Process IO management in Java

Standard I/O redirection

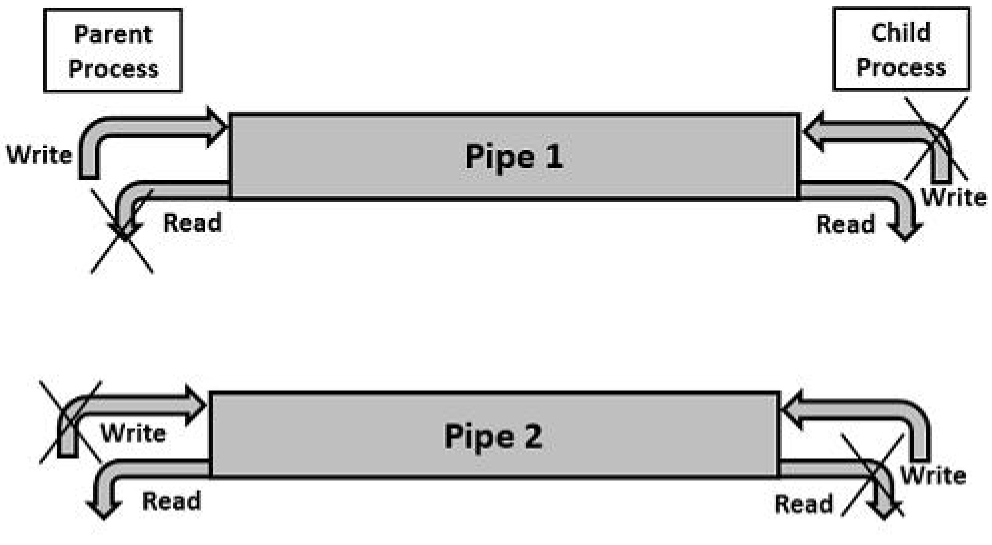
By default, a subprocess does not have its own terminal or console. All its standard I/O (i.e. stdin - keyboard -, stdout and stderr - screen -) operations will be redirected to the parent process, where they can be accessed via the streams obtained using the methods getOutputStream(), getInputStream() and getErrorStream(). The parent process uses these streams to feed input to and get output from the subprocess.

At no time, when we are programming a process, should we think about whether it is going to be launched as a parent or as a child.

In fact, all the programs we make in an IDE like IntelliJ IDEA are launched as children by the IDE and that doesn't make us change the way we program them.

A process that we are going to launch as a child should work perfectly well as an independent process and can be executed directly without having to make any changes to it.

In the parent-child relationship that is created between processes, descriptors are also redirected from the child to the parent, using 3 pipes, one for each I/O stream by default. These pipes can be used in a similar way as in Linux systems.

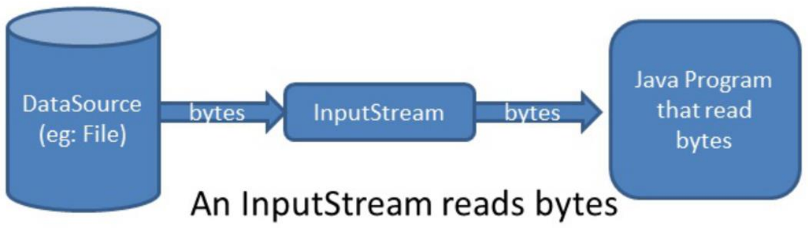


getInputStream()

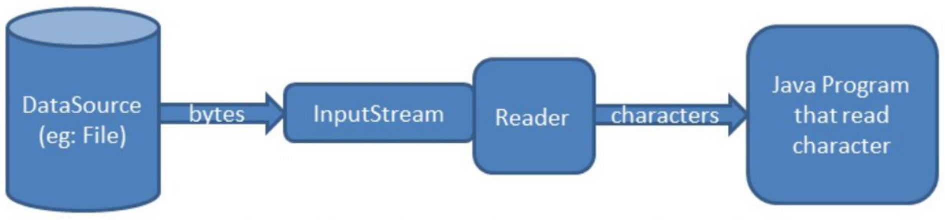
It is not only important to collect the return value of a command, but it is often very useful to be able to obtain the information that the process generates through the **standard output** or through the **error output**.

For this we are going to use the public abstract InputStream getInputStream() method of the Process class to read the output stream of the process, that is, to read what the **executed command** (child process) has sent to the **console**.

