西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object Oriented Programming

Chapter 9
Input and Output

Dr. Helei Cui

24 May 2023

Slides partially adapted from lecture notes by Cay Horstmann

Contents

Input/Output Streams

- An *input stream* is a source of bytes.
- An *output stream* is a destination for bytes.
  - These sources and destinations can be **files**, **network connections**, and **blocks of memory**.
- `InputStream` and `OutputStream` are the basis for a hierarchy of I/O classes.
- `Reader` and `Writer` are the basis for a hierarchy of I/O classes for processing Unicode characters.
  - Readers/writers process characters, not bytes.

- No relationship with `java.util.stream`.

9.1.1 Reading and Writing Bytes

- The `InputStream` class has an abstract method:

```
abstract int read()
```

  - The read method returns a single byte (as an int) or -1 at the end of input.
  - It is more common to read bytes in bulk:

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

  - Abstract read method can read a given number of bytes.
- The `OutputStream` class has an abstract method:

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

  - You can write one byte or bytes from an array:

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

9.1.1 Reading and Writing Bytes

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

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

- The `available` method lets you check the number of bytes that are currently available for reading:

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

- When writing to a stream, close it when you are done:

```
out.close();
```

- You can use one of many input/output classes that build upon the basic `InputStream` and `OutputStream` classes.

9.1.2 The Complete Stream Zoo

- Java has a whole zoo of more than 60 different input/output stream types.

