



# MEDAC

Instituto Oficial de Formación Profesional

## TEMA 3. GESTIÓN DE PROCESOS

---

PROGRAMACIÓN DE  
SERVICIOS Y PROCESOS  
2º DAM

Curso 2021-2022

# TEMA 3. GESTIÓN DE PROCESOS

## APARTADO 1. OBJETIVOS

- Crear y lanzar procesos.
- Terminar procesos.
- Conocer los mecanismos de comunicación entre procesos.
- Conocer los mecanismos de sincronización entre procesos.

# TEMA 3. GESTIÓN DE PROCESOS

## APARTADO 2. CREACIÓN DE PROCESOS EN JAVA



Para crear procesos en Java, utilizaremos la **clase Process**. Esta clase nos ofrece los métodos:

- ProcessBuilder.start(): este método inicia un un proceso nuevo.
  - ◆ Va a ejecutar el comando y los argumentos que le indiquemos en el método *command()*.
  - ◆ Se ejecutará en el directorio de trabajo que le indiquemos con el método *directory()*.
  - ◆ Podrá utilizar las variables de entorno del SO que estén definidas en *environment()*.
- Runtime.exec(String[] cmdarray, String[] envp, File dir): este método ejecutará el comando que le especifiquemos con sus correspondientes argumentos en el parámetro cmdarray, **en un proceso hijo** que será totalmente independiente.
  - ◆ Además, se ejecutará el entorno de trabajo indicado en el parámetro envp, y en el directorio de trabajo especificado en el parámetro dir.

Ambos devuelven un objeto de la clase Process.

## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 2. CREACIÓN DE PROCESOS EN JAVA

Los métodos anteriores comprobarán que el comando indicado sea un comando o fichero ejecutable válido en el SO que estemos usando. Al crear un nuevo proceso y dependiendo del SO que estemos usando, pueden ocurrir algunos problemas:

- No encontrar el ejecutable debido a la ruta indicada.
- No tener permisos de ejecución.
- No ser un ejecutable válido en el sistema.





## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 2. CREACIÓN DE PROCESOS EN JAVA

En casi todos los casos, se lanzará una excepción, dependiendo del SO que estemos usando, aunque siempre va a ser una subclase de IOException.

```
public static void main(String[] args) {  
    // Ruta del ejecutable de Google Chrome  
    String RUTA_PROCESO = "C:\\Program Files "  
        + "(x86)\\Google\\Chrome\\Application\\chrome.exe";  
    // Creamos el proceso de Google Chrome  
    ProcessBuilder pb = new ProcessBuilder(RUTA_PROCESO);  
    try {  
        // Lanzamos el proceso  
        Process process = pb.start();  
        // Obtenemos su estado de ejecución  
        int retorno = process.waitFor();  
        System.out.println("La ejecución de " + RUTA_PROCESO +  
            " devuelve " + retorno);  
    } catch (IOException ex) {  
        System.err.println("Error: " + ex.toString());  
        System.exit(-1);  
    } catch (InterruptedException ex) {  
        System.err.println("El proceso hijo finalizó de forma incorrecta");  
        System.exit(-1);  
    }  
}
```

**Actividad:** Ahora lanza tú otro ejecutable.

**Actividad:** Realiza lo mismo pero usando otra alternativa que no sea ProcessBuilder.

Lanzando Google Chrome en un proceso

## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 3. CREACIÓN DE PROCESOS EN JAVA II: LANZAR MÉTODO



En Java podemos lanzar, en un proceso, un método de clase creada por nosotros.

Para esto, debemos crear una clase y el método o métodos que queramos ejecutar en forma de proceso independiente.

Lo siguiente, será crear un método main en el que vamos a llamar al método que queremos ejecutar. Si tiene parámetros, vamos a usar el parámetro del método main `String[] args` para enviar los parámetros que necesitamos.

[Vídeo](#)



## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 3. CREACIÓN DE PROCESOS EN JAVA II: LANZAR MÉTODO

Finalmente, vamos a crear la función que vemos en la siguiente figura, que nos va a permitir ejecutar una clase en un proceso independiente.

```
public static int ejecutarClaseProceso(Class clase, int n1, int n2)
    throws IOException, InterruptedException {
    // Defino dónde está el home de java
    String javaHome = System.getProperty("java.home");
    String javaBin = javaHome
        + File.separator + "bin"
        + File.separator + "java";
    String classpath = System.getProperty("java.class.path");
    // Obtengo el nombre canónico de la clase que se va a ejecutar
    String className = clase.getCanonicalName();

    ProcessBuilder builder = new ProcessBuilder(javaBin, "-cp",
        classpath, className, String.valueOf(n1), String.valueOf(n2));
    Process process = builder.start();
    process.waitFor();
    return process.exitValue();
}
```

Como podemos ver, al método se le pasan tres atributos:

- Uno con el nombre de la clase que queremos ejecutar.
- Dos parámetros de tipo int, que son los parámetros que necesita la clase para ejecutar su método.

