PSP

Programación de Procesos

Unidad 2

Jesús Alberto Martínez
versión 0.1

Unidad 2. Programación de procesos

1 Introducción

Una vez hemos aprendido a diferenciar entre programas, procesos e hilos, en este segundo tema vamos a aprender cómo desde un programa creado por nosotros podemos lanzar otros programas, es decir, desde un proceso en ejecución, podemos crear otro proceso.

Además de lanzarlos, al establecerse una relación padre-hijo estos procesos pueden comunicarse entre sí intercambiando información. De esta forma nuestros programas podrán lanzar otras aplicaciones, comandos del SO e incluso otras aplicaciones nuestras, permitiendo cierto grado de sincronización y comunicación entre ellas.

Objetivos

Los objetivos que alcanzaremos tras esta unidad son:

- Conocer las clases de Java para la creación de procesos
- Monitorizar y controlar el ciclo de vida de un proceso
- Controlar la comunicación entre procesos padre/hijo
- Usar métodos para la sincronización entre procesos y subprocesos
- Entender el mecanismo de comunicación mediante tuberías (pipes)
- Aprender la sintaxis y uso del comando curl para probar API REST desde un programa
- Crear programas que ejecuten tareas en paralelo.

2 Creación rápida de procesos con Java con Runtime

2.1 Creación rápida de procesos

La clase java.lang.Runtime se usa principalmente para interactuar con el JRE de Java. Esta clase proporciona métodos para lanzar procesos, llamar al recolector de basura (Garbage Collector), saber la cantidad de memoria disponible y libre, etc.

> Especificación java.lang.Runtime

Cada aplicación en Java tiene acceso a una única instancia de *java.lang.Runtime* a través del método `Runtime.getRuntime()` que devuelve la instancia **singleton** de la clase Runtime.

> **Patrones de diseño: Singleton**
>
> ¿Qué son los patrones de diseño? ¿Qué es y para qué se usa el patrón de diseño singleton?
>
> Investiga cómo realizar una clase que siga el patrón de diseño singleton.
>
> Refactoring.Guru Patrones de diseño

El método que nos interesa a nosotros para la creación de procesos es

```
public Process exec(String command) throws IOException
```

Veamos un ejemplo sencillo de uso de este método

```
1  public static void main(String[] args) throws IOException {
2      // Launch notepad app
3      Runtime.getRuntime().exec("notepad.exe");
4
5      // This way always works
6      // String separator = System.getProperty("file.separator");
7      // Runtime.getRuntime()
8      //     .exec("c:"+separator+"windows"+separator+"notepad.exe");
9
10     // This way used to work (UNIX style paths)
11     // Runtime.getRuntime().exec("c:/windows/notepad.exe");
12 }
```

Se puede observar que en el parámetro que pasamos al método **exec** indicamos el programa que queremos ejecutar. En este caso, como el *notepad* se encuentra en el PATH del sistema, no es necesario indicar la ruta donde se encuentra el programa. En otro caso, sí tendríamos que hacerlo.

2.2 Propiedades del sistema y comandos del sistema

Si tenemos pensado desarrollar aplicaciones que funcionen en diferentes SO tendremos que enfrentarnos a la problemática del funcionamiento diferente de los distintos SO.

Vamos a ver algunos ejemplos que pueden servir como guía para otros problemas similares a los expuestos.

File separator

Para indicar las rutas en un sistema los sistemas UNIX emplean el carácter **/** como separador mientras que los sistemas Windows usan el carácter **** . En resumen, / en *X y \ en Windows.

¿Cómo podemos hacer entonces que nuestras aplicaciones sean independientes del SO en el que se ejecutan?

Para este tipo de cuestiones vamos a utilizar de forma recurrente las propiedades del sistema mediante `System.getProperty(String propName)`. Estas propiedades se configuran con el propio sistema operativo, aunque las podemos modificar usando los parámetros de ejecución de la máquina virtual

```
String separator = System.getProperty("file.separator");
```

o

```
-Dfile.separator
```

Aunque siempre es una buena práctica usar el carácter **/** en las rutas ya que Java es capaz de convertirlas al sistema en el que se ejecuta.

Si lo que queremos es ejecutar un comando del SO, tenemos que hacerlo, al igual que si lo hacemos manualmente, a través del shell del sistema, donde volvemos a encontrar la dicotomía entre sistemas UNIX y sistemas Windows.

Vamos a ver el código que, a través de las propiedades del sistema, nos permite obtener un listado de los archivos existentes en la carpeta personal del usuario.

```
1  // Primero obtenemos la carpeta del usuario
2  String homeDirectory = System.getProperty("user.home");
3  boolean isWindows = System.getProperty("os.name")
4    .toLowerCase().startsWith("windows");
5
6  if (isWindows) {
7      Runtime.getRuntime()
8        .exec(String.format("cmd.exe /c dir %s", homeDirectory));
9  } else {
10     Runtime.getRuntime()
11       .exec(String.format("sh -c ls %s", homeDirectory));
12 }
```

Modo shell no interactivo

Como se puede observar, tanto para Windows como UNIX se ha usado el modificador **c** del comando. Este modificador indica que se abra un shell, se ejecute el comando recibido y se cierre el proceso del shell.

A continuación podemos ver un ejemplo de respuesta ante la pulsación de un botón, en una app gráfica, para abrir una página en el navegador. Tenemos cómo se haría en sistemas *X y comentado una de las formas de hacerlo en Windows.

```
1  // Calling app example
2  public void mouseClicked(MouseEvent e) {
3    // Launch Page
4    try {
5      // Linux version
6      Runtime.getRuntime().exec("open http://localhost:8153/go");
7      // Windows version
8      // Runtime.getRuntime().exec("explorer http://localhost:8153/go");
9    } catch (IOException e1) {
10     // Don't care
11   }
12 }
```

System properties

Vamos a crear nuestro primer programa en Java, que no va a ser tan sencillo como pueda parecer

Usando métodos de las clases System y Runtime hacer un programa que muestre

- todas las propiedades establecidas en el sistema operativo y sus valores.

- memoria total, memoria libre, memoria en uso y los procesadores disponibles

Mira los métodos que proporcionan las clases Runtime y System. Intenta obtener una lista u otra estructura de datos que te permita recorrer las propiedades para ir mostrando sus nombres y valores. (Solución en los Anexos)

3 Gestión de procesos en Java - ProcessBuilder y Process

3.1 Preparación y configuración de un proceso

En el paquete `java.lang` tenemos dos clases para la gestión de procesos.

- `java.lang.ProcessBuilder` Referencia API Java
- `java.lang.Process` Referencia API Java

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.

```
1  ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
2  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.

Argumentos y parámetros

Si queremos lanzar un programa con parámetros (modificadores que hacen que cambie la forma de funcionar un programa como -h /s ...) el comando no puede ser pasado al constructor directamente como un único string, debe ser preprocesado para convertirlo en una lista y que funcione.

```
1  // Formas diferentes de pasar el comando a los constructores de ProcessBuilder
2  // 1ª forma: usando una cadena. Falla con parámetros
3  // Sólo funciona con programas que tengan argumentos
4  String command1 = "notepad.exe prueba1.txt"
5  ProcessBuilder pb = new ProcessBuilder(command1);
6
7  // 2ª forma: usando un array de cadenas. Funciona con parámetros
8  String[] command2 = {"cmd", "/c", "dir", "/o"};
9  ProcessBuilder pb = new ProcessBuilder(command2);
10
11 // 3ª forma: usando una cadena y dividiéndola para convertirla en una lista
12 String command3 =  "c:/windows/system32/shutdown -s -t 0";
13 // La expresión regular \s significa partir por los espacios en blanco
14 ProcessBuilder pb = new ProcessBuilder(command3.split("\\s"));
15 // 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.