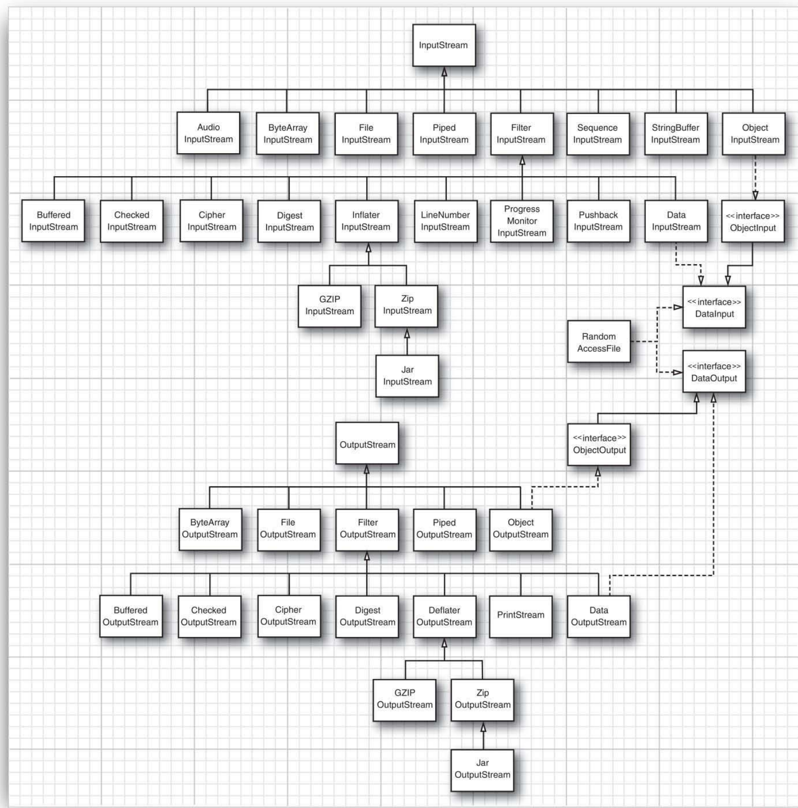

Figure 2.1 Input and output stream hierarchy

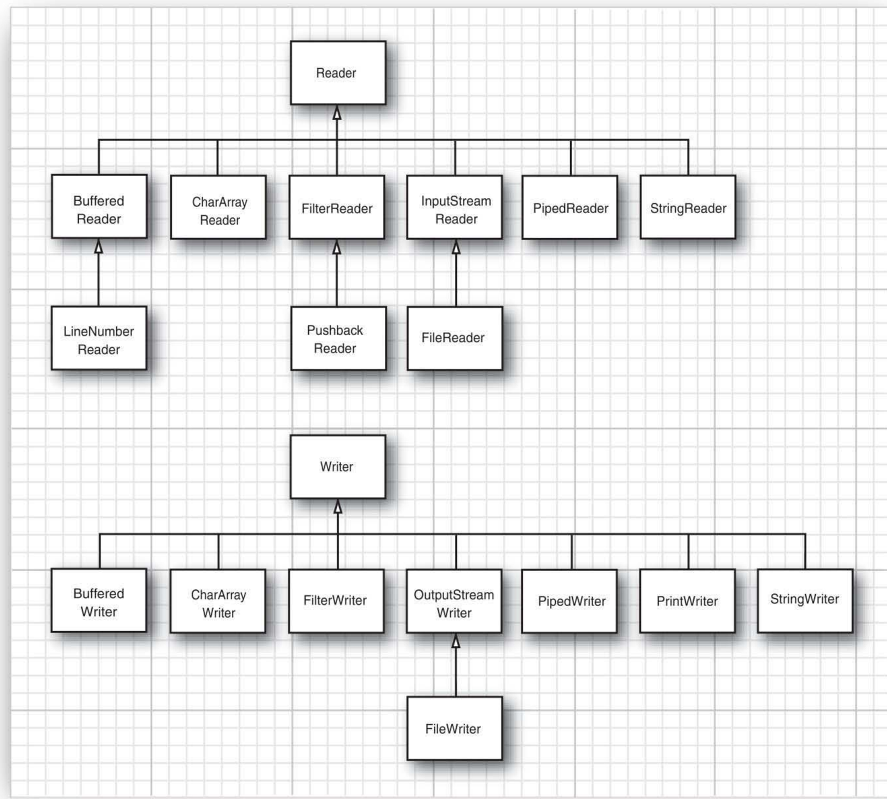

Figure 2.2 Reader and writer hierarchy

9.1.2 The Complete Stream Zoo

- The `InputStream` and `OutputStream` classes let you read and write individual bytes and arrays of bytes.

- 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.
- `ZipInputStream` and `ZipOutputStream` let you read and write files in the familiar ZIP compression format.

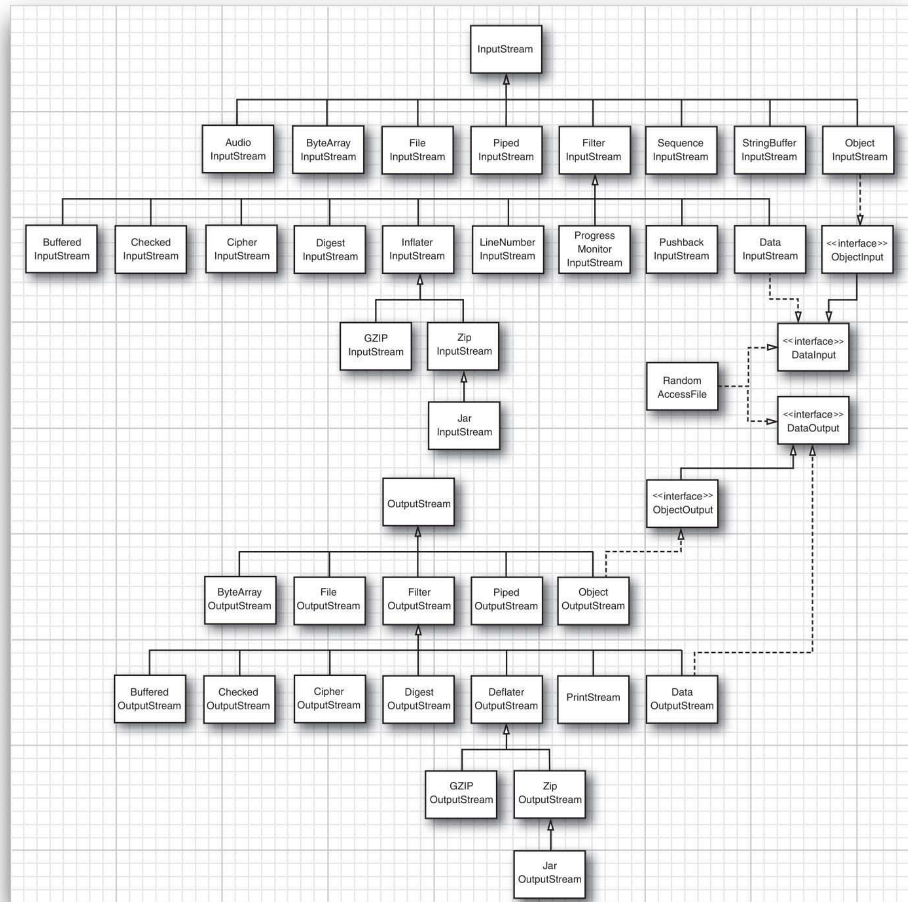

Figure 2.1 Input and output stream hierarchy

9.1.2 The Complete Stream Zoo

- For Unicode text, on the other hand, you can use subclasses of the abstract classes Reader and Writer.

