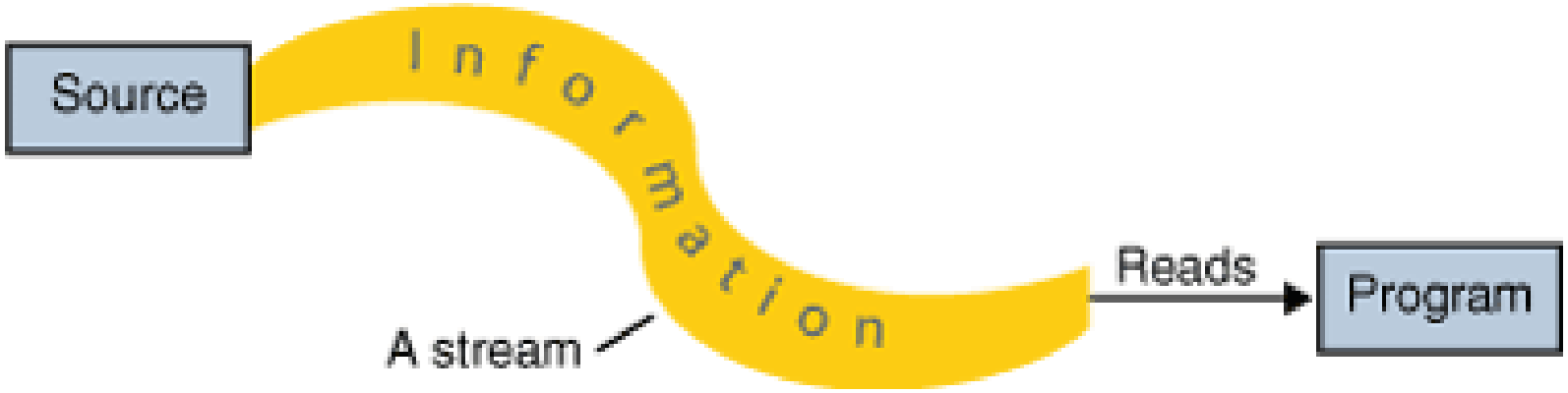
Chapter 08 - Streams

Introduction

- Often a program needs to bring in information from an external source or to send out information to an external destination.
- The information can be anywhere: in a file, on disk, somewhere on the network, in memory, or in another program.
- Also, the information can be of any type: objects, characters, images, or sounds.
- This chapter covers the Java platform classes that your programs can use to read and to write data.

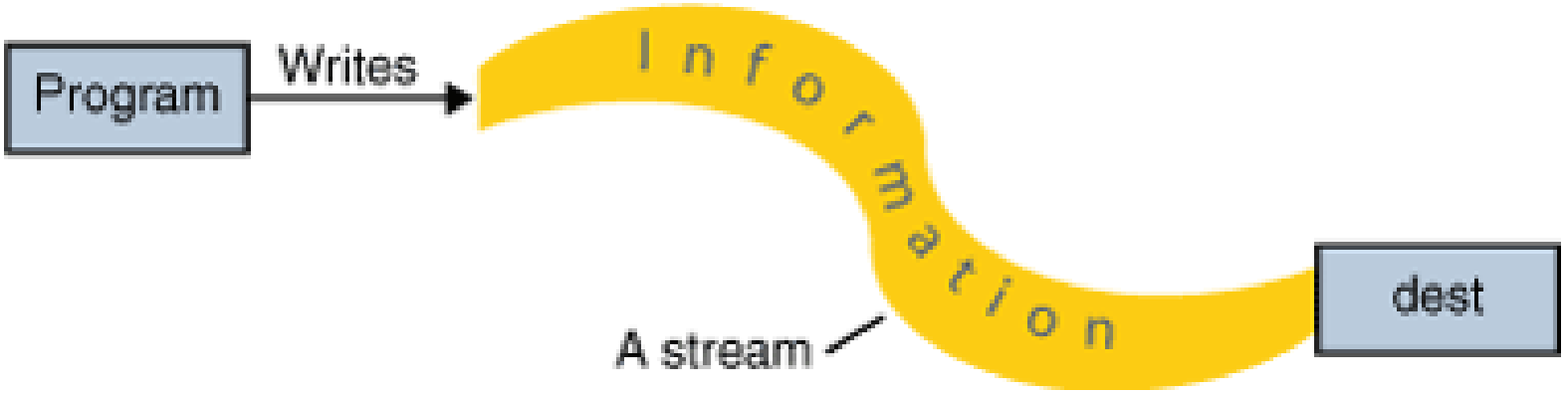
8.1 Overview of I/O Streams

Reading information into a program



- To bring in information, a program opens a *stream* on an information source (a file, memory, a socket) and reads the information sequentially.

