## Files (JFS, chap 17)

- In this section:
  - Motivation and basic file concepts
  - Reading files
  - Writing files
  - Complications and other files issues





## Motivation

- [See JFS 17 ('Introduction', 'File Access: Stream or Random?', 'The Essentials of Streams', 'The Java I/O Classes')]
- An ordinary program can change only data in memory
  - When a program exits, its data is lost
- Many programs therefore need to use data in **files**
  - Stored longer term on a "disk" of some sort
- Files could also be used for bulk data storage while a program is running
- Files can contain:
  - Pure text (e.g. Java source code)
  - Or binary data (e.g. a compiled Java program or an image)

## Basic concepts

- Most programs read data items one after another:
  - **Sequential access**
  - Also writing
- Special programs read and write data items in arbitrary order
  - **Random access**
- Java supports all these kinds of files and many ways of using them – a *bewildering variety* of library classes
- We will only consider
  - *Sequential* (stream) use of *text files*
  - (This means that we can also use a plain text editor to inspect/build the data files)
- JFS makes a different choice of file classes from these notes





- File processing is usually in three stages:
  - The file is *opened* (made available for reading or writing)
  - Then read or written
  - Finally it is *closed*
- A file has a *name* that is used when it is **opened**
- A file name can be:
  - Basic: like "data" or "Hello.java"
  - Or may include a **path**: like "C:\\database\\data", where "C:\\database" is the file system path to the file
- An open file may be **read** (data got from it) or **written** (data sent to it).

- A file is opened for reading with one of the ***Reader*** classes
- A file is opened for writing with one of the ***Writer*** classes
- For efficiency, file access is usually **buffered**
  - This means that data is read from the file or written to the file in large chunks
  - This avoids accessing the disk for every single character
- After the program is finished with the file, it should **close** it
  - Closing a file that has been written is important, because it is at this point that any left-over data in buffers is finally placed in the file
  - Also the description entry in the folder is finalized
  - And the program/JVM can release any resources





## Reading Files

- A file to be read can be opened with:

```
String fileName = "myfile.txt";
FileReader inFile;     // Declare file reader
inFile = new FileReader(fileName);
                       // Open input file
```

- The name could, of course, be given as a literal string to `FileReader`
- The `FileReader` object holds information for managing the opened file

- Now that the file is open it can be read, but essentially just as characters:

```
char ch;       // Declare one character
char text[] = new char[1000];
               // Declare array of characters
ch = inFile.read();  // Read one character
inFile.read(text);   // Read characters into array
```

- Note that each read action continues from where the previous one stopped
  - So repeated reads work sequentially through the whole file





- Many programs will wish to read text files *line-by-line*
- For this purpose, the raw file is combined with the facility to buffer data as it is read
- This changes the way the file is opened:

```
String fileName = "myfile.txt";
BufferedReader inFile; // Declare buffered reader
// Open input file and arrange to buffer it
inFile = new BufferedReader(
                          new FileReader(fileName));
```

- Opening the file actually requires two steps:
  - Open the file with `FileReader`
  - Create buffering for it with `BufferedReader`, allowing chunks to be read efficiently from the disk

- A file like this can be read *line-by-line* using:

```
String line;    // Storage for line of text
while ( (line = inFile.readLine()) != null )
                    // Read a line from file &
                    // Check if not end of file
  ...  // do something with line
```

- The while loop above does something a little strange but quite valid (and common):
  - It assigns a new value to `line` with:

```
line = inFile.readLine() // Read a line from file
```

  - It then uses this value and compares it with `null`, as if the program said:

```
line != null        // check if not end of file
```





- Note that the end-of-line marker is *removed* by `readLine`
- When there is no more data to read from the file:
  - `readLine` returns the value `null` (meaning an invalid string)
- If the result is not `null`, the line contains text that can be processed
- A blank line will be read as an *empty string* `""`, not as `null`
- The effect is therefore to read and process lines from the file until they have all been read
- Once the program has finished with the file, it is closed with:

```
inFile.close();      // Close input file
```

- Example algorithm: Report the length of each line of a file on the terminal:

```
String line;          // Storage for line of text
int lineNumber = 0;   // For counting lines
while ( (line = inFile.readLine()) != null ) {
                      // Read a line from file &
                      // Check if not end of file
  lineNumber++;       // Count the line
  int length = line.length();
  System.out.println(""+lineNumber+": "+length);
}
```