Referencia del comando shutdown de Windows

Ejercicio psp.actividades.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.

Modificar el comando en tiempo de ejecución

Puede ser que todo el comando, o parte del mismo, no lo tengamos en el momento de llamar a los constructores de ProcessBuilder. Se puede cambiar, modificar y consultar a posteriori con el método command

Al igual que con los constructores, tenemos dos versiones del método command

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

y la tercera forma de este método (sin parámetros) sirve para obtener una lista del comando pasado al constructor o puesto con alguna de las formas anterior del método command. Lo interesante es que una vez que tenemos la lista, podemos modificarla usando los métodos de la clase List.

En el siguiente ejemplo, en el momento de definir el comando, nos falta saber la última parte, el directorio temporal. Además, si queremos hacer que la ejecución sea multiplataforma, el shell a ejecutar tampoco lo sabemos. Dependiendo del SO se añaden dos valores al principio y un valor al final, con el método add de la clase List.

```
1  // Sets and modifies the command after ProcessBuilder object is created
2  String command = "java -jar install.jar -install"; // tmp dir is missing
3  ProcessBuilder pbuilder = new ProcessBuilder(command.split("\\s"));
4  if (isWindows) {
5      pbuilder.command().add(0, "cmd"); // Sets the 1st element
6      pbuilder.command().add(1, "/c"); // Sets the 2nd element
7      pbuilder.command().add("c:/temp"); // Sets the last element
8      // Command to run cmd /c java -jar install.jar -install c:/temp
9  } else {
```

```
10      pbuilder.command().add(0, "sh"); // Sets the 1st element
11      pbuilder.command().add(1, "-c"); // Sets the 2nd element
12      pbuilder.command().add("/tmp"); // Sets the last element
13      // Command to run: sh -c java -jar install.jar -install /tmp
14  }
15
16  // Starts the process
17  pbuilder.start();
```

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.

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

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

```
1  // Retrieve and modify the process environment
2  Map<String, String> environment = pbuilder.environment();
3  // Get the PATH environment variable and add a new directory
4  String systemPath = environment.get("path") + ";c:/users/public";
5  environment.replace("path", systemPath);
6  // Add a new environment variable and use it as a part of the command
7  environment.put("GREETING", "Hola Mundo");
8  processBuilder.command("/bin/bash", "-c", "echo $GREETING");
```

```
 1  // Indicamos el directorio donde se encuentra el ejecutable
 2  File directorio = new File ("bin");
 3  pb.directory(directorio);
 4
 5  // Mostramos la información de las variables de entorno
 6  Map variablesEntorno = pb.environment();
 7  System.out.println(variablesEntorno);
 8
 9  // Mostramos el nombre del proceso y sus argumentos
10  List command = pb.command();
```

```
11  Iterator iter = l.iterator();
12  while (iter.hasNext()) {
13      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.actividades.U2A2_DirectorioTrabajo`

Crea un nuevo proyecto Java (configura como nombre del proyecto U2A2_DirectorioTrabajo y como clase principal psp.actividades.U2A2_DirectorioTrabajo) 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.

3.2 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.

Especificación java.lang.Process

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.

```
 1  public class ProcessDemo {
 2
 3      public static void main(String[] args) throws Exception {
 4
 5          ProcessBuilder pb = new ProcessBuilder("C:\Program Files\Mozilla Firefox\
firefox.exe");
 6          // Effectively launch the process
 7          Process p = pb.start();
 8          // Check is process is alive or not
 9          boolean alive = p.isAlive();
10          // Wait for the process to end for 10 seconds.
11          if (p.waitFor(10, TimeUnit.SECONDS)) {
12              System.out.println("Process has finished");
13          } else {
14              System.out.println("Timeout. Process hasn't finished");
15          }
```

```
16          // Force process termination.
17          p.destroy();
18          // Check again if process remains alive
19          boolean alive = p.isAlive();
20          // Get the process exit value
21          int status = p.exitValue();
22      }
23  }
```

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.

Actividad psp.actividades.U2A3_ValorSalida

Crea un nuevo proyecto Java (configura como nombre del proyecto U2A3_ValorSalida y como clase principal psp.actividades.U2A3_ValorSalida). Prepara un programa que ejecute varios comandos (notepad, calc, comandos shell) uno detrás de otro, de forma secuencial y haz que tu programa obtenga el código de finalización de cada uno de ellos. Para cada programa imprime el nombre y su código de finalización.

Prueba a poner aplicaciones que no existan o comandos con parámetros incorrectos.

¿Qué hace el entorno de programación si pones System.exit(10) para acabar tu programa?. Fíjate en la consola. ¿Qué hace el entorno de programación si pones System.exit(0) para acabar tu programa.? ¿Cuál es entonces el valor por defecto?

Lanzar una clase Java como proceso desde otra clase java en el mismo proyecto

Para las actividades os pediré que programéis tanto el proceso padre como el proceso hijo. Para hacer eso, una de las clases tendrá que lanzar a la otra.

Para hacer esto, ambas clases deben tener un método main. Así que en las propiedades del proyecto deberemos seleccionar cuál de las clases se ejecutará primero, normalmente la clase `Lanzador` o `Launcher` (proceso padre).

Antes de que una clase pueda lanzar a otra, al menos la segunda (proceso hijo) debe estar compilada, es decir, el archivo .class debe haberse creado dentro del directorio build/classes.

Entonces y sólo entonces, previa configuración de los parámetros de ejecución del proceso tal y como se muestra en el ejemplo, se podrá ejecutar una clase desde otra.

Directorio de trabajo

Si nos fijamos en la estructura de directorios de un proyecto y entendemos cómo se invocan unas clases a otras, entenderemos porqué el directorio de trabajo debe ser *build/classes*.

En Java, cuando se ejecuta una clase `java paquete.clase`, la MV Java espera encontrar los directorios que forman el paquete (psp/u2/actividad10) a partir del punto donde se está invocando el comando. Por eso, para que pueda encontrar la clase (el archivo .class), debemos ubicarnos previamente en el directorio `build/classes`