Writing information out of a program



- Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially, as shown in the following figure.

Reading and Writing Algorithm for Data

- No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same.

Algorithms

Reading

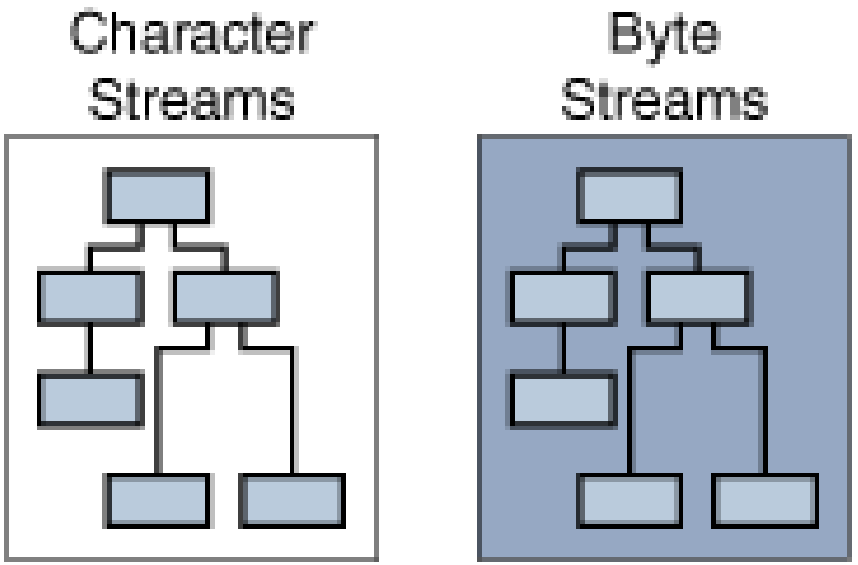
open a stream
while more information
    read information
close the stream

Writing

open a stream
while more information
    write information
close the stream

java.io Package

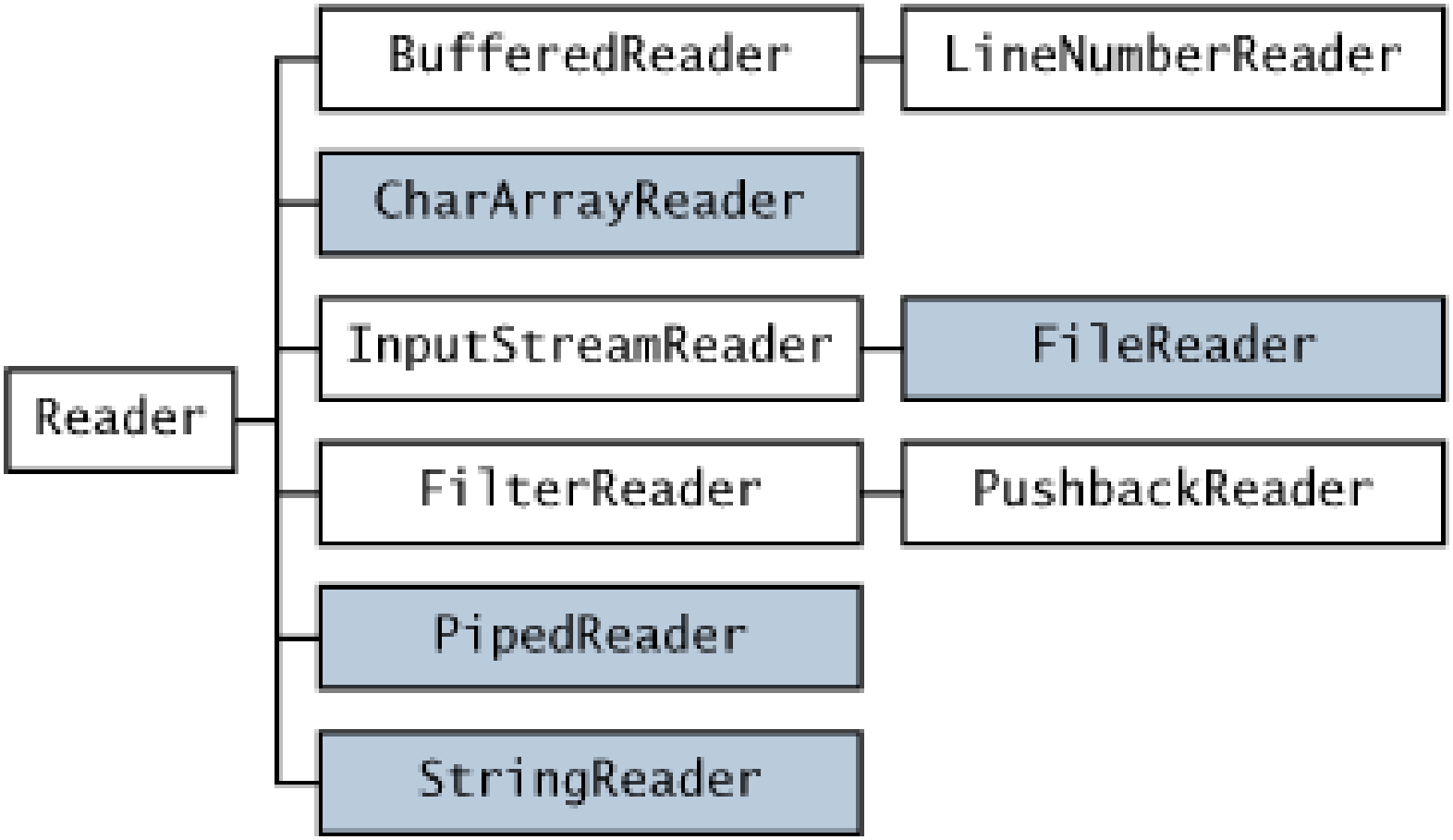
- contains a collection of stream classes that support these algorithms for reading and writing.



