Sumario

Concepto de Streams o Corrientes

Una corriente (Stream) se puede pensar como un flujo de bytes (datos) desde una fuente a un receptor, donde la fuente y el receptor puede ser cualquier objeto capaz de almacenar, emitir, leer o crear datos.
Las dos posibilidades de una corriente son ser fuente (o emisora) y receptora. Una corriente fuente inicia el flujo de bytes y también es conocida como corriente de ingreso.
Una corriente receptora finaliza el flujo de bytes y también es conocida como corriente de salida.
Fuentes y receptores son ambas corrientes nodales (nodos de comunicación), donde un nodo es cualquier tipo de objeto.
Los tipos de corrientes nodales son, por ejemplo, archivos, memoria y tuberías (pipes) entre subprocesos o procesos.

Stream de consola

Los sistemas operativos modernos definen un mínimo de tres corrientes de caracteres estándar para el acceso a los dispositivos periféricos:

- El ingreso estándar (Standard Input)	System.in
- La salida estándar (Standard Output)	System.out
- La corriente de errores estándar (Standard Error)	System.err

Las corrientes estándar están asociadas siempre a un dispositivo por defecto. Sin embargo. Como son corrientes (en inglés, streams) pueden ser dirigidas hacia a otros dispositivos (como el ejemplo típico de la salida por pantalla que se direcciona a la impresora).

Ejemplo de lectura de Stream de consola

```
String s;
// Crea un lector con buffer para cada línea del teclado.
InputStreamReader ir = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(ir);
System.out.println("Unix: Tipear ctrl-d o ctrl-c para salir."
      + "\nWindows: Tipear ctrl-z para salir ");
try {
      // Lee cada línea de entrada y lo muestra por pantalla.
      s = in.readLine();
      while (s != null) {
            System.out.println("Leído: " + s);
            s = in.readLine();
      }
      // Cierra el lector con buffer
      in.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

Creación de un objeto del tipo File

Esta clase se utiliza para representar abstractamente un archivo en disco. La representación abstracta se debe a que cada sistema operativo gestiona el camino hasta un archivo de forma diferente (un ejemplo de esto es el carácter que divide los directorios y subdirectorios que puede

ser “/” o “\”). En el caso de los nombres UNC (Uniform Naming Convention) para Windows, por ejemplo, se debe utilizar “\\” para indicar el camino del archivo en el código de un programa Java. Para evitar problemas con los separadores de nombres, se puede obtener cual es el separador correcto para la plataforma en la cual se ejecuta el programa obteniendo su valor de la propiedad del sistema file.separator
Para manejar los nombres y caminos de archivos, esta clase define un camino abstracto o ruta. Un camino abstracto tiene dos componentes:

- Una cadena de prefijo opcional dependiente del sistema con la letra del dispositivo seguido de: “/” para Unix o “\\” para Windows.
- Una secuencia de nombres con sus respectivos separadores (puede ser nulo)

Todos los nombres se interpretarán como directorios a excepción del último.

Nota: Cuando se construye el objeto nunca se crea un archivo. La máquina virtual no realiza una operación de entrada – salida hasta que un método específico intenta realizarlo, como por ejemplo un println . En ese caso, si el archivo no existe, lo crea. Si el archivo existe, lo trunca. Una opción para crear un archivo vacío sin realizar ninguna operación es invocar al método createNewFile .

```
File f=new File(“texto.txt”);
f.createNewFile();
```

Los directorios se tratan igual que archivos en Java. La clase File soporta métodos para recuperar un vector de archivos de un directorio y armar una lista con ellos para tratarlos en programa.

