



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica



# Unit 4. Input/Output. Streams and Files

Computer Programming (PRG)

Academic year 2016/17

Departament de Sistemes Informàtics i Computació



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

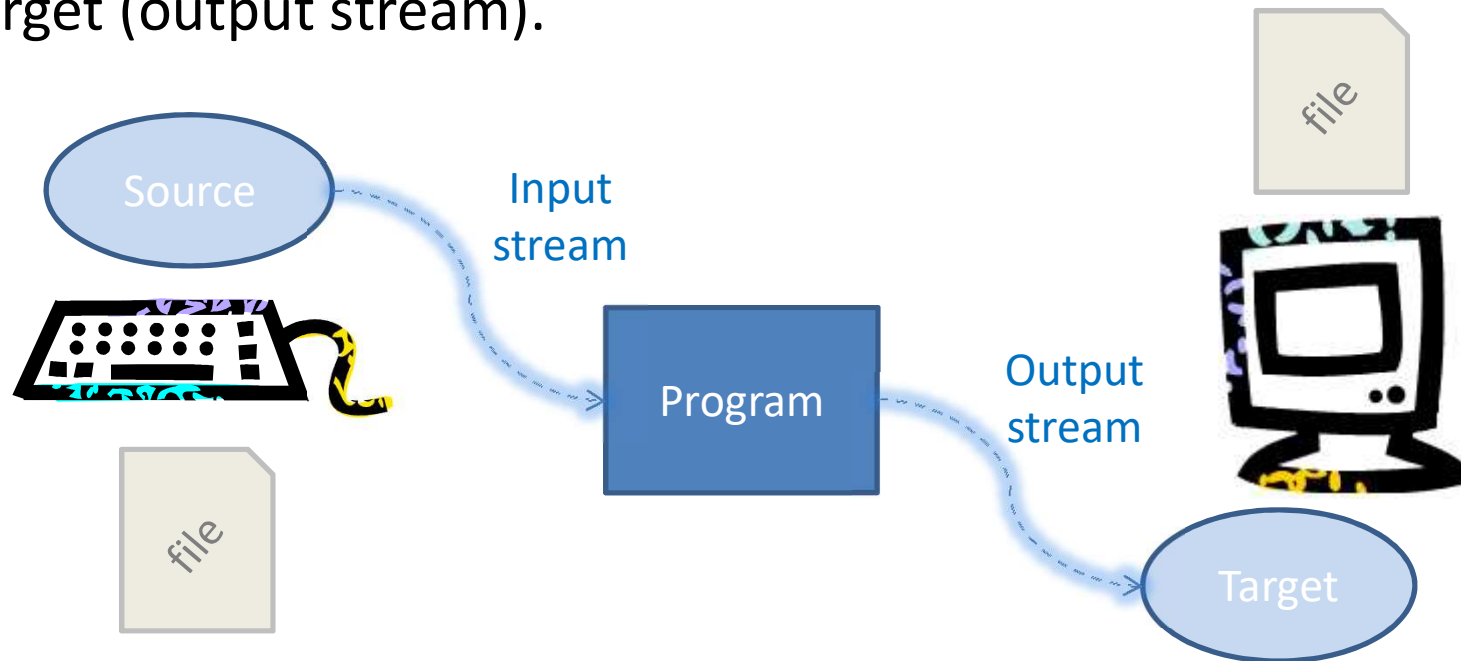


# Contents

1. Introduction: Streams and Files
2. Text files
3. Binary files

# Introduction: Streams and Files

- Input/Output in Java is performed by means of streams.
- Streams are data sequences from a source (input stream) to a target (output stream).



# Introduction: Files

- A file is a sequence of bits (bytes) stored in a secondary storage device (hard disk, USB stick, etc.).
  - We can store into a file, data stored in main memory (RAM—primary storage) which are used in a program. Later on, the same program or other ones can use such data.
- Main features of files:
  - Name (i.e. mydatafile.txt).
  - Absolute path in the secondary storage device (i.e. /home/lucas/docs/file.txt).
  - Size in bytes. Can be shown in standard multiples (Kb, Mbytes, Gbytes, etc.).
- Additional features of files:
  - Access permissions (depending on the file system).
  - Time of last modification or read access.
  - ...
- Common actions we can perform over files:
  - Open — Read — Close.
  - Open — Write — Close.

