Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Network Programming

Chapter Overview

- How do entities on one computer communicate with entities on another computer?

Many modern applications involve multiple computers. This chapter introduces Java's primary mechanisms for making such interaction possible: communication channels, or streams, over which information can be transmitted. Transmission over these channels is often mediated by a Lector -- one who reads -- and/or a Scribe -- one who writes -- on behalf of an entity. Communication may occur across a network, between co-located entities, or with persistent storage resources such as a File on disk. The stream abstraction gives these diverse kinds of communication a uniform interface.

In this chapter, we present a series of Lector/Scribes, initially relying only on local resources, ultimately establishing and controlling a network connection. We conclude with a discussion of a multi-threaded server and a brief look at the role of a server in a network architecture.

A Readable Writeable Channel

Two entities often need to communicate with one another. We have seen how this can be accomplished using direct method invocation. But that kind of communication requires that the calling object know the identity of the method's owner. This is the equivalent of having a face-to-face conversation: You must know to whom you are speaking. In this chapter, we explore a more abstract communication mechanism that uses intermediate objects -- called streams -- to allow one entity to communicate with another indirectly. Stream communication is more like talking on the telephone. The same device can be used to interact with many different people -- or entities -- without requiring direct contact. Streams similarly provide a uniform interface that can be used to communicate indirectly with a wide variety of objects or resources.

A stream -- or, more properly, a **stream of values** -- is an abstractly defined resource containing a sequence of values that can be processed, one by one. Streams come in two flavors: **input** streams, which support the reading of values, and **output** streams, which support their writing. In other words, an input stream is a stream from which these values can be read, one by one, in order; an output stream is such a resource to which values can be written one by one. In this chapter, we will concentrate on stream-like objects and how they are used.

Tin Can Telephones

One way to think about a stream is to consider the tin can telephones many of us played with as children:

> Take two tin cans with one end removed from each. Punch a whole in the center of the intact end of each can. With a long piece of string, thread the two cans so that their flat ends face each other. Tie knots in the ends of the string. Pull the string tight, so that it is stretched between the two cans. Talk into one can; have someone else listen at the other.
>
> [Pic of tin can telephone; pic of Tweedles talking; stream ends marked]

This is a simple device that allows you to put something in one end and allows someone else to retrieve it at the other end. The end into which Tweedledum is speaking is like an output stream. The end to which Tweedledee is listening corresponds to an input stream. Anything Tweedledum writes to the output stream can be heard by Tweedledee (reading) the input stream. [Footnote: Note, though, that the communications medium itself -- the tin can telephone -- isn't an input stream *or* an output stream. The medium has two ends, one of which is an input stream, and the other of which is an output stream. In the case of the tin can telephone, these roles might be marked on the two cans: "Listen here" on one can, "Speak here" on the other. In the case of streams, there's less room for confusion. A stream is either an input stream or an output stream, and the two are not interchangeable.]

One nice property of this system is that the tin can telephone doesn't rely on face to face contact between the conversationalists. Using a stream, a Java entity can talk to all different kinds of things without needing to know much about those things. Communication relies on properties of the stream, rather than on properties of the thing at the other end. It also means that the communicators don't have to be directly in contact, as long as each holds one end of the tin can telephone, or stream. For example, Tweedledum can use the same kind of device to talk to the Jabberwock, even though Tweedledum knows far better than to approach the Jabberwock face to face.

The tin can telephone story so far works well when Tweedledum wants to communicate something to Tweedledee. But what happens when Tweedledee has something to say as well? We can accomplish this using the same approach. But in order to have simultaneous two-way communication, it is useful to have *two* tin can telephones. Then Tweedledum listens to one and talks into the other; Tweedledee listens to the one Tweedledum talks into, and talks into the one Tweedledum is listening to.

> [Pic of Tweedles talking on **two** tin can telephones. Also the view from Tweedledee's perspective.]

In the same way, Java streams often come in pairs. One -- the -dum to -dee route -- is an output stream for -dum and an input stream for -dee. The other -- -dee to -dum -- is output stream for -dee and an input stream for -dum. If you are standing at one end -- like Tweedledee -- you are holding one input stream (from which you can read) and one output stream (to which you can write).

Streams

In this book so far, we have talked about cs101.io.Console, a particular concrete resource that behaves like one end of the two-stream configuration. Console behaves like an input stream -- through its `readln()` method -- and also like a separate output stream -- through its `println(String)` method. The Channels used in interlude 1 are another example of stream-like behavior, though the methods provided for reading and writing are somewhat different from Console's. Like those channels, many kinds of Java streams are used to connect two entities.

Streams can also be used to interact with things that may be more outside of your program, such as a File -- information stored on your disk -- or a network connection. Each different kind of stream-like resource has a slightly different set of (read and write) methods, and each has a very different kind of behavior at the "other" end of the resource. Writing to the Console makes the information appear on your Java console, i.e., on your screen. Writing to a file stores the information away on your disk for later retrieval. Writing to a network connection sends the information to another computer. But from the read/write end -- from Tweedledee's perspective -- the stream makes these resources seem fundamentally similar.