Estos parámetros pueden cambiar y ser tantos como necesitemos.



## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 3. CREACIÓN DE PROCESOS EN JAVA II: LANZAR MÉTODO

```
public static int ejecutarClaseProceso(Class clase, int n1, int n2)
    throws IOException, InterruptedException {
    // Defino dónde está el home de java
    String javaHome = System.getProperty("java.home");
    String javaBin = javaHome
        + File.separator + "bin"
        + File.separator + "java";
    String classpath = System.getProperty("java.class.path");
    // Obtengo el nombre canónico de la clase que se va a ejecutar
    String className = clase.getCanonicalName();

    ProcessBuilder builder = new ProcessBuilder(javaBin, "-cp",
        classpath, className, String.valueOf(n1), String.valueOf(n2));
    Process process = builder.start();
    process.waitFor();
    return process.exitValue();
}
```

Una vez hemos hecho todo esto, tenemos que llamar al método de la siguiente forma:

```
int estado= ejecutarClaseProceso(Sumador.class, numero1, numero2);
```

Sumador es el nombre de la clase que queremos ejecutar en un proceso y numero1 y numero2 los dos enteros que queremos sumar.

**Actividad:** Tenéis que mostrar el valor de la suma por la terminal de java.

Cuando estamos creando un proceso con ProcessBuilder, los dos últimos parámetros que pasemos son dos enteros citados, que son los que necesita la clase que queremos ejecutar.

Aquí podemos pasarle todos los parámetros que necesitemos (o ninguno, si es el caso), que obligatoriamente tienen que ser de tipo String, ya que, en el main de la clase vamos a usar el array de Strings que recibe para recuperar todos los parámetros que estamos pasando en este punto



## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 5. TERMINAR UN PROCESO

En Java en el momento que necesitemos vamos a poder finalizar un proceso hijo que se haya creado.



#### ¿Cómo podemos hacerlo?

Ejecutando el método **destroy()**, que pertenece a la clase Process.



Este método, lo que hace es que va a eliminar **el proceso hijo** que indiquemos, liberando a su vez todos los recursos que estuviera usando, así los deja disponibles para que el SO pueda asignarlos de nuevo.



En caso de no forzar la finalización de la ejecución de un proceso hijo, este se va a ejecutar de forma normal y completa, terminando y liberando sus recursos al finalizar. Esto se puede producir cuando el proceso hijo ejecuta la operación exit para forzar la finalización de su ejecución.

# TEMA 3. GESTIÓN DE PROCESOS

## APARTADO 5. TERMINAR UN PROCESO

En la siguiente imagen, vamos a ver el código necesario para *lanzar un proceso* de la app Google Chrome y después preguntar al usuario si quiere destruirlo:

```
public static void main(String[] args)
{
    // Ruta del ejecutable de Google Chrome
    String RUTA_PROCESO = "C:\\\\Program Files (x86)\\\\Google\\\\Chrome"
        + "\\Application\\\\chrome.exe";
    // Creamos el proceso
    ProcessBuilder pb = new ProcessBuilder(RUTA_PROCESO);
    Scanner teclado = new Scanner(System.in);

    try
    {
        // Lanzamos el proceso
        Process process = pb.start();
        System.out.println("¿Terminar el proceso? (S/N)");
        if (teclado.nextLine().charAt(0) == 'S')
        {
            // Destruimos el proceso
            process.destroy();
        }
    }
    catch (IOException ex)
    {
        System.err.println("Error: " + ex.toString());
        System.exit(-1);
    }
}
```

Como podemos observar, primero lanzamos el proceso de forma normal, y en caso de que queramos terminar su ejecución, llamaremos al método `destroy`.

**Actividad:** Ahora termina otro proceso.

Este código lo podemos encontrar en la página 7 del pdf.





## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 6. COMUNICACIÓN ENTRE PROCESOS

No se nos puede olvidar que los procesos no son más que **un programa en ejecución** y como tal podrá:

- Pedir y leer información.
- Procesarla y mostrarla por pantalla.



