# Unit 9

# Files and input/output streams


### Summary

- The concept of file
- Writing and reading text files
- Operations on files
- Input streams: keyboard, file, internet
- Output streams: file, video
- Generalized writing and reading through streams


## 9.1 The concept of file

**Files** are the most important mechanism for storing data permanently on mass-storage devices. *Permanently* means that the data is not lost when the machine is switched off. Files can contain:

- data in a format that can be interpreted by programs, but not easily by humans (*binary files*);
- alphanumeric characters, codified in a standard way (e.g., using ASCII or Unicode), and directly readable by a human user (*text files*). Text files are normally organized in a sequence of lines, each containing a sequence of characters and ending with a special character (usually the *newline character*). Consider, for example, a Java program stored in a file on the hard-disk. In this unit we will deal only with text files.

Each file is characterized by a *name* and a *directory* in which the file is placed (one may consider the whole path that allows one to find the file on the hard-disk as part of the name of the file).

The most important operations on files are: *creation*, *reading from*, *writing to*, *renaming*, *deleting*. All these operations can be performed through the operating system (or other application programs), or through suitable operations in a Java program.

## 9.2 Operations on files

To execute reading and writing operations on a file, we must *open* the file before doing the operations, and *close* it after we are finished operating on it.

- **Opening** a file means to indicate to the operating system that we want to operate on a file from within a Java program, and the operating system verifies whether such operations are possible and allowed. There are two ways of opening a file: *opening for reading* and *opening for writing*, which cause a different behavior of the operating system. For example, two files may be opened at the same time by two applications for reading, but not for writing; or a file on a CD may be opened for reading, but not for writing. In many programming languages (including Java), opening a file for writing means actually to create a new file.
- **Closing** a file means to indicate to the operating system that the file that was previously opened is not being used anymore by the program. Closing a file also has the effect of ensuring that the data written on the file are effectively transferred to the hard-disk.

## 9.3 Exceptions

File operations can typically cause unexpected situations that the program is not capable of handling (for example, if we try to open a file for reading, and specify a filename that does not exist). Such situations are called **exceptions**. They are classified according to the type of inconvenient that has happened, and they must be handled by the program by inserting suitable Java code (see Unit 10).

For this unit, it is sufficient to know that *the methods that make use of statements that can generate an exception must declare this.*

For example, the methods for opening a file for reading can generate an exception of type `IOException`. Therefore, all methods that call such methods must declare explicitly that they can themselves generate an exception of that type.

The type of exception that a method can generate is specified in the `throws` clause, which must be added to the method declaration before the body of the method. For example:

```
public static void main(String[] args) throws IOException {
  body
}
```

## 9.4 Writing text files

To write a string of text on a file we have to do the following:

1. *open* the file *for writing* by creating an object of the class `FileWriter` associated to the name of the file, and creating an object of the class `PrintWriter` associated to the `FileWriter` object just created;
2. *write text* on the file by using the `print()` and `println()` methods of the `PrintWriter` object;
3. *close* the file when we are finished writing to it.

The Java statements that realize the three phases described above are:

```
// 1. opening the file for writing (creation of the file)
FileWriter f = new FileWriter("test.txt");
PrintWriter out = new PrintWriter(f);

// 2. writing text on the file
out.println("some text to write to the file");

// 3. closing the output channel and the file
out.close();
f.close();
```

*Notes:*

- The invocation of the constructor of the class `FileWriter` has the effect of creating a file that is ready to be written on. If the file already exists, all its contents is erased first.
- To append text to an already existing file without erasing the previous content of the file, we have to use the constructor with the signature `FileWriter(String, boolean)`, specifying the value `true` for the second argument.

