

```

//tratamiento de la señal en proceso padre
signal( SIGUSR1, gestion_padre );
while(1) {
    pause(); //padre espera hasta recibir una señal del hijo
    sleep(1);
    kill(pid_hijo, SIGUSR1); //ENVIA SEÑAL AL HIJO
}
break;
}
return 0;
}

```

La Figura 1.11 muestra un momento de la ejecución de los procesos.

```

administrador@ubuntu1: ~
Archivo Editar Ver Buscar Terminal Ayuda
administrador@ubuntu1:~$ gcc sincronizar.c -o sincronizar
administrador@ubuntu1:~$ ./sincronizar
Padre recibe señal..10
Hijo recibe señal..10
^C
administrador@ubuntu1:~$ 

```

Figura 1.11. Ejecución de procesos sincronizados.

Para detener el proceso podemos pulsar las teclas [CTRL+C] o bien mediante el comando **ps** podemos ver el PID de los procesos padre e hijo que se están ejecutando:

```

administrador@ubuntu1:~$ ps -fe | grep sincronizar
1000  1678  1549  0 22:20 pts/0    00:00:00 ./sincronizar
1000  1679  1678  0 22:20 pts/0    00:00:00 ./sincronizar
1000  1687  1572  0 22:21 pts/1    00:00:00 grep --color=auto
                                         sincronizar
administrador@ubuntu1:~$ kill 1679
administrador@ubuntu1:~$ kill 1678
administrador@ubuntu1:~$ 

```

Primero eliminaremos el proceso hijo (PID 1679) y después el padre (PID 1678). Al eliminar el hijo el padre quedará esperando.

### ACTIVIDAD 1.3

Realiza un programa C en donde un hijo envíe 3 señales SIGUSR1 a su padre y después envíe una señal SIGKILL para que el proceso padre termine.

#### 1.2.3. CREACIÓN DE PROCESOS CON JAVA

Java dispone en el paquete **java.lang** de varias clases para la gestión de procesos. Una de ellas es la clase **ProcessBuilder**. Cada instancia **ProcessBuilder** gestiona una colección de atributos del proceso. El método **start()** crea una nueva instancia de **Process** con esos atributos y puede ser invocado varias veces desde la misma instancia para crear nuevos subprocesos con atributos idénticos o relacionados.

Por ejemplo, para ejecutar el comando DIR de DOS usando estas dos clases escribimos lo siguiente, indicando en el constructor de **ProcessBuilder** los argumentos del proceso que se quiere ejecutar como una lista de cadenas separadas por comas, y después usamos el método **start()** para iniciar el proceso:

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

La clase **Process** proporciona métodos para realizar la entrada desde el proceso, obtener la salida del proceso, esperar a que el proceso se complete, comprobar el estado de salida del proceso y destruir el proceso. En la siguiente tabla se muestran los más importantes:

| MÉTODOS                                   | MISIÓN   |
|---|--|
| <b>InputStream<br/>getInputStream ()</b>  | Devuelve el flujo de entrada conectado a la salida normal del subprocesso. Nos permite leer el stream de salida del subprocesso, es decir, podemos leer lo que el comando que ejecutamos escribió en la consola. |
| <b>int waitFor ()</b>                     | Provoca que el proceso actual espere hasta que el subprocesso representado por el objeto <b>Process</b> finalice. Devuelve 0 si ha finalizado correctamente.   |
| <b>InputStream<br/>getErrorStream()</b>   | Devuelve el flujo de entrada conectado a la salida de error del subprocesso. Nos va a permitir poder leer los posibles errores que se produzcan al lanzar el subprocesso.  |
| <b>OutputStream<br/>getOutputStream()</b> | Devuelve el flujo de salida conectado a la entrada normal del subprocesso. Nos va a permitir escribir en el stream de entrada del subprocesso, así podemos enviar datos al subprocesso que se ejecute.           |
| <b>void destroy ()</b>                    | Elimina el subprocesso.  |
| <b>int exitValue ()</b>                   | Devuelve el valor de salida del subprocesso.   |
| <b>boolean isAlive ()</b>                 | Comprueba si el subprocesso representado por <b>Process</b> está vivo  |

