Java I/O Streams

Michael B. Spring
Department of Information Science and Telecommunications
University of Pittsburgh
spring@imap.pitt.edu
http://www.sis.pitt.edu/~spring
Overview of Part 1 of the Course

• Demystifying Java: Simple Code
• Introduction to Java
• An Example of OOP in practice
• Object Oriented Programming Concepts
• OOP Concepts -- Advanced
• Hints and for Java
• I/O (Streams) in Java
• Graphical User Interface Coding in Java
• Exceptions and Exception handling
Objectives for this unit

- Define what streams are
- Examine the different types of streams
- Demonstrate how to work with streams
- Illustrate how streams can be layered
Concepts

- Java adopts a Unix-like view of input and output that treats all I/O as streams of bytes. The semantics of the bytes are not addressed in the conceptualization of a stream.
- In Unix, functionality and semantics are managed by the user or by libraries of functions. In Java, these are provided by classes.
- Just as in Unix, “streams” may be piped through sequences of tools to provide compound functionality.
- With the exception of the RandomAccessFile class, all of I/O in Java is unidirectional.
The Implementation as Classes

- There are several classes which play a role in I/O in Java.
  - Files are objects of Class File
  - Four abstract classes organize the various I/O streams
    - InputStream
    - OutputStream
    - Reader (an input stream optimized for Unicode text)
    - Writer (an output stream optimized for Unicode text)
- There are three predefined streams that may be used without any construction
  - System.in - for reading from the keyboard
  - System.out - for writing to the screen
  - System.err - for writing error messages to the screen
Subclasses of Streams

• A subclass of stream is either a sink, a source, or a filter.
  • Sources include FileInputStreams, PipedInputStreams, etc.
  • Sinks include FileOutputStreams, PipedOutputStreams, etc.
  • Filters include BufferedInputStream, DataInputStream, BufferedOutputStream, DataOutputStream, etc.

• These filters, or processing streams provide additional functionality on “top” of the source and sink streams.

• It should be noted that there is a RandomAccessFile class that allows both reading and writing to a file.
Data sources and sinks

• As may be deduced from the previous slides, there are different kinds of I/O streams
  • FileInputStream - for reading from files
  • FileOutputStream - for writing to files
  • PipedInputStream – for reading from a thread
  • PipedOutputStream – for writing to a thread
• There are subtle additional types for the Readers and Writers which are specialized to deal with Unicode encoded text.
Filters and Processing streams

- DataInputStream – processes the primitive Java types from an input stream allowing them to be assigned to variables.
- DataOutputStream – processes the primitive Java types to an output stream
- BufferedInputStream – optimizes I/O from a stream by using a buffer
- BufferedOutputStream – optimizes I/O to a stream by using a buffer
- PrintStream – converts various primitive data types to text before placing them in the stream
- PushbackInputStream – allows look ahead input such as might be required for tokenizing or parsing
The basic procedures for accessing a stream

• Open the stream by calling its constructor (Java streams do not use an open() method)
• Read or write to the stream using the methods associated with that particular stream (when layering streams, use the methods of the outermost stream)
• Close the stream with the close() method
Layering output streams

- Variable
- Processing Stream
- Data Sink Stream
- Destination
Key points

- Import the java.io package when working with streams
- `InputStreams` and `OutputStreams` are 8-bit and treat all data as simple streams of 8 bit bytes
- `Readers` and `Writers` are 16-bit and are used for Unicode text data
- Filters or processing streams are used for performing intermediary operations on data
- Filter streams are layered on top of sink and source streams.
- Remember to close a stream when finished using it
- Closing an output stream flushes its buffer
Streams and Exceptions

- Java has a method of handling errors that encloses code subject to failures or exceptions in “try/catch” blocks.
- I/O tends to be subject to a large number of exceptions (file not found, file locked, I/O errors, etc.)
- Care needs to be taken in writing Java code that involves streams to anticipate and handle the various exceptions that might occur.
Example

String name = new String("John Smith");
int age = 23;
double qpa = 3.756;
try{
    File F1 = new File("binary.dat");
    File F2 = new File("text.dat");
    FileOutputStream OFS1 = new FileOutputStream(F1);
    FileOutputStream OFS2 = new FileOutputStream(F2);
    DataOutputStream ODS = new DataOutputStream(OFS1);
    PrintStream OPS = new PrintStream(OFS2);
    ODS.writeChars(name);
    ODS.writeInt(age);
    ODS.writeDouble(qpa);
    OPS.print(name);
    OPS.print(age);
    OPS.print(qpa);
}
Results of the Program

- Both streams started with the following data:
  ```java
  String name = new String("John Smith");
  int age = 23;
  double qpa = 3.756;
  ```

- The DataOutputStream took the information provided and produced a file with the following contents:
  ```plaintext
  John Smith 23 3.756
  ```

- The PrintStream took the same information and produced a file with the following contents:
  ```plaintext
  John Smith 23 3.756
  ```

- If read back in with the appropriate filters the original data correctly formed would be returned.