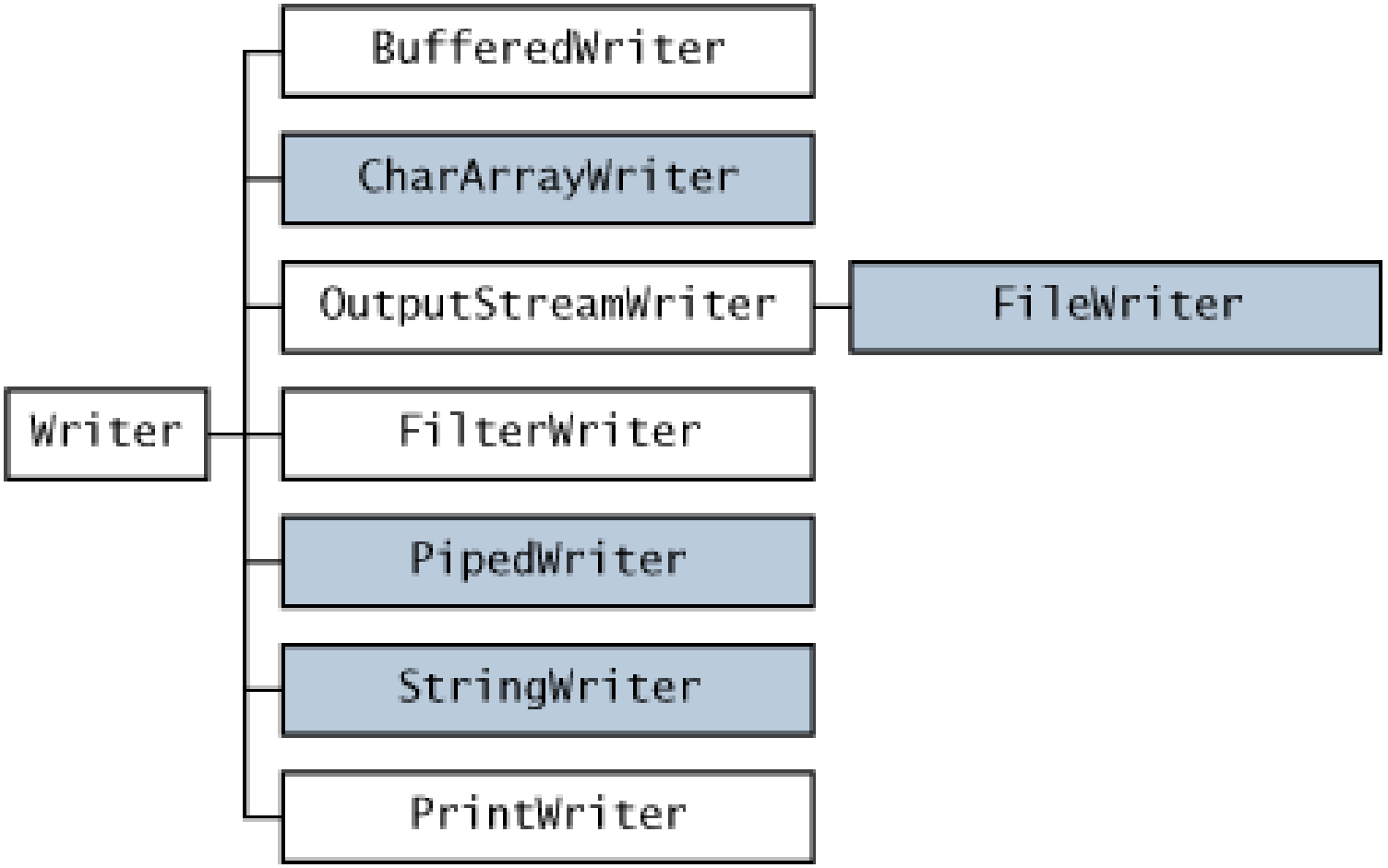
Character Streams

- Reader provides the API and partial implementation for *readers* — streams that read 16-bit characters
- Writer provides the API and partial implementation for *writers* — streams that write 16-bit characters.
- Subclasses of `Reader` and `Writer` implement specialized streams and are divided into two categories:
  - read from or write to data
  - perform some sort of processing