```
File archivo = new File("archivo.txt");
File archivo2 = new File("MisDocs", "archivo.txt");
try {
      // A partir del objeto File creamos el archivo físicamente
      if (miArchivo.createNewFile())
            System.out.println("El archivo se ha creado correctamente");
      else
            System.out.println("No ha podido ser creado el archivo");
      if (miArchivo2.createNewFile())
            System.out.println("El archivo se ha creado correctamente");
      else
            System.out.println("No ha podido ser creado el archivo");
} catch (IOException ioe) {
      ioe.printStackTrace();
}


Salida por consola del programa
El archivo se ha creado correctamente
java.io.IOException: The system cannot find the path specified
at java.io.WinNTFileSystem.createFileExclusively(Native Method)
at java.io.File.createNewFile(Unknown Source)
at archivos.ArchivosAnteriores2.main(ArchivosAnteriores2.java:22)
```

Notar que el primer intento crea al archivo por no especificar el directorio y usar el definido por defecto, que siempre es el CLASSPATH de donde se ejecuta el programa.
Pero cuando se intenta crearlo en un directorio llamado MisDocs , se produce un error de E/S si el directorio no existe.

Funciones de utilidad

Como la construcción de un objeto del tipo File implica la creación solamente de un objeto en memoria pero no se realiza ninguna entrada salida hasta el momento en que se indica explícitamente, existen distintas funciones que se pueden utilizar tanto antes como después de hacer una E/S

Nombres de archivos

String **getName()**: Retorna el nombre abstracto del archive que define la clase.

String **getPath()**: Convierte el nombre y camino completo abstracto de la clase a un nombre dependiente de la plataforma.

String **getAbsolutePath()**: Si el nombre de archivo abstracto que posee el objeto es absoluto, lo devuelve. Si es relativo, lo transforma a absoluto y lo retorna.

String **getParent()**: Si el archivo está en un directorio, retorna el nombre. Si no puede resolver el nombre del directorio retorna un nulo.

boolean **renameTo**(File newName): Renombra el archivo al nombre indicado.

Comprobación de archivos

boolean **exists()**: Verifica si el nombre de directorio o archivo con el que se .construyó el objeto existe

boolean **canWrite()**: Si el archivo existe, verifica si se puede escribir.

boolean **canRead()**: Si el archivo existe, verifica si se puede leer.

boolean **isFile()**: Verifica si el nombre con el que se construyó el objeto es un archivo.

boolean **isDirectory()**: Verifica si el nombre con el que se construyó el objeto es un directorio.

boolean **isAbsolute()**: Verifica si el nombre con el que se construyó el objeto es absoluto.

De información y utilidad en general

long **lastModified**(): Retorna cuando el objeto del sistema que está siendo representado por el que está en memoria fue modificado por última vez.

long **length**(): Si el objeto representado es un archivo, retorna su tamaño, sino (si es un directorio) puede retornar cualquier valor.

boolean **delete**(): Si el objeto representado es un archivo, lo borra. Si es un directorio, debe estar vacío para borrarlo.

Utilidades de directorio

boolean **mkdir**(): Crea un directorio con el nombre almacenado por el objeto en memoria

String[] **list**(): Si el objeto representado es un archivo, retorna un nulo. Si es un directorio, retorna un vector de String que representa los nombres de archivos del directorio.

E/S de Corrientes de Archivos

Como se mencionó anteriormente, se utilizan decoradores para simplificar el acceso y escritura de archivos, ya que estos también conforman corrientes de E/S. Un decorador a su vez, puede ser argumento de creación para otro decorador. Un ejemplo de estos casos para entradas y salidas son los siguientes:

Entradas de archivos

- Usar la clase FileReader para leer caracteres en bruto (sin formato)
- Usar la clase BufferedReader para utilizar el método readLine()

Salidas a archivos

-Usar la clase FileWriter para escribir caracteres en bruto (sin formato)
-Usar la clase PrintWriter para usar los métodos print() y println()

Ejemplo de Entrada

```
try {
      new BufferedReader(
            new FileReader(“texto.txt”))
                  .lines()
                  .foreach(System.out::println);
} catch (FileNotFoundException e1) {
      System.err.println("Archivo no encontrado : " + file);
} catch (IOException e2) {
      e2.printStackTrace();//Para cualquier otra excepción de E/S
}
```

Ejemplo de Salida

```
File file = new File(“texto.txt”); // Crear el archivo
try {
      // Crear el lector con buffer para leer cada línea del
      // estándar input
      BufferedReader in = new BufferedReader(new InputStreamReader(
            System.in));
      PrintWriter out = new PrintWriter(new FileWriter(file));
      // Para escribir este archivo
      String s;
      System.out.print("Ingresar el texto del archivo : ");
      System.out.println("[Tipee ctrl-d (or ctrl-z) para finalizar.]");
      while ((s = in.readLine()) != null) {
            // Leer cada línea y mostrarla por pantalla
            out.println(s);
      }
      in.close(); // Cerrar el lector con buffer y el print
      out.close();
```

