a native method should simply return when an exception has occurred so that the virtual machine can propagate it to the Java code. However, a native method *may* analyze the exception object to determine if it can handle the exception. If it can, then the function

```
(*env)->ExceptionClear(env);
```

http://www.rejoiceblog.com/

must be called to turn off the exception.

In our next example, we implement the `fprint` native method with all the paranoia appropriate for a native method. Here are the exceptions that we throw:

• A `NullPointerException` if the format string is `NULL`

• An `IllegalArgumentException` if the format string doesn't contain a `%` specifier that is appropriate for printing a `double`

• An `OutOfMemoryError` if the call to `malloc` fails

Finally, to demonstrate how to check for an exception when calling a Java method from a native method, we send the string to the stream, a character at a time, and call `ExceptionOccurred` after each call. Listing 12.17 shows the code for the native method, and Listing 12.18 shows the definition of the class containing the native method. Notice that the native method does not immediately terminate when an exception occurs in the call to `PrintWriter.print`—it first frees the `cstr` buffer. When the native method returns, the virtual machine again raises the exception. The test program in Listing 12.19 demonstrates how the native method throws an exception when the formatting string is not valid.

Listing 12.17 `printf4/Printf4.c`

```
/**
   @version 1.10 1997-07-01
   @author Cay Horstmann
*/

#include "Printf4.h"
#include <string.h>
#include <stdlib.h>
```

http://www.rejoiceblog.com/

```
#include <float.h>

 /**
   @param format a string containing a printf format spec
   (such as "%8.2f"). Substrings "%%" are skipped.
   @return a pointer to the format specifier (skipping th
   or NULL if there wasn't a unique format specifier
;  */
 char* find_format(const char format[])
 {
    char* p;
    char* q;

    p = strchr(format, '%');
    while (p != NULL && *(p + 1) == '%') /* skip %% */
       p = strchr(p + 2, '%');
    if (p == NULL) return NULL;
    /* now check that % is unique */
    p++;
```

http://www.rejoiceblog.com/

```
    q = strchr(p, '%');
    while (q != NULL && *(q + 1) == '%') /* skip %% */
    q = strchr(q + 2, '%');
    if (q != NULL) return NULL; /* % not unique */
    q = p + strspn(p, " -0+#"); /* skip past flags */
    q += strspn(q, "0123456789"); /* skip past field widt
    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
       /* skip past precision */
    if (strchr("eEfFgG", *q) == NULL) return NULL;
       /* not a floating-point format */
    return p;
}

JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env,
    jobject out, jstring format, jdouble x)
{
    const char* cformat;
    char* fmt;
    jclass class_PrintWriter;
    jmethodID id_print;
```

http://www.rejoiceblog.com/

```
48      char* cstr;
49      int width;
50      int i;
51
52       if (format == NULL)
53       {
54          (*env)->ThrowNew(env,
55             (*env)->FindClass(env,
56             "java/lang/NullPointerException"),
57             "Printf4.fprint: format is null");
58          return;
59       }
60
61       cformat = (*env)->GetStringUTFChars(env, format, NUL
62       fmt = find_format(cformat);
63
64       if (fmt == NULL)
65       {
66          (*env)->ThrowNew(env,
```

http://www.rejoiceblog.com/

```
            (*env)->FindClass(env,
            "java/lang/IllegalArgumentException"),
            "Printf4.fprint: format is invalid");
          return;
       }

       width = atoi(fmt);
       if (width == 0) width = DBL_DIG + 10;
       cstr = (char*)malloc(strlen(cformat) + width);

       if (cstr == NULL)
        {
            (*env)->ThrowNew(env,
               (*env)->FindClass(env, "java/lang/OutOfMemor
               "Printf4.fprint: malloc failed");
            return;
        }

        sprintf(cstr, cformat, x);

```

http://www.rejoiceblog.com/

```
87          (*env)->ReleaseStringUTFChars(env, format, cformat)

88

89          /* now call ps.print(str) */

90

91          /* get the class */

92          class_PrintWriter = (*env)->GetObjectClass(env, out

93

94          /* get the method ID */

95          id_print = (*env)->GetMethodID(env, class_PrintWrit

96

97          /* call the method */

98          for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccur

99             (*env)->CallVoidMethod(env, out, id_print, cstr[

100

101         free(cstr);

102     }

103

104

105
```

http://www.rejoiceblog.com/

Listing 12.18 `printf4/Printf4.java`

```
import java.io.*;

/**
  * @version 1.10 1997-07-01
  * @author Cay Horstmann
  */
class Printf4
{
     public static native void fprint(PrintWriter ps, S

    static
    {
        System.loadLibrary("Printf4");
    }
}
```

http://www.rejoiceblog.com/

Listing 12.19 `printf4/Printf4Test.java`

```
import java.io.*;

/**
 * @version 1.11 2018-05-01
 * @author Cay Horstmann
 */
class Printf4Test
{
      public static void main(String[] args)
     {
         double price = 44.95;
         double tax = 7.75;
         double amountDue = price * (1 + tax / 100);
         var out = new PrintWriter(System.out);
         /* This call will throw an exception--note the %
         Printf4.fprint(out, "Amount due = %%8.2f\n", amo
         out.flush();
     }
```

http://www.rejoiceblog.com/

```
}
```

Handling Java Exceptions

- `jint Throw(JNIEnv *env, jthrowable obj)`

prepares an exception to be thrown upon exiting from the native code. Returns `0` on success, a negative value on failure.

