Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

SISTEMAS Y SERVICIOS DISTRIBUIDOS

LAB MANUAL 1: JAVA I/O, NIO AND COLLECTIONS

Profesores:
Esteban Egea López

INDICE

1 Goals. 3
1. Input and output streams in Java 3
   1.1 Tasks 5
2. The new Java IO (Java NIO) 6
   2.1 Path, File and Files. 6
   2.2 Channels and Buffers 6
   2.3 Tasks 7
3. Generic containers 7
   3.1 Tasks 8
4. Exercises 9
5. References 9

1 Goals.

- ❑ Understanding the operation of Java I/O and the new I/O introduced in version 7 of Java.
- ❑ Knowing the different generic containers available.

1. Input and output streams in Java

As in most programming languages, input and output is abstracted as streams. A stream can represent both a source and a sink of data.
This abstraction allows the reading and writing process to be always the same: open the stream, read/write and close stream. What changes, of course, it is the nature of the origin/destination of the data: A stream can represent a file, a network connection, a camera, sonar, etc. And the type of data: bytes, characters, objects, etc.

To avoid an explosion of derived classes (sources x types), the decorator pattern is used: the basic class is "wrapped" in a class that provides additional functionality. That is, the decorator classes receive in their constructor an instance of the base class to decorate. These classes provide, as we say, additional functionality, for example, more efficient reading through buffer or methods for working with specific types (int, boolean, ...) instead of bytes. Moreover, the specialized classes are derived from basic classes to represent specific devices.

The InputStream / Reader and OutputStream / Writer classes provide the basic classes, with the fundamental operations of reading and writing (read () and write ()). InputStream and OutputStream are aimed at low level, ie, working with bytes, while Reader and Writer, versions are designed to allow text and use different character sets, including UTF-8. **You can make a conversion to InputStream from Reader with InputStreamReader and an equivalent for Output.**

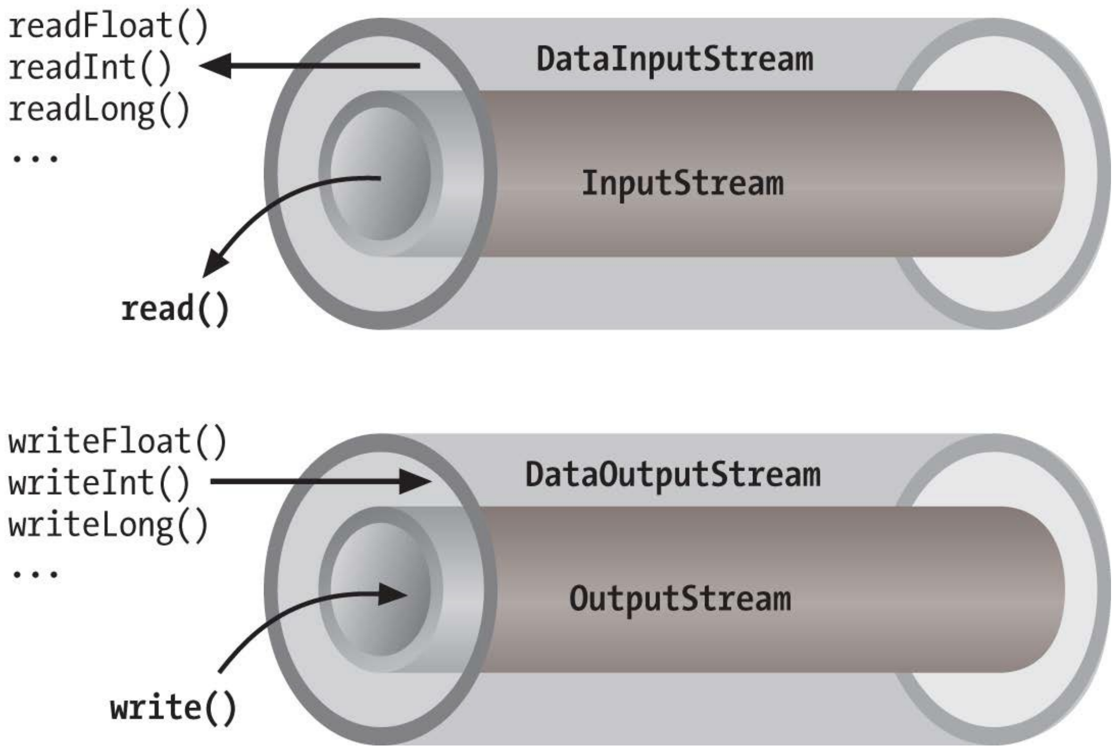
As examples, derived classes InputStream, which are used to connect to specific sources are, among others:

- ByteArrayInputStream: an array of bytes in memory.
- StringBufferInputStream: converts an array into an InputStream.
- FileInputStream: to a file.

