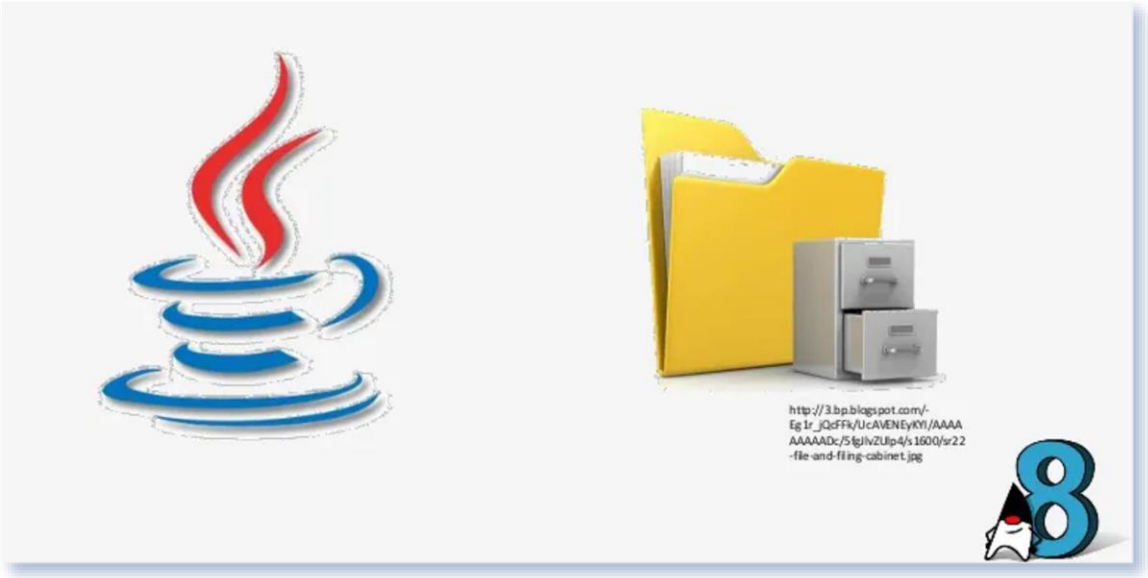


Persistence in Files. Java Streams.

Content

1. Data Persistence.

In computer science, persistence refers to the feature about the state of a system outliving (persisting more than) the process that created it. This is achieved in practice by storing the state as data in a computer data storage. Programs have to transfer data to and from storage devices and have to provide mappings from the native programming-language data structures to the storage device data structures. Picture editing programs or word processors, for example, achieve state persistence by saving their documents to files in an specific format.

Therefore, in order to provide persistency to our data we should know how the information is arranged inside the file and how to organize the files using file systems.

2. Files and File Systems

A file is a set of bytes that contains all the information related to a document, organized in a certain way. For example, an image usually contains each pixel represented by 3 bytes that indicate the amount of red, green, and blue color that each pixel has. Mixing these three values we obtain the final color. A text file, on the other hand, will be a sequence of character codes.

Broadly speaking, files can be classified into binary files (where information is stored as raw bytes) and text files (where every byte represents a position in a charset table, usually ASCII). Binary files, in general, are usually accompanied by a header information in which information about the content is provided. For example, in an image we usually have headers where the type of image is indicated (JPG, GIF, TIFF,...), the dimensions, the color palette -if it has one-, etc...

Figure 1: Text file vs Binary file

Today's computer storage media, either magnetic, optical, or any other technology, organize the information as a collection of bits readable sequentially. This can mean that accessing information, especially in huge media, can become a cumbersome task. To avoid this, these devices are mapped using a file system that allows the files to to be organized in an appropriate way.

A file system divides the physical medium into smaller parts that can be accessed independently, using subsets of variable sizes (varying from 512 bytes to 32kb) called clusters or sectors. Usually, you will also have an addressing system so that you can specify in which sector or sectors a certain file is stored. Also, to easily organize the files, it will allow the creation of a folders tree structure to storage and classify the files.

