COMP241

Software Engineering Development

Lecture 10: Java I/O 1

Mark Hall

- Overview
- IO Zoo
- Stream I/O
- File I/O
- Beyond Bytes
  - Decorator design pattern
- Buffering
- Example
- File Class
- Object Serialization

DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Overview

- IO provides communication with devices (files, console, networks etc.)
- Communication varies (sequential, random-access, binary, char, lines, words, objects, …)
- Java provides a "mix and match" solution based around byte-oriented and character-oriented I/O streams – ordered sequences of data (bytes or chars).
- System streams `System.in`, (`out` and `err`) are available to all Java programs (console I/O) – `System.in` is an instance of the InputStream class, `System.out` is an instance of PrintStream
- So I/O involves creating appropriate stream objects for your task.

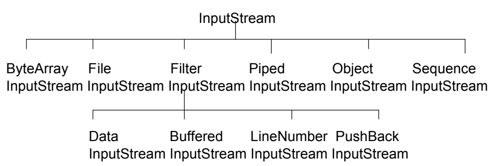
The IO Zoo

- More than 60 different stream types.
- Based around four abstract classes: `InputStream`, `OutputStream`, `Reader` and `Writer`.
  – Streams read and write 8 bit values
    - Input streams can be divided intot those that read from physical input sources (eg.file) and those that add functionality to another input stream
  – Readers and Writers read and write 16 bit *Unicode* characters

Reading Bytes

- Abstract classes provide basic common operations which are used as the foundation for more concrete classes, eg `InputStream` has
  – `int read()` — reads a byte and returns it or –1 (end of input)
  – `long skip(long n)` — skip over and discard the next n bytes
  – `int available()` — num of bytes still to read
  – `void close()`
- Concrete classes override these methods
  – eg `FileInputStream` reads one byte from a file, `System.in` is a subclass of `InputStream` that allows you to read from the keyboard

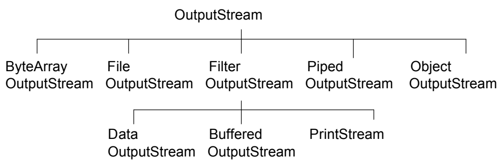
InputStream hierarchy



Writing Bytes

- Abstract class `OutputStream` provides basic common operations for output
  – `void write(int b)` — writes a single byte (least significant byte of an integer) to an output location.
  – `void write(byte[] b)` — writes an array of bytes to an output location
  – `flush()` — force any buffered output to be written
- Java IO programs involve using concrete versions of OutputStream because most data contain numbers, strings and objects rather than individual bytes

OutputStream hierarchy



File Processing

- Typical pattern for file processing is:
- OPEN A FILE
- CHECK FILE OPENED
- READ/WRITE FROM/TO FILE
- CLOSE FILE
- Input and Output streams have `close()` method (output may also use `flush()`)

File/Stream Processing (reading bytes)

- Use InputStream's `read()` method to read a single byte
  - `Read()` returns an `int`, namely either the byte that was input (0-255) or the integer -1 (indicates the end of the input stream)
  - Should test the return value and, if it is not -1 cast it to a `byte`

File/Stream Processing (reading bytes)

```
InputStream myIn = new FileInputStream("input.bin");
boolean done = false;

while (!done) {
  int next = myIn.read();
  if (next == -1) {
    done = true;
  } else {
    byte b = (byte)next;
    // process input…
  }
}
myIn.close();
```


Common Error

- Negative `byte` values
  - In Java, the `byte` type is a *signed* type — 256 values from -128 to 127
  - The first bit of the byte is the *sign bit*
  - When converting an integer to a byte, only the least significant byte of the integer is taken
    - The result can be **negative** even if the integer is positive

```
int n = 233; //binary 00000000 00000000 00000000 11101001
byte b = (byte)n; //binary 11101001, sign bit is set
if (b == n)… //not true! b is negative, n is positive
```


Moving beyond bytes

- `FileInputStream` and `FileOutputStream` give you IO from a disk file:

```
FileInputStream myInFile =
    new FileInputStream("in.txt");
```