Para leer y mostrar la información contamos con:

- La entrada de datos estándar (**stdin**): por aquí vamos a poder obtener los datos necesarios para que nuestro proceso se ejecute. Vamos a poder leer desde teclado, desde un fichero, etc.
- La salida estándar (**stdout**): por aquí, podremos mostrar información ya sea por pantalla, por un fichero, etc.
- La salida de error (**stderr**): por esta salida vamos a mostrar la información de los errores que ocurran en la ejecución de nuestros procesos.

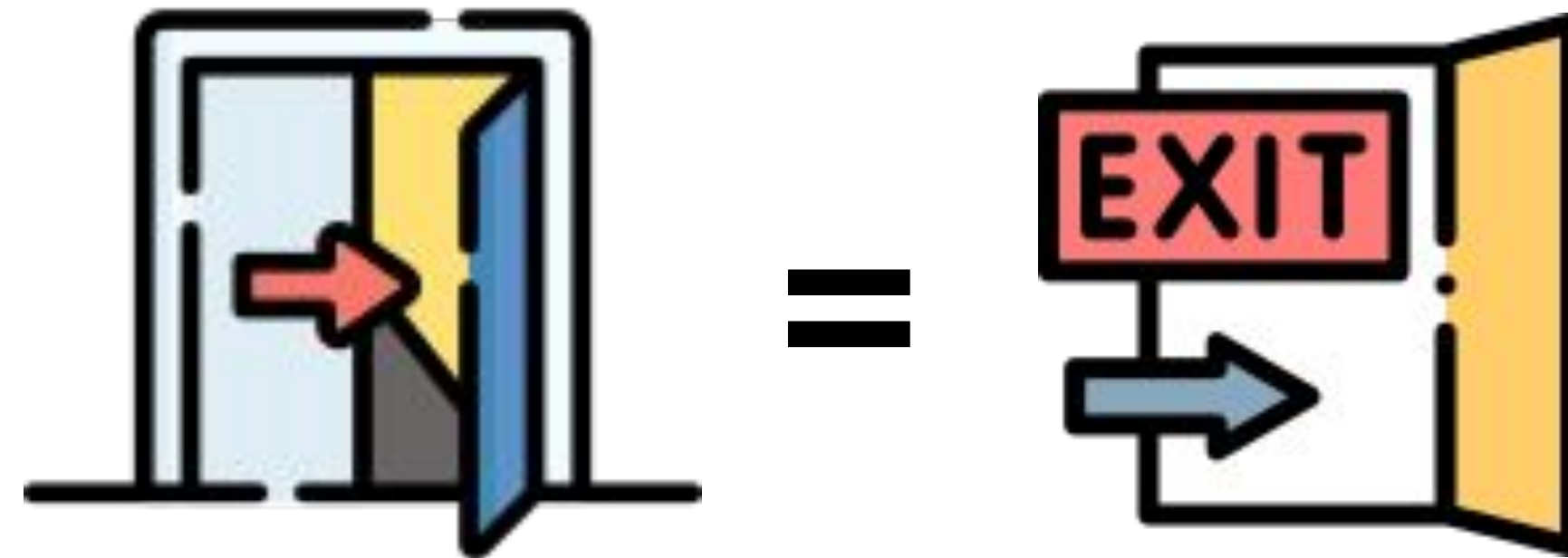
## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 6. COMUNICACIÓN ENTRE PROCESOS

Lo normal, es que las entradas y las salidas de datos de un proceso hijo sean las mismas que las de su proceso padre.

Porque si creamos un proceso hijo dentro de un programa que lee la información de un fichero y muestra los resultados por pantalla, el proceso hijo que creamos lo hará de igual forma.

Lo normal...



Pero...



## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 6. COMUNICACIÓN ENTRE PROCESOS

Pero nosotros al usar Java no ocurre así, el proceso hijo que creemos no va a tener su propia interfaz de comunicación, entonces no podrá comunicarse con el proceso padre directamente. Por lo que, debemos redireccionar todas sus salidas y entradas mediante:



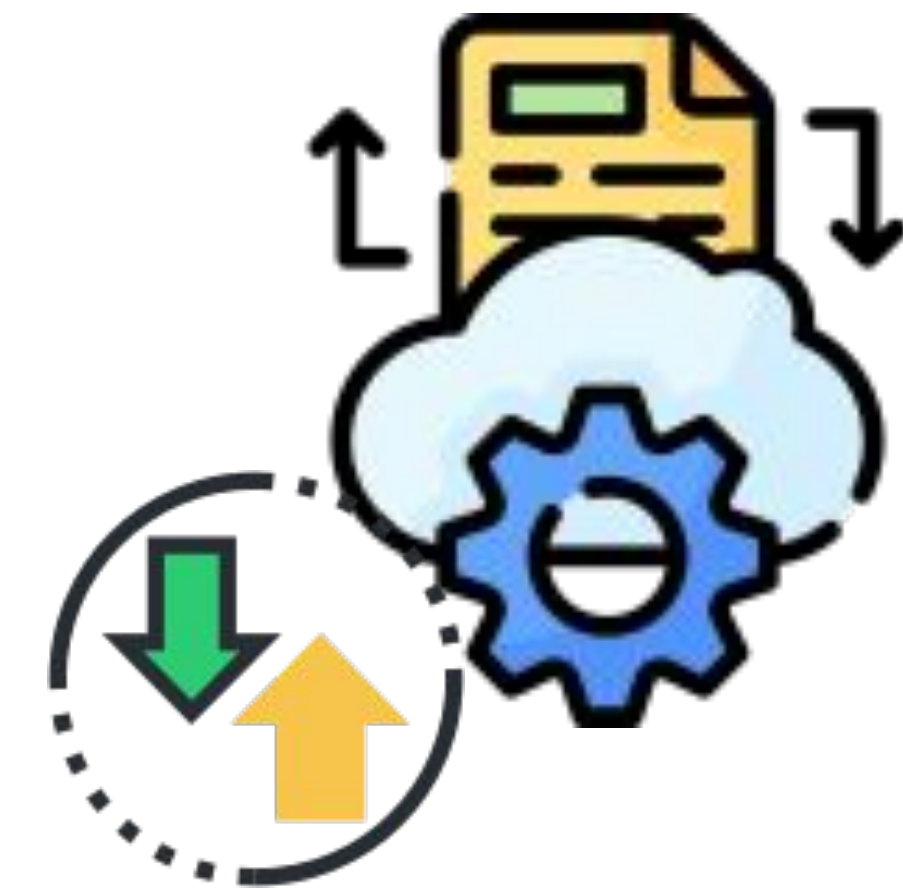
- **OutputStream**: es el flujo de entrada que está conectado a la entrada estándar del proceso hijo. Se redirecciona con el método ***redirectOutput***.
- **InputStream**: es el flujo de salida que está conectado a la salida estándar del proceso hijo. Se redirecciona con el método ***redirectInput***.
- **ErrorStream**: es el flujo de salida para los errores. Está conectado a la salida estándar de errores del proceso hijo y se redirecciona con el método ***redirectError***.

Una vez hayamos redirigido todos estos flujos, podremos hacer una comunicación entre los procesos padre e hijo. Pero si queremos redireccionar a la salida estándar, usaremos **Redirect.INHERIT**.

## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 9. SINCRONIZACIÓN ENTRE PROCESOS.

Puede ocurrir que, estemos lanzando más de un proyecto al mismo tiempo. En ese caso, no tenemos forma de saber qué línea de código estará ejecutando cada uno de ellos en un momento dado.



Esto puede ser **muy peligroso**, porque , si dos o más procesos necesitan acceder, por ejemplo, a una variable en memoria, es posible que alguno de ellos la modifique y los demás ya no puedan ver su valor original, sino el ha sido modificado por el otro proceso que accedió a ella antes.

Esto es lo que conocemos como **zona crítica**, y nos las vamos a encontrar en todos y cada uno de los programas que tengan más de un proceso en activo. Estas zonas son **muy peligrosas** y **deben protegerse** mediante una serie mecanismos, los más comunes son:

- Semáforos.
- Colas de mensajes.
- Pipes o tuberías.
- Bloques de memoria compartida.



Todo esto lo veremos en los próximos temas.



