Object-Oriented Programming
I/O Streams

Dr. Seán Russell

`sean.russell@ucd.ie`

School of Computer Science,
University College Dublin

November 26, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the concept of streams in Java

- be able to create and use all of the different types of streams

- understand the concept of object serialisation

Table of Contents

Files

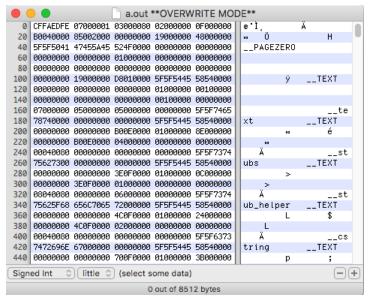- We usually distinguish between two different types of files

- Data Files

- Text Files

Data Files

- These types of files contain only data in binary

- If we try to look at these all we will see is numbers (usually in hex)

- It may not be possible to understand the meaning of the data
  - Two int values will look the same as one long value or eight bytes
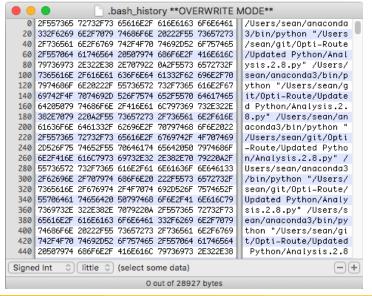
Data File

Text Files

- These files are used only to store text information

- This data is also stored in binary, but it is represented using ASCII codes

- All text editors understand how to convert this binary information to text for us to view and change

Text File

.bash_history **OVERWRITE MODE**

offset				text
0	2F557365	72732F73	65616E2F	616E6163	6F6E6461	/Users/sean/anaconda
20	332F6269	6E2F7079	74686F6E	20222F55	73657273	3/bin/python "/Users
40	2F736561	6E2F6769	742F4F70	74692D52	6F757465	/sean/git/Opti-Route
60	2F557064	61746564	20507974	686F6E2F	416E616C	/Updated Python/Anal
80	79736973	2E322E38	2E707922	0A2F5573	6572732F	ysis.2.8.py" ./Users/
100	7365616E	2F616E61	636F6E64	61332F62	696E2F70	sean/anaconda3/bin/p
120	7974686F	6E20222F	55736572	732F7365	616E2F67	ython "/Users/sean/g
140	69742F4F	7074692D	526F7574	652F5570	64617465	it/Opti-Route/Update
160	64205079	74686F6E	2F416E61	6C797369	732E322E	d Python/Analysis.2.
180	382E7079	220A2F55	73657273	2F736561	6E2F616E	8.py" ./Users/sean/an
200	61636F6E	6461332F	62696E2F	70797468	6F6E2022	aconda3/bin/python "
220	2F557365	72732F73	65616E2F	6769742F	4F707469	/Users/sean/git/Opti
240	2D526F75	74652F55	70646174	65642050	7974686F	-Route/Updated Pytho
260	6E2F416E	616C7973	69732E32	2E382E70	79220A2F	n/Analysis.2.8.py" ./
280	55736572	732F7365	616E2F61	6E61636F	6E646133	Users/sean/anaconda3
300	2F62696E	2F707974	686F6E20	222F5573	6572732F	/bin/python "/Users/
320	7365616E	2F676974	2F4F7074	692D526F	7574652F	sean/git/Opti-Route/
340	55706461	74656420	50797468	6F6E2F41	6E616C79	Updated Python/Analy
360	7369732E	322E382E	7079220A	2F557365	72732F73	sis.2.8.py" ./Users/s
380	65616E2F	616E6163	6F6E6461	332F6269	6E2F7079	ean/anaconda3/bin/py
400	74686F6E	20222F55	73657273	2F736561	6E2F6769	thon "/Users/sean/gi
420	742F4F70	74692D52	6F757465	2F557064	61746564	t/Opti-Route/Updated
440	20507974	686F6E2F	416E616C	6C6C6C6C	79736973	Python/Analysis.2.8

