File Processing in Java

What is File I/O?

I/O is an abbreviation for input and output.

Input is data coming in at runtime. Input come sin through a mouse, keyboard, touchscreen, microphone and so on.

Output is data coming out at runtime. It goes through the screen, printer, speakers and so on.

Some I/O is about files - writing to and reading from files.

Unix had the idea that 'everything is a file'.

Some files are stored on a hard drive. But they might also be on a USB memory stick, or on a LAN file server, or on remote server over the Internet, and so on.

Unix *abstracts* the idea of a file. That means it treats all files the same in terms of *what they do*, not *how they work*. A file on a remote server is very different from a local file on hard disk. We probably do not know how the remote file works (is it on hard disk or USB or DVD or what?), but for us it does not matter. A file is anything that

You can *read* data from

You can *write* data to

Maybe not both. Some files are read-only. Usually you can create and delete files - if you have the correct permissions.

Often files are on *non-volatile* mediums. This means that when you switch off power, the data is not lost.

What all files have in common is *streams of data* - in or out. A difference between these streams and real streams is that they can be turned on and off.

Operating systems and file systems

Operating systems (OS, Windows or Linux or whatever) contain code that can read or write files. That code is used by applications. In other words applications do not access files directly. They call the OS routines, through the API, to access files. Otherwise every application would need duplicate file access code.

An OS uses one or more *file systems*.

Windows has FAT and NTFS file systems. Linux has ext2, ext3, ext4 and others.

A file system has

- the logical file system. This has ideas like file names and extensions, file attributes (like read-only or hidden), drives, folders, and opening reading writing and closing files
- the device specific level. The device specific level matches the electronic details of how the file storage device works. This is through *device drivers*, normally supplied by the device manufacturer.

File descriptors

The OS creates a *file handle* through which I/O takes place (a file descriptor on Unix). A file handle uses some memory, and the number of file handles which can be used at the same time is limited. If our program opens files and never closes any, we will hit that limit - and get an exception. A file handle opened and not closed is one type of *resource leak*. So the pattern should always be

- open a file

- use it
- close it

Standard streams

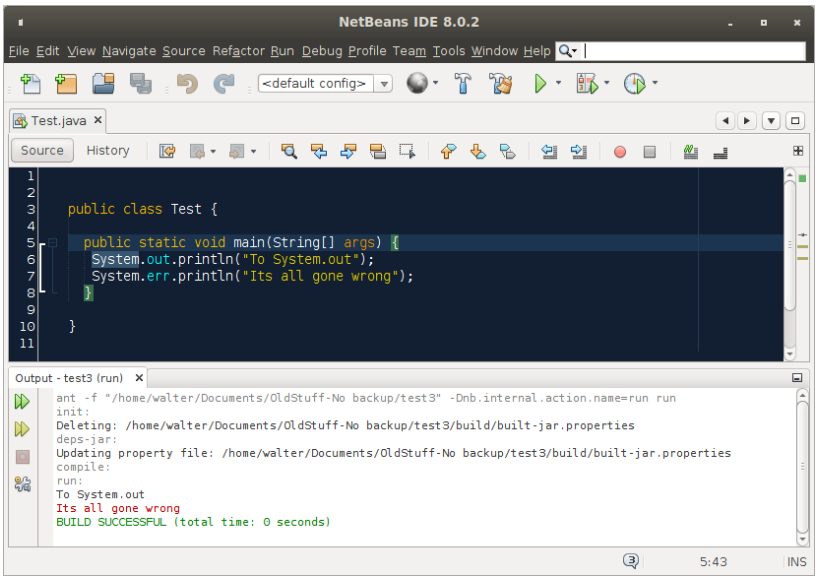
For 'normal' files we need to open them before use, when the OS creates a file descriptor for it. But staing with Unix, OS opened 3 on system startup:

A standard input, connected to the keyboard, called stdin in C, System.in in Java

Standard output, connected to the screen (originally a teletype printer), called stdout in C, and System.out in Java

Standard error, where error messages would be sent. stderr in C, System.err in Java.

