**Administrative**

**Today’s session**

Data stream

Text file

File specification

File class

**Session Topics**

**Data stream**

● A **data stream** is a data connection between an application and the outside world.

● An **input data stream** is data entering the application.

● An **output data stream** is data leaving the application.

● The keyboard is connected to an application via input stream **System.in**.

● The screen is connected to an application via output stream **System.out**.

● Data can be streamed to and from a file.

● There are two types of files an application can work with:

✓ A **text file** consists of a sequence of (mostly) human-readable characters. A text file could be thought of as a specialized binary file.

✓ A **binary file** consists of a sequence of binary data that can represent an image, a sound clip, a video, etc. A binary file tends to be more compact than a text file.

● Corresponding to the two types of files are two types of streams:

✓ A **character stream** reads or writes one character at a time.

✓ Classes ending in “Reader” or “Writer” are subclasses of abstract classes **Reader**and **Writer** which read and write characters, respectively:

Reader

InputStreamReader

FileReader

BufferedReader

Writer

OutputStreamWriter

FileWriter

BufferedWriter

PrintWriter

✓ A **byte stream** reads or writes one byte at a time.

✓ Classes ending in “Stream” are subclasses of abstract classes **java.io.InputStream** and **java.io.OutputStream** which read and write bytes, respectively.

InputStream

FileInputStream

FilterInputStream

BufferedInputStream

OutputStream

FileOutputStream

FilterOutputStream

BufferedOutputStream

● Buffering speeds up the reading and writing of larger files.

● The **default buffer size** is 8,092 bytes.

● Classes **java.io.BufferedReader** and **java.io.BufferedWriter** may be used to read and write text files, respectively.

● Classes **java.io.BufferedInputStream** and **java.io. BufferedOutputStream** classes may be used to read and write binary files, respectively.

**Text file**

● Notepad++ or another text editor may be used to:

✓ Create/edit an input file.

✓ View an output file.

● The application specifies the file name and optionally a path to the file.

● If just the file name is specified in Eclipse, the path defaults to the application’s project folder.

● If the file doesn’t exist, an error occurs.

● File name specification examples on Windows:

**No path: "USStateCapitals.txt"**

**With relative path: "Data\\USStateCapitals.txt"**

**With absolute path: "C:\\Geography\\Data\\USStateCapitals.txt"**

● File name specification examples on UNIX:

**No path: "USStateCapitals.txt"**

**With relative path: "Data/USStateCapitals.txt"**

**With absolute path: "/Geography/Data/USStateCapitals.txt"**

● File name specification examples on any operating system:

**No path: "USStateCapitals.txt"**

**With relative path: "Data" + File.separator + "USStateCapitals.txt"**

**With absolute path: "C:" + File.separator + "Geography" + File.separator + "Data" + File.separator + "USStateCapitals.txt"**

**Text file input**

● The same **Scanner** class used to get input from the keyboard may also be used to get input from a file.

● Once connected to the file using the Scanner class, familiar methods may be used to read data from the file including:

| Method | Description when used for file input |
| --- | --- |
| nextInt()  nextDouble()  next() | Reads the next token from the buffer. It will scan past any whitespace characters until it reads a token. It then converts the token to an **int/double/string**. Any characters after the token, including the new-line (\n) character, remain in the buffer. |
| nextLine() | Reads the next line from the buffer. It will NOT scan past any whitespace characters to begin reading the line. It then converts all characters read, except the new-line (\n) character, to a string. It then discards the new-line (\n) character from the buffer. |
| hasNextInt()  hasNextDouble()  hasNext()  hasNextLine() | Tests whether the input file has the specified token and returns true or false. |

● Unlike keyboard input, no prompt is needed for file input.

● File-reading requires:

✓ An input file variable:

**Scanner fileIn = null;**

✓ A try-catch block for opening and reading from the file:

**try**

**{**

**fileIn = new Scanner(new FileInputStream("DataIn.txt"));**

**while (fileIn.hasNextLine())**

**{**

**line = fileIn.nextLine();**

**…**

**}**

**fileIn.close();**

**catch (FileNotFoundException e)**

**{**

**System.out.println("Error: file 'DataIn.txt' not found.");**

**System.out.println("Error message:\n" + e.getMessage());**

**}**

● See **Text file input and output** sample application on Blackboard.

**Text file output**

● The **PrintWriter** class may be used to write output to a file.

● Once connected to a file using the PrintWriter class, familiar methods may be used to write data to the file including print, println, and printf.

● File-writing requires:

✓ An output file variable:

**PrintWriter fileOut = null;**

✓ A try-catch block for creating/opening and writing to the file:

**try**

**{**

**fileOut = new PrintWriter(new FileOutputStream("DataOut.txt", appendFlag));**

**fileOut.println("Data to file");**

**fileOut.close();**

**}**

**catch (FileNotFoundException e)**

**{**

**System.out.println("Error: file 'DataOut.txt' cannot be created or opened.");**

**System.out.println("Error message: " + e.getMessage());**

**}**

● See **Text file input and output** sample application on Blackboard.

● See **Text file output comparison** sample application on Blackboard.

**File specification**

● Data in a text file may be arranged in any fashion – one value per line or multiple values per line.

● Each datum is typically separated from the next with whitespace characters including:

✓ The space character.

✓ The tab character.

✓ The new-line (\n) character.

● A text file often organizes data into rows and columns with these two parts:

✓ Header – an optional row at the top of the file that contains column names.

✓ Detail – one or more rows of data split into columns.

● A **file specification** tells us how a text file is structured so that it may properly be read and/or written.

● A file specification includes at least this information for each column:

✓ Name

✓ Data type: str, int, float, bool

✓ Starting column

✓ Ending column

● File specification example:

**File data (with header)**

**1 2 3 4 5 6 7**

**12345678901234567890123456789012345678901234567890123456789012345678901234567**

**Manufacturer Model Model Year Satisfaction Level**

**Ford Mustang 2014 .56**

**Chevrolet Tahoe 2013 .36**

**GMC Acadia 2010 .44**

**File specification**

**Field Type Start-End**

**Manufacturer string 1-18**

**Model string 19-36**

**Model Year integer 37-46**

**Satisfaction Level real 47-69**

**This file contains a header row.**

● A file specification is written *independent* of any programming language.

● See **Text file input and output with file specification** sample application on Blackboard.

**File class**

● The **File class** enables access to a computer’s folders and files.

● The File class contains methods for:

✓ Retrieving information about folders and files.

✓ Making changes to folders and files including:

| Method | Purpose |
| --- | --- |
| createNewFile () | Create a file. |
| mkdir() | Create a folder. |
| delete() | Delete a folder or file. |

● See **File class** sample application on Blackboard.