## 9.5 Program for writing to a file

The following program creates a text file called `test.txt` and writes on it the string `"some text written on a file"`.

```
import java.io.*;

public class WritingOnFile {
  public static void main(String[] args) throws IOException {
    // opening the file for writing
    FileWriter f = new FileWriter("test.txt");
    // creation of the object for writing
    PrintWriter out = new PrintWriter(f);

    // writing text on the file
    out.println("some text written on a file");

    // closing the output channel and the file
    out.close();
    f.close();
  }
}
```

*Notes:*

- The program must handle possible errors that may occur in accessing the file by declaring:

```
public static void main(String[] args) throws IOException
```

- The classes for handling the input/output are part of the library `java.io`, and hence we must import that library in the classes that make use of input/output.

## 9.6 Loop schema for writing to a file

When we write data on a file, we usually make use of a loop structured as follows:

```
PrintWriter out = ...
while (condition) {
   out.println(data);
   ...
}
out.close();
```

*Note:* The *condition* for terminating the loop does not depend on the file, but on the data which we are writing to the file.

*Example:* Method that writes an array of strings on a file. Both the array of strings and the filename are passed as parameters to the method.

```
import java.io.*;

public static void saveArray(String[] v, String filename) throws IOException {
  FileWriter f = new FileWriter(filename);
  PrintWriter out = new PrintWriter(f);
  for (int i = 0; i < v.length; i++)
    out.println(v[i]);
  out.close();
  f.close();
}
```

## 9.7 Reading from a text file

To read strings of text from a file we have to:

1. *open* the file *for reading* by creating an object of the class `FileReader` and an object of the class `BufferedReader` associated to the `FileReader` object we have just created;
2. *read* the lines of text from the file by using the `readLine()` method of the `BufferedReader` object;
3. `close` the file when we are finished reading from it.

The Java statements that realize the three phases described above are:

```
// 1. opening the file for reading
FileReader f = new FileReader("test.txt");;
BufferedReader in = new BufferedReader(f);

// 2. reading a line of text from the file
String line = in.readLine();

// 3. closing the file
f.close();
```

*Notes:*

- If the file that we want to open for reading does not exist, then an exception of type `FileNotFoundException` is raised when the object `FileReader` is created.
- After we have opened the file, we start reading from the its first line. Then, each time we invoke the method `readLine()`, we advance the input to the next line, so that each invocation of `readLine()` will read the following line of text in the file.

## 9.8 Program for reading from a file

The following program opens a text file called `test.txt`, reads from it a line of text, and prints in on the video.

```
import java.io.*;

public class ReadingFromFile {
  public static void main(String[] args) throws IOException {
    // opening the file for reading
    FileReader f = new FileReader("test.txt");
    // creation of the object for reading
    BufferedReader in = new BufferedReader(f);

    // reading a line of text from the file
    String line = in.readLine();
    System.out.println(line);

    // closing the file
    f.close();
  }
}
```

*Notes:*

- Also in this case, the program that realizes the input must handle possible errors that may occur in accessing the file, by declaring:

  ```
  public static void main(String[] args) throws IOException
  ```

- If in a program we need to read again the content of the file starting from the first line, it is necessary to close the file and open it again for reading (see, e.g., the following paragraph).

## 9.9 Loop schema for reading from a file

When we read a set of strings from a file, we usually make use of a loop structured as follows:

```
BufferedReader in = ...
String line = in.readLine();
while (line != null) {
   process line
   line = in.readLine();
}
```

The loop schema illustrated above represents an iteration on the lines of the file. Indeed, the method `readLine()` returns the value `null` if there are not data to read from the file (e.g., when we are at the end of the file). Such a condition is used to verify the termination of the loop.

*Example:* Method that reads an array of strings from a file.

```
public static String[] loadArray (String filename) throws IOException {
  // We first read the file to count the number of lines
  FileReader f = new FileReader(filename);
  BufferedReader in = new BufferedReader(f);
  int n = 0;
  String line = in.readLine();
  while (line != null) {
    n++;
    line = in.readLine();
  }
  f.close();

  // Creation of the array
  String[] v = new String[n];

  // Loop to read the strings from the file into the array
```

```
    f = new FileReader(filename);
    in = new BufferedReader(f);
    int i = 0;
    line = in.readLine();
    while ((line != null) && (i < n)) {
      v[i] = line;
      line = in.readLine();
      i++;
    }
    f.close();
    return v;
  }
```

This method returns an array of strings corresponding to the lines of text in a file. Note that we read the file twice (using two loops): the first time to count the number of lines of the file, so that we can determine the size of the array; the second time to read the strings that fill up the array. Note also that, to read data twice from the same file, we have to close the file and the re-open it again.

