PIC 20A
Streams and I/O

David Hyde
UCLA Mathematics

Last edited: May 29, 2020

Why streams?

Often, you want to do I/O without paying attention to where you are reading from or writing to.

You can read from or write to a file, the console, a network connection, or a peripheral device. You must establish such I/O channels in their own specific ways.

Once a connection is established you have a *stream*, and you can polymorphically read from or write to a stream.

Introductory example

```
String s = "http://www.math.ucla.edu/~eryu/"
         + "courses/pic20a/lectures/RJ.txt";
URL url = new URL(s);
InputStream in = url.openStream();
OutputStream out = System.out;
while (true) {
  int nextByte = in.read();
  if (nextByte == -1)
    break;
  out.write((char) nextByte);
}
in.close();
out.close();
```

The read and write code doesn't know what kinds of streams it is using.

Outline

abstract class InputStream

An `Inputstream` reads (from somewhere) one byte at a time.

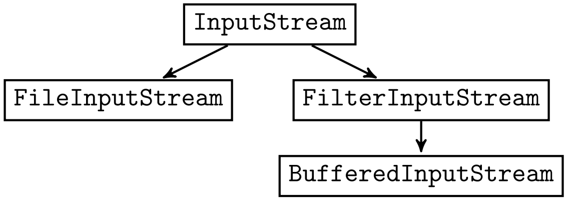
```
public abstract int read() throws IOException
```

reads and return a byte of data. It returns `-1` if the end of the stream is reached.

```
public void close() throws IOException
```

closes the stream and releases any system resources associated with the stream.

InputStream hierarchy



All classes are in the package `java.io`

abstract class OutputStream

An `Outputstream` writes (to somewhere) one byte at a time.

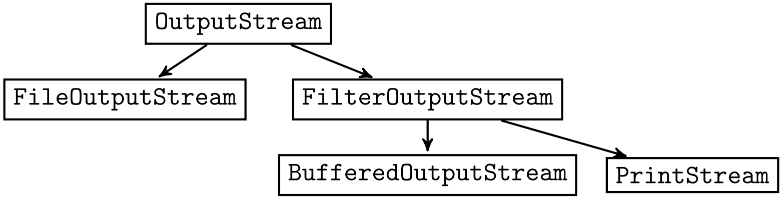
```
public abstract void write(int b)
                                throws IOException
```

writes a byte of data.

```
public void close() throws IOException
```

closes the stream and releases any system resources associated with the stream.

OutputStream hierarchy


$$\begin{tikzcd}
 & \texttt{OutputStream} \arrow[dl] \arrow[dr] & & \\
\texttt{FileOutputStream} & & \texttt{FilterOutputStream} \arrow[d] \arrow[dr] & \\
 & & \texttt{BufferedOutputStream} & \texttt{PrintStream}
\end{tikzcd}$$

All classes are in the package `java.io`

FileInputStream and FileOutputStream

`FileInputStream` reads bytes from a file and `FileOutputStream` writes bytes to a file.

```
InputStream in;
OutputStream out;
in = new FileInputStream("filename1");
out = new FileOutputStream("filename2");
while (true) {
  int nextByte = in.read();
  if (nextByte == -1)
    break;
  out.write((char) nextByte);
}
in.close();
out.close();
```

PrintStream

In addition to the methods inherited, `PrintStream` has additional methods like

```
public void println(int x)
public void println(double x)
public void println(Object x)
...
```

`PrintStream` never `throws` an `IOException`. Rather, you can use

```
public boolean checkError()
```

System I/O

Most operating systems (Windows, MacOS, and Linux) have 3 *standard streams*: standard input, standard output, and standard error. You interact with them through the command line.

Use the fields of `class System`

```
public static InputStream in
public static PrintStream out
public static PrintStream err
```

to access these standard streams.

System I/O

```
InputStream in;
OutputStream out;
in = System.in;
out = System.out;
while (true) {
  int nextByte = in.read();
  if (nextByte == -1)
    break;
  out.write((char) nextByte);
  //out.print((char) nextByte);
}
in.close();
out.close();
```

Use `CTRL+D` or `CTRL+Z`, depending on your system, to send the `EOF` (end of file) “character” to the standard input stream.

Outline

Always close streams

Streams often handle are scarce, finite resources. You can run out of them if you don't clean them up properly.

Always close them and close them within a `finally` block.

When you don't close streams, they are usually closed when your program terminates. For small programs, this is usually okay.

However, first we make our habits, then they make us. There's a reason experienced programmers insist that you close streams and that you do so with a `finally` block.