```
1  // Prepare the environment and the command
2  ProcessBuilder pb = new ProcessBuilder("java", "psp.u2.actividad10.Sumador");
3  pb.directory(new File("build/classes"));
4  Process p = pb.start();
```

Actividad psp.actividades.U2A4_Lanzador

Crea un nuevo proyecto Java (configura como nombre del proyecto **U2A4_Lanzador** y como clase principal psp.activities.U2A4_Lanzador).

Ahora, en el mismo proyecto y dentro del mismo paquete crea otra clase, **U2A4_Ejecutor**, con un método main que recibirá el nombre del programa que debe ejecutar como parámetro del método main(args). Haz que esta aplicación cree un nuevo proceso para ejecutar el programa recibido como parámetro.

La clase terminará devolviendo como código de finalización el que el programa lanzado le haya devuelto a ella.

Método System.exit()

- Cero. El código cero debe devolverse cuando la ejecución del proceso haya ido bien, esto es, que ha terminado su ejecución sin problemas.

- Distinto de cero. Un código distinto de cero indica una terminación con errores. Java nos permite usar códigos diferentes para los diferentes tipos de error.

Por último, podemos hacer que U2A4_Lanzador pregunte al usuario qué aplicación quiere ejecutar y pasársela a la clase U2A4_Ejecutor.

En Lanzador recoge el código de finalización de Ejecutor y muestra un mensaje indicando si el proceso terminó bien o con errores.

4 Gestión de la E-S de un proceso

4.1 Redirección de la E/S estándar

Ya hemos comentado que un subproceso no tiene terminal o consola en el que poder mostrar su información. Toda la E/S por defecto ((stdin - teclado -, stdout y stderr - pantalla-) 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.

Código del proceso hijo

En ningún momento, cuando estamos programando un proceso, debemos pensar si va a ser lanzado como padre o como hijo.

Es más, todos los programas que hacemos son lanzados como hijos por el IDE y eso no hace que cambiemos nuestra forma de programarlos.

Un proceso que vayamos a lanzar como hijo debería funcionar perfectamente como proceso independiente y puede ser ejecutado directamente sin tener que hacerle ningún tipo de cambio.

Este intercambio de información nos da mucha flexibilidad y proporciona una forma de control y comunicación sobre el proceso hijo.

La E/S en el SO y las tuberías

La E/S en sistemas Linux, como casi todo lo demás, es tratada como un fichero.

Dentro de cada proceso, cuando se accede a un fichero, se le asigna un identificador único. Hay tres identificadores que se crean y se abren con la creación del proceso, y que además siempre tienen el mismo identificador:

- 0: stdin
- 1: stdout
- 2: stderr

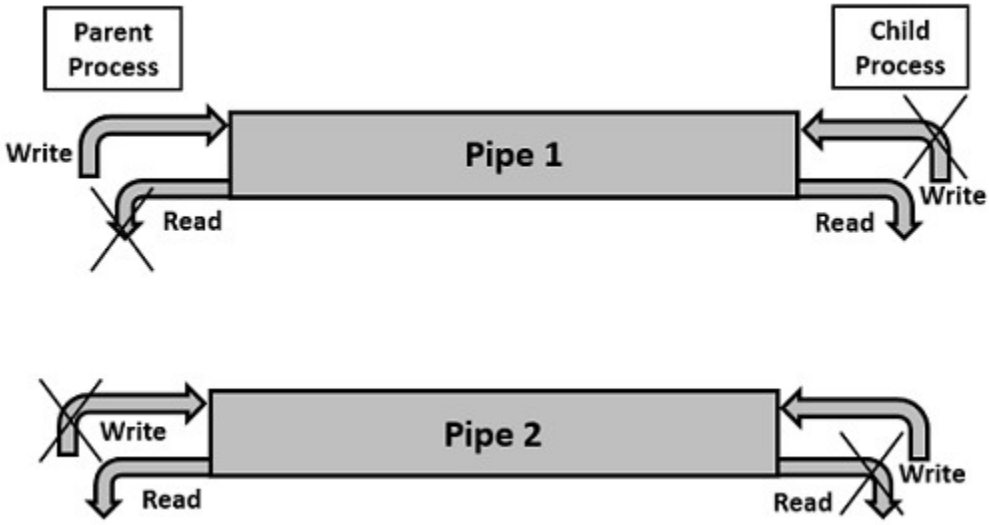
Estos ***descriptores de fichero*** permiten gestionar sus streams asociados de diferentes formas. Podemos redirigir la salida de un proceso (stdout) a un archivo y seguir viendo los mensajes de error (stderr) en la consola, o bien podemos hacer que la entrada de datos a un programa se lea desde un fichero en vez del teclado, lo que permitiría automatizar pruebas, por ejemplo.

Estos son algunos ejemplos de cómo se hacen estas redirecciones a nivel de So:

```
# Redirecciona la salida de ls a un archivo
ls > capture.txt
# Redirecciona la salida de ls a la entrada de cat (doble redirección)
# Esto es una tubería
ls | cat
# Cambia la salida de program.sh a capture.txt y los errores a error.txt
./program.sh 1> capture.txt 2> error.txt
# Redirige la salida y los errores de program.sh al mismo archivo, capture.txt
./program.sh > capture.txt 2>&1
# Cambia la entrada de program.sh al contenido de dummy.txt
./program.sh < dummy.txt
# Redirige la salida del primer comando y la pone como entrada del segundo
# Esto es una tubería
cat dummy.txt | ./program.sh
```

Redirecciones E/S en Linux

En la relación padre-hijo que se crea entre procesos los descriptores también se redirigen desde el hijo hacia el padre, usando 3 tuberías (pipes), una por cada stream de E/S por defecto. Esas tuberías pueden usarse de forma similar a cómo se hace en los sistemas Linux.

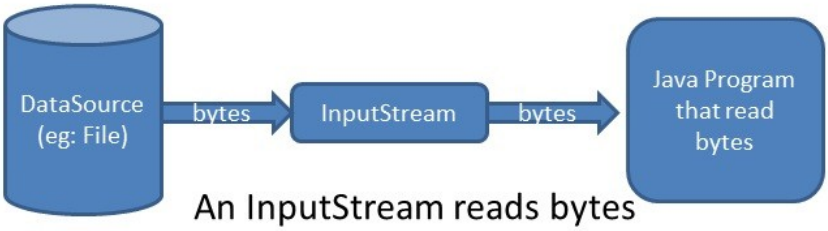


getInputStream()

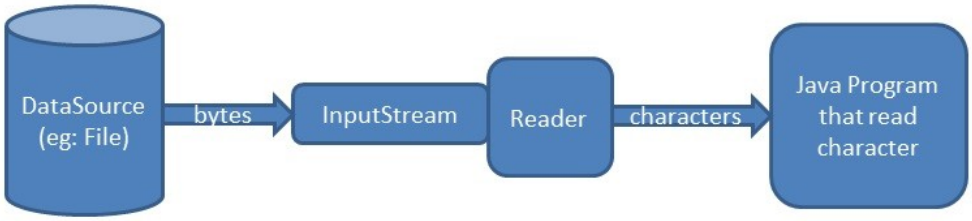
No sólo es importante recoger el valor de retorno de un comando, sino que muchas veces nos va a ser de mucha utilidad el poder obtener la información que el proceso genera por la salida estándar o por la salida de error.

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.

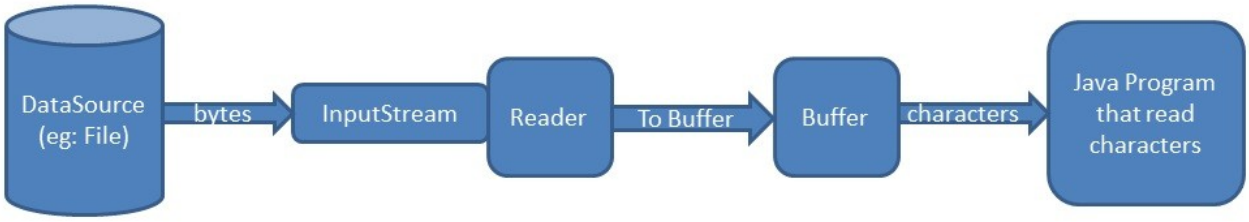
```
1  Process p = pbuilder.start();
2  BufferedReader processOutput =
3      new BufferedReader(new InputStreamReader(p.getInputStream()));
4
5  String linea;
6  while ((linea = processOutput.readLine()) != null) {
7      System.out.println("> " + linea);
8  }
9  processOutput.close();
```



An InputStream reads bytes



A Reader works with an InputStream and reads characters



A BufferedReader pre-fetches characters in to a buffer for performance

Charsets y encodings

Desde el inicio de la informática los juegos de caracteres y las codificaciones han supuesto un auténtico quebradero de cabeza para los programadores, especialmente cuando trabajamos con juegos de caracteres no anglosajones. Pues bien, la consola de Windows no iba a ser una excepción.

La consola de windows, conocida como *"DOS prompt"* o *cmd*, es la forma de ejecutar programas y comandos DOS en Windows, por lo tanto esos programas mantienen la codificación de DOS. A Microsoft no le gustan hacer cambios que pierdan la compatibilidad hacia atrás, es decir, que sean compatibles con versiones anteriores, así que cuando hagamos una aplicación que trabaje con la consola debemos tener en cuenta esta circunstancia.

En Wikipedia comentan que la codificación **CP850** teóricamente ha sido ampliamente reemplazada por **Windows-1252** y posteriormente Unicode, pero aún así **CP850** sigue presente en la consola de comandos.

Por lo tanto, si queremos mostrar información de la consola en nuestras aplicaciones, debemos trabajar con el charset adecuado, a saber, CP-850.

Para usar un encoding concreto, la clase InputStreamReader, que pasa de gestionar bytes a caracteres, tiene un constructor que permite especificar el tipo de codificación usado en el stream de bytes que recibimos, así que que debemos usar este constructor cuando trabajemos con aplicaciones de consola.

