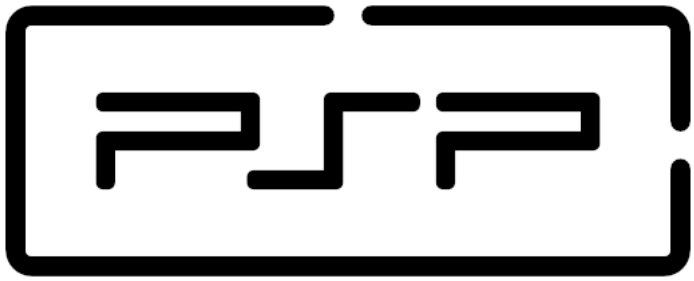


Process and Service Programming

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.

OS I/O streams and pipes

Streams in Linux, like almost everything else, are treated as though they were files.

Each file associated with a process is allocated a unique number to identify it. These values are always used for stdin, stdout, and stderr:

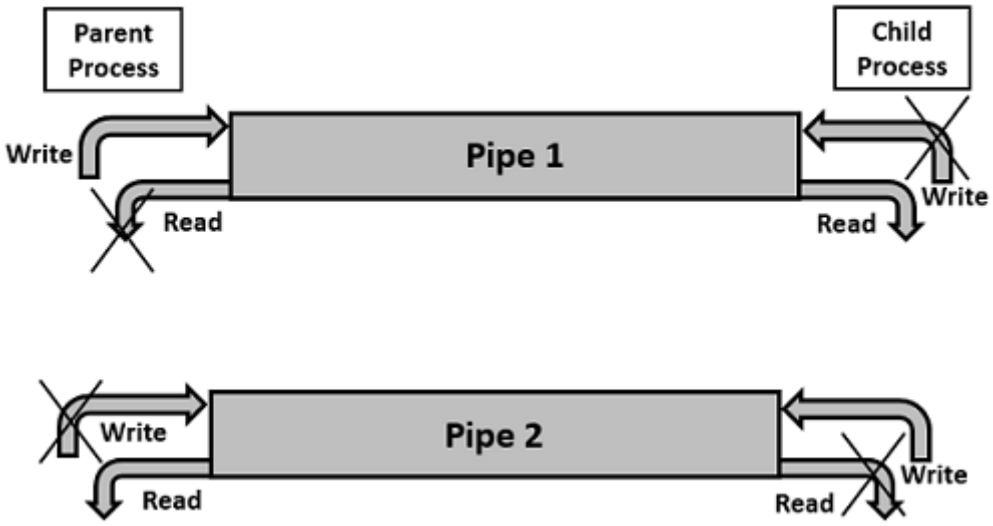
- 0: stdin
- 1: stdout
- 2: stderr

So we can manage these three streams in different ways. We can redirect a command's output (stdout) to a file and still see any error messages (stderr) in the terminal window, or we can get input to a command from another command or file. Let's look at some examples:

```sh
# Redirects ls output to a file
ls > capture.txt
# Redirects ls output to cat input
ls | cat
# Redirects program.sh output to capture.txt and its errors to error.txt
./program.sh 1> capture.txt 2> error.txt
# Redirects program.sh output and its errors to the same file, capture.txt
./program.sh > capture.txt 2>&1
# Redirects program.sh input from dummy.txt contents
./program.sh < dummy.txt
# Redirects output form first command to program.sh input
cat dummy.txt | ./program.sh
```

Introduction to Linux I/O Redirection

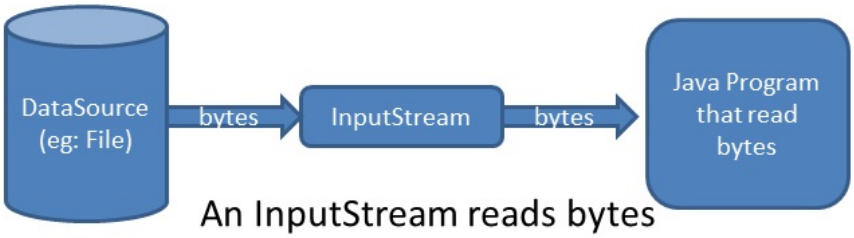
In a parent-child process relationship I/O streams are also redirected from child process to parent, using 3 pipes, one per each standard stream. Those pipes can be used like in a Linux system.



