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

Por defecto el proceso que se crea (o subproceso) 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 subproceso. En algunas plataformas se pueden producir bloqueos en el subproceso 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 subproceso 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 nulo.

- Un destino para la salida estándar y la salida de error. Por defecto, el subproceso 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 subproceso se envían a dos corrientes separadas, que se pueden acceder a través de los métodos **Process.getInputStream()** y **Process.getErrorStream()**.

Algunos de los métodos porporcionados por la clase **ProcessBuilder** son los siguientes:

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

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 **Proccess** 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 cada 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 referencie 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 exista.

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.