Updated through Java 21

Core Java for the Impatient

Fourth Edition

Cay S. Horstmann

FREE SAMPLE CHAPTER

This page intentionally left blank

Core Java for the Impatient

Fourth Edition

Cay S. Horstmann

Addison-Wesley
Hoboken, New Jersey

Cover illustration by Morphart Creation / Shutterstock
Figures 1.1, 1.3: Microsoft Corporation
Figure 1.2: Eclipse Foundation
Figures 1.4, 1.5, 1.11: Oracle Corporation

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

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

For questions about sales outside the United States, please contact intlcs@pearson.com.

Please contact us with concerns about any potential bias at pearson.com/report-bias.html.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024947133

ISBN-13: 978-0-13-540454-6
ISBN-10: 0-13-540454-1

This page intentionally left blank

To Chi—the most patient person in my life.

This page intentionally left blank

Table of Contents

This page intentionally left blank

Preface

Java has seen many changes since its initial release in 1996. The classic book, *Core Java*, covers, in meticulous detail, not just the language but all core libraries and a multitude of changes between versions, spanning two volumes and over 2,000 pages. However, if you just want to be productive with modern Java, there is a much faster, easier pathway for learning the language and core libraries. In this book, I don’t retrace history and don’t dwell on features of past versions. I show you the good parts of Java as it exists today, so you can put your knowledge to work quickly.

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

Assuming you are proficient in some other programming language, such as Python, C++, JavaScript, Swift, PHP, or Ruby, with this book you will learn how to become a competent Java programmer. I cover all aspects of Java that a developer needs to know today, including the powerful concepts of lambda expressions and streams, as well as modern constructs such as records and pattern matching.

This book is fully updated to Java 21. It uses modern features and does not dwell on historical or obsolete constructs. Preview features that may make it to the language in the future are not covered either.

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

Traditionally, books on Java have focused on user interface programming, but nowadays, few developers produce user interfaces on desktop computers. You will be able to use this book effectively without being distracted by lengthy GUI code.

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

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

If you find errors or have suggestions for improvement, please visit `http://horstmann.com/javaimpatient/bugs.html` and leave a comment.

Tip: Download the runnable code examples that complement this book at http://horstmann.com/javaimpatient/bugs.html.

Acknowledgments

My thanks go, as always, to my editor Greg Doench, who enthusiastically supported the vision of a short book that gives a fresh introduction to Java. My special gratitude goes to the excellent team of reviewers for this and previous editions who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Gail Anderson, Paul Anderson, Marcus Biel, Jean-Claude Brantschen, Brian Goetz, Mark Lawrence, Doug Lea, Ron Mak, Simon Ritter, Yoshiki Shibata, Clovis Tondo, and Christian Ullenboom.

I wrote the book using HTML and CSS, and Prince (`https://princexml.com`) turned it into PDF—a workflow that I highly recommend.

Cay Horstmann
Düsseldorf, Germany
August 2024

CHAPTER 9

Processing Input and Output

In this chapter, you will learn how to work with files, directories, and web pages, and how to read and write data in binary and text format. You will also find a discussion of regular expressions, which can be useful for processing input. (I couldn't think of a better place to handle that topic, and apparently neither could the Java developers—when the regular expression API specification was proposed, it was attached to the specification request for "new I/O" features.) Finally, this chapter shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data.

The key points of this chapter are:

1. An `InputStream` is a source of bytes, and an `OutputStream` is a destination for bytes.
2. A `Reader` reads characters, and a `Writer` writes them. Be sure to specify a character encoding.
3. The `Files` class has convenience methods for reading all bytes or lines of a file.
4. The `DataInput` and `DataOutput` interfaces have methods for writing numbers in binary format.
5. Use a `RandomAccessFile` or a memory-mapped file for random access.
6. A `Path` is an absolute or relative sequence of path components in a file system. Paths can be combined (or "resolved").
7. Use the methods of the `Files` class to copy, move, or delete files and to recursively walk through a directory tree.
8. To read or update a ZIP file, use a ZIP file system.
9. You can read the contents of a web page with the `URL` class. To read metadata or write data, use the `URLConnection` class.
10. With the `Pattern` and `Matcher` classes, you can find all matches of a regular expression in a string, as well as the captured groups for each match.
11. The serialization mechanism can save and restore any object implementing the `Serializable` interface, provided its instance variables are also serializable.

9.1. Input/Output Streams, Readers, and Writers

In the Java API, a source from which one can read bytes is called an *input stream*. The bytes can come from a file, a network connection, or an array in memory. (These streams are unrelated to the streams of Chapter 8.) Similarly, a destination for bytes is an *output stream*. In contrast, *readers* and *writers* consume and produce sequences of *characters*. In the following sections, you will learn how to read and write bytes and characters.

9.1.1. Obtaining Streams

The easiest way to obtain a stream from a file is with the static methods

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

Here, `path` is an instance of the `Path` class that is covered in Section 9.2.1. It describes a path in a file system.

If you have an `URL` object, you can read its contents from the input stream returned by the `openStream` method. (The `URL` constructors are deprecated, and you should create an `URL` instance as shown here.)

```
var url = URI.create("https://horstmann.com/index.html").toURL();
InputStream in = url.openStream();
```

Section 9.3 shows how to send data to a web server.

The `ByteArrayInputStream` class lets you read from an array of bytes.

```
byte[] bytes = ...;
var in = new ByteArrayInputStream(bytes);
Read from in
```

Conversely, to send output to a byte array, use a `ByteArrayOutputStream`:

```
var out = new ByteArrayOutputStream();
Write to out
byte[] bytes = out.toByteArray();
```

9.1.2. Reading Bytes

The `InputStream` class has a method to read a single byte:

```
InputStream in = ...;
int b = in.read();
```

This method either returns the byte as an integer between `0` and `255`, or returns `-1` if the end of input has been reached.

Caution: The Java `byte` type has values between `-128` and `127`. You can cast the returned value into a `byte` *after* you have checked that it is not `-1`.

More commonly, you will want to read the bytes in bulk. The most convenient method is the `readAllBytes` method that simply reads all bytes from the stream into a byte array:

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

Tip: If you want to read all bytes from a file, call the convenience method

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

If you want to read some, but not all bytes, provide a byte array and call the `readNBytes` method:

```
var bytes = new byte[len];
int bytesRead = in.readNBytes(bytes, offset, n);
```

The method reads until either `n` bytes are read or no further input is available, and returns the actual number of bytes read. If no input was available at all, the methods return `-1`.

Note: There is also a `read(byte[], int, int)` method whose description seems exactly like `readNBytes`. The difference is that the `read` method only attempts to read the bytes and returns immediately with a lower count if it fails. The `readNBytes` method keeps calling `read` until all requested bytes have been obtained or `read` returns `-1`.

Finally, you can skip bytes:

```
long bytesToSkip = ...;
in.skipNBytes(bytesToSkip);
```

9.1.3. Writing Bytes

The `write` methods of an `OutputStream` can write individual bytes and byte arrays.

```
OutputStream out = ...;
int b = ...;
out.write(b);
byte[] bytes = ...;
out.write(bytes);
out.write(bytes, start, length);
```

When you are done writing a stream, you must *close* it in order to commit any buffered output. This is best done with a try-with-resources statement:

```
try (OutputStream out = ...) {
    out.write(bytes);
}
```

If you need to copy an input stream to an output stream, use the `InputStream.transferTo` method:

```
try (InputStream in = ...; OutputStream out = ...) {
    in.transferTo(out);
}
```

Both streams need to be closed after the call to `transferTo`. It is best to use a try-with-resources statement, as in the code example.

