CSE 1201
Object Oriented Programming

Streams and File I/O

Acknowledgement

- For preparing the slides I took materials from the following sources
  - Course Slides of Dr. Tagrul Dayar, Bilkent University
  - Java book "*Java Software Solutions*" by Lewis & Loftus.

I/O Overview

- *I/O* = Input/Output
- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - permanent copy
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

Streams

- *Stream*: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
- *Input stream*: a stream that provides input to a program
  - `System.in` is an input stream
- *Output stream*: a stream that accepts output from a program
  - `System.out` is an output stream
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `System.in` connects a program to the keyboard

Binary Versus Text Files

- *All* data and programs are ultimately just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit
  - *byte* is a group of eight bits
- *Text files*: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Java source files are text files
  - so is any file created with a "text editor"
- *Binary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - these files are easily read by the computer but not humans
  - they are *not* "printable" files
    - actually, you *can* print them, but they will be unintelligible
    - "printable" means "easily readable by humans when printed"

Java: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
- Java binary files are portable
  - they can be used by Java on different machines
  - Reading and writing binary files is normally done by a program
  - text files are used only to communicate with humans

Java Text Files

- Source files
- Occasionally input files
- Occasionally output files

Java Binary Files

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text File I/O

- Important classes for text file output (to the file)
  - `PrintWriter`
  - `FileOutputStream`   [or `FileWriter`]
- Important classes for text file input (from the file):
  - `BufferedReader`
  - `FileReader`
- `FileOutputStream` and `FileReader` take file names as arguments.
- `PrintWriter` and `BufferedReader` provide useful methods for easier writing and reading.
- Usually need a combination of two classes
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Buffering

- Not buffered: each byte is read/written from/to disk as soon as possible
  - "little" delay for each byte
  - A disk operation per byte---higher overhead
- Buffered: reading/writing in "chunks"
  - Some delay for some bytes
    - Assume 16-byte buffers
    - Reading: access the first 4 bytes, need to wait for all 16 bytes are read from disk to memory
    - Writing: save the first 4 bytes, need to wait for all 16 bytes before writing from memory to disk
  - A disk operation per a buffer of bytes---lower overhead

Every File Has Two Names

1. the stream name used by Java
   - `outputStream` in the example
2. the name used by the operating system
   - `out.txt` in the example

Text File Output

- To open a text file for output: connect a text file to a stream for writing

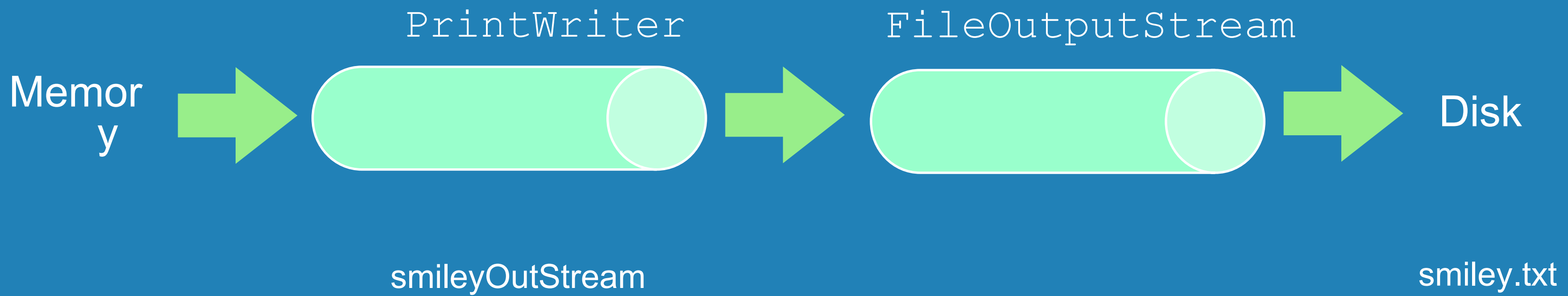
```
PrintWriter outputStream =
   new PrintWriter(new FileOutputStream("out.txt"));
```

- Similar to the long way:

```
FileOutputStream s = new FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- Goal: create a `PrintWriter` object
  - which uses `FileOutputStream` to open a text file
- `FileOutputStream` "connects" `PrintWriter` to a text file.

Output File Streams



```
PrintWriter smileyOutStream = new PrintWriter( new FileOutputStream(“smiley.txt”) );
```

Methods for `PrintWriter`

- Similar to methods for `System.out`
- `println`