## TEMA 3. GESTIÓN DE PROCESOS

### APARTADO 9. SINCRONIZACIÓN ENTRE PROCESOS.

En Java, tenemos una forma muy rápida de hacer que un bloque de código esté sincronizado entre varios procesos, consiste en incluir delante la palabra reservada **synchronized**; así, la propia máquina virtual de Java hace que ese código sea seguro en la ejecución de dos o más procesos.

```
int i = 0, j = 0;

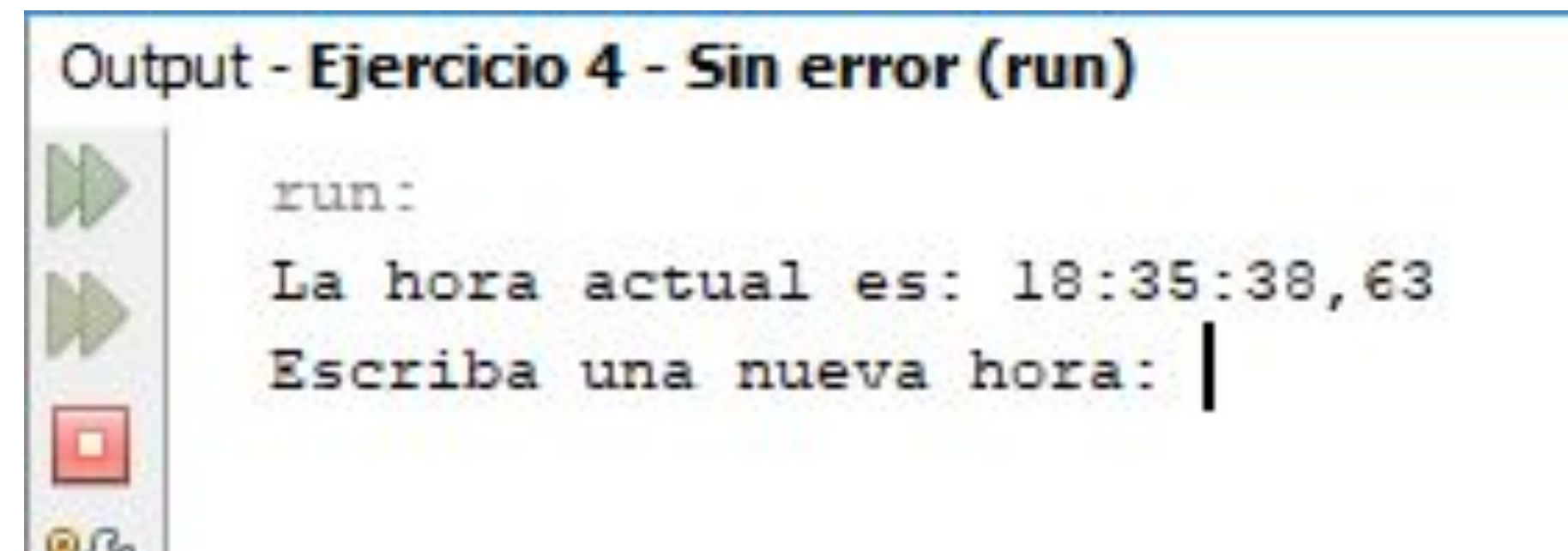
synchronized(MiClase.class)
{
    if (i >= 2)
    {
        i++;
        j++;
    }
    System.out.println("Ok");
    i = i * 2;
    j--;
}
```

## TEMA 3. GESTIÓN DE PROCESOS

### ACTIVIDAD

#### ACTIVIDAD 1:

Lanzar un proceso que ejecute en la consola de Windows el comando “time” y lo muestre en la consola de NetBeans.



The screenshot shows the 'Output' window in NetBeans, titled 'Output - Ejercicio 4 - Sin error (run)'. On the left side of the window, there are three green right-pointing arrows and a red square stop button. The output text is as follows:

```
run:  
La hora actual es: 18:35:38,63  
Escriba una nueva hora: |
```