A stream is general way to think about each of these connections. Individual connections implement different methods to actually make stream communication possible. No matter the differences among them, these methods are generally used in stereotypical ways. The ways that **reading** -- extracting information from an input stream -- and **writing** -- depositing information in an output stream -- are used within a program is the topic of the first part of this chapter. Even the general properties of read and write operations are shared by most of these stream-like connections.

In the first part of this chapter, we will simply use the notation *`inStream`*`.`*`read`*`()` and *`outStream`*`.`*`write`*`( `*`message`*` )` whenever we are accessing a read or write method. All of the code in that portion of the chapter will work if the text *`inStream`*`.`*`read`*`()` is replaced with the text `cs101.io.Console.readln()` and the text *`outStream`*`.`*`write`*`( `*`message`*` )` is replaced with the text `cs101.io.Console.println(String)`. We use the more generic notation to indicate that other substitutions are equally possible. In the latter parts of this chapter, we will introduce other kinds of streams and talk about other code that might be used to replace *`inStream`*`.`*`read`*`()` and *`outStream`*`.`*`write`*`(Object)`.

Using A Channel

This section develops a very general set of classes capable of reading from and writing to a general read/write resource. It is tempting to call the thing that does the reading a Reader and the thing that does the writing a Writer. However, Java has reserved those names for other classes, described below. Instead, we will call the thing that does the reading a **Lector** (meaning "one who reads") and the thing that does the writing a **Scribe** ("one who writes"). We will define the interfaces for these classes first.

Although the details will vary from application to application and will depend in some part on the kind of resource from which you are reading or to which you are writing, the general pattern of interaction with a read/write resource is similar. In this section, we will develop a fairly general class whose instances are capable of reading from and writing to resources of this sort. In the succeeding sections, we will modify that class to tailor it to the kind of read/write resource represented by a network connection.

For Writing

Let's say that you have a thing, say, `message`. You want to write `message` to your output stream, *`outStream`*. Accomplishing this is as easy as it sounds:

```
outStream.write( message );
```

Let's look more carefully at what is going on here. Consider the write end of a channel. This is a stream that implements a *`write`* method, such as Console.println(String). To use this *`outStream`*`.`*`write`*`(Object)` method, all that we need to do is invoke it with the appropriate Object.

The write method of a resource like this one is just a server push client. It accepts information when the server writes it. To use such a write method, you must build code that acts as a server push server towards the write method. It must explicitly take action. That is exactly what happens when we invoke

```
outStream.write( message );
```

Flushing Out the Stream

One detail is worth noting. Remember that the output stream is one end of something that has another end (like the tin can telephone). When an output stream's write method is invoked, it causes the thing written -- the message -- to pass into the stream. It does not, however necessarily cause this thing to be available at the other end of the stream. This is like speaking into a tin can telephone with a noticeable delay between the ends. It is quite possible that the message may be "inside" the stream. It will eventually appear at the other end, but not neccessarily when the writer expects it to do so.

[Pic of balls stuck inside a pipe. Caption: When you write something to an output stream, it can get stuck "inside" the stream. A call to the stream's `flush()` method will push these objects through.]

If it is really important that the information you wrote to the stream not get stuck inside, you can use a special method, called `flush()`, to push the information through. When an output stream's `flush()` method is invoked, anything that's been written gets pushed along through the stream, so that it appears at the other end. If there are multiple things that have been written, they appear at the other end one by one, in the correct order; `flush()` doesn't change anything, it just gets things moving along.

Why might something get stuck inside a stream? Imagine that you have a carton that can hold twelve eggs. You go to the henhouse and pick up an egg. (You "write" the egg to the carton.) Back in the house, the cook is waiting for eggs to make breakfast. But it is silly for you to go back to the house with just one egg if the cook can't start until s/he has enough eggs for breakfast. So you pick up another egg and write it to the carton. You keep going until you have a full carton of eggs.

Streams can work the same way. They can wait until there is a group of information to be sent, then send the whole collection at once. Just as it saves you time to collect a carton full, it can make more efficient communication to wait for a full "packet" of information.

When you invoke a `flush()` method, that causes the information to be sent, regardless of how much is waiting. In the egg collecting example, it would make you go to the house even if you'd only collected two eggs so far. Then you'd have to go back to the henhouse and collect some more. Unneccessary `flush()`es are wasteful, just as in this example.

On the other hand, a judicious `flush()` every now and then can be beneficial. What if you were determined only to return to the house when you had a full dozen eggs? But say that today the hens layed only eight eggs. You might stay in the henhouse until tomorrow rather than return with a partially full carton. In this case, a `flush()` would be just the right thing: It would get the eight eggs you had collected where they needed to go, rather than waiting for the next four eggs (that might never come).

A Scribe Example

So far, we have seen how writing to an output stream works. We can encapsulate this knowledge inside a method that takes an object and sends it out over the output stream. The interface for an object supplying this behavior might read:

```
public interface Scribe {
    public void send( MessageType m );
}
```

[Footnote: Note that we are being deliberately cagey about the type of object that can be written (or read). This is because that depends on the specific kind of stream that you're dealing with. Nothing in this section relies on the specific kind of stream or type of message.]

