File management

Accessing text files

One important aspect of every application is to store information somehow. Programs usually store the data they are managing, so that they can be retrieved later. An easy way of doing this is through text files.

Java includes a wide variety of classes for reading and writing from/to files. We are not going to learn all of them, but the most useful ones. All of them inherit from two abstract, parent classes called `Reader` and `Writer`, that define some basic behavior to deal with these files, such as reading or writing small pieces of data. Every class that we are going to cover here belongs to `java.io` package.

1. Compulsory steps when dealing with text files

When we are managing text files (or files, in general), there are some issues that we must take into account. The first one is that there can be **exceptions** in this process, such as:

- The file that we are trying to read does not exist. This throws a `FileNotFoundException`.
- There's an error trying to read or write the file, because of a hard drive damage or any other reason. In this case, an `IOException` is being thrown.

Java will produce a compilation error if we don't deal with these exceptions properly. As `FileNotFoundException` is a subclass of `IOException`, we can just catch or deal with this last exception in order to manage the possible errors, unless we want to show a specific, different error message for both situations.

The second issue to which we must pay attention is that dealing with files also causes a memory usage: data being read or written is temporarily stored in memory. In order to free this memory when we finish processing the file, we must **close** the file. This is specially important when writing files, since some text (or all of it) may not have been written yet.

2. Writing in text files

There are some files that we can use to write information into text files. All of them extend `Writer` class functionality, but some of them are more useful than the others.

A text file can be used like a screen, so that we can write data on it line by line with an instruction. Using a `PrintWriter` object is one of the easiest ways for this, since it provides methods such as `print`, `println` or even `printf`, similar to the ones used when dealing with the system console. This is an introductory example that writes two lines in a file called "*example.txt*".

```
import java.io.PrintWriter;

public class PrintWriter0
{
    public static void main(String[] args)
    {
        PrintWriter printWriter = new PrintWriter ("example.txt");
        printWriter.println ("Hello!");
        printWriter.println ("Goodbye!");
    }
}
```

As we have said before, when dealing with files, we must manage the possible exceptions that can be produced, and also close the file at the end, so this example should be written this way:

```
import java.io.*;

public class PrintWriter1
{
    public static void main(String[] args)
    {
        try
        {
            PrintWriter printWriter = new PrintWriter ("example.txt");
            printWriter.println ("Hello!");
            printWriter.println ("Goodbye!");
            printWriter.close ();
        }
        catch (IOException e)
        {
            System.err.println("Error writing file: " + e.getMessage());
        }
    }
}
```

There can also be a problem with this program: if there is an error, the file may remain open, so we would get a memory leak. To avoid this, we should close the file with a "*try-catch-finally*" block. Every piece of code placed in the `finally` block will be run anyway (whether the process works or fails).

```
import java.io.*;

public class PrintWriter2
{
    public static void main(String[] args)
    {
        PrintWriter printWriter = null;
        try
        {
            printWriter = new PrintWriter ("example.txt");
            printWriter.println ("Hello!");
            printWriter.println ("Goodbye!");
        }
        catch (IOException e)
        {
            System.err.println("Error writing file: " + e.getMessage());
        }
        finally
        {
            if (printWriter != null)
            {
                printWriter.close();
            }
        }
    }
}
```

2.1. Appending text at the end of the file

The constructor that we have used for *PrintWriter* in previous examples removes any existing contents of the file, if it exists. So, if we want to add new content to the existing one, we must use another constructor. It does not get the file path, but a `BufferedWriter` that relies on a `FileWriter` with a boolean as a second parameter. If this boolean is *true*, then new contents will be added to the end of the file (without erasing previous contents).

```
import java.io.*;

public class PrintWriter3
{
    public static void main(String[] args)
    {
        PrintWriter printWriter = null;
        try
        {
            printWriter = new PrintWriter(new BufferedWriter(
                new FileWriter("example.txt", true)));
            printWriter.println ("Hello again!");
            printWriter.println ("Goodbye!");
        }
        catch (IOException e)
        {
            System.err.println("Error writing file: " + e.getMessage());
        }
        finally
        {
            if (printWriter != null)
            {
                printWriter.close();
            }
        }
    }
}
```

> **NOTE**: we could have used `BufferedWriter` instead of `PrintWriter` in our examples, but it is a low level class, so the code is not as similar as the one that we use with the console, because there is no *println* method, but a *write* and *newLine* methods.

2.2. Auto closing files

From Java 7, `PrintWriter` and many other I/O classes implement an interface called `AutoCloseable`. Every class that implements this interface has the ability of auto-closing its own objects, if we follow some given pattern when we instantiate these objects.

Regarding file management, files are automatically closed if we instantiate the variables in the `try` clause. This concept is also known as **try-with-resources**, since we are adding resources to the original *try* clause. This is how it works regarding `PrintWriter` objects:

```
import java.io.*;

public class PrintWriter4
{
    public static void main(String[] args)
    {
        try (PrintWriter printWriter = new PrintWriter ("example.txt"))
        {
            printWriter.println ("Hello!");
            printWriter.println ("Goodbye!");
        }
        catch (IOException e)
        {
            System.err.println("Error writing file: " + e.getMessage());
        }
    }
}
```

Note that code is shorter and less error prone this way. As soon as `try` block finishes, file is automatically closed, even if there's any exception, so we don't have to care about it.

> **Exercise 1:**
>
> Create a program that asks the user to enter two sentences and stores them in a file called "twoSentences.txt".

> **Exercise 2:**
>
> Create a program that asks the user to enter sentences until he types an empty string. It must store the sentences in a file called "sentences.txt". Every time we run the program again, the file must be destroyed and replaced with a new one.

> **Exercise 3:**
>
> Create a new version of previous exercise in which the file will be called "annotations.txt" and will not be destroyed whenever we run the program. The new contents will be added at the end of the existing ones.