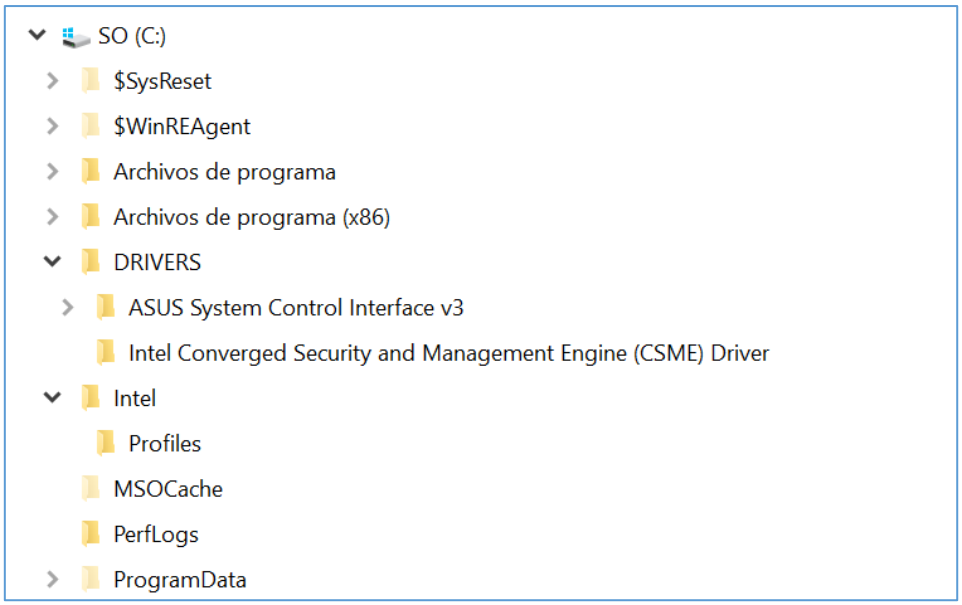


Figure 2: File System

To store our data in files from a program, we should consider then two aspects:

- Dealing with the data itself: that's made via the Stream class.
- Dealing with the file system: that's made via the File Class.

3. Streams

In Java, a stream can be defined as a sequence of bytes. It's called a stream because it is like a stream of water –or, in our case, bytes- that flows continuosly. It can flow from the source of data to the program (an input stream) or from the program to the destination data (an output stream).

3.1. Interacting with the user: standard streams

In the beggining of computer science, computers, far from having a graphical interface, offered a simple command terminal in text mode. This interface is still widely used in operating systems such as Linux, although it is also possible to use it in Windows, through the command terminal.

Figure 3: CMD - text command terminal

Every programming language provides support to use this text mode interface, and usually all of them perform this task by masking the input/output as virtual files. Indeed, the information that the user types in the terminal is considered an input file (standard input) while to display information in the terminal we must write to another file called standard output. There is a third virtual file, the standard error, which is used to display error messages, although it usually matches the standard output. Java provides support for this standard input/output through three Streams:

- `System.in`: It's the stream used to read data from the standard input.
- `System.out`: It's the stream used to display text on the screen of our terminal, and matches the standard output.
- `System.err`: This is used to display the error data produced by the program, matches the standard error.

Let's see the code to print output and error message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

And here is the code to get input from console.

```
int i=System.in.read(); //returns ASCII code of 1st character
System.out.println((char)i); //will print the character
```

3.2. The Scanner class

While printing information in console mode is pretty simple (just use `System.out.println`), reading information can be a tough process, especially if you are reading formatted data. For that purpose, Java offers a bunch of tools to do it, one of them is the `Scanner` class.

To get an instance of the `Scanner` class which reads input from the user, we need to pass the input stream to read from in the constructor of the `Scanner` class. If the input stream is `System.in`, the `Scanner` class will read from the keyboard.

```
Scanner myScanner = new Scanner(System.in);
```

Let's see a basic example of the use of `Scanner` where we are getting a single line from the user. Here, we are asking for a string through `myScanner.nextLine()` method.

```
import java.util.*;
public class ScannerExample {
public static void main( String args[] ) {
          Scanner myScanner = new Scanner(System.in);
          System.out.print("Enter your name: ");
          String name = myScanner.nextLine();
          System.out.println("Name is: " + name);
          myScanner.close();
          }
}
```

Depending on the type of data you want to read, you can use a different method. For example, to read an integer you'll use `nextInt()`, while to read an String you'll use `nextLine()`. Also, you can check if the next item complies your needs with the `hasNext()` –`hasNextInt()`, `hasNextString()`,...–. methods. You can check all the functionality of the class here:

https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

You can check this out alternatively:

https://www.javatpoint.com/Scanner-class

3.3. Activities

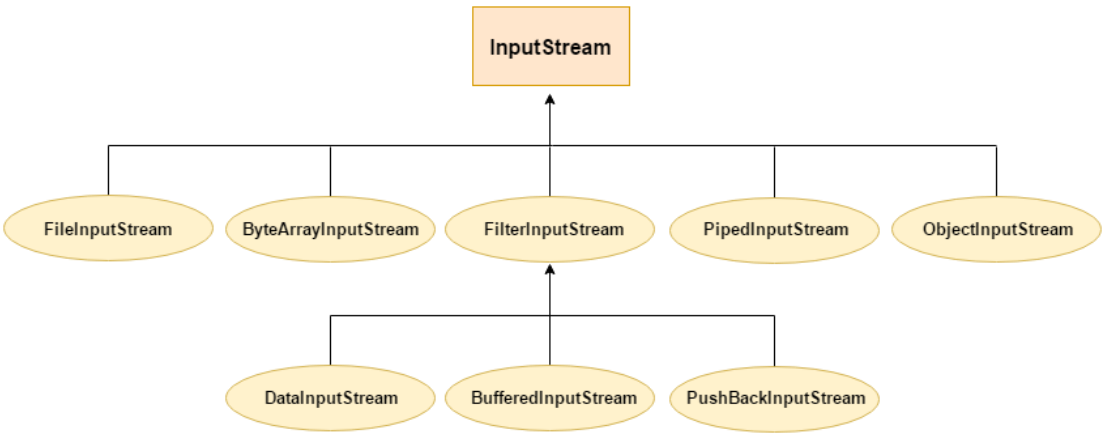
1. Write a java program that reads separately your name and surnames and welcomes you using your full name ("Hello, Jose Ramón García!")