The decorator classes that let you add additional functionality include, among others:

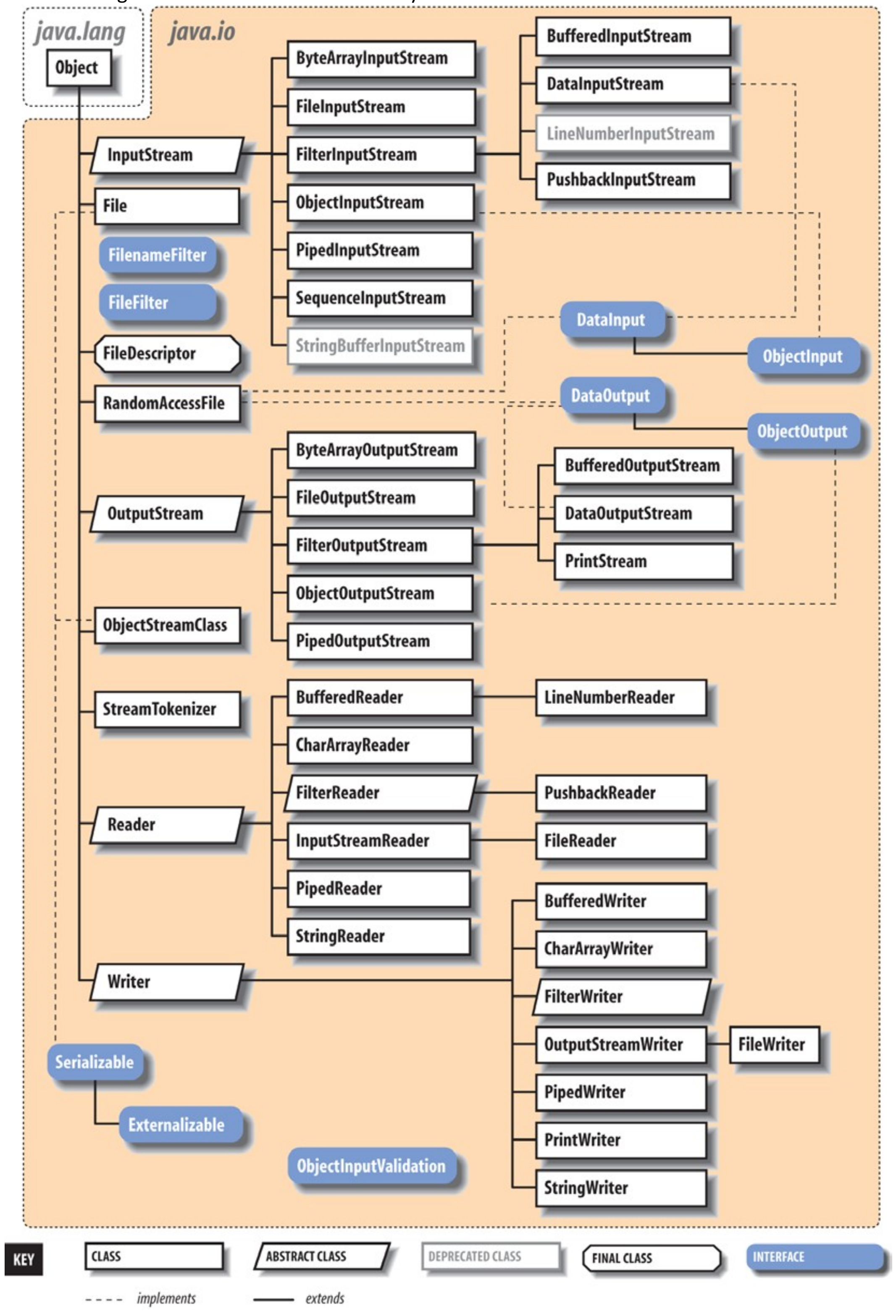
- BufferedInputStream: avoid direct physical reading, interposing a buffer.
- DataInputStream: read specific types, such as int, char, boolean, etc.