An example implementation of this method (to be encapsulated in an appropriate class) might be:

```
public void send( String m )
{
    Console.println( m );
}
```

In other words, a Scribe is an object that keeps track of its output stream and, on (send) request, writes the object to be sent to the stream.

A GenericScribeImpl class implementing this interface would need an output stream. It could then simply use that stream's write method on demand. If it is important that our writing not be delayed, we might add a `flush()` invocation to the send method as well.

```
public class GenericScribeImpl implements Scribe
{

    private OutputStreamType outStream;

    public GenericScribeImpl( OutputStreamType outStream )
    {
        this.outStream = outStream;
    }

    public void send( MessageType m )
    {
        outStream.write( m );
        outStream.flush();    // (maybe)
          }

}
```

Instances of this Scribe object are suitable for use in event-driven programs. For example, whenever something happens that needs to be communicated, the Scribe's send method could be invoked. It would then write the relevant communication to its output stream.

For example, if we have a TextField and a Scribe

```
TextField textField = new TextField();
Scribe scribe = new GenericScribeImpl();
```

we might connect them by having the Scribe write out the text in the TextField each time the return key is hit. (Recall that hitting the return key in a TextField triggers an ActionEvent).

This can be accomplished using an actionPerformed method that says:

```
public void actionPerformed( ActionEvent ae )
{
    this.scribe.send( this.textField.getText() );
}
```

[Footnote: We've omitted a few details from this example. First, the actionPerformed method is embedded in a ScribeListener class whose full definition is:

```
public class ScribeListener implements ActionListener
{

    private TextField textField;
    private Scribe scribe;

    public ScribeListener( Scribe scribe, TextField textField );
    {
        this.scribe = scribe;
        this.textField = textField;
    }

    public void actionPerformed( ActionEvent ae )
    {
        this.scribe.send( this.textField.getText() );
    }

}
```

An instance of this ScribeListener class is then used to connect the TextField with the appropriate Scribe.

```
textField.addActionListener( new ScribeListener( scribe,
                                                 textField ) );
```

]

For Reading

Writing to an output stream is fairly straightforward. Reading from an input stream is somewhat more complicated. To help us read from an input stream, we will define a class called Lector: "one who reads".

The innermost portion of the Lector says something parallel to the Scribe. We certainly want to invoke the stream's read method:

```
inStream.read()
```

Immediately, we are faced with the first complication. What should the Lector do with the message read from its input stream? There are many possibilities, depending on what you want your Lector to do. For example, the Lector could just let the user know that it has read the message (through the Java console):

```
Console.println( "Lector:  just read "
                 + inStream.read().toString() );
```

This line of code reads the message from the input stream, finds its printable equivalent using toString(), and then prints this version to the Java console. It is one example of a thing that we might want a Lector to do over and over again. We will return to this issue and see more complex solutions below.

The second difference between reading and writing is that the Lector must be an active object. The Scribe is automatically invoked whenever an object is available to be written. But the Lector must check to see for itself whether an object is available for reading. The input stream is passive.[Footnote: So is the output stream. But the Scribe is activated by the thing that asks it to send.]

The Lector must invoke the input stream's read method by itself. This means that an instruction follower has to come from the Lector itself. Not only that, but the instruction follower of the Lector may wind up spending a lot of time waiting for something to become ready to read. When there is no such value, the read request doesn't return. The instruction follower that executed it is simply stuck waiting. This is because reading is a blocking operation.

Reading and Blocking

The Lector invokes the input stream's read method -- asking for the next value -- whether or not there's a value ready to be read. It is this ready-or-not condition that poses the real issue. When there is no value to return, the Lector may get stuck waiting for one.

The read operation on almost any kind of stream is called a **blocking** read. This means that it will not return until the appropriate information becomes available. For example, if you type something on the Java console, ending with the return key, Console's `readln()` method will return this String. If you invoke `Console.readln()` again, it will return the next return-key-terminated String that you type. But what if you haven't (yet) typed another return-key-terminated String? In this case, the `readln()` method will not return. The method invocation continues until an appropriate String becomes available; the Console's `readln()` method waits for a carriage return. This waiting -- for the necessary information -- is called blocking.

Because stream reading methods almost always are blocking methods, they generally need to be invoked by a dedicated instruction follower, i.e., one that can sit around and wait until the read invocation can complete. The blocking read method itself is essentially a client pull server: it provides the information on request. To interact with a blocking read method, you must write a client pull client: active code that invokes the read method on a regular basis.

The fact that the Lector might get stuck waiting for an object to become ready -- that the read might block -- means that the Lector must have its very own dedicated instruction follower whose job is to wait for the read. This instruction follower can't be expected to get much of anything else done, because it might spend a long time waiting for the read invocation to un-block. We need a dedicated Thread -- instruction-follower -- who can afford to spend its time waiting. This is like sending one person to stand in line while the others

do something. You don't want to tie everyone up standing in line, and if you only have one person, you can't afford to block (wait); you need to hire someone to wait for you.