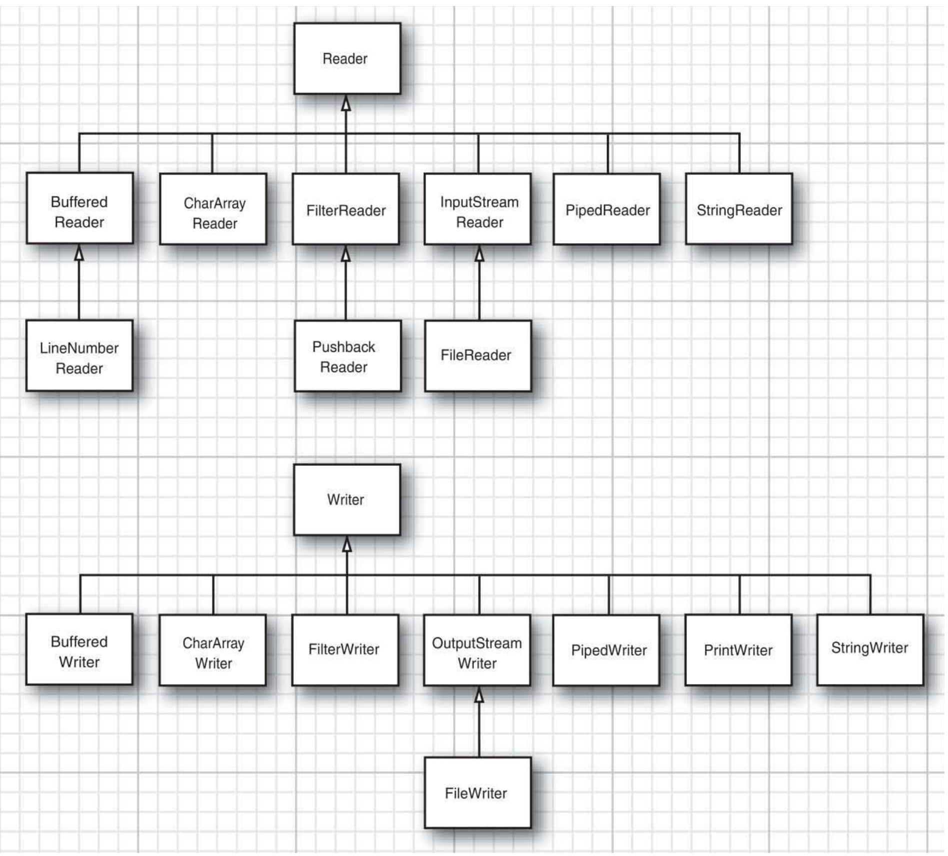

Figure 2.2 Reader and writer hierarchy

- The basic methods:

```
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.

9.1.2 The Complete Stream Zoo

- There are four additional interfaces: `Closeable`, `Flushable`, `Readable`, and `Appendable`.
  - The classes `InputStream`, `OutputStream`, `Reader`, and `Writer` all implement the `Closeable` interface.
  - `OutputStream` and `Writer` implement the `Flushable` interface.

```
void close() throws IOException
void flush()
int read(CharBuffer cb)
```

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

- The `CharBuffer` class has methods for sequential and random read/write access.
  - It represents an in-memory buffer or a memory-mapped file.
- The `Appendable` interface has two methods for appending single characters and character sequences.
- 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`.

9.1.2 The Complete Stream Zoo

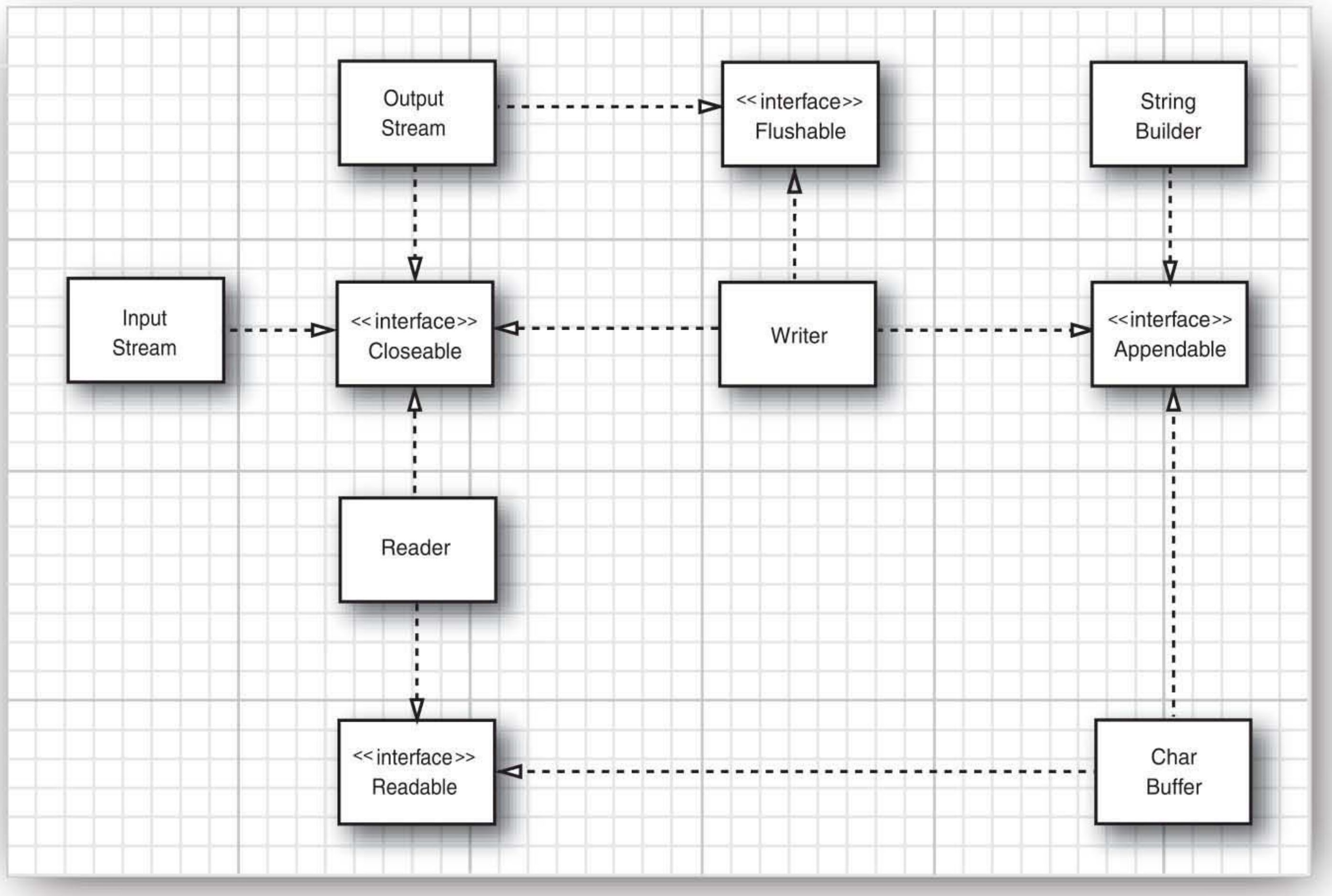

Figure 2.3 The Closeable, Flushable, Readable, and Appendable interfaces

9.1.3 Combining Input/Output Stream Filters

- `FileInputStream` and `FileOutputStream` give you input and output streams attached to a disk file.

