



# Unión Europea

Fondo Social Europeo

*El FSE invierte en tu futuro*

## UD 1 CREACIÓN DE PROCESOS (PSP)

CFGS Desarrollo de Aplicaciones Multiplataforma

Pepe

INSTITUT



GENERALITAT  
VALENCIANA

Conselleria d'Educació,  
Investigació, Cultura i Esport



GOBIERNO  
DE ESPAÑA

MINISTERIO  
DE EDUCACIÓN, CULTURA  
Y DEPORTE

## Continguts

<b>1</b>	<b>Introducción: aplicaciones, ejecutables y procesos</b>	<b>3</b>
<b>2</b>	<b>CONTROL DE PROCESOS EN LINUX</b>	<b>3</b>
2.1	Introducción . . . . .	3
<b>3</b>	<b>Estados de un Proceso</b>	<b>5</b>
<b>4</b>	<b>Creación de Procesos en Java</b>	<b>6</b>
4.1	Clases ProcessBuilder y Process . . . . .	6
4.2	Redirigiendo la salida con getInputStream . . . . .	8
4.3	Escribiendo en la entrada con getOutputStream . . . . .	10
<b>5</b>	<b>Ejercicio</b>	<b>13</b>
<b>6</b>	<b>Ejercicio</b>	<b>13</b>
<b>7</b>	<b>Ejercicio</b>	<b>14</b>
<b>8</b>	<b>Ejercicio</b>	<b>14</b>

## 1 Introducción: aplicaciones, ejecutables y procesos

A simple vista, parece que con los términos *aplicación*, *ejecutable* y *proceso*, nos estamos refiriendo a lo mismo. Pero hay sustanciales diferencias entre ellos y debemos tenerlas claras.

Una aplicación es un tipo de programa informático, diseñado como herramienta para resolver de manera automática un problema específico del usuario.

Debemos darnos cuenta de que sobre el hardware del equipo, todo lo que se ejecuta son programas informáticos, que, ya sabemos, que se llama software.

Con la definición de aplicación anterior, buscamos diferenciar las aplicaciones, de otro tipo de programas informáticos, como pueden ser: los sistemas operativos, las utilidades para el mantenimiento del sistema, o las herramientas para el desarrollo de software. Por lo tanto, son aplicaciones, aquellos programas que nos permiten editar una imagen, enviar un correo electrónico, navegar en Internet, editar un documento de texto, chatear, etc.

Recordemos, que un programa es el conjunto de instrucciones que ejecutadas en un ordenador realizarán una tarea o ayudarán al usuario a realizarla.

Nosotros, como programadores y programadoras, creamos un programa, escribiendo su código fuente; con ayuda de un compilador, obtenemos su código binario o interpretado. Este código binario o interpretado, lo guardamos en un fichero. Este fichero, es un fichero ejecutable, llamado comúnmente: ejecutable o binario.

Un ejecutable es un fichero que contiene el código binario o interpretado que será ejecutado en un ordenador.

Ya tenemos más clara la diferencia entre aplicación y ejecutable. Ahora, ¿qué es un proceso? De forma sencilla, un proceso es un programa en ejecución.

## 2 CONTROL DE PROCESOS EN LINUX

### 2.1 Introducción

Todos los ordenadores actuales realizan varias tareas a la vez, por ejemplo, ejecutar un procesador de textos, imprimir un documento, visualizar determinada información por pantalla, etc... Cuando un programa se carga en memoria para su ejecución se convierte en un proceso.

En un sistema operativo multiproceso se puede ejecutar más de un proceso a la vez, dando la sensación al usuario de que cada proceso es el único que se está ejecutando.

En los *sistemas con 1 única CPU* se va alternando la ejecución de los procesos, es decir, se quita un proceso de la CPU, se ejecuta otro y se vuelve a colocar el primero sin que se entere de nada; esta operación se realiza tan rápido que parece que cada proceso tiene una dedicación exclusiva.

Cuando un programa usa la CPU y sale, hay que hacer una especie de “foto” para guardar toda la información referente a ese proceso con el objetivo de reanuncarlo posteriormente en el mismo estado en el que encontraba cuando salió de la CPU.

Esto se conoce como el **BCP** (bloque de control de procesos), que es una estructura de datos que contiene información como **el identificador del proceso**, su **estado** y más información como, por ejemplo, el **estado de los registros de la CPU**.

