Comprobar el sistema operativo `System.getProperty("os.name");`

```
Arranca el mspaint:
Process nuevoProceso; //Definimos una variable de tipo Process

nevoProceso = Runtime.getRuntime().exec("mspaint.exe");
```

```
1) Lista por pantalla el contenido del directorio con Process
Runtime builder = Runtime.getRuntime();
		try {
			//execute the command and save the process in out
			Process out = builder.exec("CMD /C DIR");
			//take the result and print it on screen
			BufferedReader bf = new BufferedReader(new
InputStreamReader(out.getInputStream()));
			String linea;
			while( (linea=bf.readLine()) != null) {
				System.out.println(linea);
			}
		} catch (Exception ex) {
			ex.printStackTrace();
		}

2) Lista por pantalla el contenido del directorio con Process y ProcessBuilder:

Process process;
try {
	process = new ProcessBuilder("CMD","/C","DIR").start();
	//ProcessBuilder.directory(new File("ruta")); donde ruta = la carpeta del
ejecutable

	//Se lee la salida
	InputStream is = process.getInputStream();
	InputStreamReader isr = new InputStreamReader(is);
	BufferedReader br = new BufferedReader(isr);

	String line;
	while ((line = br.readLine()) != null) {
		System.out.println(line);
	}
} catch (IOException e) {
	e.printStackTrace();
}
```

```
En el paquete java.lang hay dos clases para la gestión de procesos:
java.lang.ProcessBuilder
java.lang.Process
```

Las instancias de **ProcessBuilder** gestionan los atributos de los procesos, mientras que las instancias de **Process** controlan la ejecución de esos mismos procesos cuando se ejecutan.

Antes de ejecutar un nuevo proceso, podemos configurar los parámetros de ejecución del mismo usando la clase ProcessBuilder.

ProcessBuilder es una clase auxiliar de la clase Process, que veremos más adelante, y se utiliza para controlar algunos parámetros de ejecución que afectarán al proceso. A través de la llamada al método start se crea un nuevo proceso en el sistema con los atributos definidos en la instancia de ProcessBuilder.

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

```
Process p = pb.start();
```

Si llamamos varias veces al método start, se crearán tantos nuevos procesos como llamadas hagamos, todos ellos con los mismos atributos.

La clase ProcessBuilder define un par de constructores:

```
ProcessBuilder(List<String> command)
```

```
ProcessBuilder(String... command)
```

El funcionamiento de ambos es el mismo. En el primer constructor se pasa el comando a ejecutar y la lista de argumentos como una lista de cadenas. Por contra, en el segundo constructor, el comando y sus argumentos se pasan a través de un número variable de cadenas (String ... es lo que en Java se llama varargs). La versión que utilicemos depende del formato en que tengamos los datos.

```
// Formas diferentes de pasar el comando a los constructores de ProcessBuilder
// 1ª forma: usando una cadena. Falla con parámetros
// Sólo funciona con programas que tengan argumentos
String command1 = "notepad.exe"
ProcessBuilder pb = new ProcessBuilder(command1);

// 2ª forma: usando un array de cadenas. Funciona con parámetros
String[] command2 = {"cmd", "/c", "dir", "/o"};
ProcessBuilder pb = new ProcessBuilder(command2);

// 3ª forma: usando una cadena y dividiéndola para convertirla en una lista
String command3 =  "c:/windows/system32/shutdown -s -t 0";
// La expresión regular \s significa partir por los espacios en blanco
ProcessBuilder pb = new ProcessBuilder(command3.split("\\s"));
// ESTA ES LA MEJOR FORMA PARA QUE FUNCIONE EN TODOS LOS CASOS
```

Apagar el sistema operativo

El comando shutdown -s sirve para apagar el sistema. En windows es necesario proporcionar la ruta completa al comando, por ejemplo C:\Windows\System32\shutdown.
Podemos usar como parámetro -s para apagar el sistema, , -r para reiniciar , -h para hibernar y -t para indicar un tiempo de espera antes de apagar.https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/shutdown

Activity psp.activities.U2A1_Shutdowner

Crea una nuevo proyecto Java (package psp.actividades y como clase principal U2A1_Shutdowner). Usando la línea de comandos, pide al usuario qué acción quiere realizar (apagar, reiniciar o suspender) y cuánto tiempo quiere dejar antes de realizar la acción de apagado del sistema..