Por defecto el proceso que se crea (o subprocesso) no tiene su propia terminal o consola. Todas las operaciones de E/S serán redirigidas al proceso padre, donde se puede acceder a ellas usando los métodos **getOutputStream()**, **getInputStream()** y **getErrorStream()**. El proceso padre utiliza estos flujos para alimentar la entrada y obtener la salida del subprocesso. En algunas plataformas se pueden producir bloqueos en el subprocesso debido al tamaño de búfer limitado para los flujos de entrada y salida estándar.

Cada constructor de **ProcessBuilder** gestiona los siguientes atributos de un proceso:

- Un comando. Es una lista de cadenas que representa el programa que se invoca y sus argumentos si los hay.
- Un entorno (*environment*) con sus variables.
- Un directorio de trabajo. El valor por defecto es el directorio de trabajo del proceso en curso.
- Una fuente de entrada estándar. Por defecto, el subprocesso lee la entrada de una tubería. El código Java puede acceder a esta tubería a través de la secuencia de salida devuelta por **Process.getOutputStream()**. Sin embargo, la entrada estándar puede ser redirigida a otra fuente con **redirectInput()**. En este caso, **Process.getOutputStream()** devolverá una secuencia de salida nula.

- Un destino para la salida estándar y la salida de error. Por defecto, el subprocesso escribe en las tuberías de la salida y el error estándar. El código Java puede acceder a estas tuberías a través de los flujos de entrada devueltos por `Process.getInputStream()` y `Process.getErrorStream()`. Igual que antes, la salida estándar y el error estándar pueden ser redirigido a otros destinos utilizando `redirectOutput()` y `redirectError()`. En este caso, `Process.getInputStream()` y/o `Process.getErrorStream()` devuelven una secuencia de entrada nula.
- Una propiedad `redirectErrorStream`. Inicialmente, esta propiedad es false, significa que la salida estándar y salida de error de un subprocesso se envían a dos corrientes separadas, que se pueden acceder a través de los métodos `Process.getInputStream()` y `Process.getErrorStream()`.

Algunos de los métodos proporcionados por la clase `ProcessBuilder` son los siguientes:

| MÉTODOS   | MISIÓN  |
|---|---|
| <code>ProcessBuilder command ( String argumentos ... )</code> | Define el programa que se quiere ejecutar indicando sus argumentos como una lista de cadenas separadas por comas. |
| <code>List &lt; String &gt; command ()</code>                 | Devuelve todos los argumentos del objeto <code>ProcessBuilder</code> .  |
| <code>Map &lt; String , String &gt; environment ()</code>     | Devuelve en una estructura Map las variables de entorno del objeto <code>ProcessBuilder</code> .                  |
| <code>ProcessBuilder redirectError (File file)</code>         | Redirige la salida de error estándar a un fichero.  |
| <code>ProcessBuilder redirectInput (File file)</code>         | Establece la fuente de entrada estándar en un fichero.  |
| <code>ProcessBuilder redirectOutput ( File file)</code>       | Redirige la salida estándar a un fichero.   |
| <code>File directory()</code>                                 | Devuelve el directorio de trabajo del objeto <code>ProcessBuilder</code> .  |
| <code>ProcessBuilder directory(File directorio)</code>        | Establece el directorio de trabajo del objeto <code>ProcessBuilder</code> .                                       |
| <code>Process start ()</code>                                 | Inicia un nuevo proceso utilizando los atributos del objeto <code>ProcessBuilder</code> .                         |

Para iniciar un nuevo proceso que utiliza el directorio de trabajo y el entorno del proceso en curso escribimos la siguiente orden:

```
Process p = new ProcessBuilder("Comando", "Argum1").start();
```