# Sort of Files

- Generally, we can distinguish between two kinds of files:

Text files
<ul style="list-style-type: none"><li>• Sequence of characters.</li><li>• Human readable.</li><li>• Portable (can be read in different computers).</li><li>• Less efficient read/write operations.</li><li>• Larger than binary files for storing the same information.<ul style="list-style-type: none"><li>• Ex. A 10 digit integer number needs 10 bytes if each character fits in a byte.</li></ul></li></ul>

Binary files
<ul style="list-style-type: none"><li>• Byte sequences for storing built-in data types and objects.</li><li>• No human readable.</li><li>• Usually no transportable. Files must be read in the same computer.</li><li>• Efficient read/write operations.</li><li>• Efficient information storage.<ul style="list-style-type: none"><li>• Ex. A 10 digit integer number needs 4 bytes.</li></ul></li></ul>

- Using Java, binary files are platform independent. Obviously, these files must be read by means of Java programs.

# Working with files in Java

- Java allows to ...
  1. interact with most file system types with no regard of their internal organization (FAT32, NTFS, EXT3, etc.) or operating system (Windows, Linux, MAC, etc.).
  2. create and read from text files, by using `PrintStream` and `PrintWriter` classes for writing text and `Scanner` class for reading.
  3. create and read from binary files by using other classes we'll see next. Variables of built-in data types and String objects can be written or read directly.
  4. work with binary files for writing and reading objects of any class (which implements the `Serializable` interface).
  5. manage binary files at byte level.
- Only items from 1 up to 4 will be treated in this unit.

# Accessing to the File System

- By using class [java.io.File](#):
  - Allows representing both files and directories.

Method/ Constructor	Description
<code>File( String pathname )</code>	Creates a new File object given the pathname
<code>boolean delete()</code>	Deletes file or directory
<code>boolean exists()</code>	Returns <b>true</b> whether the file or directory exists
<code>String getName()</code>	Returns the file/directory name without the path
<code>String getParent()</code>	Returns the path of the file/directory without the name
<code>long length()</code>	Returns the size in bytes. Not valid for directories
<code>File[] listFiles()</code>	Obtains a list of files/directories in the directory
<code>boolean isDirectory()</code>	Returns whether this object refers to a directory
...	



Creating a File object doesn't perform any operation in the file system while no methods are invoked.

# Using the class File

- Example of using the class file for interacting with the file system.

```
import java.io.*;
class TestFile {
    public static void main(String [] args){
        File f = new File("/home/plopez/file.txt");
        if (f.exists()) System.out.println("File exists!");
        else System.err.println("File doesn't exist!");

        System.out.println("getName(): " + f.getName());
        System.out.println("getParent(): " + f.getParent());
        System.out.println("length(): " + f.length());
    }
}
```

- `getName()` returns `"file.txt"` and `getParent()` returns `"/home/plopez"`



# Writing to text files

- The easiest way of working with text files is by using the classes [PrintWriter](#) and [Scanner](#) for writing and reading respectively.
- Class [java.io.PrintWriter](#):
  - Writes text representations with format of built-in data types.

Method / Constructor	Description
<code>PrintWriter( File file )</code>	Creates a new <code>PrintWriter</code> object given a <code>File</code> object
<code>PrintWriter( OutputStream out )</code>	Creates a new <code>PrintWriter</code> object given an <code>OutputStream</code> object
<code>void print( float f )</code>	Writes a float
<code>void println( float f )</code>	Writes a float followed by a new line
<code>void print( boolean b )</code>	Writes a boolean
...	...
<code>boolean checkError()</code>	Checks whether an error was produced after last writing operation

# Class PrintWriter

Alternatives for creating `PrintWriter` objects:

- `PrintWriter pw = new PrintWriter(new File("/tmp/f.txt"));`

If a file with the same pathname doesn't exist a new file is created in the file system, otherwise the existent file is **wiped** (size is truncated to zero).

- `PrintWriter pw2 = new PrintWriter(  
new FileOutputStream("/tmp/f.txt", true));`

If a file with the same pathname doesn't exist a new file is created in the file system, otherwise the existent file is **maintained** and new data is **appended**.

- If the specified pathname exists and the user doesn't have permissions, then the constructor of `PrintWriter` or `FileOutputStream` can throw a ***FileNotFoundException***. Our code must catch or propagate it.

