DAM. UNIT 1. ACCESS TO FILES. PART 1. INTRO, JAVA REVIEW AND BASIC FILE ACCESS

DAM. Acceso a Datos (ADA) (a distancia en inglés)

Unit 1. ACCESS TO FILES

Part 1. Intro, Java review and basic file access

Abelardo Martínez

Year 2023-2024

1. Introduction

In computing, from the goal of automated processing of information, a need arises: to make data persist beyond the execution of the process or application that created it. Programs use variables to store information: the input data, the calculated results and values generated during the calculation. All this information is ephemeral: when the program is finished, it all disappears. **Applications must be able to store and retrieve data, and therefore data must be persistent**.

This data can persist over different data management and data handling systems, such as files, databases (whether relational, non-relational, native, etc.) or even other processing systems, and not only on storage devices.

We will see that applications dealing with data are faced with the difficulty of how to store and re-access data that are in different information systems. These types of applications that work with data that have to last create the obligation to use and know different access techniques for each of the management systems that are used.

The initial aim of this module is to conceptually approach the problem of data persistence in applications, analysing different situations where it is necessary for data to be stored and retrieved considering various approaches.

2. File System Management

2.1. File systems

When talking about computers, it is often said that, in the final analysis, what is in a computer are ones and zeros. Over time, different layers of software were added "on top" of these zeros and ones to make computers easier to use.

A file is an abstraction of the operating system for generic data storage. The part of the operating system that handles files is called the "file system".

Traditional file systems are responsible for organising, storing and naming (identifying by name) files stored on permanent storage devices, such as hard disks, solid state USB sticks, DVD's, etc.

Today, the most popular file systems are:

File System	Application
ext4	Linux operating systems
NTFS	Windows-based systems
ISO9660, UDF	Optical devices such as CD's and DVD's

Although each file system offers its own view of the data stored on a disk and manages and arranges it in its own way, they all share some general aspects:

- Files are usually organised in hierarchical structures of directories (also called folders). These directories are containers for files (and other directories) that help organise the data on the disk.
- The name of a file is related to its position in the directory tree containing it, which allows not only to uniquely identify each file, but also to find it on the disk from its name.
- Files usually have a series of associated metadata, such as their creation date, the date of last modification, their owner or the permissions that different users have over them (read, write, etc.). This turns the file system into a database of the contents stored on the disk that can be consulted according to multiple search criteria.

2.2. Paths: file naming

In today's computer systems, in which a single computer can have more than a million files, it is essential to have a system that allows an efficient location management, so that users can move comfortably between so many files. Most file systems have incorporated hierarchical containers that act as directories, facilitating the classification, identification and location of files. Directories have become popular under the graphical version of folders.

We must also bear in mind that the excessive need for storage space has led OS's to work with a large number of devices and to allow remote access to other file systems distributed over the network.

The name of a file is known as its "path" (the closest translation in Spanish is "ruta", but even in Spanish we use its English name). It should be noted that the path of a file changes depending on the file system used.

For example, let's look at the full name of a file (its path) in a computer with Unix like system:

```
/home/admin/documents/AccessFiles/unit1.txt
```

On this computer, there cannot be another file with the same name (path full name), although there can be many files with the short name "unit1.txt".

2.2.1. Linux

In order to manage such a variety of file systems, some operating systems such as Linux or Unix take the strategy of unifying all the systems into a single one, to achieve a unified form of access and with a single hierarchy that facilitates reference to any of its components, regardless of the file system in which they are actually located. In Linux, whatever the device or the actual remote system where the file will be stored, the path will always have the same form.

On Linux operating systems, **all files are contained in a root directory or in directories hanging from it**. For example:

```
/home/admin/documents/AccessFiles/unit1.txt
```

2.2.2. Windows

On the other hand, the strategy of other OSs such as Windows is to keep each of the systems and devices to which it has access very distinct. In order to distinguish the system to which reference is to be made, Windows uses a specific name that is incorporated into the path of the element to be referenced. Although Microsoft has clearly opted for the UNC convention, the evolution of this operating system, which has its origin in MS-DOS, has led it to coexist with another convention that is also very widespread. We refer to the identification of devices and systems with a letter of the alphabet followed by a colon. This drive may itself contain files or directories.

Traditionally, moreover, in the Windows world, filenames usually have an extension (typically a "." and three letters) that helps to identify the type of content stored in that file.

Below we illustrate both conventions with an example. We have marked in bold the specific name that identifies the specific file system or device:

```
Option a --> F:\Documents\AccessFiles\Unit1.txt
Option b --> \\ServerHS\Documents\AccessFiles\Unit1.txt
```

2.2.3. Examples

In both systems we can locate a file by its name (its path). Let's look at a couple of examples:

Linux:

```
/home/admin/myfile
```

Under the root directory (/), there is a directory /home/, inside which there is a directory /home/admin/ (short name "admin"), inside which there is a file /home/admin/myfile, short name "myfile".

Windows:

```
C:\My documents\myfile.txt
```

Under the C drive, there is a directory C:\My documents, inside which there is a file C:\My documents\myfile.txt, short name "myfile.txt", which has a typical extension of the old Windows systems of three letters ("txt").

As you can see in these examples, paths look very different depending on the operating system. In **Linux**, everything hangs from the root directory and the **/ character** is used to separate directories and files from each other. In **Windows**, on the other hand, you start from a drive letter and separate directories with the **\ character**.