We can resolve this issue by dedicating an instruction-follower to the read task. This is a job for an animate object.

A Lector Example

We are now ready to write the Lector class. The Lector, like the Scribe, keeps track of a stream. But instances of this class, unlike those of Scribe, are animate objects, each with its own AnimatorThread. A Lector can afford to block each time it calls its input stream's read() method, because it has a dedicated instruction follower. If the instruction follower's invocation of read() blocks, it is not a problem because this instruction follower is not expected to be doing anything else other than reading from the input stream.

```
public class Lector implements Animate
{

     private InputStreamType inStream;
     private AnimatorThread mover;

     public Lector( InputStreamType inStream )
     {
         this.inStream = inStream;

         this.mover = new AnimatorThread( this );
         this.mover.start();
     }

     public void act()
     {
         Console.println( "Lector:  just read "
                          + this.inStream.read().toString() );
     }

}
```

This code shows how a Lector can print the read message to the Console. But this isn't always what we'll want to do when something is read from an input stream. For example, we might want to do a dispatch on case, depending on what the input it reads is. This might involve some giant conditional with `inStream.read()` as the switch expression. Or we might want to pass the new message around to everyone we know, as in the broadcast server towards the end of this chapter.

This situation should sound vaguely familiar. Something happens: the Lector reads something from the input stream. This is an event. There are many different ways that this event could be handled. In fact, it's not clear that the Lector should do anything itself. Maybe what the Lector should do is to delegate this responsibility to some other object. This could be done using the simple event handling of chapter 15 or the more complex event delegation of chapter 16.

Paralleling chapter 16, let's define an interface for this separate event handler object:

```
public interface LectorListener
{

     public void messageRead( MessageType m );

}
```

An example LectorListener class -- one whose instances simply print their message to the Java console -- might be:

```
public class LectorPrinter
{

     public void messageRead( MessageType message )
     {
         Console.print( "Lector:  Just read:  ");
         Console.println( message.toString() );
     }

}
```

Now we'll need a way for the LectorListener to register with the Lector. We will assume just one LectorListener per Lector for now, though we could certainly do otherwise (e.g., using a Vector). The modifications are underlined.

```
public class GenericLector implements Animate
{

     private InputStreamType inStream;
     private AnimatorThread mover;

           private LectorListener ll;

     public GenericLector( InputStreamType inStream )
     {
         this.inStream = inStream;

         this.mover = new AnimatorThread( this );
         this.mover.start();
     }


           public void addLectorListener( LectorListener ll )
     {
           this.ll = ll;
     }

     public void act()
     {
         this.ll.messageRead( this.inStream.read() );
     }
```

```
}
```

Encapsulating Communications

We have seen how to write the code for a generic Scribe, a class that manages writing to an output stream. We have also seen how to write a generic Lector that actively reads from an input stream. Often, it is useful to package these two functions together. In the single resulting class, we consolidate all management of communications with a single remote entity. This object may add functionality. It may, for example,l do some packing or unpacking for us (if we don't want to and receive objects in the same form that we use them within our program). It may do other bookkeeping, for example recording what information comes in or timestamping it. Such a communications manager might also establish the streams initially, handle exceptions, and otherwise provide a single point of contact for the rest of the entities with which it interacts directly. From within its local community, this entity provides an interface to the remote entity.

[Pic of local communications manager]

These two classes can be combined into a single class:

```
public class LectorScribe implements Scribe, Animate
{

    private OutputStreamType outStream;
    private InputStreamType inStream;
    private AnimatorThread mover;
    private LectorListener ll;

    public LectorScribe( OutputStreamType outStream,
                         InputStreamType inStream )
    {
        this.outStream = outStream;
        this.inStream = inStream;
        this.mover = new AnimatorThread( this );
        this.mover.start();
    }

    public void act()
    {
        this.ll.messageRead( this.inStream.read() );
    }

    public void send( MessageType m )
    {
        this.out.write( m );
        this.out.flush();    // (maybe)
          }

    public void addLectorListener( LectorListener ll )
    {
        this.ll = ll;
    }
```

```
    }
```

Note that this class will often have (at least) two instruction-followers active in it: the AnimatorThread named by this.mover and whatever Thread invokes this object's send method (from outside this class).

Real Streams

So far, we have been discussing input streams and output streams as hypothetical idealized objects. In Java, there are a series of classes that actually implement this stream behavior. In this section ,we will look at the Java classes that implement stream behavior. All of the classes described in this section are defined in the package java.io unless otherwise specified. Further information on many of these classes are included in the Java IO Quick Reference appendix.

Abstract Stream Classes

Java actually has four abstract classes that implement stream behavior: two input stream types, from which you can read, and two output stream types, from which you can write. The input stream classes are called InputStream and Reader. The output stream classes are OutputStream and Writer. In this chapter, we use the term stream to refer generically to all four of these classes. Each of these classes is abstract, meaning that any instance of that class is actually an instance of some subclass. They are all defined in the package java.io.