```
} catch (IOException e) {
      e.printStackTrace();
}
```

Ejemplo de Salida usando try with resources

```
File file = new File(“texto.txt”); // Crear el archivo
try (
      BufferedReader in = new BufferedReader(new InputStreamReader(
            System.in));
      PrintWriter out = new PrintWriter(new FileWriter(file));
){
      String s;
      System.out.print("Ingresar el texto del archivo : ");
      System.out.println("[Tipee ctrl-d (or ctrl-z) para finalizar.]");
      while ((s = in.readLine()) != null) {
            // Leer cada línea y mostrarla por pantalla
            out.println(s);
      }
} catch (IOException e) {
      e.printStackTrace();
}
```

Clases básicas para el manejo de Streams

El paquete java.io contiene dos clases, InputStream y OutputStream , de las que derivan la mayoría de las clases de este paquete.
La clase InputStream es una superclase abstracta que proporciona un interfaz de programación mínima y una implementación parcial de la corriente de entrada. La clase InputStream define métodos para leer bytes o vectores de bytes, marcar posiciones en la corriente, saltar bytes de la entrada, conocer el número de bytes disponibles para ser leídos, y reiniciar la posición actual dentro de la corriente. Una corriente de entrada se abre automáticamente cuando se crea. Se puede cerrar una corriente explícitamente con el método close() o puede dejarse que se cierre implícitamente cuando se recolecta la basura, lo que ocurre cuando el objeto deja de ser referenciado.
La clase OutputStream es una superclase abstracta que proporciona un interfaz de programación mínima y una implementación parcial de las corrientes de salida. Una corriente de salida se abre automáticamente cuando se crea. Se puede cerrar una corriente explícitamente con el método close() o se puede dejar que se cierre implícitamente cuando se recolecta la basura, lo que ocurre cuando el objeto deja de ser referenciado.
El paquete java.io contiene muchas subclases de InputStream y OutputStream que implementan funciones específicas de entrada y salida. Por ejemplo, FileInputStream y FileOutputStream son corrientes de entrada y salida que operan con archivos en el sistema operativo nativo en su formato.
Reader es una clase abstracta para leer corrientes de caracteres. Los únicos métodos que la subclase debe implementar son read(char[], int, int) y close() . Sin embargo, las subclases pueden sobrescribir otros métodos con el fin de lograr mejor rendimiento, mayor funcionalidad o ambos.
Writer es también una clase abstracta pero para escribir en una corriente de caracteres. Conceptualmente es la contrapartida de Reader y en este caso las subclases deben implementar obligatoriamente los métodos write(char[], int, int) , flush() , y close() . Sin embargo, análogamente a Reader , las subclases pueden sobrescribir otros métodos con el fin de lograr mejor rendimiento, mayor funcionalidad o ambos.

Streams clasicos

Todos los sistemas operativos modernos se manejan con corrientes. La principal dificultad en el

maneja de las mismas sobrevino con la aparición del Unicode como estándar para remplazar al viejo ASCII. Si bien el Unicode es compatible con su noble antecesor, la dificultad se presentó en el hecho que los caracteres en ASCII ocupan un byte, mientras que en Unicode ocupan dos. Como se debe mantener compatibilidad con los programas que se manejan aún en la actualidad sólo en ASCII, Java soporta dos tipos de corrientes:

- Orientadas al byte
- Orientadas al carácter

Para el manejo de estas, se proveen una serie de clases que permiten a través de la creación de objetos abstraerse de las dificultades de su gestión, más allá de si el carácter está escrito en Unicode o ASCII, por lo tanto, las entradas y salidas de caracteres son manejadas por “lectores” y “escritores”, que no otra cosa que instancias de las clases provistas por el lenguaje para operaciones de entrada y salida.
Las entradas y salidas de bytes son manejadas por corrientes de ingreso y corrientes de salida. Sin embargo, en la terminología del lenguaje se separa conceptualmente a los bytes de ASCII de los caracteres Unicode. Por lo general, los términos:
Corriente: se refiere a una corriente de bytes (ASCII). (**InputStream**, **OutputStream**)
Lector y Escritor: se refieren a corrientes de caracteres (Unicode). (**Reader**, **Writer**)

Métodos de InputStream

InputStream es la superclase de las clases que manejan corrientes de ingreso de bytes. Provee los métodos para leer bytes como también para manipular la corriente. Como es una clase abstracta, nunca se crea un objeto de esta clase.

Existen tres métodos básicos de lectura:

int **read**(): Lee un byte de entrada y retorna un entero representando a dicho byte. Si encuentra el fin de la corriente, retorna -1.

int **read**(byte[] buffer): Intenta llenar el vector buffer con bytes y retorna el número de ellos leídos satisfactoriamente. Si encuentra el fin de la corriente, retorna -1.

int **read**(byte[] buffer, int desplazamiento, int longitud): Intenta llenar el vector buffer con bytes partiendo de la posición indicada en desplazamiento y la cantidad indicada en longitud. Retorna el número de bytes leídos satisfactoriamente. Si encuentra el fin de la corriente, retorna -1.

Otros métodos disponibles para el manejo de corrientes:

void **close**(): Cierra la corriente de entrada y libera cualquier recurso tomada por esta.

