ITI 1121. Introduction to Computer Science II

Laboratory 7

Winter 2016

Objectives

- Introduction to Java **I/O** (input/output)
- Further understanding of exceptions

Introduction

This laboratory has two parts. The first part presents the basic concepts related to Java **I/O** that you will need for your work (save these lecture notes and examples as they will be useful for your data structure course next year!). The second part requires modifying an application called the **PlayList-Manager** for reading and writing songs from/to a file.

1 Java I/O

This document introduces the basic elements of the input/output (I/O) system in Java. In particular, it covers a subset of the classes that are found in the **java.io** package. Since Java 1.4, a new package has been introduced, **java.nio** (new io), that defines more advanced concepts, such as buffers, channels and memory mapping, these topics are not covered here.

Java I/O seems a bit complex at first. Firstly, there are many classes and secondly objects from two or more classes need to be combined for solving specific tasks. Why is that? Java is a modern language that has been developed when the World Wide Web was becoming a reality. As such, the data can be written/read to/from many sources, including the console/keyboard, external devices (such as hard disks) or the network. The presence of the Web stimulated the creation of classes for handling various encodings of the information (English and European languages but also Arabic and Asian languages; and also binary data).

1.1 Definitions

A **stream** is an ordered sequence of data that has a source or a destination. There are two major kinds of streams: **character streams** and **byte streams**.

In Java, characters are encoded with Unicodes — character streams are associated with text-based (human readable) I/O. The character streams are called **readers** and **writers**. This document focuses mainly on character streams. Byte streams are associated with data-based (binary) I/O. Examples of binary data include image and audio files, jpeg and mp3 files for example. Information can be read from an external source or written to an external source. Therefore, for (nearly) every input stream (or reader) there is a corresponding output stream (or writer). Besides input and output, there is a third access mode that is called **direct access**, which allows to read/write the data in any given order. This topic is only mentioned.

1.2 Overview

Two major categories of classes are found in the **I/O** package, **Stream** and **Access**. **Access** includes read, write, and direct. The classes related to **Stream** are as following:

- **Byte Streams:** handle I/O of raw binary data.
- **Character Streams:** handle I/O of character data, automatically handling translation to and from the local character set.
- **Buffered Streams:** optimize input and output by reducing the number of calls to the native API.
- **Scanning and Formatting:** allows a program to read and write formatted text.
- **I/O from the Command Line:** describes the Standard Streams and the Console object.
- **Data Streams:** handle binary I/O of primitive data type and String values.
- **Object Streams:** handle binary I/O of objects.

Note: We will focus on two different kinds of streams and three different access modes.

Furthermore, the medium that is used (keyboard, console, disk, memory or network) dictates its own constraints (the requirement for a buffer or not, for instance). The I/O package contains some 51 classes, 12 interfaces and 15+ Exceptions. The large number of classes involved is enough to confuse a beginner programmer. On the top of this, generally objects from 2 or 3 classes need to be combined to carry out a basic task. Example.

```
InputStreamReader in = new InputStreamReader( new FileInputStream( "data" ) );
```

where "data" is the name of the input file. The following sections are presenting the main concepts related to Java I/O. Most concepts are accompanied by simple examples and exercises, which illustrate specific topics. Compile and run all the examples. Complete all the exercises.

1.3 Streams

InputStream and **OutputStream** are two abstract classes that define the methods that are common to all input and output streams.

1.3.1 InputStream

The class **InputStream** declares the following three methods.

- **int read():** Reads the next byte of data from the input stream. The value of the byte is returned as an int in the range 0 to 255. If no byte is available, because the end of the stream has been reached, the value -1 is returned;
- **int read(byte[] b):** Reads some number of bytes from the input stream and stores them into the buffer array **b**. The number of bytes actually read is returned as an integer;
- **close():** Closes this input stream and releases any system resources associated with the stream.

Since **InputStream** is an abstract class it is never instantiated. Here are examples of its subclasses: **AudioInputStream**, **ByteArrayInputStream**, **FileInputStream**, **FilterInputStream**, **ObjectInputStream**, **PipedInputStream**, **SequenceInputStream**, **StringBufferInputStream**. One of the main **InputStream** classes that will be of interest to us is the **FileInputStream**, which obtains input bytes from a file in a file system, more on that later.

1.3.2 OutputStream

The class **OutputStream** declares the following three methods.

- **write(byte[] b):** Writes **b.length** bytes from the specified byte array to this output stream;
- **flush():** Flushes this output stream and forces any buffered output bytes to be written out;
- **close():** Closes this output stream and releases any system resources associated with this stream.