To write a file to an `OutputStream`, call

```
Files.copy(path, out);
```

Conversely, to save an `InputStream` to a file, call

```
Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
```

9.1.4. Character Encodings

Input and output streams are for sequences of bytes, but in many cases you will work with text—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 9.1). 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 9.1: 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	$11110a_{20}a_{19}a_{18}$ $10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$

A less common encoding is UTF-16, which encodes each Unicode code point into one or two 16-bit values (see Table 9.2). 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.

Table 9.2: UTF-16 Encoding

Character range	Encoding
`0...FFFF`	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$
`10000...10FFFF`	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}$ $110111a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$ where $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

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.

Because UTF-8 is so common, it has become the default encoding since Java 18. Previously, the default encoding was the *native encoding*—the character encoding that is preferred by the operating system of the computer running your program. On Windows, that is generally not UTF-8. If you are using an older version of Java, or if you are working with text in an encoding other than UTF-8, you need to explicitly specify the encoding.

Note: The native 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.

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");
```

You use the `Charset` object to specify a character encoding. For example, you can turn an array of bytes into a string as

```
var contents = new String(bytes, StandardCharsets.ISO_8859_1);
```

9.1.5. Text Input

To read text input, use a `Reader`. You can obtain a `Reader` from any input stream with the `InputStreamReader` adapter:

```
InputStream inStream = ...;
var in = new InputStreamReader(inStream, charset);
```

If you want to process the input one UTF-16 code unit at a time, you can call the `read` method:

```
int ch = in.read();
```

The method returns a code unit between `0` and `65536`, or `-1` at the end of input.

That is not very convenient. Here are several alternatives.

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

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

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

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

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

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

Note: If an `IOException` occurs as the stream fetches the lines, that exception is wrapped into an `UncheckedIOException` which is thrown out of the stream operation. This subterfuge is necessary because stream operations are not declared to throw any checked exceptions.

To read numbers or words from a file, use a `Scanner`, as you have seen in Chapter 1. For example,

```
var in = new Scanner(path);
while (in.hasNextDouble()) {
    double value = in.nextDouble();
    ...
}
```

Tip: To read alphabetic words, set the scanner's delimiter to a regular expression that is the complement of what you want to accept as a token. For example, after calling

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

the scanner reads in letters since any sequence of nonletters is a delimiter. See Section 9.4.1 for the regular expression syntax.

You can then obtain a stream of all words as

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

If your input does not come from a file, wrap the `InputStream` into a `BufferedReader`:

```
try (var reader = new BufferedReader(new InputStreamReader(url.openStream()))) {
    Stream<String> lines = reader.lines();
    ...
}
```

A `BufferedReader` reads input in chunks for efficiency. (Oddly, this is not an option for basic readers.) It has methods `readLine` to read a single line and `lines` to yield a stream of lines.

If a method asks for a `Reader` and you want it to read from a file, call `Files.newBufferedReader(path, charset)`.

9.1.6. Text Output

To write text, use a `Writer`. With the `write` method, you can write strings. You can turn any output stream into a `Writer`:

```
OutputStream outStream = ...;
var out = new OutputStreamWriter(outStream, charset);
out.write(str);
```

To get a writer for a file, use

```
Writer out = Files.newBufferedWriter(path, charset);
```

It is more convenient to use a `PrintWriter`, which has the `print`, `println`, and `printf` that you have always used with `System.out`. Using those methods, you can print numbers and use formatted output.

If you write to a file, construct a `PrintWriter` like this:

```
var out = new PrintWriter(Files.newBufferedWriter(path, charset));
```

If you write to another stream, use

```
var out = new PrintWriter(new OutputStreamWriter(outStream, charset));
```

Note: `System.out` is an instance of `PrintStream`, not `PrintWriter`. This is a relic from the earliest days of Java. However, the `print`, `println`, and `printf` methods work the same way for the `PrintStream` and `PrintWriter` classes, using a character encoding for turning characters into bytes.

If you already have the text to write in a string, call

```
String content = ...;
Files.writeString(path, content, charset);
```

or

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

Here, `lines` can be a `Collection<String>`, or even more generally, an `Iterable<? extends CharSequence>`.

To append to a file, use

```
Files.writeString(path, charset, StandardOpenOption.APPEND);
Files.write(path, lines, charset, StandardOpenOption.APPEND);
```

Caution: When writing text with a partial character set such as ISO 8859-1, any unmappable characters are silently changed to a “replacement”—in most cases, either the ? character or the Unicode replacement character `U+FFFD`.

Sometimes, a library method wants a `Writer` to write output. If you want to capture that output in a string, hand it a `StringWriter`. Or, if it wants a `PrintWriter`, wrap the `StringWriter` like this:

```
var writer = new StringWriter();
throwable.printStackTrace(new PrintWriter(writer));
String stackTrace = writer.toString();
```

9.1.7. Reading Character Input

If you read a file with a structured format such as JSON or XML, you will use a parser that someone wrote who understands the fiddly details of that format. Such a parser typically reads a character at a time.

In the uncommon case that you need to write such a parser, use a `BufferedReader` for efficiency. Keep calling its `read` method, which yields a `char` value or -1 at the end of input. The reader converts the encoding of the input stream into UTF-16.

If you want to process Unicode code points, you need to handle the UTF-16 encoding. Here is how to read one code point:

```
int ch = reader.read();
if (ch != -1)
{
   int codePoint;
   if (Character.isHighSurrogate((char) ch))
   {
      int ch2 = reader.read();
      if (Character.isLowSurrogate((char) ch2))
         codePoint = Character.toCodePoint(ch, ch2);
      else
         throw new MalformedInputException();
   }
   else
      codePoint = ch;
}
```

The `Character` class contains methods to tell whether a particular code point has a given property. For example,

```
Character.isLetter(codePoint)
```

returns `true` if `codePoint` is a letter in some language. Here are some other classification methods:

```
isUpperCase
isLowerCase
isDigit
isSpaceChar
isEmoji
```

These methods use the rules of the Unicode standard. Others refer to the rules of the Java language:

```
isJavaIdentifierStart
isJavaIdentifierPart
isWhitespace
```

After analyzing the code points, you often need to store them in strings, converting them back to UTF-16. The `appendCodePoint` method of the `StringBuilder` class turns a code point into one or two `char` values which are appended to the builder.

9.1.8. Reading and Writing Binary Data

The `DataInput` interface declares the following methods for reading a number, a character, a `boolean` value, or a string in binary format:

