SE 550: Lecture 2

Overview

Homework review

I/O in Java

OutputStream

InputStream

Filter streams

Readers and writers

Interruptable I/O

Homework review

A comment on coding style

Some rules I've used for coding:

- No public classes, only public interfaces! (Except where Java forces them.)
- Declare all variables and fields `final` unless absolutely necessary.
- Use lots of debugging println statements.
- Javadoc comments for all public fields and methods.

What are the pros and cons of these coding rules?

I/O in Java

Java has four basic classes for I/O:

- OutputSteam: sends binary data
- InputSteam: receives binary data
- Writer: sends text data
- Reader: receives text data

Everything else (sockets, files, System.out etc. are subclasses of these).

Why two of everything (binary vs text)?

OutputStream

The methods of an output stream are:

```
void write (int b) throws IOException
void write (byte[] data) throws IOException
void write (byte[], int offset, int length) throws IOException
void flush () throws IOException
void close () throws IOException
```

What do these methods do?

OutputStream

What does this code try to do? What is wrong with it? How can we fix it?

```
void printHelloWorld (...) {
    final OutputStream out = ...;
    for (int i=0; i < msg.length (); i++) {
        out.write (msg.charAt (i));
    }
    out.close ();
}
```

The fixed code is available.

OutputStream

Where do output streams come from?

- System.out and System.err.
- FileOutputStream and Socket.getOutputStream.

Where do these streams write to?

Most other output streams are built on top of these basic ones.

InputStream

The important methods of an input stream are:

```
int read () throws IOException
int read (byte[] input) throws IOException
int read (byte[] intput, int offset, int length) throws IOException
int available () throws IOException
void close () throws IOException
```

What do these methods do?

What happens at the end of file? Why does `read()` return a `int` rather than a `byte`?

InputStream

What does this code try to do? What is wrong with it? How can we fix it?

```
byte[] get128 (final InputStream in) {
    final byte[] result = new byte[128];
    for (int bytesRead = 0; bytesRead < 128;) {
        final int read = in.read (result, bytesRead, 128-bytesRead);
        bytesRead = bytesRead + read;
    }
    return result;
}
```

The fixed code is available.

InputStream

Where do input streams come from?

- System.in.
- FileInputStream and Socket.getInputStream.

Where do these streams read from?

Most other input streams are built on top of these basic ones.

Stream example

Write a Pipe class which connects an input and an output stream together:

```
class PipeStream {

    protected final InputStream in;
    protected final OutputStream out;

    protected PipeStream (final InputStream in, final OutputStream out) {
        this.in = in;
        this.out = out;
    }

    public void connect () throws IOException  {
        ...keep reading from in and writing the result to
        out until in reaches the end of stream...
    }

}
```

Here's the completed code.

Stream hints

Remember to flush output buffers.

Remember IOExceptions! (Expect the majority of your code to be exception handling.)

Remember that exception handlers can raise exceptions!

Use finally to make sure clean-up is always executed.

Use close to close any streams you open.

Filter streams

Raw InputStreams and OutputStreams aren't much use!

Often we want to add functionality to an existing stream.

For example:

- A BufferedInputStream which adds buffering to an input stream.
- A CipherInputStream which decrypts an encrypted input stream.
- A GZIPInputStream which uncompresses a compressed input stream.

Streams which read from other streams are called *filter streams* in Java.

Filter streams

Filter streams can be chained together:

```
final FileInputStream rawIn =
    new FileInputStream ("foo.gz.des");
final InputStream processedIn =
    new GZIPInputStream (
        new CipherInputStream (
            new BufferedInputStream (
                rawIn
            ),
            desCipher
        )
    );
```

This is an example of *dataflow programming*.

Readers and writers

Input and OutputStreams handle raw data.

Readers and Writers handle text.

Most protocols (e.g. HTTP) and data exchange formats (e.g. XML) these days are text based.

(Why? Isn't it much more efficient to send binary data?)

For example:

```
final OutputStream out = ...;
// Create a new text writer with encoding UTF-8
final Writer writer = new OuputStreamWriter (out, "UTF-8");
```

Common values for the encoding: ISO-8859-1, UTF-8 or UTF-16. (What are these?)

Readers and writers

Two very useful methods:

- String BufferedReader.readLine ()
- PrintWriter.println (String msg)

for example from a Socket s:

```
PrintWriter writer = new PrintWriter
    (new OutputStreamWriter (s.getOutputStream (), "UTF-8"));
writer.println ("GET /index.html HTTP/1.0");
writer.flush ();
```

and:

```
BufferedReader reader = new BufferedReader
    (new InputStreamReader (s.getInputStream (), "UTF-8"));
String request = reader.readLine ();
```

Connect these up and what happens?

Readers and writers

Unfortunately, there's a number of gotchas with Sun's code:

- PrintWriter doesn't throw any IOExceptions, you have to do your error-handling the old-fashioned C way (blech).
- Mixing text and binary is very hard!
- BufferedReader and PrintWriter can't get their act together about what a new line is!

For example, on a Mac sending on a socket:

```
writer.println ("A message");
```

and on any Java machine recieving on a socket:

```
final String line = reader.readLine ();
```

this will deadlock! Why?

Harold provides fixes: SafeBufferedReader and SafePrintWriter.

Interruptable I/O

An even worse bug with Java I/O...

What happens when one thread interrupts a thread doing I/O?

Here's an example.

Here's Sun's documentation.

Interruptable I/O

Oh dear, Sun have silently deprecated interruptable I/O!

See bugs 4103109 and 4154974 at Sun's Sun's Bug Parade.

Solution: new I/O package provides:

1. Non-blocking I/O.

2. Support for charsets.

3. Selectable I/O.

What are these?

Streams summary

Streams programming is the backbone of distributed programming.

Java's I/O library supports sophisticated dataflow programming techniques in a modular, scaleable fashion.

There are some bugs with Java I/O!

Next week: protocol specification.

Homework: Sheet 2, due before next lecture.

Reading:

Harold Chapter 2, section on *Internet Standards*.

Johnsonbaugh *Discrete Mathematics*, Section 10.3 *Languages and Grammars*.