2.3. Relative and absolute paths

So far we have only seen examples of **absolute paths**, which are those that allow a file and its location to be uniquely identified without the need for further data. A **relative path** specifies the path to a file from a reference directory, which is known as the working directory.

Let's look at some examples:

- If the working directory is /home/admin/, the relative path my_file identifies the file /home/admin/my_file.
- If the working directory is /home/admin/, the relative path my_project/my_file identifies the file /home/admin/my_project/my_file.
- If the working directory is /home/admin/, the relative path ../ada/my_file identifies the file /home/ada/my_file. The ../ directory is an alias to the parent directory of the directory it succeeds.
- If the working directory is /home/admin/, the relative path ./my_project/my_file identifies the file /home/admin/my_project/my_file. The ./ directory is an alias of the working directory.

To distinguish a relative path from an absolute path, the key is to realise that a relative path is not preceded by the root of the file system (/ on Linux) or by a drive letter on Windows (C:\ for example).

3. Files: Types and Access

We can differentiate between various types of files according to the following criteria:

3.1. Depending on the content

- Text (character) files.
- Binary (byte) files.

3.1.1. Text (character) files

Although both types end up storing sets of bits in the file, in the case of text files we will only find characters that can be viewed using any text editor. Binary files, on the other hand, store sets of bytes that can represent anything: numbers, images, sound, etc.

Text files are those that only contain text inside and, therefore, we can view and/or manipulate their content using a basic tool of the operating system: the text editor.

At this point it is necessary to distinguish between a **text editor** and a **word processor**. A text editor is a programme that allows you to view and/or edit text files, i.e. files containing only text, while a word processor allows you to view and/or edit files in which, in addition to text, there may be other types of information that the word processor has to interpret: fonts, images, etc.

3.1.2. Binary (byte) files

In contrast, we also have binary files, which are those that cannot be viewed and/or edited with a text editor, and can be of many different types: Spreadsheets (.ods, .xls, .xlsx.) images (.jpg, .png, tiff), sounds (.mp3, .wav), videos (.mp4, .avi, .mkv), word processors (.odt, .doc, .docx), etc.

3.2. Depending on the access mode

For both text files and binary files, there are two ways to access the data in them:

- Sequential.
- Direct access (or random access) files.

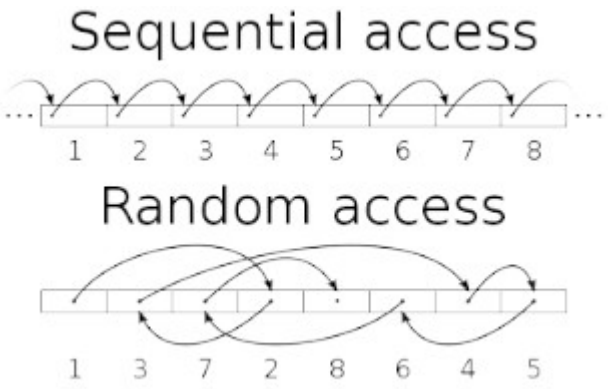
3.2.1. Sequential access

In sequential mode the information in the file is a sequence of bytes so that in order to access the i-th byte, all previous bytes must have been previously accessed. That is, it is possible to read or write a certain amount of data always starting from the beginning of the file.

It is also possible to add data starting from the end of the file (the whole file will have to be traversed, which means low efficiency). Therefore, we need tools that allow direct access to any given position in a file. To do this it is necessary to use random access files.

In Java, sequential access can be:

- Binary → FileInputStream and FileOutputStream
- Characters → FileReader and FileWriter

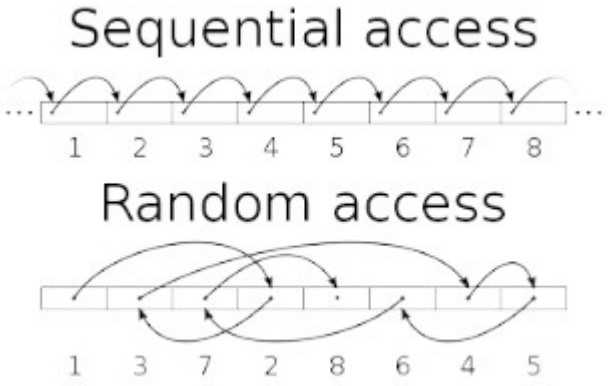


3.2.2. Random access

On the contrary, the direct access mode allows us to directly access the information of the i-th byte without the need to go through all of them. That is, they allow you to go anywhere in the file to update certain values.

The distinction, therefore, between sequential access files and random access files is merely a distinction between which operations are used to access the data.

In Java, for random access we will use RandomAccessFile.



3.3. File operations

The operations we can perform on any file (regardless of its type):

- File creation
- Opening the file
- Closing the file
- Reading data from the file
- Writing data to the file

Once opened, the operations that can be carried out are classified as follows:

- Additions
- Deletions
- Modifications
- Queries
- Any other operation, such as searches or sorting, is made up of a combination of the above.

4. Java File Management

It becomes apparent that each operating system can organise its files in different ways and use different systems to identify their location. The consequence of this is that there is no standard way to reference a file, but it depends on each operating system.

In order to overcome this dependency, Java has developed the **File class**, which allows the reference of a file to be abstracted independently of the operating system's own notation.

Instances of type File can represent either a file or a directory (or folder). In either case, they allow us to interrogate them to obtain information about the element represented. Thus, for example, we can know whether it is a file or a directory, or obtain information about its name in any of its forms (absolute or relative), its size, the date of creation, etc.