The following figure represents the decorator pattern



As shown, the method provides the basic InputStream read (), while the DataOutputStream provides more specific methods as readFloat (), readInt (), readLong (), ie, refined behavior.

The following table summarizes the hierarchy:



Finally, the following code fragment illustrates the use of the library:

```
DataInputStream din = new DataInputStream(new BufferedInputStream(new
FileInputStream(new File(“numbers.dat”))));
int i = din.readInt();
```

In this example, the File object (which represents a file descriptor) is passed to the FileInputStream constructor, which is the InputStream that allows us to read files. In addition BufferedInputStream decorator is used to interpose a buffer for reading disk, which has received the FileInputStream. Finally, we use a DataInputStream to read primitive types, passing the BufferedInputStream in the constructor.

There are plenty of tutorials about you can refer.

https://docs.oracle.com/javase/tutorial/essential/io/

Java.io package documentation is located at:

http://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html

Examine it. As you will see, in addition to multiple classes of utility streams can be combined in different ways.

In addition, reading of chapter 12 of Thinking in Java, 3rd Edition is recommended. Or, Learning Java, which can be found at:

http://chimera.labs.oreilly.com/books/1234000001805/ch12.html

1.1 Tasks

Use Eclipse to develop the following programs. For all programs, you must ALWAYS create a class that solves the exercise requested in one of its methods, but is not executable, and an associated RunX class that instantiates your class and executes the corresponding method. DO NOT USE PACKAGES (package) UNLESS YOU ARE SURE OF ITS USE

Before starting with the code create a directory and save in it several text files and images, which you will use in the following programs.

Make a program that copies a text file to another file. In a first version copy character by character. Display what you read on the screen and count the number of characters copied.

To carry out this program, create the CopyFile class with the copy () method and RunCopyFile class. The latter will simply execute the copy () method of CopyFile. In the copy () method of CopyFile, implement the program indicated. Resolve the following questions before implementing:

- You need to open a file for reading and another for writing. What classes are you going to use from the java.io class hierarchy for each one? Why?
- To open the corresponding files you need to indicate the path of the file in its constructor by means of a string. The path can be relative or absolute. First specify a relative path, that is, simply the name of the file (with extension). When the path is relative, the file has to be in the same directory where your java program runs, what is that directory?
- To copy you have to read from the input file and write to the output file using the basic methods, what are they?
- How can you display the read character on the screen?
- Remember to close the files.

In a second version, use a buffer for read and write and read and write full lines. Show what you read on the screen.

To carry out this program, create the copyBuffer () method in the CopyFile class. Resolve the following questions before implementing:

- You have to use the **decorators** corresponding to the hierarchy that allow you to both read full lines and write full lines, what are they? What are the methods of those decorators that you will use?
- You have to pass to the decorators the corresponding class that allows you to open to read or write the files.
- When you read a line, is the carriage return included in the read string?

Make a program that writes the numbers from 1 to 1000 to a file. Do it in binary format and in character format.

To carry out this program, create the WriteNumbers class with the write () method and RunWriteNumbers class. The latter will simply execute the write () method of WriteNumbers. Solve the following questions:

- Use **decorating classes** in both cases, which ones have you used?
- Are there differences in the size of the files you have created? Run the program again but now creating numbers up to 10000, 100000,… At what point does the file size start to differ? Why?

2. The new Java IO (Java NIO)

Java introduced in version 1.4 a new library called New input and output I/O (NIO) with a goal: speed and performance. For this, input and output mechanisms with performance closer to the underlying operating (or platform) system were added. In return, the API has a marked low level. With Java version 7 they also introduced new classes to work extensively with the file system of the machine.

The main classes of the new input and output are the Channels and Buffers, in particular for processing files, we have the FileChannel and ByteBuffer. Moreover, the main access points to the file system are the Files and Path classes.

For more information and examples, it is recommended that you read the following tutorial about it:
https://docs.oracle.com/javase/tutorial/essential/io/fileio.html

2.1 Path, File and Files.

The fundamental class for managing the file system is the *Path*. This class represents a path in a file system in an abstract way which can be relative or absolute and is dependent on the particular file system. This means that you cannot compare two objects *Path* from two different platforms, such as Linux and Windows. For the same reason, *Path* is not serializable. The class that represents a file independently of the platform is File. Unlike the previous one, File is serializable. It has methods to obtain from a *File* from *Path* and vice versa.