En Linux podemos ver información asociada a cada proceso tecleando el comando “ps” (process status).

<pre>pepe@pepe-server:~\$ ps   PID TTY          TIME CMD  81480 pts/0    00:00:00 bash  81498 pts/0    00:00:00 ps pepe@pepe-server:~\$</pre>	PID	Identificador del proceso
	TTY	Terminal asociado del que lee y al que escribe
	TIME	Tiempo total de CPU usado
	CMD	Nombre del proceso

(a) ps

(b) detalle

También si tecleamos “ps -f” también aparecerá el PPID, el identificador del proceso padre.

```
pepe@pepe-server:~$ ps
  PID TTY          TIME CMD
 81480 pts/0    00:00:00 bash
 81510 pts/0    00:00:00 ps
pepe@pepe-server:~$ ps -f
UID    PID    PPID  C STIME TTY          TIME CMD
pepe    81480    81462  0 16:45 pts/0    00:00:00 bash
pepe    81517    81480  0 16:46 pts/0    00:00:00 ps -f
pepe@pepe-server:~$
```

(c) ps -f

Fíjate como el proceso que lanza la instrucción “ps -f”, al crearse dentro del bash, se convierte en su “proceso hijo” y así se refleja en el resultado de la instrucción. Dicho de otra forma, el proceso “ps -f” tiene como **PPID** el 81480, que es el **PID** del bash.

En **Windows** podemos usar desde la línea de comandos la orden “tasklist” para ver los procesos que se están ejecutando:

Nombre de imagen	PID	Nombre de sesión	Núm. de ses	Uso de memoria
System Idle Process	0	Services	0	0 KB
System	4	Services	0	4,200 KB
Registry	124	Services	0	26,196 KB
smss.exe	520	Services	0	484 KB
csrss.exe	764	Services	0	2,956 KB
wininit.exe	852	Services	0	1,464 KB
csrss.exe	888	Console	1	2,204 KB
services.exe	924	Services	0	8,880 KB
lsass.exe	944	Services	0	13,824 KB
svchost.exe	992	Services	0	19,152 KB
fontdrvhost.exe	768	Services	0	19,660 KB
csrss.exe	956	Services	0	5,552 KB
svchost.exe	1072	Services	0	12,372 KB
svchost.exe	1120	Services	0	4,180 KB
winlogon.exe	1184	Console	1	2,912 KB
fontdrvhost.exe	1264	Console	1	16,852 KB
smss.exe	1336	Console	1	94,868 KB

(d) tasklist

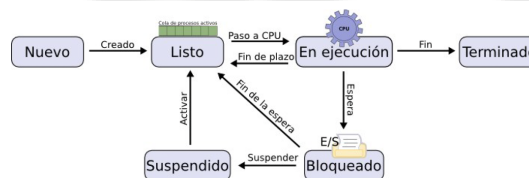
Aunque en Windows también se puede teclear [CTRL+Alt+Supr] para que se muestre por pantalla del Administrador de tareas. Una vez abierto el Administrador de tareas, podemos consultar los programas en ejecución haciendo clic en la pestaña “Procesos”:

Nombre	Estado	CPU	Memoria	Disco	Red
Adobe Acrobat Reader DC (32 b...		0%	325.5 MB	0 MB/s	0 Mbps
Google Chrome (5)		0%	166.2 MB	0 MB/s	0 Mbps
Dropbox (32 bits)		0.2%	150.3 MB	0 MB/s	0 Mbps
Antimalware Service Executable		0%	134.3 MB	0 MB/s	0 Mbps
Google Chrome		0%	126.1 MB	0 MB/s	0 Mbps

(e) tasklist

### 3 Estados de un Proceso

- **Nuevo.** Proceso nuevo, creado.
- **Listo.** Proceso que está esperando la CPU para ejecutar sus instrucciones.
- **En ejecución.** Proceso que actualmente, está en turno de ejecución en la CPU.
- **Bloqueado.** Proceso que está a la espera de que finalice una E/S.
- **Suspendido.** Proceso que se ha llevado a la memoria virtual para liberar, un poco, la RAM del sistema.
- **Terminado.** Proceso que ha finalizado y ya no necesitará más la CPU.