- `jint ThrowNew(JNIEnv *env, jclass cl, const char msg[])`

prepares an exception of type `cl` to be thrown upon exiting from the native code. Returns `0` on success, a negative value on failure. `msg` is a "modified UTF-8" byte sequence denoting the `String` construction argument of the exception object.

- `jthrowable ExceptionOccurred(JNIEnv *env)`

returns the exception object if an exception is pending, or `NULL` otherwise.

- `jboolean ExceptionCheck(JNIEnv *env)`

returns `true` if an exception is pending.

- `void ExceptionClear(JNIEnv *env)`

clears any pending exceptions.

12.9 USING THE INVOCATION API

Up to now, we have considered programs in the Java programming language that made a few C calls, presumably because C was faster or allowed access to functionality inaccessible from the Java platform. Suppose you are in the opposite situation. You have a C or C++ program and would like to make calls to Java code. The *invocation API* enables you to embed the Java virtual machine into a C or C++ program. Here is the minimal code that you need to initialize a virtual machine:

```
JavaVMOption options[1];
```

http://www.rejoiceblog.com/

```
JavaVMInitArgs vm_args;

JavaVM *jvm;

JNIEnv *env;
```

```
options[0].optionString = "-Djava.class.path=.";
```

```
memset(&vm_args, 0, sizeof(vm_args));

vm_args.version = JNI_VERSION_1_2;

vm_args.nOptions = 1;

vm_args.options = options;
```

```
JNI_CreateJavaVM(&jvm, (void**) &env, &vm_args);
```

The call to `JNI_CreateJavaVM` creates the virtual machine and fills in a pointer `jvm` to the virtual machine and a pointer `env` to the execution environment.

You can supply any number of options to the virtual machine. Simply increase the size of the `options` array and the value of `vm_args.nOptions`. For example,

```
options[i].optionString = "-Djava.compiler=NONE";
```

deactivates the just-in-time compiler.

Tip

When you run into trouble and your program crashes, refuses to initialize the

http://www.rejoiceblog.com/

JVM, or can't load your classes, turn on the JNI debugging mode. Set an option to

```
options[i].optionString = "-verbose:jni";
```

You will see a flurry of messages that indicate the progress in initializing the JVM. If you don't see your classes loaded, check both your path and class path settings.

Once you have set up the virtual machine, you can call Java methods as described in the preceding sections. Simply use the `env` pointer in the usual way.

You'll need the `jvm` pointer only to call other functions in the invocation API. Currently, there are only four such functions. The most important one is the function to terminate the virtual machine:

```
(*jvm)->DestroyJavaVM(jvm);
```

Unfortunately, under Windows, it has become difficult to dynamically link to the `JNI_CreateJavaVM` function in the `jre/bin/client/jvm.dll` library, due to the changed linking rules in Vista and Oracle's reliance on an older C runtime library. Our sample program overcomes this problem by loading the library manually. This is the same approach used by the `java` program—see the file `launcher/java_md.c` in the `src.jar` file that is a part of the JDK.

The C program in Listing 12.20 sets up a virtual machine and calls the `main` method of the `Welcome` class, which was discussed in Volume I, Chapter 2. (Make sure to compile the `Welcome.java` file before starting the invocation test program.)

Listing 12.20 `invocation/InvocationTest.c`

```
/**
      @version 1.20 2007-10-26
      @author Cay Horstmann
```

http://www.rejoiceblog.com/

```
 4   */
 5
 6   #include <jni.h>
 7   #include <stdlib.h>
 8
 9   #ifdef _WINDOWS
10
11   #include <windows.h>
12   static HINSTANCE loadJVMLibrary(void);
13   typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **, void *
14
15   #endif
16
17   int main()
18   {
19       JavaVMOption options[2];
20       JavaVMInitArgs vm_args;
21       JavaVM *jvm;
22       JNIEnv *env;
```

http://www.rejoiceblog.com/

```
23      long status;
24
25      jclass class_Welcome;
26      jclass class_String;
27      jobjectArray args;
28      jmethodID id_main;
29
30   #ifdef _WINDOWS
31      HINSTANCE hjvmlib;
32      CreateJavaVM_t createJavaVM;
33   #endif
34
35      options[0].optionString = "-Djava.class.path=.";
36
37      memset(&vm_args, 0, sizeof(vm_args));
38      vm_args.version = JNI_VERSION_1_2;
39      vm_args.nOptions = 1;
40      vm_args.options = options;
41
42
```