## 9.10 Reading from a file using the `Scanner` class

An alternative way to read from a file is by using the `Scanner` class, which has a constructor that allows one to initialize a `Scanner` object so that it reads from a file (instead of standard input).

For example:

```
File f = new File("myNumbers")
Scanner sc = new Scanner(f);
```

Then, using the `Scanner` object, we can use the `next()` and `hasNext()` methods, or their variants for reading primitive types (such as `nextInt()` and `hasNextInt()`), to read from the file.

For example, to sum up numbers read from a file:

```
File f = new File("myNumbers")
Scanner sc = new Scanner(f);

long sum = 0;
while (sc.hasNextLong()) {
  long x = sc.nextLong();
  sum = sum + x;
}
System.out.println(sum);
```

## 9.11 Renaming and deleting a file in Java

To **delete a file** (i.e., remove it completely from the mass-storage device), we invoke the method `delete()` on an object of type `File` created with the name of the file to delete.

```
File f1 = new File("garbage.txt");
boolean b = f1.delete();
// if b is true, then the file has been deleted successfully
```

*Note:* The constructor of the class `File` does not generate an exception if the file does not exist. The result of the deletion operation is returned by the `delete()` method.

To **rename a file**, we invoke the method `renameTo()` on two objects of type `File` that represent respectively the file to rename, and the new name to give to the file.

```
File f1 = new File("oldname.txt");
File f2 = new File("newname.txt");
boolean b = f1.renameTo(f2);
// if b is true, then the file has been renamed successfully
```

The file `oldname.txt`, if it exists, is renamed to the file `newname.txt`. If there already exists a file named `newname.txt`, then it is overwritten. The result of the renaming operation is returned by the `renameTo()` method.

## 9.12 Input/output streams

Java defines a common way of handling all input/output devices, namely as producers/consumers of sequences of data (characters). The concept of **stream** represents a *sequence of data generated by a input device or consumed by an output device.*

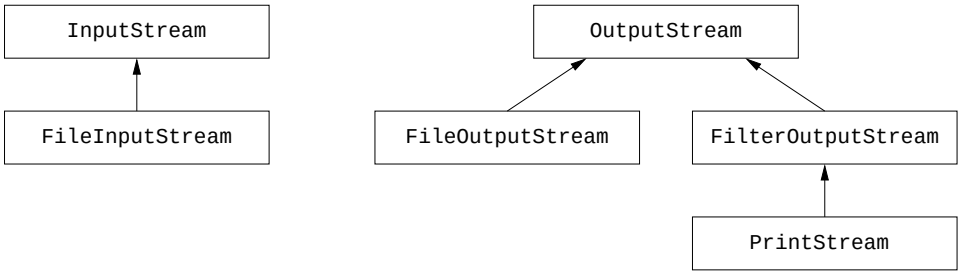
Example of input streams:

- keyboard
- file
- Internet resource

Example of output streams:

- video
- file

All classes used to define the various streams are derived from the classes `InputStream` and `OutputStream` according to the following diagram.




This common handling of all input/output devices allow us to define the readers for the input channels and the writers for the output channels independently from the specific device used to realize the input/output (as we will see in the following).

## 9.13 Input streams

### Keyboard

We use the predefined object `System.in` of the class `InputStream`.

*Example:*

```
InputStream is = System.in;
```

### File

We create an object of the class `FileInputStream` (subclass of `InputStream`) associated to the `File` object that identifies the file to open for reading.