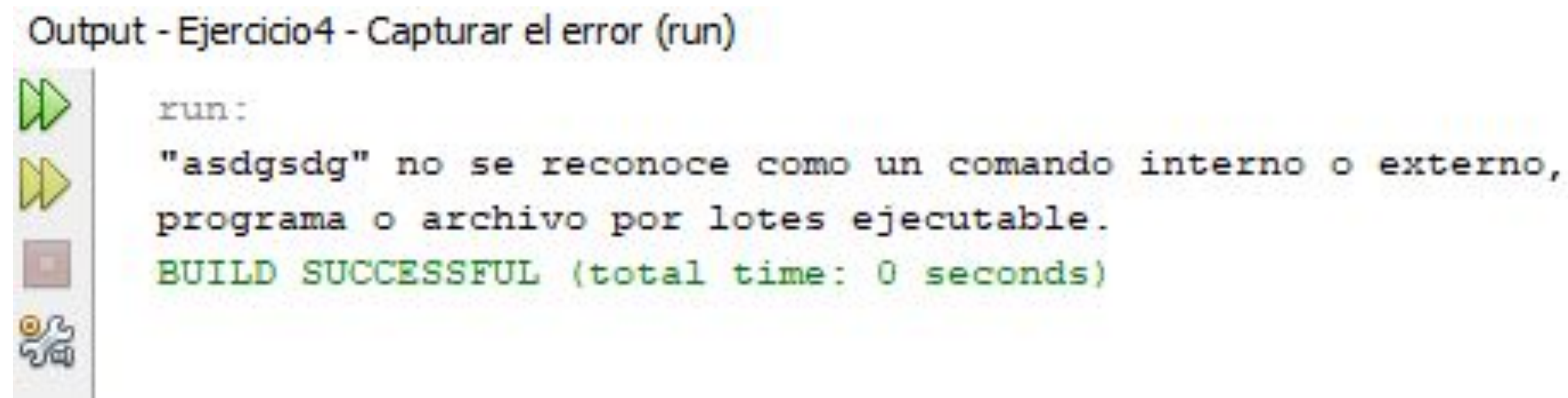


## TEMA 3. GESTIÓN DE PROCESOS

### ACTIVIDAD

#### ACTIVIDAD 2:

Lanzar un proceso que ejecute en la consola de Windows un comando mal escrito como puede ser “askdjah” y lo muestre en la consola de NetBeans.



Output - Ejercicio4 - Capturar el error (run)

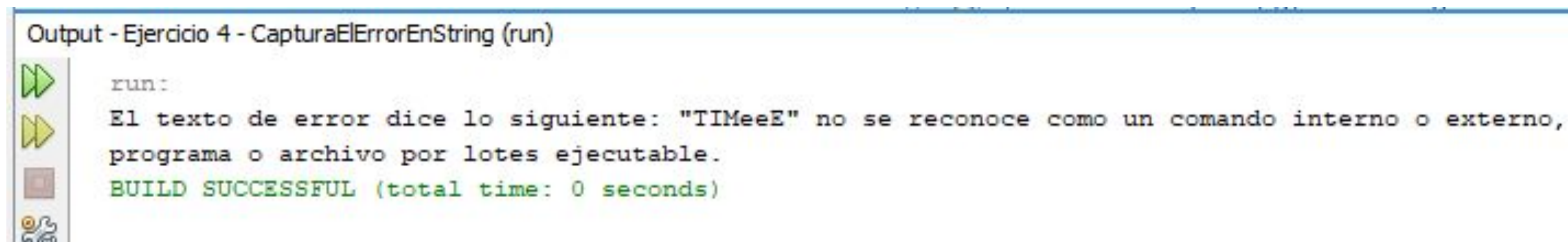
```
run:
"asdgsdg" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
BUILD SUCCESSFUL (total time: 0 seconds)
```

## TEMA 3. GESTIÓN DE PROCESOS

### ACTIVIDAD

#### ACTIVIDAD 3:

Lanzar un proceso que ejecute en la consola de Windows un comando mal escrito como puede ser “askdjah” y lo guarde en un “string” muestre en la consola de NetBeans que diga “El texto de error dice lo siguiente....” .



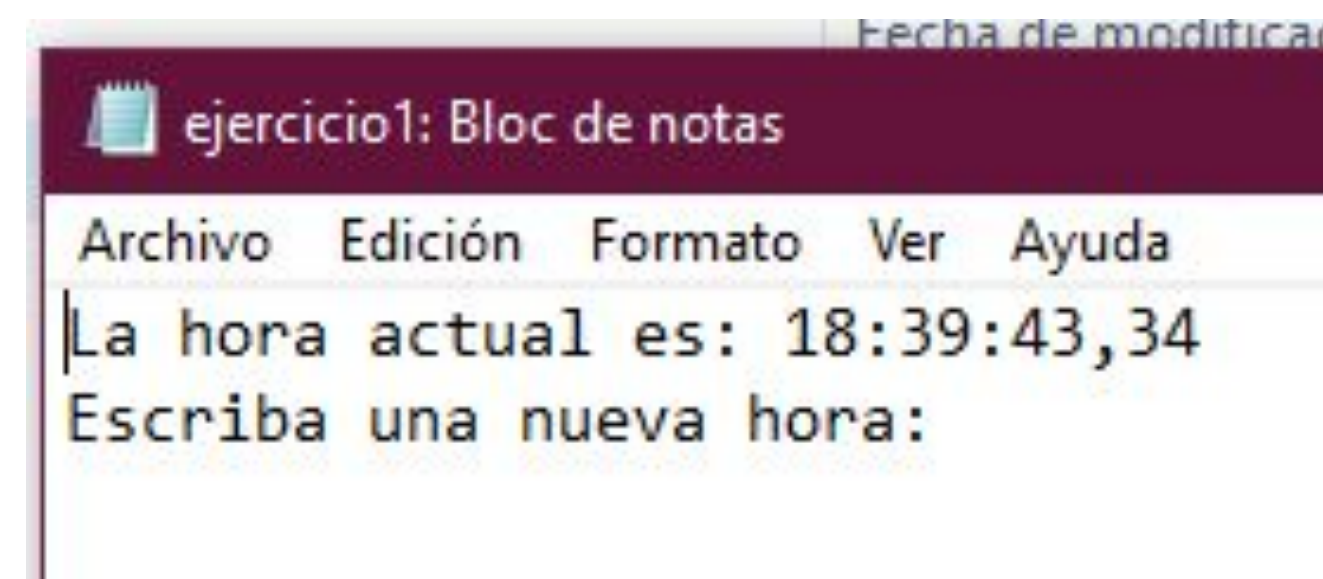
```
Output - Ejercicio 4 - CapturaElErrorEnString (run)
run:
El texto de error dice lo siguiente: "TIMeeE" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
BUILD SUCCESSFUL (total time: 0 seconds)
```