# Using class PrintWriter

- Example of using PrintWriter:

```
import java.io.*;
public class TestPrintWriter {
    public static void main(String[] args){
        String fileName= "file2.txt";
        try {
            PrintWriter pw = new PrintWriter(new File(fileName));
            pw.print("El veloz murciélago hindú");
            pw.println(" comía feliz cardillo y kiwi");
            pw.println(4.815162342);
            pw.close();
        } catch (FileNotFoundException e) { System.err.println("Error opening or creating " + fichero); }
    }
}
```

- This program writes into the file:  
El veloz murciélago hindú comía feliz cardillo y kiwi  
4.815162342

# Reading from text files

- Class [java.util.Scanner](#):
  - Allows us to read native types and Strings.
  - Automatically extracts and processes individual tokens. White spaces are used as separators (white space, tab, new line, carriage return, new page).

Method / Constructor	Description
Scanner( File source )	Creates a new Scanner object for reading from a File object
Scanner( String src )	Creates a new Scanner object for reading from a String object
boolean <b>hasNext()</b>	Returns <b>true</b> whether a new token can be read
boolean <b>hasNextInt()</b>	Returns <b>true</b> if the next token can be converted to an integer
int <b>nextInt()</b>	Returns next token as an integer. If token contains no valid characters for integer conversion an <i>InputMismatchException</i> is thrown
...	...
String <b>next()</b>	Returns next token as a String
String <b>nextLine()</b>	Returns a String from current position up to the next new line character is found
void <b>close()</b>	Close the file associated with this Scanner object.

# Exceptions when reading from text files

- If the specified file can't be accessed with the permissions of the current user, then the constructor of the class `Scanner` can throw the checked exception *`FileNotFoundException`*.
  - This exception must be caught or propagated.
- Some methods such as `nextInt()` and `nextDouble()` can throw the unchecked exception *`InputMismatchException`* if any not valid character or formatting errors are found in the token. *`NoSuchElementException`* can be thrown by any method if a read operation is performed beyond the end of the file.
  - Catching these unchecked exceptions will avoid aborting the program when trying to read a incorrectly formatted datum or non existent data.

- Example for writing 10 integers into a file and reading them before showing them on screen.

```
import java.io.*;
import java.util.Scanner;
class TestPrintWriter1{
    public static void main(String[] args){
        String fileName= "file1.txt";
        try {
            PrintWriter pw = new PrintWriter(new File(fileName));
            for (int i = 0; i < 10 ; i++) pw.println(i);
            pw.close();
            Scanner scanner = new Scanner(new File(fileName));
            while (scanner.hasNext()) { System.out.println("Read value: "+scanner.nextInt()); }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.err.println(" ERROR when opening file: " + fichero);
        }
    }
}
```

# Example using Scanner (I)

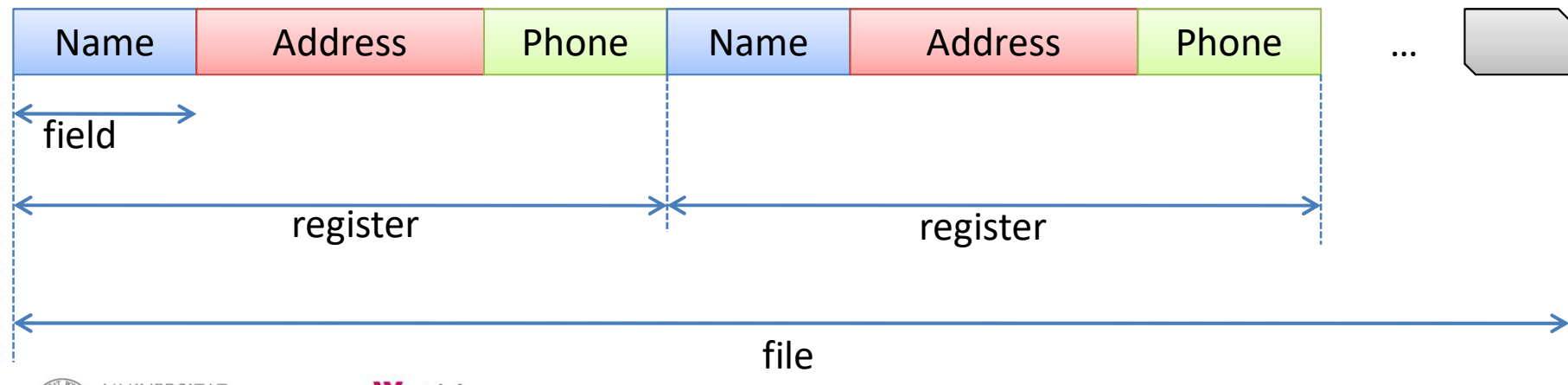
```
import java.io.*; import java.util.Scanner;
public class TestScanner {
    public static void main(String[] args){
        System.out.println("We are going to read three integers and a text line.");
        Scanner scanner = null;
        try{
            scanner = new Scanner( new File( "things.txt" ) );
        }catch( FileNotFoundException ex ){
            System.err.println( "File doesn't exist." + ex.getMessage()); System.exit(0);
        }
        int n1 = scanner.nextInt(); int n2 = scanner.nextInt(); int n3 = scanner.nextInt();
        scanner.nextLine();
        String line = scanner.nextLine();
        System.out.println( "Numbers are: " + n1 + ", " + n2 + ", " + n3 );
        System.out.println( "Line is: " + line );
        scanner.close();
    }
}
```