File does not have any utility to obtain a sorted list. Sorting will be achieved using the Java utilities in the System class, to sort collections using a Comparator.

4.1. The File class. General aspects

In Java, the "File" class is basically used to manage the file system. It is a class that must be understood as a reference to the path or location of files in the system. It does NOT represent the content of any file, but the path of the system where they are located. As it is a path, **the class can represent either files, folders or directories**.

An object of the File class can represent:

- A particular file
- A series of files located in a directory
- A new directory to be created

Objects of the File class represent File System paths. If we use a class to represent paths, we achieve total independence with respect to the notation that each operating system uses to describe them. Remember that Java is a multiplatform language and, therefore, it may happen that we have to make an application without knowing the OS where it will be executed.

The strategy used by each OS does not affect the functionality of the File class, since this, in collaboration with the virtual machine, will adapt the calls to the host OS transparently to the programmer, that is, without the programmer having to indicate or configure anything.

Instances of the File class are tightly bound to the path with which they are created. This means that instances during their entire lifecycle will only represent a single path, the one associated to them at the time of creation. The File class has no method or mechanism to modify the associated path. If new paths are needed, a new instance must always be created and it will not be possible to reuse those already created by linking them to different paths.

In implementations so close to the OS, Java programmers have to make an effort to make the implemented applications independent of the platforms where they will run. Care must therefore be taken to use parametrisation techniques that avoid writing the paths directly to the code, so that when moving the applications from platform to platform, only the paths in the configuration system need to be modified.

The **File class** encapsulates virtually all the functionality needed to manage a file system organised in a directory tree. It is a complete management that **includes**:

1. Manipulation and consultation functions of the hierarchical structure itself (creation, deletion, obtaining the location, etc., of files or folders).

2. Functions for manipulating and querying the particular characteristics of the elements (names, size or capacity, etc.).
3. Functions for manipulating and querying attributes specific to each operating system and which, therefore, will only be functional if the host operating system also supports the functionality. We refer, for example, to write permissions, execution permissions, hiding attributes, etc.

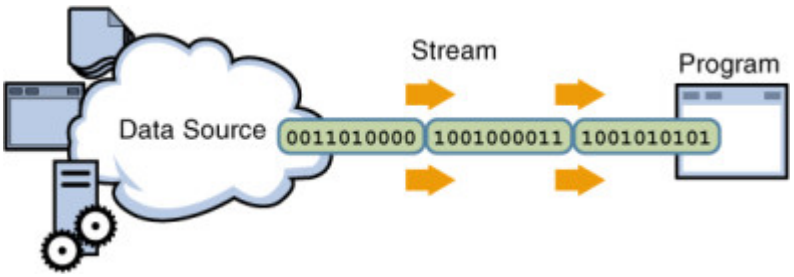
4.2. File operations. INPUT/OUTPUT Streams

Files are structured data warehouses and it could be considered as an exchanging data resources between 2 systems: one volatile (RAM memory) and another permanent (storage devices).

The Java I/O system has different classes defined in the java.io package. It uses the Stream abstraction to handle communication between a source and a destination. This source can be:

- The hard disk.
- The main memory.
- A network location.
- Another program.

Any program that needs to obtain or send information to any source needs to make use of a stream. By default, the information in a stream will be written and read serially.



Two types of streams are defined:

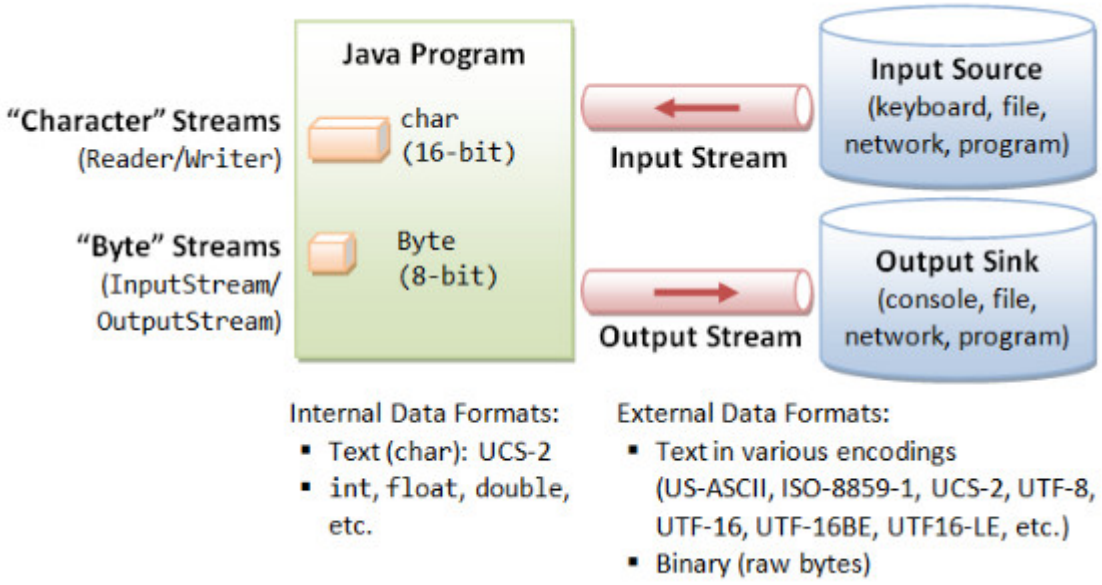
a) Byte streams (8 bits)

- Performs byte I/O operations.
- Oriented to binary data operations.
- All types of byte streams descend from InputStream and OutputStream.

b) Character streams (16-bit)

- Perform character I/O operations.
- Defined in Reader and Writer classes.

