Topic: Using InputStream.
OOP concepts involved: Classes, Objects, Constructors, Exceptions, Polymorphism.
Programming generic concepts involved: Functions, Variables, Data Types, Arrays, Control Statements, Access Modifiers, Data Streams.

➢ Theoric introduction

FILE ACCESS MECHANISMS

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files:

Sequential access

The records are read from the beginning to the end of the file in such a way that, to read a record, all the previous ones are read.

- ➔ It requires the treatment of the element, for this a sequential exploration is necessary from the first moment. (Text file to be read from beginning to end).
- ➔ This access method is the most primitive one.
- ➔ *Example*: Compilers usually access files in this fashion.

Direct access

Each record can be read/written directly only by expressing its address in the file by the relative number of the record or by transformations of the registration key in the relative number of the record to be accessed.

- ➔ It allows to process or access a specific element and reference directly by its position in the storage medium.

➔ Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.

Indexed sequential access

The records are indirectly accessed by their key, by sequential queries to a table containing the key and the relative address of each record, and subsequent direct access to the record.

➔ This mechanism is based on sequential access.
➔ An index is created for each file which contains pointers to various blocks.
➔ Index is searched sequentially and its pointer is used to access the file directly.

INPUT AND OUTPUT

Java's IO package mostly concerns itself with the reading of raw data from a source and writing of raw data to a destination. The most typical sources and destinations of data are these:

➔ Files
➔ Pipes
➔ Network Connections
➔ In-memory Buffers (e.g. arrays)
➔ System.in, System.out, System.error

The diagram below illustrates the principle of a program reading data from a source and writing it to some destination:

STREAMS

IO Streams are a core concept in Java IO. A stream is a conceptually endless flow of data. You can either read from a stream or write to a stream. A stream is connected to a data source or a data destination. Streams in Java IO can be either **byte based** (reading and writing bytes) or **character based** (reading and writing characters).

Every time we finish using a stream of data, we need to close it.

BINARY FILE READING (USING INPUTSTREAM)

If you want to read information from a binary file, the InputStream class is our option. This class is used for reading byte based data, one byte at a time.

The Java InputStream class represents an ordered stream of bytes. In other words, you can read data from a Java InputStream as an ordered sequence of bytes. This is useful when reading data from a file, or received over the network.

The Java InputStream class is the base class (superclass) of all input streams in the Java IO API. InputStream subclasses include the ***FileInputStream***, ***BufferedInputStream*** and the ***PushbackInputStream*** among others.

The *read()* method

The *read()* method of an **InputStream** returns an int which contains the byte value of the byte read. Here is an **InputStream** *read()* example:

```
int data = inputstream.read();
```

You can cast the returned int to a char like this:

```
char aChar = (char) data;
```

Subclasses of **InputStream** may have alternative *read()* methods. For instance, the **DataInputStream** allows you to read Java primitives like int, long, float, double, boolean etc. with its corresponding methods *readBoolean()*, *readDouble()*, etc.

Closing an InputStream

When you are done with a Java InputStream you must close it. You close an **InputStream** by calling the **InputStream** *close()* method. Here is an example of opening an **InputStream**, reading all data from it, and then closing it:

```
InputStream inputstream = new FileInputStream("c:\\data\\input-text.txt");

int data = inputstream.read();
while(data != -1) {
  data = inputstream.read();
}
inputstream.close();
```

TEXT FILE READING (USING READER)

The Java **Reader** class (java.io.Reader) is the base class for all Reader subclasses in the Java IO API. A Reader is like an **InputStream** except that it is character based rather than byte based. In other words, a Java Reader is intended for reading text, whereas an InputStream is intended for reading raw bytes.

Reading Characters with Reader

The *read()* method of a **Reader** returns an int which contains the char value of the next character read. If the *read()* method returns -1, there is no more data to read in the **Reader**, and it can be closed.

Reader subclasses

You will normally use a **Reader** subclass rather than a Reader directly. Java IO contains a lot of Reader subclasses. For instance, the *InputStreamReader*, *CharArrayReader*, *FileReader*, plus many others.

➢ Statement

Create a program that reads a file as a text file, line by line using the ***readLine()*** method of the **BufferedReader** class.

Every time a new line is read it will be appended to a **StringBuilder** object so that in the end it is converted to a String and later it is displayed by the console.

➢ Program Code

textfile.txt

```
This is a simple
and short example
of a text file.
```

ReadTextFile.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadTextFile {
	public static void main(String[] args) {
		BufferedReader br = null;
		try {
			br = new BufferedReader(new FileReader("textfile.txt"));
			StringBuilder sb = new StringBuilder();
			String line = br.readLine();

			while (line != null) {
				sb.append(line);
				sb.append(System.lineSeparator());
				line = br.readLine();
			}
			String everything = sb.toString();
			System.out.println(everything);
		} catch (IOException e) {
			e.printStackTrace();
		} finally {
			try {
				br.close();
```

```
                    } catch (IOException e) {
                          e.printStackTrace();
                    }
              }
        }
}
```

➢ **Program execution**

This program reads the contents of a file called "textfile.txt", this reading it does, considers it as a text file and not binary.

The program reads the content line by line, using the ***readLine()*** method provided by the **BufferedReader** class. Each time a line of the file is read, it will be appended to a StringBuilder type string, so when finishing reading the text file, we will have all the contents of our file in a StringBuilder type string and it is only matter of printing it through the console.

```
This is a simple
and short example
of a text file.
```

➢ **Conclusions**

There are different ways to access a file, either sequentially, directly or by index.

Java allows us to use a large variety of specialized subclasses from the parent classes **InputStream** and **Reader**. The **InputStream** class is used to read binary data, while the **Reader** class is used to read data in text format.

Both classes have a ***read()*** method to read information from a certain source, such as Files, Pipes, Network Connections, In-memory Buffers (eg, arrays), System.in, System.out, System.error. In order to write to certain destinations and to read from certain sources, we need the use of Streams. A Stream conceptually is an endless flow of data. Every time we finish using a stream of data, we need to close it.