COMP3015
Data Communication and Networking
I/O in Java

Streams

Streams are the underlying abstraction behind communications in Java. A stream represents an endpoint of a one-way communication channel. The streams provide communication channels between a program and a particular file or device. For example, **System.in** (an input stream) enables a program to input bytes from the default input device (i.e., keyboard). **System.out** (a print stream) enables a program to output data to the default output device (i.e., screen). **System.err** (a print stream) enables a program to output error message to the default output device. Each of these streams can be redirected to another output device.

InputStream & OutputStream

InputStream and **OutputStream** are abstract classes that define methods for performing input and output respectively. Their derived classes override these methods.
The methods of the **InputStream** class:

Method	Description
available()	Returns an **estimate** of the number of bytes that can be read.
close()	Closes the stream and releases system resources associated with the stream.
read()	Returns the next byte of data from the stream in integer format.
read(btye[] b)	Reads some number of bytes from the stream into the byte array **b**, and returns the total number of bytes read into the byte array.
read(byte[] b, int off, int len)	Reads up to **len** bytes starting from the stream into the byte array **b**. The first byte will be stored into **b[off]**.
mark(int p)	Marks the current position in the stream.
markSupported()	Tests if the stream supports the **mark()** and **reset()** methods.
reset()	Repositions the stream to the marked position.
skip(long n)	Skips over and discards n bytes of data from the stream.

The methods of the **OutputStream** class:

Method	Description
flush()	Forces any buffered bytes to be written out to the stream.
close()	Closes the stream and releases system resources associated with the stream.
write(int b)	Writes the specified byte **b** to the stream.
write(byte[] b)	Writes bytes from the byte array **b** to the stream.
write(byte[] b, int off, int len)	Writes **len** bytes from the byte array **b** starting at offset off to the stream.

Read and Write

Again, **System.in** is an **InputStream** object, which reads data from the default input device. Without any redirection, the default input device is keyboard – the keyboard input through the console. **System.out** is a **PrintStream** object, which writes data to the default output device. Without any redirection, the default output device is the console screen. **PrintStream** is a derived class of the **OutputStream** class.

In the following example code, **System.out** is upcasted from a **PrintStream** type to an **OutputStream** type. In addition, the statement inside the while-loop outputs the console inputs to the console.

```
public static void consoleIO() throws IOException {
   InputStream in = System.in;
   OutputStream out = System.out;
   int count = 0;
   while(true) {

      out.write(in.read());

      System.out.println(++count);
   }
}
```

When the **consoleIO()** method is invoked, the **read()** method blocks the program until the user presses ENTER.

For example, after running the code, the user types “HELLO WORLD” and press **ENTER**. Then, 13 bytes of the data are put in the input stream. Note that the **ENTER** key is also captured.

0	1	2	3	4	5	6	7	8	9	10	11	12	
H	E	L	L	O		W	O	R	L	D	\n	\r	Input stream

The **read()** method reads and returns a byte from the input stream. The first round, it reads ‘H’. Then, the **write()** method writes the byte ‘H’ to the output stream. The while-loop repeats total 13 times to handle the string because the **read()** and **write()** methods handle only one byte each time.

The following example code uses **read(byte[] b, int off, int len)** method and **write(byte[] b, int off, int len)** method to handle input and output.

```
public static void consoleIO2() throws IOException {
   InputStream in = System.in;
   OutputStream out = System.out;

   byte[] buffer = new byte[10];

   int count = 0;

   while(true) {

      int len = in.read(buffer, 0, buffer.length);

      out.write(buffer, 0, len);

      System.out.println(++count);
   }
}
```

With the buffer (byte array) of size 10, the while-loop repeats total 2 times only to handle the same user inputted string.

Comparing with the previous version, the overhead of the new version is less, but additional memory space is required.

Overhead means the extra activities that are not directly related to the product creation, such as the expression validation in the **while** statement, the internal data handling activities of the stream, etc.

File

In Java, the File class provides file and directory manipulation. It has many methods for retrieving information of files or directories.

The following code should the detail information of a file or directory:

```
public static void getInfo(String filename) throws IOException {
   File file = new File(filename);
   System.out.println("name : " + file.getName());
   System.out.println("size (bytes) : " + file.length());
   System.out.println("absolute path? : " + file.isAbsolute());
   System.out.println("exists? : " + file.exists());
   System.out.println("hidden? : " + file.isHidden());
   System.out.println("dir? : " + file.isDirectory());
   System.out.println("file? : " + file.isFile());
   System.out.println("modified (timestamp) : " + file.lastModified());
   System.out.println("readable? : " + file.canRead());
   System.out.println("writable? : " + file.canWrite());
   System.out.println("executable? : " + file.canExecute());
   System.out.println("parent : " + file.getParent());
   System.out.println("absolute file : " + file.getAbsoluteFile());
   System.out.println("absolute path : " + file.getAbsolutePath());
   System.out.println("canonical file : " + file.getCanonicalFile());
   System.out.println("canonical path : " + file.getCanonicalPath());
   System.out.println("partition space (bytes) : " + file.getTotalSpace());
   System.out.println("usable space (bytes) : " + file.getUsableSpace());
}
```