int **available**(): Retorna la cantidad de bytes disponibles para leer desde la corriente de entrada. El resultado puede variar de una plataforma a otra. Por ejemplo, el carácter de nueva línea en Windows se traduce como los caracteres de retorno de carro y alimentación de línea mientras que los sistemas operativos tipo Unix solo es un carácter.

long **skip**(long n): Saltea la cantidad de bytes especificados como argumento y retorna la cantidad efectivamente salteados.

boolean **markSupported**(): Retorna verdadero si se puede marcar la corriente que se utiliza de entrada.

void **mark**(int limiteLectura): Crea una marca en la posición actual de la corriente de entrada. El argumento indica la cantidad de bytes que se pueden leer a partir de la marca. Pasado este valor, la marca se vuelve inválida.

void **reset**(): Retorna la posición del objeto del tipo InputStream que se marcó con mark , en caso de ser posible la marca.

Métodos de OutputStream

OutputStream es la superclase de las clases que manejan corrientes de salida de bytes. Provee los métodos para escribir bytes como también para manipular la corriente. Como es una clase abstracta, nunca se crea un objeto de esta clase.

Existen tres métodos básicos de escritura:

void **write**(int c): Escribe un entero en la corriente de salida.

void **write**(byte[] buffer): Escribe el vector buffer con los bytes que este almacena en la corriente de salida.

void **write**(byte[] buffer, int desplazamiento, int longitud): Escribe el vector buffer con los bytes que este almacena en la corriente de salida, partiendo de la posición indicada en desplazamiento y la cantidad indicada en longitud.

Otros métodos disponibles para el manejo de corrientes:

void **close**(): Cierra la corriente de salida y libera cualquier recurso tomada por esta.

void **flush**(): Vacía hacia la corriente el contenido del buffer de salida forzando a que cualquier dato contenido en este sea escrito en el objeto asociado a la corriente de salida. En realidad este método fue diseñado para ser sobrescrito en las subclases porque en esta no hace nada.

Métodos de Reader

Reader es la superclase de las clases que manejan corrientes de ingreso de caracteres. Provee los métodos para leer caracteres como también para manipular la corriente. Como es una clase abstracta, nunca se crea un objeto de esta clase.

Existen tres métodos básicos de lectura:

int **read**(): Lee un caracter y lo retorna como un entero.

int **read**(char[] cbuf): Intenta llenar el vector cbuf con caracteres y retorna el número de ellos leídos satisfactoriamente. Si encuentra el fin de la corriente, retorna -1.

int **read**(char[] cbuf, int desplazamiento, int longitud): Intenta llenar el vector cbuf con caracteres partiendo de la posición indicada en desplazamiento y la cantidad indicada en longitud. Retorna el número de caracteres leídos satisfactoriamente. Si encuentra el fin de la corriente, retorna -1
Otros métodos disponibles para el manejo de corrientes:

void **close**(): Cierra la corriente de entrada y libera cualquier recurso tomada por esta.

boolean **ready**(): Retorna verdadero cuando el objeto del tipo Reader está listo para leer datos.

long **ski**p(long n): Saltea la cantidad de bytes especificados como argumento y retorna la cantidad efectivamente salteados.

boolean **markSupported**(): Retorna verdadero si se puede marcar la corriente que se utiliza de entrada.

void **mark**(int limiteLectura): Crea una marca en la posición actual de la corriente de entrada. El argumento indica la cantidad de bytes que se pueden leer a partir de la marca. Pasado este valor, la marca se vuelve inválida.

void **reset**(): Retorna la posición del objeto del tipo Reader que se marcó con mark , en caso de ser posible la marca.

Métodos de Writer

Writer es la superclase de las clases que manejan corrientes de salida de caracteres. Provee los métodos para escribir caracteres como también para manipular la corriente. Como es una clase abstracta, nunca se crea un objeto de esta clase.

Los métodos básicos de escritura:

void **write**(int c): Escribe un entero en la corriente de salida.

void **write**(char[] cbuf): Escribe el vector cbuf con los caracteres que este almacena en la corriente de salida.

void **write**(char[] cbuf, int desplazamiento, int longitud): Escribe el vector buffer con los bytes que este almacena en la corriente de salida, partiendo de la posición indicada en desplazamiento y la cantidad indicada en longitud.

void **write**(String cadena): Escribe el String cadena en la corriente de salida.

void **write**(String cadena, int desplazamiento, int longitud): Escribe el String cadena en la corriente de salida, partiendo de la posición indicada en desplazamiento dentro del mismo y la cantidad de caracteres indicada en longitud.

Otros métodos disponibles:
void **close**(): Cierra la corriente de salida y libera cualquier recurso tomada por esta.

void **flush**(): Vacía hacia la corriente el contenido del buffer de salida forzando a que cualquier dato contenido en este sea escrito en el objeto asociado a la corriente de salida.

Corrientes Nodales

En Java existen tres tipos fundamentales de nodos: archivos, memoria (como vectores o cadenas) y tuberías (pipes: un canal de comunicación entre dos subprocesos).

Corrientes nodales según su orientación.

Tipo	Corrientes de bytes	Corrientes de Caracteres
Archivos	FileInputStream FileOutputStream	FileReader FileWriter
Memoria: vectores	ByteArrayInputStream ByteArrayOutputStream	CharArrayReader CharArrayWriter