http://www.rejoiceblog.com/

```
43   #ifdef _WINDOWS
44      hjvmlib = loadJVMLibrary();
45      createJavaVM = (CreateJavaVM_t) GetProcAddress(hjvmli
46      status = (*createJavaVM)(&jvm, (void **) &env, &vm_ar
47   #else
48      status = JNI_CreateJavaVM(&jvm, (void **) &env, &vm_a
49   #endif
50
51      if (status == JNI_ERR)
52      {
53         fprintf(stderr, "Error creating VM\n");
54         return 1;
55      }
56
57      class_Welcome = (*env)->FindClass(env, "Welcome");
58      id_main = (*env)->GetStaticMethodID(env, class_Welcom
59
60      class_String = (*env)->FindClass(env, "java/lang/Stri
61      args = (*env)->NewObjectArray(env, 0, class_String, N
```

http://www.rejoiceblog.com/

```
    (*env)->CallStaticVoidMethod(env, class_Welcome, id_m

    (*jvm)->DestroyJavaVM(jvm);

    return 0;
}

#ifdef _WINDOWS

static int GetStringFromRegistry(HKEY key, const char *n
{
    DWORD type, size;

    return RegQueryValueEx(key, name, 0, &type, 0, &size)
        && type == REG_SZ
        && size < (unsigned int) bufsize
        && RegQueryValueEx(key, name, 0, 0, buf, &size) ==
 }

 static void GetPublicJREHome(char *buf, jint bufsize)
```

http://www.rejoiceblog.com/

```
82    {
83        HKEY key, subkey;
84        char version[MAX_PATH];
85
86        /* Find the current version of the JRE */
87        char *JRE_KEY = "Software\\JavaSoft\\Java Runtime En
88        if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0, KEY
89        {
90            fprintf(stderr, "Error opening registry key '%s'\
91            exit(1);
92       }
93
94       if (!GetStringFromRegistry(key, "CurrentVersion", ver
95       {
96          fprintf(stderr, "Failed reading value of registry
97          RegCloseKey(key);
98          exit(1);
99       }
100
```

http://www.rejoiceblog.com/

```
    /* Find directory where the current version is instal
    if (RegOpenKeyEx(key, version, 0, KEY_READ, &subkey)
    {
      fprintf(stderr, "Error opening registry key '%s\\%s
      RegCloseKey(key);
      exit(1);
    }

    if (!GetStringFromRegistry(subkey, "JavaHome", buf, b
    {
       fprintf(stderr, "Failed reading value of registry
          JRE_KEY, version);
       RegCloseKey(key);
       RegCloseKey(subkey);
       exit(1);
    }

    RegCloseKey(key);
    RegCloseKey(subkey);
  }
```

http://www.rejoiceblog.com/

```
121
122   static HINSTANCE loadJVMLibrary(void)
123   {
124      HINSTANCE h1, h2;
125      char msvcdll[MAX_PATH];
126      char javadll[MAX_PATH];
127      GetPublicJREHome(msvcdll, MAX_PATH);
128      strcpy(javadll, msvcdll);
129      strncat(msvcdll, "\\bin\\msvcr71.dll", MAX_PATH - st
130      msvcdll[MAX_PATH - 1] = '\0';
131      strncat(javadll, "\\bin\\client\\jvm.dll", MAX_PATH
132      javadll[MAX_PATH - 1] = '\0';
133
134      h1 = LoadLibrary(msvcdll);
135      if (h1 == NULL)
136      {
137         fprintf(stderr, "Can't load library msvcr71.dll\n
138         exit(1);
139      }
```

http://www.rejoiceblog.com/

```

    h2 = LoadLibrary(javadll);
    if (h2 == NULL)
    {
        fprintf(stderr, "Can't load library jvm.dll\n");
        exit(1);
    }
    return h2;
}

#endif
```

To compile this program under Linux, use

```
gcc -I jdk/include -I jdk/include/linux -o InvocationTest
   -L jdk/jre/lib/i386/client -ljvm InvocationTest.c
```

When compiling in Windows with the Microsoft compiler, use the command line

```
cl -D_WINDOWS -I jdk\include -I jdk\include\win32 InvocationTes
```

You will need to make sure that the `INCLUDE` and `LIB` environment variables include the paths to the Windows API header and library files.

http://www.rejoiceblog.com/

Using Cygwin, compile with

```
gcc -D_WINDOWS -mno-cygwin -I jdk\include -I jdk\include\win32

   -I c:\cygwin\usr\include\w32api -o InvocationTest
```

Before you run the program under Linux/UNIX, make sure that the `LD_LIBRARY_PATH` contains the directories for the shared libraries. For example, if you use the `bash` shell on Linux, issue the following command:

```
export LD_LIBRARY_PATH=jdk/jre/lib/i386/client:$LD_LIBRARY_PA
```

Invocation API Functions

- `jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)`

  initializes the Java virtual machine. The function returns `0` if successful, `JNI_ERR` on failure.

- `jint DestroyJavaVM(JavaVM* jvm)`

  destroys the virtual machine. Returns 0 on success, a negative number on failure. This function must be called through a virtual machine pointer, that is, `(*jvm)->DestroyJavaVM(jvm)`.