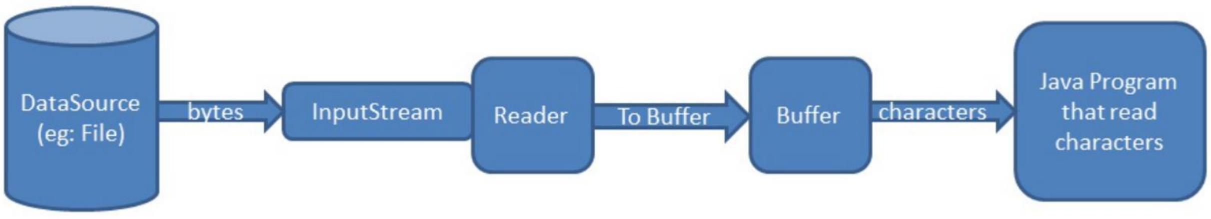
```
Process p = pbuilder.start();
BufferedReader processOutput = new BufferedReader(new
InputStreamReader(p.getInputStream()));
String line;

while ((line = processOutput.readLine()) != null) {
      System.out.println("> " + line);
}
processOutput.close();
```



An InputStream reads bytes



A Reader works with an InputStream and reads characters



A BufferedReader pre-fetches characters in to a buffer for performance

getErrorStream()

In addition to the standard output, we can also get the error output (stderr) generated by the child process to process it in the parent.

If the error output has been previously redirected using the ProcessBuilder.redirectErrorStream(true) method, then, the error output and the standard output arrive together with getInputStream() and no further processing is necessary.

If instead we want to make a differentiated treatment of the two types of output, we can use a schema similar to the one used above, except that now instead of calling **getInputStream()** we call **getErrorStream()** .

```java
Process p = pbuilder.start();
BufferedReader processError =
new BufferedReader(new InputStreamReader(p.getErrorStream()));
// In this example, we are supposing that we are sent an integer
int value = Integer.parseInt(processError.readLine());
processError.close();
```

getOutputStream()

Not only can we collect the information sent by the child process, but we can also send information from the parent process to the child process, using the last of the three remaining streams, **stdin**.

As with the inputs coming from the child process, we can send the information directly using the OutputStream of the process.

In this case, the top-level wrapper for using an **OutputStream** is the **PrintWriter** class, which offers methods like those of System.out.printxxxxx to manage the communication flow with the child process.

```java
PrintWriter toProcess = new PrintWriter(
new BufferedWriter(new OutputStreamWriter(p.getOutputStream(), "UTF-8")), true);
toProcess.println("sent to child");
```

Inherit I/O from the parent process

With the inheritIO() method, we can redirect all I/O streams from the child process to the standard I/O of the parent process.

```java
ProcessBuilder processBuilder = new ProcessBuilder("/bin/sh", "-c", "echo hello");
processBuilder.inheritIO();
Process process = processBuilder.start();
int exitCode = process.waitFor();
```

In the above example, after invoking the inheritIO() method, we can see the output of the command executed in the console of the parent process within the IDE.

Redirection of the Standard Inputs and Outputs

In a real system, we probably need to save the results of a process in a log or error file for later analysis. Fortunately, we can do this without modifying our application code by using the methods provided by the ProcessBuilder API to do exactly that.

By default, as we have already seen, child processes receive input through a pipe that we can access using the OutputStream returned by Process.getOutputStream().

However, as we will see below, that standard input can be changed and redirected to other destinations such as a file using the **redirectOutput(File)** method. If we modify the standard output, the getOutputStream() method will return ProcessBuilder.NullOutputStream.

It is important to note when each action is performed on a process.

Earlier we saw that I/O streams are queried and managed once the process is running, so the methods that give us access to those streams are methods of the Process class.

If what we want to do is to redirect the I/O, as we are going to see below, we will do it **while preparing** the **process** to be executed. So that when it is launched, its I/O streams are modified. That's why this time, the methods that allow us to redirect the I/O of the processes are methods of the **ProcessBuilder class**.

Let's see with an example how to make a program that displays the Java version. This time, the output is going to be saved in a log file instead of being sent to the parent via the standard output pipe.

```
ProcessBuilder processBuilder = new ProcessBuilder("java", "-version");
//Output error will be sent to the same place as standard one
processBuilder.redirectErrorStream(true);
File log = folder.newFile("java-version.log");
processBuilder.redirectOutput(log);
Process process = processBuilder.start();
```

In the example above, we can see how a temporary file called java-version.log is created and we tell ProcessBuilder to redirect the output to this file.

This is the same as calling our application using the output redirection operator:

java example-java-version > java-version.log

Now let's look at a variation of the previous example. What we want to do now is to append to the log file instead of overwriting the file each time the process is executed. By overwriting, we mean creating the file empty if it does not exist, or deleting the contents of the file if it already exists.

```
File log = tempFolder.newFile("java-version-append.log");
processBuilder.redirectErrorStream(true);
processBuilder.redirectOutput(Redirect.appendTo(log));
```

Again, it is important to note the call to redirectErrorStream(true) . In the event of any errors, they will be mixed with the output messages in the file.

In the ProcessBuilder API, we find methods to redirect also the standard error output and the standard input of the processes:

- **redirectError(File)**
- **redirectInput(File)**

To do the redirects, we can also use the ProcessBuilder.Redirect class as a parameter to the above methods, using one of the following values:

Value	Meaning
Redirect.DISCARD	The information is discarded
Redirect.to(File)	The information will be stored in the indicated file. If it exists, it is emptied.
Redirect.from(File)	The information will be read from the indicated file.
Redirect.appendTo(File)	The information will be added to the indicated file. If it exists, it is not emptied.