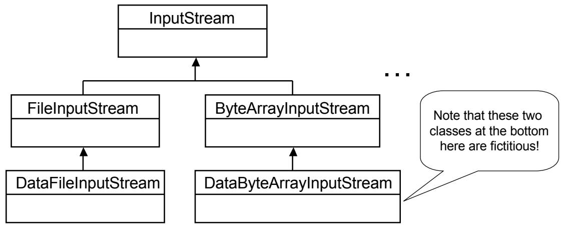
- We can now read bytes from a file but not much else!
  - Java's IO package is built on the principal that each class should have a very focussed responsibility (*cohesion*)
  - `FileInputStream` interacts with files — its job is to *get* bytes, not to analyse them
- To read numbers, strings, objects etc., you have to combine `FileInputStream` with other classes whose responsibility is to group bytes or characters together

Moving beyond bytes

- To get a file stream that can process data means making use of a *virtual* input stream
  - Don't actually directly access a physical input source (eg. file)
  - Instead, they add functionality to an underlying input stream
- `FilterInputStream` is the superclass of a number of virtual input streams that add various functionality
  - Demonstrates the combination of OO mechanisms *inheritance* and *composition*
  - Is an example of the *Decorator* (Filter or Wrapper) design *pattern*
    - See discussion of Decorator in Section 5.6 of Horstmann

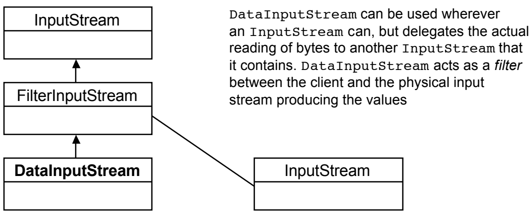
Moving beyond bytes

- Say we wanted to add the ability to read data (floats, ints, reals etc) to InputStream?
  - Could use inheritance, but we end up duplicating functionality



Moving beyond bytes

The *Decorator* design pattern is a recipe that we can follow to allow additional behaviour or responsibilities to be added to an object dynamically



Moving beyond bytes

```
FileInputStream fin =
  new FileInputStream("in.bin");
DataInputStream din =
  new DataInputStream(fin);
  – double s = din.readDouble();
  – boolean b = din.readBoolean();
  – int i = din.readInt();
  – etc…
```

- Note that these methods read *multiple* bytes from the underlying stream and return them as a primitive type
- Much nicer interface to a file!

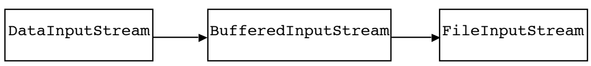
Buffering

- By default streams are not buffered, so every read or write results in a call to the OS (= very slow!).
- Buffering can be added (to any input stream) by using the BufferedInputStream
  - Another example of a **`FilteredInputStream`**
  - Values are read from the underlying input stream in large blocks
  - Calls to `read()` return bytes from BufferedInputStream's internal buffer
  - `mark()` can be used to mark a location in the internal buffer; `reset()` can be used to reset the input back to the marked location, allowing values to be read again

Buffering

```
DataInputStream din =
  new DataInputStream(new
    BufferedInputStream(new
      FileInputStream("in.txt")));
```

- DataInputStream is *last* in the chain here because we want to use its methods and we want them to use the buffered methods (eg `read()`).



Example: An Encryption Program

- Read a file and write out another file that is a scrambled copy of the first
- The Caesar Cipher (substitution cipher)
  – Simple method that uses an *encryption key* (number) that indicates the shift to be used in encrypting each byte
  – If our bytes hold characters and we use a key of 3:

M	e	e	t		m	e		a	t		t	h	e	
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
P	h	h	w	#	p	h	#	d	w	#	w	k	h	#

  – Can reverse the process (decrypt) by applying the negative key value

Encryptor.java: Encrypting Binary Data

```
public void encryptStream(InputStream in, OutputStream out)
  throws IOException {
  boolean done = false;
  while (!done) {
    int next = in.read();
    if (next == -1) {
      done = true;
    } else {
      byte b = (byte)next;
      byte c = encrypt(b); // call method to encrypt byte
      out.write(c);
    }
  }
}
```


Encryptor.java: The encrypt() method

```
/**
    Encrypts a byte.
    @param b the byte to encrypt
    @return the encrypted byte
 */
public byte encrypt(byte b) {
  return (byte)(b + mKey);
}
```

Note that mKey is an integer, so we have to cast back to a byte after the addition