*Example:*

```
FileInputStream is = new FileInputStream("data.txt");
```

We could also create a `File` object starting from the filename, and use such an object to create the `FileInputStream`:

```
File f = new File("data.txt");
FileInputStream is = new FileInputStream(f);
```

### Internet

We create an object of the class `URL` (defined in the librery `java.net`) that identifies the Internet resource we want to open for reading, and we invoke the method `openStream()`, which returns the corresponding `InputStream` object.

*Example:*

```
URL u = new URL("http://www.inf.unibz.it/");
InputStream is = u.openStream();
```

## 9.14 Output streams

**Video**

We use the predefined object `System.out` of the class `PrintStream` (subclass of `OutputStream`).

*Example:*

```
OutputStream os = System.out;
```

**File**

We create an object of the class `FileOutputStream` (subclass of `OutputStream`) associated to the `File` object that identifies the file to open for writing.

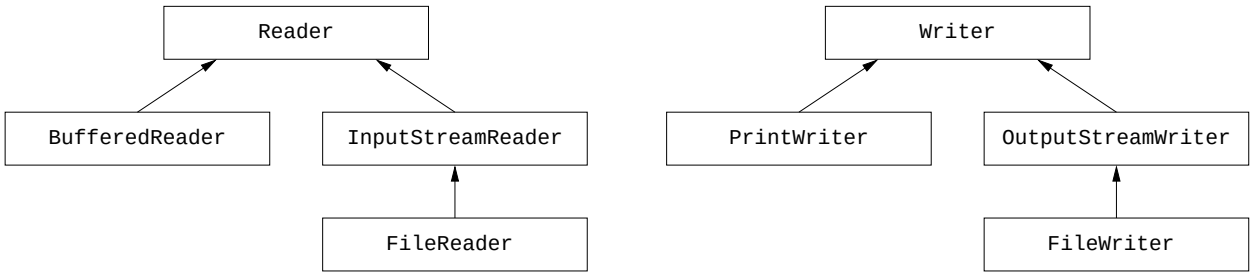
*Example:*

```
FileOutputStream os = new FileOutputStream("data.txt");
```

*Note:* It is not possible to write to an Internet resource by simply using an output stream.

## 9.15 Reading from an input stream and writing to an output stream

The operations for reading from input streams and writing to output streams are standardized, i.e., they do not depend on the particular stream being used. This means that we can use the same Java statements to read strings from a keyboard, from a file, or from the Internet. Such operations make use of the classes shown in the following diagram.




*Example 1:* reading from an input stream

```
InputStream is = ...; // keyboard, file or Internet
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
String line = br.readLine();
```

*Example 2:* writing to an output stream (1)

```
OutputStream os = ...; // video or file
PrintWriter pw = new PrintWriter(os);
pw.println("Hello");
```

*Example 3:* writing to an output stream (2)

```
OutputStream os = ...; // video or file
PrintStream ps = new PrintStream(os);
ps.println("Hello");
```

## 9.16 A program that (sometimes) determines the plural of a word

```
import java.io.*;

public class Plural {
  public static void main(String[] args) throws IOException {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader keyboard = new BufferedReader(isr);
    System.out.println("Insert a word:");
    String line = keyboard.readLine();
```

```
      System.out.print(line);
      System.out.println("s");
    }
  }
```

## 9.17 Interleaving reading and writing

In general, a program performs both input and output and these are interleaved. Output operations are internally handled via an *output buffer*, and to ensure that interleaved input and output function properly from the point of view of the user interacting with the program, we have to empty the output buffer before obtaining the next input. This can be done by using the `flush()` method associated to an `OutputStream` object.

```
import java.io.*;

public class Plural2 {
  public static void main(String[] args) throws IOException {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader keyboard = new BufferedReader(isr);
    System.out.print("Insert a word: ");
    System.out.flush();
    String line = keyboard.readLine();
    System.out.println("The plural of " + line + " is " + line + "s.");
  }
}
```

*Note:* Depending on how the `OutputStream` was created, the `flush()` operation may be performed automatically by the methods that write on the stream.