A stream is a resource containing a sequence of values. The values in the resource underlying an InputStream or an OutputStream are stored as bytes, i.e., eight bit pieces of data. The values in the resouce underlying a Reader or Writer are stored as chars, i.e., sixteen bit data. Certain contexts produce byte streams, while others produce char streams. You do not need to worry about the differences, but you do need to keep track of which one you have.

Every stream has a `public void close()` method. This method frees up the underlying resources that have been used to create this stream. When your program is done with a stream, it should call that stream's close() method. When your program shuts down, any open resources will be closed automatically; however, it is good practice to close your streams as soon as you are done with them.[Footnote: Although Java includes automatic garbage collection -- it will throw away your stream object if nothing in your program can possibly access it any more -- Java does not necessarily release the underlying system resource (i.e., the actual connection to a file or whatever else your stream is connected to) at that time.]

`InputStream`, `Reader` and their extensions support a variety of methods for reading. `InputStream`'s `read()` method returns a byte, while `Reader`'s returns a char. `OutputStream`, `Writer` and their extensions support methods for writing. The write method of `OutputStream` takes a byte as its argument.[Footnote: Actually, OutputStream's write method takes an int, but it only writes the low order byte of that int to the stream.] The write method of Writer takes either a char or an int or a String.

Each of the abstract stream classes has several subclasses that provide additional behavior. For example, some of these classes provide a wider range of methods, such as `public Object readObject()` and `public void println( String )`. You will often find it more useful to use one of these extended classes. Those classes are discussed in the next sections; their details are summarized in the Java IO Quick Reference appendix.

Many stream methods also potentially throw an exception. The most common exception to be thrown by a stream method is `IOException`.For example, when you go to read from a stream, if the underlying resource has somehow been corrupted, the read method may `throw IOException`. When using a stream method, you will often need to `catch` this exception. `IOException` also has several more specific subclasses, each applicable to a particular failure condition.

Decorator Streams

Java uses a technique called decoration to add features to streams. For example, suppose that you have an InputStream but have decided that you'd really rather have a Reader. Java has a class called InputStreamReader that is a special kind of Reader. Specifically, InputStreamReader's constructor takes an InputStream as an argument. The resulting InputStreamReader uses the same underlying stream resource as the InputStream argument, but the InputStreamReader is a Reader, not an InputStream:

> Suppose you have an `InputStream` called `in`, and execute
>
> ```
> Reader reader = new InputStreamReader( in );
> ```
>
> Now `reader.read()` returns the first char in the underlying input stream. The streams named `in` and `reader` use the same underlying input stream!

This pattern -- adding features by constructing a more sophisticated object around a simpler one -- is called **decoration**. Java streams make extensive use of decoration to add features. For example, you can now treat reader as you would any Reader, decorating it further using the appropriate constructors.[Footnote: There is, however, no way to make an InputStream from a Reader (or an OutputStream from a Writer).]

Some of the decorations that you might wish to apply to your stream include:

Buffering. This reads a larger group of data from the stream into some hidden storage, and then reads from that storage on demand. This is particularly useful when you are reading from a file or a network connection. Buffering is provided by the BufferedInputStream and BufferedReader classes. BufferedReader also has a particularly useful readLine() method that returns a whole String, up to but excluding the terminating newline.

Data. DataInputStream is a class whose instances provide a variety of read methods that allow you to read Java primitive data. These include readInt(), readBoolean(), etc. Note, however, that there is no corresponding DataReader class.

Objects. An ObjectInputStream is very much like a DataInputStream with the addition of a method for reading whole Java Objects: readObject(). However, only objects that implement the Serializable interface may be read from an ObjectInputStream.

There are similar decorations on the output side. An OutputStream can be used to create a Writer using OutputStreamWriter's constuctor: `new OutputStreamWriter( yourOutputStream )` On the output side, buffering also enhances efficiency, especially when writing to a file or network connection. The BufferedWriter also has a newLine() method. There are also Data and Object OutputStream classes. Only Serializable objects can be written to an ObjectOutputStream or ObjectWriter.

Finally, there are a pair of classes called PrintStream and PrintWriter.[Footnote: You should use PrintWriter, rather than PrintStream, if you want to create an instance of this kind of output stream. PrintStream exists only for compatibility with certain objects already built in to Java.] These output stream classes have the special advantage that none of their methods throws IOException. Their methods are called print and println, rather than write, to indicate their non-exception-throwing status. There are print and println methods for essentially every type of Java primitive. Using an Object's toString() method, print and println can also print any kind of Java Object. This makes these output stream types very useful for writing messages, e.g. to the Console.

There are several other decorator stream types defined in the Java.io package. Many of those are designed for special purposes. A few are documented in the Java IO Quick Reference appendix of this book.

Stream Sources

Now that you know how to manipulate streams, you may be wondering where you can find one. Streams come from a variety of different sources, depending on the resources that they connect.

For example, there are a series of streams that communicate with Files. These are called FileInputStream, FileOutputStream, FileReader, and FileWriter. Their constructors take the name of the file to read from or write to. These streams allow information to be read from or written to persistent storage, such as a disk. Since disk interactions are relatively slow, it is common to combine several disk access operations using the appropriate kind of buffered stream.

