10.B. Flujos.

Sitio: VIRGEN DE LA PAZ
Curso: Programación
Libro: 10.B. Flujos.

Imprimido por: Cristian Esteban Gómez
Día: miércoles, 31 de mayo de 2023, 11:25

Tabla de contenidos

1. Introducción.

Hemos visto qué es un flujo y que existe un árbol de clases amplio para su manejo. Ahora vamos a ver en primer lugar los **flujos predefinidos**, también conocidos como de entrada y salida, y después veremos los **flujos basados en bytes** y los **flujos basados en carácter**.

Para saber más

Antes, hemos mencionado Unicode. Puedes consultar el origen y más cosas sobre Unicode, en el enlace de la Wikipedia:

Acerca de Unicode.

2. Flujos predefinidos. Entrada y salida estándar.

Tradicionalmente, los usuarios del sistema operativo Unix, Linux y también MS-DOS, han utilizado un tipo de entrada/salida conocida comúnmente por entrada/salida estándar. El fichero de entrada estándar (`stdin`) es típicamente el teclado. El fichero de salida estándar (`stdout`) es típicamente la pantalla (o la ventana del terminal). El fichero de salida de error estándar (`stderr`) también se dirige normalmente a la pantalla, pero se implementa como otro fichero de forma que se pueda distinguir entre la salida normal y (si es necesario) los mensajes de error.

Java tiene acceso a la entrada/salida estándar a través de la clase `System`. En concreto, los tres ficheros que se implementan son:

- `Stdin`. Es un objeto de tipo `InputStream`, y está definido en la clase System como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.

- `Stdout. System.out` implementa `stdout` como una instancia de la clase `PrintStream`. Se pueden utilizar los métodos `print()` y `println()` con cualquier tipo básico Java como argumento.

- `Stderr`. Es un objeto de tipo `PrintStream`. Es un flujo de salida definido en la clase `System` y representa la salida de error estándar. Por defecto, es el monitor, aunque es posible redireccionarlo a otro dispositivo de salida.

Para la entrada, se usa el método read para leer de la entrada estándar:

- `int System.in.read()`;
  - Lee el siguiente `byte (char)` de la entrada estándar.

- `int System.in.read(byte[] b);`
  - Leer un conjunto de bytes de la entrada estándar y lo almacena en el vector b.

Para la salida, se usa el método print para escribir en la salida estándar:

- `System.out.`print`(String);`
  - Muestra el texto en la consola.

- `System.out.`println`(String);`
  - Muestra el texto en la consola y seguidamente efectúa un salto de línea.

Normalmente, para leer valores numéricos, lo que se hace es tomar el valor de la entrada estándar en forma de cadena y entonces usar métodos que permiten transformar el texto a números (int, float, double, etc.) según se requiera.

Funciones de conversión.	
Método	Funcionamiento
`byte Byte.parseByte(String)`	Convierte una cadena en un número entero de un byte
`short Short.parseShort(String)`	Convierte una cadena en un número entero corto
`int Integer.parseInt(String)`	Convierte una cadena en un número entero
`long Long.parseLong(String)`	Convierte una cadena en un número entero largo
`float Float.parseInt(String)`	Convierte una cadena en un número real simple
`double Double.parseInt(String)`	Convierte una cadena en un número real doble

Funciones de conversión.	
Método	Funcionamiento
`boolean Boolean.parseBoolean(String)`	Convierte una cadena en un valor lógico

2.1. Flujos predefinidos. Entrada y salida estándar. Ejemplo.

Veamos un ejemplo en el que se lee por teclado hasta pulsar la tecla de retorno, en ese momento el programa acabará imprimiendo por la salida estándar la cadena leída.

Para ir construyendo la cadena con los caracteres leídos podríamos usar la clase `StringBuffer` o la `StringBuilder`. La clase `StringBuffer` permite almacenar cadenas que cambiarán en la ejecución del programa. `StringBuilder` es similar, pero no es síncrona. De este modo, para la mayoría de las aplicaciones, donde se ejecuta un solo hilo, supone una mejora de rendimiento sobre `StringBuffer`.