Memoria: Cadenas		StringReader StringWriter
Tubería (pipe)	PipedInputStream PipedOutputStream	PipedReader PipedWriter

Corrientes de bytes

FileInputStream

Esta clase se utiliza para leer bytes desde un archivo en el sistema de archivos de la máquina local. La misma se diseñó con la intención de leer bytes en bruto desde una corriente, como por ejemplo puede ser, una serie de bytes perteneciente a una imagen.

FileOutputStream

La clase fue diseñada para escribir datos a un objeto de tipo File o FileDescriptor . Cuando un archivo está disponible, es creado o no dependerá del comportamiento del sistema de archivos de cada plataforma en particular donde se ejecute. Algunas plataformas en particular, permiten la apertura de un solo FileOutputStream para escritura de un archivo por vez. En tales situaciones, el constructor de esta clase falla si el archivo en cuestión ya se encuentra abierto.
Esta clase tiene la intención de escribir corrientes de bytes en bruto en un archivo como pueden ser, por ejemplo, archivos de imágenes.

ByteArrayInputStream

Posee un buffer interno que almacenará los bytes a leer de la corriente. Un contador interno mantiene el seguimiento del próximo byte que será suministrado por el método read cuando lee de a un byte.
Utilizar el método close de esta clase no tiene sentido, por lo tanto no tiene ningún efecto.

ByteArrayOutputStream

Esta clase implementa una corriente de salida en la cual los datos se escriben en un vector de bytes. El buffer se incrementa automáticamente a medida que se escriben datos en él. Los datos se pueden recuperar con los métodos toByteArray() y toString() .

Cerrar un objeto del tipo ByteArrayOutputStream no tiene efecto. Los métodos de esta clase pueden ser llamados después de cerrada la corriente sin generar una IOException .

PipedInputStream

Una corriente “entubada” (piped) de ingreso debe ser conectada a una corriente “entubada” de salida. La corriente entubada de salida luego provee cualquier byte de dato escrito en la corriente entubada de salida. Por lo general, los datos son leídos de un objeto del tipo PipedInputStream por un subproceso y los bytes de datos son escritos a la correspondiente PipedOutputStream por otro thread. Intentar utilizar ambos tipos de objetos sobre un mismo subproceso no es recomendable porque puede causar un deadlock (interbloqueo) sobre dicho thread. La corriente entubada de ingreso posee un buffer desacoplando las operaciones de lectura de las de escritura dentro de sus límites.

PipedOutputStream

Una corriente “entubada” (piped) de salida debe ser conectada a una corriente “entubada” de ingreso para crear un “tubo” (pipe) de comunicaciones. La corriente entubada de salida es el extremo final del tubo de envío. Por lo general, los datos son escritos a un objeto del tipo PipedOutputStream por un subproceso y los bytes de datos son leídos de la correspondiente PipedInputStream al que está conectado por otro thread. Intentar utilizar ambos tipos de objetos sobre un mismo subproceso no es recomendable porque puede causar un deadlock (interbloqueo) sobre dicho thread.

Corrientes de caracteres

FileReader

Esta clase sirve para leer caracteres de un archivo. Los constructores de la clase asumen la codificación por defecto de los caracteres y por lo tanto, el tamaño en bytes del buffer por defecto es el apropiado. Para especificar estos valores, se debe construir un objeto del tipo InputStreamReader sobre un FileInputStream .

FileWriter

Esta clase es la que conviene para escribir archivos de caracteres: Sus constructores asumen la codificación de caracteres por defecto y por lo tanto, el tamaño en bytes del buffer por defecto es el apropiado. Para especificar estos valores, se debe construir un objeto del tipo OutputStreamWriter sobre un FileOutputStream .
Cuando un archivo está disponible o es creado depende de la plataforma sobre la que se ejecute el código. Algunas plataformas en particular, permiten que los archivos que se abren para escritura sólo se los puedan abrir por medio de un solo objeto del tipo escritura, como puede ser particularmente FileWriter . En tales situaciones los constructores en esta clase fallan si el archivo ya se encuentra abierto.
FileWriter fue pensado para escribir corrientes de caracteres. Para escribir corrientes de bytes en bruto, se debe considerar el uso de FileOutputStream .

CharArrayReader

Esta clase implementa un buffer de caracteres que puede ser utilizado como una corriente de ingreso de caracteres.

CharArrayWriter

Esta clase implementa un buffer de caracteres que puede ser utilizado como un Writer . El buffer es incrementado automáticamente cuando se escriben datos en la corriente. Los datos pueden ser recuperados con los métodos toCharArray() y toString() . Invocar el método close() en esta clase no tiene efecto y sus otros métodos pueden ser invocados luego del cierre de la corriente si ocasionar un IOException .

StringReader

Una corriente de caracteres cuya fuente es un String

StringWriter

Una corriente de caracteres que recolecta su salida en un buffer de cadenas, el cual puede luego ser utilizado para construir un String . Cerrar un StringWriter no tiene efecto y sus otros métodos pueden ser invocados luego del cierre de la corriente si ocasionar un IOException .

PipedReader

Sirve para corrientes de ingreso de caracteres entubadas.

PipedWriter

Sirve para corrientes de salida de caracteres entubadas.

Stream sin buffer