It also has the *Files* class which provides static methods for working with files and directories, or to copy and move files directly. A *Path* object lets you compose, compare routes, extract and add routes. A *Path* object needs not represent an actual route, but you can use *Files* to see if the path actually exists in the filesystem. In addition, through Files you can check file attributes. The following example shows the use of both classes, plus *FileSystems*.

```
String rootdir="imagenes";
Path rootpath=FileSystems.getDefault().getPath(rootdir);
Path abs = rootpath.toAbsolutePath();
Path img =FileSystems.getDefault().getPath(rootdir,”a.gif”);
boolean e = Files.exists(img);
boolean w = Files.isWriteable(img);
```

When *FileSystems.getDefault ()* is invoked the current directory is obtained and therefore a relative path is obtained. *Files* can list the contents of a directory, which are provided in a container *DirectoryStream <Path>* that allows to use iterators to scroll through the contents of a directory.

2.2 Channels and Buffers

Java NIO is based on the use of channels on which bytes are read and write. *Channels* can abstract files, sockets or other data sources. We will focus on the use of channels with files. Moreover, channels are written and read by *Buffers*. There are different types of buffers, as *CharBuffer* or *LongBuffer*, but we will focus on *ByteBuffers*.

The *Files* class allows to get a channel of bytes, *ByteChannel*, particularly a channel on which we can move forward and backward, ie, that allows to implement random access, by *SeekableByteChannel* class. This class can be converted by a cast to a *FileChannel* that allows access to properties and attributes of the channels. The basic usage is exemplified below:

```
Path ruta = …
SeekableByteChannel sbc = Files.newByteChannel(ruta);
FileChannel fc = (FileChannel) sbc;
```

Once a channel has been obtained, a *ByteBuffer* is used to work with it. This object is very similar to an array of bytes, byte [], while providing convenience methods of higher level. In fact, you can get a *ByteBuffer* from a byte [] using the *ByteBuffer.wrap()*. The *FileChannel* provides read operations, read () and write, write (), on a *ByteBuffer* for which we previously have established ByteBuffer size, with the allocate () method. Internally the *ByteBuffer* uses a position pointer to mark the current position in the buffer. When used for filling the buffer, it advances its position. If we have used a buffer to write (fill), we need to switch the initial position to read again with the *flip* () method. And to write again we must use *clear* ( ). **Note: mind the language, note that read and write operations on channel and buffer are complementary, that is, you read bytes a channel and store (write) then on the buffer, whereas when you write a channel you are reading them from the buffer.**

The following example will clarify the meaning of the above*:*

```
FileChannel in = (FileChannel) Files.newByteChannel(ruta1);
FileChannel out=(FileChannel)
Files.newByteChannel(ruta2,StandardOpenOption.CREATE,StandardOpenOption.APPEND);
ByteBuffer buffer = ByteBuffer.allocate(1024*8);
while(in.read(buffer) != -1) { //Lee del canal in
      buffer.flip(); // Prepara el buffer para escribir en el canal (leer del
      buffer)
      out.write(buffer); //Escribe en el canal out
      buffer.clear(); // Prepara el buffer para leer de nuevo (escribir en el
buffer)
}
```

You can get more details of the above concepts and the inner workings of Buffers in the following tutorial:
http://tutorials.jenkov.com/java-nio/buffers.html

2.3 Tasks

Create a program that, given the directory, lists the contents of the directory.
To carry out this program create the DirOps class with the list () method and RunDirOps. The latter will simply execute the list () method of DirOps. Solve the following questions:

- Use Path, Files, and Filesystems to list the contents of your directory.

Programmatically create a subdirectory.
To carry out this program, create a create () method in the DirOps class.

- Get the current directory as a Path and use its resolve () method to pass the subdirectory path to the Files function that creates the directories.

Copy one of the images from your directory to the new subdirectory with another name. Use channels and buffers.

To carry out this program, create a copy () method in the DirOps class.

- Get the Path to your image with Paths.get (). Note that this method takes a URI as a parameter. A URI allows you to express a route independently of the system. For example, the path "images / a.gif" is specified the same for Windows and Linux, it is not necessary to change the slash.
- Get the path where you are going to copy again with Paths.get () and use resolve () to pass it to the channel.