> **Exercise 4:**
>
> Create a program that asks the user to enter the days of a month, the month name and the number for the first day (1 for Monday, 7 for Sunday). Then, it will create a text file with the month name (for instance, "*March.txt*"), with an agenda for this month. For instance, if the month is "September", which has 30 days and starts in day 4 (Thursday), the contents of file "*September.txt*" should be as follows:

```
September

------------------------------------------------------------
Thursday 1:
------------------------------------------------------------
Friday 2:
------------------------------------------------------------
Saturday 3:
------------------------------------------------------------
Sunday 4:
------------------------------------------------------------
Monday 5:
------------------------------------------------------------
(...)
------------------------------------------------------------
Friday 30:
------------------------------------------------------------
```

Exercise 5:

Create a program that asks the user the same information than in previous exercise (the number of days of a month, the month name and the starting day) and creates a text file with the month name with suffix "calendar" (for instance, "*MarchCalendar.txt*"). The text file must contain the calendar for this month. Here you can see an example for September, starting on Thursday:

```
September

mon tue wed thu fri sat sun

                1   2   3   4
  5   6   7   8   9  10  11
 12  13  14  15  16  17  18
 19  20  21  22  23  24  25
 26  27  28  29  30
```

3. Reading from text files

An easy way to read from a text file, if we can do it line by line, is to use a `BufferedReader` object, that relies on a *FileReader* instance, and includes a `readLine` method that returns every line (*String*) from the file. If the String is null, then the end of the file has been reached, so we usually use a *while* loop to detect this (or a *do-while*):

```
import java.io.*;

class BufferedReader1
{
    public static void main( String[] args )
    {
        BufferedReader inputFile = null;

        try
        {
            inputFile = new BufferedReader(
                    new FileReader(new File("example.txt")));

            String line = inputFile.readLine();
            while (line != null)
            {
                System.out.println(line);
                line = inputFile.readLine();
            }
        }
        catch (IOException fileError)
        {
            System.err.println(
                "Error reading file: " +
                fileError.getMessage() );
        }
        finally
        {
            try
            {
                inputFile.close();
            } catch (Exception e) {}
        }

        System.out.println("Reading finished.");
    }
}
```

We can also join the reading and checking in the same order, so the code is more compact, but less readable:

```
String line=null;
while ((line=inputFile.readLine()) != null)
{
    System.out.println(line);
}
```

Besides, we can make use of the `AutoCloseable` interface and avoid the `finally` clause, just like we did with *PrintWriter*:

```
import java.io.*;

class BufferedReader1
{
    public static void main( String[] args )
    {
        try (BufferedReader inputFile = new BufferedReader(
                new FileReader(new File("example.txt"))))
        {

            String line = null;
            while ((line = inputFile.readLine()) != null)
            {
                System.out.println(line);
            }
        }
        catch (IOException fileError)
        {
            System.err.println(
                "Error reading file: " +
                fileError.getMessage() );
        }

        System.out.println("Reading finished.");
    }
}
```

3.1. Checking if file exists

When we write data into a file we (usually) don't need to care if file exists or not, because if it doesn't exist, it will be created anyway. However, when reading a file, we often need to make sure that file exists before reading it (unless we want to delegate this issue in a *FileNotFoundException*). This piece of code before instantiating the *BufferedReader* object can help:

```
if (! (new File("example.txt")).exists() )
{
    System.err.println("File example.txt not found");
}
else
{
    // Read the file normally
}
```

> **Exercise 6:**
>
> Create a program that shows the first line of the file "twoSentences.txt" created in previous exercises.

> **Exercise 7:**
>
> Create a program that shows all the contents of the file "annotations.txt" created in previous exercises.

> **Exercise 8:**
>
> Create a program that shows the contents of the file "annotations.txt" page by page: after every 23 lines there will be a pause until the user types Enter.

> **Exercise 9:**
>
> Create a program that asks the user to enter a file name and a word to look for. The program must show every line in the text file that contains the given word.

> **Exercise 10:**
>
> Create a program that asks the user to enter a file name. The contents of the file will be stored in an *ArrayList* (line by line), and then the program will continuously ask the user to enter a word, and show all the lines in the array list that contain the specified word. If the word is not found in the array list, then the program must show "Not found". This process must be repeated until the user types an empty string.

3.2. Reading char by char

Instead of reading a text file line by line, we may need to read it char by char. In this case we can use `FileReader` class, which provides some methods to read the chars of the file, such as `read` method. This method returns an integer with the code of the char, or -1 if the end of file has been reached (that's why the function uses `int` instead of `char`, to return -1 in this last case).

```
int charRead;

try(FileReader reader = new FileReader("example.txt"))
{
    do
    {
        charRead = reader.read();
        if (charRead != -1)
            System.out.println((char)charRead);
    }
    while(charRead != -1);
}
catch (IOException ex)
{
    System.err.println("Error readig file: " + ex.getMessage());
}
```

3.3. Extending performance

It's possible to write your own implementation of one of these *java.io* classes using inheritance and add some new methods or modify some functionality. For instance, this class inherits from `BufferedReader` class to return every line in upper case:

```
public class UpperCaseReader extends BufferedReader
{
    public UpperCaseReader(Reader reader)
    {
        super(reader);
    }

    @Override
    public String readLine() throws IOException
    {
        String line = super.readLine();
        return line != null?line.toUpperCase():null;
    }
}
```

We would use this custom class in a main program this way:

```
try(UpperCaseReader buffer =
    new UpperCaseReader(new FileReader("file.txt")))
{
    String line;
    while((line = buffer.readLine()) != null)
    {
        System.out.println(line);
    }
} catch (IOException e1) {
    ...
}
```