```
new InputStreamReader(p.getInputStream(), "CP850");
```

getErrorStream()

Curiosamente, o no tanto, 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 salid, podemos usar un schema similar al usado anteriormente, con la salvedad de que ahora en vez de llamar a `getInputStream()` lo hacemos con `getErrorStream()`.

```
1  Process p = pbuilder.start();
2  BufferedReader processError =
3      new BufferedReader(new InputStreamReader(p.getErrorStream()));
4  // En este ejemplo, por ver una forma diferente de recoger la información,
```

```
5  // en vez de leer todas las líneas que llegan, recogemos la primera línea
6  // y suponemos que nos han enviado un entero.
7  int value = Integer.parseInt(processError.readLine());
8  processError.close();
```

Patrón de diseño Decorator o Wrapper

En ambos tipos de streams de entrada (input y error) estamos recogiendo la información de un objeto de tipo BufferedReader. Podríamos usar directamente el InputStream que nos devuelven los métodos de Process, pero tendríamos que encargarnos nosotros de convertir los bytes a caracteres, de leer el stream caracter a caracter y de controlar el flujo al no disponer de un buffer.

Todo esto nos lo podemos ahorrar usando clases que gestionan el flujo a un nivel de concreción más alto, usando sin llegar a ser conscientes otro patrón de diseño bastante común, **Patrón de diseño Decorator** también llamado **wrapper o envoltorio**.

Decorator es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Refactoring.Guru patrones de diseño

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

```
 1  import java.io.*;
 2  public class Ejercicio2 {
 3      public static void main(String[] args) {
 4          String comando = "notepad";
 5          ProcessBuilder pbuilder = new ProcessBuilder (comando);
 6          Process p = null;
 7          try {
 8              p = pbuilder.start();
 9              // 1- Procedemos a leer lo que devuelve el proceso hijo
10              InputStream is = p.getInputStream();
11              // 2- Lo convertimos en un InputStreamReader
12              // De esta forma podemos leer caracteres en vez de bytes
13              // El InputStreamReader nos permite gestionar diferentes
codificaciones
14              InputStreamReader isr = new InputStreamReader(is);
15              // 2- Para mejorar el rendimiento hacemos un wrapper sobre un
BufferedReader
16              // De esta forma podemos leer enteros, cadenas o incluso líneas.
17              BufferedReader br = new BufferedReader(isr);
18
19              // A Continuación leemos todo como una cadena, línea a línea
20              String linea;
21              while ((linea = br.readLine()) != null)
22                  System.out.println(linea);
23          } catch (Exception e) {
24              System.out.println("Error en: "+comando);
25              e.printStackTrace();
```

```
26          } finally {
27              // Para finalizar, cerramos los recursos abiertos
28              br.close();
29              isr.close();
30              is.close();
31          }
32      }
33  }
```

`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, usando el último de los tres streams que nos queda, el `stdin`.

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.

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

`Heredar la E/S del proceso padre`

Con el método `inheritIO()` podemos redireccionar todos los flujos de E/S del proceso hijo a la E/S estándar del proceso padre.

```
1  ProcessBuilder processBuilder = new ProcessBuilder("/bin/sh", "-c", "echo hello");
2
3  processBuilder.inheritIO();
4  Process process = processBuilder.start();
5
6  int exitCode = process.waitFor();
```

En el ejemplo anterior, tras invocar al método inheritIO() podemos ver la salida del comando ejecutado en la consola del proceso padre dentro del IDE.

Pipelines

Java 9 introdujo el concepto de `pipelines` en el API de ProcessBuilder:

```
public static List<Process> startPipeline(List<ProcessBuilder> builders)
```

El método **`startPipeline`** usa un lista de objetos ProcessBuilder. Este método estático se encarga de lanzar un proceso para cada uno de los ProcessBuilder recibidos. Y lo que automatiza es la creación de tuberías encadenadas (pipeline) haciendo que la salida de cada proceso esté enlazada con la entrada del siguiente..

Por ejemplo, si queremos realizar este tipo de operaciones tan comunes en shellscript:

```
find . -name *.java -type f | wc -l
```

Lo que haremos será crear un ProcessBuilder para cada uno de los comandos, y pasárselos todos al método startPipeline para que los ejecute y los encadene.

```
1  List builders = Arrays.asList(
2      new ProcessBuilder("find", "src", "-name", "*.java", "-type", "f"),
3      new ProcessBuilder("wc", "-l"));
4
5  List processes = ProcessBuilder.startPipeline(builders);
6  Process last = processes.get(processes.size() - 1);
7
8  // Desde el proceso padre podemos recoger la salida del último proceso para
9  // el resultado final del pipeline
```

El ejemplo anterior busca todos los archivos .java dentro del directorio src, encadena la salida hacia el comando wc para contar cuantos ficheros ha encontrado, siendo este el resultado final del pipeline.

4.2 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:

```
1  ProcessBuilder processBuilder = new ProcessBuilder("java", "-version");
2
3  // La salida de error se enviará al mismo sitio que la estándar
4  processBuilder.redirectErrorStream(true);
5
6  File log = folder.newFile("java-version.log");
7  processBuilder.redirectOutput(log);
8
9  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
```

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 sobreescribir el archivo cada vez que se ejecuta el proceso. Con sobreeescribir nos referimos a crear el archivo vacío si no existe, o bien borrar el contenido del archivo si éste ya existe.

```
1  File log = tempFolder.newFile("java-version-append.log");
2  processBuilder.redirectErrorStream(true);
```

```
3  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 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

4.3 Información de los procesos en Java

Una vez que un proceso está en ejecución podemos obtener información acerca de ese proceso usando los métodos de la clase `java.lang.ProcessHandle.Info`:

- El comando usado para lanzar el proceso
- Los argumentos/parámetros que recibió el proceso
- El instante de tiempo en el que se inició el proceso
- El tiempo de CPU que ha usado el proceso y el usuario que lo ha lanzado

Aquí tenemos un sencillo ejemplo de cómo hacerlo

```
1  ProcessHandle processHandle = ProcessHandle.current();
2  ProcessHandle.Info processInfo = processHandle.info();
3
4  System.out.println("PID: " + processHandle.pid());
5  System.out.println("Arguments: " + processInfo.arguments());
6  System.out.println("Command: " + processInfo.command());
7  System.out.println("Instant: " + processInfo.startInstant());
8  System.out.println("Total CPU duration: " + processInfo.totalCpuDuration());
9  System.out.println("User: " + processInfo.user());
```

En el ejemplo anterior hemos obtenido la información del proceso actual (`ProcessHandle.current()`), así que si estamos en el proceso padre, sólo podemos mostrar su información, pero no la de su hijo.