Always close streams

Here's how you properly close the streams.

```java
InputStream in = null;
OutputStream out = null;
try {
  in = ???;
  out = ???;
  ...
} catch(...) {
  ...
} finally {
  ...
```

Always close streams

```
  ...
  if (in!=null) {
    try {
      in.close();
    } catch (IOException ex) {
      System.out.println("Stream close failed");
    }
  }
  if (out!=null) {
    try {
      out.close();
    } catch (IOException ex) {
      System.out.println("Stream close failed");
    }
  }
}
```

If you think this is too much work, I understand.

try-finally

When you duck the exception handling, you can use `try-finally` to ensure the streams are `closed`.

```
public void fn() throws Exception {
  InputStream in = null;
  try {
    in = new FileInputStream("filename.txt");
    //do stuff with in
    ...
  } finally {
    in.close();
  }
}
```

The `try-finally` construct does not catch any exceptions, but is nevertheless useful.

Outline

Buffered streams

The streams we've talked about perform *unbuffered* I/O. This means the bytes are read or written one byte at a time. This can be inefficient if there is a fixed cost associated with each read/write.

Buffered input streams read in bulk, and stores the data in a *buffer*. Reading each byte retrieves the bytes from the buffer.

Buffered output stream stores the individual writes into a buffer. When the buffer is full, the entire buffer is written in bulk.

Buffered streams

In Java, `BufferedInputStream` is an `InputStream`, and it's created (via the constructor) with an `InputStream`.

The same is true with `BufferedOuputStream` and `OutputStream`.

`BufferedInputStream` and `BufferedOuputStream` are *wrappers*. A wrappers is something that uses another functionality and provides improved efficiency or convenience.

Buffered streams

In this example, buffered streams provide a huge speed-up.

```
InputStream in = null;
OutputStream out = null;
try {
  in = new FileInputStream("filename1");
  in = new BufferedInputStream(in);
  out = new FileOutputStream("filename2");
  out = new BufferedOutputStream(out);
  while (true) {
    int nextByte = in.read();
    if (nextByte == -1)
      break;
    out.write((char) nextByte);
  }
} finally {
  in.close(); out.close();
}
```

Outline

Avoid using byte streams

All streams are implicitly or explicitly built with byte streams, so it's important to understand them.

However, you should only use byte streams when you actually want to input/output bytes.

If you want to input/output characters, use a character stream.
It's more convenient and less error-prone.

To input/output primitive types and `Objects`, use data streams and object streams, respectively. (We won't talk about this.)

Readers and Writers

In Java, character streams are `java.io.Readers` and `java.io.Writers`.

`Readers` and `Writer` are built upon `InputStreams` and `OutputStreams`.

```
InputStream is = new FileInputStream("name1");
OutputStream os = new FileOutputStream("name2");
Reader r = new InputStreamReader(is);
Writer w = new OutputStreamWriter(os);
//Same as
//Reader r = new FileReader("name1");
//Writer w = new FileWriter("name2");
```

BufferedReaders and BufferedWriters

For efficiency sake, you want to use BufferedReaders and BufferedWriters.

```
Reader r;
Writer w;
...
BufferedReader br = new BufferedReader(r);
BufferedWriter bw = new BufferedWriter(w);
```

With BufferedReader, you can read an entire line.

```
String line = br.readLine();
```

Character encoding and internationalization

The simplest way to encode English characters is via ASCII, which uses 8 bits per character.

Java, however, supports 16-bit unicode characters (which is fortunately backwards-compatible with ASCII).

To represent characters from languages like Arabic, Chinese, Greek, Hebrew, Hindi, Japanese, Korean, or Russian, you need to use unicode. We won't talk about unicode because it's a mess.

Internationalization is adapting a program to different languages, regional differences and technical requirements of a target locale.

Just remember that Java's characters streams have great support for internationalization.

Outline

Scanning and formatting

I/O often involves translating between computer-readable and human-readable representations of data.

`Scanner` translates human-readable strings into computer-readable data.

`format` translates computer-readable data into human-readable strings.

Scanner

java.util.Scanner can produce ints, doubles, etc.
from a String or an InputStream.

```
Scanner s;
s = new Scanner("FFF");
int i = s.nextInt(16); //read as base 16
s = new Scanner(System.in);
double d = s.nextDouble(); //from command line
```

Scanner offers many more useful features.

format

`PrintStream` and `PrintWriter` have the method `format`.

```java
int i = 2;
double r = Math.sqrt(i);
PrintStream o = System.out;
o.println("sqrt of "+i+" is "+r+".");
o.format("sqrt of %d is %.2f.%n",i,r);
```

The output is

```
sqrt of 2 is 1.4142135623730951.
sqrt of 2 is 1.41.
```

Find instructions on how to use `format` at
`https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html`