Unidad 2. Programación de procesos: Anexo I

Descargar estos apuntes

Índice

I.1 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.

```java
// Primero obtenemos la carpeta del usuario
String homeDirectory = System.getProperty("user.home");
boolean isWindows = System.getProperty("os.name")
  .toLowerCase().startsWith("windows");

if (isWindows) {
    Runtime.getRuntime()
      .exec(String.format("cmd.exe /c dir %s", homeDirectory));
} else {
    Runtime.getRuntime()
      .exec(String.format("sh -c ls %s", homeDirectory));
}
```

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.

```java
// Calling app example
public void mouseClicked(MouseEvent e) {
  // Launch Page
  try {
    // Linux version
    Runtime.getRuntime().exec("open http://localhost:8153/go");
    // Windows version
    // Runtime.getRuntime().exec("explorer http://localhost:8153/go");
  } catch (IOException e1) {
    // Don't care
  }
}
```

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 propuesta para la actividad anterior

```
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);
```

I.2 Formato de salida por consola

> ⚠ **Codificación de caracteres**
>
> Un aspecto a tener en cuenta cuando trabajamos con flujos es la codificación de la información intercambiada entre procesos, que depende del sistema operativo en el que estemos trabajando. La mayoría de los sistemas (GNU/Linux, Mac OS, Android, iOS...) utilizan la codificación UTF-8, basada en el estándar Unicode.
>
> Por su parte, MS Windows utiliza sus propios formatos, incompatibles con el resto, como Windows-1252. Por lo que para manejar correctamente los datos en Java al usar mecanismos de comunicación entre procesos más avanzados, será necesario tener en cuenta el tipo de codificación que utiliza el propio sistema.
>
> ```
> // Getting the default encoding
> System.out.println(System.getProperty("file.encoding"));
> // Setting the encoding
> System.setProperty("file.encoding", "UTF-8");
>
> // Reading with a specific encoding
> BufferedReader reader = new BufferedReader(new InputStreamReader(System.in, "UTF-8"));
> // Writing with a specific encoding
> PrintWriter writer = new PrintWriter(new OutputStreamWriter(System.out, "UTF-8"));
>
> // Stdin with a specific encoding
> Scanner scanner = new Scanner(System.in, "UTF-8");
> // Stdout with a specific encoding
> System.out.println(new String("Hello, World!".getBytes("UTF-8")));
> ```

Salida por consola

En Java, podemos usar el objeto `System.out` para imprimir en la consola. Podemos usar el método `println` para imprimir una línea en la consola.

```
System.out.println("¡Hola, Mundo!");
```

Entrada por consola

En Java, podemos usar el objeto `System.in` para leer de la consola. Podemos usar la clase `Scanner` para leer de la consola.

```
Scanner scanner = new Scanner(System.in);
String nombre = scanner.nextLine();
```

Formato de cadena

En Java, podemos usar la clase `String` para formatear la salida. Podemos usar el método `format` para formatear la salida. Este método es similar al método `printf` en C.

```
String.format("El valor de PI es %.2f", Math.PI);
```

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.

```
DecimalFormat numberFormat = new DecimalFormat("#.00");
// Si usamos hashes en vez de ceros permitimos que .30 se vea como 0.3
// (los dígitos adicionales son opcionales)
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.

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

Uso de 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.

```
public class UsarColoresEnConsola {

public static final String ANSI_RESET = "\u001B[0m";
public static final String ANSI_BLACK = "\u001B[30m";
public static final String ANSI_RED = "\u001B[31m";
public static final String ANSI_GREEN = "\u001B[32m";
public static final String ANSI_YELLOW = "\u001B[33m";
public static final String ANSI_BLUE = "\u001B[34m";
public static final String ANSI_PURPLE = "\u001B[35m";
public static final String ANSI_CYAN = "\u001B[36m";
public static final String ANSI_WHITE = "\u001B[37m";

public static final String ANSI_BLACK_BACKGROUND = "\u001B[40m";
public static final String ANSI_RED_BACKGROUND = "\u001B[41m";
public static final String ANSI_GREEN_BACKGROUND = "\u001B[42m";
public static final String ANSI_YELLOW_BACKGROUND = "\u001B[43m";
public static final String ANSI_BLUE_BACKGROUND = "\u001B[44m";
public static final String ANSI_PURPLE_BACKGROUND = "\u001B[45m";
public static final String ANSI_CYAN_BACKGROUND = "\u001B[46m";
public static final String ANSI_WHITE_BACKGROUND = "\u001B[47m";

    public static void main(String[] args) {
        System.out.println(ANSI_GREEN + ANSI_WHITE_BACKGROUND + "Hola"
                           + ANSI_BLUE + ANSI_YELLOW_BACKGROUND + " Adiós" + ANSI_RESET);
    }
}
```

En el código anterior es importante resaltar que el último color que se muestra es el que se aplicará a la siguiente línea de texto. Por lo que es importante usar el ANSI_RESET para volver al color por defecto.