things.txt

```
1 2
3 4
Hello world!
```

# Binary files: sequential access

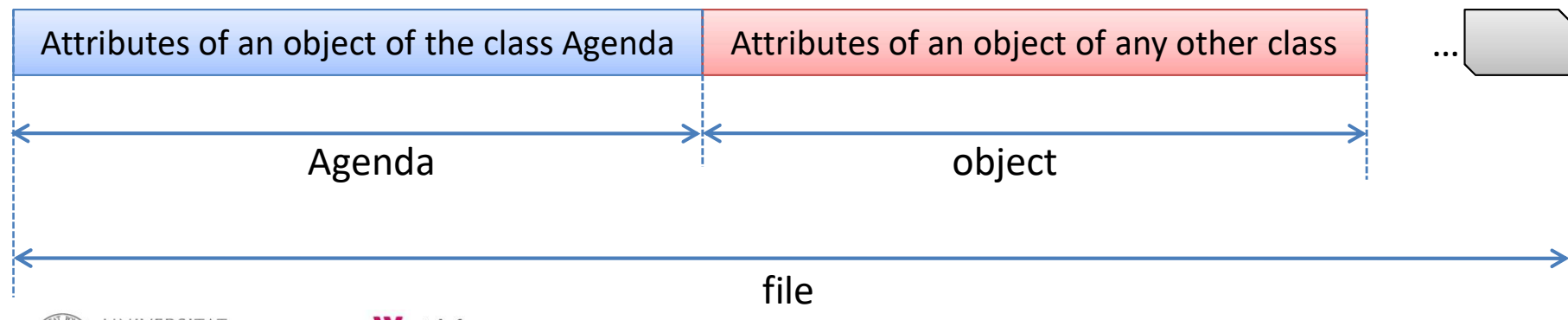
- Variables of native data types can be stored into and loaded from binary files. An int occupies 4 bytes, a double 8 bytes, etc.
- Data must be read in the same order it was written.
- For example: every register of an agenda is stored following the same sequence for its components, i.e., structuring the information as a sequence of registers:






# Binary files: sequential access

- It is possible in Java storing a sequence of objects for reading them later. This is known as **serialization**.
- For example, it is possible to save the contents of an agenda by storing all the objects of class **ItemAgenda** one following each other.
- And it is possible to save directly the object of the class **Agenda**, including all the objects of class **ItemAgenda** contained in it.
- Furthermore, a file can contain objects of different classes and values of built-in data types.



# Binary files: sequential access

- Java facilitates us writing data into files using binary format, and reading them:
  - Values of built-in data types: boolean, int, double, etc.
  - Objects with all its internal attributes and referenced objects, then the internal attributes of the referenced objects, and so on, in a recursive way.
- For performing these operations we have to use the following classes from the package `java.io`: [`java.io.ObjectInputStream`](#) (for reading) and [`java.io.ObjectOutputStream`](#) (for writing).



**Notice:** it is mandatory that classes, whose objects will be written to a stream and later they will be read from a stream, implement the `serializable` interface. For example:

```
public class ItemAgenda implements Serializable {  
    .....  
} // End of class ItemAgenda
```

# Binary files: sequential access

- For **reading** from or **writing** to a binary file we have to:
  1. Create an object of the class **File**, the data **source/target**.
  2. Enveloping it with an object of the class **FileInputStream** / **FileOutputStream** for creating a data stream **from/to** the file.
  3. Enveloping the previous object with an object of the class **ObjectInputStream/ObjectOutputStream** for **reading/writing** values of native data types or objects **from/to** a data stream.
- Methods for performing read/write operations:
  - `writeInt()`, `writeDouble()`, `writeBoolean()`, `writeObject()`, ...
  - `readInt()`, `readDouble()`, `readBoolean()`, `readObject()`, ...

