Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-eight)

8.2: I/O and inheritance in Java

The java.io package illustrates many of the principles we discussed in Module 5, "Class Inheritance and Polymorphism." There are four abstract classes: Reader, Writer, InputStream, and OutputStream, that provide rudimentary character- or byte-based I/O functions to read and write single characters (bytes) and arrays of characters. The byte-based classes are older, and use of the character-based classes is now preferred whenever textual information is being processed. (Remember that Java uses Unicode characters, **16** bits in size. Using the character streams makes your code easier to sell in an international market.)

Table: java.jo Classes and Subclasses

Class	Description	Subclasses
java.io.InputStream	Abstract superclass of all byte (8-bit) input streams	ByteArrayInputStream FileInputStream FilterInputStream ObjectInputStream PipedInputStream SequenceInputStream StringBufferInputStream
java.io.OutputStream	Abstract superclass of all byte (8-bit) output streams	ByteArrayOutputStream FileOutputStream FilterOutputStream ObjectOutputStream PipedOutputStream
java.io.Reader	Abstract superclass of all double-byte Unicode character input streams	BufferedReader CharArrayReader FilterReader InputStreamReader LineNumberReader PipedReader StringReader
java.io.Writer	Abstract superclass of all double-byte Unicode character output streams	BufferedWriter CharArrayWriter FilterWriter OutputStreamWriter PipedWriter PrintWriter StringWriter

In Reader, the method

```
public abstract int read(char cbuf[],
                         int off,
                         int len) throws IOException ...
```

is abstract, while in Writer,

```
public abstract void write(char cbuf[],
                           int off,
                           int len) throws IOException ...
```

is abstract. Similarly, InputStream contains the method

public abstract int read() throws IOException ...

and OutputStream contains the method

```
public abstract void write(int b) throws IOException ...
```

As a result, you can't create a Reader, Writer, InputStream, or OutputStream directly. What makes these classes useful is that you can define a subclass of Reader (for example) that implements read() to read data out of a disk file. Such a class might be called FileReader. In fact, Java provides such a class, and that is in fact the name: java.io.FileReader. There is also a StringReader, which lets you treat a string as an I/O source; and an InputStreamReader, which lets you adapt byte-based and character-based streams to work together.

The networking classes provide instances of special network stream classes. (These latter are Java implementation details. The actual classes are not public, so you cannot create them directly or even know their names; but some network-related functions return instances of them to you nonetheless.) Methods that are declared to return instances of some base class (especially an abstract base class) but actually return instances of a derived class are often called Factory methods, because they produce objects. The associated design pattern is called the Factory pattern. Using the Factory pattern is an excellent way to hide implementation details from parts of a program that don't need to know them. This kind of hiding tremendously simplifies program maintenance.

Making all types of stream classes inherit from a few generic abstract classes is a marvelous system, because you can easily write methods that read data from a generic Reader or InputStream object. When you then invoke such a method, you can pass to it a FileReader or NetworkInputStream object, and the method will take its input from a file, or from the network, or from the console, with equal ease. Your method will call read( ), for example, and the particular object's implementation of read() will be invoked. It may load data over the Internet, or it may read it from a file, or from the keyboard, but your method won't need to know. Polymorphism at work!