3. Generic containers

Java provides a complete library of generic containers, i.e. classes used to manage a collection of objects. The collections are classified into different types, with different properties, for example, a *Set* is a collection of objects that can not contain duplicate elements, while a *List* is an ordered collection and Map is a collection of pairs of key-objects, where the key is unique.

The types of collections are defined by interfaces, which are then carried out by a particular implementations. The following table shows the interfaces (types of collections) and their implementations.

Interfaces	**Hash table Implementations**	**Resizable array Implementations**	**Tree implementations**	**Linked list Implementations**	**Hash table + Linked list Implementations**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

The collections are implemented using parameterized types, also called templates (templates), or generics in Java. This means that the data type that stores and manages a collection is specified as a parameter between < and >. The type will be checked at compile time to avoid mistakes. Consider the following example:

```
List<String> list = new ArrayList<String>();
list.add("hello");
```

```
String s = list.get(0)
```

The collections are traveled by iterators, which can be considered pointers to items in the list. Most collections provide iterator objects. Consider the following example:

```
ArrayList<Path> al = new
ArrayList<Path>(); Iterator itr =
al.iterator(); while(itr.hasNext()) {
   Path element = itr.next();
   System.out.print(element + "
   ");
}
System.out.println();


ListIterator<Path> litr =
al.listIterator(); while(litr.hasNext()) {
   Path element =
   litr.next();
   litr.set(element + "+");
}
```

Or you can get a collection of a certain type from others. For example, a key of a Map are unique, so they can be represented by a set. For example, a Map <int, String> returns a Set <int> when the keySet () method. We can get an iterator from Set and go through all elements of the Map from that iterator:

```
for (int key : m.keySet())
    System.out.println(key+ “
    “+m.get(key));
```

In most cases you can create a collection from another, passing in the constructor.

Finally, in addition to a large number of algorithms for working with collections it is available as well. It has algorithms for sorting, mixing, copy, exchange, reverse the order of items, fill, search, compose, etc.

Algorithms are accessed by the Collections class. For example, to sort a list:

```
List<String> list = Arrays.asList(args);
Collections.sort(list);
System.out.println(list);
```

To apply an algorithm on a class by itself, it is necessary to indicate how such objects are compared. For example, we have a Student class, which internally stores an ID by a long, plus a String with the student 's name. We can create a list of these objects, but to order it, we have to indicate how they compare to each other and to order them. There are several ways to indicate it, one of which is to make our type implements the Comparable interface. Thus, our class must implement a method *int compareTo (Student)* that returns a negative, zero or positive integer if the object is less than, equal to or greater than the other object. There are algorithms that simply check equality. For those cases, our class must implement both *equals ()* and *hashCode ()*. If our class is composed of other primitive types that define equality or comparison, in most cases it is preferable to delegate all these methods to primitive types. For example, if we make the comparison by name, we can do:

```
int compareTo(Alumno a) {
 this.getNombre().compareTo(a.getNombre());
}
```

More information in:
https://docs.oracle.com/javase/tutorial/collections/

3.1 Tasks

Declare a Complex object that represents a complex number and has a method that returns the module of it. Fill a list with complex numbers and use sort to sort by module and display it on screen.

To carry out this program, create the Complex class with the module () method. This method will be in charge of calculating the modulus of the complex number. Create the RunComplex class that, in its main () method, will be in charge of creating several complexes, adding them to a collection, displaying them on the screen, sorting them and showing them again.

- Use the Comparable interface to implement complex comparison.

Copy some of the files from your directory to another directory through the operating system. Create a program that reads the contents of both directories, stores them in an appropriate container, and then displays on screen the files common to both directories and the files that appear in one and not the other, for both. Do it using only containers and the methods and algorithms they provide.

To perform this program, add a compare () method to the DirOps class that you created.

- Decide what type of container you will use: Set, List, Map… why?
- List each directory and add its contents (filename) to the container.
- Use appropriate container operations or Collection algorithms to make the comparison.

4. Exercises

Use Eclipse to implement the following programs.

a) **Create a program that overwrites a fragment of a text file with another fragment from another file. To do this, read a block from a text file and write it in the other file from the position 10. What happens if the file is opened with the option APPEND?**

b) **Inserting a text into another text file is more complicated. As an exercise, if you have time, do a program to insert a piece of text in another text file, from the position or 10 without reading the contents of both files to memory.**

5. References

0. Bruce Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2006
(Prior editions are available at http://www.mindviewinc.com/Books/downloads.html)