The following are the sample results:

```
> D:/
                   name :
           size (bytes) : 8192
         absolute path? : true
                exists? : true
                hidden? : true
                   dir? : true
                  file? : false
   modified (timestamp) : 1535685446303
              readable? : true
              writable? : true
            executable? : true
                 parent : null
          absolute file : D:\
          absolute path : D:\
         canonical file : D:\
         canonical path : D:\
partition space (bytes) : 475915087872
   usable space (bytes) : 440655437824
```

```
> D:\img1.jpg
                   name : img1.jpg
           size (bytes) : 844818
         absolute path? : true
                exists? : true
                hidden? : false
                   dir? : false
                  file? : true
   modified (timestamp) : 1471969391181
              readable? : true
              writable? : true
            executable? : true
                 parent : D:\
          absolute file : D:\img1.jpg
          absolute path : D:\img1.jpg
         canonical file : D:\img1.jpg
         canonical path : D:\img1.jpg
partition space (bytes) : 475915087872
   usable space (bytes) : 440655437824
```

If the **File** object associates with an existing directory, we can use its **listFiles()** method to get a list of the files in the directory.

The following code prints the files and subdirectories of the specified directory.

```
public static void printFileList(String path) throws IOException {
   File dir = new File(path);

   if (dir.isDirectory()) {
      System.out.println(dir.getCanonicalPath());

      for (File file : dir.listFiles())

         if (file.isFile())
            System.out.println(file.getName() + ": " + file.length());
         else
            System.out.println(file.getName() + ": [dir]");
   }
}
```

File I/O

File Input/output is done with **FileInputStream** and **FileOutputStream**. **FileInputStream** is a subclass of **InputStream**, and **FileOutputStream** is a subclass of **OutputStream**.

The following example code uses **FileInputStream** to read and print the content of a text file to the console.

```
public static void readFile(String filename) throws IOException {
   byte[] buffer = new byte[1024];

   File file = new File(filename);
   FileInputStream in = new FileInputStream(file);
   long size = file.length();

   while(size > 0) {
      int len = in.read(buffer);
      size -= len;

      System.out.println(new String(buffer, 0, len));
   }
}
```

The code above works for the text file of any size. If the length of the file is larger than the buffer, the while-loop then repeats until the last byte of the file is read. The variable **size** is used as a counter to determine when the loop should stop.

Note that we should not always use the file size to create the buffer, because the size of a file can be up to 16 TB (terabytes) in Windows NTFS, 4GB (gigabytes) in FAT32, and 8EB (exabytes) in MacOS.

The following example code uses **FileOutputStream** to write the text to a file.

```
public static void writeFile(String filename) throws IOException {

   File file = new File(filename);

   System.out.println("Please enter the content. (enter @@quit to quit)");

   Scanner scanner = new Scanner(System.in);

   FileOutputStream out = new FileOutputStream(file);

   while(true) {
      String str = scanner.nextLine();
      if (str.equals("@@quit"))
         break;

      byte[] buffer = str.getBytes();

      out.write(buffer);

      out.write('\n');

   }

   out.close();
   scanner.close();
}
```

In the code above, we convert a string to a byte array using the **getBytes()** method. Then, we use the **write(byte[] b)** method to write the bytes to the output stream.

When an output stream is opened and the file object is associated with a nonexistent file, a new file will be created automatically. When an output stream is opened on an existing file, the file content will be overwritten by default.

If you want to append the new content to the existing file, you should use the following method to open an output stream on a file. The second parameter (Boolean) of the constructor specifies whether using the append mode or not.

```
FileOutputStream out = new FileOutputStream(file, true);
```

In addition, Java views each file as a sequential stream of bytes. So, we can use **FileInputStream** and **FileOutputStream** respectively to read and write files in any formats including JPEG, DOCX, XLSX, PDF, MP4, etc.

Advanced I/O Stream – DataInputStream & DataOutputStream

DataInputStream class and **DataOutputStream** class provides different methods for different primitive data types. With these methods, we can read data from the stream or write data to the stream using the specific data types.

The followings are commonly used methods provided by **DataInputStream**: *readByte(), readInt(), readFloat(), readDouble(), readLong(), readShort(), readBoolean(), readChar(),* and *read()*.

The followings are commonly used methods provided by **DataInputStream**: *write(), writeInt(), writeFloat(), writeDouble(), writeLong(), writeShort(), writeBoolean(), writeChar(),* and *writeBytes()*.