Since **OutputStream** is an abstract class, subclasses are used to create objects associated with specific types of I/O: **ByteArrayOutputStream**, **FileOutputStream**, **FilterOutputStream**, **ObjectOutputStream** and **PipedOutputStream**. **FileOutputStream** is commonly used. A file output stream is an output stream for writing data to a **File**.

1.3.3 System.in and System.out

Two objects are predefined and readily available for your programs. **System.in** is an input stream associated with the keyboard, and **System.out** is an output stream associated with the console.

1.4 Steps

Writing to (or reading from) a file generally involves three steps:

- Opening the file
- Writing to (or reading from) the file
- Closing the file

Closing the file is important to ensure that the data are written to the file, as well as to free the associated internal and external resources.

1.5 Reading

Let's narrow down the discussion related to reading from a file or reading from the keyboard.

1.5.1 Reading from a file

Reading from a file involves creating a **FileInputStream** object. We'll consider two constructors.

- **FileInputStream(String name):** This constructor receives the name of the file as an argument. Example,

  ```
  InputStream in = new FileInputStream( "data" );
  ```

- **FileInputStream(File file):** This constructor receives as an argument a **File** object, which is the representation of an external file.

```
File f = new File( "data" );
InputStream in = new FileInputStream( f );
```

Having a **File** object allows for all sorts of operations, such as,

```
f.delete();
f.exists();
f.getName();
f.getPath();
f.length();
```

to name a few. **FileInputStream** is a direct subclass of **InputStream**. The methods that it provides allow to read bytes.

1.5.2 InputStreamReader

Because a **FileInputStream** allows to read bytes only, Java defines an **InputStreamReader** as bridge from byte to character streams. Its usage is as follows.

```
InputStreamReader in = new InputStreamReader( new FileInputStream( "data" ) );
```

or

```
InputStreamReader in = new InputStreamReader( System.in );
```

where **System.in** is generally associated with the keyboard of the terminal.

- **int read():** Reads a single character. Returns -1 if the end of stream has been reached. The return type is **int**, the value -1 indicates the end-of-file (eof) (or end-of-stream (eos)). The value must be interpreted as a character, i.e. must be converted,

  ```
  int i = in.read();
  if ( i != -1 ) {
      char c = (char) i;
  }
  ```

  See Unicode.java and Keyboard.java.

- **int read(char [] b):** Reads characters into an array. Returns the number of characters read, or -1 if the end of the stream has been reached. Examples:

  ```
  InputStreamReader in = new InputStreamReader( new FileInputStream( "data" ) );

  int i = in.read();
  if ( i != -1) {
      char c = (char) i;
  }
  ```

  or

  ```
  char[] buffer = new char[ 256 ];
  num = in.read( buffer );
  String str = new String( buffer );
  ```

Exercise 1 *Write a class that reads characters for the keyboard using the method* **read(char[] b);** *the maximum number of characters that can be read at a time is fixed and determined by the size of the buffer. Use the class* ***Keyboard*** *from the above example as a starting point.*

Run some tests, do you notice anything bizarre? No matter how many characters you've entered the resulting String is always 256 characters long, furthermore, it contains several characters that are not printable. You must use the method **trim** to remove those non-printable characters.

1.5.3 BufferedReader

For some applications, the input should be read one line at a time. For this, you will be using an object of the class **BufferedReader**. A BufferedReader uses an InputStreamReader to read the content. The InputStreamReader uses an InputStream to read the raw data. Each layer (object) adds new functionalities. The InputStream reads the data (bytes). The InputStreamStream converts the bytes to characters. Finally, the BufferedReader regroups the characters into groups, here lines.

```
FileInputStream f = FileInputStream( "data" );
InputStreamReader is = new InputStreamReader( f );
BufferedReader in = new BufferedReader( is );
```

or

```
BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                            new FileInputStream("data") ) );
String s = in.readLine();
```

See Copy.java for an example related to this class — see Section Exception for further information about exception handling.

Exercise 2 *Write a class that prints all the lines that contain a certain word. For each line containing the word, print the line number followed by the line itself.*

Exercise 3 *Write a class that counts the number of occurrences of a given word within a file.*

Exercise 4 *Write a class that fetches the content of a Web page and prints it on the console.*

1.6 Writing out

Let's narrow down the discussion to writing to the console and writing to a file. Notice the similarities with reading in; how the process is mirrored.

1.6.1 Writing to a file

Writing to a file involves creating a **FileOutputStream** object. We'll consider two constructors.