Por ejemplo, para ejecutar el comando DIR de DOS podemos escribir lo siguiente:

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

El siguiente ejemplo Java muestra como se puede ejecutar una aplicación de Windows, en este caso el NOTEPAD, que es el bloc de notas de Windows:

```
public class Ejemplo1 {
    public static void main(String[] args) throws IOException {
        Process pb = new ProcessBuilder("NOTEPAD").start();
    }
} //Ejemplo1
```

El ejemplo es equivalente al siguiente:

```
public class Ejemplo1 {
    public static void main(String[] args) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("NOTEPAD");
        Process p = pb.start();
    }
} //Ejemplo1
```

Para los comandos de Windows que no tienen ejecutable (como por ejemplo DIR o ATTRIB) es necesario utilizar el comando CMD.EXE. Entonces para hacer un DIR desde un programa Java tendríamos que construir un objeto **ProcessBuilder** con los siguientes argumentos: "CMD", "/C" y "DIR".

#### Sabías que...

CMD Inicia una nueva instancia del intérprete de comandos de Windows. Para ver la sintaxis del comando escribimos desde el indicador del DOS: HELP CMD.

Para ejecutar un comando escribimos:

CMD /C comando: Ejecuta el comando especificado y luego finaliza.

CMD /K comando: Ejecuta el comando especificado, pero sigue activo.

El siguiente ejemplo ejecuta el comando DIR. Usaremos el método **getInputStream()** de la clase **Process** para leer el stream de salida del proceso, es decir, para leer lo que el comando DIR envía a la consola. Definiremos así el stream:

```
InputStream is = p.getInputStream();
```

Para leer la salida usamos el método **read()** de **InputStream** que nos devolverá carácter a carácter la salida generada por el comando. El programa Java es el siguiente:

```
import java.io.*;
public class Ejemplo2 {
    public static void main(String[] args) throws IOException {
        //Ejecutamos el proceso DIR
        Process p = new ProcessBuilder("CMD", "/C", "DIR").start();
        //Mostramos carácter a carácter la salida generada por DIR
        try {
            InputStream is = p.getInputStream();
            int c;
            while ((c = is.read()) != -1)
                System.out.print((char) c);
            is.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        //COMPROBACIÓN DE ERROR - 0 bien - 1 mal
    }
}
```

```

        int exitVal;
        try {
            exitVal = p.waitFor(); //recoge la salida de System.exit()
            System.out.println("Valor de Salida: " + exitVal);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
} // Ejemplo2

```

Al ejecutarlo, desde el entorno Eclipse, se muestra una salida similar a la siguiente:

```

El volumen de la unidad D es Data
El número de serie del volumen es: 9013-7A66

```

```

Directorio de D:\CLASE\PSP_2018\CAPITULO1
14/06/2018 00:14 <DIR> .
14/06/2018 00:14 <DIR> ..
14/06/2018 00:14 396 .classpath
14/06/2018 00:14 385 .project
14/06/2018 00:14 <DIR> .settings
14/06/2018 00:15 <DIR> bin
14/06/2018 00:15 <DIR> src
2 archivos 781 bytes
5 dirs 134.146.781.184 bytes libres
Valor de Salida: 0

```

El método `waitFor()` hace que el proceso actual espere hasta que el subproceso representado por el objeto `Process` finalice. Este método recoge lo que `System.exit()` devuelve, por defecto en un programa Java si no se incluye esta orden el valor devuelto es 0, que normalmente responde a una finalización correcta del proceso.