When a Java application is run in Netbeans, stdout goes to the output window, and stderr does the same, in red:



Basic text output

We start by simply writing some characters into a text file:

```
public static void main(String[] args) {

  try {

    FileOutputStream os = new FileOutputStream("data.txt");
    OutputStreamWriter osw = new OutputStreamWriter(os);
    osw.write('A');
    osw.write('B');
    osw.write('C');
    osw.close();
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

The FileOutputStream class is the basic class for writing bytes to a file.

An OutputStreamWriter specialises this to writing characters. A common pattern in Java is to make an instance:

```
    FileOutputStream os = new FileOutputStream("data.txt");
```

and then to wrap it inside another class:

```
        OutputStreamWriter osw = new OutputStreamWriter(os);
```
This can be written in one step:

```
      OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("data.txt"));
```
but this looks more confusing, and will compile to very similar bytecode.

Then we write three characters into the file, then close it.

In this example the program ends after using the file, and it would be closed at that point anyway, so here it makes no difference. But it is a good habit to get into, to always close a file after use.

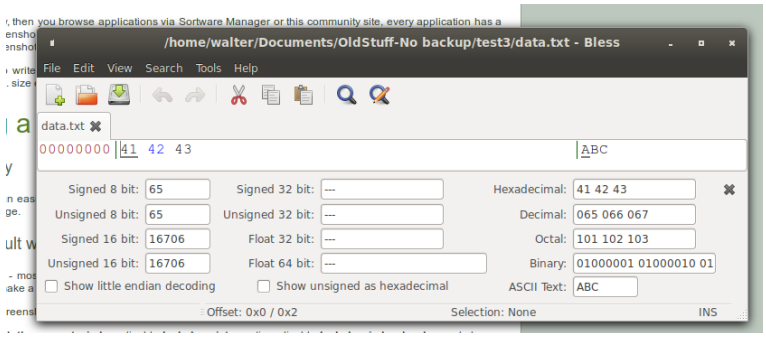
This takes place in a try.. catch construct. File operations may fail - for example here we are creating a new file, and we may not be allowed to write into the current folder. So in Java we must deal with the possible exception.

What does this program actually do? It

1. creates a new file, named data.txt or opens an existing one. Where is that file? It depends - running this as a Netbeans project, it appears in the project directory.

2. then it writes characters A B C into it.

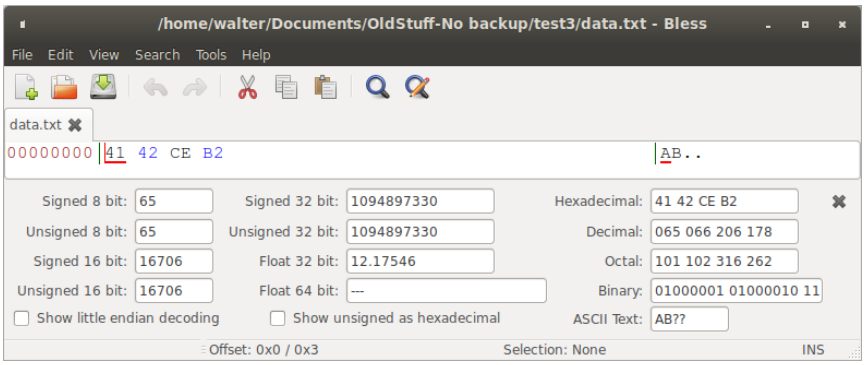
So what is in the file now? The best way is to look at the file using a *hex editor,* a utility which enables the user to look at what is in a file byte by byte. WinHex is a Windows version, and Bless is a Linux hex editor:



So data.txt contains 41 42 43. That is, 3 bytes, in hex, which in decimal are 65 66 and 67

Suppose we change this:

```
try {
     FileOutputStream os = new FileOutputStream("data.txt");
     OutputStreamWriter osw = new OutputStreamWriter(os);
     osw.write('A');
     osw.write('B');
     osw.write('β');
     osw.close();
   } catch (Exception ex) {
     ex.printStackTrace();
   }
 }
```
We get



Now we have four bytes. Why?

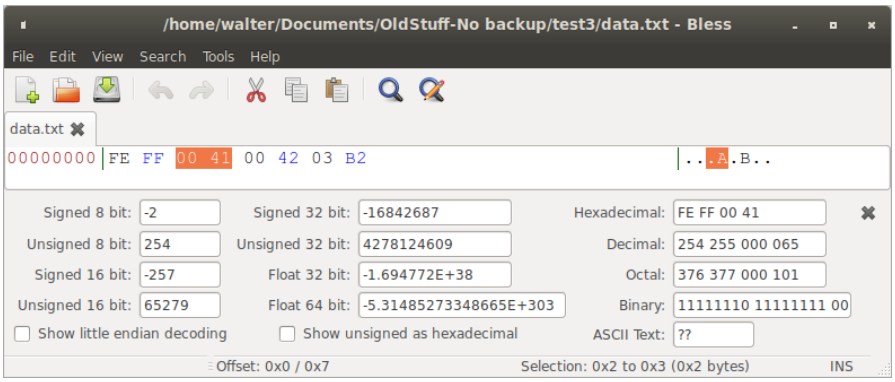
Java uses *Unicode* as its *character set*. This is a very large set of characters, each having its own unique integer, or *code point*. The code point of 'A' for example is 65. An *encoding* is a way of representing the code point in binary. For old extended ASCII character set this was a non-issue; there were 256 characters, so a code point could be stored in one byte. Unicode code points can be larger than 256, so will not always fit into one byte.

The default encoding in Java is *UTF-8*, which is what 41 42 CE B2 is. In UTF-8 some characters are in 1 byte, others 2 bytes. The 41 and 42 in hex is A and B. The β is the CE B2, in UTF-8.

We can use other encodings:

```
public static void main(String[] args) {
  try {
    FileOutputStream os = new FileOutputStream("data.txt");
    OutputStreamWriter osw = new OutputStreamWriter(os, "UTF-16");
    osw.write('A');
    osw.write('B');
    osw.write('β');
    osw.close();
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

Then in the file we have



In UTF-16, all characters are encoded as two bytes. So A is 00 41. The initial FE FF is a *byte order mark* - meaning the the most significant byte is first (otherwise A would be 41 00). The 03 B2 is the β.

When we *read* a text file in Java, we must guess what the encoding is - maybe UTF-8, maybe not - there is no way to tell. A file is just a stream of bytes. If we get it wrong we get garbage - then we know.

Buffered I/O

An output buffer is a section of memory used to hold data before it is output.

This is to make the process faster. An I/O operation takes time. If we have 1000 bytes to output, we could do that 1 byte at a time, in 1000 operations. Or we could fill a buffer with that 1000 bytes and output it in one operation, which is much faster.

Similarly for input buffers.

The Java class BufferedWriter does this. The second parameter to the constructor is how big the buffer should be. Here we write 1000000 characters to the file, through a 100k buffer, and measure the time taken:

```
  public static void main(String[] args) {
    try {
      long start = System.nanoTime();
      FileOutputStream fstream = new FileOutputStream("data.txt");
      BufferedWriter br = new BufferedWriter(new OutputStreamWriter(fstream), 100000);
      for (int count = 0; count < 1000000; count++) {
        br.write(99);
      }
      System.out.println((System.nanoTime() - start)/1000000000.0);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

A smaller buffer takes longer time.

Text file format

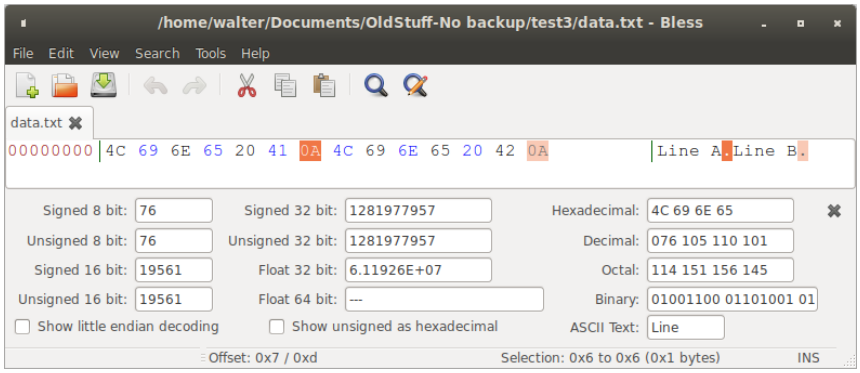
The *format* of a file is how the data in it is arranged - what data goes where.

Text files are made of characters, which humans use for reading and writing, which we do in 'lines'. So a basic format is to have the file made out of a sequence of lines.

For example:

```
try {
      FileOutputStream fstream = new FileOutputStream("data.txt");
      BufferedWriter br = new BufferedWriter(new OutputStreamWriter(fstream));
      br.write("Line A"); br.newLine();
      br.write("Line B");br.newLine();
      br.close();