2. Write a java program that reads a date, reading separately the day, the month and the year, and checking that the values for both month and day are correct (1...12, 1..31).
3. Rewrite the previous activity creating a class (MyDate, for example). Use properties (setters) to check the values.
4. Rewrite activity 3 to check the whole consistence of the date (not allowing February the 30th, for example), and informing the user if the year is a leap year (a leap year is a year with 366 days). The class created in activity 3 must be extended.

4. Binary Streams

4.1. Input Streams

`InputStream` class is an abstract class. It is the base class (superclass) of all input streams in the Java IO API, and defines the common functionality for every input stream. `InputStream` subclasses include the `FileInputStream`, `BufferedInputStream` and the `ObjectInputStream` among others, that we will see later.



Java `InputStream's` are used for reading byte-based data, one byte at a time. Here is a Java `InputStream` example:

```
InputStream is = new FileInputStream("c:\\data\\ input-text.
txt");
int data = is.read();
while( data != -1 ) {
     //do something with data...
     doSomethingWithData(data);
     data = is.read();
}
is.close();
```

This example creates a new FileInputStream instance. FileInputStream is a subclass of InputStream so it is safe to assign an instance of FileInputStream to an InputStream variable (the inputstream variable). As InputStream is an abstract class, it's not possible to create InputStream instances. But, as you can see in the example, an InputStream variable can point to any subclass instance.

From Java 7 you can use the try-with-resources construct to make sure the InputStream is properly closed after use. Here is a simple example that show how it works this construction:

```
try( InputStream inputstream = new FileInputStream("file.txt") ) {
    int data = inputstream.read();
    while(data != -1){
        System.out.print((char) data);
        data = inputstream.read();
    }
}
```

Once the executing thread exits the try block, the `InputStream` variable is closed.

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();
```

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.

If the `read()` method returns -1, the end of stream has been reached, meaning there is no more data to read in the InputStream. That is, -1 as int value, not -1 as byte or short value. There is a difference here!

read(byte[])

We usually won't read binary files byte by byte, because is quite unefficient. Instead, following the same method that it's used in file systems, we will read block of bytes. We have two methods available to do this:

- `int read( byte[] )`
- `int read( byte[], int offset, int length )`

The `read( byte[] )` method will attempt to read as many bytes into the byte array given as parameter as the array has space for. The `read( byte[] )` method returns an `int` telling how many bytes were actually read. In case less bytes could be read from the InputStream than the byte array has space for, the rest of the byte array will contain the same data as it did before the read started.

The `read( byte[], int offset, int length )` method also reads bytes into a byte array, but starts at offset bytes into the array, and reads a maximum of length bytes into the array from that position. Again, the `read( byte[], int offset, int length )` method returns an int telling how many bytes were actually read into the array, so remember to check this value before processing the read bytes. For both methods, if the end of stream has been reached, the method returns -1 as the number of bytes read.

Here is an example of how it could look to use the InputStream's `read(byte[])` method:

```
try (InputStream is = new FileInputStream("c:\\data\\input-text.txt"))
{
     byte[] data  = new byte[1024];
     int bytesRead = is.read( data );
     while( bytesRead != -1 ) {
       doSomethingWithData( data, bytesRead );
       bytesRead = is.read(data);
     }
}
```

mark() and reset()

The `InputStream` class has two methods called `mark()` and `reset()` which subclasses of `InputStream` may or may not support. If an `InputStream` subclass supports the `mark()` and `reset()` methods, then that subclass must override the `markSupported()` method to return `true`. If the method returns `false`, then `mark()` and `reset()` are not supported.

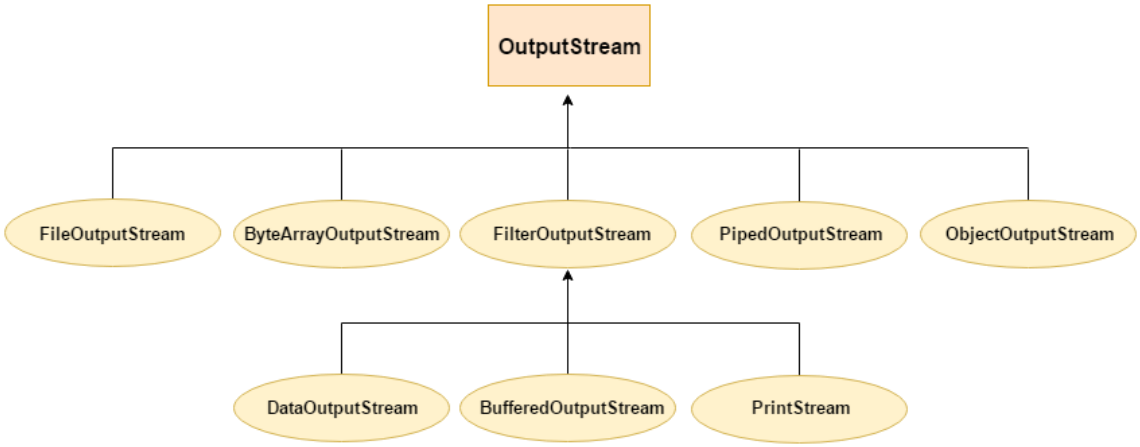
The `mark()` method sets internally a mark in the InputStream which shows the point in the stream to which data has been read so far. Once marked, we can continue reading data from the stream. If we want to go back to the point in the stream where the mark was set, we can call `reset()` on the `InputStream`. The `InputStream` then "rewinds" and go back to the mark, and start returning (reading) data from that point again. This will of course result in some data being returned more than once from the `InputStream`.

The methods `mark()` and `reset()` methods are typically used when implementing parsers. Sometimes a parser may need to read ahead in the

`InputStream` and if the parser doesn't find what it expected, it may need to rewind back and try to match the read data against something else.

4.2. Output Streams

Java applications use an output stream to write data to a destination, it may be a file, peripheral device or socket. 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.



write(byte)

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 be written. Only the first byte of the `int` value is written. The rest is ignored.

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.

Here is an `OutputStream` `write()` example:

```
try (OutputStream os = new FileOutputStream("c:\\data\\output.txt"))
{
     while ( hasMoreData() ) {
         int data = getMoreData();
         output.write(data);
     }
}
```

This OutputStream `write()` example first creates a `FileOutputStream` to which the data will be written. Then the example enters a `while` loop. The condition to exit the `while` loop is the return value of the method `hasMoreData()`. The implementation of `hasMoreData()` is not shown, but imagine that it returns true if there is more data to write, and false if not.

Inside the `while` loop the example calls the method `getMoreData()` to get the next data to write to the `OutputStream`, and then writes that data to the `OutputStream`. The `while` loop continues until `hasMoreData()` returns false.

write(byte[])

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 theOutputStream. The `write(byte[] bytes, int offset, int length)` method writes `length` bytes starting from index `offset` from the byte array to the `OutputStream`.

flush()

When referring to computers, a buffer is a temporary storage (either in memory or a storing device) where we leave the data until is needed for processing. This buffers are frequently used to speed up the input/output operations. Buffers are used widely, from printing jobs to streaming tasks.

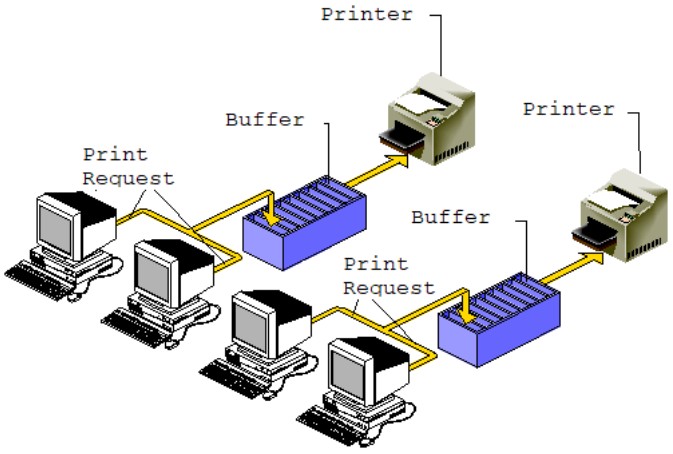


Figure 4: Buffering a printing job

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).

close()

Once you are done writing data to the `OutputStream` you should close it. You close an `OutputStream` by calling its `close()` method. Since the `OutputStream's` various `write()` methods may throw an `IOException`, you should close the `OutputStream` inside a `finally` block. Here is a simple `OutputStream.close()`:

```
OutputStream output = null;
try{
     output = new FileOutputStream("c:\\data\\output-text.txt");
     while( hasMoreData() ) {
          int data = getMoreData();
          output.write( data );
     }
}
catch( IOExcepction e ) {
      System.out.println( e.getMessage() );
}
finally {
     if( output != null ) {
          output.close();
     }
}
```

This simple example calls the `OutputStream.close()` method inside a `finally` block.

4.3. FileInputStream and FileOutputStream

The `FileInputStream` and `FileOuputStream` classes are the classes offered by Java to work directly with files, and inherit from `InputStream` and `OutputStream` respectively. They are read or write binary files, such as images, audio, video, etc. Although they can also be used to read text files, it is recommended to use instead the `FileReader` and `FileWriter` classes (which we will see later).

The following example makes an exact copy of a given file:

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main( String args[] ) {
        FileInputStream fIn = null;
        FileOutputStream fOut = null;
        try {
            fIn = new FileInputStream("D:\\testin.txt");
            fOut = new FileOutputStream("D:\\testout.txt");
            int i = 0;
            while( ( i = fIn.read() ) != -1 ) {
                  fOut.write( (byte)i );
            }
        }
        catch( IOException e ) {
            System.out.println( e.getMessage() );
        }
        finally {
            fIn.close();
            fOut.close();
        }
    }
}
```

4.4. Activities

1. Rewrite the example above to read and write 128-bytes blocks instead of reading and writing a single byte.

2. Write a program to detect the format of an image file. To detect the type of the file, the first bytes should be readed. Here you have a table with the header bytes for every format:

File Type	Header Bytes (Offset = 0)
.BMP	42 4D
.GIF	47 49 46 38 39 61 / 37 61
.ICO	00 00 01 00
.JPEG	FF D8 FF
.PNG	89 50 4E 47

You can get more information about file headers here:

https://en.wikipedia.org/wiki/List_of_file_signatures

https://www.welivesecurity.com/la-es/2015/10/01/extension-de-un-archivo-cabeceras/

3. Write a program to read the header of a BMP file and report its size, width, height and if the bits per pixel. You must read the reader as a block. Here you can get more information about the BMP header:

   http://www.fastgraph.com/help/bmp_header_format.html

4. Write the test class (using JUnit) for program 4.4.3.

5. Character Streams

5.1. FileReader and FileWriter

Java byte streams are used to perform input and output of 8-bit bytes, whereas Java character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams, the most frequently used classes are `FileReader` and `FileWriter`. Though internally `FileReader` uses `FileInputStream` and `FileWriter` uses `FileOutputStream`, here the major difference is that `FileReader` reads two bytes at a time and `FileWriter` writes two bytes at a time.

The major difference is obviously the `FileWriter` write methods:

Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.

5.2. BufferedReader and BufferedWriter

Java `BufferedReader` class is used to read text from a character-based input stream. It can be used to read data line by line by `readLine()` method. It makes the performance fast. It inherits from the `Reader` class.

The `Java.io.BufferedReader` class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. Following are the important points about `BufferedReader`:

- The buffer size may be specified, or the default size may be used.
- Each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.

5.3. PrintWriter

The class `PrintWriter` provides an efficient way to write formatted information to a text file. For example, you can easily write new lines just by using the `println` method. Combining the class `PrintWriter` with other classes, is possible not only to overwrite information in a text file but also to append it, as we can see in the following example:

```
import java.io.*;
public class MyPrintWriter {
    public static void main( String[] args ) {
        PrintWriter printWriter = null;
        try {
            printWriter = new PrintWriter(new BufferedWriter(
                    new FileWriter( "example.txt", true )));
            printWriter.println("Hello!");
            printWriter.println("and...");
            printWriter.println("see you soon!");
        }
        catch ( IOException e ) {
            e.printStackTrace();
        }
        finally {
            if ( printWriter != null ) {
                printWriter.close();
            }
        }
    }
}
```

Note the `true` value in the `FileWriter` constructor, indicating that the info should be append to the file *ejemplo.txt*.

5.4. Activities

1. Write a program to create a notepad. The program will ask for sentences to the user and write it to a file. The file can exist or not, if it does, the user should be asked whether to overwrite the information or append it. Lines will be numbered sequentially. You may need to read point 6 first.
2. Write a program to sort alphabetically two files previously sorted (the name of the files will be provided by the user). The destination file will be "sorted.txt". For example, if the file 1 contains the lines:

Car
Horse
Ship

And the file 2 contains:

Alley
Cat
Horn
Show
Tree

The file "sorted.txt" will contain:

Alley
Car
Cat
Horn
Horse
Ship
Show
Tree

3. Write a program to show the content of the file created in the exercise one (or any other file) on the screen. The program must stop every 23 lines, awaiting for the user to press 'space' (or newline, or any other key) in order to allow the whole content of the file to be readed.
4. Write a program to search for a string inside a file text. It must print out every line containing the string, indicating the line number. File existence must be controlled.

6. File System Management: the File class

Streams allow us to deal with the content of the files, independently of the type of content. But files (the containers) usually reside in a file system, to deal with files itself (to rename them, move them from one directory to another, to create or delete files, …) we use the File class.

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Fields

Modifier	Type	Field	Description
static	String	pathSeparator	It is system-dependent path-separator character, represented as a string for convenience.
static	char	pathSeparatorChar	It is system-dependent path-separator character.
static	String	separator	It is system-dependent default name-separator character, represented as a string for convenience.
static	char	separatorChar	It is system-dependent default name-separator character.

Constructors

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Useful Methods

Modifier and Type	Method	Description
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.

boolean	delete()	Deletes the file or directory denoted by this abstract pathname.
boolean	exists()	Tests whether the file or directory denoted by this abstract pathname exists.
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
boolean	mkdir()	It creates the directory named by this abstract pathname.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.

7. Serialization

So far, we have seen how to write both binary or text information to a file. However, the type of information we deal with when writing an application is not text neither bytes but objects, especially when it comes to an OOP language like Java. Thus, we need to figure out how to write this objects to a file. This process is called serialization.

According to Wikipedia, "In computing, serialization (US spelling) or serialisation (UK spelling) is the process of translating a data structure or object state into a format that can be stored (for example, in a file or memory data buffer) or

transmitted (for example, over a computer network) and reconstructed later (possibly in a different computer environment)".

In Java, to serialize an object, we need:

- Implement the interface 'Serializable'. This interface is a 'marker' interface, that is, it has no members neither methods associated.
- Use an special streams called ObjectInputStream and ObjectOutputStream.