Busca información sobre el funcionamiento del comando shutdown en GNU/Linux y haz que tu aplicación funcione para ambos sistemas..

La aplicación tiene que preparar el comando correcto para la selección que haya hecho el usuario y para el sistema operativo en el que la esté ejecutando.

Muestra por consola el resultado del método ProcessBuilder.command() de forma legible

```
public class U2A1_Shutdowner {

    public static void main(String[] args) throws IOException {
        // Ask for the required information to prepare the command
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Select your option (s-shutdown / r-reboot / h-hibernate): ");
        String shutdownOption = keyboard.nextLine();

        System.out.print("How much seconds will the command wait to be run? (0 means immediately): ");
        String shutdownTime = keyboard.nextLine();

        // Prepare the command
        String command;
        if (System.getProperty("os.name").toLowerCase().startsWith("windows")) {
```

```
        command = "C:/Windows/System32/shutdown -" + shutdownOption + " -t " +
shutdownTime;
    } else {
        command = "shutdown -" + shutdownOption + " -t " + shutdownTime;
    }

    // Prepare the process and launch it
    ProcessBuilder shutdowner = new ProcessBuilder(command.split("\\s"));
    //shutdowner.start();

    // Show the command to be run
    System.out.print("El comando a ejecutar es:  ");
    for (String commandPart: shutdowner.command()) {
        System.out.print(commandPart + " ");
    }
    System.out.println("");
  }
}
```

Configuraciones adicionales de un proceso

Algunos de los atributos que podemos configurar para un proceso son:

- Establecer el directorio de trabajo donde el proceso se ejecutará

Podemos cambiar el directorio de trabajo por defecto llamando al método directory y pasándole un objeto de tipo File. **Por defecto, el directorio de trabajo se establece al valor de la variable del sistema user.dir**. Este directorio es el punto de partida para acceder a ficheros, imágenes y todos los recursos que necesite nuestra aplicación.

```
// Cambia el directorio de trabajo a la carpeta personal del usuario
pbuilder.directory(new File(System.getProperty("user.home")));
```

Configurar o modificar variables de entorno para el proceso con el método environment()

```
// Retrieve and modify the process environment
Map<String, String> environment = pbuilder.environment();
// Get the PATH environment variable and add a new directory
String systemPath = environment.get("path") + ";c:/users/public";
```

```
environment.replace("path", systemPath);
// Add a new environment variable and use it as a part of the command
environment.put("GREETING", "Hola Mundo");
processBuilder.command("/bin/bash", "-c", "echo $GREETING");

// Indicamos el directorio donde se encuentra el ejecutable
File directorio = new File ("bin");
pb.directory(directorio);

// Mostramos la información de las variables de entorno
Map variablesEntorno = pb.environment();
System.out.println(variablesEntorno);

// Mostramos el nombre del proceso y sus argumentos
List command = pb.command();
Iterator iter = l.iterator();
while (iter.hasNext()) {
  System.out.println(iter.next());
}
```

`Variables de entorno vs Propiedades del sistema`

Con la clase Runtime accedemos a las variables del sistema mientras que con ProcessBuilder lo hacemos a las propiedades del sistema, que son diferentes.

`Redireccionar la entrada y salida estándar`

- Heredar la E/S estándar del proceso padre usando el método ProcessBuilder.inheritIO()

Estas dos configuraciones se verán en el siguiente apartado.

`Actividad psp.activities.U2A2_DirectorioTrabajo`

Crea un nuevo proyecto Java (Ant > Java Application) (configura como nombre del proyecto U2A2_DirectorioTrabajo y como clase principal psp.activities.U2A2_WorkingDirectory) Escribe un programa que ejecute el comando ls o dir. Modifica el valor de la propiedad user.dir. En la misma aplicación, cambiar el directorio de trabajo a la carpeta c:/temp o /tmp, dependiendo del sistema operativo.

Muestra el valor devuelto por el método directory()

- Después de crear la instancia de ProcessBuilder
  - Después de cambiar el valor de user.dir
  - Después de cambiar el directorio de trabajo al directorio temporal del sistema.