12.10 A COMPLETE EXAMPLE: ACCESSING THE WINDOWS REGISTRY

In this section, we describe a full, working example that covers everything we discussed in this chapter: using native methods with strings, arrays, objects, constructor calls, and error handling. We'll show you how to put a Java platform wrapper around a subset of the ordinary C-based APIs used to work with the Windows registry. Of course, the Windows registry being a Windowsspecific feature, such a program is inherently nonportable. For that reason, the standard Java library has no support for the registry,

http://www.rejoiceblog.com/
and it makes sense to use native methods to gain access to it.

12.10.1 Overview of the Windows Registry

The Windows registry is a data depository that holds configuration information for the Windows operating system and application programs. It provides a single point for administration and backup of system and application preferences. On the downside, the registry is also a single point of failure—if you mess up the registry, your computer could malfunction or even fail to boot!

We don't suggest that you use the registry to store configuration parameters for your Java programs. The Java preferences API is a better solution (see Volume I, Chapter 13 for more information). We'll simply use the registry to demonstrate how to wrap a nontrivial native API into a Java class.

The principal tool for inspecting the registry is the *registry editor*. Because of the potential for error by naive but enthusiastic users, there is no icon for launching the registry editor. Instead, start a DOS shell (or open the Start → Run dialog box) and type `regedit`. Figure 12.4 shows the registry editor in action.

Figure 12.4 The registry editor

The left side shows the keys, which are arranged in a tree structure. Note that each key

http://www.rejoiceblog.com/

starts with one of the `HKEY` nodes like

```
HKEY_CLASSES_ROOT

HKEY_CURRENT_USER

HKEY_LOCAL_MACHINE

. . .
```

The right side shows the name/value pairs associated with a particular key. For example, if you installed Java 11, the key

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment
```

contains a name/value pair such as

```
CurrentVersion="11.0_10"
```

In this case, the value is a string. The values can also be integers or arrays of bytes.

12.10.2 A Java Platform Interface for Accessing the Registry

We create a simple interface to access the registry from Java code, and then implement this interface with native code. Our interface allows only a few registry operations; to keep the code size down, we omitted some important operations such as adding, deleting, and enumerating keys. (It should be easy to add the remaining registry API functions.)

Even with the limited subset that we supply, you can

- Enumerate all names stored in a key
- Read the value stored with a name
- Set the value stored with a name

http://www.rejoiceblog.com/

Here is the Java class that encapsulates a registry key:

```
public class Win32RegKey

{

   public Win32RegKey(int theRoot, String thePath) { . . . }

   public Enumeration names() { . . . }

   public native Object getValue(String name);

   public native void setValue(String name, Object value);

   public static final int HKEY_CLASSES_ROOT = 0×80000000;

   public static final int HKEY_CURRENT_USER = 0×80000001;

   public static final int HKEY_LOCAL_MACHINE = 0×80000002;

   . . .

}
```

The `names` method returns an enumeration that holds all the names stored with the key. You can get at them with the familiar `hasMoreElements/nextElement` methods. The `getValue` method returns an object that is either a string, an `Integer` object, or a byte array. The value parameter of the `setValue` method must also be of one of these three types.

12.10.3 Implementation of Registry Access Functions as Native Methods

We need to implement three actions:

• Get the value of a name

• Set the value of a name

http://www.rejoiceblog.com/

• Iterate through the names of a key

Fortunately, you have seen essentially all the tools that are required, such as the conversion between Java strings and arrays and those of C. You also saw how to raise a Java exception in case something goes wrong.

Two issues make these native methods more complex than the preceding examples. The `getValue` and `setValue` methods deal with the type `Object`, which can be one of `String`, `Integer`, or `byte[]`. The enumeration object stores the state between successive calls to `hasMoreElements` and `nextElement`.

Let us first look at the `getValue` method. The method (shown in Listing 12.22) goes through the following steps:

1. Opens the registry key. To read their values, the registry API requires that keys be open.

2. Queries the type and size of the value associated with the name.

3. Reads the data into a buffer.

4. Calls `NewStringUTF` to create a new string with the value data if the type is `REG_SZ` (a string).

5. Invokes the `Integer` constructor if the type is `REG_DWORD` (a 32-bit integer).

6. Calls `NewByteArray` to create a new byte array, then `SetByteArrayRegion` to copy the value data into the byte array, if the type is `REG_BINARY`.

7. If the type is none of these or if an error occurred when an API function was called, throws an exception and releases all resources that had been acquired up to that point.

8. Closes the key and returns the object (`String`, `Integer`, or `byte[]`) that had been created.

As you can see, this example illustrates quite nicely how to generate Java objects of different types.

In this native method, coping with the generic return type is not difficult. The `jstring`, `jobject`, or `jarray` reference is simply returned as a `jobject`. However, the `setValue` method receives a reference to an `Object` and must determine the `Object`'s exact type to save the `Object` as a string, integer, or byte array. We can make

http://www.rejoiceblog.com/

this determination by querying the class of the `value` object, finding the class references for `java.lang.String`, `java.lang.Integer`, and `byte[]`, and comparing them with the `IsAssignableFrom` function.

If `class1` and `class2` are two class references, then the call