## 9.18 Generalized input and output

The next program shows how to separate the creation of the input/output streams from the operations to read from and write to such streams.

```
import java.io.*;
import java.net.*;

public class IOStreams {

  public static void readWrite(InputStream in, OutputStream out)
                                                        throws IOException {
    InputStreamReader isr = new InputStreamReader(in);
    BufferedReader br = new BufferedReader(isr);
    PrintStream ps = new PrintStream(out);
    String line = br.readLine();
    while (line != null) {
      ps.println(line);
      line = br.readLine();
    }
  }

  public static void main(String[] args) throws IOException {
    // Reads from the Internet and writes to a file
    System.out.println("*** Internet -> File ***");
    URL u = new URL("http://www.inf.unibz.it/");
    InputStream net = u.openStream();
    File fout = new File("CShome.html");
    FileOutputStream fileos = new FileOutputStream(fout);
    readWrite(net, fileos);
    fileos.close();

    // Reads from a file and writes to video
    System.out.println("*** File -> Video ***");
    File fin = new File("CShome.html");
```

```
      FileInputStream fileis = new FileInputStream(fin);
      readWrite(fileis, System.out);
      fileis.close();
    }
  }
```

## 9.19 Static method for reading

In order to construct an object according to data coming from an input channel (e.g., a text file), it is common to define a static method that reads the data from the input channel and constructs and returns an object of the class in which the method is defined.

For example, consider the class `Person`, where each object of the class is characterized by the information about its name and surname. Suppose that the information on the persons that we want to deal with in an application is stored in a text file that contains for each person two lines as follows:

*name*
*surname*

Then, we can define in the class `Person` a static method for reading the data from the file and creating a `Person` object as follows:

```
import java.io.*;

public class Person {

  private String name, surname;

  public Person (String n, String c) {
    name = n; surname = c;
  }

  public static Person read (BufferedReader br) throws IOException {
    String s = br.readLine();
    if (s == null)
      return null;
    else
      return new Person(s, br.readLine());
  }
}
```

*Note:* The static method `read()` returns the value `null` if on the input channel no more data on persons is available, or an object of the class `Person` containing the data read from the input channel. The method can be used to read and process the data about all persons contained in a file by making use of the following loop:

```
BufferedReader br = ...;
Person p = Person.read(br);
while (p != null) {
   process p
   p = Person.read(br);
}
```

## 9.20 Exercise: car sales

*Specification:*

We want to realize a program for handling new and used cars to sell. For each car to sell, the information about the car is stored on a text file.

Realize a class `Car`, to handle a single car. Each car is characterized by the following information: model, manufacturing year, driven kilometers, and price. New cars can be distinguished by the fact that the driven kilometers are 0.

The class `Car` should export the following methods:

- a constructor to construct a car object, given all the data about the car as parameters;

- suitable `get`-methods to obtain the data about the car;
- a `toString()` method, which does overriding of the `toString()` method inherited from `Object`, and returns a string containing the data about the car;
- `boolean equalTo(Car c)` : that returns `true` if the car coincides with the car `c` in all of its data, and `false` otherwise.

Realize a class `CarList`, each of whose objects represents a list of cars to sell. A `CarList` object does not directly store the data about the cars to sell, but maintains the name of a text file in which such data are stored, according to the following format:

*model*
*manufacturing year*
*driven kilometers*
*price*

The class `CarList` should export the following methods:

- `CarList(String filename)` : constructor with a parameter of type `String`, representing the name of the file in which the data about the cars to sell are stored;
- `int countNewCars()` : that returns the number of new cars in the list of cars to sell;
- `Car mostExpensiveCar()` : that returns the `Car` object corresponding to the most expensive car in the list of cars to sell.
- `void addCar(Car c)` : that adds the car `c` to the end of the list of cars to sell.
- `void remove(Car c)` : that removes from the list of cars to sell the car whose data coincides with that of `c`, if such a car is present, and leaves the list unchanged otherwise.

## 9.21 Car sales: representation of the objects

We determine the instance variables to use for representing objects of the classes `Car` and `CarList`.