En este momento tu programa todavía no mostrará ningún listado.

```
public class U2A2_WorkingDirectory {

  public static void main(String[] args) throws IOException, InterruptedException {
    // Prepare the command
    String command;
    if (System.getProperty("os.name").toLowerCase().startsWith("windows")) {
      command = "cmd /c dir";
    } else {
      command = "sh -c ls";
    }

    //1st - Default working directory

    // Prepare the process
    ProcessBuilder commander = new ProcessBuilder(command.split("\\s"));
    commander.inheritIO();

    // Show Process and System properties
    System.out.println("Working directory: " + commander.directory());
    System.out.println("user.dir variable: " + System.getProperty("user.dir"));

    // Launch the process and show its result
    // Working directory is null but the process is run on the current dir
    commander.start().waitFor();


    //2nd - Change user.dir but not the working directory

    // Change the user.dir system property
    System.setProperty("user.dir", System.getProperty("user.home"));
```

```
    // Prepare the process
    commander = new ProcessBuilder(command.split("\\s"));
    commander.inheritIO();

    // Show Process and System properties
    System.out.println("Working directory: " + commander.directory());
    System.out.println("user.dir variable: " + System.getProperty("user.dir"));

    // Launch the process and show its result
    // Working directory is null but the process is run on the current dir
    commander.start().waitFor();


    // 3rd - Change the working directory

    // Prepare the process
    commander = new ProcessBuilder(command.split("\\s"));
    commander.inheritIO();

    // Show Process and System properties
    commander.directory(new File(System.getProperty("user.home")));
    System.out.println("Working directory: " + commander.directory());
    System.out.println("user.dir variable: " + System.getProperty("user.dir"));

    // Launch the process and show its result
    // Working directory is user.home and the process is run on it
    commander.start().waitFor();
  }
}
```

2.2.2 Acceso al proceso una vez en ejecución

LA clase Process es una clase abstracta definida en el paquete java.lang y contiene la información del proceso en ejecución. Tras invocar al método start de ProcessBuilder, éste devuelve una referencia al proceso en forma de objeto Process.

Los métodos de la clase Process pueden ser usados para realizar operaciones de E/S desde el proceso, para comprobar su estado, su valor de retorno, para esperar a que termine de ejecutarse

y para forzar la terminación del proceso. Sin embargo estos métodos no funcionan sobre procesos especiales del SO como pueden ser servicios, shell scripts, demonios, etc.

Entrada / Salida desde el proceso hijo

Curiosamente **los procesos lanzados con el método start() no tienen una consola asignada.**. Por contra, estos procesos redireccionan los streams de E/S estándar (stdin, stdout, stderr) al proceso padre. Si se necesita, se puede acceder a ellos a través de los streams obtenidos con los métodos definidos en la clase Process como getInputStream(), getOutputStream() y getErrorSteam(). Esta es la forma de enviar y recibir información desde los subprocesos.

Los principales métodos de esta clase son:

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.
Process destroyForcibly()	

Veamos un sencillo ejemplo. Una vez lanzado el programa espera durante 10 segundos y a continuación mata el proceso.

```
public class ProcessDemo {

  public static void main(String[] args) throws Exception {

    ProcessBuilder pb = new ProcessBuilder("C:\\Program
Files\\Notepad++\\notepad++.exe");
            Process p;
              try {
                    p = pb.start();
                    boolean alive = p.isAlive();// Check is process is alive or not
               if (p.waitFor(10, TimeUnit.SECONDS)) { // Wait for the process to end for
10 seconds.
                              System.out.println("Process has finished");
               } else {
                              System.out.println("Timeout. Process hasn't finished");
```

```
                }
                // Force process termination.
            p.destroy();
            // Check again if process remains alive
            alive = p.isAlive();
            // Get the process exit value
  int status = p.exitValue();
            System.out.println(status);
            } catch (InterruptedException | IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }


 }
}
```

Códigos de terminación

Un código de salida (exit code o a veces también return code) es el valor que un proceso le devuelve a su proceso padre para indicarle cómo ha acabado. Si un proceso acaba con un valor de finalización 0 es que todo ha ido bien, cualquier otro valor entre 1 to 255 indica alguna causa de error.

Gestión de excepciones

La llamada al método **waitFor** hace que el proceso padre se bloquee hasta que el proceso hijo termina o bien hasta que el bloqueo es interrumpido por alguna señal del sistema (Excepción) que recibe el proceso padre.

Es mejor gestionar las excepciones lo más cerca posible del origen en vez de pasarlas a niveles superiores.

