

Procesos. ProcessBuilder I

- ▶ Creación de procesos en Java: a partir de 1.5 de JDK
 - La clase que representa un proceso es: **ProcessBuilder**.
 - Los métodos de **ProcessBuilder.start()** y **Runtime.exec()** crean un proceso nativo en el sistema operativo subyacente y devuelven uno de la clase **Process** que puede ser utilizado para controlar dicho proceso.
- ▶ El método **start()** crea una nueva estancia de **Process** y puede ser invocado varias veces desde la misma instancia para crear nuevos subprocesos.
- ▶ 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
 - Un **entorno** (Environment) con sus variables
 - Un **directorio de trabajo**. El valor por defecto es el directorio de trabajo del proceso en curso
 - Una **fuentes de salida estandar**. `Process.getOutputStream()`.
 - Una **fuentes de entrada estandar y de error**. `Process.getInputStream()` y `Process.getErrorStream()`



Procesos. ProcessBuilder II

- ▶ El método ***start()*** inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso ejecuta el comando y los argumentos indicados en el método ***command()***, ejecutándose en el directorio de trabajo especificado por ***directory()***, utilizando las variables de entorno definidas en ***environment()***
- ▶ El método ***exec(cmdarray, envp, dir)*** ejecuta el comando especificado y argumentos en ***cmdarray*** en un proceso hijo independiente con el entorno ***envp*** y el directorio de trabajo especificado en ***dir***

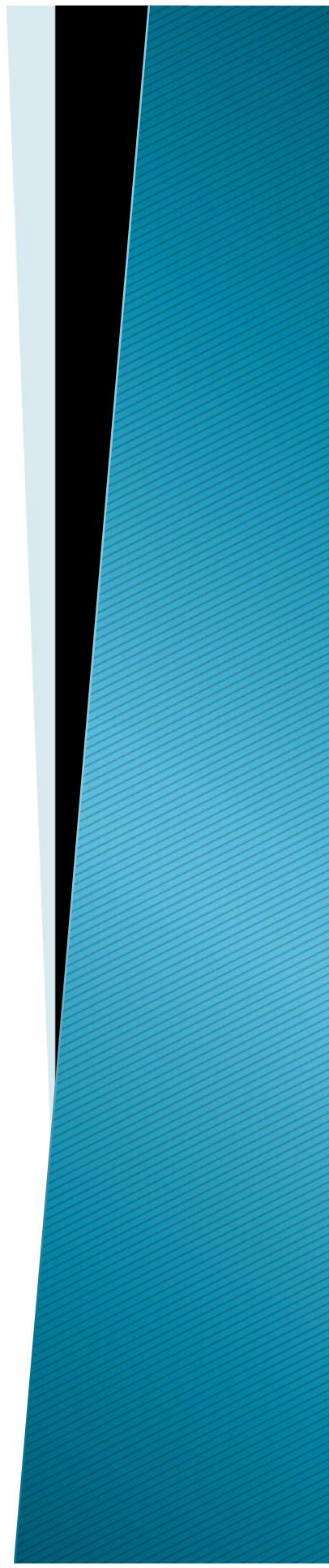


Ej. ProcessBuilder

```
import java.io.IOException;
import java.util.Arrays;
public class RunProcess { //En Argumentos incluir la ejecución de una aplicación
    public static void main(String[] args) throws IOException {
        if (args.length <= 0) {
            System.err.println("Se necesita un programa a ejecutar");
            System.exit(-1);
        }
        ProcessBuilder pb = new ProcessBuilder(args);
        try {
            Process process = pb.start();
            int retorno = process.waitFor(); // El Padre espera a la finalización del hijo
            System.out.println("La ejecución de " + Arrays.toString(args) + " devuelve " + retorno);
        } catch (IOException ex) {
            System.err.println("Excepción de E/S!!");
            System.exit(-1);
        } catch (InterruptedException ex) {
            System.err.println("El proceso hijo finalizó de forma incorrecta");
            System.exit(-1);
        }
    }
}
```



COMUNICACIÓN DE PROCESOS



Comunicación de procesos

- ▶ Un proceso recibe información, la transforma y produce resultados mediante su:
 - **Entrada estándar** (*stdin*): lugar de donde el proceso lee los datos de entrada que requiere para su ejecución.
 - Normalmente es el teclado.
 - No se refiere a los parámetros de ejecución del programa.
 - **Salida estándar** (*stdout*): sitio donde el proceso escribe los resultados que obtiene.
 - Normalmente es la pantalla.
 - **Salida de error** (*stderr*): sitio donde el proceso envía los mensajes de error.
 - Habitualmente es el mismo que la salida estándar, pero puede especificarse que sea otro lugar, cómo un fichero.