Signed Int   little   (select some data)   (−) (+)

0 out of 28927 bytes

Table of Contents

IO Streams

- This lecture focuses on the use of I/O streams in Java by using the examples of file reading

- However, most of the examples can be applied to streams that are connected to another source or destination such as a network location

- Investigating this will be left for you to do in your own time

I/O Stream

- An I/O stream represents an input source and an output destination

- This means that an I/O stream is really just a way to get data from one location to another

- There are many different kinds of sources and destinations
  - Files, devices, other programs, network resources, memory locations
- Streams can carry different types of data
  - Bytes, characters and other primitive types, objects

Sequence of Data

- A stream can be viewed as a sequence of data
  - A program can use an input stream to read data from a source
  - A program can use an output stream to write data to a destination
- When we describe it as a sequence of data, what is really meant is that the data will have some order
- If the source is a file it will be stored one piece of data after another, if the destination is a file we can only add one piece of data at a time

Streams

- There are many different types of streams
  - Byte streams

  - Character streams

  - Buffered streams

  - Data streams

  - Object streams

Table of Contents

Byte Streams

- The most basic type of stream is a byte stream

- These streams only read or write data one byte at a time

- When a program needs to understand the meaning of the data, a byte stream is not very good

- If we wish to read an `int`, it is 4 bytes of information

- To correctly read this int we would need to read all 4 bytes and then perform the correct operations to calculate the original value based on the 4 bytes

Why Have Byte Streams?

- Given that these streams are not very useful, why are we bothering to learn about them at all?

- The reason is that this basic functionality is used by other more complex streams

- The more complex streams are capable or reading the 4 bytes and automatically converting them into an int for us

- But internally they operate in this same basic way of reading a single byte at a time and then combing the data into the correct format

Types of Byte Streams

- There are many different types of byte streams, all of them extend the classes `InputStream` and `OutputStream` in the package `java.io`

- In our examples, we will be using the `FileInputStream` and `FileOutputStream`

- We will skip the information of reading and writing with byte streams

Table of Contents

Closing Streams

- Whenever we create a stream in Java, a number of objects are created in memory to help

- If we do not correctly deal with streams, these will continue to waste memory in our program, causing it to run more slowly or crash

- The JVM limits the amount of memory available to each program

- After we have finished using a stream we should close it

Closing Streams

- This is done using the method called `close`

- This is something that we should always do

- There are a number of strategies for choosing where to do this

- In the example above, it is done when we are finished writing to the file

- However, if there is a problem before we have finished writing, the close method will not be executed and therefore the stream is not closed

Close in Finally

- Alternatively, we can use the finally section to close the stream we are using

- However, because the close method might throw `IOException`, we must include another try-catch statement

- Additionally, because the stream may be null (if there is an exception before or during its creation), we must first check that it is not null before trying to close it

- This can make our code look much more complicated

Close in Finally

```
try{
    ...
} catch (IOException e){
    e.printStackTrace();
} finally {
    if(fw != null){
        try { fw.close(); } catch (Exception e2)
    {
            System.out.println("Steam could not be
    closed");
        }
    }
}
```

Table of Contents

Try With Resources

- This gives us code that is very messy and difficult to understand
- For this reason in Java 7, a new feature was added to simplify the process of using and closing streams
- The try with resources statement, is a special version of the try-catch statement that is designed to automatically close streams when we are finished using them
- Instead of creating the streams inside the try section, there are created in brackets before it
- Java will then automatically close the stream for us when the try section is finished or an exception happens

Try With Resources

```
1 try(declare and construct stream){
2   // use stream
3 } catch (expected exception){
4   // handle expected exception
5 }
```

- This allows us to declare and construct our stream
- It is also possible to not use a catch statement, if the exceptions thrown in your code a advertised instead of handled
- If we look at the write character example again, we can see that the code is much shorter and the easier to understand

