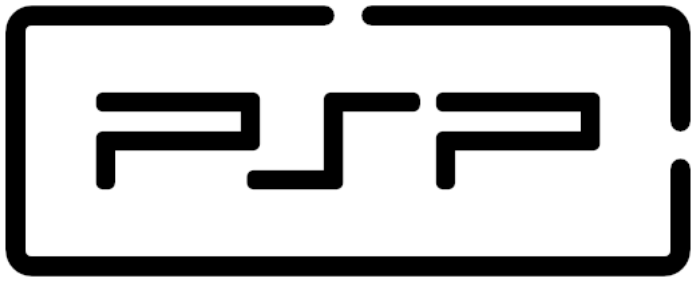


Process and Service Programming

2.3 Handling Process Streams

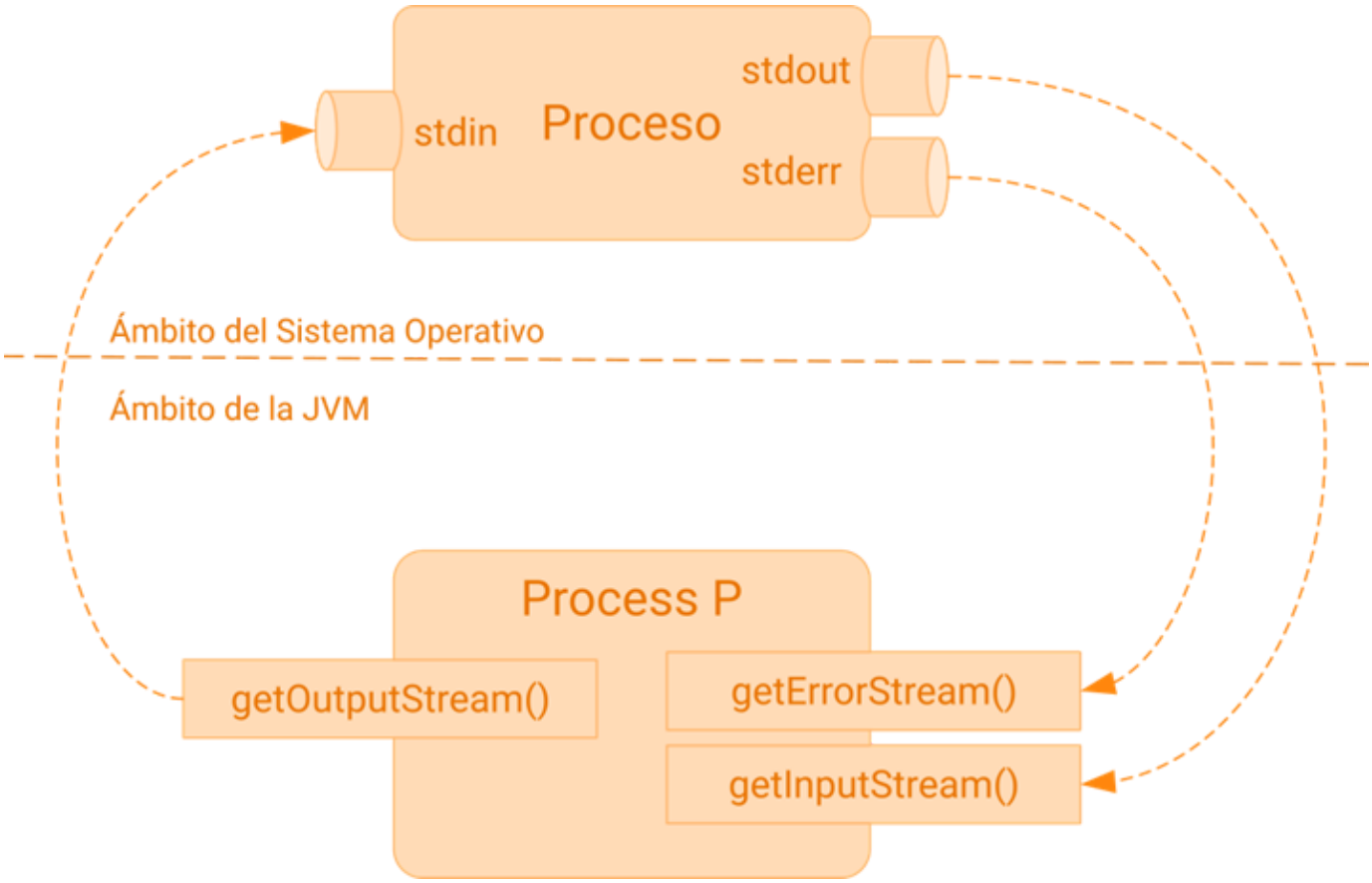
2.3 Handling Process Streams

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.

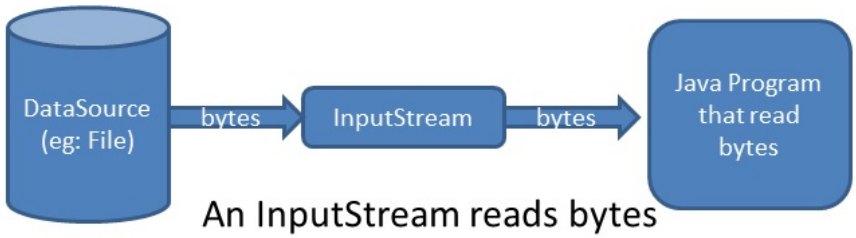
In a parent-child process relationship I/O streams are also redirected from child process to parent, using 3 pipes, one per each standard stream. Those pipes can be used like in a Linux system.



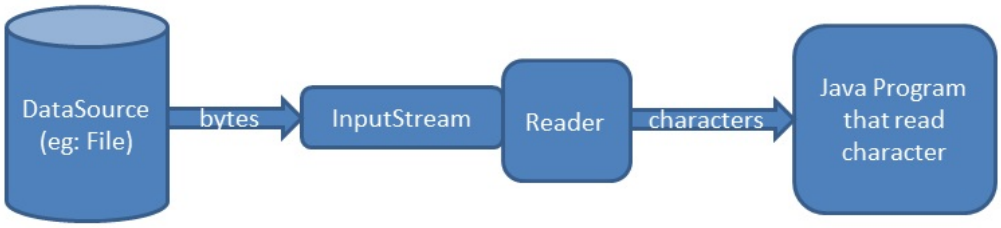
getInputStream()

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

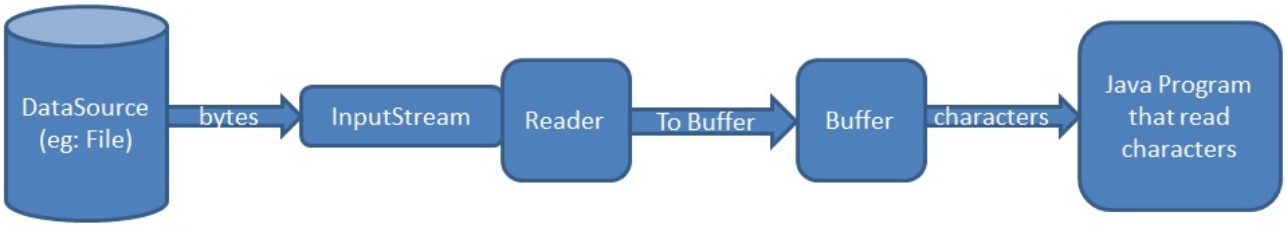
```java
Process p = pbuilder.start();
BufferedReader processOutput =
    new BufferedReader(new InputStreamReader(p.getInputStream()));

String linea;
while ((linea = processOutput.readLine()) != null) {
    System.out.println("> " + linea);
}
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

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.

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

> 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

```java
Process p = pbuilder.start();
BufferedReader processError =
    new BufferedReader(new InputStreamReader(p.getErrorStream()));
int value = Integer.parseInt(processError.readLine());
processError.close();
```

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