Comunicación de procesos

- ▶ En Java, el proceso hijo no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente.
- ▶ *Stdin*, *stdout* y *stderr* están redirigidas al proceso padre a través de los flujos de datos:
 - *OutputStream*: flujo de salida del proceso hijo.
 - El *stream* está conectado por un *pipe* a la entrada estándar (*stdin*) del proceso hijo.
 - *InputStream*: flujo de entrada del proceso hijo.
 - El *stream* está conectado por un *pipe* a la salida estándar (*stdout*) del proceso hijo.
 - *ErrorStream*: flujo de error del proceso hijo.
 - El *stream* está conectado por un *pipe* a la salida estándar (*stderr*) del proceso hijo.
 - Por defecto, está conectado al mismo sitio que *stdout*.



Clase Process. Métodos

Método	Tipo de Retorno	Descripción
getOutputStream()	OutputStream	Obtiene el flujo de salida del proceso hijo conectado a stdin
getInputStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado a stdout
getErrorStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado al stderr del proceso hijo
destroy()	void	Implementa la operación destroy. Obliga la finalización del proceso hijo y libera sus recursos
waitFor()	int	Implementa la operación wait. Bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante exit
exitValue()	int	Obtiene el valor de retorno del proceso hijo



Comunicación de procesos

- ▶ Además de la posibilidad de comunicarse mediante flujos de datos, existen otras alternativas para la comunicación de procesos:
 - Usando *sockets*.
 - Utilizando JNI (*Java Native Interface*) para acceder desde Java a aplicaciones en otros lenguajes de programación de más bajo nivel, que pueden sacar partido al sistema operativo subyacente.
 - Librerías de comunicación no estándares entre procesos. Permiten aumentar las capacidades del estándar Java para comunicar procesos mediante:
 - Memoria compartida: se establece una región de memoria compartida entre varios procesos a la cual pueden acceder todos ellos.
 - *Pipes*: permite a un proceso hijo comunicarse con su padre a través de un canal de comunicación sencillo.
 - Semáforos: mecanismo de bloqueo de un proceso hasta que ocurra un evento.



Comunicación de Procesos

- Comunicación de procesos utilizando un Buffer. Requiere Argumentos.

```
import java.io.BufferedReader;  
import java.io.IOException;
```

```
import java.io.InputStream;  
import java.io.InputStreamReader;  
import java.util.Arrays;
```

```
public class comunicacionentreprocesos {  
    public static void main(String args[])throws IOException{
```

**Compilación y ejecución
desde consola**

Configurar path a bin de java

Compilar

C:\>Javac usaproceso.java

Ejecutar

C:\>Java usaproceso

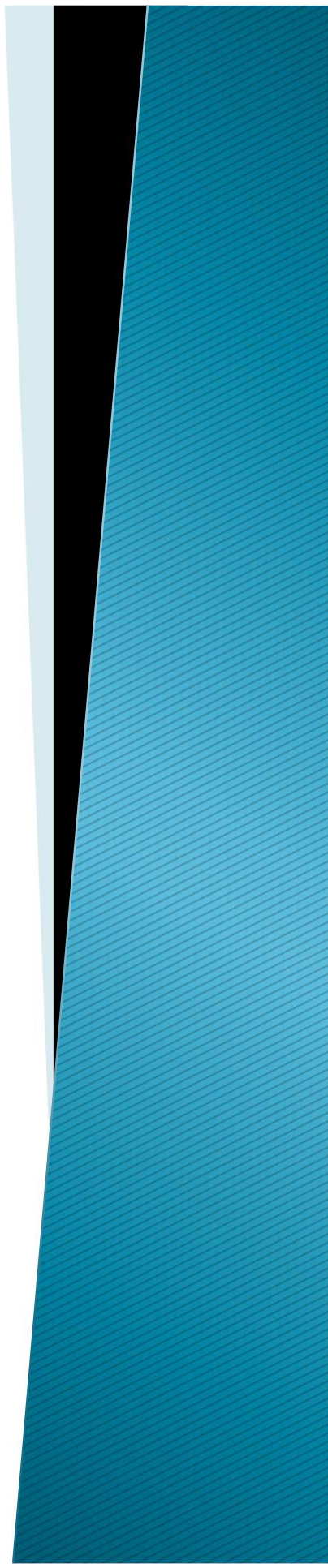
```
Process process= new ProcessBuilder(args).start();  
InputStream is= process.getInputStream();  
InputStreamReader isr= new InputStreamReader(is);  
BufferedReader br = new BufferedReader(isr);  
String line;
```

```
System.out.println("Salida del proceso "+ Arrays.toString  
(args)+ ":");  
while ((line =br.readLine())!=null) {  
    System.out.println(line);  
}
```

```
}
```

```
}
```

SINCRONIZACION DE PROCESOS



Sincronización de procesos

- ▶ Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.
- ▶ Además de la utilización de los flujos de datos se puede esperar por la finalización del proceso hijo mediante la operación ***wait***.
 - Bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante ***exit***.
 - Como resultado el padre recibe la información de finalización del proceso hijo.
 - El valor de retorno especifica mediante un número entero, cómo resultó la ejecución.
 - No tiene nada que ver con los mensajes que se pasan padre e hijo a través de los *streams*.
 - Por convención se utiliza 0 para indicar que el hijo ha acabado de forma correcta.

