COMP241
Software Engineering Development
Lecture 18: Java I/O 2 (text I/O)

Mark Hall

- Text IO
- Writing Text
- Reading Text
- Bridging Text and Binary Streams
- Tokenizing Text Input
  - `Scanner`
- Example

THE UNIVERSITY OF WAIKATO *DEPARTMENT OF COMPUTER SCIENCE*
TARI ROROHIKO

Text IO

- Previously we looked at reading data in *binary* format
  - Used **`InputStream`** and **`OutputStream`** classes and their subclasses to read and interpret 8 bit *bytes*
- In *text* format, data items are represented in human-readable form, as a sequence of characters
  - Eg the integer 12,345 is stored as the sequence of five characters: '1' '2' '3' '4' '5'

Text IO

- Text input and output is more convenient for humans
  - Easier to produce input — just use a text editor
  - Easier to check output — just look at the output file in an editor
- However, binary storage is more compact and efficient
- If you store information in text form you need to use the **`Reader`** and **`Writer`** abstract class and their subclasses to process input and output
  - Read/write characters — not the same as bytes in all languages!
  - Convert bytes to/from 16 bit *Unicode* characters

Common Error

- Reading characters using the **`read`** method from an `InputStream`

```
InputStream myIn = . . .
byte next = (byte)myIn.read();
If (next == 'é') . . . //never true!
```

- Previously we saw that bytes hold values in the range –128—127
  - `é` has the unicode value of 233
- Ok if we cast next to a `char` instead of a byte in this example, but…
  - the read method in **`InputStream`** still only returns a value in the range 0-255, so international programmers who use characters with Unicode values outside this range would have problems

Writing Text Files

- **`FileWriter`** `w = new` **`FileWriter`**`("output.txt")`
  - Now we can send output to a file, one character at a time by calling the **`write`** method
- Typically, we don't have our output available one character at a time
  - Instead, we have numbers and strings
  - Need the **`PrintWriter`** class to break up numbers and strings into individual characters for us
  - Can construct a **`PrintWriter`** using any `Writer`
  - `PrintWriter` is another example of the *Decorator* design pattern — adds extra functionality to any **`Writer`**

Writing Text Files

```
Writer w = new FileWriter("output.txt");
PrintWriter pw = new PrintWriter(w);
```

- Now you can use the familiar `print` and `println` methods to print numbers, strings and objects

```
pw.println(29.5);
pw.println(new Rectangle(5,10,15,25));
pw.println("Hello World!");
```

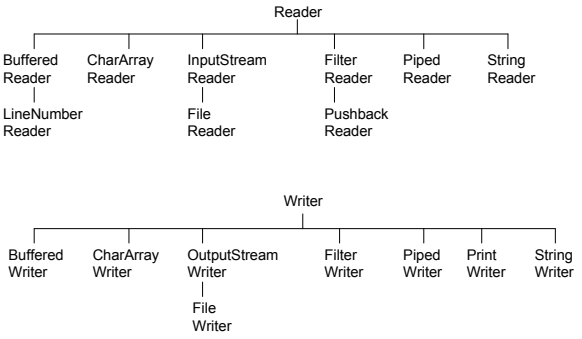
Writing Text Files

- The **`print`** and **`println`** methods in `PrintWriter` are overloaded methods that
  - Convert numbers to their decimal string representations
  - Use the **`toString`** method to convert objects to strings
- Strings are then broken up into individual characters and passed on to the underlying `Writer` (a `FileWriter` in this example)

Reading Text Files

- Is there an analogue of `DataInputStream` (eg. DataInputReader) that lets you read in data from text files?
  - Prior to Java 1.5, No… The best you could do is use the **`BufferedReader`** class
    - Has a `readLine` method that lets you read a line at a time
    - `readLine` keeps calling the read method of the supplied underlying `Reader`, until it has collected an entire input line
    - `readLine` returns `null` when there is no further input
  - From Java 1.5 we can use the **`Scanner`** class
    - Has methods for reading numbers (floats, doubles, ints, shorts), strings, lines and bytes from the input
    - Can *tokenize* the input on the basis of arbitrary sets of delimiters, including regular expressions

Reader and Writer hierarchies



Bridging Text and Binary Streams

- **`FileReader`**
  - Assumes that the file is encoded using the default character encoding for the default locale
  - Converts characters from the local encoding to Unicode as it reads them
  - Does the most of the hard work involved in internationalizing the character set handling of your program
- `FileReader` extends **`InputStreamReader`**
  - `InputStreamReader` is a bridge from byte streams to character streams
    - Another example of the ***Adapter*** design pattern
  - Allows the specification of a character encoding to use so that files encoded using a character set other than the default for the locale can be read

