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

➢ Theoric introduction

PERSISTENT DATA

Persistence, in computer science, is a noun describing data that outlives the process that created it. *Persistent data* is data that's considered durable at rest with the coming and going of software and devices. Master data that's stable—that is set and recoverable whether in flash or in memory.

With persistent data, there is **reasonable confidence that changes will not be lost** and the data will be available later. Depending on the requirements, in-cloud or in-memory systems can qualify. We care most about the "data" part. If it's data, we want to enable customers to read, query, transform, write, add-value, etc.

There are many ways to make data persist in Java, including (to name a few): JDBC, **serialization**, file IO, JCA, object databases, and XML databases. However, the majority of data is persisted in databases, specifically relational databases. Most things that you do on a computer or website that involve storing data involve accessing a relational database. Relational databases are the standard mode of persistent storage for most industries, from banking to manufacturing.

OBJECT FLOWS

It is possible to write and read objects of a flow of any kind.

The **ObjectInputStream** and **ObjectOutputStream** classes offer the ability to write both primitive data and objects.

These classes allow you to permanently save the state of an object and then retrieve it. This is possible through the ***serialization*** of objects.

Serialization

Serialization is the process of transferring the corresponding bytes to an object through a flow. This process guarantees that the object is transmitted completely.

In Java, the Serializable interface is used to implement the serialization of objects. This interface does not have methods, it only indicates to the virtual machine that this object can be transmitted by a flow.

Here is an example of a class that implements the Java *Serializable* interface:

```
public static class Person implements Serializable {
    public String name = null;
    public int    age  =   0;
}
```

As you can see, the **Person** class implements the *Serializable* interface, but does not actually implement any methods. As mentioned earlier, the Java *Serializable* interface is just a marker interface so there are no methods to implement.

THE FILEINPUTSTREAM CLASS

The Java FileInputStream class makes it possible to read the contents of a file as a stream of bytes. The Java FileInputStream class is a subclass of Java InputStream. This means that you use the Java FileInputStream as an InputStream (FileInputStream behaves like an InputStream).

Most Common FileInputStream Constructors

➔ *Taking a String path as a parameter*

This String should contain the path in the file system to where the file to read is located. Here is a code example:

```
String path = "C:\\user\\data\\thefile.txt";

FileInputStream fileInputStream = new FileInputStream(path);
```

Notice the path String. It needs double backslashes (\\) to create a single backslash in the String because backslash is an escape character in Java Strings. To get a single backslash you need to use the escape sequence \\.

On a Unix like system the file path could have looked like this:

```
String path = "/home/user/data/thefile.txt";
```

➔ *Taking a File object as a parameter*

The File object has to point to the file you want to read. Here is an example:

```
String path = "C:\\user\\data\\thefile.txt";
File   file = new File(path);

FileInputStream fileInputStream = new FileInputStream(file);
```

Which of the constructors you should use depends on what form you have the path in before opening the FileInputStream. If you already have a String or File, just use that as it is. There is no particular gain in converting a String to a File, or a File to a String first.

THE FILEREADER CLASS

The Java ***FileReader*** class (java.io.FileReader) makes it possible to read the contents of a file as a stream of characters. It works much like the FileInputStream except the FileInputStream reads bytes, whereas the FileReader reads characters. The ***FileReader*** is **intended to read text**, in other words. One character may correspond to one or more bytes depending on the character encoding scheme (UTF-8, UTF-16, etc.).

THE FILEOUTPUTSTREAM CLASS

The FileOutputStream class makes it possible to write a file as a stream of bytes. The FileOutputStream class is a subclass of OutputStream meaning you can use a FileOutputStream as an OutputStream.

Most Common FileOutputStream Constructors

➔ *Taking a String path as a parameter*

This String should contain the path of the file to write to. Here is an example:

```
String path = "C:\\user\\data\\binaryfile.txt";

FileOutputtStream output = new FileOutputStream(path);
```

Notice the path String. It needs double backslashes (\\) to create a single backslash in the String because backslash is an escape character in Java Strings. To get a single backslash you need to use the escape sequence \\.

On a Unix like system the file path could have looked like this:

```
String path = "/home/user/data/binaryfile.txt";
```

➔ *Taking a File object as a parameter*

The File object has to point to the file you want to write in the file system. Here is an example:

```
String path = "C:\\user\\data\\textfile.txt";
File   file = new File(path);

FileOutputStream output = new FileOutputStream(file);
```

Overwriting vs. Appending the File

When you create a *FileOutputStream* pointing to a file that already exists, **you can decide if you want to overwrite the existing file, or if you want to append to the existing file**. You decide that

based on which of the ***FileOutputStream*** constructors you choose to use.