Let's see how it work with an example:

```
import java.io.Serializable;
import java.io.File;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;
import java.io.FileNotFoundException;

class City implements Serializable {
    protected String code;  // properties should be used here
    protected String name;

    public City( String _cod, String _name ) {
        code = _cod;
        name = _name;
    }

    public void write() {
        System.out.println(code + ": " + name);
    }
}

public class Serialize {
    public static void main( String[] args )throws
FileNotFoundException, IOException {
        ObjectOutputStream objectsFile = new ObjectOutputStream(
                new FileOutputStream( new File("cities.dat")));
        objectsFile.writeInt(2); // Number of objects actually written
        City aCity = new City("A", "Alicante");
        objectsFile.writeObject( aCity );
        aCity = new City("Gr", "Granada");
        objectsFile.writeObject( aCity );
        objectsFile.close();
    }
}
```

As it can be seen in the example, no extra functionality must be added to the class in order to be serializable.

When reading, the process is exactly the same, but we should control one more exception, `ClassNotFound`. This exception will be raisen when the actual type of object to be read doesn't match the object found in the file.

```
public class Unserialize {
    public static void main( String[] args ) throws
FileNotFoundException, IOException, ClassNotFoundException {
        ObjectInputStream objectsFile = new ObjectInputStream(
                new FileInputStream( new File("cities.dat") ));
        int numObjects = objectsFile.readInt();
        for ( ; numObjects > 0 ; numObjects-- ) {
            City aCity = (City) objectsFile.readObject();
            aCity.write();
        }
    }
```

As we can see at the example above, we need to write not only the objects but the number of them stored in the file, in order to avoid an `IOException` to be raisen. Also, we can see that different types of objects can be stored in the same file.

To conclude, collections in Java (like `List<>`, for example) are also objects, so it's perfectly possible to directly store the whole collection in an `ObjectStream`.

7.1. Activities

1. Write a program to create a contact list. First of all, you will need a class Contact, with the following information: name, surname, e-mail, phone number, description. The program must permit to create new contacts, show the current list of contacts and search for a contact by its full name or phone. Data must be stored and retrieved in a file named contacts.obj. If it doesn't exist, it must be created.

8. XML Parsing

As you probably know (because you are due to have taken the module 'Markup languages and information management systems'), the XML ("eXtensible Markup Language') format is a widely used markup language, especially for information interchange between systems.

There are at least two ways to parse (analyze) an XML format from Java:

- Using SAX (Simple API for XML), that allows to read an XML file sequentially, without storing the information in memory.
- Using DOM (Document Object Model), that builds in memory the XML tree. That approach is more useful in many cases but it takes more resources, especially memory.

For the following examples we will use this XML file (related with activity 7.1):

```
<contactlist>
	<contact id='c001'>
		<name>Jose</name>
		<surname>García</surname>
		<emails>
			<work>jrgarcia@iesmarenostrum.com</work>
		</emails>
		<phones>
			<cell>616116611</cell>
		</phones>
	</contact>
	<contact id='c002'>
		<name>Antonio</name>
		<surname>Luque</surname>
		<emails>
			<home>mymail@protonmail.com</home>
			<work>antonio@iesmarenostrum.com</work>
		</emails>
		<phones>
			<cell>626226622</cell>
			<home>965223344</home>
		</phones>
	</contact>
	<contact id='c003'>
		<name>Ana</name>
		<surname>Pérez</surname>
		<phones>
			<cell>616116611</cell>
			<work>965119192</work>
		</phones>
	</contact>
</contactlist>
```

8.1. SAX

SAX (Simple API for XML) is an event-driven algorithm for parsing XML documents. SAX is an alternative to the Document Object Model (DOM). Where

the DOM reads the whole document to operate on XML, SAX parsers read XML node by node, issuing parsing events while making a step through the input stream. SAX processes documents state-independently (the handling of an element does not depend on the elements that came before). SAX parsers are read-only.

SAX parsers are faster and require less memory. On the other hand, DOM is easier to use and there are tasks, such as sorting elements, rearranging elements or looking up elements, that are faster with DOM.

A SAX parser comes with JDK, so there is no need to download a dependency.

The necessary steps to write our own SAX parser are:

- Create an object `SAXParserFactory`
- From the previous object, create a `SAXParser`
- Create our own event handler, extending the class `DefaultHandler`. This is the critical steps, because in this handler we should define the actions to be taken when the parser founds an specific tag. There are a lot of events that can be overloaded (such as `startDocument` or `endDocument`) but the basic events to be rewritten are:
  - `startElement` (tag opening)
  - `characters` (content of a tag, usually CDATA)
  - `endElement` (tag closing)
- Finally, call the method `parse` from `SAXParser`, providing the name of the file to parse and our event handler.