Writing some Characters

```java
public static void main(String[] args) {
  String sos = "Hello darkness, my old
   friend\nI've come to talk with you again";

  try(FileWriter fw = new
   FileWriter("lChars.txt")){
    for (int i = 0; i < sos.length(); i++) {
      char c = sos.charAt(i); // get
   character
      fw.write(c); // write character
    }
  } catch (IOException e){
    e.printStackTrace();
  }
}
```

Table of Contents

Reading and Writing Lines

- Reading and writing characters is useful, but characters are small pieces of information

- Generally, we are used to working with larger pieces of information such as strings

- When reading and writing strings, generally work one line at a time

- So we will look at some classes that allow us to read and write data using strings.

Buffered Streams

- The classes that we are studying next are known as buffered

- These streams are designed to be much more efficient than the streams that we have been using

- When using normal I/O streams, every time we call the read or write method the functionality is performed by the operating system

- This causes the OS to have to access a file or a network resource, which is very slow

- When we use a buffered stream, data is read from or written to an area of memory called a buffer each time we use the read or write methods

The Buffer

- When the buffer is full (for writing) or empty (for reading), all of the data is written or new data is read

- Because this is done in a single operation for larger amounts of data, the process happens much faster

- If we wrote 6 bytes to a file, and the code was being executed on a machine with a magnetic hard drive, then every time we write a byte the OS would have to start spinning the drive, move the head to the correct location and then write the data

- Then the process is be repeated for the next byte

The Buffer

- However, if we use a buffered stream nothing is written to the disk until the buffer is full

- Because of the way hard drives work (separated into sectors of between 512 and 4096 bytes), it takes the same amount of time to write one byte as to write 100

- The reverse is also true for reading

Stream Wrapping

- Any stream can be converted into a buffered stream
- The idea is that we take a stream that will write data to a source and we place it inside a buffered stream
- When the buffer needs to be filled or emptied, it then calls the read or write method of the inner stream
- This means that in order to create a buffered stream, first we have to a stream to pass as a parameter
- This process is referred to as stream wrapping.

Types of Buffered Streams

- There are different types of buffered streams for different types of data

- For example to create a buffered byte stream we would use the `BufferedInputStream` or `BufferedOutputStream`

- For character streams we would use the `BufferedReader` and `BufferedWriter`

Table of Contents

Constructing a Buffered Stream

- To create a buffered character stream, we first need to create the character stream
- If we are reading a file `"ex.txt"`, first we would create `FileReader fr = new FileReader("ex.txt");`
- Once this is created, we can use it to create a BufferedReader object
- `BufferedReader br = new BufferedReader(fr);`
- These statements can be combined into a single statement by passing the result of the constructor of the first into the constructor of the second
- E.g. `BufferedReader br = new BufferedReader(new FileReader("chars.txt"))`

Table of Contents

Flushing Streams

- When writing data using a buffered stream, the data is only written when the buffer is full
- But what if the buffer is not full and we want the data to be written now?
- Java provides a method that we can use at any time to write any data that is currently in the buffer into the destination
- This method is called `flush`
- Whenever the flush method is executed, all data in the buffer will be written even if it is only a single byte

Table of Contents

Reading and writing with buffered streams

- The buffered streams provide the methods `read` and `write` which work exactly the same way as in the character and byte streams that we have seen already

- However, they also add methods that are much easier to use when reading or writing data in larger amounts

- Lets look at the methods of the `BufferedReader` and `BufferedWriter`

BufferedReader API

readLine

`public String readLine() throws IOException`
Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.
Returns: A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached
Throws: `IOException` - if an I/O error occurs.

BufferedWriter API

write

```
public void write(String str) throws IOException
```
Writes a string.
Parameters: str - String to be written
Throws: IOException - if an I/O error occurs.