El siguiente ejemplo muestra un programa Java que ejecuta el programa Java anterior, en este caso el programa se ejecutará desde el entorno Eclipse. Como el proceso a ejecutar se encuentra en la carpeta `bin` del proyecto será necesario crear un objeto `File` que refiera a dicho directorio. Después para establecer el directorio de trabajo para el proceso que se va a ejecutar se debe usar el método `directory()`, a continuación se ejecutará el proceso y por último será necesario recoger el resultado de salida usando el método `getInputStream()` del proceso:

```

import java.io.*;
public class Ejemplo3 {
    public static void main(String[] args) throws IOException {
        //Creamos objeto File al directorio donde esta Ejemplo2
        File directorio = new File(".\\bin");

        //El proceso a ejecutar es Ejemplo2
        ProcessBuilder pb = new ProcessBuilder("java", "Ejemplo2");

        //se establece el directorio donde se encuentra el ejecutable
        pb.directory(directorio);

        System.out.printf("Directorio de trabajo: %s%n", pb.directory());
        //se ejecuta el proceso
        Process p = pb.start();
    }
}

```

```

//obtener la salida devuelta por el proceso
try {
    InputStream is = p.getInputStream();
    int c;
    while ((c = is.read()) != -1)
        System.out.print((char) c);
    is.close();
} catch (Exception e) {
    e.printStackTrace();
}
}
// Ejemplo3

```

La salida mostrará los ficheros y carpetas del directorio definido en la variable *directorio*. Si ambos ficheros están en la misma carpeta o directorio, no será necesario establecer el directorio de trabajo para el objeto **ProcessBuilder**. Si el *Ejemplo2* a ejecutar se encontrase en la carpeta D:\PSP, tendríamos que definir el objeto *directorio* de la siguiente manera: *File directorio = new File("D:\\PSP")*.

#### ACTIVIDAD 1.4

Crea un programa Java llamado *LeerNombre.java* que reciba desde los argumentos de *main()* un nombre y lo visualice en pantalla. Utiliza *System.exit(1)* para una finalización correcta del programa y *System.exit(-1)* para el caso que no se hayan introducido los argumentos correctos en *main()*.

A continuación, haz un programa parecido a *Ejemplo3.java* para ejecutar *LeerNombre.java*. Utiliza el método **waitFor()** para comprobar el valor de salida del proceso que se ejecuta. Prueba la ejecución del programa dando valor a los argumentos de *main()* y sin darle valor. ¿Qué valor devuelve **waitFor()** en un caso y en otro?

Realiza el Ejercicio 4.

La clase **Process** posee el método **getErrorStream()** que nos va a permitir obtener un stream para poder leer los posibles errores que se produzcan al lanzar el proceso. En el *Ejemplo2.java* si cambiamos los argumentos y escribimos algo incorrecto, por ejemplo lo siguiente:

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

Al ejecutarlo aparecerá como valor de salida 1 indicando que el proceso no ha finalizado correctamente. Pero si añadimos el siguiente código al ejemplo:

```

try {
    InputStream er = p.getErrorStream();
    BufferedReader brer =
        new BufferedReader(new InputStreamReader(er));
    String liner = null;
    while ((liner = brer.readLine()) != null)
        System.out.println("ERROR >" + liner);
} catch (IOException ioe) {
    ioe.printStackTrace();
}

```

Se obtendrá la siguiente salida indicando el error que se ha producido:

ERROR >"DIRR" no se reconoce como un comando interno o externo,  
 ERROR >programa o archivo por lotes ejecutable.  
 Valor de Salida: 1

### ACTIVIDAD 1.5

Partiendo del *Ejemplo3.java*, muestra los errores que se producen al ejecutar un programa Java que no existe.

Realiza los ejercicios 5 y 6

### ENVIAR DATOS AL STREAM DE ENTRADA DEL PROCESO

Supongamos ahora que queremos ejecutar un proceso que necesita información de entrada. Por ejemplo, si ejecutamos DATE desde la línea de comandos y pulsamos la tecla [Intro] nos pide escribir una nueva fecha:

```
D:\CAPIT1>DATE
La fecha actual es: 14/06/2018
Escriba la nueva fecha: (dd-mm-aa) 15-06-18
```