El proceso de lectura ha de estar en un bloque `try..catch.`

```
import java.io.IOException;

public class leeEstandar {
    public static void main(String[] args) {
        // Cadena donde iremos almacenando los caracteres que se escriban
        StringBuilder str = new StringBuilder();
        char c;
        // Por si ocurre una excepción ponemos el bloque try-cath
        try{
            // Mientras la entrada de terclado no sea Intro
            while ((c=(char)System.in.read())!='\n'){
                // Añadir el character leído a la cadena str
                str.append(c);
            }
        }catch(IOException ex){
            System.out.println(ex.getMessage()); }

        // Escribir la cadena que se ha ido tecleando
        System.out.println("Cadena introducida: " + str);
    }
}
```

Mismo código copiable:

```
import java.io.IOException;

public class leeEestandar {

    public static void main(String[] args) {

// Cadena donde iremos almacenando los caracteres que se escriban

        StringBuilder str = new StringBuilder();

        char c;

// Por si ocurre una excepción ponemos el bloque try-cath

        try{

         // Mientras la entrada de terclado no sea Intro
```

```
            while ((c=(char)System.in.read())!='\n'){

              // Añadir el character leído a la cadena str

                     str.append(c);

           }

      }catch(IOException ex){

           System.out.println(ex.getMessage()); }


// Escribir la cadena que se ha ido tecleando

       System.out.println("Cadena introducida: " + str);

   }

}
```

Autoevaluación

Señala la opción correcta:

- ○ Read es una clase de System que permite leer caracteres.
- ○ StringBuffer permite leer y StringBuilder escribir en la salida estándar.
- ○ La clase keyboard es la clase a utilizar al leer flujos de teclado.
- ○ Stderr por defecto se dirige al monitor pero se puede direccional a otro dispositivo.

3. Flujos basados en bytes.

Este tipo de flujos es el idóneo para el manejo de entradas y salidas de bytes, y su uso por tanto está orientado a la lectura y escritura de datos binarios.

Para el tratamiento de los flujos de bytes, Java tiene dos clases abstractas que son `InputStream` y `OutputStream`. Cada una de estas clases abstractas tiene varias subclases concretas, que controlan las diferencias entre los distintos dispositivos de E/S que se pueden utilizar.

```
class FileInputStream extands InputStream {

   FileInputStream (String fichero) throws FileNotFoundException;

   FileInputStream (File fichero) throws FileNotFoundException;

   ... ... ...

 }

class FileOutputStream extends OutputStream {

   FileOutputStream (String fichero) throws FileNotFoundException;

   FileOutputStream (File fichero) throws FileNotFoundException;

   ... ... ...

          }
```

`OutputStream` y el `InputStream` y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Por ejemplo, podemos copiar el contenido de un fichero en otro:

```
void copia (String origen, String destino) throws IOException {
  try{
    // Obtener los nombres de los ficheros de origen y destino
    // y abrir la conexión a los ficheros.
    InputStream fentrada = new FileInputStream(origen);
    OutputStream fsalida = new FileOutputStream(destino);
        // Crear una variable para leer el flujo de bytes del origen
    byte[] buffer= new byte[256];
    while (true) {
             // Leer el flujo de bytes
         int n = fentrada.read(buffer);
             // Si no queda nada por leer, salir del bucle
         if (n < 0)
             break;
             // Escribir el flujo de bytes leídos al fichero destino
         fsalida.write(buffer, 0, n);
        }
    // Cerrar los ficheros
    fentrada.close();
    fsalida.close();
  }catch(IOException ex){
        System.out.println(ex.getMessage()); }
}
```

Mismo código copiable:

```
void copia (String origen, String destino) throws IOException {

    try{

          // Obtener los nombres de los ficheros de origen y destino

          // y abrir la conexión a los ficheros.
```

```
        InputStream fentrada = new FileInputStream(origen);

        OutputStream fsalida = new FileOutputStream(destino);

            // Crear una variable para leer el flujo de bytes del origen

        byte[] buffer= new byte[256];

        while (true) {

                // Leer el flujo de bytes

             int n = fentrada.read(buffer);

                // Si no queda nada por leer, salir del while

             if (n < 0)

                break;

                // Escribir el flujo de bytes leídos al fichero destino

             fsalida.write(buffer, 0, n);

            }

            // Cerrar los ficheros

        fentrada.close();

        fsalida.close();

      }catch(IOException ex){

            System.out.println(ex.getMessage()); }
```

4. Flujos basados en caracteres.

Las clases orientadas al flujo de bytes nos proporcionan la suficiente funcionalidad para realizar cualquier tipo de operación de entrada o salida, pero no pueden trabajar directamente con **caracteres Unicode**, los cuales están **representados por dos bytes**. Por eso, se consideró necesaria la creación de las clases orientadas al flujo de caracteres para ofrecernos el soporte necesario para el tratamiento de caracteres.

