File management

Accessing binary files

Dealing with text files can be a good option in many situations, when we want to store some simple data, or retrieve it from an existing file. However, sometimes information is not stored in text mode, and we may need to access it. This is the case of many file types in our day to day life, such as audio files or images. In these files, information is stored as a sequence of **bytes**, and we can't just open them and read them with text editors.

Regarding binary files, Java provides two base, abstract classes called `InputStream` and `OutputStream`, which provide some simple methods to read and write bytes (or groups of bytes) from/to these files. In this document we will see some easy steps to manage these classes and their subtypes, and do some simple operations.

1. Reading from binary files

`FileInputStream` class provides a `read` method to read a binary file byte by byte. It returns -1 when the end of file has been reached.

```java
try(FileInputStream fIn = new FileInputStream("file.jpg"))
{
    int data;

    while ((data = fIn.read()) != -1)
    {
        System.out.println(data);
    }
}
catch(IOException ex)
{
    System.err.println("Error reading file: " + ex.getMessage());
}
```

The class also has another overloaded version of this method to read a bunch of bytes and store it in an array.

```
byte[] data = new byte[100]:

try(FileInputStream fIn = new FileInputStream("file.jpg"))
{
    int size = 0;

    while ((size = fIn.read(data)) != -1)
    {
        ...
    }
}
catch(IOException ex)
{
    System.err.println("Error reading file: " + ex.getMessage());
}
```

In this last case, `read` method returns the total number of bytes read, or -1 if the end of file has been reached. Data read will be stored in the byte array that we pass as parameter.

2. Writing to binary files

`FileOutputStream` class has a `write` method to write a byte to a binary file.

```
try(FileOutputStream fOut = new FileOutputStream("file.dat"))
{
    fOut.write(23);
    fOut.write(44);
}
catch(IOException ex)
{
    System.err.println("Error writing file: " + ex.getMessage());
}
```

It also has an overloaded version of this method to write an array of bytes to the file.

```java
byte[] data = new byte[100];

// ... Fill array with data

try(FileOutputStream fOut = new FileOutputStream("file.dat"))
{
    fOut.write(data);
}
catch(IOException ex)
{
    System.err.println("Error writing file: " + ex.getMessage());
}
```

3. Some advanced operations with binary files

Dealing with single bytes in a binary file is not a usual operation. However, sometimes we may need to access a concrete byte or group of bytes in a binary file to check their value, or modify it. But, instead of going byte by byte throughout the file until we reach the desired position, we can make use of **random access files**.

`RandomAccessFile` class lets us open a read and/or write stream over a binary file, so that we can move to a given position and read/change the information stored in this file. To do this, we can make use of some useful methods, such as:

- `read`, `readInt`, `readFloat` ... to read information in a given basic type
- `write`, `writeInt`, `writeFloat` ... to write information from a given basic type
- `seek(bytes)` or `skipBytes(int)` to go to a given position in the binary file

For instance, we can check if a PNG file is valid by checking the starting sequence of bytes: 137, 80, 78, 71, 13, 10, 26 and 10. This little piece of code checks this:

```java
final int[] VALID_HEADER = {137, 80, 78, 71, 13, 10, 26, 10};

try (RandomAccessFile raf = new RandomAccessFile("image.png", "rw"))
{
    int[] headerBytes = new int[VALID_HEADER.length];
    for (int i = 0; i < VALID_HEADER.length; i++)
    {
        headerBytes[i] = raf.read();
    }

    boolean ok = true;
    for (int i = 0; i < VALID_HEADER.length; i++)
    {
        if (headerBytes[i] != VALID_HEADER[i])
            ok = false;
    }

    if (ok)
    {
        System.out.println("Valid PNG");
    }
    else
    {
        System.out.println("Not valid PNG");
    }
}
catch (IOException e)
{
    System.err.println("Error processing image");
}
```

We could also modify the value of any of these bytes so that PNG turns into a not valid image, and could not be opened by an image editor:

```java
raf.seek(0);
raf.write(0);
```

Exercise 1:

Create a program that opens a BMP image file and then:

- Make sure that BMP is a valid file by checking the first two bytes. They must correspond to letters *B* and *M*.
- Determine the width and height of the BMP file by checking the integers stored at bytes 18-21 (width) and 22-25 (height).

> **HELP**: when reading data greater than a byte (you must read an integer in this case), you must know that information is sometimes stored in *little endian* format. This means that the lowest byte is stored at the leftmost position. However, if we call `readInt` method, Java tries to read the information in *big endian*, this is, lowest byte is stored at the rightmost position. So, when you read the integer containing the width or height, the bytes will be inverted. In order to reverse this value, you can just use `Integer.reverseBytes(value)` method.