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.

➢ 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!

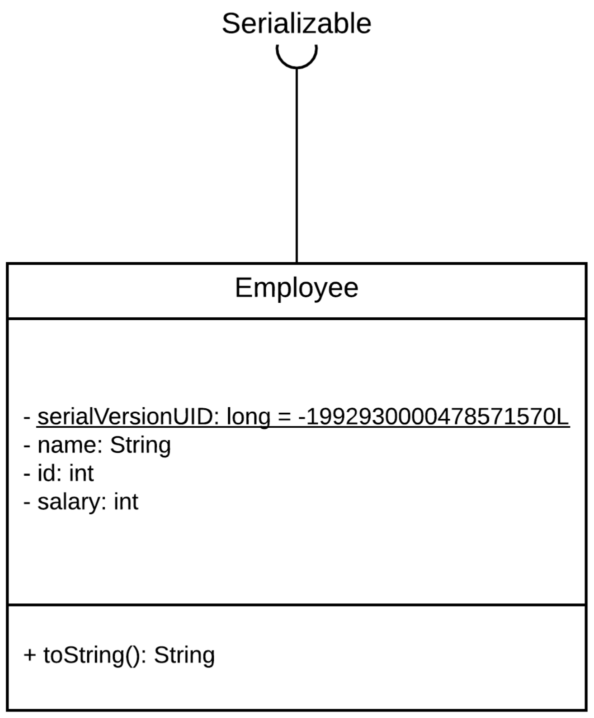
If you want to specify a different character encoding scheme, don't use a *FileWriter*. Use an **OutputStreamWriter** on a **FileOutputStream** instead. The **OutputStreamWriter** lets you specify the character encoding scheme to use when writing bytes to the underlying file.

➢ Statement

Create a class called **Employee** which implements the **Serializable** interface. This class must have an attribute of type String for the name, an attribute of type int for the id, a serialVersionUID attribute of type static final long for the universal version identifier and finally an attribute of type int for the salary, the latter attribute must use the ***transient*** keyword to not be included in the Serialization process.

Once you have the Employee class ready, create a class that contains a **public static void main** to create the instance of an object of type Employee, which will be serialized and deserialized and then displayed on the screen through the console. You will have to call your ***toString()*** method.

➢ Class design (UML)



➢ Program Code

Employee.java

```
import java.io.Serializable;

// Employee class needs to implement Serializable from being able to serialize an
object of this type
public class Employee implements Serializable {

	private static final long serialVersionUID = -1992930000478571570L;
//universal version identifier
	private String name;
	private int id;
	transient private int salary;	//transient means that won't be serialized

	public Employee(String name, int id, int salary) {
		this.name = name;
		this.id = id;
		this.salary = salary;
	}
```

```
        @Override
        public String toString(){
                return "Employee{name="+name+",id="+id+",salary="+salary+"}";
        }

        //Getters and Setters
        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public int getId() {
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }

        public int getSalary() {
                return salary;
        }

        public void setSalary(int salary) {
                this.salary = salary;
        }
}
```

OwnObjectSerialization.java

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

public class OwnObjectSerialization {

        public static void main(String[] args) {
                Employee employee = new Employee("John Wick", 1234, 20000);

                // Writing the employee serialized object to a file
                try(FileOutputStream fos = new
FileOutputStream("objectSerialized.dat")) {
                    ObjectOutputStream oos = new ObjectOutputStream(fos);
                    oos.writeObject(employee);

                    oos.flush();
                    System.out.println("\tObject serialized!\n");
                    oos.close();
             } catch (IOException e) {
                    e.printStackTrace();
             }

             // Deserializing the employee object
             try(FileInputStream fis = new
FileInputStream("objectSerialized.dat")){
                    ObjectInputStream ois = new ObjectInputStream(fis);

                    Employee deserializedEmployee = (Employee) ois.readObject();
                    System.out.println(deserializedEmployee.toString());

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

➢ Program execution

This program creates an object of type **Employee**, which implements the **Serializable** interface so that our object can be serialized.

When creating our object of type Employee we serialize it to a file called "objectSerialized.dat", followed by this we deserialize it to check if it was saved without any failure, and since we have recovered it, we use its *toString()* method to print its contents through console.

```
Object serialized!

Employee{name=John Wick,id=1234,salary=0}

Object deserialized!
```

<u>objectSerialized.dat</u>

```
¬í sr "createdclassserialization.EmployeeäW°¹;AsÎ   I  idL namet Ljava/lang/String;xp  Òt      John Wick
```

➢ 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.

<u>We use the **ObjectInputStream** class to deserialize data and, the **ObjectOutputStream** class, to serialize data.</u>