```
(*env)->IsAssignableFrom(env, class1, class2)
```

returns `JNI_TRUE` when `class1` and `class2` are the same class or when `class1` is a subclass of `class2`. In either case, references to objects of `class1` can be cast to `class2`. For example, when

```
(*env)->IsAssignableFrom(env, (*env)->GetObjectClass(env, val
```

is `true`, we know that `value` is a byte array.

Here is an overview of the steps in the `setValue` method:

1. Open the registry key for writing.

2. Find the type of the value to write.

3. Call `GetStringUTFChars` to get a pointer to the characters if the type is `String`.

4. Call the `intValue` method to get the integer stored in the wrapper object if the type is `Integer`.

5. Call `GetByteArrayElements` to get a pointer to the bytes if the type is `byte[]`.

6. Pass the data and length to the registry.

7. Close the key.

8. Release the pointer to the data if the type is `String` or `byte[]`.

Finally, let us turn to the native methods that enumerate keys. These are methods of the `Win32RegKeyNameEnumeration` class (see Listing 12.21). When the enumeration process starts, we must open the key. For the duration of the enumeration, we must

http://www.rejoiceblog.com/
retain the key handle—that is, the key handle must be stored with the enumeration object. The key handle is of type `DWORD` (a 32-bit quantity), so it can be stored in a Java integer. We store it in the `hkey` field of the enumeration class. When the enumeration starts, the field is initialized with `SetIntField`. Subsequent calls read the value with `GetIntField`.

In this example, we store three other data items with the enumeration object. When the enumeration first starts, we can query the registry for the count of name/value pairs and the length of the longest name, which we need so we can allocate C character arrays to hold the names. These values are stored in the `count` and `maxsize` fields of the enumeration object. Finally, the `index` field, initialized with `-1` to indicate the start of the enumeration, is set to `0` once the other instance fields are initialized, and is incremented after every enumeration step.

Let's walk through the native methods that support the enumeration. The `hasMoreElements` method is simple:

1. Retrieve the `index` and `count` fields.

2. If the index is `-1`, call the `startNameEnumeration` function, which opens the key, queries the count and maximum length, and initializes the `hkey`, `count`, `maxsize`, and `index` fields.

3. Return `JNI_TRUE` if `index` is less than `count`, and `JNI_FALSE` otherwise.

The `nextElement` method needs to work a little harder:

1. Retrieve the `index` and `count` fields.

2. If the index is `-1`, call the `startNameEnumeration` function, which opens the key, queries the count and maximum length, and initializes the `hkey`, `count`, `maxsize`, and `index` fields.

3. If `index` equals `count`, throw a `NoSuchElementException`.

4. Read the next name from the registry.

5. Increment `index`.

6. If `index` equals `count`, close the key.

http://www.rejoiceblog.com/

Before compiling, remember to run `javac -h` on both `Win32RegKey` and `Win32RegKeyNameEnumeration`. The complete command line for the Microsoft compiler is

```
cl -I jdk\include -I jdk\include\win32 -LD Win32RegKey.c advapi
```

With Cygwin, use

```
gcc -mno-cygwin -D __int64="long long" -I jdk\include -I jdk\in

     -I c:\cygwin\usr\include\w32api -shared -Wl,--add-stdcal

Win32RegKey.c
```

As the registry API is specific to Windows, this program will not work on other operating systems.

Listing 12.23 shows a program to test our new registry functions. We add three name/value pairs, a string, an integer, and a byte array to the key

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

We then enumerate all names of that key and retrieve their values. The program will print

```
Default user=Harry Hacker

Lucky number=13

Small primes=2 3 5 7 11 13
```

Although adding these name/value pairs to that key probably does no harm, you might want to use the registry editor to remove them after running this program.

http://www.rejoiceblog.com/

Listing 12.21 `win32reg/Win32RegKey.java`

```
import java.util.*;

/**
 * A Win32RegKey object can be used to get and set valu
 * @version 1.00 1997-07-01
 * @author Cay Horstmann
 */
public class Win32RegKey
{
  public static final int HKEY_CLASSES_ROOT = 0x80000000
  public static final int HKEY_CURRENT_USER = 0x80000001
  public static final int HKEY_LOCAL_MACHINE = 0x8000000
  public static final int HKEY_USERS = 0x80000003;
  public static final int HKEY_CURRENT_CONFIG = 0x800000
  public static final int HKEY_DYN_DATA = 0x80000006;

  private int root;
  private String path;
```

http://www.rejoiceblog.com/

```
19
20      /**
21       * Gets the value of a registry entry.
22       * @param name the entry name
23       * @return the associated value
24       */
25      public native Object getValue(String name);
26
27      /**
28       * Sets the value of a registry entry.
29       * @param name the entry name
30       * @param value the new value
31       */
32      public native void setValue(String name, Object value)
33
34      /**
35       * Construct a registry key object.
36       * @param theRoot one of HKEY_CLASSES_ROOT, HKEY_CURRE
37       * HKEY_CURRENT_CONFIG, HKEY_DYN_DATA
38       * @param thePath the registry key path
```

http://www.rejoiceblog.com/