4. Escribe en un fichero dentro del proyecto el contenido del directorio con ProcessBuilder:

```
try {
                        ProcessBuilder pb = new ProcessBuilder("CMD","/C","DIR");
                        //ProcessBuilder pb = new ProcessBuilder("CMD /C DIR");

                            //redirect the output to a new file named output.txt
                            pb.redirectOutput(new File("output.txt"));

                            pb.start();//execute the command
```

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

BUSCAR INFO SOBRE CARACTERÏSTICAS DEL PROCESO pid y demás

2.3.1 Redirección de la E/S estándar

un subproceso no tiene terminal o consola en el que poder mostrar su información. Toda la E/S por defecto se redirige al proceso padre. Es el proceso padre el que puede usar estos streams para recoger o enviar información al proceso hijo.

getInputStream()

No sólo es importante recoger el valor de retorno de un comando, sino que muchas veces nos va a ser de mucha utilidad el poder obtener la información que el proceso genera por la salida estándar o por la salida de error.

Para esto vamos a utilizar el método public abstract InputStream getInputStream() de la clase Process para leer el stream de salida del proceso, es decir, para leer lo que el comando ejecutado (proceso hijo) ha enviado a la consola.

```
Process p = pbuilder.start();
BufferedReader processOutput =
  new BufferedReader(new InputStreamReader(p.getInputStream()));

String linea;
while ((linea = processOutput.readLine()) != null) {
  System.out.println("> " + linea);
}
processOutput.close();
```

getErrorStream()

Además de la salida estándar, también podemos obtener la salida de error (stderr) que genera el proceso hijo para procesarla desde el padre.

Si la salida de error ha sido previamente redirigida usando el método ProcessBuilder.redirectErrorStream(true) entonces la salida de error y la salida estándar llegan juntas con getInputStream() y no es necesario hacer un tratamiento adicional.

Si por el contrario queremos hacer un tratamiento diferenciado de los dos tipos de salida, podemos usar un esquema similar al usado anteriormente, con la salvedad de que ahora en vez de llamar a getInputStream() lo hacemos con getErrorStream()

```
Process p = pbuilder.start();
BufferedReader processError =
  new BufferedReader(new InputStreamReader(p.getErrorStream()));
// En este ejemplo, por ver una forma diferente de recoger la información,
// en vez de leer todas las líneas que llegan, recogemos la primera línea
// y suponemos que nos han enviado un entero.
int value = Integer.parseInt(processError.readLine());
processError.close();
```

Vamos a ver un ejemplo completo de uso de todas las funcionalidad anteriores

```
import java.io.*;
public class Ejercicio2 {
  public static void main(String[] args) {
    String comando = "notepad";
    ProcessBuilder pbuilder = new ProcessBuilder (comando);
    Process p = null;
    try {
      p = pbuilder.start();
      // 1- Procedemos a leer lo que devuelve el proceso hijo
      InputStream is = p.getInputStream();
      // 2- Lo convertimos en un InputStreamReader
      // De esta forma podemos leer caracteres en vez de bytes
      // El InputStreamReader nos permite gestionar diferentes codificaciones
      InputStreamReader isr = new InputStreamReader(is);
      // 2- Para mejorar el rendimiento hacemos un wrapper sobre un BufferedReader
      // De esta forma podemos leer enteros, cadenas o incluso líneas.
      BufferedReader br = new BufferedReader(isr);

      // A Continuación leemos todo como una cadena, línea a línea
      String linea;
      while ((linea = br.readLine()) != null)
        System.out.println(linea);
    } catch (Exception e) {
      System.out.println("Error en: "+comando);
```

```
        e.printStackTrace();
      } finally {
        // Para finalizar, cerramos los recursos abiertos
        br.close();
        isr.close();
        is.close();
      }
    }
}
```

getOutputStream()

No sólo podemos recoger la información que envía el proceso hijo sino que, además, también podemos enviar información desde el proceso padre al proceso hijo.

Igual que con las entradas que llegan desde el proceso hijo, podemos enviar la información usando directamente el OutputStream del proceso, pero lo haremos de nuevo con un Decorator.

En este caso, el wrapper de mayor nivel nivel para usar un OutputStream es la clase PrintWriter que nos ofrece métodos similares a los de System.out.printxxxxx para gestionar el flujo de comunicación con el proceso hijo.

```
PrintWriter toProcess = new PrintWriter(
  new BufferedWriter(
    new OutputStreamWriter(
      p.getOutputStream(), "UTF-8")), true);
toProcess.println("sent to child");
```

2.3.2 Redirección de las Entradas y Salidas Estándar

En un sistema real, probablemente necesitemos guardar los resultados de un proceso en un archivo de log o de errores para su posterior análisis. Afortunadamente lo podemos hacer sin modificar el código de nuestras aplicaciones usando los métodos que proporciona el API de ProcessBuilder para hacer exactamente eso.