Bridging Text and Binary Streams

- **`InputStreamReader`** allows text to be read from *any* underlying byte stream
  - Eg. java.net has classes for socket based TCP/IP network communication
  - **`Socket`** class has `getInputStream` and `getOutputStream` methods — more on this later
- **`System.in`** and **`System.out`** (standard in and standard out) are instances of **`InputStream`** and **`OutputStream`** respectively
  - This is a legacy from Java 1.0 — before Readers and Writers were introduced
  - For top efficiency and Unicode support wrap in/out in `Input/OutputStreamReader/Writer` and Buffering

```
BufferedReader myIn =
   new BufferedReader(new InputStreamReader(System.in));
```


Tokenizing Text Input with Scanner

- Not actually an input stream as such
  - Can be constructed with a `File`, `InputStream`, `String` or `Reader`

```
Scanner st = new Scanner(r);
```

- Very useful for breaking a text file into a sequence of *tokens*
  - Has a default delimiter pattern that matches all "whitespace"
  - The resulting tokens can be converted into values of different types using various `next` methods
    - `nextInt, nextDouble, nextLine` etc.

Scanner

- Has methods to search for patterns that operate independently of the delimiter pattern
  - `findInLine` — takes a pattern to search for
  - `findWithinHorizon` — takes a pattern to search for and an integer as arguments
  - `skip` — takes a pattern to skip in the input


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

public class ScannerTest {
  public static void main(String [] args) {
    if (args.length != 1) {
      System.err.println("Usage ScannerTest <filename>");
      System.exit(1);
    }
    try {
      Scanner s = new Scanner(new File(args[0]));
      while (s.hasNext()) {
        if (s.hasNextInt() || s.hasNextDouble()) {
          System.out.println("number: " + s.next());
        } else if (s.hasNextBoolean()) {
          System.out.println("boolean: " + s.next());
        } else {
          System.out.println("word: " + s.next());
        }
      }
    } catch (Exception ex) { ex.printStackTrace(); }
  }
}
```

```
here is a test 29.3!
Testing my boolean true
** hi there bob!
```

Input file "scTest"

```
bash-2.05a$ java ScannerTest scTest
word: here
word: is
word: a
word: test
word: 29.3!
word: testing
word: my
word: boolean
boolean: true
word: **
word: hi
word: there
word: bob!
```

Running the ScannerTest.java program and output to the console

Example: A LZ77 Compressor for Text Files Using Text IO

- Substitution (or dictionary) based compression
  - Replace an occurrence of a particular phrase in a piece of data with a *reference* to a previous occurrence of that phrase
- LZ77 — invented by two Israeli professors (Ziv and Lempel
  - Keep track of the last *n* symbols seen in a search buffer
  - When a sequence of **input** symbols is encountered that is in the search buffer output a triple to encode it:
    - *<position, length, following-symbol>*
    - *position* is the index in the search buffer of the match
    - *length* is the length of the match
    - *following-symbol* is the **input** symbol that occurs immediately after the match

Example: A LZ77 Compressor for Text Files Using Text IO

- LZ77
  - The reason for outputting the *following-symbol* is to take care of the case when there is no match in the search buffer for the current input symbol
    - In this case *position* and *length* are set to 0 and the *following-symbol* is set to the input symbol

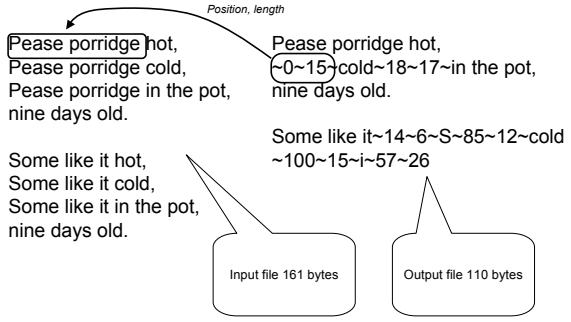
```
// LZ77 compression algorithm
While there are more input symbols to read {
  match as many consecutive input symbols to the searchBuffer;