Subclasses of Reader



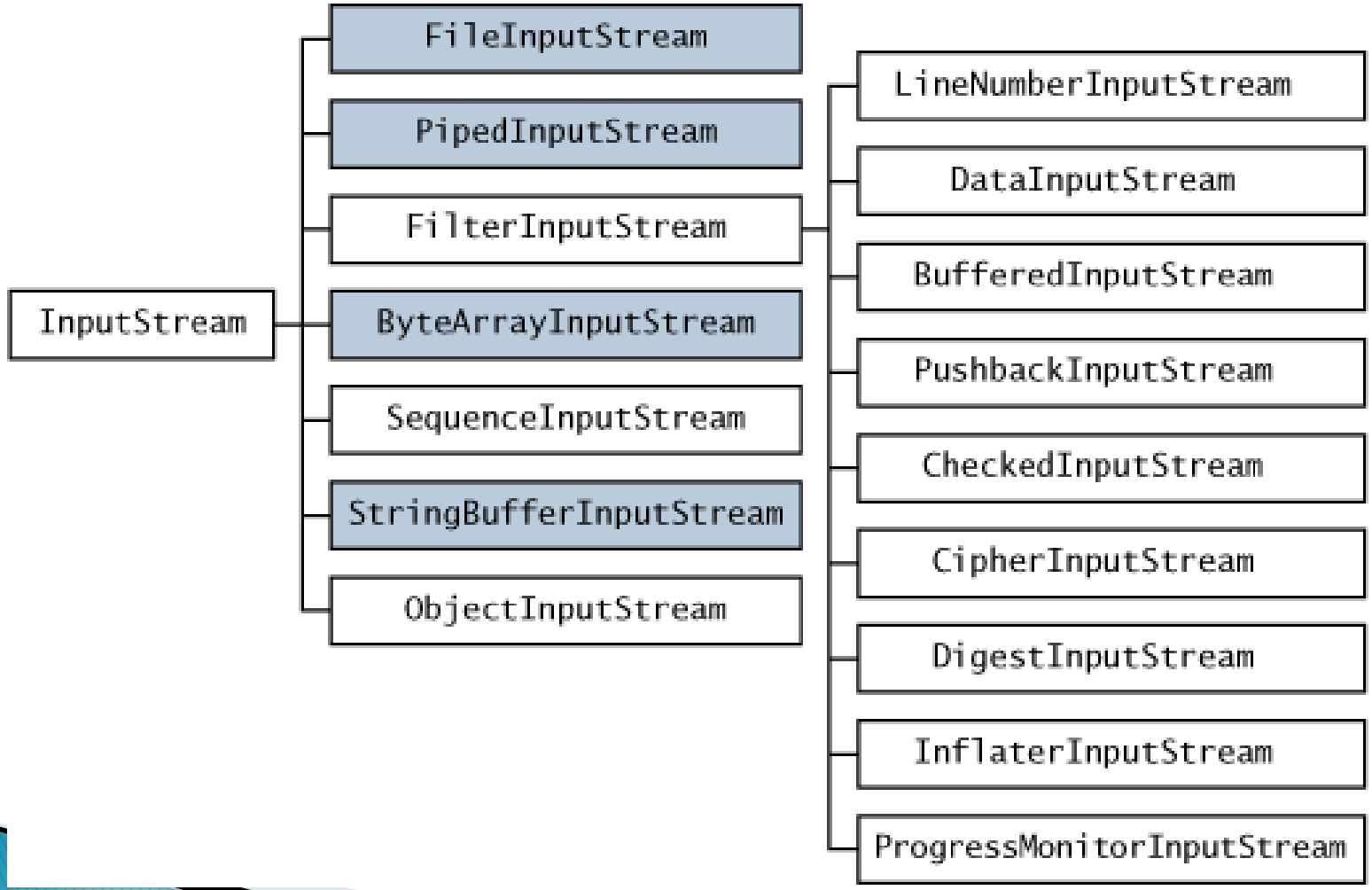
Subclasses of Writer



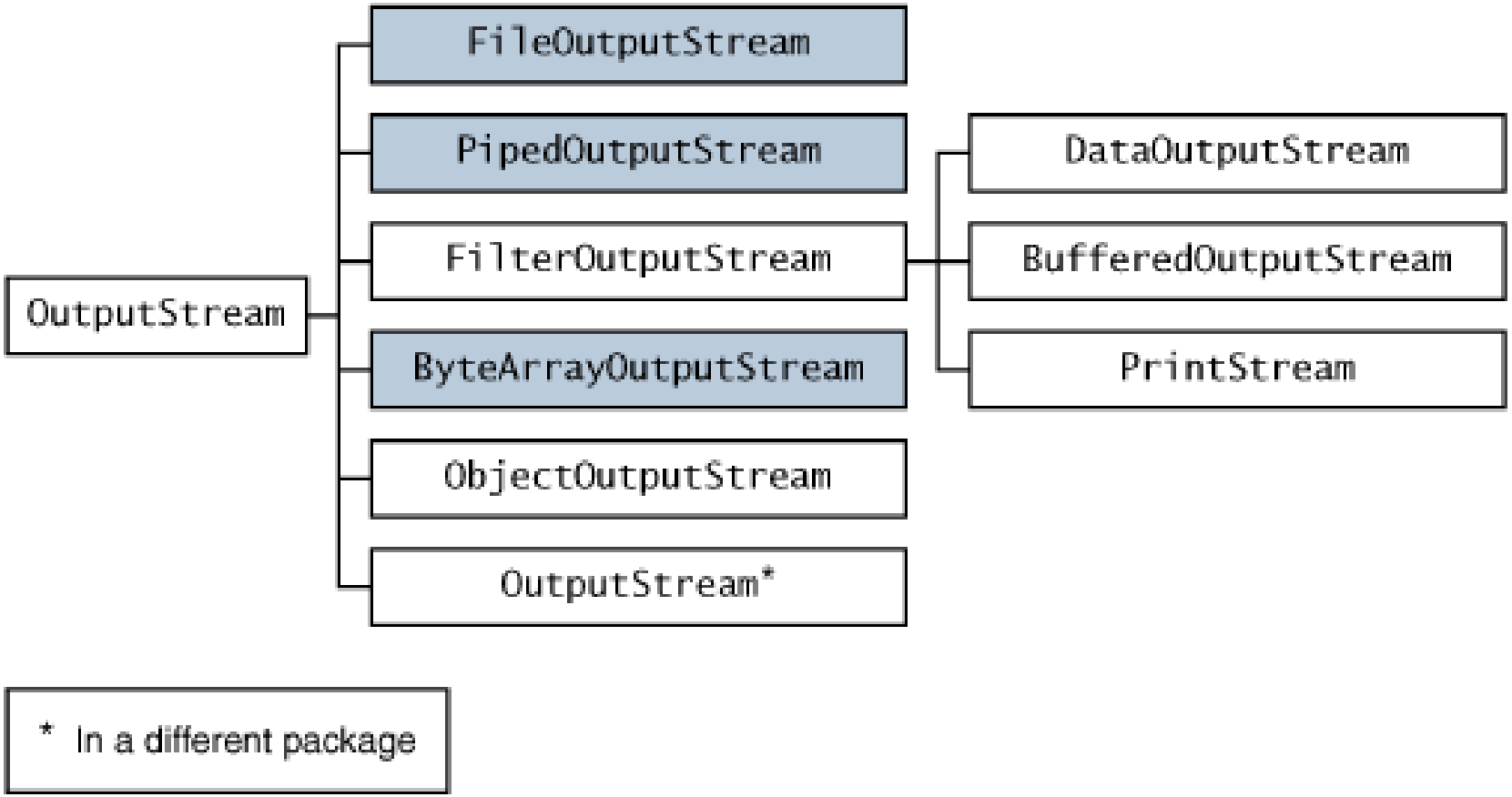
Byte Streams

- To read and write 8-bit bytes, programs should use the byte streams, descendents of `InputStream` and `OutputStream`
- `InputStream` and `OutputStream` provide the API and partial implementation for *input streams* (streams that read 8-bit bytes) and *output* streams (streams that write 8-bit bytes).
- These streams are typically used to read and write binary data such as images and sounds.

Subclasses of InputStream



Subclasses of OutputStream



Understanding the I/O Superclasses

- Reader contains these methods for reading characters and arrays of characters:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

- InputStream defines the same methods but for reading bytes and arrays of bytes:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

- Writer defines these methods for writing characters and arrays of characters:

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

- And OutputStream defines the same methods but for bytes:

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

- All of the streams are automatically opened when created.