```
outputStream.println(count + " " + line);
```

- `print`
- `format`
- `flush`: write buffered output to disk
- `close`: close the `PrintWriter` stream (and file)

TextFileOutputDemo Part1

```
public static void main(String[] args)
{
    PrintWriter outputStream = null;
    try
    {
        outputStream =
              new PrintWriter(new
   FileOutputStream("out.txt"));
    }
    catch(FileNotFoundException e)
    {
        System.out.println("Error opening the file out.txt. "
                                  + e.getMessage());
        System.exit(0);
    }
```

A try-block is a block: outputStream would not be accessible to the rest of the method if it were declared inside the try-block

Opening the file

Creating a file can cause the FileNotFound-Exception if the new file cannot be made.

TextFileOutputDemo Part 2

```
System.out.println("Enter three lines of text:");
String line = null;
int count;
   for (count = 1; count <= 3; count++)
   {
       line = keyboard.nextLine();
       outputStream.println(count + " " + line);
   }
   outputStream.close();
   System.out.println("... written to out.txt.");
}
```

Writing to the file

Closing the file

The `println` method is used with two different streams: `outputStream` and `System.out`

Overwriting a File

- Opening an output file creates an empty file
- Opening an output file creates a new file if it does not already exist
- Opening an output file that already exists eliminates the old file and creates a new, empty one
  - data in the original file is lost

Appending to a Text File

- To add/append to a file instead of replacing it, use a different constructor for `FileOutputStream`:

```
outputStream =
   new PrintWriter(new FileOutputStream("out.txt", true));
```

- Second parameter: append to the end of the file if it exists?
- Sample code for letting user tell whether to replace or append:

```
System.out.println("A for append or N for new file:");
        char ans = keyboard.next().charAt(0);
     boolean append = (ans == 'A' || ans == 'a');
           outputStream = new PrintWriter(
        new FileOutputStream("out.txt", append));
```

true if user enters 'A'

Closing a File

- An output file should be closed when you are done writing to it (and an input file should be closed when you are done reading from it).
- Use the `close` method of the class `PrintWriter` (`BufferedReader` also has a `close` method).
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open.

Why Bother to Close a File?

If a program automatically closes files when it ends normally, why close them with explicit calls to `close`?

Two reasons:

1. To make sure it is closed if a program ends abnormally (it could get damaged if it is left open).

2. A file opened for writing must be closed before it can be opened for reading.
   - Although Java does have a class that opens a file for both reading and writing, it is not used in this text.

Text File Input

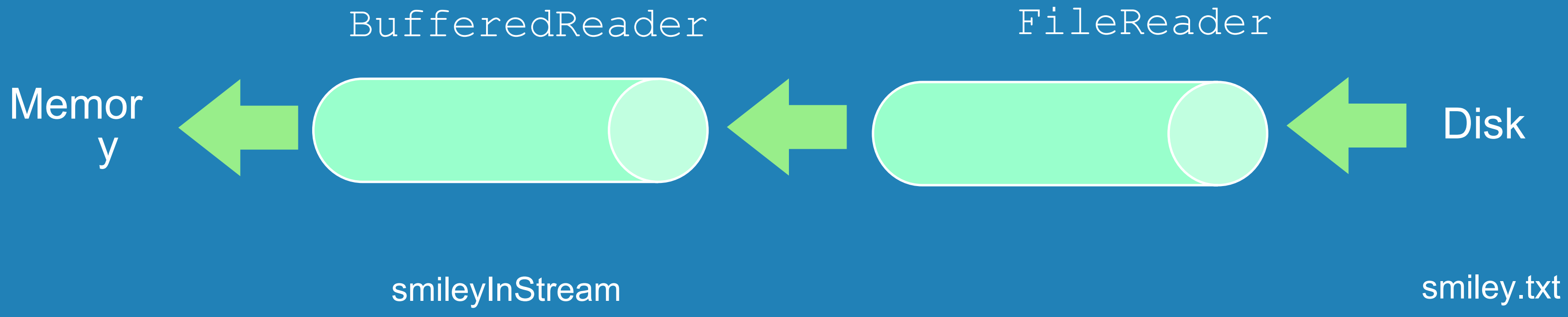
- To open a text file for input: connect a text file to a stream for reading
  - Goal: a `BufferedReader` object,
    - which uses `FileReader` to open a text file
  - `FileReader` "connects" `BufferedReader` to the text file
- For example:

```
BufferedReader smileyInStream =
    new BufferedReader(new FileReader("smiley.txt"));
```

- Similarly, the long way :

```
FileReader s = new FileReader("smiley.txt");
BufferedReader smileyInStream = new
    BufferedReader(s);
```

Input File Streams



```
BufferedReader smileyInStream = new BufferedReader( new FileReader(“smiley.txt”)
);
```

Methods for `BufferedReader`

- `readLine`: read a line into a `String`
- no methods to read numbers directly, so read numbers as `Strings` and then convert them (`StringTokenizer` later)
- `read`: read a `char` at a time
- `close`: close `BufferedReader` stream

Exception Handling with File I/O

Catching IOExceptions

- `IOException` is a predefined class
- File I/O might throw an `IOException`
- catch the exception in a catch block that at least prints an error message and ends the program
- `FileNotFoundException` is derived from `IOException`
  - therefore any catch block that catches `IOExceptions` also catches `FileNotFoundExceptions`
  - put the more specific one first (the derived one) so it catches specifically file-not-found exceptions
  - then you will know that an I/O error is something other than file-not-found

Example: Reading a File Name from the Keyboard

```
public static void main(String[] args)
  {
     String fileName = null;  // outside try block, can be used in catch
     try
     { Scanner keyboard = new Scanner(System.in);
       System.out.println("Enter file name:");
       fileName = keyboard.next();
       BufferedReader inputStream =
          new BufferedReader(new FileReader(fileName));
       String line = null;
       line = inputStream.readLine();
       System.out.println("The first line in " + filename + " is:");
       System.out.println(line);
       // . . . code for reading second line not shown here . . .
       inputStream.close();
     }
     catch(FileNotFoundException e)
     {
       System.out.println("File " + filename + " not found.");
     }
     catch(IOException e)
     {
       System.out.println("Error reading from file " + fileName);
     }
  }
```

reading a file name from the keyboard

using the file name read from the keyboard

reading data from the file

closing the file

Reading Words in a String: Using `StringTokenizer` Class

- There are `BufferedReader` methods to read a line and a character, but not just a single word

- `StringTokenizer` can be used to parse a line into words
  - import `java.util.*`
  - some of its useful methods are shown in the text
    - e.g. test if there are more tokens
  - you can specify *delimiters* (the character or characters that separate words)
    - the default delimiters are "white space" (space, tab, and newline)

Example: `StringTokenizer`

- **Display the words separated by any of the following characters: space, new line (\n), period (.) or comma (,).**

```
String inputLine = keyboard.nextLine();
StringTokenizer wordFinder =
    new StringTokenizer(inputLine, " \n.,");
//the second argument is a string of the 4 delimiters
while(wordFinder.hasMoreTokens())
{
    System.out.println(wordFinder.nextToken());
}
```

Entering "`Question,2b.or !tooBee.`" gives this output:

```
Question
2b
or
!tooBee
```

Testing for End of File in a Text File

- **When `readLine` tries to read beyond the end of a text file it returns the special value *`null`***
  - **so you can test for `null` to stop processing a text file**

- **`read` returns -1 when it tries to read beyond the end of a text file**
  - **the `int` value of all ordinary characters is nonnegative**

- **Neither of these two methods (`read` and `readLine`) will throw an `EOFException`.**

Example: Using Null to Test for End-of-File in a Text File

When using **`readLine`** test for `null`

Excerpt from `TextEOFDemo`

```
int count = 0;
String line = inputStream.readLine();
while (line != null)
{
   count++;
   outputStream.println(count + " " + line);
   line = inputStream.readLine();
}
```

When using **`read`** test for -1

Using Path Names

- *Path name*—gives name of file and tells which directory the file is in
- *Relative path name*—gives the path starting with the directory that the program is in
- Typical UNIX path name:

```
/user/smith/home.work/java/FileClassDemo.java
```

- Typical Windows path name:

```
D:\Work\Java\Programs\FileClassDemo.java
```

- When a backslash is used in a quoted string it must be written as two backslashes since backslash is the escape character:

```
"D:\\Work\\Java\\Programs\\FileClassDemo.java"
```

- Java will accept path names in UNIX or Windows format, regardless of which operating system it is actually running on.

`File` Class [java.io]

- Acts like a wrapper class for file names
- A file name like "`numbers.txt`" has only `String` properties
- `File` has some very useful methods
  - `exists`: tests if a file already exists
  - `canRead`: tests if the OS will let you read a file
  - `canWrite`: tests if the OS will let you write to a file
  - `delete`: deletes the file, returns true if successful
  - `length`: returns the number of bytes in the file
  - `getName`: returns file name, excluding the preceding path
  - `getPath`: returns the path name—the full name