```
byte readByte()
int readUnsignedByte()
char readChar()
short readShort()
int readUnsignedShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
void readFully(byte[] b)
```

The `DataOutput` interface declares corresponding `write` methods.

Note: These methods read and write numbers in big-endian format.

Caution: There are also `readUTF`/`writeUTF` methods that use a "modified UTF-8" format. These methods are *not* compatible with regular UTF-8, and are only useful for JVM internals.

The advantage of binary I/O is that it is fixed width and efficient. For example, `writeInt` always writes an integer as a big-endian 4-byte binary quantity regardless of the number of digits. The space needed is the same for each value of a given type, which speeds up random access. Also, reading binary data is faster than parsing text. The main drawback is that the resulting files cannot be easily inspected in a text editor.

You can use the `DataInputStream` and `DataOutputStream` adapters with any stream. For example,

```
DataInput in = new DataInputStream(Files.newInputStream(path));
DataOutput out = new DataOutputStream(Files.newOutputStream(path));
```

9.1.9. Random-Access Files

The `RandomAccessFile` class lets you read or write data anywhere in a file. 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. For example,

```
var file = new RandomAccessFile(path.toString(), "rw");
```

A random-access file has a *file pointer* that indicates the position of the next byte to be read or written. The `seek` method sets 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 (which you can obtain with the `length` method). The `getFilePointer` method returns the current position of the file pointer.

The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces. To read and write numbers from a random-access file, use methods such as `readInt`/`writeInt` that you saw in the preceding section. For example,

```
int value = file.readInt();
file.seek(file.getFilePointer() - 4);
file.writeInt(value + 1);
```

9.1.10. Memory-Mapped Files

Memory-mapped files provide another, very efficient approach for random access that works well for very large files. However, the API for data access is completely different from that of input/output streams. First, get a *channel* to the file:

```
FileChannel channel = FileChannel.open(path,
    StandardOpenOption.READ, StandardOpenOption.WRITE)
```

Then, map an area of the file (or, if it is not too large, the entire file) into memory:

```
ByteBuffer buffer = channel.map(FileChannel.MapMode.READ_WRITE,
    0, channel.size());
```

Use methods `get`, `getInt`, `getDouble`, and so on to read values, and the equivalent `put` methods to write values.

```
int offset = ...;
int value = buffer.getInt(offset);
buffer.put(offset, value + 1);
```

At some point, and certainly when the channel is closed, these changes are written back to the file.

Note: By default, the methods for reading and writing numbers use big-endian byte order. You can change the byte order with the command

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

9.1.11. File Locking

When multiple simultaneously executing programs modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem.

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 channel = FileChannel.open(path, StandardOpenOption.WRITE);
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 lock or the channel is closed. It is best to use a try-with-resources statement:

```
try (FileLock lock = channel.lock()) {
    ...
}
```

9.2. Paths, Files, and Directories

You have already seen `Path` objects for specifying file paths. In the following sections, you will see how to manipulate these objects and how to work with files and directories.

9.2.1. 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 we are running on a Unix-like file system.

```
Path absolute = Path.of("/", "home", "cay");
Path relative = Path.of("myapp", "conf", "user.properties");
```

The static `Path.of` 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.

You can also provide a string with separators to the `Path.of` method:

```
Path homeDirectory = Path.of("/home/cay");
```

Note: A `Path` object does not have to correspond to a file that actually exists. It is merely an abstract sequence of names. To create a file, first make a path, then call a method to create the corresponding file—see Section 9.2.2.

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`.
- if `q` does not have a root, then the result is obtained by joining `p` and `q`.
- Otherwise, the result depends on the rules of the file system.

For example, suppose your application needs to find its configuration file relative to the home directory. Here is how you can combine the paths:

```
Path workPath = homeDirectory.resolve("myapp/work");
    // Same as homeDirectory.resolve(Path.of("myapp/work"));
```

There is a convenience method `resolveSibling` that resolves against a path's parent, yielding a sibling path. For example, if `workPath` is `/home/cay/myapp/work`, the call

```
Path tempPath = workPath.resolveSibling("temp");
```

yields `/home/cay/myapp/temp`.

The opposite of `resolve` is `relativize`. The call `p.relativize(r)` yields the path `q` which, when resolved with `p`, yields `r`. For example,

```
Path.of("/home/cay").relativize(Path.of("/home/fred/myapp"))
```

yields `../fred/myapp`, assuming we have a file system that uses `..` to denote the parent directory.

The `normalize` method removes any redundant `.` and `..` components (or whatever the file system may deem redundant). For example, normalizing the path `/home/cay/../fred/./myapp` yields `/home/fred/myapp`.

The `toAbsolutePath` method yields the absolute path of a given path. If the path is not already absolute, it is resolved against the *working directory*—that is, the directory of the process in which the JVM was invoked. For example, if you launched a Java program from `/home/cay/myapp`, then `Path.of("config").toAbsolutePath()` returns `/home/cay/myapp/config`.

Note: You can obtain the working directory by a call to `System.getProperty("user.dir")`.

The `Path` interface has methods for taking paths apart and combining them with other paths. This code sample shows some of the most useful ones:

```
Path p = Path.of("/home", "cay", "myapp.properties");
Path parent = p.getParent(); // The path /home/cay
Path file = p.getFileName(); // The last element, myapp.properties
Path root = p.getRoot(); // The initial segment / (null for a relative path)
Path first = p.getName(0); // The first element
Path dir = p.subpath(1, p.getNameCount());
    // All but the first element, cay/myapp.properties
```

The `Path` interface extends the `Iterable<Path>` element, so you can iterate over the name components of a `Path` with an enhanced `for` loop:

```
for (Path component : path) {
    ...
}
```

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.

9.2.2. Creating Files and Directories

To create a new directory, call

```
Files.createDirectory(path);
```

All but the last component in the path must already exist. To create intermediate directories as well, use

```
Files.createDirectories(path);
```

You can create an empty file with

```
Files.createFile(path);
```

The call throws an exception if the file already exists. The checks for existence and the creation are atomic. If the file doesn't exist, it is created before anyone else has a chance to do the same.

The call `Files.exists(path)` checks whether the given file or directory exists. To test whether it is a directory or a "regular" file (that is, with data in it, not something like a directory or symbolic link), call the static methods `isDirectory` and `isRegularFile` of the `Files` class.

There are convenience methods for creating a temporary file or directory in a given or system-specific location.

```
Path tempFile = Files.createTempFile(dir, prefix, suffix);
Path tempFile = Files.createTempFile(prefix, suffix);
Path tempDir = Files.createTempDirectory(dir, prefix);
Path tempDir = 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`.

9.2.3. Copying, Moving, and Deleting Files

To copy a file from one location to another, simply call

```
Files.copy(fromPath, toPath);
```

To move the file instead, call

```
Files.move(fromPath, toPath);
```

You can also use this command to move an empty directory.

The copy or move will fail if the target exists. If you want to overwrite an existing target, use the `REPLACE_EXISTING` option. If you want to copy all file attributes, use the `COPY_ATTRIBUTES` option. You can supply both like this:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES);
```

You can specify that a move should be atomic. Then you are assured that either the move completed successfully, or the source continues to be present. Use the `ATOMIC_MOVE` option:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

See Table 9.3 for a summary of the options that are available for file operations.

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.

Table 9.3: 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.

Option	Description
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.

9.2.4. 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-with-resources block:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
    ...
}
```

The list method does not enter subdirectories. To process all descendants of a directory, use the Files.walk method instead.