- A program should close a stream as soon as it is done with it, in order to free up system resources.

8.2 How to Use File Streams

- The file streams - `FileReader`, `FileWriter`, `FileInputStream`, and `FileOutputStream` - each read or write from a file on the native file system.
- You can create a file stream from a file name in the form of a string, a `File` object, or a `FileDescriptor` object.

Understanding the File Class

- enable you to work with the file system on your machine
- represent a file or a directory in the file system
- navigate the file system
- perform several operations on the files in the file system

The Path Name

- Unix/Linux machine: `/java/scjp/temp`
- Windows machine: `C:\java\scjp\temp`
- A system-independent way of dealing with the path names
- a system-independent way -> an abstract path name:
  - A system-dependent prefix string, such as a disk-drive specifier or a forward slash (/)
  - A sequence of zero or more string names which may represent a directory or a file name, separated from the next name by a system-dependent separator (\ for Windows, / for Unix/Linux).

File Path Names and File Constructors

- The *path name* is an address of a file in the file system

Table 8-1. *Constructors for the File Class*

Constructor	Description
`File(String pathname)`	Creates an instance of the `File` class by converting the path name `String` to an abstract path name.
`File(String parent, String child)`	Creates an instance of the `File` class by concatenating the child `String` to the parent `String`, and converting the combined `String` to an abstract path name.
`File(File parent, String child)`	Creates an instance of the `File` class by constructing an abstract path name from the abstract path name of the parent `File`, and the `String` path name of child.

Example

Listing 8-1. *TestFileConstructors.java*

```
1.  import java.io.*;
2.  class TestFileConstructors {
3.   public static void main (String[] args){
4.    try{
5.       File f1 = new File("java/scjp");
6.       File f2 = new File("java/scjp", "temp/myProg.java");
7.       File f3 = new File(f1, "temp/myProg.java");

8.       System.out.println("path for f1: " + f1.getCanonicalPath());
9.       System.out.println("path for f2: " + f2.getCanonicalPath());
10.      System.out.println("path for f3: " + f3.getCanonicalPath());
11.   }catch (IOException ioe){
12.       ioe.printStackTrace();
13.   }
14.  }
15. }
```

File Path Names and File Constructors (cont.)