**The class** `Car`

```
public class Car {
  // representation of the objects of the class
  private String model;
  private int year;
  private int km;
  private double price;

  // public methods that realize the requested functionalities
}
```

**The class** `CarList`

```
public class CarList {
  // representation of the objects of the class
  private String filename;

  // public methods that realize the requested functionalities
}
```

## 9.22 Car sales: public interface

We can now choose the interface of the classes, through which the clients can make use of the objects of the classes `Car` and `CarList`. Specifically, for each functionality we have to define a public method that realizes it and determine its header.

**The class** `Car` has the following skeleton

```
public class Car {
  // representation of the objects of the class
  private String model;
  private int year;
  private int km;
  private double price;
```

```
  // public methods that realize the requested functionalities
  public Car(String m, int y, int k, double p) { ... }
  public String toString() { ... }
  public String getModel() { ... }
  public int getYear() { ... }
  public int getKm() { ... }
  public double getPrice() { ... }
  public boolean equalTo(Car c) { ... }
  public static Car read(BufferedReader br) throws IOException { ... }
}
```

**The class** `CarList` has the following skeleton

```
public class CarList {
  // representation of the objects of the class
  private String filename;

  // public methods that realize the requested functionalities
  public CarList (String fn) { ... }
  public int countNewCars()  throws IOException { ... }
  public Car mostExpensiveCar() throws IOException { ... }
  public void addCar(Car c) throws IOException { ... }
  public void removeCar(Car c) throws IOException { ... }
}
```

## 9.23 Car sales: solution

**The class** `Car`

```
import java.io.*;

public class Car {

  private String model;
  private int year;
  private int km;
  private double price;

  public Car(String m, int y, int k, double p) {
    model = m;   year = y;   km = k;   price = p;
  }

  public String toString() {
    return model + ", year: " + year + ", km: " + km + ", price: " + price;
  }

  public String getModel() {
    return model;
  }

  public int getYear() {
    return year;
  }

  public int getKm() {
    return km;
  }

  public double getPrice() {
    return price;
  }
```

```
  public boolean equalTo(Car c) {
    return this.model.equals(c.model) &&
      this.year == c.year &&
      this.km == c.km &&
      this.price == c.price;
  }

  public static Car read(BufferedReader br) throws IOException {
    String s = br.readLine();
    if (s == null)
      return null;
    else
      return new Car(s, Integer.parseInt(br.readLine()),
                     Integer.parseInt(br.readLine()),
                     Double.parseDouble(br.readLine()));
  }
}
```

**The class** `CarList`

```
import java.io.*;

public class CarList {

  private String filename;

  public CarList (String fn) {
    filename = fn;
  }

  public int countNewCars()  throws IOException {
    FileReader fr = new FileReader(filename);
    BufferedReader br = new BufferedReader(fr);

    int count = 0;
    Car c = Car.read(br);
    while (c != null) {
      if (c.getKm() == 0)
        count++;
      c = Car.read(br);
    }
    br.close();
    return count;
  }

  public Car mostExpensiveCar() throws IOException {
    // this is the second alternative for opening a file for reading
    FileInputStream is = new FileInputStream(filename);
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);

    double max = 0;
    Car cmax = null;
    Car c = Car.read(br);
    while (c != null) {
      if (c.getPrice() > max) {
        max = c.getPrice();
        cmax = c;
      }
      c = Car.read(br);
    }
    br.close();
```

```
    return cmax;
  }

  public void addCar(Car c) throws IOException {
    FileWriter fw = new FileWriter(filename, true); //open file in append mode
    PrintWriter pw = new PrintWriter(fw);

    pw.println(c.getModel());
    pw.println(c.getYear());
    pw.println(c.getKm());
    pw.println(c.getPrice());
    pw.close();
  }

  public void removeCar(Car c) throws IOException {
    File f = new File(filename);
    FileInputStream is = new FileInputStream(f);
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);

    // this is the second alternative for opening a file for writing
    File ftemp = new File("cars-temporary.txt");
    FileOutputStream os = new FileOutputStream(ftemp);
    PrintWriter pw = new PrintWriter(os);

    Car curr = Car.read(br);
    while (curr != null) {
      if (!curr.equalTo(c)) {
        pw.println(curr.getModel());
        pw.println(curr.getYear());
        pw.println(curr.getKm());
        pw.println(curr.getPrice());
      }
      curr = Car.read(br);
    }
    br.close();
    pw.close();

    ftemp.renameTo(f);
  }
}
```

