



# Process and Service Programming

## 2.3 Handling Process Streams

---



I.E.S.  
Doctor Balmis

PSP class notes ([https://psp2dam.github.io/psp\\_sources](https://psp2dam.github.io/psp_sources)) by Vicente Martínez is licensed under  
CC BY-NC-SA 4.0  (<http://creativecommons.org/licenses/by-nc-sa/4.0/?ref=chooser-v1>)

## 2.3 Handling Process Streams

- 2.3.1 Redirecting Standard Input and Output

- `getInputStream()`
- `getErrorStream()`
- `getOutputStream()`
- Inheriting the I/O of the parent process
- Pipelines

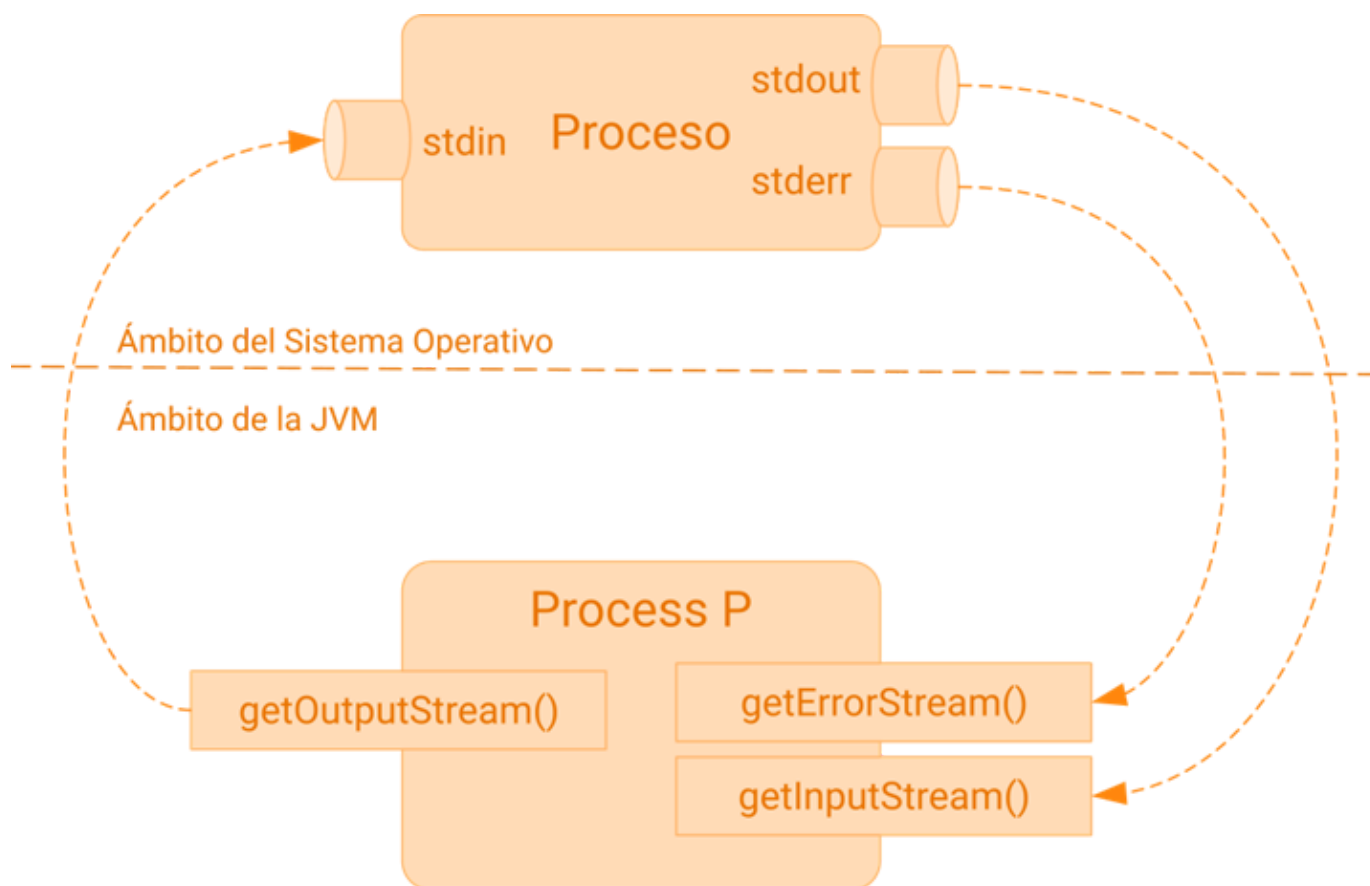
- 2.3.2 Redirecting Standard Input and Output