También es posible acceder a la información de un proceso lanzado (proceso hijo). En este caso necesitamos la instancia de `java.lang.Process` para llamar a su método `toHandle()` y obtener la instancia de `java.lang.ProcessHandle` del proceso hijo..

```
1  Process process = processBuilder.inheritIO().start();
2  ProcessHandle childProcessHandle = process.toHandle();
3  ProcessHandle.Info child processInfo = processHandle.info();
```

A partir de ahí, el código para acceder a la información y detalles del proceso hijo es idéntico al ejemplo anterior.

5 Anexo I - Soluciones

5.1 System properties

```
long freeMemory = Runtime.getRuntime().freeMemory();
long availableMemory = Runtime.getRuntime().totalMemory();
long usedMemory = availableMemory - freeMemory;

/*** Runtime.getRuntime() usage ***/
// Show system information
// Memory will be shown in MBytes formatted with 2-decimal places
DecimalFormat megabytes = new DecimalFormat("#.00");
System.out.println("Available memory in JVM(Mbytes): " +
        megabytes.format((double)availableMemory/(1024*1024)));
System.out.println("Free memory in JVM(Mbytes): " +
        megabytes.format((double)freeMemory/(1024*1024)));
System.out.println("Used memory in JVM(Mbytes): " +
        megabytes.format((double)usedMemory/(1024*1024)));

System.out.println ("Processors in the system: "
        + Runtime.getRuntime().availableProcessors());

/*** System.getProperties() usage ***/
// Show each pair of property:value from System properties

// 1st. As a lambda expression using anonymous classes
System.getProperties().forEach((k,v) -> System.out.println(k + " => " + v));

// 2nd. As a Map.entrySet
for (Map.Entry<Object, Object> entry : System.getProperties().entrySet()) {
    Object key = entry.getKey();
    Object val = entry.getValue();
    System.out.println("> " + key + " => " + val);
}

// 3rd. As a Map.keySet
for (Object key : System.getProperties().keySet().toArray())
{
    System.out.println(">> " + key+":"+System.getProperty(key.toString()));
}

// Other methods found by students, based on a Properties object methods.
Properties prop = System.getProperties();
for (String propName: prop.stringPropertyNames()) {
  System.out.println(propName +  ":" + System.getProperty(propName));
}

// Or directly to the console using
prop.list(System.out);
```

Formato numérico

Todos los lenguajes de programación tienen varias formas de mostrar la información al usuario. Cuando se trata de mostrar información a a través de la consola, tenemos un par de alternativas para formatear la información numérica.

NumberFormat

Si usamos la clase NumberFormat o cualquiera de sus descendientes podemos controlar con bastante precisión cómo se verán los números, usando patrones.

```
1  DecimalFormat numberFormat = new DecimalFormat("#.00");
2  // Si usamos hashes en vez de ceros permitimos que .30 se vea como 0.3
3  // (los dígitos adicionales son opcionales)
4  System.out.println(numberFormat.format(number));
```

System.out.printf

Heredado de la sintaxis de la función printf de C, podemos utilizar la sintaxis de java.util.Formatter para configurar cómo será visualizada la información.

```
1  System.out.printf("\n$%10.2f",shippingCost);
2  // % rellena con hasta 10 posiciones los números
3  // para justificarlos a la derchan.
4  System.out.printf("%n$%.2f",shippingCost);
```

Usando colores en la salida por consola

Hay una forma sencilla de mostrar información por consola usando diferentes colores. Os dejo un ejemplo de código con la definición de algunos colores y la forma de usarlos.

```
1  public class UsarColoresEnConsola {
2
3  public static final String ANSI_RESET = "\u001B[0m";
4  public static final String ANSI_BLACK = "\u001B[30m";
5  public static final String ANSI_RED = "\u001B[31m";
6  public static final String ANSI_GREEN = "\u001B[32m";
7  public static final String ANSI_YELLOW = "\u001B[33m";
8  public static final String ANSI_BLUE = "\u001B[34m";
9  public static final String ANSI_PURPLE = "\u001B[35m";
10 public static final String ANSI_CYAN = "\u001B[36m";
11 public static final String ANSI_WHITE = "\u001B[37m";
12
13 public static final String ANSI_BLACK_BACKGROUND = "\u001B[40m";
14 public static final String ANSI_RED_BACKGROUND = "\u001B[41m";
15 public static final String ANSI_GREEN_BACKGROUND = "\u001B[42m";
16 public static final String ANSI_YELLOW_BACKGROUND = "\u001B[43m";
17 public static final String ANSI_BLUE_BACKGROUND = "\u001B[44m";
18 public static final String ANSI_PURPLE_BACKGROUND = "\u001B[45m";
19 public static final String ANSI_CYAN_BACKGROUND = "\u001B[46m";
20 public static final String ANSI_WHITE_BACKGROUND = "\u001B[47m";
21
22     public static void main(String[] args) {
23               System.out.println(ANSI_GREEN + ANSI_WHITE_BACKGROUND + "Hola" +
ANSI_BLUE + ANSI_YELLOW_BACKGROUND + " Adiós" + ANSI_RESET);
24     }
25 }
```

5.2 U2A1_Shutdowner

```
1  public class U2A1_Shutdowner {
2
3      public static void main(String[] args) throws IOException {
4          // Ask for the required information to prepare the command
5          Scanner keyboard = new Scanner(System.in);
6
```

```
 7          System.out.print("Select your option (s-shutdown / r-reboot / h-
hibernate): ");
 8          String shutdownOption = keyboard.nextLine();
 9
10          System.out.print("How much seconds will the command wait to be run? (0
means immediately): ");
11          String shutdownTime = keyboard.nextLine();
12
13          // Prepare the command
14          String command;
15          if (System.getProperty("os.name").toLowerCase().startsWith("windows")) {
16              command = "C:/Windows/System32/shutdown -" + shutdownOption + " -t " +
shutdownTime;
17          } else {
18              command = "shutdown -" + shutdownOption + " -t " + shutdownTime;
19          }
20
21          // Prepare the process and launch it
22          ProcessBuilder shutdowner = new ProcessBuilder(command.split("\\s"));
23          //shutdowner.start();
24
25          // Show the command to be run
26          System.out.print("El comando a ejecutar es:  ");
27          for (String commandPart: shutdowner.command()) {
28              System.out.print(commandPart + " ");
29          }
30          System.out.println("");
31      }
32  }
```