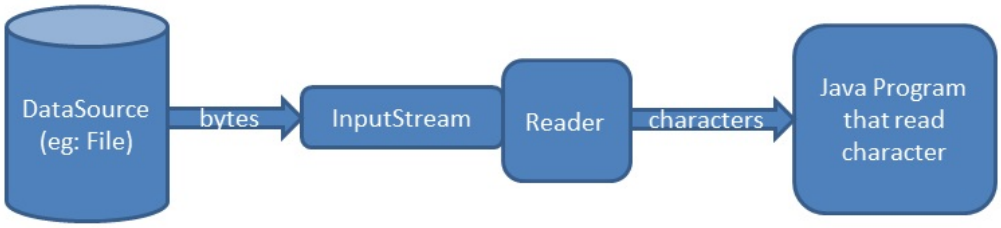
getInputStream()

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

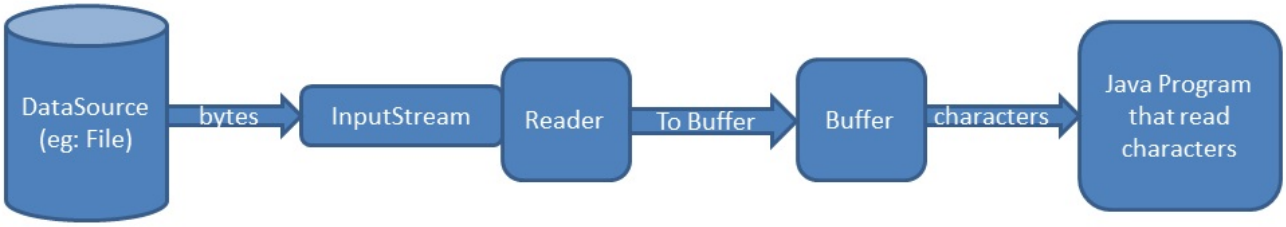
```java
Process p = pbuilder.start();
BufferedReader processOutput =
    new BufferedReader(new InputStreamReader(p.getInputStream()));

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/Netbeeans-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

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();
        }

```

```
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.

```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();

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:

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

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

```java
List builders = Arrays.asList(
    new ProcessBuilder("find", "src", "-name", "*.java", "-type", "f"),
    new ProcessBuilder("wc", "-l"));

List processes = ProcessBuilder.startPipeline(builders);
Process last = processes.get(processes.size() - 1);

// We can get lats process output to get the final results
```

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:

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

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.

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)

And for each of them we can also set the following redirections

- Redirect.to(File);
- Redirect.from(File);
- Redirect.appendTo(File);
- Redirect.DISCARD

2.3.3 Current Java Process Information

We can now obtain a lot of information about the process via the API `java.lang.ProcessHandle.Info` API:

- the command used to start the process
- the arguments of the command
- time instant when the process was started
- total time spent by it and the user who created it

Here's how we can do that

```java
ProcessHandle processHandle = ProcessHandle.current();
ProcessHandle.Info processInfo = processHandle.info();

System.out.println("PID: " + processHandle.pid());
System.out.println("Arguments: " + processInfo.arguments());
System.out.println("Command: " + processInfo.command());
System.out.println("Instant: " + processInfo.startInstant());
System.out.println("Total CPU duration: " + processInfo.totalCpuDuration());
System.out.println("User: " + processInfo.user());
```

It is also possible to get the process information of a newly spawned process. In this case, after we spawn the process and get an instance of the `java.lang.Process`, we invoke the `toHandle()` method on it to get an instance of java.lang.ProcessHandle.

```java
Process process = processBuilder.inheritIO().start();
ProcessHandle processHandle = process.toHandle();
```

The rest of the details remain the same as in the section above