```
try {
      FileReader entrada = new FileReader(args[0]);
      FileWriter salida = new FileWriter(args[1]);
      char[] buffer = new char[128];
      int caracLeidos;
      // primera lectura sobre el buffer
      caracLeidos = entrada.read(buffer);
      while (caracLeidos != -1) {
            // escribir la salida del buffer al archivo de salida
            salida.write(buffer, 0, caracLeidos);
            // Próxima lectura sobre el buffer
            caracLeidos = entrada.read(buffer);
      }
      entrada.close();
      salida.close();
      } catch (IOException e) {
            e.printStackTrace();
      }
```

Manejar el buffer de esta manera es tedioso y puede provocar errores. Sin embargo, existen clases que permiten despreocuparse del manejo del buffer ya que los objetos de su tipo se encargan de este detalle brindando la posibilidad de leer de a una “línea a la vez”. Entre este tipo de clases se encuentra, por ejemplo, BufferedReader como se verá posteriormente.

Corrientes con buffer

La lectura con buffer, lo cual permite como se mencionó anteriormente, leer de a una línea por vez. Notar como mejora el código siendo más compacto y claro. Tener en cuenta que el buffer se construye a partir de “envolver” la clase FileReader con BufferedReader en la sentencia BufferedReader bufEntrada = new BufferedReader(entrada) . La lectura de “una línea por vez” se

realiza cuando se invoca al método readLine() .