```
File numFile = new File("numbers.txt");
if (numFile.exists())
  System.out.println(numfile.length());
```

`File` Objects and Filenames

- `FileInputStream` and `FileOutputStream` have constructors that take a `File` argument as well as constructors that take a `String` argument

```
PrintWriter smileyOutStream = new PrintWriter(new
   FileOutputStream("smiley.txt"));

File smileyFile = new File("smiley.txt");
if (smileyFile.canWrite())
   PrintWriter smileyOutStream = new PrintWriter(new
   FileOutputStream(smileyFile));
```

Alternative with Scanner

- Instead of `BufferedReader` with `FileReader`, then `StringTokenizer`
- Use `Scanner` with `File`:

```
Scanner inFile =
    new Scanner(new File("in.txt"));
```

- Similar to `Scanner` with `System.in`:

```
Scanner keyboard =
    new Scanner(System.in);
```

Reading in `int`’s

```
Scanner inFile = new Scanner(new File(“in.txt"));
int number;
while (inFile.hasInt())
   {
      number = inFile.nextInt();
      // …
   }
```

Reading in lines of characters

```
Scanner inFile = new Scanner(new File(“in.txt"));
String line;
while (inFile.hasNextLine())
  {
     line = inFile.nextLine();
     // …
  }
```

Multiple types on one line

```
// Name, id, balance
Scanner inFile = new Scanner(new File("in.txt"));
while (inFile.hasNext())
  {
    name = inFile.next();
    id = inFile.nextInt();
    balance = inFile.nextFloat();
    // …  new Account(name, id, balance);
  }
--------------------
String line;
while (inFile.hasNextLine())
  {
    line = inFile.nextLine();
    Scanner parseLine = new Scanner(line) // Scanner again!
    name = parseLine.next();
    id = parseLine.nextInt();
    balance = parseLine.nextFloat();
    // …  new Account(name, id, balance);
  }
```

Multiple types on one line

```
// Name, id, balance
Scanner inFile = new Scanner(new File(“in.txt"));
String line;
while (inFile.hasNextLine())
  {
    line = inFile.nextLine();
    Account account = new Account(line);
  }
--------------------
public Account(String line) // constructor
{
  Scanner accountLine = new Scanner(line);
  _name = accountLine.next();
  _id = accountLine.nextInt();
  _balance = accountLine.nextFloat();
}
```

`BufferedReader` VS `Scanner` (parsing primitive types)

- **`Scanner`**
  - **`nextInt()`, `nextFloat()`, … for parsing types**
- **`BufferedReader`**
  - **`read()`, `readLine()`, … none for parsing types**
  - **needs `StringTokenizer` then wrapper class methods like `Integer.parseInt(token)`**

`BufferedReader` VS `Scanner` (Checking End of File/Stream (EOF))

- **`BufferedReader`**
  - **`readLine()` returns `null`**
  - **`read()` returns -1**
- **`Scanner`**
  - **`nextLine()` throws exception**
  - **needs `hasNextLine()` to check first**
  - **`nextInt()`, `hasNextInt()`, …**

```
BufferedReader inFile = …
line = inFile.readline();
while (line != null)
{
   // …
   line = inFile.readline();
}

-------------------

Scanner inFile = …
while (inFile.hasNextLine())
{
   line = infile.nextLine();
   // …
}
```

```
BufferedReader inFile = …
line = inFile.readline();
while (line != null)
{
  // …
  line = inFile.readline();
}

-------------------

BufferedReader inFile = …
while ((line = inFile.readline()) != null)
{
  // …
}
```

My suggestion

- Use `Scanner` with `File`
  - `new Scanner(new File("in.txt"))`
- Use `hasNext…()` to check for EOF
  - `while (inFile.hasNext…())`
- Use `next…()` to read
  - `inFile.next…()`
- Simpler and you are familiar with methods for `Scanner`

My suggestion cont...

- File input
  - `Scanner inFile =`

    `new Scanner(new File(“in.txt”));`
- File output
  - `PrintWriter outFile =`

    `new PrintWriter(new File(“out.txt”));`
  - `outFile.print(), println(), format(), flush(), close(), …`