```
var fin = new FileInputStream("employee.dat");
    // pass the file name or full path name of the file
```

- Can only read bytes and byte arrays from the object `fin`.

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

- `DataInputStream` can read numeric types. But it has no method to get data from a file.

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

- **You can combine the two responsibilities(retrieve bytes ; assemble bytes).**

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

9.1.3 Combining Input/Output Stream Filters

- You can add multiple capabilities by nesting the filters. If you want buffering and the data input methods for a file:

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

- Sometimes you'll need to keep track of the intermediate input streams when chaining them together.

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

int b = pbin.read();//speculatively read the next byte

if (b != '<') pbin.unread(b);//throw it back
```

- Reading and unreading are the only methods that apply to a pushback input stream.

9.1.3 Combining Input/Output Stream Filters

- The ability to mix and match filter classes to construct useful sequences of input/output streams is flexible.

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

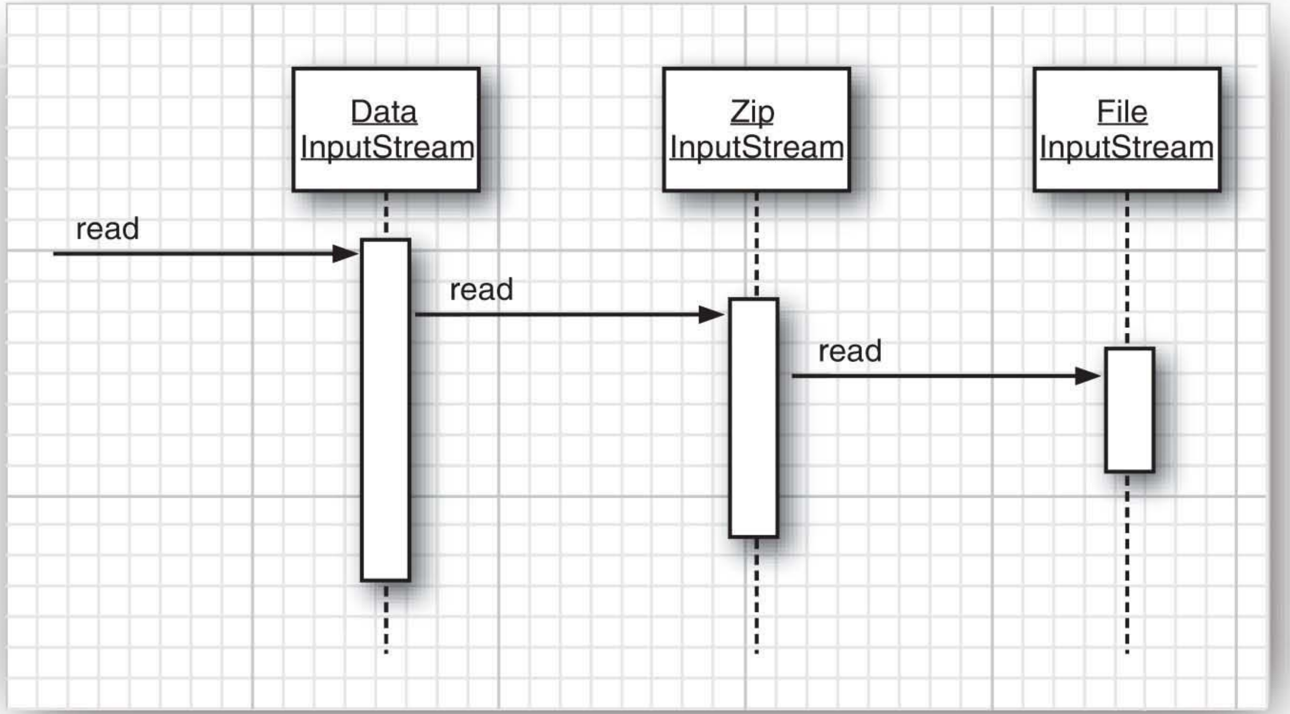

Figure 2.4
A sequence of filtered input streams

9.1.4 Text Input and Output

- When saving data, you have the choice between binary and text formats.
  - When saving text strings, you need to consider the character encoding.
- The `OutputStreamWriter` class turns an output stream of Unicode code units into a stream of bytes.
- The `InputStreamReader` class turns an input stream that contains bytes into Unicode code units.

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

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

- Use subclasses for processing strings and numbers.

9.1.5 How to Write Text Output

- `PrintWriter` class has methods to print strings and numbers in text format.

```
var out = new PrintWriter("employee.txt", StandardCharsets.UTF_8);
    //construct a PrintStream from a file name and a character encoding
```

- 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.

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

Harry Hacker 75000.0
```

9.1.5 How to Write Text Output

- The `println` method adds the correct end-of-line character for the target system ("\r\n" on Windows, "\n" on UNIX) to the line.
- You can enable or disable autoflushing by using the *`PrintWriter`(Writer writer, boolean autoFlush)* constructor:

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

  - By default, autoflushing is not enabled.
- The `print` methods don't throw exceptions.
  - You can call the `checkError` method to see if something went wrong with the output stream.

9.1.6 How to Read Text Input

- The easiest way to process arbitrary text is the `Scanner` class. You can construct a `Scanner` from any input stream.
- Can read a short text file into a string like this:

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

- 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)){
    . . .
}
```

- Use a scanner to read `tokens`(strings separated by a delimiter). The default delimiter is white space.
  - You can change the delimiter to any regular expression.

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

9.1.6 How to Read Text Input

- 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();
```

- The `BufferedReader` class has a lines method that yields a Stream<String>.
- Unlike a Scanner, a `BufferedReader` has no methods for reading numbers.

9.1.7 Saving Objects in Text Format

- An example program that stores an array of Employee records in a text file. We use a vertical bar (|) as our delimiter.
- 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
```

- Write all fields, followed by either a | or, for the last field, a newline character.

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

9.1.7 Saving Objects in Text Format

- 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.

9.1.7 Saving Objects in Text Format

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

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