Let's see an example on how to show the content of the *contacts.xml* file:

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

class myXMLContactsHandler extends DefaultHandler {
    protected String tagContent;

    // Tag opening found
    //
    public void startElement(String uri, String localName,
           String qName, Attributes attributes) throws SAXException {
```

```
        if ( qName.equals("contact") ) {
            System.out.println( "ID: " + attributes.getValue("id"));
        }
    }

    // Tag content, usually CDATA
    //
    public void characters( char ch[], int start, int length )
            throws SAXException {
        tagContent = new String( ch, start, length );
    }

    // Tag ending
    //
    public void endElement(String uri, String localName, String qName)
            throws SAXException {
        if ( !qName.isBlank() ) {
            if ( !qName.equals( "contact" ) ) {
                System.out.println(" " + qName + ": " + tagContent);
            }
        }
    }
}

public class Main {
    public static void main( String[] args ) {
        try {
            SAXParser saxParser = SAXParserFactory.
                  newInstance().newSAXParser();
            saxParser.parse("contacts.xml", new
                  myXMLContactsHandler());
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }
}
```

8.2. Activities

1. Rewrite the previous code just to show the name and surname of the contacts (all together) with a phone number. The phone number will be chosen in this order: cell, work, home.

2. Write an utility to allow the user to change the format of the file created in activity 7.1 from obj to XML and reverse. Check point 9 to know how to reuse your code. You can use this link as a guide: https://zetcode.com/java/sax/

8.3. DOM

Document Object Model (DOM) is a standard tree structure, where each node contains one of the components from an XML structure. Element nodes and text nodes are the two most common types of nodes. With DOM functions we can create nodes, remove nodes, change their contents, and traverse the node hierarchy.

Figure 5: DOM representation of the contacts file

DOM is part of the Java API for XML processing (JAXP). Java DOM parser traverses the XML file and creates the corresponding DOM objects. These DOM objects are linked together in a tree structure. The parser reads the whole XML structure into the memory.

The following is a list of some important node types:

Type	Description
Attr	represents an attribute in an Element object
CDATASection	escapes blocks of text containing characters that would otherwise be regarded as markup
Comment	represents the content of a comment
Document	represents the entire HTML or XML document
DocumentFragment	a lightweight or minimal Document object used to represent portions of an XML Document larger than a single node
Element	element nodes are basic branches of a DOM tree; most items except text are elements
Node	the primary datatype for the entire DOM and each of its elements
NodeList	an ordered collection of nodes
Text	represents the textual content (called character data in XML) of an Element or Attr

To rewrite the example in section 8.1, we should do the following:

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.traversal.DocumentTraversal;
import org.w3c.dom.traversal.NodeFilter;
import org.w3c.dom.traversal.NodeIterator;
import org.xml.sax.SAXException;

public class XmlDomReadExample {

    public static void main( String[] args )
          throws ParserConfigurationException,
                 SAXException, IOException {

        DocumentBuilderFactory factory = DocumentBuilderFactory.
                newInstance();
        DocumentBuilder loader = factory.newDocumentBuilder();
        Document document = loader.parse("contacts.xml");

        DocumentTraversal traversal = (DocumentTraversal) document;

        NodeIterator iterator = traversal.createNodeIterator(
                document.getDocumentElement(),    NodeFilter.SHOW_TEXT,
null, true);

        for ( Node n = iterator.nextNode(); n != null;
                  n = iterator.nextNode() ) {

            String text = n.getTextContent().trim();

            if ( !text.isEmpty() ) {
                System.out.println( text );
            }
        }
    }
}
```

This code uses the class `NodeIterator` to sequentially iterate over every node in the XML file, in a similar way to SAX. But it is possible to search for an specific node or element once the file has been readed, as it can be seen in the following example:

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class XmlDomReadExample {

    public static void main( String[] args )
         throws ParserConfigurationException,
                SAXException, IOException {

         DocumentBuilderFactory factory = DocumentBuilderFactory.
                newInstance();
         DocumentBuilder loader = factory.newDocumentBuilder();
         Document doc = loader.parse("contacts.xml");

         doc.getDocumentElement().normalize();

         System.out.println("Root element: " + doc.getDocumentElement().
                getNodeName());

         NodeList nList = doc.getElementsByTagName("contact");

         for ( int i = 0; i < nList.getLength(); i++ ) {

             Node nNode = nList.item(i);

             System.out.println("\nCurrent Element: " + nNode.
                     getNodeName());

             if ( nNode.getNodeType() == Node.ELEMENT_NODE ) {

                 Element elem = (Element) nNode;

                 String uid = elem.getAttribute("id");

                 Node node1 = elem.getElementsByTagName("name").item(0);
                 String fname = node1.getTextContent();

                 Node node2 = elem.getElementsByTagName("surname")
                      .item(0);
                 String lname = node2.getTextContent();
```