  output a (position, length, symbol) triple;
  append processed input symbols to the searchBuffer;
}
```


Example: A LZ77 Compressor for Text Files Using Text IO

- To improve compression we can:
  - Omit outputting the *position* and *length* when there is no match for the current input symbol in the search buffer
  - Only output our encoded triple when it comprises fewer symbols than the raw data
- LZ77 forms the basis of many popular compression formats such as zip, gif etc.

Example: A LZ77 Compressor for Text Files Using Text IO



Example: A LZ77 Compressor for Text Files Using Text IO

- We'll use a **`StringBuffer`** (from `java.lang`) to be our search buffer
- For compressing
  - **`BufferedReader`** to read characters from an input file
  - **`PrintWriter`** to write out our encoded text
- For uncompressing
  - **`Scanner`** to parse words and numbers
    - We will use "~" to separate numbers from words
  - **`System.out`** to write out uncompressed text to the console

LZ77T.java: The compress() method

```
public void compress(String infile) throws IOException {
   // set up input and output
   mIn = new BufferedReader(new FileReader(infile));
   mOut = new PrintWriter(new BufferedWriter(
          new FileWriter(infile+".lz77")));

   int nextChar;
   String currentMatch = "";
   int matchIndex = 0, tempIndex = 0;

   // while there are more characters - read a character
   while ((nextChar = mIn.read()) != -1) {
     // look in our search buffer for a match
     tempIndex = mSearchBuffer.indexOf(currentMatch +
                                       (char)nextChar);
   // continued on next slide
```

```
     // if match then append nextChar to currentMatch
     // and update index of match
     if (tempIndex != -1) {
        currentMatch += (char)nextChar;
        matchIndex = tempIndex;
     } else {
        // found longest match, now should we encode it?
        String codedString =
          "~"+matchIndex + "~" + currentMatch.length() +
          "~" + (char)nextChar;
        String concat = currentMatch + (char)nextChar;
        // is coded string shorter than raw text?
        if (codedString.length() <= concat.length()) {
           mOut.print(codedString);
           // append to the search buffer
           mSearchBuffer.append(concat);
           currentMatch = ""; matchIndex = 0;
        } else {

           // continued from next slide
```

```
        // otherwise, output chars one at a time from
        // currentMatch until we find a new match or
        // run out of chars
        currentMatch = concat; matchIndex = -1;
        while (currentMatch.length() > 1 &&
               matchIndex == -1) {
          mOut.print(currentMatch.charAt(0));
          mSearchBuffer.append(currentMatch.charAt(0));
          currentMatch =
            currentMatch.substring(1,
                                   currentMatch.length());
          matchIndex =
            mSearchBuffer.indexOf(currentMatch);
        } // end inner while loop
      } // end if/else coded string shorter than raw text
      // Adjust search buffer size if necessary
      if (mSearchBuffer.length() > mBufferSize) {
        mSearchBuffer = mSearchBuffer.delete(0,
            mSearchBuffer.length() - mBufferSize);
      }
    } // end if/else found match for next char
  } // end while there are more characters to read
```

```
  // flush any match we may have had when EOF encountered
  if (matchIndex != -1) {
    // note that there is no following-symbol now,
    // nor is there any need to append a final "~"
    String codedString =
       "~" + matchIndex + "~" + currentMatch.length();
    if (codedString.length() <= currentMatch.length()) {
      mOut.print("~"+matchIndex+"~"+currentMatch.length());
    } else {
      mOut.print(currentMatch);
    }
  }
  // close files
  mIn.close();
  mOut.flush(); mOut.close();
} // end compress()
```

LZ77T.java: The unCompress() method

```
public void unCompress(String infile) throws IOException {
  Scanner st =
    new Scanner(new FileReader(infile+".lz77")).
      useDelimiter("~");

  int offset, length;

  . . .
```



```
  while (st.hasNext()) {
    if (st.hasNextInt()) {
      offset = st.nextInt();
      if (st.hasNextInt()) {
        // Then it's the length
        length = st.nextInt();
        // Also need to read the third part of the triple (following word)
        String following = st.next();
        // Output substring from search buffer
        String output = mSearchBuffer.substring(offset, offset+length);
        System.out.print(output+following);
        mSearchBuffer.append(output+following);
        // Adjust search buffer size if necessary
          trimSearchBuffer();
      } else {
        // the first number must have been part of a normal word
        mSearchBuffer.append(offset + st.next());
      }
    } else {
      String output = st.next();
      mSearchBuffer.append(output);
      System.out.print(output);
      // Adjust search buffer if necessary
      trimSearchBuffer();
    }
  }
  st.close();
```