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {
    // Contains all descendants, visited in depth-first order
}
```

Here is a sample traversal of the unzipped src.zip tree:

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
...
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
...
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
...
```

As you can see, whenever the traversal yields a directory, it is entered before continuing with its siblings.

You can limit the depth of the tree that you want to visit by calling `Files.walk(pathToRoot, depth)`. Both `walk` methods have a varargs parameter of type `FileVisitOption...`, but there is only one option you can supply: `FOLLOW_LINKS` to follow symbolic links.

Note: If you filter the paths returned by `walk` and your filter criterion involves the file attributes stored with a directory, such as size, creation time, or type (file, directory, symbolic link), then use the `find` method instead of `walk`. Call that method with a predicate function that accepts a path and a `BasicFileAttributes` object. The only advantage is efficiency. Since the directory is being read anyway, the attributes are readily available.

This code fragment uses the `Files.walk` method to copy one directory to another:

```
Files.walk(source).forEach(p -> {
    try {
        Path q = target.resolve(source.relativize(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    } catch (IOException ex) {
```

```
        throw new UncheckedIOException(ex);
    }
});
```

Unfortunately, you cannot easily use the `Files.walk` method to delete a tree of directories since you need to first visit the children before deleting the parent. In that case, use the `walkFileTree` method. It requires an instance of the `FileVisitor` interface. Here is when the file visitor gets notified:

1. Before a directory is processed:

   ```
   FileVisitResult preVisitDirectory(T dir, IOException ex)
   ```

2. When a file is encountered:

   ```
   FileVisitResult visitFile(T path, BasicFileAttributes attrs)
   ```

3. When an exception occurs in the `visitFile` method:

   ```
   FileVisitResult visitFileFailed(T path, IOException ex)
   ```

4. After a directory is processed:

   ```
   FileVisitResult postVisitDirectory(T dir, IOException ex)
   ```

In each case, the notification method returns one of the following results:

- 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.

The `SimpleFileVisitor` class implements this interface, continuing the iteration at each point and rethrowing any exceptions.

Here is how you can delete a directory tree:

```
Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
    public FileVisitResult visitFile(Path file,
            BasicFileAttributes attrs) throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
```

```
    public FileVisitResult postVisitDirectory(Path dir,
            IOException ex) throws IOException {
        if (ex != null) throw ex;
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
    }
});
```

Caution: The `Files.walk` method throws an exception if any of the subdirectories are not readable. If you only want to visit readable directories, use the `walkFileTree` method.

9.2.5. ZIP File Systems

The `Paths` class looks up paths in the default file system—the files on the user's local disk. You can have other file systems. One of the more useful ones is a ZIP file system. If `zipname` is the name of a ZIP file, then the call

```
FileSystem zipfs = FileSystems.newFileSystem(Path.of(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(zipfs.getPath(sourceName), targetPath);
```

Here, `zipfs.getPath` is the analog of `Path.of` for an arbitrary file system.

To list all files in a ZIP archive, walk the file tree:

```
Files.walk(zipfs.getPath("/")).forEach(p -> {
    Process p
});
```

You have to work a bit harder to create a new ZIP file. Here is the magic incantation:

```
Path zipPath = Path.of("myfile.zip");
var uri = URI.create("jar:" + zipPath.toUri());
    // Constructs the URI jar:file://myfile.zip
try (FileSystem zipfs = FileSystems.newFileSystem(uri,
        Collections.singletonMap("create", "true"))) {
    // To add files, copy them into the ZIP file system
    Files.copy(sourcePath, zipfs.getPath("/").resolve(targetPath));
}
```

Note: There is an older API for working with ZIP archives, with classes `ZipInputStream` and `ZipOutputStream`, but it's not as easy to use as the one described in this section.

9.3. HTTP Connections

You can read from a URL by using the input stream returned from `URL.getInputStream` method. However, if you want additional information about a web resource, or if you want to write data, you need more control over the process than the `URL` class provides. 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 the following sections, I provide a cookbook for using the `HttpURLConnection` class, and then give an overview of the API.

9.3.1. The `URLConnection` and `HttpURLConnection` Classes

To use the `URLConnection` class, follow these steps:

1. Get an `URLConnection` object:

   ```
   URLConnection connection = url.openConnection();
   ```

   For an HTTP URL, the returned object is actually an instance of `HttpURLConnection`.
2. If desired, set request properties:

   ```
   connection.setRequestProperty("Accept-Charset", "UTF-8, ISO-8859-1");
   ```

   If a key has multiple values, separate them by commas.
3. To send data to the server, call

   ```
   connection.setDoOutput(true);
   try (OutputStream out = connection.getOutputStream()) {
       // Write to out
   }
   ```

4. If you want to read the response headers and you haven't called `getOutputStream`, call

   ```
   connection.connect();
   ```

   Then query the header information:

```
Map<String, List<String>> headers = connection.getHeaderFields();
```

For each key, you get a list of values since there may be multiple header fields with the same key.

5. Read the response:

```
try (InputStream in = connection.getInputStream()) {
    // Read from in
}
```

A common use case is to post form data. The `URLConnection` class automatically sets the content type to `application/x-www-form-urlencoded` when writing data to a HTTP URL, but you need to encode the name/value pairs:

```
URL url = ...;
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
try (var out = new OutputStreamWriter(
        connection.getOutputStream())) {
    Map<String, String> postData = ...;
    boolean first = true;
    for (Map.Entry<String, String> entry : postData.entrySet()) {
        if (first) first = false;
        else out.write("&");
        out.write(URLEncoder.encode(entry.getKey(), "UTF-8"));
        out.write("=");
        out.write(URLEncoder.encode(entry.getValue(), "UTF-8"));
    }
}
try (InputStream in = connection.getInputStream()) {
    ...
}
```

9.3.2. The HTTP Client API

The HTTP client API provides another mechanism for connecting to a web server which is simpler than the `URLConnection` class with its rather fussy set of stages. More importantly, the implementation supports HTTP/2.

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, use a builder API like this:

```
HttpClient client = HttpClient.newBuilder()
    .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.

Follow the same pattern for formulating requests. Here is a `GET` request:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://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).

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.BodyHandlers.ofString()`, like this:

```
HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());
```

The `HttpResponse` class is a template whose type 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 a file. One can hope that eventually the JDK will support JSON and provide a JSON handler.

With a `POST` request, you similarly 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. Again, one can hope that the library designers will wake up to the reality that most `POST` requests involve form data, file uploads, or JSON objects, and provide appropriate publishers.

Nowadays, the most common `POST` request body contains JSON, which you need to convert to a string. Then you can form the following request:

```
HttpRequest request = HttpRequest.newBuilder()
   .uri(URI.create(urlString))
   .header("Content-Type", "application/json")
   .POST(HttpRequest.BodyPublishers.ofString(jsonString))
   .build();
```

The book's companion code has examples for posting form data and file uploads.

The `HttpRequest.Builder` class also has build methods for the less common `PUT`, `DELETE`, and `HEAD` requests.

Java 16 adds a builder for filtering the headers of an existing `HttpRequest`. You provide the request and a function that receives the header names and values, returning `true` for those that should be retained. For example, here we modify the content type:

```
HttpRequest request2 = HttpRequest.newBuilder(request,
        (name, value) -> !name.equalsIgnoreCase("Content-Type")) // Remove old content type
    .header("Content-Type", "application/xml") // Add new content type
    .build();
```

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:

```
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, call the `firstValue` method:

```
Optional<String> lastModified = headerMap.firstValue("Last-Modified");
```

You get the response value or an empty optional if none was supplied.