## TEMA 3. GESTIÓN DE PROCESOS

### ACTIVIDAD

#### ACTIVIDAD 4:

Lanzar un proceso que obtenga la hora actual del equipo y la guarde en un archivo de texto.





## TEMA 3. GESTIÓN DE PROCESOS

### REPASO



## ¿Qué hace la clase ProcessBuilder?

Esta clase facilita el envío de comandos a través de la línea de comandos.

Todo lo que necesita es una lista de cadenas que conforman los comandos que se ingresará. Después simplemente llamará al método `start()` en su instancia de `ProcessBuilder` para ejecutar el comando.

Cosas que debemos saber:

- Los comandos deben ser una cadena de `String`.
- Los comandos deben estar en orden que sería como si se hiciera la llamada al programa en la misma línea de comandos, es decir, el nombre del archivo.exe no puede ir después del primer argumento.

## TEMA 3. GESTIÓN DE PROCESOS

### REPASO

## Bloqueo frente a llamadas

En general, al realizar una llamada a la línea de comandos, el programa envía el comando y luego continúa su ejecución.

Sin embargo, es posible que quiera esperar a que finalice el programa llamado antes de continuar con su propia ejecución. Por ejemplo, si el programa llamado va a escribir los datos en un archivo y su programa necesita ese acceso para acceder a esos datos.

Esto se puede hacer fácilmente llamando al método **waitFor()** desde la instancia Process devuelta.



### ¿Qué hace la clase Runtime?

Hace lo mismo que ProcessBuilder, pero su sintaxis sería:

```
Process p= Runtime.getRuntime().exec("Cadena");
```

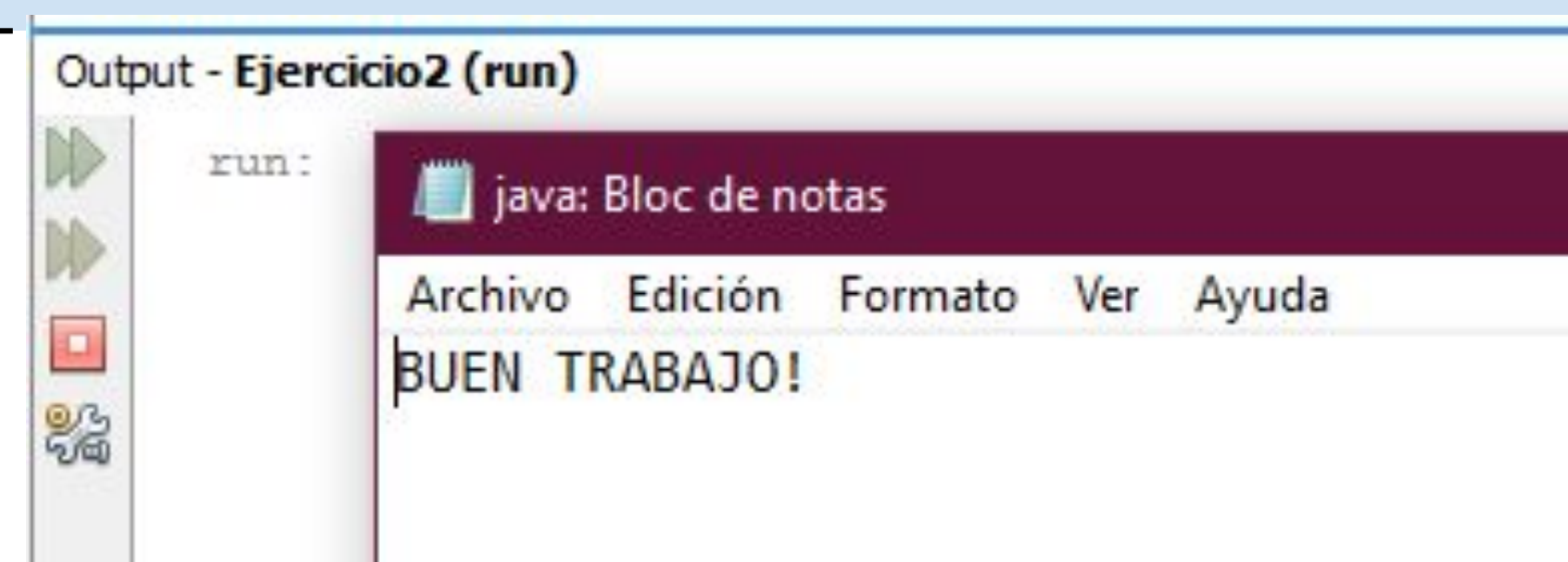
Aquí podemos pasarle un array de String, directamente las cadenas separadas por espacios o "cadena" + "cadena"