```
     */
    public Win32RegKey(int theRoot, String thePath)
    {
        root = theRoot;
        path = thePath;
    }

    /**
     * Enumerates all names of registry entries under the p
     * @return an enumeration listing all entry names
     */
    public Enumeration<String> names()
    {
        return new Win32RegKeyNameEnumeration(root, path)
    }

    static
    {
        System.loadLibrary("Win32RegKey");
```

http://www.rejoiceblog.com/

```
    }
  }

  class Win32RegKeyNameEnumeration implements Enumeration<
  {
    public native String nextElement();
    public native boolean hasMoreElements();
    private int root;
    private String path;
    private int index = -1;
    private int hkey = 0;
    private int maxsize;
    private int count;

    Win32RegKeyNameEnumeration(int theRoot, String thePath
    {
       root = theRoot;
       path = thePath;
    }
  }
```

http://www.rejoiceblog.com/

```

class Win32RegKeyException extends RuntimeException
{
  public Win32RegKeyException()
  {
  }

  public Win32RegKeyException(String why)
  {
     super(why);
  }
}
```

Listing 12.22 `win32reg/Win32RegKey.c`

```
/**
   @version 1.00 1997-07-01
   @author Cay Horstmann
*/

```

http://www.rejoiceblog.com/

```
6   #include "Win32RegKey.h"
7   #include "Win32RegKeyNameEnumeration.h"
8   #include <string.h>
9   #include <stdlib.h>
10  #include <windows.h>
11
12  JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(JNIE
13  {
14    const char* cname;
15    jstring path;
16    const char* cpath;
17    HKEY hkey;
18    DWORD type;
19    DWORD size;
20    jclass this_class;
21    jfieldID id_root;
22    jfieldID id_path;
23    HKEY root;
24    jobject ret;
25    char* cret;
```

http://www.rejoiceblog.com/

```
26
27      /* get the class */
28      this_class = (*env)->GetObjectClass(env, this_obj);
29
30      /* get the field IDs */
31      id_root = (*env)->GetFieldID(env, this_class, "root",
32      id_path = (*env)->GetFieldID(env, this_class, "path",
33
34      /* get the fields */
35      root = (HKEY) (*env)->GetIntField(env, this_obj, id_ro
36      path = (jstring)(*env)->GetObjectField(env, this_obj,
37      cpath = (*env)->GetStringUTFChars(env, path, NULL);
38
39      /* open the registry key */
40      if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != E
41      {
42         (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32
43            "Open key failed");
44         (*env)->ReleaseStringUTFChars(env, path, cpath);
```

http://www.rejoiceblog.com/

```
        return NULL;
    }

    (*env)->ReleaseStringUTFChars(env, path, cpath);
    cname = (*env)->GetStringUTFChars(env, name, NULL);

    /* find the type and size of the value */
    if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &s
    {
        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32
            "Query value key failed");
        RegCloseKey(hkey);
        (*env)->ReleaseStringUTFChars(env, name, cname);
        return NULL;
     }

     /* get memory to hold the value */
     cret = (char*)malloc(size);

     /* read the value */
```

http://www.rejoiceblog.com/

```
    if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &
    {
        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win3
           "Query value key failed");
        free(cret);
        RegCloseKey(hkey);
        (*env)->ReleaseStringUTFChars(env, name, cname);
        return NULL;
    }

    /* depending on the type, store the value in a string
        integer or byte array */
    if (type == REG_SZ)
    {
        ret = (*env)->NewStringUTF(env, cret);
    }
    else if (type == REG_DWORD)
    {
        jclass class_Integer = (*env)->FindClass(env, "jav
```

http://www.rejoiceblog.com/

```
        /* get the method ID of the constructor */
        jmethodID id_Integer = (*env)->GetMethodID(env, cl
        int value = *(int*) cret;
        /* invoke the constructor */
        ret = (*env)->NewObject(env, class_Integer, id_Int
    }
    else if (type == REG_BINARY)
    {
        ret = (*env)->NewByteArray(env, size);
        (*env)->SetByteArrayRegion(env, (jarray) ret, 0, s
    }
    else
    {
        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win3
           "Unsupported value type");
        ret = NULL;
    }

    free(cret);
    RegCloseKey(hkey);
```

http://www.rejoiceblog.com/

```
    (*env)->ReleaseStringUTFChars(env, name, cname);

    return ret;
}

JNIEXPORT void JNICALL Java_Win32RegKey_setValue(JNIEnv
    jstring name, jobject value)
{
    const char* cname;
    jstring path;
    const char* cpath;
    HKEY hkey;
    DWORD type;
    DWORD size;
    jclass this_class;
    jclass class_value;
    jclass class_Integer;
    jfieldID id_root;
    jfieldID id_path;
```

http://www.rejoiceblog.com/

```
    HKEY root;

    const char* cvalue;

    int ivalue;

    /* get the class */

    this_class = (*env)->GetObjectClass(env, this_obj);

    /* get the field IDs */

    id_root = (*env)->GetFieldID(env, this_class, "root"

    id_path = (*env)->GetFieldID(env, this_class, "path"

    /* get the fields */

    root = (HKEY)(*env)->GetIntField(env, this_obj, id_r

    path = (jstring)(*env)->GetObjectField(env, this_obj

    cpath = (*env)->GetStringUTFChars(env, path, NULL);

    /* open the registry key */

    if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey) !