5.3 U2A2_DirectorioTrabajo

```
 1  public class U2A2_WorkingDirectory {
 2
 3      public static void main(String[] args) throws IOException {
 4          // Prepare the command
 5          String command;
 6          if (System.getProperty("os.name").toLowerCase().startsWith("windows")) {
 7              command = "cmd /c dir";
 8          } else {
 9              command = "sh -c ls";
10          }
11
12          // Prepare the process and launch it
13          ProcessBuilder commander = new ProcessBuilder(command.split("\\s"));
14
15          //1st - Default working directory
16          System.out.println("Working directory: " + commander.directory());
17          System.out.println("user.dir variable: " +
System.getProperty("user.dir"));
18
19          //2nd - Set the user.dir
20          System.setProperty("user.dir", System.getProperty("user.home"));
21          System.out.println("Working directory: " + commander.directory());
22          System.out.println("user.dir variable: " +
System.getProperty("user.dir"));
```

```
23
24          // 3rd - Change the working directory
25          commander.directory(new File(System.getProperty("user.home")));
26          System.out.println("Working directory: " +
commander.directory().toString());
27          System.out.println("user.dir variable: " +
System.getProperty("user.dir"));
28
29          commander.start();
30      }
31  }
```

5.4 U2A3_ValorSalida

```
 1  public class U2A3_ValorSalida {
 2
 3      public static void main(String[] args) {
 4          do {
 5              // Código para pedir un programa/comando a ejecutar
 6              Scanner teclado = new Scanner(System.in);
 7              System.out.println("Introduce el programa / comando que quieres
ejecutar (intro para acabar): ");
 8              String comando = teclado.nextLine();
 9
10              if (comando.equals("")) System.exit(0);
11
12              try {
13                  // Preparamos el entrono de ejecución del proceso
14                  // Como no sabemos el contenido del comando, forzamos su
conversión
15                  // a una lista para que no haya problemas con su ejecución
16                  ProcessBuilder pb = new ProcessBuilder(comando.split("\\s"));
17
18                  // Lanzamos el proceso hijo
19                  Process p = pb.start();
20
21                  // Esperamos a que acabe para recoger el valor de salida
22                  int exitValue = p.waitFor();
23
24                  if (exitValue == 0) {
25                      System.out.println("El comando " + pb.command().toString() + "
ha finalizado bien");
26                  } else {
27                      System.out.println("El comando " + pb.command().toString() + "
ha finalizado con errores. Código (" + exitValue + ")");
28                  }
29
30              } catch (InterruptedException | IOException ex) {
31                  System.err.println(ex.getLocalizedMessage());
32                  ex.printStackTrace();
33              }
```

```
34          } while (true);
35      }
36  }
```

5.5 U2A4_Lanzador U2A4_Ejecutor

```
 1  public class U2A4_Lanzador {
 2
 3      public static void main(String[] args) {
 4
 5          // Código para pedir un programa/comando a ejecutar
 6          Scanner teclado = new Scanner(System.in);
 7          System.out.println("Introduce el programa / comando que quieres ejecutar:
");
 8          String comando = teclado.nextLine();
 9
10          try {
11              // Preparamos el entrono de ejecución del proceso
12              // Como no sabemos el contenido del comando, forzamos su conversión
13              // a una lista para que no haya problemas con su ejecución
14              comando = "java psp.actividades.U2A4_Ejecutor " + comando;
15              ProcessBuilder pb = new ProcessBuilder(comando.split("\\s"));
16              pb.directory(new File("build/classes"));
17
18              // Lanzamos el proceso hijo
19              Process p = pb.start();
20
21              // Esperamos a que acabe para recoger el valor de salida
22              int exitValue = p.waitFor();
23
24              if (exitValue == 0) {
25                  System.out.println("El comando " + pb.command().toString() + " ha
finalizado bien");
26              } else {
27                  System.out.println("El comando " + pb.command().toString() + " ha
finalizado con errores. Código (" + exitValue + ")");
28              }
29          } catch (InterruptedException | IOException ex) {
30              System.err.println(ex.getLocalizedMessage());
31              ex.printStackTrace();
32          }
33      }
34  }
35
36  public class U2A4_Ejecutor {
37
38      public static void main(String[] args) throws Exception {
39          // Lectura de información desde los parámetros de entrada
40          // Se supone que recibimos: args[0] args[1] args[2] .....
args[args.length-1 --> comando a ejecutar
41          String comando = "";
42          for (int i = 0; i < args.length; i++) {
43              comando += args[i] + " ";
44          }
45          comando.trim();
```

```
46
47          ProcessBuilder pb = new ProcessBuilder(comando.split("\\s"));
48
49          // Lanzamos el proceso hijo
50          Process p = pb.start();
51
52          // Esperamos a que acabe para recoger el valor de salida
53          int exitValue = p.waitFor();
54
55          System.exit(exitValue);
56      }
57  }
```

Programación de clases "hijas"

Debemos tener en cuenta que una clase se puede ejecutar de forma independiente o puede ser llamada desde otro proceso.

Por eso, el código de las clases, hijas o padres, se hace sin tener en cuenta cómo van a ser llamadas. Debe ser **independiente** tal y como lo son unos procesos de otros.

Así, en el anterior proyecto yo podría ejecutar la clase psp.actividades.U2A4_Ejecutor de forma independiente, pasándole el nombre de los programas.

Puedo hacerlo invocándola desde línea de comandos o desde el entorno de desarrollo:

- java psp.actividades.U2A4_Ejecutor programa1 programa2 programa3
- En las propiedades del proyecto, seleccionando U2A4_Ejecutor como clase principal y poniendo como argumentos: programa1 programa2 programa3

6 Anexo II - Curl

Curl está diseñado para funcionar como una forma de verificar la conectividad a las URL y como una gran herramienta para transferir datos. Tiene especial relevancia e interés cunado se trata de usar servicios basados en API REST, tanto para comprobar su funcionamiento durante la fase de pruebas o bien para cuando el sistema ya está en producción.

Curl es una herramienta de línea de comandos que nos permite hacer peticiones HTTP desde la consola. Su principal uso es obtener una respuesta de un sitio web (pro ejemplo, para saber que no está caído) o para comprobar tiempos de respuesta.

> **curl means ...**
>
> La herramienta fué pensada para subir y descargar información usando una URL. Es una aplicación cliente (de ahí la 'c'), y a su vez es un cliente de URLs. Por lo tanto, 'c' de cliente y URL: cURL.
>
> En inglés "curl" se pronuncia con un sonido inicial /k/, rimando con la pronunciación de la palabra girl.
>
> Por el contrario, si lo deletreamos como c-URL, entonces us significado también adquiere un sentido lógico, ver-URL (see-URL), lo cual también es una buena definición de la utilidad de esta herramienta.

Curl funciona con protocolos que permiten la transferencia de datos en una o dos direcciones. Soporta protocolos basados en una "URI" y que estén descritos en una RFC, por lo que curl funciona principalmente con URLs (URIs en realidad) como el origen y/o destino de las transferencias e intercambios de información que realiza.

Actualmente curl ofrece soporte para los siguiente protocolos:

> DICT, FILE, FTP, FTPS, GOPHER, GOPHERS, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS,
>
> MQTT, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET, TFTP.

6.1 Cómo obtener curl

curl es una utilidad libre, abierta y disponible en varios formatos. Podemos descargarla y configurarla en la mayoría de los sistemas operativos y arquitecturas hardware. Algunos SO ya la incluyen en sus distribuciones.

La fuente principal de información y descarga siempre debería ser el sitio oficial de CURL donde podemos descargar, entre otros, los instaladores para nuestros sistemas.

