Process Management in Java – ProcessBuilder and Process

In the java.lang package, we have two classes for process management:

- java.lang.ProcessBuilder
- java.lang.Process

ProcessBuilder instances manage the **attributes** of **processes**, while **Process** instances control the **execution** of those same processes when they are executed.

Before running a new **process**, we can configure the execution parameters of the process using the **ProcessBuilder** class.

ProcessBuilder is an **auxiliary class** of the **Process class**, which we will see later, and it is used to control some execution parameters that will affect the process. Through the call to the **start** method, a new process is created in the system with the **attributes** defined in the ProcessBuilder instance.

```
ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
Process p = pb.start();
```

If we call the **start** method several times, new processes will be created due to the calls, all of them with the same attributes.

ProcessBuilder start method creates a new **Process** instance with the following **attributes**:

- command
- environment
- working directory
- source of input
- destination for standard output and standard error output
- redirectErrorStream

The **ProcessBuilder** class defines a couple of constructors:

```
ProcessBuilder(List<String> command)
ProcessBuilder(String... command)
```

The operation of both is the same. In the first constructor, the command to be executed and the list of arguments are passed as a list of strings. On the other hand, in the second constructor, the command and its arguments are passed through a variable number of strings (String ... this is what in Java is called varargs). The version we use depends upon the way to pass the parameters.

If we want to launch a program with parameters (modifiers that change the way a program works like -h /s ...) the command cannot be passed to the constructor directly as a single string, it must be preprocessed into a list to make it work.

```
//Different ways of passing the command to the constructors of ProcessBuilder
//First way: using a string. It fails with parameters
//Only Works with programs using arguments
String command1 = "notepad.exe prueba1.txt"
ProcessBuilder pb = new ProcessBuilder(command1)

//Second way: using an array of strings. It works with parameters
String[] command2 = {"cmd", "/c", "dir", "/o"};
ProcessBuilder pb = new ProcessBuilder(command2);

//Third way: using a string and splitting it to convert the string into a list
String command3 = "c:/windows/system32/shutdown -s -t 0";
//The regular expresión \s means cut by white spaces
ProcessBuilder pb = new ProcessBuilder(command3.split("\\s"));
//This third way always works
```

Modify the command at runtime

Maybe all or part of the command is not available at the time of calling the ProcessBuilder constructors. It can be changed, modified and queried later with the command method.

As with the constructors, we have two versions of the command method:

```
command(List<String> command)
command(String... command)
```

The **third form** of this method (without parameters) is used to obtain a list from the command passed to the constructor or set with one of the previous forms of the command method. The interesting thing is that once we have the list, we can modify it using the methods of the List class.

In the following example, when defining the command, we are missing the last part, the temporary directory. In addition, if we want to make the execution cross-platform, we don't know the shell to execute either. Depending on the OS, two values are added at the beginning and one value at the end, with the add method of the List class.

```
//Sets and modifies the command after ProcessBuilder object is created
String command = "java -jar install.jar -install"; // tmp dir is missing
ProcessBuilder pbuilder = new ProcessBuilder(command.split("\\s"));
if (isWindows) {
  pbuilder.command().add(0, "cmd"); // Sets the 1st element
  pbuilder.command().add(1, "/c"); // Sets the 2nd element
  pbuilder.command().add("c:/temp"); // Sets the last element
  //Command to run cmd /c java -jar install.jar -install c:/temp
} else {
  pbuilder.command().add(0, "sh"); // Sets the 1st element
  pbuilder.command().add(1, "-c"); // Sets the 2nd element
  pbuilder.command().add("/tmp"); // Sets the last element
  //Command to run: sh -c java -jar install.jar -install /tmp
}
//Starts the process
pbuilder.start();
```

Additional configurations of a process

Some attributes we can configure for a process are:

Set the **working directory** where the process will run. Subprocesses subsequently started by object's start() method will use it as their working directory.