- **FileOutputStream(String name):** Creates an output file stream to write to the file with the specified name. Example,

  ```
  OutputStream out = new FileOutputStream( "data" );
  ```

- **FileOutputStream(File file):** This constructor receives as an argument a **File** object, which is the representation of an external file.

  ```
  File f = new File( "data" );
  OutputStream out = new FileOuputStream( f );
  ```

  FileOutputStream is a direct subclass of **OutputStream**. The methods that it provides are for writing bytes.

- **OutputStreamWriter:** Because **FileOutputStream** writes bytes only. Java defines an **OutputStreamWriter** as a bridge from character streams to byte streams. Its usage is as follows.

  ```
  OutputStreamWriter out = new OutputStreamWriter( new FileOutputStream( "data " ) );
  ```

  or

  ```
  OutputStreamWriter out = new OutputStreamWriter( System.out );
  ```

  ```
  OutputStreamWriter err = new OutputStreamWriter( System.err );
  ```

System.err is the standard destination for error messages. The methods of an **OutputStreamWriter** are:

- **write(int c):** Writes a single character.
- **write(char[] buffer):** Writes an array of characters.
- **write(String s):** Writes a String.

Exercise 5 *Modify the class Copy.java so that it has a second argument that will be the name of a destination file. An accordingly, write a copy of the input to the output file.*

- **PrintWriter:** Prints formatted representations of objects to a text-output stream. It implements the following methods.

```
print( boolean b )  : Prints a boolean value.
print( char c )     : Prints a character.
print( char[] s )   : Prints an array of characters.
print( double d )   : Prints a double-precision floating-point number.
print( float f )    : Prints a floating-point number.
print( int i )      : Prints an integer.
print( long l )     : Prints a long integer.
print( Object obj ) : Prints an Object.
print( String s )   : Prints a String.
```

Similarly, the following methods also terminate the current line by writing the line separator string (which varies from one operating system to the next).

```
println()             : Prints a line separator string.
println( boolean b )  : Prints a boolean value.
println( char c )     : Prints a character.
println( char[] s )   : Prints an array of characters.
println( double d )   : Prints a double-precision floating-point number.
println( float f )    : Prints a floating-point number.
println( int i )      : Prints an integer.
println( long l )     : Prints a long integer.
println( Object obj ) : Prints an Object.
println( String s )   : Prints a String.
```

1.7 Exceptions

This section revisits some of the concepts related to exception handling in Java and presents some of the concepts that are specific to Java I/O.

1.7.1 IOException

"Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations." This is a direct subclass of **Exception**, this is therefore a checked exception that must be handled, caught by a **catch** clause or declared.

1.7.2 FileNotFoundException

The constructor **FileInputStream(String name)** throws an exception of type **FileNotFoundException**, which is a direct subclass of **IOException**. This exception must be handled, i.e. caught by a **catch** clause or declared.

1.7.3 finally

The finally clause of a **try** statement is executed whether or not an exception was thrown. It is often used in constructions such as this one.

```
Object val = null;
try {
    val = pre();
    // one or more statements
} finally {
    if ( val != null ) {
       post();
    }
}
```

If **pre()** succeeds it will return an **Object**. In the **finally** clause, we know that **pre()** has succeeded if **val** is not **null**. Some post-processing operations can then be performed.

Here is an application of this idiom for closing a file, even if an error has occurred.

```
public static void copy( String fileName ) throws IOException, FileNotFoundException {

   InputStreamReader input = null;
   try {
       input = new InputStreamReader( new FileInputStream( fileName ) );
       int c;
       while ( ( c = input.read() ) != -1 ) {
           System.out.write( c );
       }
   } finally {
       if ( input != null )
           input.close();
   }
}
```

Most operating systems impose a limit on the maximum number of files that can be opened simultaneously; Linux sets this limit to 16 by default. When a large number of files need to be read, searching for some information on the entire disk space, it is imperative to close all the files that were opened as soon as possible.

Notice that the **try** statement has no **catch** clause, therefore both checked exceptions that can be thrown must be declared.
See Copy.java

1.8 Formatting (numbers)

Again here, the solution proposed by Java is more complex than with most other programming languages. Java has been developed recently compared to other languages. For example, C has been developed in the early 1970s. Java proposes solutions that allows to internationalize programs — using "." or "," to separate the decimals depending on the set up of the local computer where the program is executed, for example.