## TEMA 3. GESTIÓN DE PROCESOS

### ACTIVIDAD

#### ACTIVIDAD 5:

Crear un documento.txt con el mensaje que queramos en la ubicación que queramos de nuestro pc y lanzar un proceso que ejecute el editor de textos y abra el archivo creado previamente. Lo haremos primero con ProcessBuilder y después con Runtime.





## TEMA 3. GESTIÓN DE PROCESOS

### ACTIVIDAD

#### ACTIVIDAD 6:

Antes teníamos un proceso que nos sumaba los números que había entre un número y otro.  
Teníamos lo siguiente:

$$\text{Sumar}(1,10) = 55$$



Pero, ¿qué podemos hacer si queremos que esta operación se haga de una forma más rápida?

Una idea sería lanzar en paralelo dos procesos.

Un proceso que nos sumará del 1 al 5:

P1 -> sumar (1,5) = 15

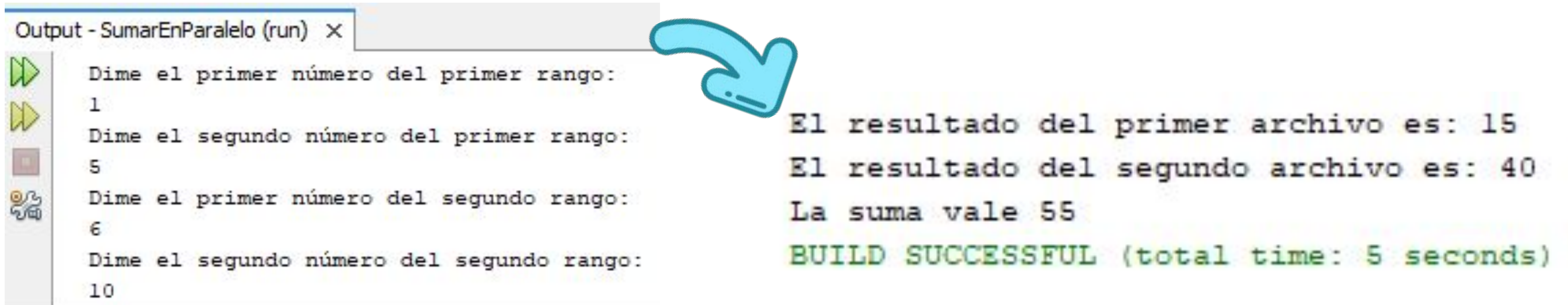
Otro proceso, que nos sume del 1 al 6:

P2 -> sumar(6,10) = 40

Finalmente, esperamos a que acaben ambos programas para sumar 15+40 y que nos de 55.

### ACTIVIDAD 6:

Lanzar dos procesos en paralelo del primer programa que será Sumar, que sumen los números de dos rangos, esta vez no lo vamos a hacer con `getInputStream()` sino que lo haremos con `redirect`, que es un método que crea ficheros. Por tanto vamos a leer esos valores desde ficheros y finalmente, mostraremos el resultado por pantalla.



```
Output - SumarEnParalelo (run) X
Dime el primer número del primer rango:
1
Dime el segundo número del primer rango:
5
Dime el primer número del segundo rango:
6
Dime el segundo número del segundo rango:
10

El resultado del primer archivo es: 15
El resultado del segundo archivo es: 40
La suma vale 55
BUILD SUCCESSFUL (total time: 5 seconds)
```



# MEDAC

Instituto Oficial de Formación Profesional

- Muchas gracias -

---