## Exercises

**Exercise 9.1.** Write a class `IOStrings` containing two public static methods:

- `String[] loadArray(InputStream is, int n)` : that returns an array of `n` strings read from the input channel specified by `is`;
- `void saveArray(OutputStream os, String[] sa)` : that writes the array of strings `sa` to the output channel specified by `os`.

**Exercise 9.2.** Write a public static method that reads from the keyboard a sequence of positive integers until the value 0 is inserted, and returns a `double` representing the average of the read values (without considering the final 0).

**Exercise 9.3.** Write a public static method that prints on the video all lines of a file that start with one of the characters `'/'`, `';'`, or `'%'`. The name of the file should be given as a parameter.

**Exercise 9.4.** Write a public static method that takes as parameters a string representing a filename and an integer $n$, and writes on the file the multiplication table of size $n$. For example, for $n = 3$, the method should write:

```
1  2  3
2  4  6
3  6  9
```

**Exercise 9.5.** Write a public static method that reads from a file information about exam marks obtained by students, organized as follows:

```
Rossi 25 24 26 30 24 30
Bianchi 20 24 25
Verdi 30 24 30 27
```

The method should return a string representing the name of the student with the highest average marking.

**Exercise 9.6.** Realize a class `IOFile` that exports some functionalities on text files. The class should have a constructor with one parameter of type `String`, representing the name of the file on which to operate, and should export the following methods:

- `int countLines()` : that returns the number of lines of the file;
- `void write(OutputStream os)` : that writes the content of the file to `os`;
- `void print()` : that prints the content of the file to the video;
- `void copy(String filename)` : that copies the content of the file to the file specified by `filename`;
- `void delete()` : that deletes the file from mass-storage.

**Exercise 9.7.** Realize a class `HandleBAs` to handle a set of bank accounts, represented as in Units 4 and 5. The data for the bank accounts are stored on a text file according to the following format:

*name*
*surname*
*balance*

The class `HandleBAs` should export the following methods:

- `HandleBAs(String filename)` : constructor that takes as parameter the name of a text file on which the data about the bank accounts is stored, and creates a object associated to that file;
- `void interests(double rate)` : that updates the balance of all accounts by applying the interest rate specified by the parameter;
- `printNegative()` : that prints on the video the data about all bank accounts with a negative balance.

**Exercise 9.8.** Add to the class `Apartment` of Exercise 7.12 a public method `saveToFile` that takes as a parameter a filename and saves the data about the apartment to the file. Add also a public static method `readFromFile` that takes as a parameter a `BufferedReader` object specifying an input channel, reads from the channel the data about an apartment (organized as written by `saveToFile`), and constructs and returns the apartment. Then, write a static method, client of the class `Apartment`, that takes as parameter a filename and prints on the video all the information about the apartments on the file.

**Exercise 9.9.** Realize a Java class `Matrix` to represent bidimensional matrices of real numbers. The class should export the following methods:

- `Matrix(int n, int m)` : constructor that creates a matrix of size `n`x`m`, with all values initially set to 0;
- `void save(String filename)` : that saves the content of the matrix on the file specified by `filename`;
- `static Matrix read(String filename)` : that reads the data about a matrix from the file specified by `filename`, creates the matrix, and returns it;
- `Matrix sum(Matrix m)` : that returns the matrix that is the sum of the object and of `m`, if the two matrices have the same dimensions, and `null` otherwise;
- `Matrix product(Matrix m)` : that returns the matrix that is the product of the object and of `m`, if the two matrices have compatible dimensions, and `null` otherwise.