(f) estados de un proceso

## 4 Creación de Procesos en Java

### 4.1 Clases `ProcessBuilder` y `Process`

Java dispone de varias clases para la gestión de procesos:

- `ProcessBuilder`: que define el proceso que se quiere lanzar.
- `Process`: al invocar al método “start” de `ProcessBuilder`, se genera un objeto de tipo `Process`, que se corresponde con un proceso en ejecución.

Por ejemplo, el siguiente código lanzaría la aplicación firefox:

```
import java.io.IOException;

public class Ej1 {
    public static void main(String[] args) throws IOException{
        // TODO Auto-generated method stub

        ProcessBuilder pb = new ProcessBuilder("firefox");
        Process p = pb.start();
    }
}
```

o la version compactada:

```
import java.io.IOException;

public class Ej1 {
    public static void main(String[] args) throws IOException{
        // TODO Auto-generated method stub

        Process p = new ProcessBuilder("firefox").start();
    }
}
```

De hecho, y de un modo genérico, el uso del comando `ProcessBuilder` admite todos los argumentos que necesitemos separados por comas.

```
Process p = new ProcessBuilder("Comando", "Arg1", "Arg2"... "Argn").start();
```

o también un array

```
String[] cmd = {"bash", "-c", "-l"};
Process p = new ProcessBuilder(cmd).start();
```

Como por ejemplo:

```
import java.io.IOException;

public class Ej1 {
    public static void main(String[] args) throws IOException{
        // TODO Auto-generated method stub
        Process p = new ProcessBuilder("firefox","www.marca.com","www.sport.es")
            .start();
    }
}
```

De hecho, desde un programa en Java vamos a ser capaces de ejecutar otro programa nuestro en Java. Para ello, primero hemos de decirle a Java donde se encuentra el binario a ejecutar, para lo que se usa el método **directory()**.

En mi caso, el programa que visita la web de superdeporte está dentro de la carpeta “bin” del proyecto “ProyectoPSP” de Eclipse, por lo que la carpeta es:

**/home/pepe/eclipse-workspace/ProyectoPSP/bin/**

podemos averiguar la ruta en java:

```
final String dir = System.getProperty("user.dir");
System.out.println("current dir = " + dir);
```

Por lo que el código que ejecutaría el programa anterior sería:

```
import java.io.*;

public class Principal {

    public static void main(String[] args) throws IOException{
        // TODO Auto-generated method stub
        final String dir = System.getProperty("user.dir");
        System.out.println("current dir = " + dir);
        File directorio = new File("/Users/pepedevesa/eclipse-
            workspace_new/PSP-2DAM/bin");

        ProcessBuilder pb = new ProcessBuilder("java","Saludo");
        pb.directory(directorio);
        Process p = pb.start();

    }
}
```

## 4.2 Redirigiendo la salida con `getInputStream`

El método `getInputStream` de la clase `Process` en Java se utiliza para obtener un flujo de entrada (input stream) desde el cual se puede leer la salida estándar (stdout) del proceso que se ha iniciado. En otras palabras, cuando se inicia un proceso utilizando `ProcessBuilder` y `start()`, este método permite capturar cualquier salida generada por dicho proceso.

En el siguiente código usaremos el método **`getInputStream`** para leer el stream de salida del proceso, es decir, para leer la ejecución que envía a la consola.

Podeis observar que hay comentado la misma sentencia para con **`BufferedReader`** para leer lineas.

```
import java.io.*;

public class Principal {

    public static void main(String[] args) throws IOException{
        // TODO Auto-generated method stub
        final String dir = System.getProperty("user.dir");
        System.out.println("current dir = " + dir);
        File directorio = new File("/Users/pepedevesa/eclipse-workspace_new
            /PSP-2DAM/bin");

        ProcessBuilder pb = new ProcessBuilder("java","Saludo");
        pb.directory(directorio);

        try {
            Process p = pb.start();

            // Capture and print the output of the process
            InputStream is = p.getInputStream();
            int c;
            while ((c = is.read()) != -1) {
                System.out.print((char)c);
            }

            /*
            BufferedReader reader = new BufferedReader(new
                InputStreamReader(p.getInputStream()));

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }

            */

        } catch (IOException e) {
```



```
        e.printStackTrace();
    }
}
```

Veamos otro ejemplo:

El siguiente programa ejecuta el comando “ls” dentro de un directorio

```
public static void main(String[] args) throws IOException {
    // TODO Auto-generated method stub
    final String dir = System.getProperty("user.dir");
    System.out.println("current dir = " + dir);
    //String[] cmd = {"bash", "-c", "-l"}; de manera alternativa.
    ProcessBuilder pb = new ProcessBuilder("bash", "-c", "ls");
    Process p=pb.start();
    try {

        InputStream is = p.getInputStream();

        int c;
        while((c = is.read())!=-1)
            System.out.print((char)c);

        is.close();

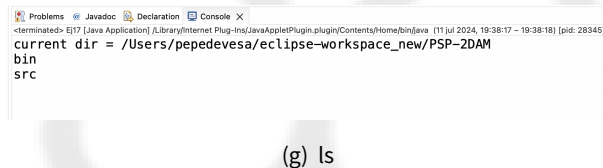
    }catch(Exception e) {
        e.printStackTrace();
    }
    /*
    // Se lee la salida
    InputStream is = pb.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);

    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
    */
}
```

- De este proceso, quédate con lo que se envía a la consola en el stream “is”.
- El método “read” devuelve el siguiente byte de información que, si lo casteamos a un char, nos devuelve el carácter que se está enviando.

Cuando ejecutamos el programa anterior, el resultado de ejecutar el comando “ls” se convierte en la

entrada de nuestro proceso, por lo que somos capaces de escribirlo en la consola de Eclipse:



```
current dir = /Users/pepedevesa/eclipse-workspace_new/PSP-2DAM
bin
src

(g) ls
```

Opcionalmente, podríamos pedirle al proceso actual que espere a que finalice el que hemos creado (proceso ls en este caso), para comprobar que ha terminado bien.

```
int exitVal = p.waitFor();
if (exitVal == 0) {
    System.out.println("Process completed successfully.");
} else {
    System.err.println("Process exited with error code: " +
        exitVal);
}
```

También podemos utilizar el método **getErrorStream()** para capturar los errores que devuelva la ejecución

```
InputStream isError = p.getErrorStream();
int cError;
while ((cError = isError.read()) != -1)
    System.out.print((char) cError);
is.close();
```

### 4.3 Escribiendo en la entrada con **getOutputStream**

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) {
        // TODO Auto-generated method stub
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br= new BufferedReader(in);
        String texto;
        try {
            System.out.println("Introduce Cadena:");
            texto=br.readLine();
            System.out.println("Su Cadena: "+texto);
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        }catch(Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Con el método *getOutputStream* podemos enviar datos a la entrada estándar del programa “EjemploLectura.java” anterior. Por ejemplo, “Hola soy Pepe Devesa\n”.

```
import java.io.*;  
  
public class EjemploEscritura {  
  
    public static void main(String[] args) throws IOException {  
        // TODO Auto-generated method stub  
        File directorio = new File("bin");  
  
        String cadena="Hola soy Pepe...\n";  
        ProcessBuilder pb = new ProcessBuilder("java","EjemploLectura");  
        pb.directory(directorio);  
        Process p=pb.start();  
  
        // escritura -- envia entrada a DATE  
        OutputStream os = p.getOutputStream();  
        os.write(cadena.getBytes());  
        os.flush(); // vacía el buffer de salida  
  
        // lectura -- obtiene la salida de DATE  
        try{  
            InputStream is = p.getInputStream();  
            int c;  
            while ((c = is.read()) != -1)  
                System.out.print((char) c);  
            is.close();  
        }catch(Exception e) {  
            e.getMessage();  
        }  
  
    }  
}
```

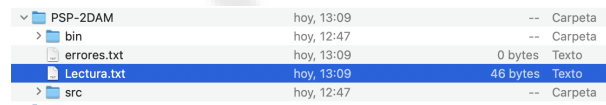


Si solo utilizamos “getOutputStream” y luego no ejecutamos la instrucción “getInputStream”, el programa finalizará correctamente pero no seremos capaces de ver por pantalla el resultado de la ejecución, de ahí que sea necesario hacer uso de los 2 métodos de redirección.