- The static method first reads in the length of the array, then reads in each record.

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

- 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);
}
```

9.1.8 Character Encodings

- Java uses the Unicode standard for characters.
- The most common encoding is UTF-8, which encodes each Unicode code point into a sequence of one to four bytes.

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	$11110a_{20}a_{19}a_{18}$ $10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$

- Another common encoding is UTF-16.

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 $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

9.1.8 Character Encodings

- In addition to the UTF encodings, there are partial encodings that cover a character range suitable for a given user population (ISO 8859-1; Shift-JIS).
- There is no reliable way to automatically detect the character encoding from a stream of bytes. You should always explicitly specify the encoding.
- The `StandardCharsets` class has static variables of type `Charset` for the character encodings.
- 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.

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

Contents

9.2.1 The DataInput and DataOutput interfaces

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

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

- The `writeUTF` method writes string data using a modified version of the 8-bit Unicode Transformation Format.
- To read the data back in, use the following methods defined in the `DataInput` interface:

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

9.2.1 The DataInput and DataOutput interfaces

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

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

- To write binary data, use the `DataOutputStream` class that implements the `DataOutput` interface:

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

9.2.2 Random-Access Files

- The `RandomAccessFile` class lets you read or write data anywhere in a file.
- Specify the option by using the string "r" (for read access) or "rw" (for read/write access).

```
var in = new RandomAccessFile("employee.dat", "r");
var inOut = new RandomAccessFile("employee.dat", "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 can be used to set the file pointer to an arbitrary byte position within the file.
  - The `getFilePointer` method returns the current position of the file pointer.
  - The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces.

9.2.2 Random-Access Files

- An example program:

```
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, set the file pointer back to the beginning of the record:

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

- Use the `length` method to determine the total number of bytes in a file:

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

9.2.2 Random-Access Files

- There are two helper methods to write and read strings of a fixed size.
- The writeFixedString writes the specified number of code units, starting at the beginning of the string.

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

- If there are too few code units, the method pads the string, using zero values.

9.2.2 Random-Access Files

- The `readFixedString` method uses the `StringBuilder` class to read in a string.

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

- Place the `writeFixedString` and `readFixedString` methods inside the `DataIO` helper class.

9.2.2 Random-Access Files

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

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

- Reading the data back is just as simple.

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

9.2.3 ZIP Archives

- ZIP archives store one or more files in compressed format.
  - Each ZIParchive has a header with information .
  - Use a `ZipInputStream` to read a ZIP archive.
  - The `getNextEntry` method returns an object of type `ZipEntry` that describes the entry.
  - Do not close zin until you read the last entry.
- A typical code sequence to read through a ZIP file:

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

9.2.3 ZIP Archives

- Use a `ZipOutputStream` to write a ZIP file.