Encryptor.java: Setting up the input and output streams

```
public void encryptFile(File inFile, File outFile)
  throws IOException {
  InputStream in = null; OutputStream out = null;
  try {
    in = new BufferedInputStream(
       new FileInputStream(inFile));
    out = new BufferedOutputStream(
       new FileOutputStream(outFile));
    encryptStream(in, out); //process the data
  } finally {
    if (in != null) {
      in.close();
    }
    if (out != null) {
      out.flush(); out.close();
    }
  }
}
```

The File Class

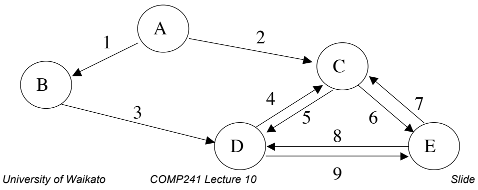
- Encryptor.java sets up input streams using *File* objects rather than file names as strings
- `File` class describes disk files and directories
  – Uses *abstract* pathnames — conversion to and from abstract pathnames is system dependent
  – Some methods:
    - static char `pathSeparator` — system dependent path separator character
    - `boolean exists()`
    - `boolean canRead()` — returns true if file exists and application can read it (ie. depends on security restrictions)
    - `boolean isFile()` — returns false if the `File` object corresponds to a directory
    - `boolean delete()`
    - `boolean mkdir()` — create a directory named by the pathname
  – `File myFile = new File("input.dat");`

Object Serialization

- So far we've seen sequential reading/writing of binary data
- In Java there is an even easier way to write sequential data — object *serialization*
  – Entire objects can be written to disk in **binary** form with almost no extra work on the part of the programmer
- *Serialization* — is the ability to save the state of an object (or several objects) to a **stream**
  – The stream is typically associated with a file, but need not be (eg sending serialized objects over a network connection)
- *Deserialization* — is the ability to restore the state of an object (or several objects) from a stream

Object Serialization

- If an object contains references to other objects, these are also saved
  - The process is automatic and recursive
  - Ensures that only a single copy of each referenced object is saved to the stream



Object Serialization

- To save object data we need to use the **`ObjectOutputStream`** class

```
Scores myScore = new Scores("Chris Harris",135,"India");
ObjectOutputStream os =
   new ObjectOutputStream(new FileOutputStream("scores.dat"));
os.writeObject(myScore);
```

- The object output stream automatically saves all instance variables of the object

Object Serialization

- To read the object back in, use the **`readObject`** method of the **`ObjectInputStream`** class

```
ObjectInputStream is =
   new ObjectInputStream(new FileInputStream("scores.dat"));
Scores myScore = (Scores)is.readObject();
```

- `readObject` returns an `Object` reference, so we need to cast to the appropriate type
- `readObject` can throw a **`ClassNotFoundException`** as well as the normal `IOException`
  - ClassNotFoundException gets thrown if the virtual machine cannot find the class of the read object in the classpath

Object Serialization

- Now if we want to save a collection of Scores all we have to do is write out the collection object

```
ArrayList<Scores> myScoresList = new ArrayList<Scores>();
// add a whole bunch of scores into the ArrayList
os.writeObject(myScoresList);
```


Object Serialization

- To place objects of a particular class into an object stream, the class must implement the **`java.io.Serializable`** interface
  - Is an indicator interface (ie. has no methods)
  - A java.io.NotSerializableException is thrown if a class does not implement Serializable

```
Public class Scores implements Serializable {
   protected String mName;
   protected int mScore;
   protected String mCountry;

   public Scores(String name, int score, String country) {
      mName = name; mScore = score; mCountry = country;
   }
}
```

Object Serialization

- Only *nonstatic* and *nontransient* parts of an object's state are saved by serialization
  - **Static** fields are considered part of the state of the **class**, not the state of an object
  - **Transient** fields are not saved, since they contain temporary data not needed to correctly restore the object later

Object Serialization

- Many of the classes provided with the JDK libraries have been designed to be serializable
- However, there are some that are not serializable
  - Almost none of the classes in **java.io** are serializable
    - Ridiculous to consider “freezing” info about file handles, read/ write positions etc and expect to use it later - even on the same machine
  - Objects of type **Thread** are not serializable
    - Implementation of threads is tightly coupled with the particular platform on which the JVM (java virtual machine) is running