The `HttpClient` is autocloseable, so you can declare it in a `try`-with-resources statement. Its `close` method waits for the completion of submitted requests and then closes its connection pool.

Tip: To enable logging for the `HttpClient`, add this line to `net.properties` in 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 named `jdk.httpclient.HttpClient` to `INFO`, for example by adding this line to the `logging.properties` file in your JDK:

```
jdk.httpclient.HttpClient.level=INFO
```

9.4. Regular Expressions

Regular expressions specify string patterns. Use them whenever you need to locate strings that match a particular pattern. For example, suppose you want to find hyperlinks in an HTML file. You need to look for strings of the pattern <a href="...">. But wait—there may be extra spaces, or the URL may be enclosed in single quotes. Regular expressions give you a precise syntax for specifying what sequences of characters are legal matches.

In the following sections, you will see the regular expression syntax used by the Java API, and how to put regular expressions to work.

9.4.1. The Regular Expression Syntax

In a regular expression, a character denotes itself unless it is one of the reserved characters

```
. * + ? { | ( ) [ \ ^ $
```

For example, the regular expression Java only matches the string Java.

The symbol . matches any single character. For example, .a.a matches Java and data.

The * symbol indicates that the preceding constructs may be repeated 0 or more times; for a +, it is 1 or more times. A suffix of ? indicates that a construct is optional (0 or 1 times). For example, be+s? matches be, bee, and bees. You can specify other multiplicities with { } (see Table 9.4).

A | denotes an alternative: .(oo|ee)f matches beef or woof. Note the parentheses—without them, .oo|eef would be the alternative between .oo and eef. Parentheses are also used for grouping—see Section 9.4.4.

A *character class* is a set of character alternatives enclosed in brackets, such as [Jj], [0-9], [A-Za-z], or [^0-9]. Inside a character class, the - denotes a range (all characters whose Unicode values fall between the two bounds). However, a - that is the first or last character in a character class denotes itself. A ^ as the first character in a character class denotes the complement (all characters except those specified).

Table 9.4 contains a number of *predefined character classes* such as \d (digits). There are many more with the \p prefix, such as \p{Sc} (Unicode currency symbols)—see Table 9.5.

The characters ^ and $ match the beginning and end of input.

If you need to have a literal . * + ? { | ( ) [\ ^ $, precede it by a backslash. Inside a character class, you only need to escape [and \, provided you are careful about the positions of] - ^. For example, []^-] is a class containing all three of them.

Caution: If the regular expression is in a string literal, each backslash needs to be escaped with another backslash. If you forget that second backslash, you usually get an error because sequences such as `\$` or `\.` are not valid in string literals. But if you want to match a word boundary and accidentally use `\b` instead of `\\b`, then you have a problem: `\b` is a valid escape sequence, indicating a backspace.

Instead of using backslashes, you can surround a string with `\Q` and `\E`. For example, `\(\$0\.99\)` and `\Q($0.99)\E` both match the string `($0.99)`.

Tip: If you have a string that may contain some of the many special characters in the regular expression syntax, you can escape them all by calling `Pattern.quote(str)`. This simply surrounds the string with `\Q` and `\E`, but it takes care of the special case where `str` may contain `\E`.

Table 9.4: Regular Expression Syntax

Expression	Description	Example
Characters		
c, not one of `. * + ? { | ( ) [ \ ^ $`	The character *c*.	`J`
`.`	Any character except line terminators, or any character if the `DOTALL` flag is set.	
`\X`	Any Unicode “extended grapheme cluster”, which is perceived as a character or symbol	
`\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`

Expression	Description	Example
\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 9.5); its complement.	\p{L} matches a Unicode letter, and so does \pL—you can omit braces around a single letter.
\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.

Expression	Description	Example
\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 (see Section 9.4.4)		
(*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 (see Section 9.4.7) on or off (after -).	(?i:jpe?g) is a case-insensitive match.
Other (?...)	See the Pattern API documentation.	
Quantifiers		
X?	Optional *X*.	\+? is an optional + sign.
*X**, *X*+	0 or more *X*, 1 or more *X*.	[1-9][0-9]+ is an integer ≥ 10.

Expression	Description	Example
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.	`.*(<.+?>).*` matches 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`.
`\b{g}`	Grapheme cluster boundary	Useful with `split` to decompose a string into grapheme clusters
`\R`	A Unicode line break.	
`\G`	The end of the previous match.	

Table 9.5: Predefined Character Classes `\p{...}`

Name	Description
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`.

Name	Description
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).

9.4.2. Testing a Match

Generally, there are two ways to use a regular expression: Either you want to test whether a string matches the expression, or you want to find one or more matches of the expression in a string.

The static matches method tests whether an *entire string* matches a regular expression:

```
String regex = "[+-]?\\d+";
CharSequence input = ...;
if (Pattern.matches(regex, input)) {
    // input matches the regular expression
    ...
}
```

If you need to use the same regular expression many times, it is more efficient to compile it. Then, create a Matcher for each input:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) ...
```

If the match succeeds, you can retrieve the location of matched groups—see Section 9.4.4.

To test whether a string *contains* a match, use the find method instead:

```
if (matcher.find()) {
    // A substring of input matches the regular expression
    ...
}
```

The match and find methods mutate the state of the Matcher object. If you just want to find out whether a given Matcher has found a match, call the hasMatch method instead.

You can turn the pattern into a predicate. This is particularly useful with the filter method of a stream:

```
Pattern digits = Pattern.compile("[0-9]+");
List<String> strings = List.of("December", "31st", "1999");
List<String> matchingStrings = strings.stream()
    .filter(digits.asMatchPredicate())
    .toList(); // ["1999"]
```

The result contains all strings that match the regular expression.

Use the asPredicate method to test whether a string *contains* a match:

```
List<String> sringsContainingMatch = strings.stream()
    .filter(digits.asPredicate())
    .toList(); // ["31st", "1999"]
```

9.4.3. Finding All Matches

In this section, we consider a common use case for regular expressions—finding all matches in an input. Use this loop:

```
String input = ...;
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    String match = matcher.group();
    int matchStart = matcher.start();
    int matchEnd = matcher.end();
    ...
}
```

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(MatchResult::group)
    .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);
Stream<String> words = in.findAll("\\pL+")
    .map(MatchResult::group);
```

9.4.4. Groups

It is common to use groups for extracting components of a match. For example, suppose you have a line item in the invoice with item name, quantity, and unit price such as

```
Blackwell Toaster    USD29.95
```

Here is a regular expression with groups for each component:

```
(\p{Alnum}+(\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

After matching, you can extract the `n`th group from the matcher as

```
String contents = matcher.group(n);
```

Groups are ordered by their opening parenthesis, starting at 1. (Group 0 is the entire input.) In this example, here is how to take the input apart:

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) {
    item = matcher.group(1);
    currency = matcher.group(3);
    price = matcher.group(4);
}
```