Por defecto, tal y como ya hemos visto, los procesos hijos reciben la entrada a través de una tubería a la que podemos acceder usando el OutputStream que nos devuelve Process.getOutputStream().

Sin embargo, tal y como veremos a continuación, esa entrada estándar se puede cambiar y redirigirse a otros destinos como un fichero usando el método redirectOutput(File). Si modificamos la salida estándar, el método getOutputStream() devolverá ProcessBuilder.NullOutputStream.

`Redirección antes de ejecutar`

Es importante fijarse en qué momento se realiza cada acción sobre un proceso.

Antes hemos visto que los flujos de E/S se consultan y gestionan una vez que el proceso está en ejecución, por lo tanto los métodos que nos dan acceso a esos streams son métodos de la clase Process.

Si lo que queremos es redirigir la E/S, como vamos a ver a continuación, lo haremos mientras preparamos el proceso para ser ejecutado. De forma que cuando se lance sus streams de E/S se modifiquen. Por eso en esta ocasión los métodos que nos permiten redireccionar la E/S de los procesos son métodos de la clase ProcessBuilder.

Vamos a ver con un ejemplo cómo hacer un programa que muestre la versión de Java. Ahora bien, en esta ocasión la salida se va a guardar en un archivo de log en vez de enviarla al padre por la tubería de salida estándar:

```
ProcessBuilder processBuilder = new ProcessBuilder("java", "-version");

// La salida de error se enviará al mismo sitio que la estándar
processBuilder.redirectErrorStream(true);

File log = folder.newFile("java-version.log");
processBuilder.redirectOutput(log);

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

En el ejemplo anterior podemos observar como se crea un archivo temporal llamado java-version.log e indicamos a ProcessBuilder que la salida la redirija a este archivo.

Es lo mismo que si llamásemos a nuestra aplicación usando el operador de redirección de salida:

```
java ejemplo-java-version > java-version.log
```

`Código del proceso hijo`

Si el proceso hijo que lanzamos, en vez de ser un comando del sistema, es otra clase java, en ningún momento tenemos que modificar el código de este proceso para que funcione como hijo.

Por lo tanto, el proceso hijo seguirá haciendo

System.out.println("Versión de Java: " + System.getProperty("java.version"));

y será el sistema operativo el que se encargue de redirigir las salidas o entradas al fichero, o donde se haya configurado con los métodos de redirección de la clase ProcessBuilder.

Ahora vamos a fijarnos en una variación del ejemplo anterior. Lo que queremos hacer ahora es añadir (append to) información al archivo de log file en vez de sobrescribir el archivo cada vez que se ejecuta el proceso. Con sobrescribir nos referimos a crear el archivo vacío si no existe, o bien borrar el contenido del archivo si éste ya existe.

```
File log = tempFolder.newFile("java-version-append.log");
processBuilder.redirectErrorStream(true);
processBuilder.redirectOutput(Redirect.appendTo(log));
```

Otra vez más, es importante hacer notar la llamada a redirectErrorStream(true). En el caso de que se produzca algún error, se mezclarán con los mensajes de salida en el fichero..

En el APi de ProcessBuilder encontramos métodos para redireccionar también la salida de error estándar y la entrada estándar de los procesos.

- redirectError(File)
- redirectInput(File)

Para hacer las redirecciones también podemos utilizar la clase ProcessBuilder.Redirect como parámetro para las versiones sobrecargadas de los métodos anteriores, utilizando uno de los siguientes valores

Valor	Significado
Redirect.DISCARD	La información se descarta
Redirect.to(File)	La información se guardará en el fichero indicado. Si existe, se vacía.
Redirect.from(File)	La información se leerá del fichero indicado
Redirect.appendTo(File)	La información se añadirá en el fichero indicado. Si existe, no se vacía

Estos valores son campos estáticos de la clase Redirect y pueden ser usados como parámetros para los métodos sobrecargados redirectOutput, redirectError y redirectInput.

```
File log = folder.newFile("sampleInputData.csv");
processBuilder.redirectInput(Redirect.from(log));
```