Another source of streams is pipes. A PipedInputStream (or OutputStream, Reader, or Writer) can be used to communicate between two Java objects. To do this, you must create a matched pair (PipedInputStream and PipedOutputStream or PipedReader and PipedWriter), then use the connect() method of one piped stream to join it to its mate.

We will look more closely at networked streams -- streams that communicate between two computers -- below. There are also streams that read from or write to arrays or Strings.

There are two additional streams with which you are already familiar, though you do not know it. These are the streams called the **standard input** and **standard output**. They are the streams that connect to the Java console. So far, you have used these through the cs101.io.Console class. In fact, there are two streams corresponding to the methods of Console.

Both `System.in` and `System.out` are static fields of the class `java.lang.System`. System.in is the standard input (or "stdin") stream, while System.out is the standard output ("stdout"). There a third stream, System.err, the standard error stream ("stderr"), that also writes to the Java console by default. System.err is intended for error messages, while System.out is intended to output to the user.

The type of System.in is simply InputStream. The type of System.out, however, is `java.io.PrintStream`. A PrintStream supports textual output of most Java primitive types as well as objects. It also avoids most of the otherwise-ubiquitous `IOException`s.

Decoration in Action

As we have seen, the four abstract IO classes lack some basic useful features and methods. Frequently, you would really rather be using one of their non-abstract subclasses. For example, one very common reader is `BufferedReader`. Instances of the `BufferedReader` class support useful methods such as

```
public int read() throws IOException;
public String readLine() throws IOException;
```

The first of these reads a single character from the input; the second reads an entire line of input. For a more complete list of `BufferedReader` methods, see the Java IO Quick Reference appendix.

To create a `BufferedReader`, you first need to have a `Reader`. Java doesn't come with any predefined `Reader`s, but it does come with a built-in `InputStream`: `System.in`, which reads from the Java console.The following (extremely useful) code assigns the name `myIn` to a `BufferedReader` that gets its input from `System.in`:

```
BufferedReader myIn = new
                        BufferedReader( new
                                        InputStreamReader( System.in ) );
try
{
   System.out.println( "I just read this line:  " + myIn.readLine() );
}
catch (IOException e)
{
    System.err.println( "Oops, I couldn't read a line!" );
}
```

In general, you can cascade the feature types, i.e., take any arbitrary stream and make another (more featureful) stream out of it. You begin with a particular stream, either based on an external (non-stream) structure or built in. Below, we will focus on streams created from a network interface.

Network Streams: An Example

In this final section, we will revisit the LectorScribe class that we defined above. Using what we have learned about actual Java streams, we will embellish that class so that it can be used to communicate with other computers running over the network. To do this, we will need to add the machinery of network communications: sockets. In Java, sockets and other network communication classes are implemented in a package called java.net. These classes are also covered in the Java IO Quick Reference appendix of this book.

Starting from Streams

The following code reproduces the LectorScribe class, above, with a few minor modifications. First, we have used actual java.io stream classes (in this case, Reader and Writer) as the stream types. We have also specified String as our *MessageType* and added some error messages when IOExceptions are caught.

```
public class LectorScribe implements Scribe, Animate
{
```

```
        private Writer out;
        private Reader in;
        private AnimatorThread mover;
        private LectorListener ll;

        public LectorScribe( Writer out, Reader in )
        {
            this.out = out;
            this.in = in;
            this.mover = new AnimatorThread( this );
            this.mover.start();
        }

        public void act()
        {
            try
            {
                this.ll.messageRead( this.in.read() );
            }
            catch (IOException e)
            {
                System.err.println( "Oops, I couldn't read a line!" );
            }
        }

        public void send( String m )
        {
            try
            {
                this.out.write( m );
                this.out.flush();    // (maybe)
            }
            catch (IOException e)
            {
                System.err.println( "Oops, I couldn't write a line!" );
            }
        }

        public void addLectorListener( LectorListener ll )
        {
            this.ll = ll;
        }

    }
```

Recall the logic of this code: A LectorScribe is responsible for reading from and writing to its streams. This involves continually monitoring the input with an active Thread (in the act() method) as well as being responsive to requests to write to the output (when send(String) is called from another Thread).

Decorating Streams

But what if you are given an InputStream and an OutputStream rather than a Reader and a Writer? In this case, we might add another constructor to this class, one which decorates these byte streams with their char equivalents. Only the additional constructor is reproduced here.

```
public LectorScribe( OutputStream out, InputStream in )
{
    this( new OutputStreamWriter( out ),
          new InputStreamReader( in ) );
}
```

Recall that a this() constructor invokes another constructor of the same class. Invoking new LectorScribe(System.in, System.out) results in the invocation of

```
new LectorScribe( new OutputStreamWriter( System.out ),
                  new InputStreamReader( System.in ) )
```

This would create a LectorScribe that writes on demand to the standard output stream and continually reads from the standard input stream.

Sockets and Ports

Where might you get input and output streams in the first place? The answer depends on what these streams are supposed to connect you to. For example, if you are reading from (or writing to) a file, you could use the FileInputStream/FileOutputStream or FileReader/FileWriter class pairs. In this section, we will explore streams that connect you to other computers. Java contains a standard library package called `java.net` that provides most of the infrastructure for making network connections.