# Reading/writing binary data

- The class `ObjectOutputStream` provides methods for writing values of native data types.
- The class `ObjectInputStream` provides methods for reading them.

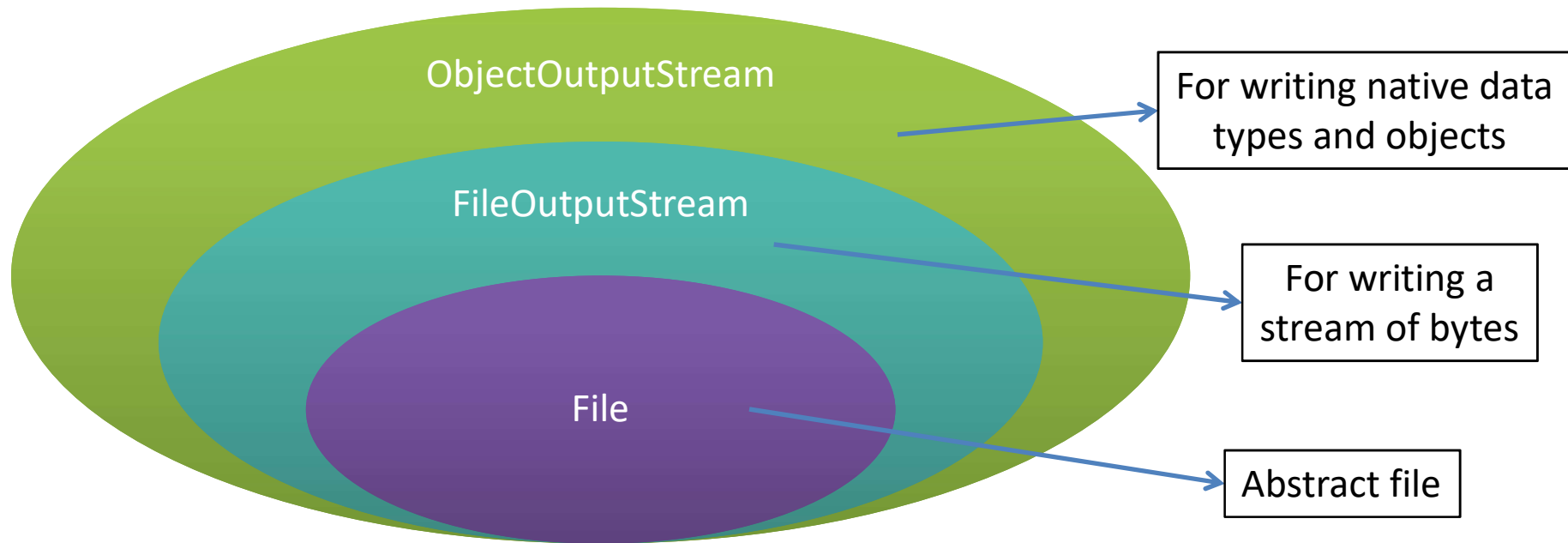
Method / Constructor	Description
<code>ObjectOutputStream( OutputStream out )</code>	Creates a new object from an existing output stream.
<code>void writeInt( int v )</code>	Writes an integer number in binary format using 4 bytes
<code>void writeLong( long v )</code>	Idem but using 8 bytes
<code>void writeUTF( String str )</code>	Writes a character sequence by using codification UTF-8
<code>void writeDouble( double v )</code>	Writes a real number in binary format using 8 bytes
<code>void writeObject( Object obj )</code>	Writes the object <b>obj</b> and all the objects it contains recursively

- Writing methods can throw an exception of class `IOException`.



Binary files which were created by using `ObjectOutputStream` only can be read by means of an object of class `ObjectInputStream` due to the format conversions that are performed.

