COMP2221 Networks

David Head

University of Leeds

Lecture 6

Reminder of the last lecture

After covering ports, IP addresses and the DNS, last time we started on some simple Java coding:

- Saw the differences between IPv4 and IPv6.
- How to create instances of the `InetAddress` class.
- How it accesses a DNS server upon construction.
- Instances are **immutable**; there are no public 'set' methods.

Before moving onto today's new material, will first see an example usage of `InetAddress`.

Example usage of InetAddress: nslookup

nslookup is a Unix command line tool to query DNS interactively.

- Early tool for testing the network.

Can convert in two directions:

- Hostname to IP address; or
- IP address to hostname.

Can implement a simple Java version using the standard InetAddress methods.

```
1  import java.net.*;
2
3  public class Lookup {
4    private InetAddress inet = null;
5    public void resolve(String host) {
6      try {
7        inet = InetAddress.getByName(host);
8        System.out.println("Host name :"
9                                          +inet.getHostName());
10       System.out.println("IP Address:"
11                                         +inet.getHostAddress());
12     }
13     catch( UnknownHostException e ) {
14       e.printStackTrace();
15     }
16   }
17
18   public static void main(String[] args) {
19     Lookup lookup = new Lookup();
20     lookup.resolve(args[0]);
21   }
22 }
```

Notes

Takes a single string as a command line argument, which can be:

- A web server URL, *e.g.* `www.comp.leeds.ac.uk`, or any other one you can think of such as `www.google.com`
- If you are in the lab, the name of your machine, which you can find by calling `hostname` from a shell.
- Can work backwards from an IP address you have just found, *i.e.* `129.11.144.10`

A commented version of this code is on Minerva alongside the slides for this lecture.

Today's lecture

Any useful networking application will want to send bytes of data between hosts.

In Java, this is performed using **I/O streams**.

- Implemented in the `java.io` package.

This is for TCP-based IO. We will also later see some alternatives:

- UDP-based **datagram packets** *(Lecture 14)*.
- TCP- or UDP- based `java.nio` channels for **non-blocking** communication (can improve performance) *(Lecture 12)*.

Streams

Streams allow us to **abstract** the I/O process away from the source/destination[1].

- *i.e.* file I/O is the same as network I/O is the same as terminal I/O ...
- Abstraction is one of the design principles of OO languages, including Java.

Streams are **unidirectional**:

- **Input streams** read data, **output streams** write data.
- Two-way communication requires two streams.

[1] C++ also has streams of sorts, implemented *via* the << and >> operators.

The java.io package

Fundamentally there are two **abstract** Java classes to handle I/O.

Abstract classes are 'incomplete' - **they cannot be instantiated**.

- Has at least one method that must be **overridden** in a subclass.
- That subclass is then a **concrete** class that **can** be instantiated[1].

```
public abstract class OutputStream
public abstract class InputStream
```

They provide minimal I/O functionality that subclasses can exploit.

[1] In C++, abstract methods are declared '=0.' Also possible in Python 3 using the abc module ('abstract base class').

Output streams

`public abstract class OutputStream` provides these public methods:

`public abstract void write( int b )`

- Given an integer, writes one byte to the output stream.
- **Abstract**; **must** be overridden.
- Implementation depends on medium.

`public void write( byte[] bytes )`

- Sends **an array** of bytes to the output stream.
- **Not abstract**.
- By default, will call `write(b)` for each element `b` of `bytes`.
- Can also be **overridden** with a more efficient implementation.

`public void flush()`

- Send any data that may be residing in a **buffer**.
- Buffers useful for performance — *i.e.* better to send a few large blocks that many small blocks.
- Does **not** mean that the data is **immediately** sent; only that it is passed to the operating system for output.

`public void close()`

- Releases resources used by this stream.

Input streams

`public abstract class InputStream` provides these public methods:

`public abstract int read()`

- Reads a single byte from the stream.
- **Abstract**: Depends on medium and must be overridden.
- `public int read(byte[] bytes)` reads up to `bytes.length` bytes and stores in the array `bytes`..
  - Returns the number of bytes actually read.

`public long skip( long n )`

- Skips over **and discards** `n` bytes of data from the stream.
- Returns actual number skipped (*e.g.* if end of file reached).

`public void close()`

Efficiency

Recall that TCP sends data as **packets** with header information.

- TCP header is usually 20 bytes.
- Additional headers at network and link layers.
- Total header at least 40 bytes, but can be much larger.

Sending **one byte** of data per packet is **very inefficient**.

Buffers are used to manage the communication of reasonably-sized packets.

- Only send when the buffer is 'sensibly' bigger than the header.
- Will always send to OS when `flush()` is called.

Buffered output

```
public BufferedOutputStream( OutputStream out )
```

- 'Wrapper' around an instance of a concrete object derived from `OutputStream` *(polymorphism)*.
- Only adds a buffer.
- Defaults to a 512 byte buffer, but can be changed.
- Inherits basic `OutputStream` methods.

Buffered input

```
public BufferedInputStream( InputStream in )
```

- Buffer defaults to 2048 bytes, but can be changed.
- Optimum buffer size depends on the application.

Larger than the write buffer:

- Don't want to write more than we can read.
- Have more control over output [`write()`].
- `read()` is 'when reading' and may come in bursts.

`read()`/`write()` correspondence?

There is **none**.

That is, do **not** assume that a single `write()` is matched by a single `read()`:

- One write may correspond to multiple reads.
- Conversely, one read may correspond to multiple writes.

We have no direct access to the buffer.

- Our only control is to call `flush()`, which sends all buffered data to the operating system (and then the network, file *etc.*).

Blocking

`read()` is designed as a **blocking** call.

- Our application will **wait** until data is received.