La clase **Process** posee el método **getOutputStream()** que nos permite escribir en el stream de entrada del proceso, así podemos enviarle datos. El siguiente ejemplo ejecuta el comando DATE y le da los valores 15-06-18. Con el método **write()** se envían los bytes al stream, el método **getBytes()** codifica la cadena en una secuencia de bytes que utilizan juego de caracteres por defecto de la plataforma:

```
import java.io.*;
public class Ejemplo4 {
    public static void main(String[] args) throws IOException {
        Process p = new ProcessBuilder("CMD", "/C", "DATE").start();

        // escritura -- envia entrada a DATE
        OutputStream os = p.getOutputStream();
        os.write("15-06-18".getBytes());
        os.flush(); // vacía el buffer de salida

        // lectura -- obtiene la salida de DATE
        InputStream is = p.getInputStream();
        int c;
        while ((c = is.read()) != -1)
            System.out.print((char) c);
        is.close();

        // COMPROBACION DE ERROR - 0 bien - 1 mal
        int exitVal;
        try {
            exitVal = p.waitFor();
            System.out.println("Valor de Salida: " + exitVal);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
} //Ejemplo4
```

La ejecución muestra la siguiente salida:

```
La fecha actual es: 14/06/2018
Escriba la nueva fecha: (dd-mm-aa) 15-06-18
Valor de Salida: 0
```

Supongamos que tenemos un programa Java que lee una cadena desde la entrada estándar y la visualiza:

```
import java.io.*;
public class EjemploLectura{
    public static void main (String [] args)
    {
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader (in);
        String texto;
        try {
            System.out.println("Introduce una cadena....");
            texto= br.readLine();
            System.out.println("Cadena escrita: "+texto);
            in.close();
        }catch (Exception e) { e.printStackTrace(); }
    }
}//EjemploLectura
```

Con el método `getOutputStream()` podemos enviar datos a la entrada estándar del programa *EjemploLectura.java*. Por ejemplo si queremos enviar la cadena "Hola Manuel" cambiaríamos varias cosas en el *Ejemplo4.java*:

```
File directorio = new File(".\\bin");
ProcessBuilder pb = new ProcessBuilder("java", "EjemploLectura");
pb.directory(directorio);

// se ejecuta el proceso
Process p = pb.start();

// escritura - se envia la entrada
OutputStream os = p.getOutputStream();
os.write("Hola Manuel\n".getBytes());
os.flush(); // vacía el buffer de salida
```

Cada línea que mandemos a *EjemploLectura* debe terminar con "\n", igual que cuando escribimos desde el terminal la lectura termina cuando pulsamos la tecla [Intro]. Suponiendo que hemos guardado estos cambios en *Ejemplo5.java*, la ejecución muestra la siguiente salida:

```
Introduce una cadena....
Cadena escrita: Hola Manuel
Valor de Salida: 0
```

#### **ACTIVIDAD 1.6**

Escribe un programa Java que lea dos números desde la entrada estándar y visualice su suma. Controlar que lo introducido por teclado sean dos números. Haz otro programa Java para ejecutar el anterior. Realiza el ejercicio 7.

El siguiente ejemplo usa varios métodos de la clase **ProcessBuilder**: **environment()** que devuelve las variables de entorno del proceso; el método **command()** sin parámetros, que devuelve los argumentos del proceso definido en el objeto **ProcessBuilder**; y con parámetros donde se define un nuevo proceso y sus argumentos. Después se ejecutará este último proceso:

```
import java.io.*;
import java.util.*;

public class Ejemplo6 {
    public static void main(String args[]) {
        ProcessBuilder test = new ProcessBuilder();
        Map entorno = test.environment();
        System.out.println("Variables de entorno:");
        System.out.println(entorno);

        test = new ProcessBuilder("java", "LeerNombre", "Maria Jesús");

        // devuelve el nombre del proceso y sus argumentos
        List l = test.command();
        Iterator iter = l.iterator();
        System.out.println("\nArgumentos del comando:");
        while (iter.hasNext())
            System.out.println(iter.next());

        test = test.command("CMD", "/C", "DIR");
        try {
            Process p = test.start(); //se ejecuta DIR
            InputStream is = p.getInputStream();

            System.out.println();
            // mostramos en pantalla caracter a caracter
            int c;
            while ((c = is.read()) != -1)
                System.out.print((char) c);
            is.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
// Ejemplo6
```

