CSE 143

Streams

Reading: Ch. 16

Overview

- Topics
  - Data representation – bits and bytes
  - Streams – communicating with the outside world
  - Basic Java files
  - Other stream classes

GREAT IDEAS IN COMPUTER SCIENCE

REPRESENTATION VS. RENDERING

Data Representation

- Underneath it's all bits (binary digits – 0/1)
- Byte – group of 8 binary digits
  - Smallest addressable unit of memory
- Meaning depends on interpretation
  - Non-negative base-10 integers represented as base-2 integers
  - Characters formats include ASCII (1 byte) or Unicode (2 byte) encodings

    01000001 = integer 65 = ASCII 'A'
    Unicode 'A' is 0000000001000001
    00111111 = integer 63 = ASCII '?'
    00110110 = integer 54 = ASCII '6'
- But it's still just bits

Representation of Primitive Java Types

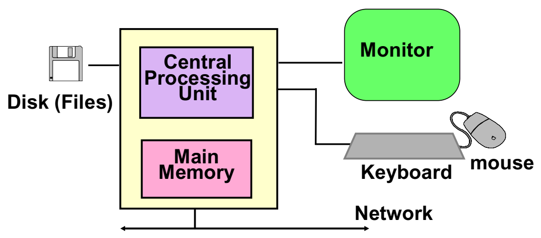
- Boolean – 1 byte (0 = false; 1 = true)
- Integer types
  - `byte` – 1 byte (-128 to 127)
  - `short` – 2 bytes (-32768 to 32767)
  - `int` – 4 bytes (-2147483648 to 2147483647)
  - `long` – 8 bytes (-9223372036854775808 to 9223372036854775807)
- Floating-point (real number) types
  - `float` – 4 bytes; approx. 6 decimal digits precision, magnitude ~$10^{\pm 38}$
  - `double` – 8 bytes; approx. 15 decimal digits precision, ~$10^{\pm 308}$
- Character type
  - `char` – 2 bytes; *Unicode* characters w/decimal values 0 to 65535

Unicode

- International standard
  - Java was first major language to adopt
- Intended to include all the world's writing systems
- Characters are 2 bytes (16 bits)
  - Given by Hex digits, e.g. 4EB9
- Specifications: www.unicode.org
- Unicode 3.1 (2001) introduced characters outside the original 16-bit range
  - Not yet well-supported

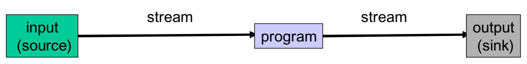
Input and Output

- Communicating with the outside world



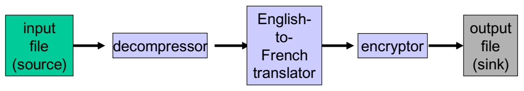
Streams

- Java model of communication: streams
  - Sequence of data flowing from a source to a program, or from a program to a destination (sink)
  - Files are common sources and sinks
  - Others: network sockets, interprogram communication, ...



Stream after Stream...

- Stream are a useful model for processing data along the way, in a pipeline



Other Possible Kinds of Stream Converters

- Compression
- Encryption
- Filtering
- Translation
- Statistics gathering
- Security monitoring
- Routing/Merging
- Reducing Bandwidth (Size & Detail), e.g. of graphics or sound
  - "lossy compression"
- Noise reduction, image sharpening, …
- Many, many more…

Streams vs. Files

- Many languages don't make clear distinction
  - Programmers, too!
- In Java:
  - "file" is the collection of data, managed by the operating system
  - "stream" is a flow of data from one place to another
- A stream is an abstraction for data flowing from or to a file, remote computer, URL, hardware device, etc.

Java Stream Library

- Huge variety of stream classes in java.io.*
  - Some are data sources or sinks
  - Others are converters that take data from a stream and transform it somehow to produce a stream with different characteristics
- Highly modular
  - Lots of different implementations all sharing a common interface; can be mixed and matched and chained easily
  - Great OO design example, in principle
  - In practice, it can be very confusing (simple I/O is messy)
    (improved simple I/O coming in Java 1.5)

Common Stream Processing Pattern

- Basic idea the same for input & output

```
// input                      // output
open a stream                 open a stream
while more data {             while more data {
    read & process next data      write data to stream
}                             }
close stream                  close stream
```

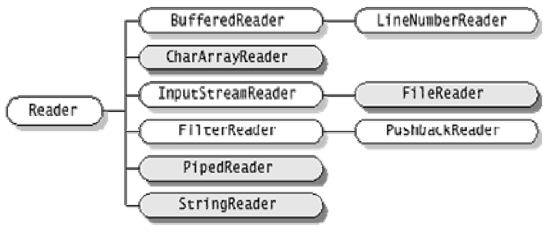
Opening & Closing Streams

- Before a stream can be used it must be *opened*
  - Create a stream object and connect it to source or destination of the stream data
  - Often done implicitly as part of creating stream objects
- When we're done with a stream, it should be *closed*
  - Cleans up any unfinished operations, then breaks the connection between the program and the data source/destination