```
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();
```

- 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.
  - The source of the bytes in a ZIP stream need not be a file - the ZIP data can come from a network connection.

Contents

9.3.1 Saving and Loading Serializable Objects

- Use the `writeObject` method of the `ObjectOutputStream` class to save an object.

```
var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
var boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

- To read the objects back in, first get an `ObjectInputStream` object:

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

  - Then, use the `readObject` method to retrieve the objects in the same order in which they were written:

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

9.3.1 Saving and Loading Serializable Objects

- The class must implement the `Serializable` interface that save to an output stream and restore from an object input stream:

```
class Employee implements Serializable { . . . }
```

- The `Serializable` interface has no methods.
- An `ObjectOutputStream` looks at all the fields of the objects and saves their contents.
- **What happens when one object is shared by several objects as part of their state?**

```
class Manager extends Employee {
    private Employee secretary;
    . . .
} // Assume that each manager has a secretary
```

9.3.1 Saving and Loading Serializable Objects

```
var harry = new Employee("Harry Hacker", . . .);
var carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
var tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

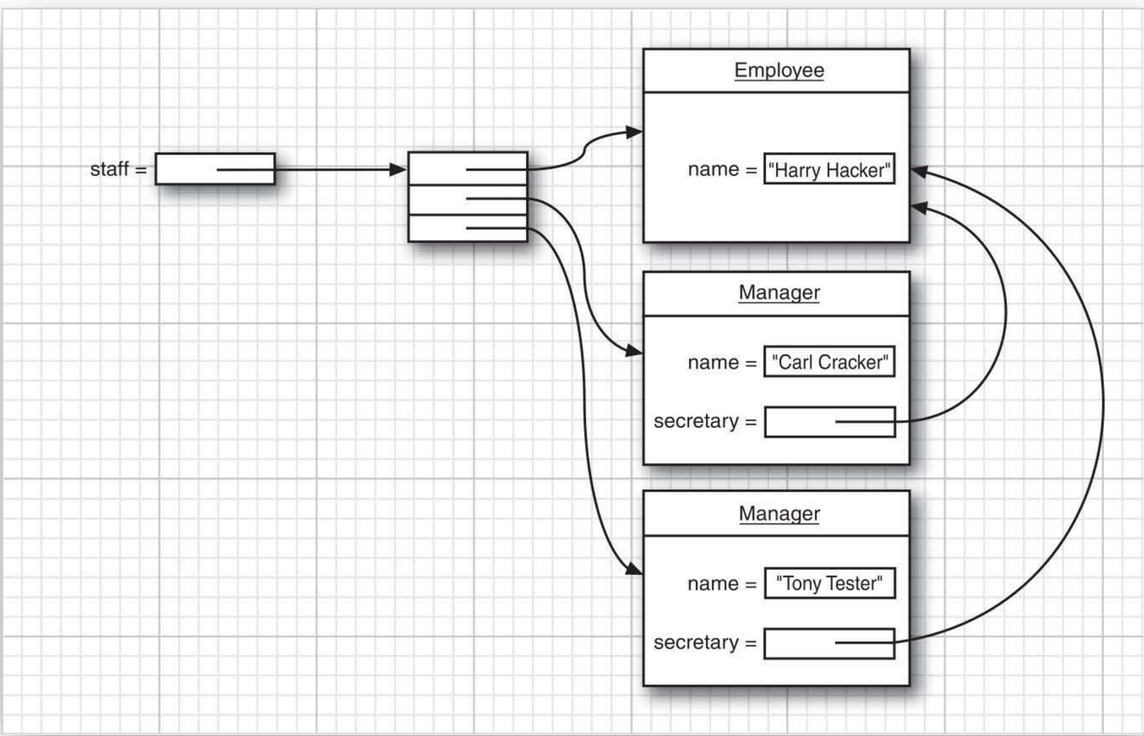

Figure 2.5
Two managers can share a mutual employee.

9.3.1 Saving and Loading Serializable Objects

- Each object is saved with the serial number - hence the name object serialization for this mechanism.

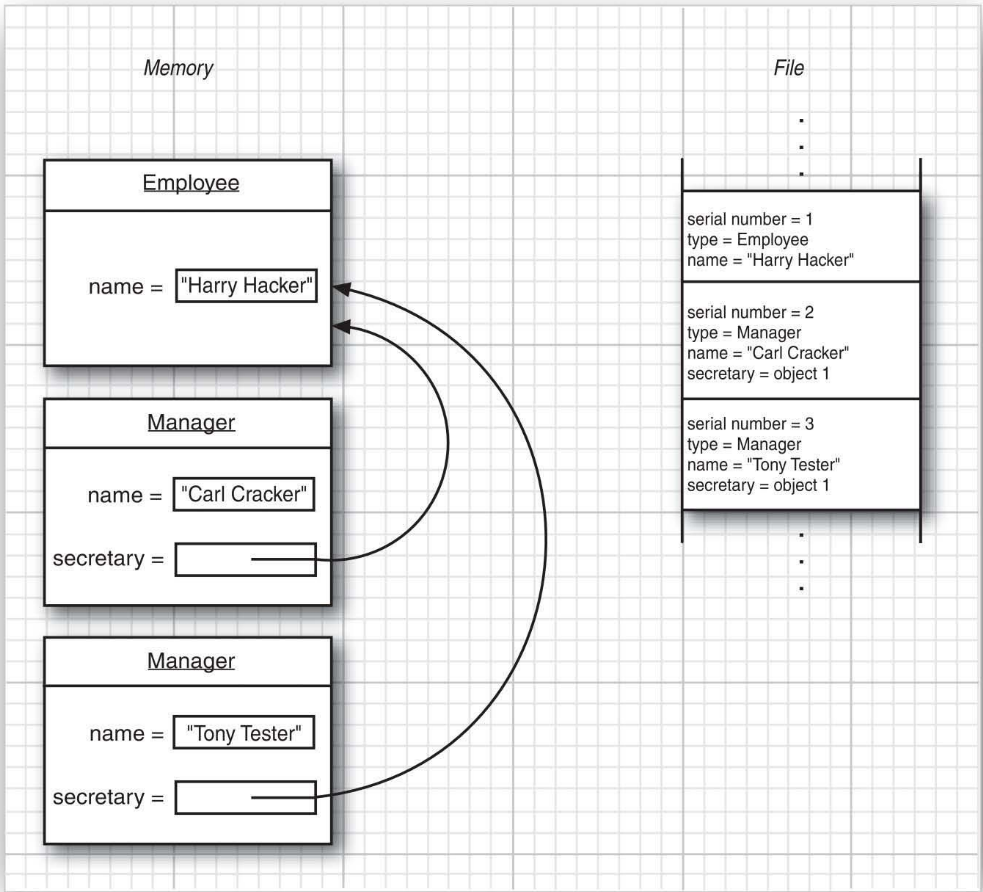

Figure 2.6
An example of object serialization

9.3.2 Understanding the Object Serialization File Format

- **Object serialization saves object data in a particular file format.**

- 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.

Contents

9.4.1 Paths

- Path objects specify abstract path names (which may not currently exist on disk).
  - **A `Path` is a sequence of directory names, optionally followed by a file name.**
  - First component may be a root component such as / or C:\
  - Path starting with a root is absolute. Other paths are relative.