# Using ObjectOutputStream



- Example for creating an stream for writing objects:
  - `ObjectOutputStream out = new ObjectOutputStream(  
new FileOutputStream( new File( fileName ) ) );`
  - `out.writeInt(45);`

The file name can be passed directly to the constructor of the class `FileOutputStream` instead of using an intermediate object of the class `File`



# Exception handling

- The `FileOutputStream/FileInputStream` constructor can throw an exception of the class `FileNotFoundException`
  - If the specified file doesn't exist in the file system.
- Writing methods of `ObjectOutputStream` and reading methods of `ObjectInputStream` can throw an exception of the class `IOException`, or of any subclass derived from it.
  - An error occurs while performing the operation, which can be due to incorrect permissions, hardware failure, etc.
  - If the object we try to write is an instance of a class that no implements the `Serializable` interface, i.e., it is not a serializable object.
- In addition, the method `readObject()` of `ObjectInputStream` can throw a `ClassNotFoundException` if it is not possible to detect the class of the object being read.
- All these exceptions must be caught by using `try-catch-finally`.

# Example of using binary files in a sequential way for built-in data types

Program that writes data into a file, reads them from the file and shows them on screen

```
import java.io.*;
class Qualifications {
    public static void main(String[] args){
        String fileName = "qualifications.data"; String name = "PRG"; int call= 1; double grade = 7.8;
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(fileName));
            out.writeUTF(name); out.writeInt(call); out.writeDouble(grade);
            out.close();
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(new File(fileName)));
            System.out.println( "Name: " + in.readUTF() );
            System.out.println( "Call: " + in.readInt() );
            System.out.println( "Grade: " + in.readDouble() );
            in.close();
        } catch (FileNotFoundException e) {
            System.err.println( "Problems with file " + fileName + "." + e.getMessage() );
        } catch (IOException e) {
            System.err.println( "Problems writing on file " + fileName );
        }
    }
}
```

# Example of using binary files for writing objects in a sequential way (1)

- As an example, we are going to write and read objects in a file for saving and recovering the data of an agenda.
- We'll use previously defined classes:
  - ItemAgenda (information about a person)
  - Agenda (the set of all the contacts)

```
import java.io.*;

public class ItemAgenda implements Serializable {

    private String name; private String phone; private int postal;

    public ItemAgenda( String n, String t, int p) { name = n; phone = t; postal = p; }
    public String toString() { return nom + ": " + phone + "(" + postal + ")"; }

    // other methods...

} // End of class ItemAgenda
```



# Example of using binary files for writing objects in a sequential way (2)

- Remember, it is mandatory to specify that all the classes whose objects we will write to a stream must implement the **Serializable** interface.

```
import java.io.*;

public class Agenda implements Serializable {

    public static final int MAX = 8;

    private ItemAgenda[] contacts; private int num;

    public Agenda() { contacts = new ItemAgenda[MAX]; num = 0; }

    public void add(ItemAgenda b) {
        if (num >= contacts.length) growContacts(); contacts[num++]=b; }

    public String toString() {
        String str=""; for (int i=0;i<num;i++) str += contacts[i]+"\\n";
        str += "=====\\n";
        return str; }

    // Other methods such as deleteItem, findByName, ...
}
```

# Example of using binary files for writing objects in a sequential way (3)

- Methods for saving and loading Agenda objects:

```
// In class Agenda ...
public void save( String filename ) {
    try {
        FileOutputStream fos = new FileOutputStream( fileName );
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(this); oos.close();
    } catch (IOException ex) { System.err.println("Error saving: " + ex.getMessage()); } }

public static Agenda load( String fileName ) {
    Agenda aux = null;
    try {
        FileInputStream fis = new FileInputStream( fileName );
        ObjectInputStream ois = new ObjectInputStream( fis );
        aux = (Agenda)ois.readObject(); ois.close();
    } catch (IOException ex) { System.err.println("Error loading: " + ex.getMessage()); }
    catch (ClassNotFoundException ex) { System.err.println("Error: " + ex.getMessage()); }
    return aux; }
} // End of class Agenda
```

# Example of using binary files for writing objects in a sequential way (4)

- In class `TestManager` it is created, saved and loaded an Agenda with some contacts:

```
import java.io.*;

public class TestManager {

    public static void main(String[] args) {
        ItemAgenda i1 = new ItemAgenda("Enrique Perez","622115611",46022);
        ItemAgenda i2 = new ItemAgenda("Rosalía","963221153",46010);
        ItemAgenda i3 = new ItemAgenda("Juan Duato","913651228",18011);

        Agenda a1 = new Agenda();
        a1.add(i1); a1.add(i2); a1.add(i3);

        // Save and show Agenda a1:
        a1.save( "agenda1.dat" );
        System.out.println( "SAVED AGENDA:" );  System.out.println( a1 );

        // Load from file and show the recovered agenda:
        Agenda a2 = Agenda.load( "agenda1.dat" );
        System.out.println( "LOADED AGENDA: " );  System.out.println( a2 );
    }
} // End of class TestManager
```