La compilación y ejecución muestra la siguiente salida:

#### Variables de entorno:

```
variables de entorno:  
{configsetroot=C:\WINDOWS\ConfigSetRoot, USERDOMAIN_ROAMINGPROFILE=PC-  
ASUS, PROCESSOR_LEVEL=6, FP_NO_HOST_CHECK=NO, SESSIONNAME=Console,  
ALLUSERSPROFILE=C:\ProgramData, PROCESSOR_ARCHITECTURE=AMD64,  
PSModulePath=C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\;C:\Pro  
gram Files\VisualSVN Server\PowerShellModules, SystemDrive=C:,  
JRE_HOME=C:\Program Files\Java\jre-10.0.1, USERNAME=mjesus,  
NUMBER_OF_PROCESSORS=4}
```

Argumentos del comando:  
java  
LeerNombre

Maria Jesús

El volumen de la unidad D es Data  
El número de serie del volumen es: 9013-7A66

Directorio de D:\CLASE\PSP\_2018\CAPITULO1

|                        |                                     |
|------------------------|-------------------------------------|
| 14/06/2018 00:14 <DIR> | .                                   |
| 14/06/2018 00:14 <DIR> | ..                                  |
| 14/06/2018 00:14       | 396 .classpath                      |
| 14/06/2018 00:14       | 385 .project                        |
| 14/06/2018 00:14       | .settings                           |
| 14/06/2018 00:14 <DIR> | bin                                 |
| 14/06/2018 00:15 <DIR> | src                                 |
| 14/06/2018 00:15 <DIR> | 2 archivos 781 bytes                |
|                        | 5 dirs 134.146.215.936 bytes libres |

### REDIRECCIONANDO LA ENTRADA Y LA SALIDA

Los métodos `redirectOutput()` y `redirectError()` nos permiten redirigir la salida estándar y de error a un fichero. El siguiente ejemplo ejecuta el comando DIR y envía la salida al fichero `salida.txt`, si ocurre algún error se envía a `error.txt`:

```
import java.io.File;
import java.io.IOException;

public class Ejemplo7 {
    public static void main(String args[]) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");

        File fOut = new File("salida.txt");
        File fErr = new File("error.txt");

        pb.redirectOutput(fOut);
        pb.redirectError(fErr);
        pb.start();
    }
} // Ejemplo7
```

También podemos ejecutar varios comandos del sistema operativo dentro de un fichero BAT. El siguiente ejemplo ejecuta los comandos MS-DOS que se encuentran en el fichero `fichero.bat`. Se utiliza el método `redirectInput()` para indicar que la entrada al proceso se encuentra en un fichero, es decir la entrada para el comando CMD será el `fichero.bat`. La salida del proceso se envía al fichero `salida.txt` y la salida de error al fichero `error.txt`:

```
import java.io.File;
import java.io.IOException;

public class Ejemplo8 {
    public static void main(String args[]) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD");

        File fBat = new File("fichero.bat");
        File fOut = new File("salida.txt");
        File fErr = new File("error.txt");
```

```

    pb.redirectInput(fBat);
    pb.redirectOutput(fOut);
    pb.redirectError(fErr);
    pb.start();

}
} // Ejemplo8

```

Suponiendo que los comandos MS-DOS del *fichero.bat* son estos (este fichero se debe crear en el proyecto Eclipse):

```

MKDIR NUEVO
CD NUEVO
ECHO CREO FICHERO > Mifichero.txt
DIR
DIRR
ECHO FIN COMANDOS

