`java.io` Tool Talk: Streams and Files

David Millard and Matthew Saltz
{dmillard,saltzm}@uga.edu

January 14, 2013

I/O Streams

- An *I/O Stream* provides an input source or output destination to a program (e.g. disk files, devices, other programs, memory arrays, etc.)
- Streams support many data types (including objects) and may be passive (forward data) or active (transform data)
- All streams provide the same model to the program: streams are sequences of data

$$\boxed{\text{Data}} \xrightarrow{\text{Input Stream}} \boxed{\text{Program}}$$

$$\boxed{\text{Program}} \xrightarrow{\text{Output Stream}} \boxed{\text{Data}}$$

Figure 1: Visualization of streams

Simple Example

To demonstrate common stream functions, we will use the very basic *byte stream* with the following input file `xanadu.txt`, which contains:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

To follow along with the examples, run:

```
git clone https://github.com/dmillard/javaio_examples.git
```

Simple Example (contd.)

- Programs use *byte streams* to directly input and output bytes.
- All byte stream classes are descended from `InputStream` and `OutputStream`
- This example uses `FileInputStream` and `FileOutputStream`
- The code for this example is in `Example1_CopyBytes.java`

Notes on Example 1

- Always close streams: this helps to prevent resource leaks
- As Example 1 showed, closing in a `finally` block helps ensure that all streams are appropriately closed
- Although Example 1 is very simple, it is too low level: as we are dealing with character data, we should be using *character streams*

Character Streams

- Character stream I/O is generally no more complex than byte stream I/O
- Character stream I/O, however, automatically translates from internal Unicode to the local character set (useful for internationalization)
- All character stream classes are descended from `Reader` and `Writer`. As with byte streams, there are character stream classes specialized for files: `FileReader` and `FileWriter`
- The code for this example is in `Example2_CopyCharacters.java`

Byte vs. Character Streams

- Both examples have the same effect: what is different?
- Obviously, different classes: `FileReader` vs. `FileInputStream`
- Internally, `int c` holds different values:
  - In the byte stream example, the last 8 bits hold the byte to be copied
  - In the character stream example, the last 16 bits hold the character to be copied
- Character streams can often be wrappers for byte streams: `FileReader` for example, uses `FileInputStream` internally

Line-Oriented I/O

- Character I/O rarely happens character-by-character; commonly it is line-by-line
- A line is a string of characters with a line terminator (`"\r\n"`, `"\r"`, or `"\n"`)
- This example uses `BufferedReader` and `PrintWriter`, which will be discussed more in depth later
- The code for this example is in `Example3_CopyLines.java`

Buffered Streams

- Examples 1 and 2 use *unbuffered* I/O, meaning that each read and write request is handled directly by the OS
- This is often slow: each request often will trigger disk access
- We can reduce this overhead with *buffered streams*, which read data from a memory area (known as a *buffer*)
- Thus, OS calls are only made when the buffer is empty
- This functionality is implemented with `BufferedReader` and `BufferedWriter`, which are buffered drop-ins for `Reader` and `Writer`
- To buffer input to Example 2, we could have written:

```
out = new BufferedWriter(new FileWriter("xanadu.txt"));
```

Tokenizing and Translating Input

- `Scanner` objects are useful for breaking down formatted input into individual tokens and translating according to type
- By default, `Scanner`s use whitespace characters to delimit tokens
- The code for this example is in `Example4_Tokenize.java`
- Scanners can interpret character encoded data as types, as well (e.g. "15.2" $\rightarrow$ 15.2 and "1,234.5" $\rightarrow$ 1234.5)
- The code for this example is in `Example5_Translate.java`

Data Streams

- *Data streams* support binary I/O of primitive data type values, as well as String values
- All data streams implement either the `DataInput` interface or the `DataOutput` interface
- The commonly used implementations are `DataInputStream` and `DataOutputStream`
- The code for this example is in `Example6_DataStreams.java`
- Of course, you should never use `double` to represent currency values, which leads us to...

Object Streams

- ≫ Like data streams provide stream I/O for primitive data types, object streams support stream I/O for objects
- ≫ Most (not all) standard classes support serialization
- ≫ Classes supporting serialization implement `Serializable`
- ≫ The object streams are `ObjectInputStream` and `ObjectOutputStream`
- ≫ These implement `ObjectInput` and `ObjectOutput`, which are sub-interfaces of `DataInput` and `DataOutput`

Complex Objects with Streams

- `writeObject` and `readObject` are simple to use, but they contain some sophisticated object management logic
- Consider an object which contains references to other objects
- `writeObject(object0)` will write all objects necessary to reconstitute `object0`
- What if two objects written to the same stream both contain references to a single object?
  - Both will refer to a single object when they are read back
  - A stream can only contain one copy of an object, buy any number of references to that object
  - Therefore, two writes of the same object is actually a single write of the object and two writes of references to the object
- A simple example is in `Example7_ObjectStream.java`

Files

- Java `File`s are abstract representations of file and directory pathnames
- Java `File` provides some methods for manipulating pathnames, like `getParent()`, which gives the parent directory of the file
- Though all examples use pathnames directly as `FileReader` constructor parameters, they could use `File` objects with more flexibility
- For more control over how information is written to disk, we turn to `RandomAccessFile`s

Random Access Files

- `RandomAccessFiles` support both reading and writing to a random access file
- A random access file behaves like a large array of bytes on disk
- There is an index, or cursor, called the *file pointer*, which which determines at which point bytes are written and read
- The *file pointer* can be read by the `getFilePointer` method and set by the `seek` method
- `RandomAccessFile` implements `DataOutput` and `DataInput`, and therefore provides the reading and writing functionality we would expect from streams

References

- ≫ docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html
- ≫ docs.oracle.com/javase/7/docs/api/java/io/File.html
- ≫ docs.oracle.com/javase/tutorial/essential/io/index.html