This has implications for TCP-based application design.

- Our application will 'freeze' once `read()` is called.
- **Deadlock** can result if both ends of the connection are in blocking states.
  - *i.e.* Each application waiting for data from the other, but neither can reach their `write()` methods.
  - Can resolve by the sending process `flush`ing its buffer.

Also has implications for efficiency.

- We could be doing useful work while waiting for data to arrive.

Filters

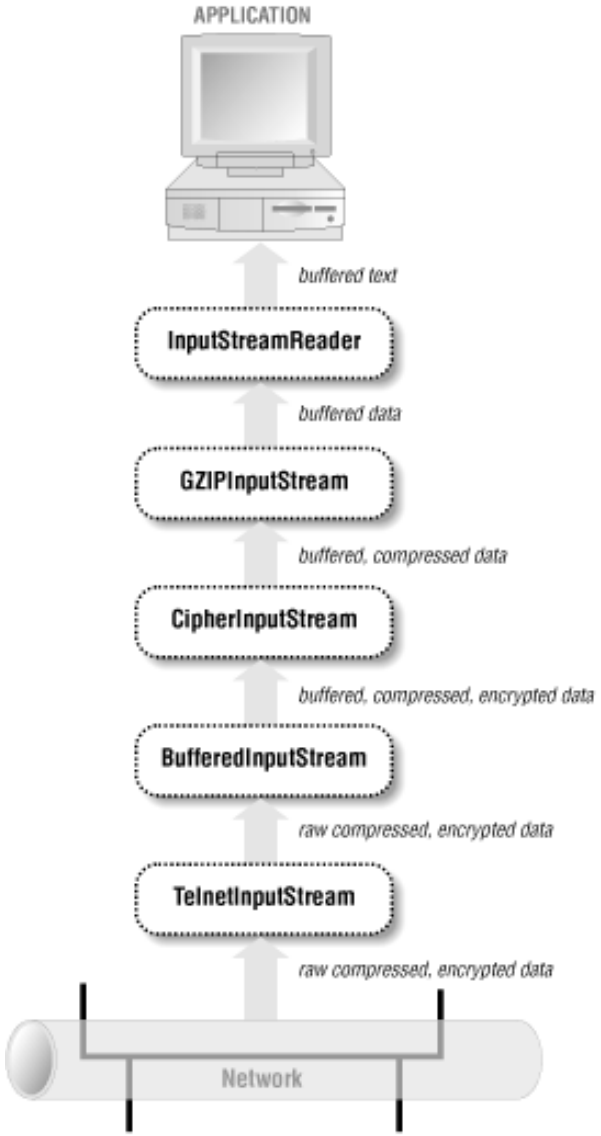
The basic `InputStream`/`OutputStream` classes just read and write bytes.

Filters are provided to handle standard data formats (*i.e.* text files, compression, encryption).

There are two types:

- **Filter streams**: Sits above raw I/O to provide data in a given format (can't sit above reader/writer).
- **Readers** and **writers**: Sits above raw, filtered or other read/write streams; typically for text encodings.

The result is a layered programming model known as **chaining**.



From Harold, *Java Network Programming* (O'Reilly).

Chaining example

Every link in the chain:

- Receives data from the previous filter or stream;
- Passes data to the next filter or stream.

In the example on the previous slide:

- `TelnetInputStream` receives a compressed, encrypted file from the network;
- `BufferedInputStream` adds a buffer for efficiency;
- `CipherInputStream` decrypts the data;
- `GZIPInputStream` decompresses the data;
- `InputStreamReader` converts to Unicode text.

Chaining in Java: Method 1

Instantiate each class separately:

```
1 FileOutputStream fileOut = new
2                FileOutputStream("dat.txt");
3 BufferedOutputStream buffOut = new
4                BufferedOutputStream(fileOut);
5 DataOutputStream dataOut = new
6                DataOutputStream(buffOut);
```

- We could replace any of the filters (abstraction).
- We only use `dataOut`.
- `fileOut` and `buffOut` are chained filter streams that we never directly use.

Chaining in Java: Method 2

Explicitly chained (usually preferred):

```
1 DataOutputStream dOut = new DataOutputStream(
2                          new BufferedOutputStream(
3                           new FileOutputStream("dat.txt")));
```

- Only the usable stream `dOut`, and the source `dat.txt`, is exposed.
- Prevents accidentally using the wrong one.
- The others are **anonymous**, with implied instantiation.
- The connection between filters is **permanent**.

Abstraction

Read from a file:

```
1 BufferedReader fIn = new BufferedReader(
2                       new FileReader("dat.txt"));
```

Read from keyboard:

```
1 BufferedReader kIn = new BufferedReader(
2                       new InputStreamReader(System.in)
3                      );
```

Read from a socket *(next lecture)*:

```
1 BufferedReader sIn = new BufferedReader(
2                       new InputStreamReader(
3                        socket.getInputStream()));
```

In each case our application receives an **identical** buffered stream of data.

Warning: Changing streams mid-flow

When using streams, it is *not* advisable to alter the chaining **during** communication.

- Results in **undefined behaviour**.
- *i.e.* results may differ each time — may work sometimes but not others.
- Flushing between changes does **not** solve this.

Instead, select one chain and stick to it until **all** communication is finished.

- This observation can save much hair-pulling when developing communication applications, *e.g.* clients and servers.

Summary and next time

Today we have seen how to use Java I/O streams:

- Abstract classes for reading and writing.
- **Buffering** for performance.
- **Chaining** of multiple filters.
- Object oriented approach, *i.e.* abstraction.

More information in Chapter 2 of Harold, *Java Network Programming* 4^{th} ed. (2014).

Next time we will look at the Java `Socket` class, and write a TCP-based client side application.