1.8.1 java.text.Format

Format is the abstract superclass of all the formatting classes, namely **DateFormat** and **NumberFormat**. See what occurs when printing a floating point number (Test.java).

```
> java Test
3.141592653589793
```

Sometimes this what you want and sometimes not (e.g. printing dollar amounts). In order to print only a fixed number of decimals, one needs to create a **NumberFormat** instance and tell this instance how to format numbers. First, the **NumberFormat** class is not imported by default, therefore you will have to import it into your program. Next, you will create and instance and then use the instance methods **setMaximumFractionDigits** and **setMinimumFractionDigits** to set the number of digits for the fractional part to 2. Finally, you can use the instance method **format**, of the number format object, to create the string representations. See Test.java

```
import java.text.NumberFormat;

public class Test {

    public static void main( String[] args ) {

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);

        System.out.println( nf.format( 1.0/3.0 ) );
        System.out.println( nf.format( 100.0 ) );
    }
}
```

1.8.2 DecimalFormat

Alternatively, an **Object** of the class **DecimalFormat**, a direct subclass of **NumberFormat**, can be used. The following example prints a fractional number with 3 decimals and a width of 8 characters.

```
(from Jean Vaucher (and Guy Lapalme)'s tutorial)

import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;

float X,Y = .....;
String format1 = "###0.000" ;

static DecimalFormat fm1 = new DecimalFormat( format1,
                               new DecimalFormatSymbols( Locale.US ) );

System.out.println("X=" + fm1.format(x) + ", Y=" + fm1.format(y));
```

See Test.java

2 PlayListManager

For this laboratory, you will modify the program **PlayListManager** to read and write **Song**s from and to a file.

2.1 Reading Songs from a file

Modify the **PlayListManager** to read **Song**s from a file. For example, the name of the input file can be specified on the command line,

```
> java Run songs.csv
```

The input file contains one entry per line. Each entry consists of the title of the **Song**, the name of the **Artist** and the title of the **Album**. The fields are separated by ":".

```
A Dream Within A Dream:Alan Parsons Project:Tales Of Mystery & Imagination
Aerials:System Of A Down:Toxicity
Bullet The Blue Sky:U2:Joshua Tree
Clint Eastwood:Gorillaz:Clint Eastwood
Flood:Jars Of Clay:Jars Of Clay
Goodbye Mr. Ed:Tin Machine:Oy Vey, Baby
Here Comes The Sun:Nina Simone:Anthology
In Repair:Our Lady Peace:Spiritual Machines
In The End:Linkin Park:Hybrid Theory
Is There Anybody Out There?:Pink Floyd:The Wall
Karma Police:Radiohead:OK Computer
Le Deserteur:Vian, Boris:Titres Chansons D'auteurs
Les Bourgeois:Brel, Jacques:Le Plat Pays
Mosh:Eminem:Encore
Mosquito Song:Queens Of The Stone Age:Songs For The Deaf
New Orleans Is Sinking:Tragically Hip, The:Up To Here
Pour un instant:Harmonium:Harmonium
Sweet Dreams:Marilyn Manson:Smells Like Children
Sweet Lullaby:Deep Forest:Essence of the forest
Yellow:Coldplay:Parachutes
```

2.2 Writing Songs to a file

Modify the **PlayListManager** to write the **Song**s from the new **PlayList** to a file. For example, the name of the output file can be specified on the command line,

```
> java Run songs.csv workout.csv
```

The output file contains one entry per line. Each entry consists of the title of the **Song**, the name of the **Artist** and the title of the **Album**. The fields are separated by ":" (same as the input).

Solution

Suggestion: develop and test the methods **PlayList getSongsFromFile(String fileName)** and **void writeSongsToFile(String fileName)** in a separate class, say **Utils**. Once the methods are working add them to the **PlayListManager** application.

- media.jar (starting point)

3 Quiz (1 mark)

- Add all the necessary exception declarations to remove the compile-time errors (L8.java)

```
import java.io.*;

public class L8 {

    public static String cat( String fileName ) {

        FileInputStream fin = new FileInputStream( fileName );

        BufferedReader input = new BufferedReader( new InputStreamReader( fin ) );

        StringBuffer buffer = new StringBuffer();

        String line = null;

        while ( ( line = input.readLine() ) != null ) {

            line = line.replaceAll( "\\s+", " " );

            buffer.append( line );

        }

        fin.close();

        return buffer.toString();

    } // End of cat

    public static void main( String[] args ) {

        System.out.println( cat( args[ 0 ] ) );

    }
}
```

- Write your answer to the above question directly in the text field of the submission Web page;
- https://uottawa.blackboard.com/

See also

- http://download.oracle.com/javase/6/docs/api/java/util/Scanner.html

Resources

- java.sun.com/docs/books/tutorial/essential/io
- www.iro.umontreal.ca/∼vaucher/Java/tutorials/Java_files.txt

Last Modified: March 2, 2016