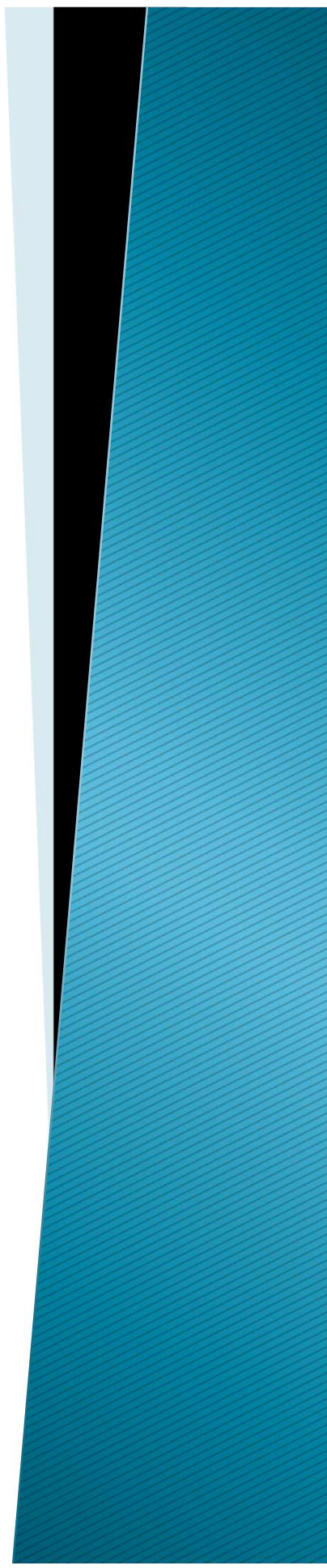


Sincronización de procesos

```
import java.io.IOException;
import java.util.Arrays;
public class ProcessSincronization {
    public static void main(String[] args)
        throws IOException, InterruptedException {
    try{
        Process process = new ProcessBuilder(args).start();
        int retorno = process.waitFor();
        System.out.println("Comando " + Arrays.toString(args) + " devolvió: " + retorno);
    } catch (IOException e) {
        System.out.println("Error ocurrió ejecutando el comando:" + e.getMessage());
    } catch (InterruptedException e) {
        System.out.println("El comando fue interrumpido. Descripción del error: " +
            e.getMessage());
    }
}
}
```



PROGRAMACION MULTIPROCESO



Programación de aplicaciones multiproceso

- ▶ La programación concurrente es una forma eficaz de procesar la información al permitir que diferentes sucesos o procesos se vayan alternando en la CPU para proporcionar multiprogramación.
- ▶ El sistema operativo se encarga de proporcionar multiprogramación entre todos los procesos del sistema
 - Oculta esta complejidad tanto a los usuarios como a los desarrolladores.
- ▶ Si se pretende realizar procesos que cooperen entre sí, debe ser el propio desarrollador quien lo implemente utilizando comunicación y sincronización de procesos.



Creación de Procesos. Multiprocesos

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

```
import java.util.Arrays;
```

```
public class creamultiprocesos {
```

```
    public static void main(String[] args) {
```

```
        ProcessBuilder pb1 = new ProcessBuilder("NOTEPAD");
```

```
        ProcessBuilder pb2 = new ProcessBuilder("EXPLORER");
```

```
        try{
```

```
            Process process =pb1.start();
```

```
            //int retorno = process.waitFor();
```

```
            System.out.println("ejecución de NOTEPAD");
```

```
        }catch(IOException ex) {
```

```
            System.err.println("Excepción de E/S!!");
```

```
            System.exit(-1);
```

```
        }
```

```
        //catch(InterruptedException ex){
```

```
        //System.err.println("El proceso hijo finalizó de forma  
incorrecta");
```

```
        //System.exit(-1);
```

```
        //}
```

```
        try{
```

```
            Process process =pb2.start();
```

```
            //int retorno = process.waitFor();
```

```
            System.out.println("Ejecución de
```

```
EXPLORER");
```

```
        }catch(IOException ex) {
```

```
            System.err.println("Excepción de E/S!!");
```

```
            System.exit(-1);
```

```
        }
```

```
        //catch(InterruptedException ex){
```

```
        //System.err.println("El proceso hijo finalizó de forma  
incorrecta");
```

```
        //System.exit(-1);
```

```
        //}
```

```
    }
```

```
}
```



