I/O and Decorators

Mark Allen Weiss
Copyright 2000

Overview

- Console I/O not a priority; GUIs are expected for user interaction.
- Lots of classes to do I/O.
- Major revision after Java 1.0; lots of stuff deprecated.
- Java I/O library meant to be extensible, so lots of classes and inheritance
- Uses same notion of streams seen in C++
  - Associate a “stream” object with a file, terminal, etc
  - All I/O is directed to the stream
Four Important Classes

- Byte-oriented abstract classes
  - InputStream
  - OutputStream
- Char-oriented abstract classes
  - Reader
  - Writer
- Java has single inheritance, so no such thing as a stream open for reading and writing at same time

Various Byte-Oriented Subclasses

- **InputStream** subclasses include:
  - FileInputStream, SocketInputStream, ByteArrayInputStream, FilterInputStream, DataInputStream, ObjectInputStream, ZipInputStream
- **OutputStream** subclasses include:
  - FileOutputStream, SocketOutputStream, ByteArrayOutputStream, FilterOutputStream, DataOutputStream, ObjectOutputStream, ZipOutputStream
Various Char-Oriented Subclasses

- **Reader** subclasses include:
  - FileReader, BufferedReader, InputStreamReader, CharArrayReader, PipedReader, PushbackReader, LineNumberReader

- **Writer** subclasses include:
  - FileWriter, PrintWriter, OutputStreamWriter, CharArrayWriter, PipedWriter

Reading and Writing Primitives

- **Both InputStream and Reader support**
  
  ```java
  int read() throws IOException
  ```
  - for InputStream, returns 1 byte
  - for Reader, returns 1 char
  - for both, returns -1 for EOF

- **Both OutputStream and Writer support**
  
  ```java
  void write( int x) throws IOException
  ```
  - for OutputStream, writes 1 byte
  - for Writer, writes 1 char
Copying Using Primitives

- Can copy using primitives:
  ```java
  public static void copy( Writer out, Reader in )
  throws IOException
  {
    int ch;

    while( ( ch = in.read( ) ) != -1 )
      out.write( ch );
  }
  ```
- Works for any Reader/Writer pair.
- Cannot pass in InputStream or OutputStream; those are separate hierarchies.
- Thus, cannot pass in System.in

Bad Code

- This code compiles but fails with Unicode:
  ```java
  public static void copy( Writer out, InputStream in )
  throws IOException
  {
    int ch;

    while( ( ch = in.read( ) ) != -1 )
      out.write( ch );
  }
  ```
- Most of 1.0 I/O deprecated because of assumptions that this code works.
- Moral: don’t mix and match byte-oriented and char-oriented stuff
Bridging From Bytes to Chars

- Construct `InputStreamReader` or `OutputStreamWriter` to get from byte-oriented world to char-oriented world.

Correct Code

- This code works with Unicode

```java
public static void copy(Writer out, InputStream in) throws IOException {
    int ch;
    Reader cin = new InputStreamReader(in);

    while ( ( ch = cin.read() ) != -1 )
        out.write( ch );
}
```
Bulk Reading and Writing

- **InputStream** and **OutputStream** can read/write byte [ ]
  
  ```java
  int read(byte[] b) throws IOException
  void write(byte[] b) throws IOException
  ```

- **Reader** and **Writer** can read/write char [ ]
  (similar calls)

- Usually more efficient than single byte/char at a time

- **read** returns number of bytes/chars read or -1 if EOF

---

Files

- Just get appropriate stream:
  - FileInputStream, FileOutputStream, FileReader, FileWriter

- I/O is NOT buffered (more on that in a minute)

- Construct stream object with name of the file

- **FileNotFoundException** (for input) or **IOException** (for output) may be thrown

- Output files will truncate if already present, but can open in append mode if you need to

- You are responsible for closing file streams
Closing Streams

- Actually fairly cumbersome to close a stream
- Need to
  - put the `close` inside a finally block to make sure it is reached
  - declare the stream reference outside the try block so it is visible to the finally block
  - initialize stream reference to `null` to satisfy definite assignment rules
  - check if stream reference is `null` before call to `close` or you can get a `NullPointerException`
  - put in try/catch block because `close` might throw an exception itself!