```
Path absolute = Paths.get("/home", "harry");
Path relative = Paths.get("myprog", "conf", "user.properties");
```

- The static `Paths.get` method receives strings, which it joins with the path separator of the default file system.
- Path separator is supplied for the default file system.
  - / for a UNIX-like system
  - \ for Windows

9.4.1 Paths

- The `get` method can get a single string containing multiple components.

```
String baseDir = props.getProperty("base.dir");
// May be a string such as /opt/myprog or c:\Program Files\myprog
Path basePath = Paths.get(baseDir); // OK that baseDir has separators
```

- The call `p.resolve(q)` returns a path according to rules:
  - If q is absolute, that's just q.
  - Otherwise, first follow p, then follow q:

```
Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);
```

- A shortcut for the `resolve` method takes a string instead of a path:

```
Path workPath = basePath.resolve("work");
```

9.4.1 Paths

- `resolveSibling` resolves against a path's parent, yielding a sibling path.

```
Path tempPath = workPath.resolveSibling("temp");
    //if workPath is /opt/myapp/work, create /opt/myapp/temp
```

- The opposite of resolve is `relativize`, yielding "how to get from p to q".
  - E.g., relativizing /home/harry against /home/fred/input.txt yields ../fred/input.txt
- The `normalize` method removes `.` and `..` or other redundancies.
  - Normalizing the path /home/harry/../fred/./input.txt yields /home/fred/input.txt
- The `toAbsolutePath` method makes a path absolute.
  - Such as /home/fred/input.txt or c:\Users\fred\input.txt

9.4.1 Paths

- The Path interface has many useful methods for taking paths apart.

```
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 /
```

- You can construct a Scanner from a Path object:

```
var in = new Scanner(Paths.get("/home/fred/input.txt"));
```

9.4.2 Reading and Writing Files

- The Files class makes quick work of common file operations.

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

- You can read the content of a text file as:

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

- If you want the file as a sequence of lines, call:

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

- if you want to write a string, call:

```
Files.write(path, content.getBytes(charset));
```

- To append to a given file, use:

```
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
```

- You can also write a collection of lines with:

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

9.4.2 Reading and Writing Files

- If your files are large or binary, you can still use the familiar input/output streams or readers/writers:

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

9.4.3 Creating Files and Directories

- To create a new directory, call:

```
Files.createDirectory(path); // 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); //throws an exception if the file exists
```

- 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);
```

9.4.4 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 overwrite an existing target, use the REPLACE_EXISTING option.
    - If copy all file attributes, use the COPY_ATTRIBUTES option.