Consider the following code:

```
public static void consoleToFile() throws IOException {

   DataOutputStream out = new DataOutputStream(new FileOutputStream("test.dat"));

   Scanner scanner = new Scanner(System.in);

   int n;

   while (true) {
      try {

         n = scanner.nextInt();

      } catch (InputMismatchException ex) {

         break;

      }
      out.writeInt(n);
   }

   scanner.close();
   out.close();
}
```

The code above writes the inputted integers in the file named test.dat using the **writeInt()** method provided by **DataOutputStream**. If you open the file after inputting integers through the code, you will find that the integers stored in the file are unreadable. It is because the integers are converted in bytes (4 bytes for an integer) before writing.

To retrieve the integers from the file, we need to use the **readInt()** method provided by **DataInputStream**. The following code shows you how to read integers from the data file:

```
public static void fileToConsole() throws IOException {

   DataInputStream in = new DataInputStream(new FileInputStream("test.dat"));

   int n;

   while (in.available() > 0) {

      n = in.readInt();

      System.out.println(n);
   }
}
```

Note that you must use the corresponding method to read the data file otherwise you will not get the original values.

Applications of I/O Streams

I/O streams are usually using in programs, such as file copying, logging, and network-based applications. Consider the following scenario:

- A system logs all inputted commands in a data file for the troubleshooting purpose.
- Each log has to record a command with arguments and its execution time (timestamp).
- The lengths of the commands' names are different. Moreover, the number of arguments accepted by the commands are different. The example commands are:
    - dir c:\temp*.txt
    - copy c:\temp\123.txt d:\
    - edit d:\123.txt

Reading/Writing Plain Text

There are two approaches for saving the logs in the text file. The first approach is that the commands are written line-by-line in plain text format as follows:

```
1569150729152||dir c:\temp\*.txt
1569150759024||copy c:\temp\123.txt d:\
1569150768742||edit d:\123.txt
```

In the example content above, each line contains a timestamp and the command with the arguments.

The following is the sample code of the first approach:

```
public static void log1() throws IOException {

   Scanner scanner = new Scanner(System.in);
   FileOutputStream fo = new FileOutputStream("plaintext.log");

   while(true) {
      System.out.print("> ");

      String command = scanner.nextLine();

      if (command.equals("exit")) break;

      Date d = new Date();
      String log = String.format("%d||%s\n", d.getTime(), command);
      fo.write(log.getBytes());

   }

   fo.close();
   scanner.close();
}
```

Read/Writing Data in Different Data Type

Another approach is to save data using corresponding types. The following is the sample data stored in the data file:

```
mX□□0dir c:\temp\*.txtmX□Ycopy c:\temp\123.txt d:\mX□qedit d:\123.txt
```

In the example content above, all logs are saved in a single line with some unreadable characters. A log is followed by another. The actual format of saving a log is as follows:

Timestamp	Length of the command	Command
1569150729152	17	dir c:\temp*.txt
8 bytes	4 bytes	Dynamic length

The following is the sample code of the second approach:

```
public static void log2() throws IOException {

   Scanner scanner = new Scanner(System.in);
   DataOutputStream fo = new DataOutputStream(new FileOutputStream("type.log"));

   while(true) {
      System.out.print("> ");

      String command = scanner.nextLine();

      if (command.equals("exit")) break;

      Date d = new Date();

      fo.writeLong(d.getTime());
      fo.writeInt(command.length());
      fo.write(command.getBytes());
   }

   fo.close();
   scanner.close();

}
```

Note that we will use the second approach for data transmission over the network.

Exercise

Task1 (in-class):

Follow the steps below:

1. Create a Java project in Eclipse named **Lab1Task1** with a new class named **FileManager**.
2. Add a method named **copy()** to the **FileManager** class.
3. The **copy()** method accepts two parameters – source file name and target file name.
4. The method copies the content from the source file to the target file.
5. The method should be able to handle all file format, such as pdf, bmp, wav, etc.
6. Meaningful error messages should be prompted rather than terminating the program directly if an exception occurs.

Task 2:

Follow the steps below:

1. Create a Java project in Eclipse named **Lab1Task2a** and add **log1()** method to a new class named **PlainTextLog**.
2. Run the program and log some commands.
3. Create another Java project named **Lab1Task2b** with a new class named **PLotReader**. The **PLotReader** class has a method named **readLog()** that reads and prints the logs.

Task 3:

Follow the steps below:

1. Create a Java project in Eclipse named **Lab1Task3a** and add **log2()** method to a new class named **AdvLog**.
2. Run the program and log some commands.
3. Create another Java project named **Lab1Task3b** with a new class named **AdvLogReader**. The **AdvLogReader** class has a method named **readLog()** that reads and prints the logs.