We aren’t interested in group 2; it only arose from the parentheses that were required for the repetition. For greater clarity, you can denote that group as “non-capturing”. Then it doesn’t show up as a group in the matcher.

```
(\p{Alnum}+(?:\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

Or, even better, use named groups:

```
(?<item>\p{Alnum}+(\s+\p{Alnum}+)*)\s+(?<currency>[A-Z]{3})(?<price>[0-9.]*)
```

Then, you can retrieve the groups by name:

```
item = matcher.group("item");
```

With the start and end methods of the Matcher and MatchResult classes, you can get the group positions in the input:

```
int itemStart = matcher.start("item");
int itemEnd = matcher.end("item");
```

The namedGroups method yields a Map<String, Integer from group names to numbers.

Note: When you have a group inside a repetition, such as (\s+\p{Alnum}+)* in the example above, it is not possible to get all of its matches. The group method only yields the last match, which is rarely useful. You need to capture the entire expression with another group.

9.4.5. 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);
```

To also collect the delimiters, use the splitWithDelimiters method:

```
tokens = commas.splitWithDelimiters(input, -1); // ["1", ", ", "2", " , ", "3", ",", "4"]
```

If the second argument is a positive number *n*, the separator pattern is applied at most *n* - 1 times. and the last element is the remaining string. Otherwise, the pattern is applied as often as possible. With a limit of zero, trailing empty strings are discarded.

If you don't care about precompiling the pattern or lazy fetching, you can just use the split and splitWithDelimiter methods of the String class:

```
tokens = input.split("\\s*,\\s*");
```

Caution: It is easy to forget that the argument of split is a regular expression. For example,

```
"com.horstmann.corejava".split(".")
```

does not split along the dots. Instead, every character is a separator, and the result is an empty array!

You need to escape the dot with a backslash in the regular expression, and therefore with two backslashes in the string literal:

```
"com.horstmann.corejava".split("\\.")
```

Alternatively, use the `Pattern.quote` method:

```
"com.horstmann.corejava".split(Pattern.quote("."));
```

If the input is in a file, use a scanner:

```
var in = new Scanner(path);
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

9.4.6. Replacing Matches

If you want to replace all matches of a regular expression with a string, call `replaceAll` on the matcher:

```
Matcher matcher = commas.matcher(input);
String result = matcher.replaceAll(",");
    // Normalizes the commas
```

Or, if you don't care about precompiling, use the `replaceAll` method of the `String` class.

```
String result = input.replaceAll("\\s*,\\s*", ",");
```

The replacement string can contain group numbers `$`*n* or names `${`*name*`}`. They are replaced with the contents of the corresponding captured group.

```
String result = "3:45".replaceAll(
    "(\\d{1,2}):(?<minutes>\\d{2})",
    "$1 hours and ${minutes} minutes");
    // Sets result to "3 hours and 45 minutes"
```

You can use `\` to escape `$` and `\` in the replacement string, or you can call the `Matcher.quoteReplacement` convenience method:

```
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")
    .replaceAll(m -> m.group().toUpperCase());
    // Yields "MARY had a LITTLE LAMB"
```

The `replaceFirst` method replaces only the first occurrence of the pattern.

9.4.7. Flags

Several *flags* change the behavior of regular expressions. You can specify them when you compile the pattern:

```
Pattern pattern = Pattern.compile(regex,
    Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CHARACTER_CLASS);
```

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 taking 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.

9.5. Serialization

In the following sections, you will learn about object serialization—a mechanism for turning an object into a bunch of bytes that can be shipped somewhere else or stored on disk, and for reconstituting the object from those bytes.

Serialization is an essential tool for distributed processing, where objects are shipped from one virtual machine to another. It is also used for fail-over and load balancing, when serialized objects can be moved to another server. If you work with server-side software, you will often need to enable serialization for classes. The following sections tell you how to do that.

9.5.1. The `Serializable` Interface

In order for an object to be serialized—that is, turned into a bunch of bytes—it must be an instance of a class that implements the `Serializable` interface. This is a marker interface with no methods, similar to the `Cloneable` interface that you saw in Chapter 4.

For example, to make `Employee` objects serializable, the class needs to be declared as

```
public class Employee implements Serializable {
    private String name;
    private double salary;
    ...
}
```

It is appropriate for a class to implement the `Serializable` interface if all instance variables have primitive or `enum` type, or contain references to serializable objects. Many classes in the standard library are serializable. Arrays and the collection classes that you saw in Chapter 7 are serializable provided their elements are.

In the case of the `Employee` class, and indeed with most classes, there is no problem. In the following sections, you will see what to do when a little extra help is needed.

To serialize objects, you need an `ObjectOutputStream`, which is constructed with another `OutputStream` that receives the actual bytes.

```
var out = new ObjectOutputStream(Files.newOutputStream(path));
```

Now call the `writeObject` method:

```
var peter = new Employee("Peter", 90000);
var paul = new Manager("Paul", 180000);
out.writeObject(peter);
out.writeObject(paul);
```

To read the objects back in, construct an `ObjectInputStream`:

```
var in = new ObjectInputStream(Files.newInputStream(path));
```

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();
```

When an object is written, the name of the class and the names and values of all instance variables are saved. If the value of an instance variable belongs to a primitive type, it is saved as binary data. If it is an object, it is again written with the `writeObject` method.

When an object is read in, the process is reversed. The class name and the names and values of the instance variables are read, and the object is reconstituted.

There is just one catch. Suppose there were two references to the same object. Let's say each employee has a reference to their boss:

```
var peter = new Employee("Peter", 90000);
var paul = new Manager("Barney", 105000);
var mary = new Manager("Mary", 180000);
peter.setBoss(mary);
paul.setBoss(mary);
out.writeObject(peter);
out.writeObject(paul);
```

When reading these two objects back in, both of them need to have the *same* boss, not two references to identical but distinct objects.

In order to achieve this, each object gets a *serial number* when it is saved. When you pass an object reference to `writeObject`, the `ObjectOutputStream` checks if the object reference was previously written. In that case, it just writes out the serial number and does not duplicate the contents of the object.

In the same way, an `ObjectInputStream` remembers all objects it has encountered. When reading in a reference to a repeated object, it simply yields a reference to the previously read object.

Note: If the superclass of a serializable class is not serializable, it must have an accessible no-argument constructor. Consider this example:

```
class Person // Not serializable
class Employee extends Person implements Serializable
```

When an Employee object is deserialized, its instance variables are read from the object input stream, but the Person instance variables are set by the Person constructor.

9.5.2. Transient Instance Variables

Certain instance variables should not be serialized—for example, database connections that are meaningless when an object is reconstituted. Also, when an object keeps a cache of values, it might be better to drop the cache and recompute it instead of storing it.

To prevent an instance variable from being serialized, simply tag it with the transient modifier. Always mark instance variables as transient if they hold instances of nonserializable classes. Transient instance variables are skipped when objects are serialized.

9.5.3. The readObject and writeObject Methods

In rare cases, you need to tweak the serialization mechanism. A serializable class can add any desired action to the default read and write behavior, by defining methods with the signature

```
@Serial private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
@Serial private void writeObject(ObjectOutputStream out)
    throws IOException
```

Then, the object headers continue to be written as usual, but the instance variables fields are no longer automatically serialized. Instead, these methods are called.