- Supports 16-bit Unicode characters.

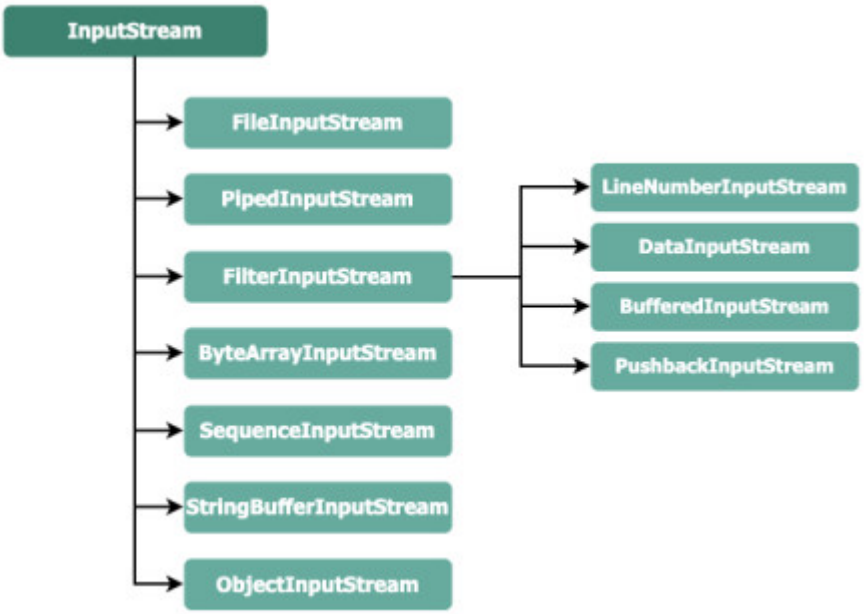


4.2.1. Byte Streams

a) InputStream

Represents classes that produce inputs from different sources: a Byte Array, a String object, a file, a pipe, a stream, a network connection, etc. The following table shows the different classes that inherit from InputStream:

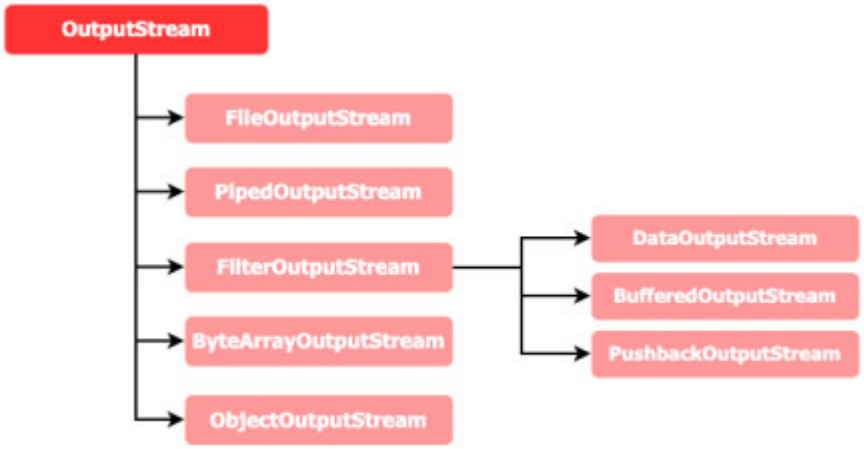
Class	Description
ByteArrayInputStream	Allows the use of memory buffer space.
StringBufferInputStream	Converts a String to an InputStream.
FileInputStream	Used to read data from a file.
PipedInputStream	Implements the pipeline concept.
FilterInputStream	Provides additional functionality to other streams.
SequenceInputStream	Concatenates two or more InputStream objects.



b) OutputStream

Represents the output streams of a program. The following table shows the different classes that inherit from OutputStream:

Class	Description
ByteArrayOutputStream	It creates a storage space in memory. All data sent to this stream is stored in it.
FileOutputStream	It is used to write data to a file.
PipedOutPutStream	Any information written to an object of this class shall be used as input to a PipedInputStream associated to it. It implements the pipeline concept.
FilterOutputStream	Provides additional functionality to other OutputStreams.



4.2.2. Character Streams

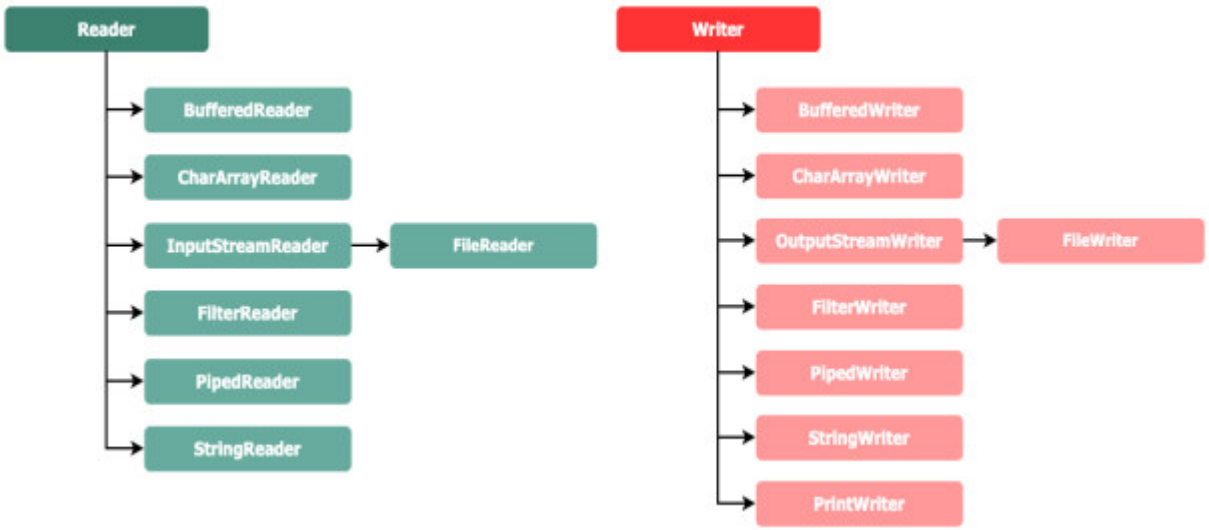
The **Reader** and **Writer** classes handle Unicode character streams. It is common to use these streams in combination with the streams that handle Bytes.