### 2.3.1 Redirecting Standard Input and Output

By default, the created subprocess does not have its terminal or console. All its standard I/O (i.e., `stdin`, `stdout`, `stderr`) operations will be sent to the parent process. Thereby the parent process can use these streams to feed input to and get output from the subprocess.

Consequently, this gives us a huge amount of flexibility as it gives us control over the input/output of our sub-process.

In a parent-child process relationship I/O streams are also redirected from child process to parent, using 3 pipes, one per each standard stream. Those pipes can be used like in a Linux system.



#### `getInputStream()`

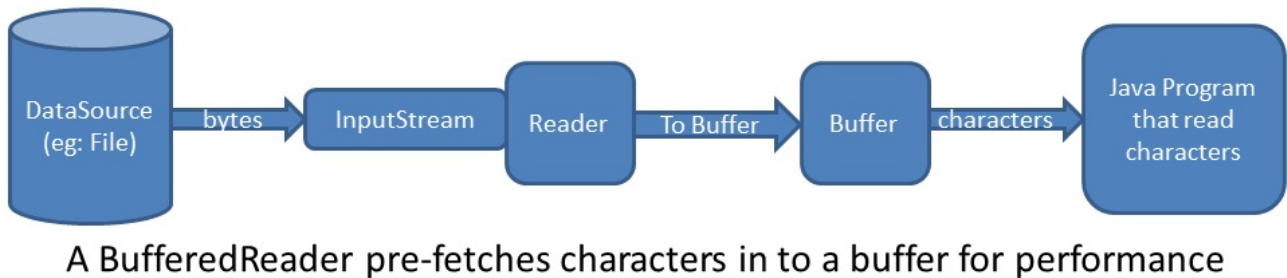
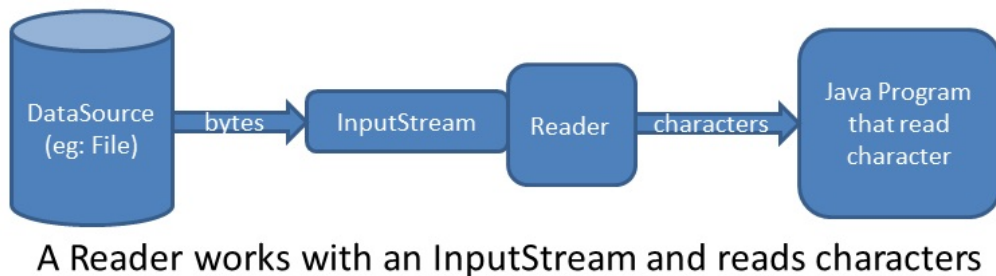
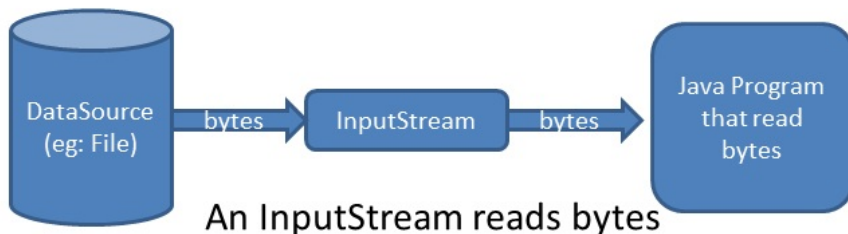
We can fetch the output generated by a subprocess and consume within the parent process thus allowing share information between the processes

java

```

1  Process p = pbuilder.start();
2  BufferedReader processOutput =
3      new BufferedReader(new InputStreamReader(p.getInputStream()));
4
5  String linea;
6  while ((linea = processOutput.readLine()) != null) {
7      System.out.println("> " + linea);
8  }
9  processOutput.close();

```



### Charset and encodings

From the time being computer science started we've been in trouble with encodings and charsets. And windows console is not an exception.