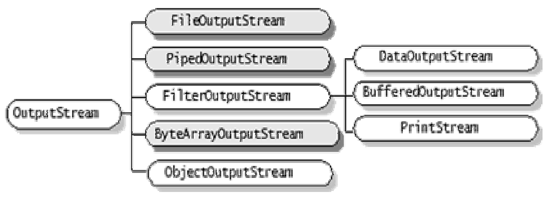
Java Streams

- 2 major families of stream classes
- *Byte streams* – read/write `byte` values
  - Corresponds to physical data – network and disk I/O streams
  - Abstract classes: InputStream and OutputStream
- *Character streams* – read/write `char` values
  - Added in Java 1.1
  - Primary (Unicode) text input/output stream classes
  - Abstract classes: Reader and Writer
  - Footnote: System.in and System.out should be character streams, but are byte streams for historical reasons (existed before Java 1.1, when character streams were added, and remain unchanged to preserve backward compatibility)

Character Input Streams



Byte Output Streams



8/8/2004 (c) 2001-4, University of Washington 12-17

Streams and Exceptions

- Many operations can throw IOException
  - All input operations, in particular
- Normally throws a specific subclass of IOException
  - depending on the actual error
- IOException is "checked"
  - (Review question: what does a "checked" exception imply?)

8/8/2004 (c) 2001-4, University of Washington 12-18

Basic Reader/Writer Operations

- Reader

```
int read( );              // return Unicode value of next character;
                          //    return -1 if end-of-stream
int read(char[ ] cbuf);   // read several characters into array;
                          //    return -1 if end-of-stream
void close( );            // close the stream
```

- Writer

```
void write(int c);        // write character whose Unicode value is c
void write(char[ ] cbuf); // write array contents
void write(String s);     // write string
void close( );            // close the stream
```

- To convert Unicode int to char, or vice versa: use a cast

8/8/2004 (c) 2001-4, University of Washington 12-19

File Readers and Writers

- To read a (Unicode) text file (not a binary data file), instantiate FileReader
  - A subclass of Reader: implements read and close operations
  - Constructors take a File object or the name of the file to open and read from
- To write to a text file, instantiate FileWriter
  - A subclass of Writer: implements write and close operations
  - Constructors take a File object or the name of the file to open/create and overwrite (can also append to an existing file using a different constructor)

8/8/2004 (c) 2001-4, University of Washington 12-20

Text Files vs Char Data

- Most of the world’s text files use 8-bit characters
  - ASCII and variations of ASCII
  - Internal to Java, char data is *always* 2-byte Unicode
  - Java Reader deals only with Unicode
- Big problem: how to read and write normal (ASCII) text files in Java?
- Solution: stream classes which adapts 8-bit chars to Unicode
  - Generally taken care of automatically – normally don’t need to worry about the distinction

Copy a Text File, One Character at a Time

```
public void copyFile(String sourceFilename, String destFilename)
                          throws IOException {
    FileReader inFile = new FileReader(sourceFilename);
    FileWriter outFile = new FileWriter(destFilename);
    int ch = inFile.read( );
    while (ch != -1) {
        outFile.write(ch);
        System.out.println("The next char is \"" + (char)ch + "\"");    // why \' ?
        ch = inFile.read( );
    }
    inFile.close( );
    outFile.close( );
}
```

Interlude: Where is the File?

- In the previous slide, we opened the files with

```
FileReader inFile = new FileReader(sourceFilename);
FileWriter outFile = new FileWriter(destFilename);
```

- The file names could be complete paths like “c:\Documents and Settings\J User\story.txt”, but…
  - Not portable – different operating systems have different file naming conventions
  - Not convenient – what if we move the document?
- Would like to be able to use a name like “story.txt” to open the file
  - But if we do, where should we put the file?

File Directories

- When we use a simple file name

```
FileReader inFile = new FileReader(“story.txt”);
```

Java looks for that file in the “current directory”

- Current directory
  - If the program is executed from a command-line prompt, it is the current directory when the “java” command is entered
  - If it is executed by DrJava, BlueJ, or other development tools, it likely is not
  - Is there a portable scheme way to find the file, assuming it’s in the same directory or jar file as the main program .class file?