Reading Some Lines

- Lets look at an example where we read 7 lines from a file and store them in an array to be printed later.

```java
public static void main(String[] args) {
    String[] lyrics = new String[7];
    try(BufferedReader br = new BufferedReader(new
        FileReader("chars.txt"))){
        for (int i = 0; i < lyrics.length; i++) {
            lyrics[i] = br.readLine();
        }
    }catch(IOException e){
        e.printStackTrace();
    }
    for (int i = 0; i < lyrics.length; i++) {
        System.out.println(lyrics[i]);
    }
}
```

Writing Some Lines

- And another example where we write every string in an array into a file, each as a separate line.

```java
public static void main(String[] args) {
  String[] words = new String[] {"Hi", "my", "name",
   "is", "Sean"};

  try (BufferedWriter bw = new BufferedWriter(new
    FileWriter("words.txt"))) {
    for (int i = 0; i < words.length; i++) {
      bw.write(words[i] + "\n");
    }
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Table of Contents

Processing Strings

- Chapter 9 in the book covered some basic string processing

- This can be easily applied to strings that have been read from files

- Consider an example of a program that reads lyrics of a song from one file and replaces some of the words before writing it into another file

- In this example we will use the `replace` method in the String class to achieve our aim

Knowing When to Stop

- The biggest problem that we face is that we do not know how many lines that there are in the file
- We could check and read that much information, but every time we want to use the program on a different file, we would need to find the number of lines
- Instead, it is easier to check every time if we have reached the end of the file
- In the API documentation of the `readLine` method, it states that null is returned when the end of the stream is reached
- So we try to read a line and loop as long as the variable is not null

Example

```java
public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(new
      FileReader("chars.txt"));
        BufferedWriter bw = new BufferedWriter(new
      FileWriter("words.txt"))) {
        String line = br.readLine();
        while(line != null){
          bw.write(line.replace("silence", "music") +
      "\n");
          line = br.readLine();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Why add \n?

- It should be noted, that because the readLine method does not include any new line characters in the strings it returns when writing these strings to the file the must be added

- This is why every time we write a line of text, we add a new line character too

Table of Contents

Data Representation

- Data is represented differently in the computer than the way it is presented to us
- For example when we print the number 123456789 we see this in the console as 9 characters, however in the computer internally it is represented as 00000111 01011011 11001101 00010101
- This is convenient because the second is very difficult to understand for humans
- If we were to write this number into a file using a character stream the information will require 9 bytes to be represented, while the second version takes on 4 bytes to be represented inside the computer

Data Streams

- Data streams allow us to store primitive data type values and strings in the same format as they are stored in memory in the computer

- This means that the files are in a format the the computer can easily understand, but to a human will often not make any sense

- This makes the operating with files more efficient, because we are no longer required to convert between the human version and the computer version

Types of Data Streams

- For writing and reading data we will use the `DataInputStream` and `DataOutputStream`

- These streams are constructed in the same way as the buffered streams

- In fact they are usually constructed **using** buffered streams

- This means that if we want our program to be efficient we need to create a total of three streams to read or write data

Constructing Data Streams

- First we need to construct a `FileInputStream`, then using that we construct a `BufferedInputStream` and finally using that we construct a `DataInputStream`

- It is acceptable to not use the buffered input stream and instead simply pass the file input stream to the data input stream, however this will result in a program that is less efficient

- The process is similar for the data output stream

Constructing Data Input Stream

```
1 FileInputStream fis = new
    FileInputStream("grades.dat");
2 BufferedInputStream bis = new
    BufferedInputStream(fis);
3 DataInputStream dis = new
    DataInputStream(bis);
4
5 OR
6
7 DataInputStream dis = new
    DataInputStream(new
    BufferedInputStream(new
    FileInputStream("grades.dat")));
```