Programación de aplicaciones multiproceso

► Fases:

- **Descomposición funcional.** Identificar previamente las diferentes cosas que debe realizar la aplicación y las relaciones existentes entre ellas.
- **Partición.** Distribución de las diferentes funciones en procesos estableciendo el esquema de comunicación entre los mismos.
 - La comunicación entre procesos requiere una pérdida de tiempo tanto de comunicación como de sincronización.
 - El objetivo debe ser maximizar la independencia entre los procesos minimizando la comunicación entre los mismos.
- **Implementación.** En este caso, Java permite únicamente métodos sencillos de comunicación y sincronización de procesos para realizar la cooperación.



Ej. Procesos. ProcessBuilder I

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

public class EjemploProcessBuilder {

    Public static void main(String args[]) throws IOException {

        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "dir");

        File fOut = new File ("c:\salida.txt");
        File fErr = new File ("C:\error.txt");

        pb.redirectOutput(fOut);
        pb.redirectError(fWrr);
        pb.start();
    }
}
```

```
ProcessBuilder pb = new ProcessBuilder("cmd.exe", "/C", "start");
Runtime.getRuntime().exec("cmd.exe /C start");
```

Ej. Procesos. ProcessBuilder II

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

```
public class EjemploProcessBuilder2 {
```

```
    Public static void main(String args[]) throws IOException {
```

```
        ProcessBuilder pb = new ProcessBuilder("CMD");
```

```
        File fBat = new File ("c:\fichero.bat");
```

```
        File fOut = new File ("c:\salida.txt");
```

```
        File fErr = new File ("c:\error.txt");
```

```
        pb.redirectinput(fBat);
```

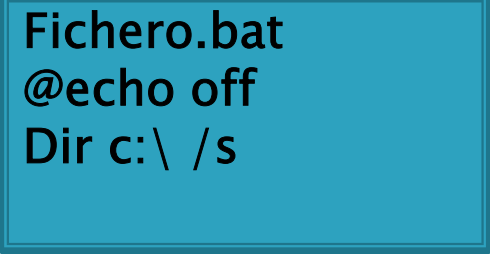
```
        pb.redirectOutput(fOut);
```

```
        pb.redirectError(fErr);
```

```
        pb.start();
```

```
    }
```

```
}
```

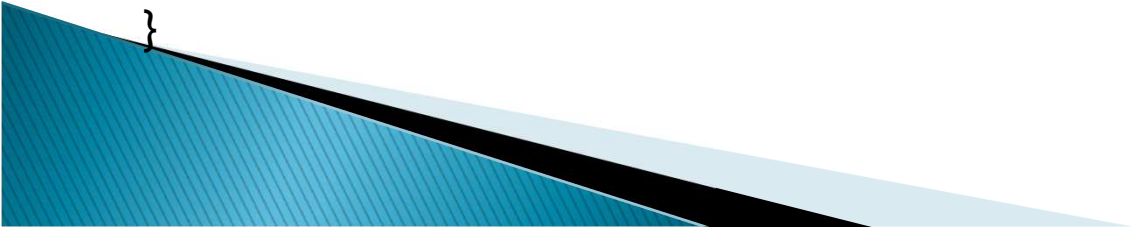


```
Fichero.bat  
@echo off  
Dir c:\ /s
```



Leer Ficheros

```
public class unfichero {  
    public static void leer(File f) throws Exception{  
  
        BufferedReader linea = new BufferedReader(new FileReader(f));  
        try  
        {  
            String cadena=linea.readLine();  
            while(cadena!=null)  
            {  
                System.out.println(cadena);  
                cadena=linea.readLine();  
            }  
        }catch(EOFException e){}  
        linea.close();  
    }  
}
```



Ej. Procesos. Aleatorios. Enunciado

- crear un proceso hijo que esté encargado de generar números aleatorios. Para su creación usar cualquier lenguaje de programación, generando el ejecutable correspondiente.
- Este proceso hijo escribirá en su salida estándar un número aleatorio del 0 al 10 cada vez que reciba una petición de ejecución por parte del padre.
- El proceso padre lee líneas de la entrada estándar y por cada línea que lea solicitará al hijo que le envíe un número aleatorio, lo leerá y lo imprimirá en pantalla.
- Cuando el proceso padre reciba la palabra “fin”, finalizará la ejecución del hijo y procederá a finalizar su ejecución.