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

- Use the ATOMIC_MOVE option to specify that a move should be atomic:

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

9.4.4 Copying, Moving, and Deleting Files

- Copy an input stream to a Path:

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

- Copy a Path to an output stream:

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

- To delete a file, call:

```
Files.delete(path);
```

- This method throws an exception if the file doesn't exist.

```
boolean deleted = Files.deleteIfExists(path);
```

- The deletion methods can also be used to remove an empty directory.

9.4.5 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.

9.4.5 Getting File Information

- The basic file attributes are:
  - The times at which the file was created, last accessed, and last modified, as instances of the class java.nio.file.attribute.FileTime.
  - Whether the file is a regular file, a directory, a symbolic link, or none of these.
  - The file size.
  - The file key—an object of some class, specific to the file system, that may or may not uniquely identify a file.
- To get these attributes, call:

```
BasicFileAttributes attributes = Files.readAttributes(path,
    BasicFileAttributes.class);
```

- You can instead get an instance of `PosixFileAttributes`:

```
PosixFileAttributes attributes = Files.readAttributes(path,
    PosixFileAttributes.class);
```

9.4.6 Visiting Directory Entries

- The static `Files.list` method returns a `Stream<Path>` that reads the entries of a directory.
- 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)) {
    . . .
}
```

- Use the `Files.walk` method to process all descendants of a directory.

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

9.4.6 Visiting Directory Entries

- 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
. . .
```

- **Whenever the traversal yields a directory, it is entered before continuing with its siblings.**

9.4.6 Visiting Directory Entries

- You can limit the depth of the tree that you want to visit by calling `Files.walk(pathToRoot, depth)`.
- 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);
     }
});
```

- Cannot easily use the `Files.walk` method to delete a tree of directories.
  - As you need to delete the children before deleting the parent.

9.4.7 Using Directory Streams

- If you need more fine-grained control over the traversal process, use the `Files.newDirectoryStream` object.

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir)) {
    for (Path entry : entries)
        Process entries
}
```

- The `try`-with-resources block ensures that the directory stream is properly closed.
- There is no specific order in which the directory entries are visited.
- You can filter the files with a glob pattern:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

9.4.7 Using Directory Streams

Table 2.4 Glob Patterns

Pattern	Description	Example
`*`	Matches zero or more characters of a path component.	`*.java` matches all Java files in the current directory.
`**`	Matches zero or more characters, crossing directory boundaries.	`**.java` matches all Java files in any subdirectory.
`?`	Matches one character.	`????.java` matches all four-character (not counting the extension) Java files.
`[. . .]`	Matches a set of characters. You can use hyphens `[0-9]` and negation `[!0-9]`.	`Test[0-9A-F].java` matches `Test`*x*`.java`, where *x* is one hexadecimal digit.
`{. . .}`	Matches alternatives, separated by commas.	`*.{java,class}` matches all Java and class files.
`\`	Escapes any of the above as well as `\`.	`*\**` matches all files with a * in their name.

9.4.7 Using Directory Streams

- 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)`

9.4.7 Using Directory Streams

- 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.
- A convenience class `SimpleFileVisitor` implements the `FileVisitor` interface.

9.4.7 Using Directory Streams

- Example: print out all subdirectories of a given directory:

```
Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path>() {
    public FileVisitResult preVisitDirectory(Path path,
            BasicFileAttributes attrs) throws IOException{
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult postVisitDirectory(Path dir, IOException exc){
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path path, IOException exc)
            throws IOException{
        return FileVisitResult.SKIP_SUBTREE;
    }
});
```

- Override `postVisitDirectory` and `visitFileFailed`.
- The attributes of the path are passed as a parameter to the `preVisitDirectory` and `visitFile` methods.

9.4.7 Using Directory Streams

- The FileVisitor interface are useful if you need to do some work when entering or leaving a directory.

```
// Delete the directory tree starting at root
Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
            throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult postVisitDirectory(Path dir, IOException e)
            throws IOException {
        if (e != null) throw e;
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
    }
});
```

9.4.8 ZIP File Systems

- The Paths class looks up paths in the default file system - the files on the user's local disk.
- If zipname is the name of a ZIP file, then the call:

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

- Copy a file out of that archive if you know its name:

```
Files.copy(fs.getPath(sourceName), targetPath);
```

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

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Path>(){
    public FileVisitResult visitFile(Path file, BasicFileAttributes
            attrs) throws IOException{
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }
});
```

Recap

- 9.1 I/O Streams
- 9.2 Reading and Writing Binary Data
- 9.3 Object I/O Streams and Serialization
- 9.4 Working with Files