Table of Contents

Different Data

- Because we can store different types of data and each will be stored in the same format as in memory, we need a method for each of these types of data
- For example, if I want to write an integer into a file I cannot use the same method that writes a double because these types are stored in different ways
- For example to write data we have the methods
  - `writeInt` for ints
  - `writeChar` for chars
  - `writeDouble` for doubles
- We the same for other primitive types
- To write a string we use the method `writeUTF`
- There are also similar methods for reading the same information

DataInputStream - readInt

`public int` readInt() `throws` IOException
Reads four input bytes and returns an int value. Let a-d be the first through fourth bytes read. The value returned is: `(((a & 0xff) << 24) | ((b & 0xff) << 16) | ((c & 0xff) << 8) | (d & 0xff)))`
Returns: the next four bytes of this input stream, interpreted as an int.
Throws: `EOFException` - if this input stream reaches the end before reading four bytes.
`IOException` - the stream has been closed and the contained input stream does not support reading after close, or another I/O error occurs.

DataInputStream - readUTF

```
public final String readUTF() throws IOException
```
Reads in a string that has been encoded using a modified UTF-8 format. The general contract of readUTF is that it reads a representation of a Unicode character string encoded in modified UTF-8 format;

Returns: a Unicode string.

Throws: `EOFException` - if this input stream reaches the end before reading four bytes.

`IOException` - the stream has been closed and the contained input stream does not support reading after close, or another I/O error occurs.

`UTFDataFormatException` - if the bytes do not represent a valid modified UTF-8 encoding of a string.

Reading Example

- Assume that we have a file that contains the names, student numbers (as ints) and three grades for a number of students

- Lets write a program to read the information about the first student stored in the file and then print the information to the screen

- Just like with the scanner and user input, the order that we write our statements must match the order that the data appears in the file

Reading Example

```java
public static void main(String[] args) {
    String name = "";
    int studentNumber = -1;
    double[] grades = new double[3];
    try {
        DataInputStream dis = new DataInputStream(new
        BufferedInputStream(new
        FileInputStream("grades.dat")));
        name = dis.readUTF();
        studentNumber = dis.readInt();
        for (int i = 0; i < grades.length; i++) {
            grades[i] = dis.readDouble();
        }
```

Reading Example

```java
12    } catch (IOException e){
13      e.printStackTrace();
14    }
15    System.out.print("Student " + name + " Number: "+
      studentNumber +" got grades ");
16    for (int i = 0; i < grades.length; i++) {
17      System.out.print(grades[i] + " ");
18    }
19 }
```

Reading Example Notes

- Note here that first we read the name of the student using the `readUTF` method, then the student number and then the three grades

- Because each of these pieces of information is stored in a different way, mixing up the order would result in data that does not make any sense and might crash the program

- Now lets look at an example of using the `DataOutputStream` to write the same data into a file for a single student

DataOutputStream - writeInt

`public final void writeInt(int v) throws IOException`
Writes an int to the underlying output stream as four bytes, high byte first.
Parameters: v - an int to be written.
Throws: `IOException` - if an I/O error occurs.

DataOutputStream - writeUTF

```
public final void writeUTF(String str) throws
IOException
```
Writes a string to the underlying output stream using modified UTF-8 encoding in a machine-independent manner. First, two bytes are written to the output stream as if by the `writeShort` method giving the number of bytes to follow. This value is the number of bytes actually written out, not the length of the string. Following the length, each character of the string is output, in sequence, using the modified UTF-8 encoding for the character.

Parameters: str - a string to be written.

Throws: `IOException` - if an I/O error occurs.

Writing Example

```java
public static void main(String[] args) {
  String name = "Sean Russell";
  int sn = 12345678;
  double[] grades = new double[] { 100, 95, 77.7 };

  try (DataOutputStream dos = new
   DataOutputStream(new BufferedOutputStream(new
   FileOutputStream("grades.dat")))) {
    dos.writeUTF(name);
    dos.writeInt(sn);
    for (int i = 0; i < grades.length; i++) {
      dos.writeDouble(grades[i]);
    }
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```