```java
import java.io.*;
public class Ejercicio2 {
    public static void main(String[] args) {
        String comando = "notepad";
        ProcessBuilder pbuilder = new ProcessBuilder (comando);
        Process p = null;
        try {
            p = pbuilder.start();
            // 1- Procedemos a leer lo que devuelve el proceso hijo
            InputStream is = p.getInputStream();
            // 2- Lo convertimos en un InputStreamReader
            // De esta forma podemos leer caracteres en vez de bytes
            // El InputStreamReader nos permite gestionar diferentes codificaciones
            InputStreamReader isr = new InputStreamReader(is);
            // 2- Para mejorar el rendimiento hacemos un wrapper sobre un BufferedReader
            // De esta forma podemos leer enteros, cadenas o incluso líneas.
            BufferedReader br = new BufferedReader(isr);

            // A Continuación leemos todo como una cadena, línea a línea
            String linea;
```

```
            while ((linea = br.readLine()) != null)
                System.out.println(linea);
        } catch (Exception e) {
            System.out.println("Error en: "+comando);
            e.printStackTrace();
        } finally {
            // Para finalizar, cerramos los recursos abiertos
            br.close();
            isr.close();
            is.close();
        }
    }
}
```

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.

java

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

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.

java

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

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.

java

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

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)

java

```
ProcessBuilder processBuilder = new ProcessBuilder("/bin/sh", "-c", "echo hello");

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

```
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.

> **Use it just for debugging purposes**
>
> This method is useful for debugging purposes, but it's not recommended for production code. It's better to use the getInputStream() and getErrorStream() methods to read the output and error streams of the subprocess, and the getOutputStream() method to write to the subprocess.

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.

> **Redirect before running the process**
>
> It's important to notice when we perform each action over a process.
>
> Before we've seen that I/O streams are consulted and managed once the process is running, so the methods that give us access to those streams are methods of the `Process` class.
>
> If we want to redirect I/O, as we are going to see next, we will do it while preparing the process to be executed. So when it's 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 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:

```java
ProcessBuilder processBuilder = new ProcessBuilder("java", "-version");

// Error output will be sent to the same place as the standard
processBuilder.redirectErrorStream(true);

File log = folder.newFile("java-version.log");
processBuilder.redirectOutput(log);

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.

Es lo mismo que si llamásemos a nuestra aplicación usando el operador de redirección de salida:

> java ejemplo-java-version > java-version.log

Código del proceso hijo

Si el proceso hijo que lanzamos, en vez de ser un comando del sistema, es otra clase java, en ningún momento tenemos que modificar el código de este proceso para que funcione como hijo.

Por lo tanto, el proceso hijo seguirá haciendo

> System.out.println("Versión de Java: " + System.getProperty("java.version"));

y será el sistema operativo el que se encargue de redirigir las salidas o entradas al fichero, o donde se haya configurado con los métodos de redirección de la clase `ProcessBuilder`.

It's the same as if we called our application using the output redirection operator:

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

Child process code

If the child process we launch, instead of being a system command, is another Java class, we don't have to modify the code of this process to work as a child.

Therefore, the child process will continue to do

> System.out.println("Java version: " + System.getProperty("java.version"));

and it will be the operating system that will take care of redirecting the outputs or inputs to the file, or where it has been configured with the redirection methods of the `ProcessBuilder` class.

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:

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

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 overloaded version of the previous methods, using one of the following values

Valor	Significado
Redirect.DISCARD	Information is discarded
Redirect.to(File)	Information is saved in the file. If it exists, it's emptied.

Valor	Significado
Redirect.from(File)	Information is read from the file
Redirect.appendTo(File)	Information is saved in the file. If it exists, it's not emptied.

This values are static fields of the Redirect class and can be used as parameters for the overloads of the redirectOutput, redirectError and redirectInput methods.

```java
File log = folder.newFile("sampleInputData.csv");
processBuilder.redirectInput(Redirect.from(log));
```