- The `getCanonicalPath()` method returns the system-dependent abstract path name.
- The above program was executed in the C:\temp directory on a Windows machine and the output was as follows:

```
path for f1: C:\temp\java\scjp
path for f2: C:\temp\java\scjp\temp\myProg.java
path for f3: C:\temp\java\scjp\temp\myProg.java
```

Notes about the Abstract Path Name

- We did not care what system it was while entering the path name
- Abstract path names returned by the `getCanonicalPath()` method are system dependent
- The system-dependent disk-drive symbol (`C:\` in this case) has been prefixed to the path
- If you do want to use the Windows-based file separator, you should use double backslash (`\\`) as a separator

Navigating the File System

- When you create an instance of the class `File`, it does not create a file in the file system
- After you've created an instance of the File class:
  - navigate the file system,
  - create a file,
  - and perform several operations

Some commonly used methods of the File class

- `boolean canRead()`
- `boolean canWrite()`
- `boolean createNewFile()`
- `boolean delete()`
- `boolean exists()`
- `String getAbsolutePath()`
- `String getCanonicalPath()`
- `String getName()`
- `String getParent()`

Some commonly used methods of the File class (cont.)

- `boolean isAbsolute()`
- `boolean isDirectory()`
- `boolean isFile()`
- `String[] list()`
- `String[] listFiles()`
- `boolean mkDir()`
- `boolean mkDirs()`
- `boolean renameTo(File <newName>)`

Listing 8-2. *FileNavigator.java*

```
1.  import java.io.*;
2.  class FileNavigator {
3.   public static void main (String[] args){
4.     String treeRoot = "."; //default root
5.     if (args.length >=1)treeRoot=args[0];
6.    File rootDir = new File(treeRoot);
7.    System.out.println("Root of navigation:" + rootDir.getAbsolutePath());
8.   // check if the root exists as a directory.
9.   if(!(rootDir.isDirectory())){
10.    System.out.println("The root of the naviagtion subtree does not exist
           as a directory!");
11.    System.exit(0);
12.    }
13.      FileNavigator fn = new FileNavigator();
14.      fn.navigate(rootDir);
15.   }

16.   public void navigate(File dir){
17.      System.out.println(" ");
18.      System.out.println("Directory " + dir + ":");
19.      String[] dirContent = dir.list();
20.      for (int i=0; i<dirContent.length; i++){
21.        System.out.print(" " + dirContent[i]);
22.        File child = new File(dir, dirContent[i]);
23.        if(child.isDirectory())navigate(child);
24.      }
25.   }
26. }
```

Copy Program

```
import java.io.*;
public class Copy {
  public static void main(String[] args) throws
  IOException {
      File inputFile = new File("farrago.txt");
      File outputFile = new File("outagain.txt");
      FileReader in = new FileReader(inputFile);
      FileWriter out = new FileWriter(outputFile);
      int c;
      while ((c = in.read()) != -1)
            out.write(c);
      in.close();
      out.close();
  }
}
```

CopyBytes Program

```
import java.io.*;
public class CopyBytes {
  public static void main(String[] args) throws IOException
  {
      File inputFile = new File("farrago.txt");
      File outputFile = new File("outagain.txt");
      FileInputStream in = new FileInputStream(inputFile);
      FileOutputStream out = new
      FileOutputStream(outputFile);
      int c;
      while ((c = in.read()) != -1)
             out.write(c);
      in.close();
      out.close();
  }
}
```

8.3 Working with Filter Streams

- A filter stream filters data as it's being read from or written to the stream:
  - `FilterInputStream`
  - `FilterOutputStream`
- A filter stream is constructed on another stream (the *underlying* stream).
- The `read()` method in a readable filter stream reads input from the underlying stream, filters it, and passes on the filtered data to the caller.
- The `write()` method in a writable filter stream filters the data and then writes it to the underlying stream.
- Some streams buffer the data, some count data as it goes by, and others convert data to another form.

Using Filter Streams

- To use a filter input or output stream, attach the filter stream to another input or output stream when you create it.
- For example, you can attach a filter stream to the standard input stream, as in the following code:

```
BufferedReader d = new BufferedReader(new
  InputStreamReader(System.in));
String input;
while ((input = d.readLine()) != null) {
  ... //do something interesting here
}
```