También podemos redireccionar la salida a un fichero:

```
public class EjemploEscritura {  
  
    public static void main(String[] args) throws IOException {  
        // TODO Auto-generated method stub  
        File directorio = new File("bin");  
        String cadena="Hola soy Pepe...\n";  
        ProcessBuilder pb = new ProcessBuilder("java","EjemploLectura");  
        pb.directory(directorio);  
  
        pb.redirectError(new File("errores.txt"));  
        pb.redirectOutput(new File("Lectura.txt"));  
  
        Process p=pb.start();  
  
        // escritura -- envia entrada a DATE  
        OutputStream os = p.getOutputStream();  
        os.write(cadena.getBytes());  
        os.flush(); // vacía el buffer de salida  
  
        // lectura -- obtiene la salida de DATE  
        try{  
            InputStream is = p.getInputStream();  
            int c;  
            while ((c = is.read()) != -1)  
                System.out.print((char) c);  
            is.close();  
        } catch (Exception e) {  
            e.getMessage();  
        }  
  
    }  
  
}
```

Aunque tenemos el método *getInputStream()* redirecciona la salida a un fichero y no lo muestra por pantalla.



(h) redireccionar a un fichero

#### 4.3.1 Métodos que podemos ver en Process

Método	Descripción
int exitValue()	Código de finalización devuelto por el proceso hijo (ver Info más abajo)
Boolean isAlive()	Comprueba si el proceso todavía está en ejecución.
int waitFor()	hace que el proceso padre se quede esperando a que el proceso hijo termine. El entrono que devuelve es el código de finalización del proceso hijo
Boolean waitFor(long timeOut, TimeUnit unit)	El funcionamiento es el mismo que en el caos anterior sólo que en esta ocasión podemos especificar cuánto tiempo queremos esperar a que el proceso hijo termine. El método devuelve true si el proceso termina antes de que pase el tiempo indicado y false si ha pasado el tiempo y el proceso no ha terminado.
void destroy()	Estos dos métodos se utilizan para matar al proceso. El segundo lo hace de forma forzosa.

## 5 Ejercicio

Crea un proyecto llamado **ProcessListPNG** con un programa que le pida al usuario que introduzca una ruta (por ejemplo, /micarpeta/fotos) y luego inicie un proceso que muestre una lista de todas las imágenes PNG encontradas en esta ruta.

## 6 Ejercicio

Crea un proyecto llamado **ProcessKillNotepad** con un programa que arranque el bloc de notas o cualquier editor de texto similar desde tu sistema operativo. Luego, el programa esperará 10 segundos

a que “finalice” el subproceso y, transcurrido ese periodo, será destruido. Para dormir 10 segundos, utilice estas instrucciones: `Thread.sleep(10000);`

## 7 Ejercicio

Crear una clase Java que sea capaz de sumar todos los números comprendidos entre dos valores incluyendo ambos valores.

Para resolverlo crearemos una clase **Sumador** que tenga un método que acepte dos números  $n_1$  y  $n_2$  y que devuelva la suma de todo el intervalo.

Además, incluiremos un método `main` que ejecute la operación de suma tomando los números de la línea de comandos (es decir, se pasan como argumentos al `main`).

Una vez hecha la prueba de la clase `Sumador`, le quitamos el `main`, y crearemos una clase **Lanzador** que sea capaz de lanzar varios procesos utilizando el `ProcessBuilder` de la siguiente manera

## 8 Ejercicio

Vamos a codificar 2 programas Java que se encargarán de interactuar con el navegador Mozilla Firefox del siguiente modo:

- **Leerweb:** solicitará al usuario hasta un máximo de 3 páginas webs que quiera abrir en distintas pestañas de Mozilla. Estas páginas webs las introducirá el usuario en una sola línea e irán separadas por el carácter arroba “@”.

Por ejemplo, imaginemos que quiere visitar las páginas relacionadas con la Universidad de Valencia, la de Alicante y la de Castellón, el usuario debería introducir la siguiente cadena: `www.uv.es@www.uji.es@www.ua.es`. Una vez el usuario introduzca esta información, el proceso “Leerweb” se encargará de reenviarla llamando al proceso “Visitaweb”.

- **Visitaweb:** el proceso recogerá la información que el usuario introdujo en “Leerweb” y abrirá las webs en Mozilla. El proceso estará preparado para:
  - Recibir 1 web: Abrirá Mozilla con esa web
  - Recibir 2 webs: Abrirá Mozilla con esas 2 webs, una en cada pestaña.
  - Recibir 3 webs: Abrirá Mozilla con esas 3 webs, una en cada pestaña.
  - Recibir > 3 webs: Informará del ERROR (demasiadas webs!) y finalizará el programa.