    {

        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win
```

http://www.rejoiceblog.com/

```
143            "Open key failed");
144          (*env)->ReleaseStringUTFChars(env, path, cpath);
145          return;
146       }
147
148       (*env)->ReleaseStringUTFChars(env, path, cpath);
149       cname = (*env)->GetStringUTFChars(env, name, NULL);
150
151       class_value = (*env)->GetObjectClass(env, value);
152       class_Integer = (*env)->FindClass(env, "java/lang/In
153       /* determine the type of the value object */
154       if ((*env)->IsAssignableFrom(env, class_value, (*env
155      {
156         /* it is a string--get a pointer to the characters
157         cvalue = (*env)->GetStringUTFChars(env, (jstring)
158         type = REG_SZ;
159         size = (*env)->GetStringLength(env, (jstring) valu
160      }
161      else if ((*env)->IsAssignableFrom(env, class_value, c
```

http://www.rejoiceblog.com/

```
    {
        /* it is an integer--call intValue to get the valu
        jmethodID id_intValue = (*env)->GetMethodID(env, c
        ivalue = (*env)->CallIntMethod(env, value, id_intV
        type = REG_DWORD;
        cvalue = (char*)&ivalue;
        size = 4;
    }
    else if ((*env)->IsAssignableFrom(env, class_value, (
    {
        /* it is a byte array--get a pointer to the bytes
        type = REG_BINARY;
        cvalue = (char*)(*env)->GetByteArrayElements(env,
        size = (*env)->GetArrayLength(env, (jarray) value)
    }
    else
    {
        /* we don't know how to handle this type */
        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win3
            "Unsupported value type");
```

http://www.rejoiceblog.com/

```
182         RegCloseKey(hkey);
183         (*env)->ReleaseStringUTFChars(env, name, cname);
184         return;
185     }
186
187     /* set the value */
188     if (RegSetValueEx(hkey, cname, 0, type, cvalue, size)
189     {
190         (*env)->ThrowNew(env, (*env)->FindClass(env, "Win3
191             "Set value failed");
192     }
193
194     RegCloseKey(hkey);
195     (*env)->ReleaseStringUTFChars(env, name, cname);
196
197     /* if the value was a string or byte array, release t
198     if (type == REG_SZ)
199     {
200         (*env)->ReleaseStringUTFChars(env, (jstring) value
```

http://www.rejoiceblog.com/

```
    }
    else if (type == REG_BINARY)
      {
          (*env)->ReleaseByteArrayElements(env, (jarray)
      }
}

/* helper function to start enumeration of names */
static int startNameEnumeration(JNIEnv* env, jobject th
{
    jfieldID id_index;
    jfieldID id_count;
    jfieldID id_root;
    jfieldID id_path;
    jfieldID id_hkey;
    jfieldID id_maxsize;

    HKEY root;
    jstring path;
    const char* cpath;
```

http://www.rejoiceblog.com/

```
221     HKEY hkey;
222     DWORD maxsize = 0;
223     DWORD count = 0;
224
225     /* get the field IDs */
226     id_root = (*env)->GetFieldID(env, this_class, "root"
227     id_path = (*env)->GetFieldID(env, this_class, "path"
228     id_hkey = (*env)->GetFieldID(env, this_class, "hkey"
229     id_maxsize = (*env)->GetFieldID(env, this_class, "ma
230     id_index = (*env)->GetFieldID(env, this_class, "inde
231     id_count = (*env)->GetFieldID(env, this_class, "coun
232
233     /* get the field values */
234     root = (HKEY)(*env)->GetIntField(env, this_obj, id_r
235     path = (jstring)(*env)->GetObjectField(env, this_obj
236     cpath = (*env)->GetStringUTFChars(env, path, NULL);
237
238     /* open the registry key */
239     if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) !=
```

http://www.rejoiceblog.com/

```
    {
        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win
          "Open key failed");
        (*env)->ReleaseStringUTFChars(env, path, cpath);
        return -1;
    }
    (*env)->ReleaseStringUTFChars(env, path, cpath);

    /* query count and max length of names */
    if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, NU
            NULL, NULL, NULL) != ERROR_SUCCESS)
    {
         (*env)->ThrowNew(env, (*env)->FindClass(env, "Wi
            "Query info key failed");
         RegCloseKey(hkey);
         return -1;
    }

    /* set the field values */
    (*env)->SetIntField(env, this_obj,   id_hkey, (DWORD
```

http://www.rejoiceblog.com/

```
    (*env)->SetIntField(env, this_obj,   id_maxsize, max
    (*env)->SetIntField(env, this_obj,   id_index, 0);
    (*env)->SetIntField(env, this_obj,   id_count, count
    return count;
}