# Example of using binary files for writing objects in a sequential way (5)

- The output of executing the main method of class TestManager is the following:

```
SAVED AGENDA:
Enrique Perez: 622115611 (46022)
Rosalía: 963221153 (46010)
Juan Duato: 913651228 (18011)
=====
LOADED AGENDA:
Enrique Perez: 622115611 (46022)
Rosalía: 963221153 (46010)
Juan Duato: 913651228 (18011)
=====
```

- And the file “agenda1.dat” exists in the current working directory:

```
$ ls -l *.dat
-rw-r--r-- 1 profesor PRG 276 2012-03-20 18:00 agenda1.dat
```

# How to assess when the end-of-file has been reached.

- Sometimes, the amount of elements (registers) stored in a file is unknown.
- A read operation beyond the end-of-file throws an exception of a class derived from *IOException*. If our code catches this exception we can manage the situation.
- Furthermore, working with *ObjectInputStream* and trying to read values of native data types beyond the end-of-file throws an exception of type *EOFException*, a subclass derived from *IOException*.
- **In conclusion**, we can read all the elements in a file, one after another, until an exception is thrown.
- The following example shows this strategy that, in addition, can be used for working with any kind of stream that throws exceptions similar to *EOFException*.

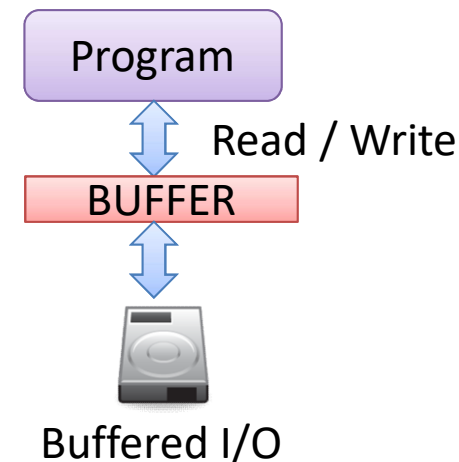
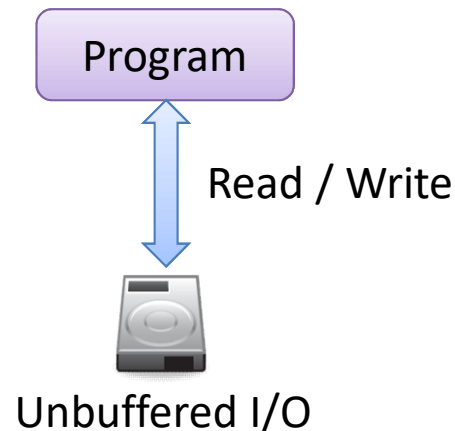
# How to assess when the end-of-file has been reached.

- The following example shows how to read from a file of integer numbers when size is unknown.
- Values are written on screen as they are read, a notice is shown when end-of-file is reached.

```
public static void read( String fileName ) {  
    try {  
        ObjectInputStream ois = new ObjectInputStream( new FileInputStream( fileName ) );  
  
        try {  
            while ( true ) {  
                int val = ois.readInt();  
                System.out.println(val);  
            }  
        } catch ( EOFException ef ) { System.out.println( "End of file." ); }  
  
        ois.close();  
    } catch ( IOException fex ) { System.err.println( "ERROR: " + fex.getMessage() ); }  
}
```

