by Nacho Cabanes and Javier Carrasco

Development Environments
Block 3
Unit 2: 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.

3.2.1. Writing in 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.

A text file can be used like a screen, so that we can write data on it line by line with a "*println*" instruction. Using a `PrintWriter` object is one of the easiest ways for this. We *must* use a *try-catch* block to get every possible exception when accessing the file (such as *FileNotFoundException* or *IOException*, which is more general).

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class PrintWriter1 {
    public static void main(String[] args) {
        try {
            PrintWriter printWriter = new PrintWriter ("example.txt");
            printWriter.println ("Hello!");
            printWriter.close ();
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Alternatively, we can use a more compact way (which is not as elegant as the code above, but can be suitable for small source files). It consists in throwing the exception instead of catching it, this way:

```java
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class PrintWriter1b {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter printWriter = new PrintWriter ("example.txt");
        printWriter.println ("Hello!");
        printWriter.close ();
    }
}
```

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. Besides, we can intercept any possible I/O error with **IOException**.

```java
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 ("and...");
            printWriter.println ("goodbye!");
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            if (printWriter != null) {
                printWriter.close();
            }
        }
    }
}
```

The constructor that we have used for *PrintWriter* removes previous 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).

```java
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 ("and...");
            printWriter.println ("goodbye!");
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            if (printWriter != null)  {
                printWriter.close();
            }
        }
    }
}
```

We could have used *BufferedWriter* without using a *PrintWriter*, 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, as you can see in this basic example:

```
import java.io.*;

class BufferedWriter1 {

    public static void main( String[] args ) {
        try {
            BufferedWriter outputFile = new BufferedWriter(
                new FileWriter(new File("example.txt")));

            outputFile.write("Line 1");
            outputFile.newLine();
            outputFile.write("Line 2");
            outputFile.newLine();
        }
        catch (IOException fileError) {
            System.out.println(
                "There has been some problems: " +
                fileError.getMessage() );
        }
        finally {
            if (outputFile != null)  {
                outputFile.close();
            }
        }
    }
}
```

Proposed exercises:

3.2.1.1. Create a program that asks the user to enter two sentences and stores them in a file called "twoSentences.txt".

3.2.1.2. 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.

3.2.1.3. 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.

3.2.1.4. Create a new version of previous exercise in which we will add the date and time in which every annotation is being written. You can use `java.time.LocalDateTime` class, with its static `now` method.

3.2.1.5. Create a program that asks the user to enter the width and height of a rectangle, and then creates a file called "rectangle.txt" with a rectangle of asterisks with the given width and height. For instance, if the user specifies a width of 5 and a height of 3, the contents of the file would be:

```
*****
*****
*****
```

3.2.1.6. 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:
-----------------------------------------------------------
```

3.2.1.7. 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
```

3.2.2. 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 ) {

        // First we check if file exists
        if (! (new File("example.txt")).exists() ) {
            System.out.println("File example.txt not found");
            return;
        }

        // If it exists, we try to read it
        System.out.println("Reading file...");

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

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

            inputFile.close();
        }
        catch (IOException fileError) {
            System.out.println(
                "There has been some problems: " +
                fileError.getMessage() );
        }

        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);
}
```

Proposed exercises:

3.2.2.1. Create a program that shows the first line of the file "twoSentences.txt" created in previous exercises.

3.2.2.2. Create a program that shows all the contents of the file "annotations.txt" created in previous exercises.

3.2.2.3. 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.

3.2.2.4. Create a program that shows the contents of the text file whose name must be entered by the user. The program must show an error message if the file does not exist.

3.2.2.5. Create a program that reads the contents of a text file and stores them in another text file, converting the contents to upper case.

3.2.2.6. 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.

3.2.2.7. Create a program that reads the contents of file "rectangle.txt" created in previous exercises, and calculate and print the width and height of the rectangle read from the file.

3.2.2.8. 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.

3.2.3. To learn more

If you want to learn more about file management in Java, you can look the following official references:

- File input output