- Linux (Ubuntu / Debian). curl viene instalado por defecto. De todas formas, podemos actualizarlo e instalarlo usando el gesto de paquetes APT

```
 apt install curl
```

- Windows 10 viene con la herramienta curl instalada en sus sistemas desde la versión 1804

```
podemos descargar e instalar la última versión oficial de curl para windows
desde curl windows binaries
```

- MacOS también trae de serie la herramienta curl desde hace ya bastantes años. Si necesitamos actualizar la versión que tenemos en el sistema, se recomienda instalar homebrew (un gesto de paquetes para macOS)

```
   brew install curl
```

6.2 Realizando una petición GET

La sintaxis básica de Curl se ve así:

curl [OPTIONS] [URL] El uso más simple de Curl es mostrar el contenido de una página. El siguiente comando mostrará la página de inicio de testdomain.com.

```
curl testdomain.com
```

Esto generará el código fuente completo de la página de inicio del dominio. Si no se especifica ningún protocolo, curl lo interpretará a HTTP.

```
$> curl http://www.net.net
<head><title>Document Moved</title></head>
```

```
<body><h1>Object Moved</h1>This document may be found <a
HREF="http://net.net">here</a></body>
```

Si no se indica lo contrario, curl realiza una solicitud HTTP con el método GET a la URL indicada. La salida del programa para el comando enviado será el cuerpo de la respuesta HTTP, en el caos anterior, el código HTML de la URL solicitada.

Si en vez de mostrar la salida recibida por pantalla nos interesa guardar la información en un archivo, podemos usar el parámetro `-o (--output)`:

```
curl –o output.html www.net.net
// Es equivalente a hacer esto en el SO
curl www.net.net > output.html
```

Como hemos visto en la sintaxis, por norma general la URL debe ponerse al final del comando. Opcionalmente podemos especificar la URL en cualquier lugar del comando si usamos el modificador `-s (--silent)`, pudiendo así alterar el orden de los argumentos de curl.

```
curl –s http://www.net.net –o output.html
```

En los ejemplos anteriores, el resultado que estamos obteniendo no es el deseado ya que el recurso se ha cambiado de ubicación o bien el sitio nos está redirigiendo a otra URI. Si usamos el parámetro `-L (--location)` podemos hacer que curl siga las redirecciones y obtenga el contenido final que buscamos.

```
$> curl http://www.dataden.tech
Redirecting
$> curl -L http://www.dataden.tech
<html><head><title>Loading...</title></head><body><script
type='text/javascript'>window.location.replace('http://www.dataden.tech/?
js=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOiJKb2tlbiIsImV4cCI6MTYzMzM4O
TE3OSwiaWF0IjoxNjMzMzgxOTc5LCJpc3MiOiJKb2tlbiIsImpzIjoxLCJqdGkiOiIycWxmMGdkZm
g2YWlzaHMxdjgwdWx0aTQiLCJuYmYiOjE2MzMzODE5NzksInRzIjoxNjMzMzgxOTc5NzgzNzQ1fQ.
y5LwDoSoZCpe2tzro_FbX7cSGIw4nx1XweNBqjpLXoo&sid=da601018-2557-11ec-a001-
58f389072b17');</script></body></html>
$> curl -L http://www.net.net
<html>
        <head>
                <title>NET.NET [The first domain name on the
Internet!]</title>
        </head>
        <body>
                <!-- Begin: Google Analytics -->
                <script>
```

```
                  (function(i,s,o,g,r,a,m)
{i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
                  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),
                  m=s.getElementsByTagName(o)
[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)

})(window,document,'script','//www.google-analytics.com/analytics.js','ga');
                  ga('create', 'UA-32196-28', 'auto');
                  ga('send', 'pageview');
                </script>
                <!-- End: Google Analytics -->
                <center>
                        <br /><br /><br /><br /><br /><br /><br /><br
/><br /><br />
                        <font face="impact, Arial, Helvetica, sans-serif"
size="14">
                                NET.NET
                        </font>
                        <br /><br /><br /><br />
                        <font face="Arial, Helvetica, sans-serif" size="1">
                                <a
href="http://who.godaddy.com/whoischeck.aspx?domain=NET.NET"
target="_blank">NET.NET</a> is the first and the best domain name on the
Internet!
                                <br />
                                Coming Soon...
                        </font>
                </center>
        </body>
</html>
```

De momento sólo hemos obtenido el HTML del recurso solicitado. Si queremos ver las cabeceras HTTP de nuestra solicitud y de la respuesta recibida podemos usar el parámetro `-v` (`--verbose`) que mostrará toda la información que intercambia el protocolo HTTP.

```
$> curl -v  http://www.net.net
*   Trying 34.250.90.28:80...
* TCP_NODELAY set
* Connected to net.net (34.250.90.28) port 80 (#0)
> GET / HTTP/1.1
> Host: net.net
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Cache-Control: private
< Content-Type: text/html
< Server: Microsoft-IIS/10.0
< Set-Cookie: ASPSESSIONIDASRSRRAR=IMFFLMBBBIFJNLNDHLOACDAI; path=/
```

```
< X-Powered-By: ASP.NET
< Date: Mon, 04 Oct 2021 21:40:49 GMT
< Content-Length: 1080
<
<html>
        <head>
                <title>NET.NET [The first domain name on the
Internet!]</title>
        </head>
        ...
```

En el ejemplo anterior podemos ver las cabeceras de nuestra solicitud (REQUEST) precedidas del carácter > mientras que las cabeceras de la respuesta (RESPONSE) se muestran precedidas dek carácter <.

formato corto y largo de los parámetros

Como ya hemos visto, los parámetros sirven para modificar el comportamiento por defecto de curl en función de las necesidades.

Los parámetros de una sola letra son rápidos de usar y recordar, pero tenemos un número limitado de parámetros de este tipo por lo que no podemos tener uno para cada opción. Los parámetros que usan palabras están para solucionar la falta de opciones con formato corto. Por otro lado, su uso hace que los comandos sean más legibles, por este motivo la mayoría de parámetros cortos tienen su equivalente largo.

En el formato corto, los parámetros están formados por un guion seguido de la letra correspondiente a la opción. Se puede usar un guion para cada opción, o incluir varias opciones detrás de un único guion.

```
$> curl -v -L http://example.com $> curl -vL http://example.com
```

En el formato largo, los parámetros se preceden de dos guiones y la palabra o palabras que forman la opción. Tras cada doble guion solo se puede indicar una opción.

```
$> curl --verbose --location http://example.com
```

Por último, aunque ya hemos visto como obtener la información de las cabeceras, podemos visualizar la información de la respuesta de forma completa usando la opción `-i (--include)` u obtener solo las cabeceras de la respuesta con el uso de la opción `-I (--head)`. Esto solo afecta al información de la respuesta (HTTP RESPONSE) y en el primer caso veremos la respuesta completa (headers & data) o solo los headers con la segunda opción.

```
$>  curl -I https://jsonplaceholder.typicode.com/todos/1
HTTP/2 200
date: Mon, 04 Oct 2021 21:57:55 GMT
content-type: application/json; charset=utf-8
```

```
content-length: 83
x-powered-by: Express
x-ratelimit-limit: 1000
x-ratelimit-remaining: 999
x-ratelimit-reset: 1631546224
vary: Origin, Accept-Encoding
access-control-allow-credentials: true
cache-control: max-age=43200
pragma: no-cache
expires: -1
x-content-type-options: nosniff
etag: W/"53-hfEnumeNh6Yirfjyjaujc0PPT+s"
via: 1.1 vegur
cf-cache-status: HIT
age: 10926
accept-ranges: bytes
expect-ct: max-age=604800,
report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"
report-to: {"endpoints":[{"url":"https:\/\/a.nel.cloudflare.com\/report\/v3?
s=LxJlkSosQdWmBFBOx1fB6zrbjSbU0iStl7jjtlVL27CtOEPxem%2Ffl9y
%2BCajMUopcZINOEsaufiU8A2glOLEmNpO5a40%2FyWb%2B4dBuspS8VGWnCRW4jxBBmh
%2F3FbupAEaxy66TUPariKJLqe3PL5iq"}],"group":"cf-nel","max_age":604800}
nel: {"success_fraction":0,"report_to":"cf-nel","max_age":604800}
server: cloudflare
cf-ray: 6991ab2c1a5037c7-MAD
alt-svc: h3=":443"; ma=86400, h3-29=":443"; ma=86400, h3-28=":443"; ma=86400, h3-
27=":443"; ma=86400
```

Para acabar, y como curiosidad, añadiendo la opción `-w "%{time_total}\n"` podremos ver el tiempo total que se ha tardado en recibir la respuesta del servidor.