```
                System.out.printf("User id: %s%n", uid);
                System.out.printf("First name: %s%n", fname);
                System.out.printf("Last name: %s%n", lname);
            }
        }
    }
}
```

8.4. Activities

1. Rewrite activies in point 8.2 using DOM

9. To learn more

9.1. Java GUI's

Swing

In this unit, all the activities proposed are console-oriented applications. In Java, there are several options to create Graphical User Interfaces. One of the more popular options is Swing.

Swing is a GUI widget toolkit for Java. It is part of Oracle's Java Foundation Classes (JFC) – an API for providing a graphical user interface (GUI) for Java programs.

Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT). Swing provides a look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and therefore are platform-independent.

You can find more information about creating a project with IntelliJ IDEA below:

https://tutorials.tinyappco.com/Java/SwingGUI

https://zetcode.com/javaswing/

https://www.javatpoint.com/java-swing

Some useful information can also be found here:

- Adding a menu bar: https://www.javatpoint.com/java-jmenuitem-and-jmenu
- Showing a message box: https://www.roseindia.net/java/example/java/swing/MessageBox.shtml
- Changing the visual style: https://www.geeksforgeeks.org/java-swing-look-feel/
- Select a file with a dialog: https://docs.oracle.com/javase/7/docs/api/javax/swing/JFileChooser.html
- Filling a combo box with a set of strings (for example, the values returned by a query): https://stackoverflow.com/questions/14078840/jcombobox-modify-index-using-defaultcomboboxmodel

JavaFX

As Swing, JavaFX is a GUI for Java applications. JavaFX is a software platform for creating and delivering desktop applications, as well as rich web applications that can run across a wide variety of devices. JavaFX has support for desktop computers and web browsers on Microsoft Windows, Linux, and macOS, as well as mobile devices running iOS and Android.

JavaFX was intended to replace Swing as the standard GUI library for Java SE, but it has been dropped from new Standard Editions while Swing and AWT remain included, supposedly because JavaFX's marketshare has been "eroded by the rise of 'mobile first' and 'web first applications." With the release of JDK 11 in 2018, Oracle made JavaFX part of the OpenJDK under the OpenJFX project, in order to increase the pace of its development. Oracle support for JavaFX is also available for Java JDK 8 through March 2025.

An annex providing information about JavaFX is provided within this course. You can also find more information here:

https://www.jetbrains.com/help/idea/javafx.html

https://www.javatpoint.com/javafx-tutorial

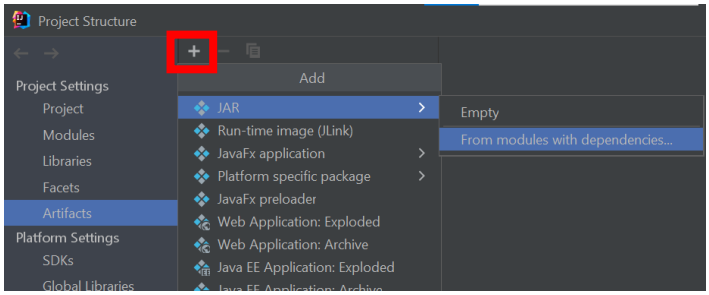
https://zetcode.com/gui/javafx/

https://blog.jetbrains.com/idea/2021/01/intellij-idea-and-javafx/ (working with IntelliJ IDEA and Maven/Gradle)

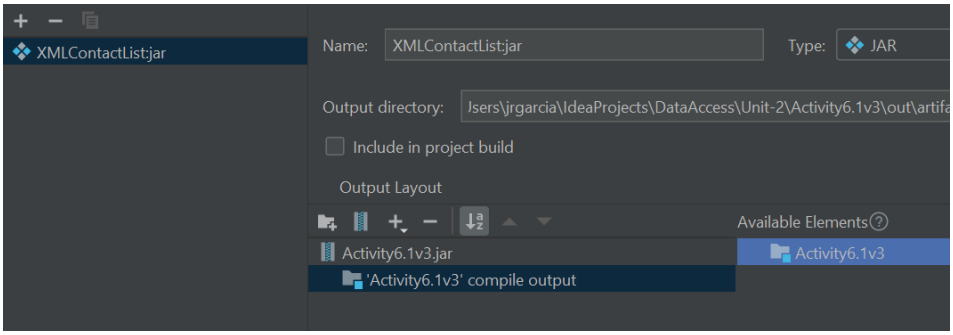
9.2. Component reutilization

One of the main goals in software developing is component reutilization. In Java, to be able to reuse a class previously developed, you must create a JAR file with the class or classes you want to reuse and then, add it to the new project.

Suppose you want to extend, in activity 8.2 (XML contact list), the functionality of the class you created in activity 7.1 (contact list). The first step would be to load this last project and, in the *File/Project Structure* dialog, choose the option *Artifacts* and create a new artifact:

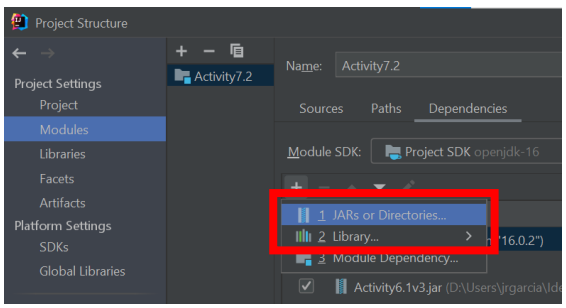


We will create a JAR file for the current module with the default options. There is no need to choose a Main function, because we don't want to execute this JAR file. You can customize the name of the file.



To create the JAR file, simply pick the *Build/Build Artifacts* option. The JAR file will be created in the output directory.

To use this classes in another project, open the new project, open the Project Structure dialog and, in the Modules section, add the JAR file previously created:



10. Bibliography

https://www.jetbrains.com/help/idea/discover-intellij-idea.html

https://docs.oracle.com/en/java/javase/16/docs/api/index.html

https://www.javatpoint.com/java-tutorial

https://zetcode.com/all/#java