Concurrent programming

Process management in Java

Now that we know what a process is, let's see how Java deals with them. In fact, there are just a few classes and methods that we need to know, since Java is focused on threads, not on processes (every Java main application is a thread indeed). However, there are some functionalities added that allow us to call external programs or create processes from a Java application.

1. Creating processes

To create a process in Java, we need to get a `Process` object. This can be achieved by two different ways:

- Using the `ProcessBuilder` class. We need to create an array of strings with the name of the program to run and its arguments, and then, call the `start` method.

```java
String[] cmd = {"ls", "-l"};
ProcessBuilder pb = new ProcessBuilder(cmd);
Process p = pb.start();
```

- Using the `Runtime` class. We also need to create an array of strings with the name of the program and its arguments, and then we call the `exec` method with that array as a parameter.

```java
String[] cmd = {"notepad.exe"};
Runtime rt = Runtime.getRuntime();
Process p = rt.exec(cmd);
```

In both cases, we are running an existing command or program in the operating system where Java is currently running. It can be a Linux shellscript, a Windows *exe* file, or even another Java application through a *java* command. If the program can't be found, or we do not have permission to run it, an exception will be thrown when we try to call the `start` or `exec` methods from `ProcessBuilder` or `Runtime` classes, respectively. This exception will be a subtype of `IOException`.

```
try
{
    Process p = pb.start();
    ...
} catch (IOException e) {
    System.err.println("Exception: " + e.getMessage());
    System.exit(-1);
}
```

1.1. Differences between *ProcessBuilder* and *Runtime*

You may be wondering... why are there two ways of doing the same thing? Well, `Runtime` class belongs to Java core since its very first version, whereas `ProcessBuilder` was added in Java 5. With `ProcessBuilder` you can add environment variables and change the current working directory for the process to be launched. Such features are not available for `Runtime` class. Besides, there are some subtle differences between these two classes. For instance, `Runtime` class lets us execute a command by passing the whole string as an argument, without dividing it into separate arguments in an array:

```
Process p = Runtime.getRuntime.exec("ls -l");
```

1.2. Using *ProcessHandle* (Java 9+)

Starting from Java 9, Java introduced the `ProcessHandle` interface, which provides more functionalities for managing and controlling processes. With `ProcessHandle`, you can get information about the process, such as its PID, and handle process termination.

```
ProcessBuilder pb = new ProcessBuilder("ls", "-l");
Process p = pb.start();
ProcessHandle handle = p.toHandle();
System.out.println("Process ID: " + handle.pid());
handle.onExit().thenRun(() -> System.out.println("Process exited"));
```

2. Synchronizing processes

We have just learnt how to create and launch a process in Java. After calling the `start` or `exec` method, our Java program keeps going, and it runs its next instruction. If we want it to stop until the subprocess finishes its task, we can call the `waitFor` method from the `Process` object that we created. This causes the main program to halt until this process is completed.

Calling the `waitFor` method can throw an `InterruptedException` if the subprocess has been interrupted unexpectedly. If everything is OK, then the control comes back to the Java main application as soon as the subprocess finishes.

```
try
{
    Process p = pb.start();
    p.waitFor();
    ...
} catch (IOException e) {
    System.err.println("Exception: " + e.getMessage());
    System.exit(-1);
} catch (InterruptedException e) {
    System.err.println("Interrupted: " + e.getMessage());
}
```

Note: From Java 8 onwards, you can also use the `waitFor(long timeout, TimeUnit unit)` method to specify a timeout for waiting.

The `waitFor` method returns an integer value. This value is usually a 0 when the process has finished correctly, and any other number if it finished unexpectedly. So we can check the final state of a process by comparing its return value:

```
int value = p.waitFor();
if (value != 0)
    System.out.println("The task finished unexpectedly");
```

3. Finishing processes

We can finish a process that we previously created in our program by calling the `destroy` method. By doing this, the Java *garbage collector* will free all the resources associated to that process.

```
ProcessBuilder pb = new ProcessBuilder(...)
Process p = pb.start();
...
p.destroy();
```

Note: If you need to forcibly terminate the process, you can use the `destroyForcibly` method, which ensures that the process is terminated immediately.

4. Communicating with processes

A process usually needs to get some information (from the user, or from a file, for instance), and output some results (to a file, to a screen...). In many operating systems, when a process is using a given input/output, its children use the same input/output. In other words, if a process is reading data from a file as its standard input, and it creates a subprocess, this subprocess will also have the same file as its default input.

However, Java does not have such behavior. When a process is created in Java from another (parent) process, it has its own communication interface. If we want to communicate with this subprocess, we have to get its input and output streams. By doing this, we will be able to send data to that subprocess from its parent process, and get its results from its parent as well.

Note: You can also use `ProcessBuilder.redirectOutput` and `ProcessBuilder.redirectInput` to redirect the standard input and output streams of a process.

The following example gets the output of the subprocess and prints it to the screen:

```
Process p = pb.start();
BufferedReader br = new BufferedReader(
    new InputStreamReader(p.getInputStream()));
String line = "";

System.out.println("Process output:");
while ((line = br.readLine()) != null)
{
    System.out.println(line);
}
```

Using try-with-resources:

```
Process p = pb.start();
try (BufferedReader br = new BufferedReader(
        new InputStreamReader(p.getInputStream()))) {
    String line;
    System.out.println("Process output:");
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

There is something you must know when you deal with your process data. Some operating systems (such as Linux, Android, Mac OS X...) use UTF-8 as their encoding format, whereas other systems (Windows) use their own encoding format. This can be a problem if, for instance, we save a text file in Linux and we read it in Windows. To avoid these problems, we can use a second argument when creating the `InputStreamReader` object, to tell the JVM which is the expected encoding format for the input:

```
BufferedReader br = new BufferedReader(
new InputStreamReader(p.getInputStream(), "UTF-8"));
```

5. Example

This example creates a process to call the *"ls"* command (it is expected to run on Linux or Mac OS X), with the option *"-l"* to have a detailed list of files and folders from current directory. Then, it captures the output and prints it in the console (or standard output).

```
import java.io.*;

public class FolderListing
{
    public static void main(String[] args)
    {
        String[] cmd = {"ls", "-l"};
        String line = "";
        ProcessBuilder pb = new ProcessBuilder(cmd);

        try
        {
            Process p = pb.start();
            BufferedReader br = new BufferedReader(
                new InputStreamReader(p.getInputStream()));
            System.out.println("Process output:");
            while ((line = br.readLine()) != null)
            {
                System.out.println(line);
            }
        } catch (Exception e) {
            System.err.println("Exception:" + e.getMessage());
        }
    }
}
```

Exercise 1:

Create a project called **ProcessListPNG** with a program that asks the user to introduce a path (for instance, */myfolder/photos*), and then launches a process that prints a list of all PNG images found in this path. Try to do it recursively (either with a command from the operating system or with your own script).

Exercise 2:

Create a project called **ProcessKillNotepad** with a program that launches the notepad or any similar text editor from your operating system. Then, the program will wait 10 seconds for the subprocess to finish and, after that period, it will be destroyed. To sleep 10 seconds, use this instruction:

```
Thread.sleep(milliseconds);
```