This constructor which takes just one parameter, the file name, will overwrite any existing file:

```
OutputStream output = new FileOutputStream("c:\\data\\output-text.txt");
```

There is a constructor that takes 2 parameters too: The file name and a boolean. The boolean indicates whether to append or overwrite an existing file. Here are two examples:

```
OutputStream output = new FileOutputStream("c:\\data\\output-text.txt",
true); //appends to file

OutputStream output = new FileOutputStream("c:\\data\\output-text.txt",
false); //overwrites file
```

THE FILEWRITER CLASS

The Java ***FileWriter*** class (java.io.FileWriter) makes it possible to write characters to a file. In that respect, it works much like the FileOutputStream except that a FileOutputStream is byte-based, whereas a FileWriter is character based. The FileWriter is intended to write text, in other words. One character may correspond to one or more bytes, depending on the character encoding scheme in use.

Here is a simple Filewriter example:

```
Writer fileWriter = new FileWriter("data\\filewriter.txt");

fileWriter.write("data 1");
fileWriter.write("data 2");
fileWriter.write("data 3");

fileWriter.close();
```

The ***FileWriter*** has other constructors too, letting you specify the file to write to in different ways.

As in the ***FileOutputStream*** class, an object of the ***FileWriter*** class can overwrite a file or append information to a file. If you want to append information to a file, the constructor that receives two parameters (path and permission) must be used when instantiating an object of FileWriter type.

FileWriter Character Encoding

The ***FileWriter*** assumes that you want to encode the bytes to the file using the default character encoding for the computer your application is running on. This may not always be what you want, and you cannot change it!

If you want to specify a different character encoding scheme, don't use a ***FileWriter***. Use an **OutputStreamWriter** on a **FileOutputStream** instead. The **OutputStreamWriter** lets you specify the character encoding scheme to use when writing bytes to the underlying file.

➢ Statement

Create a program that captures people names and that stores them in an ArrayList.

Once the list of names has been saved within the ArrayList type object, use the **FileOutputStream** and ObjectOutputStream classes to serialize the object in a file called "serialization.txt". Print the message: "Object serialized" once the object has been serialized.

After you have serialized the object of type ArrayList, Deserialize it using the classes **FileInputStream** and **ObjectInputStream**. Following this, print the content (people names) of the deserialized object.

➢ Program Code

ObjectSerialization.java

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
```

```
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class ObjectSerialization {

	public static void main(String[] args) {
		ArrayList<String> names = new ArrayList<String>();
		names.add("Joseph");
		names.add("Nicolas");
		names.add("Jonathan");
		names.add("Jacob");

		// Serializing the names object to a serialization.txt file
		try(FileOutputStream fos = new FileOutputStream("serialization.txt"))
{
			ObjectOutputStream oos = new ObjectOutputStream(fos);

			oos.writeObject(names);
			oos.flush();
			System.out.println("\tObject serialized!\n");
			oos.close();

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

		// Deserializing the names object
		try(FileInputStream fis = new FileInputStream("serialization.txt")){
			ObjectInputStream ois = new ObjectInputStream(fis);

			ArrayList<String> deserializedNames = (ArrayList<String>)
                                                    ois.readObject();
			for (String name: deserializedNames) {
				System.out.println(name);
			}

			System.out.println("\n\tObject deserialized!");
		} catch (IOException | ClassNotFoundException e) {
			e.printStackTrace();
		}
	}
}
```

➢ Program execution

This program creates an object of type ArrayList where names of people are stored.

Once the object contains names of people (variables of type String), it is time to serialize it, so we will save it in a file with the name of "serialization.txt". At the end of the serialization, a message will be displayed on the screen with the following: "Object serialized".

Since we did the Serialization process, we will do the Deserialization process, where what is done is to read the serialized file "serialization.txt", save the content in an object of the same type (ArrayList) with the help of the ***readObject()*** method of the **ObjectInputStream** class. In the end, as we will have recovered our ArrayList type object, we will print the following message through console: "Object deserialized", as well as the names contained in the ArrayList object, just to verify that it was deserialized correctly.

```
Object serialized!

Joseph
Nicolas
Jonathan
Jacob

Object deserialized!
```

serialization.txt

```
¬í sr java.util.ArrayListx€Ò™Ça€ I sizexp    w    t Josepht Nicolast Jonathant Jacobx
```

➢ Conclusions

Serialization is a tool that Java provides us to have persistent data, this represents that the data we serialize will not be lost and will be available in the future, taking into account the coming and going of the software and the new devices.

In order to perform serialization in Java, we need to implement object flows. It is possible to read and write objects of a flow of any kind.

We use the **ObjectInputStream** class to deserialize data and, the **ObjectOutputStream** class, to serialize data.