Terminal in Windows was also known as "DOS prompt": so a way to run DOS programs in Windows, so they keep the code page of DOS. Microsoft dislikes non-backward compatible changes, so your DOS program should works also on Windows terminal without problem.

Wikipedia indicates that **CP850** has theoretically been "largely replaced" by **Windows-1252** and, later, Unicode, but yet it's here, right in the OS's terminal.

Then, if we want to print information from the console in our applications we must deal with the right charset and encoding, that is, CP-850.

Fortunately, InputStreamReader has a constructor to manage streams with any encoding, so we must use it when working with console commands or applications.

```
new InputStreamReader(p.getInputStream(), "CP850");
```

java

We can force Netbeans to use a UTF-8 as default encoding. To do so we must modify its config file `C:/Program Files/Netbeans-xx.x/netbeans/etc/netbeans.conf`, changing directive `netbeans_default_option` and adding `-J-Dfile.encoding=UTF-8` to the end.

## getErrorStream()

Interestingly we can also fetch the errors generated from the subprocess and thereon perform some processing.

if error output has been redirected by calling method `ProcessBuilder.redirectErrorStream(true)` then, the error stream and the output stream will be shown using the same stream.

If we want to have it differentiated from the output, then we can use asimilar schema than before

```
1 Process p = pbuilder.start();
2 BufferedReader processError =
3     new BufferedReader(new InputStreamReader(p.getErrorStream()));
4 int value = Integer.parseInt(processError.readLine());
5 processError.close();
```

java

## Decorator or Wrapper design pattern

In both input and error streams we are getting information from a `BufferedReader`. Although we are not aware of using a design pattern, we are using the *"decorator design pattern"* or the so called **wrapper**.

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the required behaviors.

Refactoring.Guru design patterns (<https://refactoring.guru/design-patterns/java>)

Let's look at a complete example code using all the above operations

```
1 import java.io.*;
2 public class Ejercicio2 {
3     public static void main(String[] args) {
4         String comando = "notepad";
5         ProcessBuilder pbuilder = new ProcessBuilder (comando);
6         Process p = null;
7         try {
8             p = pbuilder.start();
9             // 1- Procedemos a Leer Lo que devuelve el proceso hijo
10            InputStream is = p.getInputStream();
11            // 2- Lo convertimos en un InputStreamReader
12            // De esta forma podemos Leer caracteres en vez de bytes
13            // El InputStreamReader nos permite gestionar diferentes codificaciones
14            InputStreamReader isr = new InputStreamReader(is);
15            // 2- Para mejorar el rendimiento hacemos un wrapper sobre un BufferedReader
16            // De esta forma podemos Leer enteros, cadenas o incluso líneas.
17            BufferedReader br = new BufferedReader(isr);
18
19            // A Continuación Leemos todo como una cadena, línea a línea
20            String linea;
```

java

```

21         while ((linea = br.readLine()) != null)
22             System.out.println(linea);
23     } catch (Exception e) {
24         System.out.println("Error en: "+comando);
25         e.printStackTrace();
26     } finally {
27         // Para finalizar, cerramos los recursos abiertos
28         br.close();
29         isr.close();
30         is.close();
31     }
32 }
33 }

```

## getOutputStream()

We can even send input to a subprocess from a parent process

There are three different ways of sending information to a child process. The first one is based on an `OutputStream`. Here no wrapper is used and the programmer has to manage all elements of the stream flow. From newline characters and type conversions to force sending information over the stream.

```

1 // Low-Level objects. We have to manage all elements of communication
2 OutputStream toProcess = p.getOutputStream();
3 toProcess.write((String.valueOf(number1)).getBytes("UTF-8"));
4 toProcess.write("\n".getBytes());
5 toProcess.flush();