Think back to the tin can telephone example. What we really want is a sort of a place on the other computer that we can connect to: someplace to "plug in" the tin can telephone. Computers have a number of such things, called **ports**, but you won't see them if you look at the back of a computer. Instead, a port is a virtual place to plug in a special kind of connection, called a socket**.**

A **socket** is an abstraction of actual network connections and protocols. It contains two streams: one for input, one for output. In other words, it is the virtual equivalent of a two-way pair of tin can telephones.

To establish a socket connection, you need to run a program at each end (i.e., one program on each of the two computers that the socket will connect).

- One of these programs "listens" for connection requests; this is called the server because it is providing the service of enabling socket connections. The server provides this service on a particular port of its machine. That is, the server needs to know which port to be watching to see whether anyone is trying to connect.
- The other program is called the client, and it contacts the server to open a socket connection. The client program needs to specify what machine to contact, typically using the name of that machine, and also what port on that machine to try to connect to.

Remember that the terms client and server are relative to a particular service. In this case, the server is providing the service of listening for socket connections, while the client is making use of that service. Once the socket is in place, though, it looks exactly the same from both ends.

Using A Socket

Using this idea of sockets, we can now read and write across the network. In Java, a socket is implemented by an instance of the class `java.net.Socket`. Suppose that we have one of these Java `Socket`s and want to read from and write to it using a LectorScribe.

We already know how to create a LectorScribe if we are given either a Reader and a Writer, or an InputStream and an OutputStream. A Java `Socket` has a method to access each of its streams: `getInputStream()`, which returns an `InputStream`, and `getOutputStream()`, which returns an `OutputStream`. If we had a socket -- one end of a virtual two-way tin can telephone -- we could access its input and output streams using these methods. We can accomplish this using yet another LectorScribe constructor:

```
public LectorScribe( Socket sock ) throws IOException
{
    this( sock.getOutputStream(), sock.getInputStream() );
}
```

In creating a LectorScribe for this `Socket`, we simply extract the streams and use them to create a LectorScribe on an output and an input stream. Using the remainder of the LectorScribe code above, we have a simple program that takes a `Socket` as an argument and transmits what it reads and writes over the `Socket` to the user via the Java console. Note, however, that this constructor risks throwing an IOException. This is because the `Socket` might be corrupt and the streams might not be accesible.

A final note on `Socket`s: Like a stream, a `Socket` has a close() method. You should make a point of closing your `Socket` when you're done with it.

Opening a Client-Side Socket

Now we have code to read and write from a `Socket`, we need to figure out where to get a `Socket` in the first place. As described above, we can get a `Socket` by connecting to a server -- a machine that is listening for connection requests -- on a particular port. We need to know what machine to connect to, specified by a String corresponding to its hostname, such as "www-cs101.ai.mit.edu". We also need to know on what port the server is listening for our connection. The port is specified by an integer. By convention, ports numbered below 1024 are reserved for "standard" protocols. Otherwise, you have fairly free choice of ports.

A `java.net.Socket` is created by calling its constructor with a `String` corresponding to the hostname of the machine you want to connect to and an `int` representing the port on that machine where something is listening for connections. So, if we had this information, we could use the following LectorScribe constructor:

```
public LectorScribe( String hostname, int port ) throws IOException
{
    this( new Socket( hostname, port ) );
}
```

[Footnote: Note that the constructor for `Socket()` may throw IOException.]

This would enable us to say, e.g.,

```
new LectorScribe( "www-cs101.ai.mit.edu", 8080 )
```

If we put this expression into our public static void main method, running this program would create a program that connects the user to the machine www-cs101.ai.mit.edu on port 8080. Anything the user types would be sent to that port on that machine, and anything that www-cs101.ai.mit.edu writes to port 8080 would be printed on the Java console. This is the complete program!

Opening a Single Server-Side Socket

Of course, to make the client side of this program work, something has to be listening on the appropriate port of the appropriate machine. What code should we run on www-cs101.ai.mit.edu to listen on port 8080?

The port listener code requires another class from the package java.net. This one is somewhat misleadingly named ServerSocket. A Java program uses a `java.net.ServerSocket` to listen for connections. To create a ServerSocket, you need to specify what port to listen on. Remember that this is the local port -- the port on the machine this code is running on -- and you are not making any connections, just waiting for someone else to contact you. (If someone throws you a pair of tin cans, you should catch them and use them to communicate.

The port number on which you listen is arbitrary, but it must match the port number on which the client will try to connect. (The client should also use the hostname of the computer on which this ServerSocket is running.) Remember that the port number should be at least 1024.