For this purpose we have so-called "**bridge classes**", which convert from byteStream to characterStream

- InputStreamReader → Converts an InputStream to a Reader.
- OutputStreamWriter → Converts an OutputStream into a Writer.

The most important **character stream classes**:

- **FileReader** and **FileWriter** → Reads and writes characters to files.
- **CharArrayReader** and **CharArrayWriter** → Read and write character arrays.
- **BufferedReader** and **BufferedWriter** → Used to use an intermediate buffer between the program and the source or target file.

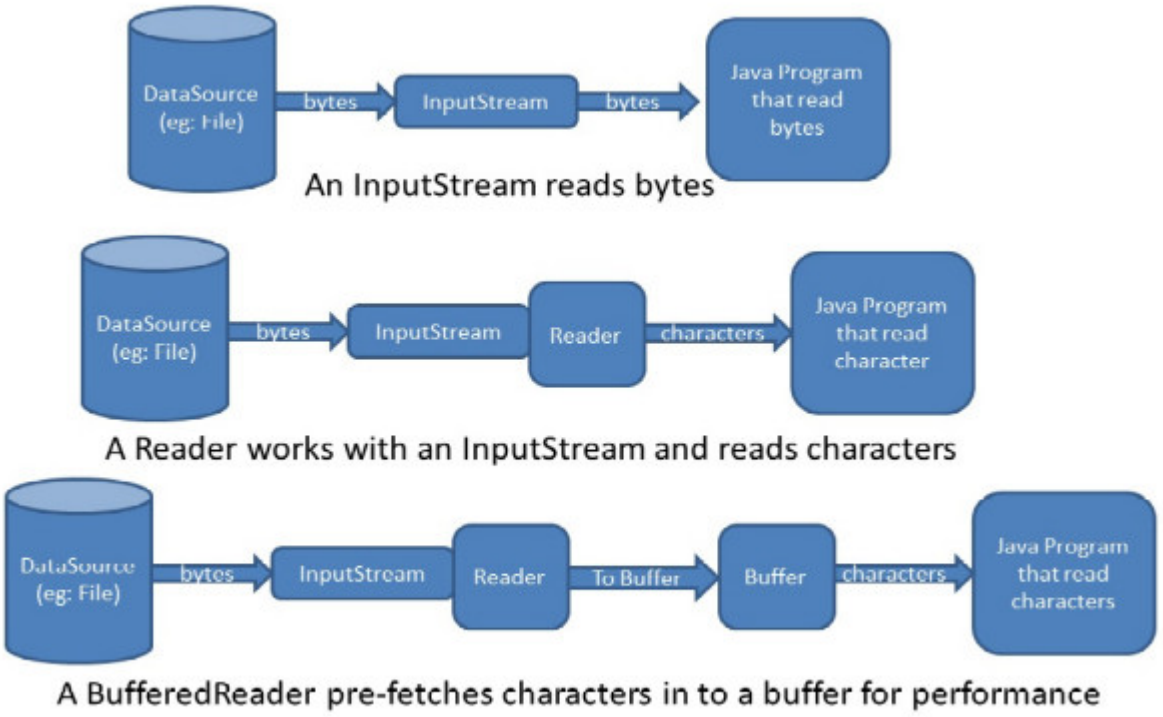


4.2.3. Using Buffers

A buffer is a data structure that allows chunked access to a collection of data. Buffers are useful to avoid storing large amounts of data in memory; instead, small portions of the data are requested from the buffer and processed separately.

They are also very useful for the application to be able to ignore the specific details of underlying hardware efficiency, the application can write to the buffer whenever it wants, and the buffer will write to disk at the most appropriate and efficient rates.

Example using buffers to read binary data:



5. File Access with Java

Java File class is Java's representation of a file or directory pathname. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. Java File class contains several methods for working with the pathname, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

- It is an abstract representation of files and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
- First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

5.1. How to Create a File Object?

A File object is created by passing in a string that represents the name of a file, a String, or another File object. Depending on the operating OS family for which we are programming, we must use one nomenclature or another for the file paths. For example:

- **Unix like** (Linux, freeBSD, Mac OS, Android, IOS, etc.):

```
File fiUnix = new File("/usr/local/bin/ada.txt");
```

This defines an abstract file name for the ada.txt file in the directory /usr/local/bin. This is an absolute abstract file name.

- **Windows**:

```
File fiWindows = new File("C:\\Users\\Admin\\DocumentsandFiles\\ada.txt");
```

This defines an abstract file name for the ada.txt file in the directory C:\\Users\Admin \DocumentsandFiles\. This is an absolute abstract file name.