```

Donde se crea una carpeta, nos dirigimos a dicha carpeta, se crea el fichero *Mifichero.txt*, se hace un DIR del directorio actual, el siguiente comando DIRR es erróneo y se visualiza FIN COMANDOS. Al ejecutarlo desde el entorno Eclipse el contenido del fichero de salida *salida.txt* es el siguiente:

```

Microsoft Windows [Versión 10.0.17134.48]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

D:\CLASE\PSP_2018\CAPITULO1>MKDIR NUEVO

D:\CLASE\PSP_2018\CAPITULO1>CD NUEVO

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>ECHO CREO FICHERO > Mifichero.txt

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>DIR
El volumen de la unidad D es Data
El número de serie del volumen es: 9013-7A66

Directorio de D:\CLASE\PSP_2018\CAPITULO1\NUEVO

14/06/2018 00:34 <DIR> .
14/06/2018 00:34 <DIR> ..
14/06/2018 00:34 15 Mifichero.txt
                 1 archivos          15 bytes
                 2 dirs  134.146.215.936 bytes libres

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>DIRR

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>ECHO FIN COMANDOS
FIN COMANDOS

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>
Y el del fichero de error error.txt:
"DIRR" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

```

**ACTIVIDAD 1.7**

Modifica el *Ejemplo5.java* para que la salida del proceso y la salida de error se almacenen en un fichero de texto, y la entrada la tome desde otro fichero de texto.

Para llevar a cabo el redireccionamiento, tanto de entrada como de salida del proceso que se ejecuta, también podemos usar la clase **ProcessBuilder.Redirect**. El redireccionamiento puede ser uno de los siguientes:

- El valor especial **Redirect.INHERIT**, indica que la fuente de entrada y salida del proceso será la misma que la del proceso actual.
- **Redirect.from (File)**, indica redirección para leer de un fichero, la entrada al proceso se encuentra en el objeto **File**.
- **Redirect.to(File)**, indica redirección para escribir en un fichero, el proceso escribirá en el objeto **File** especificado.
- **Redirect.appendTo (File)**, indica redirección para añadir a un fichero, la salida del proceso se añadirá al objeto **File** especificado.

El ejemplo anterior usando esta clase quedaría de esta manera:

```
pb.redirectInput(ProcessBuilder.Redirect.from(fBat));
pb.redirectOutput(ProcessBuilder.Redirect.to(fOut));
pb.redirectError(ProcessBuilder.Redirect.to(fErr));
```

El siguiente ejemplo muestra en la consola la salida del comando DIR,

```
import java.io.IOException;
public class Ejemplo9 {
    public static void main(String args[]) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
        pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
        Process p = pb.start();
    }
} // Ejemplo9
```

**ACTIVIDAD 1.8**

Usando **ProcessBuilder.Redirect**, modifica el *Ejemplo5.java* para que la salida del proceso se muestre en la consola, la entrada la tome desde un fichero de texto, y la salida la lleve a un fichero de texto. Realiza los ejercicios 7, 8 y 9.

## 1.3. PROGRAMACIÓN CONCURRENTE

El diccionario *WordReference.com* (<http://www.wordreference.com/definicion/>) nos muestra varias acepciones de la palabra concurrencia. Nos quedamos con la tercera: “*Acaecimiento o concurso de varios sucesos en un mismo tiempo*”. Si sustituimos sucesos por procesos ya tenemos una aproximación de lo que es la concurrencia en informática: la existencia simultánea de varios procesos en ejecución.

### 1.3.1. PROGRAMA Y PROCESO

Al principio del tema se definió un **proceso** como un programa en ejecución. Y ¿qué es un **programa**? podemos definir **programa** como un conjunto de instrucciones que se aplican a un conjunto de datos de entrada para obtener una salida. Un proceso es algo activo que cuenta con una serie de recursos asociados, en cambio un programa es algo pasivo, para que pueda hacer algo hay que ejecutarlo.