We will need to add two constructors to LectorScribe. The first simply creates the ServerSocket and invokes the LectorScribe constructor that takes a ServerSocket as an argument:

```
public LectorScribe( int port ) throws IOException
{
    this( new ServerSocket( port ) );
}
```

The action is really in this second constructor. This constructor says "listen on your port". The method

```
public void Socket ServerSocket.accept() throws IOException;
```

is a blocking method that returns a `Socket` when a connection has been made:

A ServerSocket's accept() method returns a Socket. Specifically, it waits until some program tries to connect to that port, then returns its own side of that connection.

```
public LectorScribe( ServerSocket serv ) throws IOException
{
    this( serv.accept() );
}
```

This complete LectorScribe is now ready to run on both sides of the network. By having one main program -- on a computer named *yourComputerName* -- run

```
new LectorScribe( 4321 )
```

and the other run

```
new LectorScribe( yourComputerName, 4321 )
```

you can create a simple two-way chat program. The number 4321 is, of course, an arbitrary choice, but both programs must use the same number.

The complete LectorScribe code is included in the code supplement (as LectorScribe.html).

A Multi-Connection Server

The accept() method, like an input stream's read() method, blocks until there is a connection ready to accept. So, like a read(), accept() -- and this method -- may wait for a very long time before returning. This means that it may be useful to have the accept() invocation run in its own Thread. We can write a variant on the LectorScribe by separating the connection listening from the rest of the program.

In fact, we may want to go further. A single application can have several connections active at once. There is no problem with having multiple connections running over the same port. A port is simply a place where a ServerSocket can be listening for connection requests. For these reasons, it is common to write a more sophisticated kind of server than a simple LectorScribe.

Essentially, the LectorScribe that we have seen so far is run on a Socket, not on a ServerSocket. An additional class is used solely to listen on the Socket. This class needs to have its own instruction follower, so it is an animate object. When it accepts a connection -- yielding a Socket -- it simply creates a LectorScribe on that Socket.

```
public class MultiServer implements Animate
{

    private ServerSocket serv;
    private AnimatorThread mover;

    public MultiServer( int port ) throws IOException
    {
        this.serv = new ServerSocket( port );
        this.mover = new AnimatorThread( this );
        this.mover.start();
    }

    public void act()
    {
        try
        {
            new LectorScribe( this.serv.accept() );
        }
        catch (IOException e)
```

```
            {
                System.err.println( "Failed to establish a connection!" );
            }
        }

    }
```

Server Bottlenecks

The server architecture that we have just described puts one computer in the middle of a network. This is sometimes called a **hub-and-spoke** architecture, since all connections run through the central server, or hub. There are advantages and disadvantages to this architecture. One of the major potential disadvantages is that the server can be overwhelmed if it receives more traffic than it can handle. In this case, the server has become a **bottleneck**, the difficult point where congestion must be relieved. The good news is that in a single-server model, upgrading the server is likely to significantly improve system performance.

Hub-and-spoke architecture is very common in networks. When increased reliability is needed, there are variant architectures that reduce the reliance on a single potential point of failure. The most extreme of these is one in which every computer connects with every other computer (on an as-needed basis). This amounts to a whole lot of LectorScribes talking with each other, without the added MultiServer code. This kind of architecture is called **peer-to-peer** communication, because neither of the participants is particularly more important. In that case, one plays the role of the server and the other the client only to establish the socket connection; after that, the two machines are equivalent.

A common variant on the hub and spoke, in which each server is in turn the client of a super-server (which may itself be a client...) makes for more efficient routing. This is called a **hierarchical** architecture. It is the basis of, for example, computer name lookup (also called **domain name service**) on the Internet.

Chapter Summary

- An InputStream is a Java abstraction describing an entity from which Things can be read; an OutputStream is an entity to which Things can be written.
- Streams can be used for I/O on the console, files and network connections, as well as certain Java objects like arrays and strings.
- Streams can have features like buffering, filtering, or automatic data formatting. These features can be cascaded using the appropriate stream class's constructor.
- Every Java instantiation has a PrintStream called System.out and an InputStream called System.in.
- ObjectInputStream and ObjectOutputStream are stream types that can be particularly useful for sending objects across the network.
- A Socket is one side of a network connection. It has an InputStream and an OutputStream. You can create a Socket by specifying the hostname and port to which you wish to connect.
- A ServerSocket is something that can accept connection requests on a particular port. You can create a ServerSocket, by specifying which port to listen on. A ServerSocket's accept() method returns a Socket object each time a new connection is made.

- A multithreaded server is an entity that creates a new self-animating object to handle each connection accepted by its ServerSocket.
- Such a server can be a hub for a network, but when it is overloaded, it can also be a communications bottleneck.

Exercises

Q. Write code to open a file and read it, one line at a time, printing each line to the standard output.

Q. Modify the LectorScribe so that it shuts down gracefully. That is, when a stream throws an exception -- e.g., when there is nothing more to read -- it should close its streams and its Socket.

Q. Modify the MultiServer so that it keeps track of the LectorScribes that it has created. Add something to the act() method of the MultiServer that sends a message over the output stream of each LectorScribe when a new connection is accept()ed. ("Congratulations on your new sibling!")

Q. Create a new kind of LectorListener event handler that notifies the MultiServer whenever one of its LectorScribes reads something from its input stream.

Q. Combine the answers to the previous two questions so that, when a message is read by one LectorScribe, it is broadcast to all of the LectorScribes' output streams. Bonus: Can you avoid sending the message to the initiating client?

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>