How to Use DataInputStream and DataOutputStream

- It features an example, `DataIODemo`, that reads and writes tabular data (invoices for merchandise).
- The tabular data is formatted in columns separated by tabs:
  - the sales price,
  - the number of units ordered,
  - and a description of the item.
- Conceptually, the data looks like this, although it is read and written in binary form and is non-ASCII:

```
19.99     12     Java T-shirt
9.99      8      Java Mug
```

```
// DataIODemo (1)

import java.io.*;
public class DataIODemo {
public static void main(String[] args)
  throws IOException {
  // write the data out
  DataOutputStream out = new
  DataOutputStream(new
  FileOutputStream("invoice1.txt"));
  double[] prices = { 19.99, 9.99, 15.99,
  3.99, 4.99 };
  int[] units = { 12, 8, 13, 29, 50 };
  String[] descs = { "Java T-shirt", "Java
  Mug", "Duke Juggling Dolls", "Java Pin",
  "Java Key Chain" };
```

```
// DataIODemo (2)
for (int i = 0; i < prices.length; i ++) {
  out.writeDouble(prices[i]);
  out.writeChar('\t');
  out.writeInt(units[i]);
  out.writeChar('\t');
  out.writeChars(descs[i]);
  out.writeChar('\n');
}
out.close();
```

```
// DataIODemo (3)
// read it in again
DataInputStream in = new DataInputStream(new
  FileInputStream("invoice1.txt"));
double price; int unit; StringBuffer desc;
double total = 0.0;
String lineSepString =
  System.getProperty("line.separator"); char
  lineSep =
  lineSepString.charAt(lineSepString.length(
  )-1);
```

```
    try {
      while (true) {
            price = in.readDouble();
            in.readChar(); // throws out the tab
            unit = in.readInt();
            in.readChar(); // throws out the tab
            char chr;
            desc = new StringBuffer(20);
            while ((chr = in.readChar()) != lineSep)
                        desc.append(chr);
                  System.out.println("You've ordered "
    + unit + " units of " + desc + " at $" + price);

            total = total + unit * price;
      }
    }
    catch (EOFException e) { }
    System.out.println("For a TOTAL of: $" + total);
    in.close();
  }
}
```

Investigation

- DataOutputStream, like other filtered output streams, must be attached to another OutputStream.
- In this case, it's attached to a FileOutputStream that is set up to write to a file named invoice1.txt:

```
DataOutputStream out = new
  DataOutputStream( new
    FileOutputStream("invoice1.txt"))
```

- Next, DataIODemo uses DataOutputStream's specialized write() methods to write the invoice data contained within arrays in the program according to the type of data being written:

```
for (int i = 0; i < prices.length; i ++) {
  out.writeDouble(prices[i]);
  out.writeChar('\t');
  out.writeInt(units[i]);
  out.writeChar('\t');
  out.writeChars(descs[i]);
  out.writeChar('\n');
}
out.close();
```

- Next, DataIODemo opens a DataInputStream on the file just written:

```
DataInputStream in = new
  DataInputStream( new
  FileInputStream("invoice1.txt"));
```

- DataInputStream also must be attached to another InputStream; in this case, a FileInputStream set up to read the file just written, invoice1.txt.
- Then DataIODemo just reads the data back in using DataInputStream's specialized read() methods.

```
try {
  while (true) {
       price = in.readDouble();
       in.readChar(); //throws out the tab
       unit = in.readInt();
       in.readChar(); //throws out the tab
       char chr;
       desc = new StringBuffer(20);
       char lineSep =
              System.getProperty("line.separator").charAt(0);
       while ((chr = in.readChar() != lineSep) {
              desc.append(chr);
       }
       System.out.println("You've ordered " + unit + " units of "
  +
              desc + " at $" + price);
       total = total + unit * price;
  }
}
catch (EOFException e) { }
System.out.println("For a TOTAL of: $" + total);
in.close();
```

- Note the loop that `DataIODemo` uses to read the data from the `DataInputStream`. Normally, when data is read, you see loops like this:

```
while ((input = in.read()) != null) { . . . }
```

- The `read()` method returns a value, `null`, which indicates that the end of the file has been reached.
- Many of the `DataInputStream` `read()` methods can't do this, because any value that could be returned to indicate the end of file may also be a legitimate value read from the stream.
- For example, suppose that you want to use -1 to indicate end of file. Well, you can't, because -1 is a legitimate value that can be read from the input stream, using `readDouble()`, `readInt()`, or one of the other methods that reads numbers.
- So DataInputStreams read methods throw an `EOFException` instead. When the `EOFException` occurs, the `while (true)` terminates.

Program Output

- When you run the DataIODemo program you should see the following output:

```
You've ordered 12 units of Java T-shirt at $19.99
You've ordered 8 units of Java Mug at $9.99
You've ordered 13 units of Duke Juggling Dolls at $15.99
You've ordered 29 units of Java Pin at $3.99
You've ordered 50 units of Java Key Chain at $4.99
For a TOTAL of: $892.8800000000001
```