Note the @Serial annotation. The methods for tweaking serialization don't belong to interfaces. Therefore, you can't use the @Override annotation to have the compiler check the method declarations. The @Serial annotation is meant to enable the same checking for serialization methods. Up to Java 17, the javac compiler doesn't do that checking, but it might happen in the future. Some IDEs check the annotation.

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, 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.

```
@Serial before private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    out.writeDouble(point.getX());
    out.writeDouble(point.getY());
}
```

In the `readObject` method, we reverse the process:

```
@Serial before private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    double x = in.readDouble();
    double y = in.readDouble();
    point = new Point2D.Double(x, y);
}
```

Another example is the `HashSet` class that supplies its own `readObject` and `writeObject` methods. Instead of saving the internal structure of the hash table, the `writeObject` method simply saves the capacity, load factor, size, and elements. The `readObject` method reads back the capacity and load factor, constructs a new table, and inserts the elements.

The `readObject` and `writeObject` methods only need to save and load their data. They do not concern themselves with superclass data or any other class information.

The `Date` class uses this approach. Its `writeObject` method saves the milliseconds since the "epoch" (January 1, 1970). The data structure that caches calendar data is not saved.

Caution: Just like a constructor, the `readObject` method operates on partially initialized objects. If you call a non-final method inside `readObject` that is overridden in a subclass, it may access uninitialized data.

Note: If a serializable class defines a field

```
@Serial private static final ObjectStreamField[] serialPersistentFields
```

then serialization uses those field descriptors instead of the non-transient non-static fields. There is also an API for setting the field values before serialization or reading them after deserialization. This is useful for preserving a legacy layout after a class has evolved. For example, the BigDecimal class uses this mechanism to serialize its instances in a format that no longer reflects the instance fields.

9.5.4. The readExternal and writeExternal Methods

Instead of letting the serialization mechanism save and restore object data, a class can define its own mechanism. For example, you can encrypt the data or use a format that is more efficient than the serialization format.

To do this, a class implements the Externalizable interface instead of the Serializable interface. This, in turn, requires two methods:

```
public void readExternal(ObjectInputStream in)
    throws IOException
public void writeExternal(ObjectOutputStream out)
    throws IOException
```

Unlike the readObject and writeObject methods, 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.

In this example, the LabeledPixel class extends the serializable Point class, but it takes over the serialization of the class and superclass. The fields of the object are not stored in the standard serialization format. Instead, the data are placed in an opaque block.

```
public class LabeledPixel extends Point implements Externalizable {
    private String label;

    public LabeledPixel() {} // required for externalizable class

    @Override public void writeExternal(ObjectOutput out)
            throws IOException {
        out.writeInt((int) getX());
        out.writeInt((int) getY());
        out.writeUTF(label);
    }

    @Override public void readExternal(ObjectInput in)
            throws IOException, ClassNotFoundException {
        int x = in.readInt();
        int y = in.readInt();
        setLocation(x, y);
        label = in.readUTF();
```

```
    }
    ...
}
```

Note: The `readExternal` and `writeExternal` methods should not be annotated with `@Serial`. Since they are defined in the `Externalizable` interface, you can simply annotate them with `@Override`.

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.

Note: You cannot customize the serialization of enumerations and records. If you define `readObject`/`writeObject` or `readExternal`/`writeExternal` methods, they are not used for serialization.

9.5.5. The `readResolve` and `writeReplace` Methods

We take it for granted that objects can only be constructed with the constructor. However, a deserialized object is *not constructed*. Its instance variables are simply restored from an object stream.

This is a problem if the constructor enforces some condition. For example, a singleton object may be implemented so that the constructor can only be called once. As another example, database entities can be constructed so that they always come from a pool of managed instances.

You shouldn't implement your own mechanism for singletons. If you need a singleton, make an enumerated type with one instance that is, by convention, called `INSTANCE`.

```
public enum PersonDatabase {
    INSTANCE;

    public Person findById(int id) { ... }
    ...
}
```

This works because `enum` are guaranteed to be deserialized properly.

Now let's suppose that you are in the rare situation where you want to control the identity of each deserialized instance. As an example, suppose a `Person` class wants to restore its instances from a database when deserializing. Then you should not serialize the object itself. Instead request that a proxy instance is saved. When restored, that proxy locates and constructs the desired object. Your class needs to provide a `writeReplace` method that returns the proxy object:

```
public class Person implements Serializable {
    private int id;
    // Other instance variables
    ...
    @Serial private Object writeReplace() {
        return new PersonProxy(id);
    }
}
```

When a `Person` object is serialized, none of its instance variables are saved. Instead, the `writeReplace` method is called and *its return value* is serialized and written to the stream.

The proxy class needs to implement a `readResolve` method that yields a `Person` instance:

```
class PersonProxy implements Serializable {
    private int id;

    public PersonProxy(int id) {
        this.id = id;
    }

    @Serial private Object readResolve() {
        return PersonDatabase.INSTANCE.findById(id);
    }
}
```

When the `readObject` method finds a `PersonProxy` in an `ObjectInputStream`, it deserializes the proxy, calls its `readResolve` method, and returns the result.

Note: Unlike the `readObject` and `writeObject` methods, the `readResolve` and `writeReplace` methods need not be private.

Note: With enumerations and records, `readObject`/`writeObject` or `readExternal`/`writeExternal` methods are not used for serialization. With records, but not with enumerations, the `writeReplace` method will be used.

9.5.6. Versioning

Serialization was intended for sending objects from one virtual machine to another, or for short-term persistence of state. If you use serialization for long-term persistence, or in any situation where classes can change between serialization and deserialization, you will need to consider what happens when your classes evolve. Can version 2 read the old data? Can the users who still use version 1 read the files produced by the new version?

The serialization mechanism supports a simple versioning scheme. When an object is serialized, both the name of the class and its `serialVersionUID` are written to the object stream. That unique identifier is assigned by the implementor, by defining an instance variable

```
@Serial private static final long serialVersionUID = 1L; // Version 1
```

When the class evolves in an incompatible way, the implementor should change the UID. Whenever a deserialized object has a nonmatching UID, the `readObject` method throws an `InvalidClassException`.

If the `serialVersionUID` matches, deserialization proceeds even if the implementation has changed. Each non-transient instance variable of the object to be read is set to the value in the serialized state, provided that the name and type match. All other instance variables are set to the default: `null` for object references, zero for numbers, and `false` for `boolean` values. Anything in the serialized state that doesn't exist in the object to be read is ignored.

Is that process safe? Only the implementor of the class can tell. If it is, then the implementor should give the new version of the class the same `serialVersionUID` as the old version.

If you don't assign a `serialVersionUID`, one is automatically generated by hashing a canonical description of the instance variables, methods, and supertypes. You can see the hash code with the `serialver` utility. The command

```
serialver ch09.sec05.Employee
```

displays

```
private static final long serialVersionUID = -4932578720821218323L;
```

When the class implementation changes, there is a very high probability that the hash code changes as well.

If you need to be able to read old version instances, and you are certain that is safe to do so, run `serialver` on the old version of your class and add the result to the new version.

Note: If you want to implement a more sophisticated versioning scheme, override the `readObject` method and call the `readFields` method instead of the `defaultReadObject` method. You get a description of all fields found in the stream, and you can do with them what you want.