We can change the default working directory by calling the directory method and passing it an object of type File. By default, the working directory is set to the value of the system variable **user.dir**. This directory is the starting point for accessing files, images and all the resources needed by our application.

```
//Change the directory to C:\users
pbuilder.directory(new File("C:\\users"));
```

The following example changes the directory to open a file named **file.txt** using **Notepad++:**

```
package org.example;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
public class ProcessBuilderDirectory {
    public static void main(String[] args) throws IOException {
        List<String> commandsList=new ArrayList<String>();
        commandsList.add("C:/Program Files/Notepad++/notepad++.exe");
        commandsList.add("file.txt");
        ProcessBuilder pb=new ProcessBuilder(commandsList);
        pb.directory(new File("src/main/resources"));
        System.out.println(pb.directory());
        Process process=pb.start();
    }
}
```

Set or modify **environment variables** for the process with the **environment()** method. This method returns process builder's environment in form of a **Map**, its initial value is a copy of the **environment** of the **current process**. One can always add new variables that will only apply to the current process builder and not to other instances of **ProcessBuilder** object.

Changing **environment variables** should only be used in a **testing context**.

```
// Retrieve and modify the process environment
Map<String, String> environment = pbuilder.environment();
//Get the PATH environment variable and add a new directory
String systemPath = environment.get("path") + ";c:/users/public";
environment.replace("path", systemPath);
//Add a new environment variable and use it as a part of the command
environment.put("GREETING", "Hello Process");
processBuilder.command("/bin/bash", "-c", "echo $GREETING")
```

Another example:

```
// Specify the directory where the exe file is
File directory = new File ("bin");
pb.directory(directorio);

// Show the information of the environment variables
Map<String, String> environment = pb.environment();
System.out.println(environment);

// Show the process name and its arguments
List<String> command = pb.command();
Iterator<String> iter = command.iterator();
while (iter.hasNext()) {
          System.out.println(iter.next());
}
```

Access to the process once it is running

The Process class is an abstract class defined in the java.lang package and contains the information of the running process. After invoking the ProcessBuilder start method, it returns a reference to the process in the form of a Process object.

The methods of the **Process** class can be used to perform **I/O operations** from the process, to check its status, its return value, to wait for it to finish executing and to force the termination of the process. However, these methods do not operate on special OS processes such as services, shell scripts, daemons, etc.

Interestingly, processes launched with the **start()** method do not have a console assigned to them. Instead, these processes redirect the standard I/O streams (stdin, stdout, stderr) to the parent process. If needed, they can be accessed through the streams obtained with the methods defined in the Process class as getInputStream(), getOutputStream() and getErrorSteam(). This is the way to send and receive information from the subprocesses.

The main methods of this class are:

Method	**Description**
int exitValue()	Completion code returned by child process
Boolean isAlive()	Checks if the process is still running
int waitFor()	It causes the parent process to wait for the child process to terminate. The value that returns is the termination code of the child process.
Boolean waitFor(long timeOut, TimeUnit unit)	The operation is the same as in the previous case, except that this time we can specify how long we want to wait for the child process to finish. We can specify how long we want to wait for the child process to finish. The method returns true if the process terminates before the specified time elapses and false if the time has passed and the process has not finished.
void destroy()	These two methods are used to kill the process. The second one does it by force.
Process destroyForcibly()	It should only be used if destroy() failed after a certain timeout.

Let's look at a simple example. Once the program is launched, it waits for 10 seconds and then kills the process.

```
public class ProcessDemo {
        public static void main(String[] args) throws Exception {
          ProcessBuilder pb = new ProcessBuilder("C:/Program 
Files(x86)/Notepad++/notepad++.exe");
          // Effectively launch the process
          Process p = pb.start();
          // Check is process is alive or not
          boolean alive = p.isAlive();
          // Wait for the process to end for 10 seconds.
          if (p.waitFor(10, TimeUnit.SECONDS)) {
             System.out.println("Process has finished");
          } else {
             System.out.println("Timeout. Process hasn't finished");
          }
          // Force process termination.
          p.destroy();
          // Check again if process remains alive
          alive = p.isAlive();
          // Get the process exit value
          int status = p.exitValue();
        }
}
```

Termination codes

An exit code (or sometimes also return code) is the value that a process returns to its parent process to indicate how it has finished. If a process ends with an exit value of 0, everything went well, any other value between 1 to 255 indicates some cause of error.

Exception handling

The **waitFor** method call causes the parent process to block until the child process terminates or until the blocking is interrupted by some system signal (Exception) received by the parent process.

It is better to handle exceptions as close to the source as possible rather than passing them upstream.