Decorator Pattern For Streams

- Basic classes don’t have much functionality.
- Nothing in these classes for
  - reading line at a time
  - writing line at a time
  - reading bytes as a primitive
  - unreading
  - buffering
- Could put all these in each class (i.e. socket class, file class, bytarray class, etc.)
- Maintenance nightmare: what if I want to add a new property to everyone?
Decorator Pattern For Streams

- Java provides classes that allow you to add (i.e. decorate) the basic classes.
- For example, BufferedReader.
- Want buffering? Take any Reader, and wrap in a BufferedReader.
- BufferedReader IS-A Reader so it has a read method. Its read checks its internal buffer, and if empty, forwards the call for a block read to the Reader it is wrapping.
- Generally: construct Reader from any Reader, InputStream from any InputStream etc ...

Buffered Copy

// Input is now Buffered
// Output we don’t know about; could wrap in a
// PrintWriter, which is unusual because it
// buffers by default
public static void copy( Writer out, InputStream in )
throws IOException
{
    int ch;
    Reader cin = new InputStreamReader( in );
    Reader bin = new BufferedReader( cin );

    while( ( ch = bin.read( ) ) != -1 )
        out.write( ch );
}
Decorator Analysis

- Constructor frenzy is a pain
- Ability to mix and match exactly what you want is great
- Ability to add new decorations makes library very extensible

Classes I Like

- **DataInputStream, DataOutputStream:**
  - Construct with any other ...Stream
  - Can write primitives as sequence of bytes (int is 4 bytes, double is 8 bytes, etc.)
- **BufferedReader:**
  - Construct with any other Reader
  - Turns on buffering, and also provides readLine
- **PrintWriter**
  - Construct with any other Writer or OutputStream (unusual)
  - Buffers and provides print and println
Example of a Copy Program

```java
public static void main(String[] args) { // Most error checks omitted
    PrintWriter fileOut = null;
    BufferedReader fileIn = null;

    try {
        if (args.length == 1)
            fileOut = new PrintWriter(System.out);
        else
            fileOut = new PrintWriter(new BufferedWriter(new FileWriter(args[1])));
        fileIn = new BufferedReader(new FileReader(args[0]));

        String oneLine;
        while ((oneLine = fileIn.readLine()) != null)
            fileOut.println(oneLine);
    }
    catch (IOException e) { e.printStackTrace(); }

    finally {
        try {
            if (fileOut != null) fileOut.close(); // doesn’t throw IOException
            if (fileIn != null) fileIn.close(); // might throw IOException
        }
        catch (IOException e) { e.printStackTrace(); }
    }
}
```

Reading Formatted Data

- So you want to read numbers from the terminal (`System.in`)
- Can’t use `DataInputStream`; that reads bytes not formatted text.
- Instead:
  - Wrap `System.in` into an `InputStreamReader`
  - Wrap `InputStreamReader` into `BufferedReader`
  - Use `readLine` to get line at a time as a `String`
  - parse the `String` into its constituent tokens
  - For files use `FileReader` to construct `BufferedReader`
java.util.StringTokenizer

- Used for parsing a String into tokens
- Construct with a String
- By default tokens are separated by white space; can change the default if you need to
- Tokens are Strings
- Important methods are: countTokens, hasMoreTokens, nextToken

```java
int[] nums = new int[3];
StringTokenizer st = new StringTokenizer("37 64 29");
int numTokens = st.countTokens();  // returns 3
while (st.hasMoreTokens())
    nums[i] = Integer.parseInt(st.nextToken());
```

StringTokenizer Example

```java
import java.io.*;
import java.util.StringTokenizer;

// Read two integers and output the maximum.
public class MaxTest {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter 2 ints on one line: ");
        try {
            String oneLine = in.readLine();
            StringTokenizer str = new StringTokenizer(oneLine);
            if (str.countTokens() != 2) {
                System.err.println("Error: need two numbers"); return;
            }
            int x = Integer.parseInt(str.nextToken());
            int y = Integer.parseInt(str.nextToken());
            System.out.println("Max: " + Math.max(x, y));
        }
        catch (IOException e) { System.err.println("Unexpected I/O exception"); }
        catch (NumberFormatException e) { System.err.println("#s not both ints"); }
    }
}
```
Getting File System Info

- Can use the `File` class to find out
  - if a file exists
  - info about the file such as size and date modified
  - if the file is a directory
  - if it is a directory, what files are in the directory

Summary

- Four abstract classes: `InputStream`, `OutputStream`, `Reader`, `Writer`
- Don’t mix bytes and char; use the two bridges to cross: `InputStreamReader`, `OutputStreamReader`
- Library uses decorator pattern to add options
- Can do simple formatting input parsing with `readLine` and `StringTokenizer`
- `File` class used to get file system info
Next Time

- The Collections API
  - Interesting design decisions
  - Lists
  - Sets
  - Maps
  - Iterators
  - Sorting