## Writing Files

- A file to be written can be opened with:

```
String fileName = "myfile.txt";
FileWriter outFile;  // Declare file writer
outFile = new FileWriter(fileName);
                     // Open output file
```

- The name could, of course, be given as a literal string to `FileWriter`
- Opening a file in this way *replaces* any existing version
  - A new *empty* file is created
  - An existing file might not actually be deleted until the new version is *closed* properly
- To *append* to an existing file use:

```
new FileWriter(fileName, true)
```

- Now that the file is open it can be written, but essentially just as characters:

```
char ch = 'a';   // Declare one character
char text[] = "word";
                 // Declare array of characters
outFile.write(ch);   // Write one character
outFile.write(text); // Write array of characters
```

- Note that each write add more characters *to the end* of the file written so far
  - Sequential writing





- Many programs will wish to write text files line-by-line
- For this purpose, the raw file is combined with the facility to buffer data as it is written.
- This changes the way the file is opened:

```
String fileName = "myfile.txt";
BufferedWriter outFile; // Declare buffered writer
// Open output file and arrange to buffer it
outFile = new BufferedWriter(
                        new FileWriter(fileName));
```

- This two-step procedure is much as for reading
- A file like this can be written using:

```
String line = "Now is the time";
outFile.write(line);  // Write string to file
```

- Or, of course, supplying the string directly to write

- There really ought to be a `writeLine` method
  - But sadly this does not exist
- Instead there are two options:
  - Include end-of-line (`"\r\n"` in Windows) at the end of the string to be written:

```
outFile.write("Now is the time\r\n");
```

  - Or call the `newLine` method to append a new line:

```
outFile.write("Now is the time");
outFile.newLine();  // Append a newline
```

- The second of these is preferable
  - It uses the correct form on each OS
- Once the program has finished with the file, it is closed in the same way as an input file:

```
outFile.close();
```





## Complications with Files

- The classes needed to handle files are imported with:

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

- File operations can give rise to *many error conditions* such as:
  - Trying to read a file that does not exist
  - Trying to write a file when the application does not have permission
  - Disk input-output errors
- For this reason, file operations may generate an exception (`IOException`) that should be *caught* to recover cleanly.

- Usually all file operations are carried out inside a `try...catch` pair:

```
try {      // Try file operations
  ...      // Do things with files
}
catch (IOException exception) {
                // catch input-output error
  ...    // handle file error
}
```

- The *compiler* checks that we have `IOException` handlers





## Terminal input/output streams

- [JFS Chap 17 ('Console I/O', 'The System Class', 'Reading from A Remote Site')]
- Three streams are automatically opened:
  - `System.in` is the standard input of the program – normally what the user types at the keyboard
  - `System.out` is the standard output of the program – normally what is sent to a console window
  - `System.err` is the standard error of the program – normally error messages sent to a console window
- This is principally important if we are running our Java application in a command prompt window

- Like other files, they can be used directly:

```
char ch;
ch = System.in.read(); // Read character from user
System.out.println("Insert disk");
                              // Send message to user
System.err.println("File failure");
                              // Report error to user
```

- These kinds of file include methods `print` and `println` (not `printLine`!) for formatted output of the built-in types
- In more advanced programming:
  - Internet connections (e.g. to a remote computer or to a URL) can also be treated as files





- Although it is not a file operation, the `System` class also contains the following method that is sometimes useful:

```
int code = 0;       // Set and declare exit code
System.exit(code); // Exit appl with code
```

- Conventionally, the normal exit code is 0, meaning the program finished normally.
- Codes such as 1, 2, etc. can be used for abnormal error conditions.

## File Example

- Adapting the earlier algorithm: Report the length of each line of a file as lines of text in a new file:

```
String line;          // Storage for line of text
int lineNumber = 0;   // For counting lines
outFile.write("Line length report:"); // Heading
outFile.newline();
while ( (line = inFile.readLine()) != null ) {
                    // Read lines from file
  lineNumber++;     // Count the line
  int length = line.length(); // Calculate length
  outFile.write(""+lineNumber+": "+length);
                                // Report length
  outFile.newline();
}
outFile.write("End of report:"); // Footer
outFile.newline();
```





## End of section