```

java

The next one is based on a `Writer` object as a wrapper for the `OutputStream`, where communication management is easier, but the programmer still has to manage elements as new lines.

```

1 Writer w = new OutputStreamWriter(p.getOutputStream(), "UTF-8");
2 w.write("send to child\n");

```

java

Finally, the top-level wrapper for using the `OutputStream` is the `PrintWriter` object, where we can use the wrapper with the same methods as the `System.out` to handle child communication flow.

```

1 PrintWriter toProcess = new PrintWriter(
2     new BufferedWriter(
3         new OutputStreamWriter(
4             p.getOutputStream(), "UTF-8"), true);
5 toProcess.println("sent to child");

```

java

## Inheriting the I/O of the parent process

With the `inheritIO()` method We can redirect the sub-process I/O to the standard I/O of the current process (parent process)

```

1 ProcessBuilder processBuilder = new ProcessBuilder("/bin/sh", "-c", "echo hello");
2
3 processBuilder.inheritIO();
4 Process process = processBuilder.start();

```

java

```

5
6  int exitCode = process.waitFor();

```

In the above example, by using the `inheritIO()` method we see the output of a simple command in the console in our IDE.

## Pipelines

Java 9 introduced the concept of pipelines to the `ProcessBuilder` API:

```
public static List<Process> startPipeline(List<ProcessBuilder> builders)
```

java

Using the `startPipeline` method we can pass a list of `ProcessBuilder` objects. This static method will then start a `Process` for each `ProcessBuilder`. Thus, creating a pipeline of processes which are linked by their standard output and standard input streams.

For example, if we want to run something like this:

```
find . -name *.java -type f | wc -l
```

What we'd do is create a process builder for each isolated command and compose them into a pipeline

```

1  List builders = Arrays.asList(
2      new ProcessBuilder("find", "src", "-name", "*.java", "-type", "f"),
3      new ProcessBuilder("wc", "-l"));
4
5  List processes = ProcessBuilder.startPipeline(builders);
6  Process last = processes.get(processes.size() - 1);
7
8  // We can get lats process output to get the final results

```

java

In the example, we're searching for all the java files inside the `src` directory and piping the results into another process to count them.

### 2.3.2 Redirecting Standard Input and Output

In the real world, we will probably want to capture the results of our running processes inside a log file for further analysis. Luckily the `ProcessBuilder` API has built-in support for exactly this.

By default, our process reads input from a pipe. We can access this pipe via the output stream returned by `Process.getOutputStream()`.

However, as we'll see shortly, the standard output may be redirected to another source such as a file using the method `redirectOutput(File)`. In this case, `getOutputStream()` will return a `ProcessBuilder.NullOutputStream`.

Let's prepare an example to print out the version of Java. But this time let's redirect the output to a log file instead of the standard output pipe:

```

1  ProcessBuilder processBuilder = new ProcessBuilder("java", "-version");
2
3  processBuilder.redirectErrorStream(true);
4  File log = folder.newFile("java-version.log");
5  processBuilder.redirectOutput(log);
6

```

java

```
7 Process process = processBuilder.start();
```

In the above example, we create a new temporary file called log and tell our ProcessBuilder to redirect output to this file destination.

Now let's take a look at a slight variation on this example. For instance when we wish to `append to` a log file rather than create a new one each time:

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

java

It's also important to mention the call to `redirectErrorStream(true)`. In case of any errors, the error output will be merged into the normal process output file.

We can also redirect error stream an input stream for the subprocess with methods

- `redirectError(File)`
- `redirectInput(File)`

To make the redirections we can use the `ProcessBuilder.Redirect` class as a parameter for the previous methods, using one of the following values

Valor	Significado
<code>Redirect.DISCARD</code>	Information is discarded
<code>Redirect.to(File)</code>	Information is saved in the file. If it exists, it's emptied.
<code>Redirect.from(File)</code>	Information is read from the file
<code>Redirect.appendTo(File)</code>	Information is saved in the file. If it exists, it's not emptied.