5.2. Fields in File Class in Java

Field summary:

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

5.3. Methods of File Class in Java

The File class methods can be categorized into several categories:

- GET
- SET
- CAN
- IS
- FUNCTIONAL

For further information, please refer to the Oracle documentation at https://docs.oracle.com/javase/8/docs/api/java/io/File.html

5.3.1. GET methods

Return Type	Method	Description
File	getAbsoluteFile()	Returns the absolute form of this abstract pathname.
String	getAbsolutePath()	Returns the absolute pathname string of this abstract pathname.
File	getCanonicalFile()	Returns the canonical form of this abstract pathname.
String	getCanonicalPath()	Returns the canonical pathname string of this abstract pathname.
long	getFreeSpace()	Returns the number of unallocated bytes in the partition named by this abstract path name.
String	getName()	Returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
File	getParentFile()	Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
String	getPath()	Converts this abstract pathname into a pathname string.
long	getTotalSpace()	Returns the size of the partition named by this abstract pathname.
long	getUsableSpace()	Returns the number of bytes available to this virtual machine on the partition named by this abstract pathname.

5.3.2. SET methods

Return Type	Method	Description
boolean	setExecutable(boolean executable)	A convenience method to set the owner's execute permission for this abstract pathname.
boolean	setExecutable(boolean executable, boolean ownerOnly)	Sets the owner's or everybody's execute permission for this abstract pathname.
boolean	setLastModified(long time)	Sets the last-modified time of the file or directory named by this abstract pathname.
boolean	setReadable(boolean readable)	A convenience method to set the owner's read permission for this abstract pathname.
boolean	setReadable(boolean readable, boolean ownerOnly)	Sets the owner's or everybody's read permission for this abstract pathname.
boolean	setReadOnly()	Marks the file or directory named by this abstract pathname so that only read operations are allowed.
boolean	setWritable(boolean writable)	A convenience method to set the owner's write permission for this abstract pathname.
boolean	setWritable(boolean writable, boolean ownerOnly)	Sets the owner's or everybody's write permission for this abstract pathname.

5.3.3. CAN methods

Return Type	Method	Description
boolean	canExecute()	Tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	Tests whether the application can read the file denoted by this abstract pathname.
boolean	canWrite()	Tests whether the application can modify the file denoted by this abstract pathname.

5.3.4. IS methods

Return Type	Method	Description
boolean	isAbsolute()	Tests whether this abstract pathname is absolute.
boolean	isDirectory()	Tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	Tests whether the file denoted by this abstract pathname is a normal file.

5.3.5. FUNCTIONAL methods

Return Type	Method	Description
int	compareTo(File pathname)	Compares two abstract pathnames lexicographically.
boolean	createNewFile()	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
static File	createTempFile(String prefix, String suffix)	Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static File	createTempFile(String prefix, String suffix, File directory)	Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
boolean	delete()	Deletes the file or directory denoted by this abstract pathname.
void	deleteOnExit()	Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
boolean	equals(Object obj)	Tests this abstract pathname for equality with the given object.
boolean	exists()	Tests whether the file or directory denoted by this abstract pathname exists.
int	hashCode()	Computes a hash code for this abstract pathname.
long	lastModified()	Returns the time that the file denoted by this abstract pathname was last modified.
long	length()	Returns the length of the file denoted by this abstract pathname.
String[ ]	list()	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String[ ]	list(FilenameFilter filter)	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()	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
File[ ]	listFiles(FileFilter filter)	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[ ]	listFiles(FilenameFilter filter)	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
static File[ ]	listRoots()	List the available filesystem roots.
boolean	mkdir()	Creates the directory named by this abstract pathname.

boolean	mkdirs()	Creates the directory named by this abstract pathname, including any necessary but non existent parent directories.
boolean	renameTo(File dest)	Renames the file denoted by this abstract pathname.
Path	toPath()	Returns a java.nio.file.Path object constructed from the this abstract path.
String	toString()	Returns the pathname string of this abstract pathname.
URI	toURI()	Constructs a file: URI that represents this abstract pathname.

5.4. Constructors of Java File Class

We need to create a new instance to work with the file/folder. This can be done very simply and its constructor has **4 overloads**:

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

5.5. Hungarian notation

The data used by a program is assigned an identifier name. In Java -as in other programming languages- we can give any name to a variable or constant (except reserved words). However, this flexible system can result in code that is complex to read and understand.

To solve this problem, Charles Simonyi invented the Hungarian notation. This notation is a commonly used system for creating variable names and consists of lowercase prefixes added to variable names to indicate their type. The rest of the name indicates, as clearly as possible, the function performed by the variable.

In our case, we will rely on the Hungarian notation for variable names, although with some differences. It is recommended to follow the notation in the table below:

Data type	Hungarian notation
Array	ar
ArrayList	arl
boolean	b
BufferedReader	br
BufferedWriter	bw
char	ch
Date	dt
double	d
File	fi
float	f
int	i
List	lst
long	l
Object	obj
Scanner	sc
String	st

5.6. Java File Class Examples

Example 1:

Program to check if a file or directory physically exists or not.

```
import java.io.File;
import java.util.Scanner;

/*
 * In this Java program, we accepts a file or directory name from keyboard.
 * Then, the program will check if that file or directory physically exist or
 * and it displays the property of that file or directory.
 */
public class Example1 {

    /*
     * ---------------------------------
     * GLOBAL VARIABLES AND CONSTANTS
     * ---------------------------------
     */
    //Path constant. We assume that our file is located in the folder "Documer
    final static String PATH = "/home/"+System.getProperty("user.name")+"/Docu
    //We can only use ONE Scanner.
    //We can pass the instance to every method or create this static global
    public static Scanner scKeyboard = new Scanner(System.in);

    /*
     * ---------
     * METHODS
     * ---------
     */
    //read the file or directory by keyboard
    public static String ReadUserFile ()
```

```
    {
        System.out.print("Input the file name: ");
        String stFilename = scKeyboard.next();
        return PATH+stFilename;
    }

    /*
     * --------------
     * MAIN PROGRAMME
     * --------------
     */
     public static void main(String[] stArgs) {
        //read file name
        String stFilename = ReadUserFile();
        // pass the filename or directory name to File object
        File fiFile = new File(stFilename);
        // apply File class methods on File object
        System.out.println("File name: " + fiFile.getName());
        System.out.println("Path: " + fiFile.getPath());
        System.out.println("Absolute path: " + fiFile.getAbsolutePath());
        System.out.println("Parent: " + fiFile.getParent());
        System.out.println("Exists: " + fiFile.exists());
        // Displaying file property
        if (fiFile.exists()) {
            System.out.println("Is writable: " + fiFile.canWrite());
            System.out.println("Is readable: " + fiFile.canRead());
            System.out.println("Is a directory: " + fiFile.isDirectory());
            System.out.println("File Size in bytes: " + fiFile.length());
        }
        System.out.println();
        scKeyboard.close(); //we close the scanner at the end
     }
}
```

Output:

```
Input the file name: filetest.txt
File name: filetest.txt
Path: /home/lliurex-admin/Documentos/filetest.txt
Absolute path: /home/lliurex-admin/Documentos/filetest.txt
Parent: /home/lliurex-admin/Documentos
Exists: true
Is writable: true
Is readable: true
Is a directory: false
File Size in bytes: 13
```

Example 2:

Program to display all the contents of a directory. Here we will accept a directory name from the keyboard and then display all the contents of the directory. For this purpose, list() method can be used as:

```
String arFile[ ]=fiFile.list();
```

In the preceding statement, the list() method causes all the directory entries copied into the array arFile[ ]. Then pass these array elements arFile[ii] to the File object and test them to know if they represent a file or directory.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;

/*
 * Java Program to display all the contents of a directory
 */
public class Example2 {

    // Displaying the contents of a directory
    public static void main(String[] stArgs) throws IOException {
        // enter the path and name from keyboard
```

```
        BufferedReader brLine = new BufferedReader(new InputStreamReader(Syste

        System.out.println("Enter directory path: ");
        String stPath = brLine.readLine();
        System.out.println("Enter directory name: ");
        String stName = brLine.readLine();

        // create File object with path and name
        File fiFile = new File(stPath, stName);

        // if directory exists,then
        if (fiFile.exists()) {
            // get the contents into arFolders[]
            // now arFolders[i] represents either a File or Directory
            String arstFolders[] = fiFile.list();

            // find no. of entries in the directory
            int iNum = arstFolders.length;

            // displaying the entries
            for (int ii = 0; ii < iNum; ii++) {
                System.out.println(arstFolders[ii]);
                // create File object with the entry and
                // test if it is a file or directory
                File fiCurrent = new File(arstFolders[ii]);
                if (fiCurrent.isFile())
                    System.out.println(": is a file");
                if (fiCurrent.isDirectory())
                    System.out.println(": is a directory");
            }
            System.out.println("Number of entries in this directory " + iNum);
        }
        else
            System.out.println("Directory not found");
        }
}
```

6. Accessing Text Files in Java

To work with text files we will use:

- **FileReader** to read.
- **FileWriter** to write.

Whenever we work with these classes we must always carry out a correct management of exceptions since they can produce

- **FileNotFoundException** → In case of not finding the file.
- **IOException** → When some kind of writing error occurs.

6.1. FileReader

The following table shows the **methods** used by the FileReader class to read files, in case of being at the end of the file (EOF) all of them return -1.

Method	Description
int read()	Reads a character and returns it.
int read(char[ ] buf)	Reads up to buf.length characters of data. The characters read are stored in buf.
int read(char buf, int offset, int n)	Reads up to n characters of data from the offset position.

In Java, to open and read a text file, we must:

- Create an instance of the File class.
- Create an input stream with the FileReader class.
- Perform the pertinent reading operations.
- Close the file with the .close() method.

Example (reading a file character by character):

```
        //We create the file in our personal folder
        File fiFile= new File("/home/"+System.getProperty("user.name") + "/Sam

        try {
            //We create the input stream
            FileReader frFile= new FileReader(fiFile);
            int iCaracter;

            //We read the file character by character
            while((iCaracter=frFile.read())!=-1) {
                System.out.print((char)iCaracter);
            }
            //Close the file
            frFile.close();
```

```
        } catch (FileNotFoundException fnfe) {
            fnfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
```

Example (reading a file 20 by 20):

```
//We read the file 20 by 20
char[] chBuffer= new char[20];
while((iCaracter=frFile.read(chBuffer))!=-1) {
    System.out.print(chBuffer);
}
```

6.2. FileWriter

The following table shows the **methods** used by the FileWriter class to write to files:

Method	Description
void write(int c)	Write a character.
void write(char[ ] buf)	Write an array of characters.
void write(char buf, int offset, int n)	Writes n characters of data starting from the offset position.
void write(String str)	Write a string of characters.
void append(char c)	Add a character to the end of the file.

In Java, when writing to a text file we must:

- Create an instance of the File class.
- Create an output stream with FileWriter.
- We perform the writing operations.
- Close the file with the .close() method.

Example 1:

```
public static void main(String[] stArgs) {
    //We create the file in our personal folder
    File fiFile= new File("/home/"+System.getProperty("user.name") + "/Fil
    try {
        //Create an output stream
        FileWriter fwFile= new FileWriter(fiFile);
        String stTexttowrite="This is my first writing on disk.";
        //We write in the file
        fwFile.write(stTexttowrite);
        //Close the file
        fwFile.close();
    } catch (IOException ioe) {
        ioe.printStackTrace();
```

```
        }
    }
```

Example 2:

In the following example we are going to write the entire contents of an array into a file.