Para los flujos de caracteres, Java dispone de dos clases abstractas: `Reader` y `Writer`.

`Reader`, `Writer`, y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Hay que recordar que **cada vez que se llama a un constructor se abre el flujo de datos y es necesario cerrarlo** cuando no lo necesitemos.

Existen muchos tipos de flujos dependiendo de la utilidad que le vayamos a dar a los datos que extraemos de los dispositivos.

Un flujo puede ser envuelto por otro flujo para tratar el flujo de datos de forma cómoda. Así, un `bufferWriter` nos permite manipular el flujo de datos como un buffer, pero si lo envolvemos en un `PrintWriter` lo podemos escribir con muchas más funcionalidades adicionales para diferentes tipos de datos.

En este ejemplo de código, se ve cómo podemos escribir la salida estándar a un fichero. Cuando se teclee la palabra "salir", se dejará de leer y entonces se saldrá del bucle de lectura.

Podemos ver cómo se usa `InputStreamReader` que es un puente de flujos de bytes a flujos de caracteres: lee bytes y los decodifica a caracteres. `BufferedReader` lee texto de un flujo de entrada de caracteres, permitiendo efectuar una lectura eficiente de caracteres, vectores y líneas.

Como vemos en el código, usamos `FileWriter` para flujos de caracteres, pues para datos binarios se utiliza `FileOutputStream`.

```
try{
  PrintWriter out = null;
  out = new PrintWriter(new FileWriter("c:\\salida.txt", true));

  BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
  String s;
  while (!(s = br.readLine()).equals("salir")){
          out.println(s);
  }
  out.close();
 }
catch(IOException ex){
 System.out.println(ex.getMessage());
```

Mismo código copiable:

```
try{
    PrintWriter out = null;
    out = new PrintWriter(new FileWriter("c:\\salida.txt", true));

    BufferedReader br = new BufferedReader(
              new InputStreamReader(System.in));
    String s;
    while (!(s = br.readLine()).equals("salir")){
            out.println(s);
    }
    out.close();
   }
```

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

```
          System.out.println(ex.getMessage()); }
```

Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación:

Para flujos de caracteres es mejor usar las clases Reader y Writer en vez de InputStream y OutputStream.

- ○ Verdadero.
- ○ Falso.

5. Rutas de los ficheros.

En los ejemplos que vemos en el tema estamos usando la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:

```
c:\\datos\Programacion\fichero.txt
```

Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas suele cambiar dependiendo del sistema operativo en el que se esté ejecutando el programa.

Para evitar problemas en la ejecución de los programas cuando se ejecuten en uno u otro sistema operativo, y por tanto persiguiendo que nuestras aplicaciones sean lo más portables posibles, se recomienda usar en Java: `File.separator.`

Podríamos hacer una función que al pasarle una ruta nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

```
static String substFileSeparator(String ruta){
String separador = "\\";

        try{
            // Si estamos en Windows
            if ( File.separator.equals(separador) )
                separador = "/" ;
            // Reemplaza todas las cadenas que coinciden con la expresión
            // regular dada oldSep por la cadena File.separator
            return ruta.replaceAll(separador, File.separator);
        }catch(Exception e){
            // Por si ocurre una java.util.regex.PatternSyntaxException
            return ruta.replaceAll(separador + separador, File.separator);
        }
 }
```

Mismo código copiable:

```
String substFileSeparator(String ruta){
```

```
 String separador = "\\";
```

```
        try{
```

```
     // Si estamos en Windows
```

```
          if ( File.separator.equals(separador) )
```

```
               separador = "/" ;
```

```
        // Reemplaza todas las cadenas que coinciden con la expresi?n
```

```
        // regular dada oldSep por la cadena File.separator
```

```
          return ruta.replaceAll(separador, File.separator);
```

```
        }catch(Exception e){
```

```
    // Por si ocurre una java.util.regex.PatternSyntaxException
```

```
          return ruta.replaceAll(separador + separador, File.separator);
```

```
        }
```

```
       }
```

Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación.

Cuando trabajamos con fichero en Java, no es necesario capturar las excepciones, el sistema se ocupa automáticamente de ellas. ¿Verdadero o Falso?

- ○ Verdadero.
- ○ Falso.