    } catch (Exception ex) {
      ex.printStackTrace();
    }
```

In the file we get:



So the br.newLine() wrote hex 0A into the file. Different platforms represent a 'new line' in different ways. Some (Linux) uses hex code 0A, which is old ASCII 'line feed', and in Java is written \n. On Windows it is two characters - CR LF hex 0D 0A or in Java \r\n.

br.newLine writes the correct line separator for the platform it is on (so if the file is then transferred to a different platform, it may not work properly.

We can read a file like this:

```
 try {
      FileInputStream fstream = new FileInputStream("data.txt");
      BufferedReader br = new BufferedReader(new InputStreamReader(fstream));
      String str;
     while ((str=br.readLine())!=null)
       System.out.println(str);
      br.close();

    } catch (Exception ex) {
      ex.printStackTrace();
    }
```

We use Input not Output, and Reader not Writer - the Java classes are in pairs for input and output.

readLine reads a line of text from the file, from the current position, up to the next \r or \n or both. It returns null at the end of the file. So..

```
str=br.readLine()
```

reads a line and assigns it to the String str

```
str=br.readLine())!=null
```

checks it is not null (so not end of file). So

```
while ((str=br.readLine())!=null)
```

reads lines, until we hit the end of file.

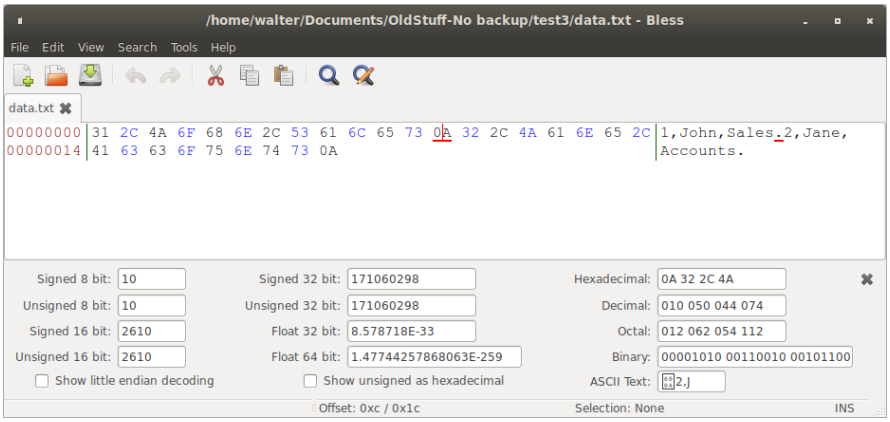
Data is often structured into *records* containing *fields*. For example data about employees would have one record for each employee, with fields of payroll number, name and department number. In Java this would be a class:

```
class Employee
{
  String ID;
  String name;
  String departmentNumber;

  Employee(String ID, String name, String departmentNumber)
  {
    this.ID=ID;
    this.name=name;
    this.departmentNumber=departmentNumber;
  }

  public String toString()
  {
    return ID+","+name+","+departmentNumber;

  }
}
```

We can write data into a text file, one record per line, fields separated by commas, like this:

```
try {
      FileOutputStream fstream = new FileOutputStream("data.txt");
      BufferedWriter br = new BufferedWriter(new OutputStreamWriter(fstream));
      Employee e1=new Employee("1", "John", "Sales");
      br.write(e1.toString()); br.newLine();
      Employee e2=new Employee("2", "Jane", "Accounts");
      br.write(e2.toString()); br.newLine();
      br.close();

    } catch (Exception ex) {
      ex.printStackTrace();
    }
```

The file contains:



We can recover the data by reading the file one line at a time then splitting the line into fields:

```
try {
      FileInputStream fstream = new FileInputStream("data.txt");
      BufferedReader br = new BufferedReader(new InputStreamReader(fstream));
      String str;
      while ((str = br.readLine()) != null) {
        String[] fields = str.split(",");
        for (String s : fields) {
          System.out.print(s + " ");
        }
        System.out.println();
      }
      br.close();

    } catch (Exception ex) {
      ex.printStackTrace();
    }
```

Output:

```
1 John Sales
2 Jane Accounts
```

Having ID and departmentNumber as strings is not a good idea - we do so because we are dealing with text files.