JNIEXPORT jboolean JNICALL Java_Win32RegKeyNameEnumerat
    jobject this_obj)
{   jclass this_class;
    jfieldID id_index;
    jfieldID id_count;
    int index;
    int count;
    /* get the class */
    this_class = (*env)->GetObjectClass(env, this_obj);

    /* get the field IDs */
    id_index = (*env)->GetFieldID(env, this_class, "inde
    id_count = (*env)->GetFieldID(env, this_class, "coun
```

http://www.rejoiceblog.com/

```
    index = (*env)->GetIntField(env, this_obj, id_index)
    if (index == -1) /* first time */
    {
        count = startNameEnumeration(env, this_obj, this_
        index = 0;
    }
    else
        count = (*env)->GetIntField(env, this_obj, id_cou
    return index < count;
}

JNIEXPORT jobject JNICALL Java_Win32RegKeyNameEnumerati
    jobject this_obj)
{
    jclass this_class;
    jfieldID id_index;
    jfieldID id_hkey;
    jfieldID id_count;
    jfieldID id_maxsize;
```

http://www.rejoiceblog.com/

```
299
300     HKEY hkey;
301     int index;
302     int count;
303     DWORD maxsize;
304
305     char* cret;
306     jstring ret;
307
308     /* get the class */
309     this_class = (*env)->GetObjectClass(env, this_obj);
310
311     /* get the field IDs */
312     id_index = (*env)->GetFieldID(env, this_class, "inde
313     id_count = (*env)->GetFieldID(env, this_class, "coun
314     id_hkey = (*env)->GetFieldID(env, this_class, "hkey"
315     id_maxsize = (*env)->GetFieldID(env, this_class, "ma
316
317     index = (*env)->GetIntField(env, this_obj, id_index)
```

http://www.rejoiceblog.com/

```
318      if (index == -1) /* first time */
319      {
320         count = startNameEnumeration(env, this_obj, this_
321         index = 0;
322      }
323      else
324         count = (*env)->GetIntField(env, this_obj, id_cou
325
326      if (index >= count) /* already at end */
327      {
328         (*env)->ThrowNew(env, (*env)->FindClass(env, "jav
329            "past end of enumeration");
330         return NULL;
331      }
332
333      maxsize = (*env)->GetIntField(env, this_obj, id_maxs
334      hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_h
335      cret = (char*)malloc(maxsize);
336
337      /* find the next name */
```

http://www.rejoiceblog.com/

```
338      if (RegEnumValue(hkey, index, cret, &maxsize, NULL, 
339      {
340         (*env)->ThrowNew(env, (*env)->FindClass(env, "Win
341            "Enum value failed");
342         free(cret);
343         RegCloseKey(hkey);
344         (*env)->SetIntField(env, this_obj, id_index, coun
345         return NULL;
346      }
347
348       ret = (*env)->NewStringUTF(env, cret);
349       free(cret);
350
351       /* increment index */
352       index++;
353       (*env)->SetIntField(env, this_obj, id_index, index)
354
355       if (index == count) /* at end */
356       {
```

http://www.rejoiceblog.com/

```
            RegCloseKey(hkey);

        }


         return ret;

    }


```

Listing 12.23 `win32reg/Win32RegKeyTest.java`

```
import java.util.*;

/**
    @version 1.03 2018-05-01
    @author Cay Horstmann
*/
public class Win32RegKeyTest
{
   public static void main(String[] args)
   {
      var key = new Win32RegKey(
```

http://www.rejoiceblog.com/

```
            Win32RegKey.HKEY_CURRENT_USER, "Software\\Java

        key.setValue("Default user", "Harry Hacker");
        key.setValue("Lucky number", new Integer(13));
        key.setValue("Small primes", new byte[] { 2, 3, 5

        Enumeration<String> e = key.names();

        while (e.hasMoreElements())
        {
            String name = e.nextElement();
            System.out.print(name + "=");

            Object value = key.getValue(name);

            if (value instanceof byte[])
                for (byte b : (byte[]) value) System.out.pr
            else
                System.out.print(value);
```

http://www.rejoiceblog.com/

```
31
32            System.out.println();
33          }
34        }
35    }
```

Type Inquiry Functions

- `jboolean IsAssignableFrom(JNIEnv *env, jclass cl1, jclass cl2)`

returns `JNI_TRUE` if objects of the first class can be assigned to objects of the second class, and `JNI_FALSE` otherwise. This tests if the classes are the same, or `cl1` is a subclass of `cl2`, or `cl2` represents an interface implemented by `cl1` or one of its superclasses.

- `jclass GetSuperclass(JNIEnv *env, jclass cl)`

returns the superclass of a class. If `cl` represents the class `Object` or an interface, returns `NULL`.

You have now reached the end of the second volume of *Core Java*, completing a long journey in which you encountered many advanced APIs. We started out with topics that every Java programmer needs to know: streams, XML, networking, databases, and internationalization. We concluded with very technical chapters on security, annotation processing, advanced graphics, and native methods. We hope that you enjoyed your tour through the vast breadth of the Java APIs, and that you will be able to apply your newly gained knowledge in your projects.