6.3 Puntos finales y rutas

El término técnico endpoint hace referencia la la URL que se utiliza URL para hacer una petición. Para un mismo servicio, esta URL suele ser siempre la misma y es una de las características de las API REST.

Para un API, los *puntos finales* son URLs y se describen en la documentación de la APU de forma que los programadores sepan cómo usarla y acceder a los recursos y servicios que proporciona.

En el siguiente ejemplo tenemos una API con el siguiente endpoint.

```
GET https://my-api.com/Library/Books
```

Esto devolvería una lista completa de los libros que hay en la biblioteca.

Una ruta ("route") no es más que la parte de la URL que completa al endpoint y que se utiliza para seleccionar y/o acceder a diferentes componentes del servicio o API.

```
GET https://my-api.com/Library/Books/341
```

El ejemplo anterior devolvería el libro con identificador 341 de los que hay en la biblioteca.

Como ejemplo en SWAPI (Star Wars API) el endpoint es https://swapi.dev/api/. Este es el punto de entrada para todas las peticiones.

Además, hay muchas rutas dependiendo de la información que queramos recuperar/añadir/modificar/borrar.

```
$> curl https://swapi.dev/api/people/1
$> curl https://swapi.dev/api/planet/3
$> curl https://swapi.dev/api/vehicles
```

6.4 Métodos HTTP y cabeceras

Todas las peticiones HTTP tienen un método asociado, también llamado verbo. Por defecto ya hemos visto que se usa GET, pero también tenemos POST, HEAD y PUT que se utilizan bastante.

POST es el método HTTP que se ideó para enviar y recibir información de una aplicación web y es el que utilizan, entre otros, los formularios web.

Cuando los datos de un formulario se envían desde un navegador se mandan `URL encoded`, como una serie de pares nombre=valor separados con símbolos ampersand (&).

Para enviar datos con curl lo indicamos con la opción `-d (--data)` con el siguiente formato:

```
curl -d 'name=admin&shoesize=12' http://example.com/
```

Curl es capaz de seleccionar el método HTTP que debe utilizar en función de las opciones que recibe. Si utilizamos `-d` curl hará un POST, con `-I` hará un HEAD, etc. Con la opción `-X (--request)` podemos indicar qué método queremos que use curl.

```
curl -X POST -d 'imageSize=big&imageType=jpg' http://example.org/
```

Si usamos la opción POSTing with curl's `-d` ya hemos visto que por defecto se hace POST. Además de eso, se incluye una cabecera para indicar el formato de los datos enviados `Content-Type: application/x-www-form-urlencoded`. Esto es lo que hace un navegador cuando enviamos un formulario.

Puede ocurrir que los datos que estamos enviando no tengan ese formato, bien porque estemos subiendo algún archivo, enviando información binaria, usando JSON, XML, ... en estos

casos podemos, a través de la misma cabecera Content-Type, indicar el formato de la información que estamos enviando.

Por ejemplo si queremos enviar información en formato jSON:

```
curl -X "POST" -d '{"imageSize":"big","imageType":"jpg","scale":"false"}' -H
'Content-Type: application/json' https://example.com
```

6.5 Authentication

Cada solicitud HTTP puede ser autenticada. Un servidor o proxy puede necesitar que el usuario confirme su identidad o que tiene los permisos necesarios para acceder a una URL para realizar una acción. En ese caso se le enviará al cliente una respuesta en la que se le indique que debe proporcionar información de autenticación en la cabecera HTTP para que se le permita completar esa solicitud.

En curl, la forma de realizar una petición HTTP autenticadaes con la opción `-u (--user)` y proporcionando un usuario y contraseña (separados por `:`).

```
curl --user daniel:secret http://example.com/
```

Esto hará que curl utilice el método de autenticación HTTP por defecto, denominado "Basic"

Otras aplicaciones hacen uso de una clave secreta `secret key` o testigos de autorización `Authorization token`. Esta información nos la proporciona del servicio cuando lo creamos y configuramos.

Trello API Introduction

Azure Translator API Reference

Por ejemplo, si queremos usar el servicio `translate` de Azure tendremos que obtener una *secret key* y utilizarla en la cabecera de cada solicitud que realicemos

```
$> curl -X POST "https://api.cognitive.microsofttranslator.com/translate?api-
version=3.0&to=en&to=it" -H "Ocp-Apim-Subscription-Key: <here goes your susbcription
key>" -H "Content-Type: application/json; charset=UTF-8" -d "[{'Text':'Hola, com
esteu?'}]"
[{"detectedLanguage":{"language":"ca","score":1.0},"translations":[{"text":"Hello, how
are you?","to":"en"},{"text":"Ciao come stai?","to":"it"}]}]
```

Otras veces, para no enviar siempre la secret key, solicitamos una autorización temporal, obteniendo un `Authorization token` que podemos usar para acceder al servicio durante un

breve período de tiempo. Una vez que el token expire, debemos solicitar otro. Para el envío de tokens se utiliza la cabecera `Authorization: Bearer <token>`.

```
$> curl -X POST "https://api.cognitive.microsoft.com/sts/v1.0/issueToken"
        -H "Ocp-Apim-Subscription-Key: <here goes the secret key>" -d {}
eyJhbGciOiJodHRwOi8vd3d3LnczLm9yZy8...NJE
$> curl -X POST
"https://api.cognitive.microsofttranslator.com/translate?api-
version=3.0&to=en&to=it"
-H "Authorization: Bearer eyJhbGciOiJodHRwOi8vd3d3LnczLm9yZy8...NJE"
-H "Content-Type: application/json; charset=UTF-8" -d "[{'Text':'Hola, com
esteu?'}]"
{"error":{"code":401000,"message":"The request is not authorized because
credentials are missing or invalid."}}
$> curl -X POST
"https://api.cognitive.microsofttranslator.com/translate?api-
version=3.0&to=en&to=it"
-H "Authorization: Bearer eyJhbGciOiJodHRwOi8vd3d3LnczLm9yZy8...NJE
-H "Content-Type: application/json; charset=UTF-8" -d "[{'Text':'Hola, com
esteu?'}]"
[{"detectedLanguage":{"language":"ca","score":1.0},"translations":
[{"text":"Hello, how are you?","to":"en"},{"text":"Ciao come
stai?","to":"it"}]}]
```

6.6 Referencias

Everything curl es un libro gratuito que cubre con detalle prácticamente todo lo que hay que saber sobre curl.

freecodecamp.org

Sitio oficial de curl