Note: Enumerations and records ignore the `serialVersionUID` field. An enumeration always has a `serialVersionUID` of `0L`. You can declare the `serialVersionUID` of a record, but the IDs don't have to match for deserialization.

Note: In this section, you saw what happens when the reader's version of a class has instance variables that aren't present in the object stream. It is also possible during class evolution for a superclass to be added. Then a reader using the new version may read an object stream in which the instance variables of the superclass are not set. By default, those instance fields are set to their `0`/`false`/`null` default. That may leave the superclass in an unsafe state. The superclass can defend against that problem by defining an initialization method

```
@Serial private void readObjectNoData() throws ObjectStreamException
```

The method should either set the same state as the no-argument constructor or throw an `InvalidObjectException`. It is only called in the unusual circumstance where an object stream is read that contains an instance of a subclass with missing superclass data.

9.5.7. Deserialization and Security

During deserialization of a serializable class, objects are created without invoking any constructor of the class. Even if the class has a no-argument constructor, it is not used. The field values are set directly from the values of the object input stream.

Note: For serializable *records*, deserialization calls the canonical constructor, passing it the values of the components from the object input stream. (As a consequence, cyclic references in records are not restored.)

Bypassing construction is a security risk. An attacker can craft bytes describing an invalid object that could have never been constructed. Suppose, for example, that the `Employee` constructor throws an exception when called with a negative salary. We would like to think that no `Employee` object can have a negative salary as a result. But it is not difficult to inspect the bytes for a serialized object and modify some of them. This way, one can craft bytes for an employee with a negative salary and then deserialize them.

A serializable class can optionally implement the `ObjectInputValidation` interface and define a `validateObject` method to check whether its objects are properly deserialized. For example, the `Employee` class can check that salaries are not negative:

```
public void validateObject() throws InvalidObjectException {
    System.out.println("validateObject");
    if (salary < 0)
        throw new InvalidObjectException("salary < 0");
}
```

Unfortunately, the method is not invoked automatically. To invoke it, you also must provide the following method:

```
@Serial private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
    in.registerValidation(this, 0);
    in.defaultReadObject();
}
```

The object is then scheduled for validation, and the `validateObject` method is called when this object and all dependent objects have been loaded. The second parameter lets you specify a priority. Validation requests with higher priorities are done first.

There are other security risks. Adversaries can create data structures that consume enough resources to crash a virtual machine. More insidiously, any class on the class path can be deserialized. Hackers have been devious about piecing together “gadget chains”—sequences of operations in various utility classes that use reflection and culminate in calling methods such as `Runtime.exec` with a string of their choice.

Any application that receives serialized data from untrusted sources over a network connection is vulnerable to such attacks. For example, some servers serialize session data and deserialize whatever data are returned in the HTTP session cookie.

You should avoid situations in which arbitrary data from untrusted sources are deserialized. In the example of session data, the server should sign the data, and only deserialize data with a valid signature.

A *serialization filter* mechanism can harden applications from such attacks. The filters see the names of deserialized classes and several metrics (stream size, array sizes, total number of references, longest chain of references). Based on those data, the deserialization can be aborted.

In its simplest form, you provide a pattern describing the valid and invalid classes. For example, if you start our sample serialization demo as

```
java -Djdk.serialFilter='serial.*;java.**;!*' serial.ObjectStreamTest
```

then the objects will be loaded. The filter allows all classes in the `serial` package and all classes whose package name starts with `java`, but no others. If you don’t allow `java.**`, or at least `java.util.Date`, deserialization fails.

You can place the filter pattern into a configuration file and specify multiple filters for different purposes. You can also implement your own filters. See https://docs.oracle.com/en/java/javase/21/core/serialization-filtering1.html for details.

9.6. Exercises

1. Write a utility method for copying all of an `InputStream` to an `OutputStream`, without using any temporary files. Provide another solution, without a loop, using operations from the `Files` class, using a temporary file.
2. Write a program that reads a text file and produces a file with the same name but extension `.toc`, containing an alphabetized list of all words in the input file together with a list of line numbers in which each word occurs. Assume that the file's encoding is UTF-8.
3. Write a program that reads a file containing text and, assuming that most words are English, guesses whether the encoding is ASCII, ISO 8859-1, UTF-8, or UTF-16, and if the latter, which byte ordering is used.
4. Using a `Scanner` is convenient, but it is a bit slower than using a `BufferedReader`. Read in a long file a line at a time, counting the number of input lines, with (a) a `Scanner` and `hasNextLine`/`nextLine`, (b) a `BufferedReader` and `readLine`, (c) a `BufferedReader` and `lines`. Which is the fastest? The most convenient?
5. When an encoder of a `Charset` with partial Unicode coverage can't encode a character, it replaces it with a default—usually, but not always, the encoding of `"?"`. Find all replacements of all available character sets that support encoding. Use the `newEncoder` method to get an encoder, and call its `replacement` method to get the replacement. For each unique result, report the canonical names of the charsets that use it.
6. The BMP file format for uncompressed image files is well documented and simple. Using random access, write a program that reflects each row of pixels in place, without writing a new file.
7. Look up the API documentation for the `MessageDigest` class and write a program that computes the SHA-512 digest of a file. Feed blocks of bytes to the `MessageDigest` object with the `update` method, then display the result of calling `digest`. Verify that your program produces the same result as the `sha512sum` utility.
8. Write a utility method for producing a ZIP file containing all files from a directory and its descendants.
9. Using the `URLConnection` class, read data from a password-protected web page with "basic" authentication. Concatenate the user name, a colon, and the password, and compute the Base64 encoding:

   ```
   String input = username + ":" + password;
   String encoding = Base64.getEncoder().encodeToString(input.getBytes());
   ```

   Set the HTTP header `Authorization` to the value `"Basic " + encoding`. Then read and print the page contents.
10. Using a regular expression, extract all decimal integers (including negative ones) from a string into an `ArrayList<Integer>` (a) using `find`, and (b) using `split`. Note that a `+` or `-` that is not followed by a digit is a delimiter.

11. Using regular expressions, extract the directory path names (as an array of strings), the file name, and the file extension from an absolute or relative path such as `/home/cay/myfile.txt`.
12. Come up with a realistic use case for using group references in `Matcher.replaceAll` and implement it.
13. Implement a method that can produce a clone of any serializable object by serializing it into a byte array and deserializing it.
14. Implement a serializable class `Point` with instance variables for `x` and `y`. Write a program that serializes an array of `Point` objects to a file, and another that reads the file.
15. Continue the preceding exercise, but change the data representation of `Point` so that it stores the coordinates in an array. What happens when the new version tries to read a file generated by the old version? What happens when you fix up the `serialVersionUID`? Suppose your life depended upon making the new version compatible with the old. What could you do?
16. Which classes in the standard Java library implement `Externalizable`? Which of them use `writeReplace/readResolve`?
17. Unzip the API source and investigate how the `LocalDate` class is serialized. Why does the class define `writeExternal` and `readExternal` methods even though it doesn't implement `Externalizable`? (Hint: Look at the `Ser` class.) Why does the class define a `readObject` method? How could it be invoked?

This page intentionally left blank

Index

Symbols

A

D

E

F

G

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z