```
//We create the file in our personal folder
File fiFile= new File("/home/"+System.getProperty("user.name") + "/File02.txt'
try {

FileWriter fwFile= new FileWriter(fiFile);
String stRomanprovinces[]={
    "Baetica", "Balearica", "Carthaginensis",
    "Gallaecia", "Lusitania", "Tarraconensis"
};
for (int ii = 0; ii < stRomanprovinces.length; ii++) {
    fwFile.write(stRomanprovinces[ii]);
}

//Close the file
fwFile.close();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

We must take into account that if the file already contained information previously, we will overwrite its content when we write to it. If our intention is to add new information, we must create the output stream in the following way:

```
Filewriter fw = new FileWriter(f, true)
```

The second parameter of the constructor is the append parameter:

- true → Append the information to the end of the file.
- false → Delete the previous content to insert the new one.

6.3. BufferedReader

With the BufferedReader class we can do a more advanced management of file reading. The **java.io.BufferedReader** class is ideal for reading text files and processing them. It allows you to efficiently read single characters, arrays or complete lines such as **strings**.

Each read to a BufferedReader triggers a read to the file it is associated with. It is the BufferedReader itself that remembers the last position of the file read, so that subsequent reads access consecutive positions in the file. The **readLine()** method reads a line of the file and returns it in the form of a String. It is especially useful if the record separator is the character \n. This method returns a String with the string read or null if we are at the end of the file (EOF).

To create a BufferedReader we need to start from the FileReader class. This is because the class that actually manages the stream is FileReader. BufferedReader uses it to do a buffered read.

Example:

```
public static void main(String[] stArgs) {
    //We create the file in our personal folder
    File fiFile= new File("/home/"+System.getProperty("user.name") + "/File06.
    try {
        BufferedReader brFile= new BufferedReader(new FileReader(fiFile));
        String stLine="";
        while((stLine=brFile.readLine())!=null) {
            System.out.println(stLine);<br>     }
        brFile.close();

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

6.4. BufferedWriter

We also have the BufferedWriter class, which allows us to write line by line. As with reading, writing from buffers is much more efficient than using byte arrays for large files.

- The **.write()** method writes a line to our file.
- The **.newLine()** method writes a line break.

Example:

```
//We create the file in our personal folder
File fiFile= new File("/home/"+System.getProperty("user.name") + "/File07.txt'
try {
    BufferedWriter bwFile= new BufferedWriter(new FileWriter(fiFile));
    String stRomanprovinces[]={
            "Baetica", "Balearica", "Carthaginensis",
            "Gallaecia", "Lusitania", "Tarraconensis"
        };
    for (int ii = 0; ii < stRomanprovinces.length; ii++) {
        bwFile.write(stRomanprovinces[ii]);
        bwFile.newLine();
    }
    bwFile.close();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

6.5. PrintWriter

The PrintWriter class has print(String) and println(String) methods whose behaviour is similar to System.out methods.

- Both receive a String and write it to the file.
- **println** also inserts a line break.

To build a PrintWriter, once again, we need a FileWriter.

Example:

```
//We create the file in our personal folder
File fiFile= new File("/home/"+System.getProperty("user.name") + "/File08.txt"
try {
    PrintWriter pwFile= new PrintWriter(new FileWriter(fiFile));
    String stRomanprovinces[]={
            "Baetica", "Balearica", "Carthaginensis",
            "Gallaecia", "Lusitania", "Tarraconensis"
        };
    for (int ii = 0; ii < stRomanprovinces.length; ii++) {
        pwFile.println(stRomanprovinces[ii]);
    }
    pwFile.close();

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

7. Bibliography

Sources

- Wikipedia. Sequential access to files. https://en.wikipedia.org/wiki/Sequential_access
- Wikipedia. Random access to files. https://en.wikipedia.org/wiki/Random_access
- Tipos de ficheros. https://www.palabrasbinarias.com/articles/2022-03-31-tipos-de-ficheros/
- Josep Cañellas Bornas, Isidre Guixà Miranda. Accés a dades. Desenvolupament d'aplicacions multiplataforma. Creative Commons. Departament d'Ensenyament, Institut Obert de Catalunya. Dipòsit legal: B. 29430-2013. https://ioc.xtec.cat/educacio/recursos
- Alberto Cortés. Entrada/Salida con ficheros en Java. Universidad Carlos III de Madrid.
- Alberto Oliva Molina. Acceso a datos. UD 1. Trabajo con ficheros XML. IES Tubalcaín. Tarazona (Zaragoza, España).
- Geeksforgeeks. https://www.geeksforgeeks.org/file-class-in-java/
- Oracle Documentation. Class File. https://docs.oracle.com/javase/8/docs/api/java/io/File.html