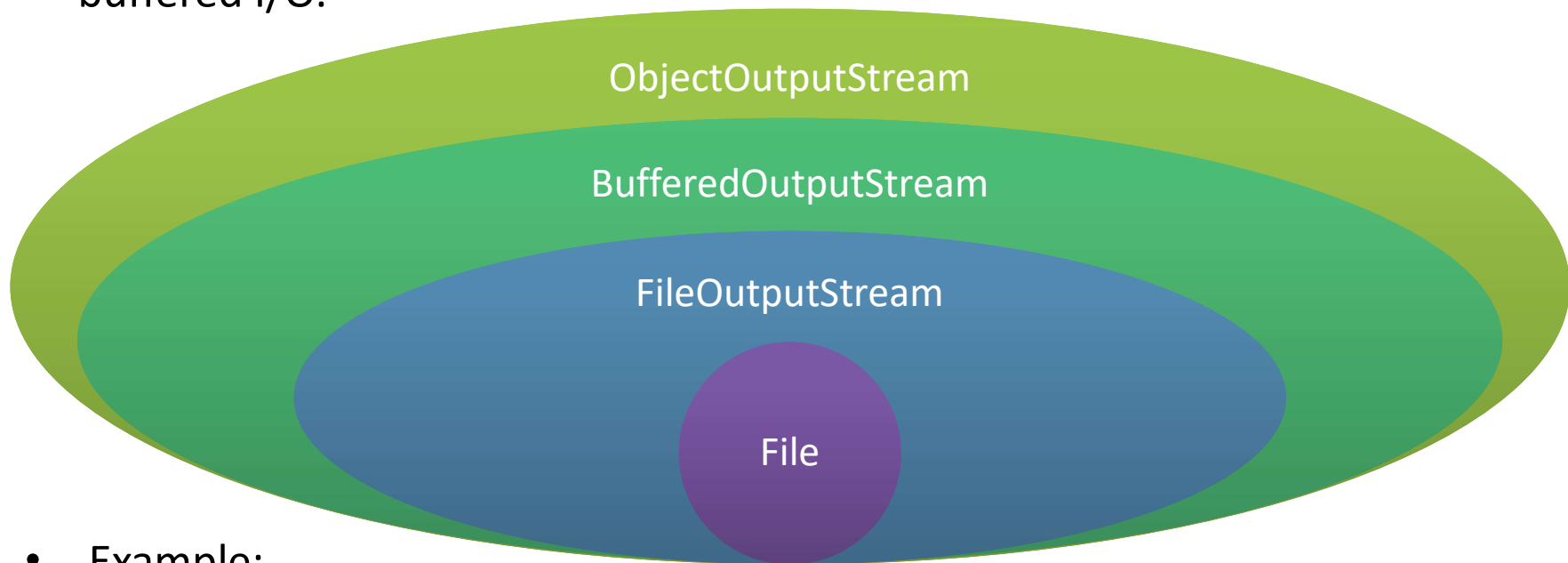
# Read/write operations on binary files using buffered streams (1)

- All write operations performed by the methods of class `ObjectOutputStream`, using in the lower level the class `FileOutputStream`, trigger write operations on disk individually and immediately. This way of proceeding is not efficient.
- The efficient solution consists in using buffers for temporal storage of data into memory, and performing write operations to disk when a sufficient amount of data is available.
  - It is more efficient performing 1 write operation on disk for 100 KB that 100 write operations of 1 KB.



# Read/write operations on binary files using buffered streams (2)

- We can use classes `BufferedInputStream` and `BufferedOutputStream` for buffered I/O.



- Example:

```
ObjectOutputStream out = new ObjectOutputStream(  
    new BufferedOutputStream(  
        new FileOutputStream( new File( fileName ) ) ) );  
out.writeInt(45);
```



# Hints for working with files

- Files (streams) must be explicitly closed after using them.
  - In particular a file (stream) that has been used for writing data must be closed before using it for reading. Only in this case is warranted a correct reading.
- JVM automatically closes all open files that programmer forgets to close, but
  - if the program ends prematurely due to an outage and not all the buffered data was flushed into the disk, then the file can be corrupted and some data could be lost.
- It is important to handle all the possible exceptions related with I/O:
  - Non existent files ([\*FileNotFoundException\*](#)).
  - I/O failures ([\*IOException\*](#)).
  - Inconsistencies related with data types when using [\*Scanner\*](#) ([\*InputMismatchException\*](#)).
  - End-Of-File reached ([\*EOFException\*](#))

# Conclusions

- Using files allows that information is held when program ends execution. Files are stored in secondary storage devices.
- Java allows us working with binary files by means of streams and with a format independent of the computer architecture or operating system.
- With text files also.
- The easiest way for working with **text files** is using:
  - PrintWriter (**write**), Scanner (**read**).
- The easiest way for performing read/write operations on **binary files (sequential access)** in order to operate with values of built-in data types or objects is using:
  - ObjectOutputStream (**write**), ObjectInputStream (**read**)