    Yes – but you might not really want to have to know the details

Finding Files (optional)

- The industrial-strength solution is to use a class loader method that will search all directories it knows about
  - Includes the directory or jar file containing the program's .class files, Java standard libraries, any additional libraries on the *classpath*, etc.
- Ready?

```
URL url = getClass().getClassLoader().getResource(fileName);
```

  - If url!=null, then it can be used to open the file (also works for other resources like images and icons)
  - Credit: Found on bluej.org; see their tip #10 for more details
  - No, this won't be on the test

Opening Files Using File Dialogs

- Easy, portable solution for our purposes is JFileDialog
- Lots (tons) of options, but basic use is quite simple

```
JFileChooser chooser = new JFileChooser();
int result1 = chooser.showOpenDialog(null);
File inFile = chooser.getSelectedFile();
System.out.println("Input file selected is " + inFile);
int result2 = chooser.showSaveDialog(null);
File outFile = chooser.getSelectedFile();
System.out.println("Output file selected is " + outFile);
```

  - The int results of the show...Dialog methods indicate whether the dialog was dismissed with ok, cancel, or something else
    Should really check this before getting the selected file info

More Efficient I/O – BufferedReader/Writer

- Can improve efficiency by reading/writing many characters at a time
- BufferedReader: a converter stream that performs this chunking
  - BufferedReader constructor takes any kind of Reader as an argument -- can make any read stream buffered
  - BufferedReader supports standard Reader operations -- clients don't have to change to benefit from buffering
  - Key addition: provides a portable readLine( )

```
String readLine( );     // return an entire line of input; or null if
                        //    end-of-stream reached
```

[handles the complexities of how end-of-line is represented on different systems]

BufferedWriter

- BufferedWriter: a converter stream that performs chunking on writes
  - BufferedWriter constructor takes any kind of Writer as an argument
  - BufferedWriter supports standard Writer operations
  - Also supports newLine( )

```
void newLine( );     // write an end-of-line character
```

Copy a Text File, One *Line* at a Time

```
public void copyFile(String sourceFilename, String destFilename)
                              throws IOException {
    BufferedReader inFile = new BufferedReader(new FileReader(sourceFilename));
    BufferedWriter outFile = new BufferedWriter(new FileWriter(destFilename));
    String line = inFile.readLine( );
    while (line != null) {
        outFile.write(line);
        outFile.newLine( );
        System.out.println("The next line is \"" + line + "\"");
        line = inFile.readLine( );
    }
    inFile.close( );
    outFile.close( );
}
```

PrintWriter

- PrintWriter is another converter for a write stream
  - Adds print & println methods for primitive types, strings, objects, etc., just as we've used for System.out
  - Does not throw exceptions (to make it more convenient to use)
  - Optional 2nd boolean parameter in constructor to request output be flushed (force all output to actually appear) after each println

    Useful for interactive consoles where messages need to appear right away

Copy a Text File, Using PrintWriter

```
public void copyFile(String srcFilename, String destFilename)
                              throws IOException {
    BufferedReader inFile = new BufferedReader(new FileReader(srcFilename));
    PrintWriter outFile =
        new PrintWriter(new BufferedWriter(new FileWriter(destFilename)));
    String line = inFile.readLine( );
    while (line != null) {
        outFile.println(line);
        System.out.println("The next line is \"" + line + "\"");
        line = inFile.readLine( );
    }
    inFile.close( );
    outFile.close( );
}
```

StringReader and StringWriter

- Strings as streams(!)
- StringReader: construct character stream from a String

```
StringReader inStream = new StringReader("the source");
// could now write inStream to a file, or somewhere else
```

- StringWriter: write stream to a String

```
StringWriter outStream = new StringWriter( );
// now write onto outStream, using outStream.write(…), outStream.print(…), etc.
String theResult = outStream.toString( );
```

Binary Streams

- For processing binary data (encoded characters, executable programs, other low-level data), use InputStreams and OutputStreams
- Operations are similar to Reader and Writer operations
  - Replace char with byte in read; no write(String)
- Many analogous classes to Readers and Writers:
  - FileInputStream, FileOutputStream
  - BufferedInputStream, BufferedOutputStream
  - ByteArrayInputStream, ByteArrayOuputStream
  - ObjectInputStream, ObjectOutputStream -- read & write whole objects!

Conversion from Binary to Text Streams

- InputStreamReader: creates a Reader from an InputStream

```
// System.in is of type InputStream
Reader inStream = new InputStreamReader(System.in);
// now can treat it nicely as a character stream
```

- OutputStreamWriter: creates a Writer from an OutputStream

```
// System.out is of type OutputStream
Writer outStream = new OutputStreamWriter(System.out);
// now can treat it nicely as a character stream
```

Network Streams

- Import java.net.*
- Use URL to create a name of something on the web
- Use openStream() method to get a InputStream on the contents of the URL

```
URL url = new URL("http://www.cs.washington.edu/index.html");
InputStream inStream = url.openStream( );
… // now read from inStream
```

- Use openConnection( ) and URLConnection methods to get more control

```
URLConnection connection = url.openConnection( );
OutputStream outStream = connection.getOutputStream( );
… // now write to outStream (assuming target url allows writing!)
```

- Socket class for even more flexible network reading & writing

Summary

- Java stream libraries
  - Comprehensive, flexible, easy to compose multiple streams in a chain
  - But not simple to do simple things
- What to take away
  - BufferedReader and readLine( ) for text input
  - PrintWriter and print( )/println( ) for text output
  - JFileChooser to select files when opening
  - close( ) when done