Topic: Using OutputStream.
OOP concepts involved: Classes, Objects, Constructors, Exceptions, Polymorphism.
Programming generic concepts involved: Functions, Variables, Data Types, Arrays, 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 WRITING (USING OUTPUTSTREAM)

If you want to write information to a binary file, the OutputStream class is our option. This class is used for writing byte based data, one byte at a time.

The **OutputStream** class is the base class of all output streams in the Java IO API. Subclasses include the **BufferedOutputStream** and the **FileOutputStream** among others. An **OutputStream** is typically always connected to some data destination, like a file, network connection, pipe etc.

The OutputStream's data destination is where all data written to the OutputStream will eventually end.

The *write()* method

The **write(byte)** method is used to write a single byte to the **OutputStream**. The *write()* method of an OutputStream takes an int which contains the byte value of the byte to write. Only the first byte of the int value is written. The rest is ignored.

The OutputStream class also has a **write(byte[] bytes)** method and a **write(byte[] bytes, int offset, int length)** which both can write an array or part of an array of bytes to the OutputStream.

- ➔ The **write(byte[] bytes)** method writes all the bytes in the byte array to the OutputStream.
- ➔ The **write(byte[] bytes, int offset, int length)** method writes length bytes starting from index offset from the byte array to the OutputStream.

Subclasses of **OutputStream** may have alternative *write()* methods. For instance, the **DataOutputStream** allows you to write Java primitives like int, long, float, double, boolean etc. with its corresponding methods *writeBoolean(), writeDouble()* etc.

The *flush()* method

The OutputStream's *flush()* method flushes all data written to the **OutputStream** to the underlying data destination. For instance, if the **OutputStream** is a **FileOutputStream** then bytes written to the **FileOutputStream** may not have been fully written to disk yet. The data might be buffered in

memory somewhere, even if your Java code has written it to the **FileOutputStream**. By calling ***flush()*** you can assure that any buffered data will be flushed (written) to disk (or network, or whatever else the destination of your **OutputStream** has).

Closing an OutputStream

When you are done writing data to the OutputStream you should close it. You close an **InputStream** by calling its ***close()*** method. Here is an example of opening an **InputStream**, reading all data from it, and then closing it:

```
OutputStream output = null;

try{
  output = new FileOutputStream("c:\\data\\output-text.txt");
  // imagine hasMoreData() returns true if there's more data
  while(hasMoreData()) {
    int data = getMoreData();    // imagine we get data
    output.write(data);
  }
} finally {
    if(output != null) {
        output.close();
    }
}
```

TEXT FILE WRITING (USING WRITER)

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

A Java **Writer** is typically connected to some destination of data like a file, char array, network socket etc.

Reader subclasses

You will normally use a **Writer** subclass rather than a **Writer** directly. Subclasses of Writer include *OutputStreamWriter*, *CharArrayWriter*, *FileWriter*, plus many others.

JAVA IO CLASS OVERVIEW TABLE

Here is a table listing most (if not all) Java IO classes divided by input, output, being byte based or character based, and any more specific purpose they may be addressing, like buffering, parsing etc.

	Byte Based		**Character Based**	
	Input	Output	Input	Output
Basic	*InputStream*	*OutputStream*	*Reader* *InputStreamReader*	*Writer* *OuputStreamWriter*
Arrays	*ByteArrayInputStream*	*ByteArrayOutputStream*	*CharArrayReader*	*CharArrayWriter*
Files	*FileInputStream* *RandomAccessFile*	*FileOutputStream* *RandomAccessFile*	*FileReader*	*FileWriter*
Pipes	*PipedInputStream*	*PipedOutputStream*	*PipedReader*	*PipedWriter*
Buffering	*BufferedInputStream*	*BufferedOutputStream*	*BufferedReader*	*BufferedWriter*
Filtering	*FilterInputStream*	*FilterOutputStream*	*FilterReader*	*FilterWriter*
Parsing	*PushbackInputStream*	*StreamTokenizer*	*PushbackReader*	*LineNumberReader*
Strings			*StringReader*	*StringWriter*
Data	*DataInputStream*	*DataOutputStream*		
Data - Formatted		*PrintStream*		*PrintWriter*
Objects	*ObjectInputStream*	*ObjectOutputStream*		
Utilities	*SequenceInputStream*			

➢ Statement

Create a program that represents a String in a byte array.

The byte array must be written to the binary.bin file, after that the program writes 12 bytes starting from the index 14 of the byte array, and finally, writes the first byte of the byte array.

Once the bytes have been written to the binary.bin file, use an InputStream type object to read and display the content previously saved in binary.bin through the console.

➢ Program Code

WriteBinaryArray.java

```
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class WriteBinaryArray {
	public static void main(String[] args) {

		String content = "Hello World, OOP Students! ";

		byte[] bytes = content.getBytes();	// Making an array of bytes with te
content of a String

		// using try-with-resources statement
		// A buffer size of 1024 bytes
		try (OutputStream out = new BufferedOutputStream(new
FileOutputStream("binary.bin"),1024)) {

			// write a byte sequence
			out.write(bytes);
			// write a single byte
			out.write(bytes[0]);
			// write sub sequence of the byte array
			out.write(bytes,14,12); // writing 12 bytes starting at index 14
		} catch (IOException e) {
			e.printStackTrace();
```

```
        }

        // Printing the content of the binary.bin file that we just created
        int i;
        // using try-with-resources statement
        try (InputStream inputStream = new FileInputStream("binary.bin")) {
             while((i = inputStream.read()) != -1) {
                   System.out.print((char) i);
             }
        } catch(IOException e) {
             e.printStackTrace();
        }
    }
}
```

➢ **Program execution**

This program writes bytes to a file named “binary.bin” and then it retrieves them and prints them through console.

binary.bin

```
Hello World, OOP Students! HOP Students!
```

➢ **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 **OuputStream** and **Writer**. The **OutputStream** class is used to write binary data, while the **Writer** class is used to write data in text format.

Both classes have a ***write()*** method to write information to a certain destination 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.