```
try {
      BufferedReader in = new BufferedReader(new FileReader(“file1.txt”));
      BufferedWriter out= new BufferedWriter(new FileWriter(“file2.txt”));
      for(Object obj:in.lines().toArray()){
            out.write(obj.toString());
            out.newLine();
      }
      in.close();
      out.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

Tuberías

Como se mencionó anteriormente, las clases de tuberías están diseñadas para trabajar en conjunto y en subprocesos separados. Por este motivo, se diseña la clase ThreadDeESBytes que permite construir threads indicando en su constructor el nombre del proceso (en este caso un subprocesos), el tipo de corriente de entrada y el de salida. Esta clase otorga la flexibilidad de manejar threads con los dos tipos de corrientes (entrada y salida) leyendo de una y escribiendo sobre la otra. Cabe destacar que puede utilizarse con cualquier tipo de corriente orientada al byte. Particularmente, esta clase permite manejar una tubería de lectura en un subprocesos y una de escritura en otro.

```
import java.io.InputStream;
import java.io.OutputStream;
public class ThreadDeESBytes extends Thread {
      InputStream is = null;
      OutputStream os = null;
      String proceso = null;
      ThreadDeESBytes(String proceso, InputStream is, OutputStream os) {
            this.is = is;
            this.os = os;
            this.proceso = proceso;
      }
      public void run() {
            byte[] buffer = new byte[512];
            int bytes_read;
            try {
                  for (;;) {
                        bytes_read = is.read(buffer);
                        if (bytes_read == -1) {
                              os.close();
                              is.close();
                              return;
                        }
                        os.write(buffer, 0, bytes_read);
                        os.flush();
                        System.out.println("Procesando: " + proceso);
                  }
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

El siguiente código, permite crear dos subprocesos por medio de la clase ThreadDeESBytes e indicarle cual es la corriente de ingreso y cual la de salida a usar. En este caso se utilizará un

archivo de texto para la entrada y otro de salida, con lo que el resultado final es la copia de uno en otro.

```
try {
      PipedInputStream escribirTubo = new PipedInputStream();
      PipedOutputStream leerTubo = new PipedOutputStream(escribirTubo);
      FileOutputStream fos = new FileOutputStream("Tubería.txt");
      FileInputStream fis = new FileInputStream("CantoDeBilbo.txt");
      ThreadDeESBytes tLee = new ThreadDeESBytes("lector", fis, leerTubo);
      ThreadDeESBytes tEscribe=new ThreadDeESBytes("escritor",escribirTubo,fos);
      tLee.start();
      tEscribe.start();
} catch (Exception e) {
      e.printStackTrace();
}
```

Decoración de corrientes de E/S

Es raro que un programa utilice directamente una corriente o un objeto que la maneje. En lugar de esto se encadenan una serie de clases diseñadas para el manejo de corrientes que facilitan su uso.

Corrientes de procesamiento

Las corrientes de procesamiento realizan algún tipo de conversión sobre una corriente previamente definida. Las corrientes de procesamiento también son conocidas como corrientes de “filtrado”. Una corriente de filtrado de ingreso se crea con una conexión a una corriente de ingreso existente. Esto se realiza de manera que cuando se trata de leer de un objeto del tipo de una corriente de ingreso filtrada, esta provee los caracteres que provienen originalmente de otro objeto que maneja la corriente. Fundamentalmente permite convertir datos en bruto en formatos más amigables para las aplicaciones.

Corrientes para filtrado de elementos

Tipo	Corrientes de bytes	Corrientes de Caracteres
Con Buffer	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Filtrado	FilterInputStream FilterOutputStream	FilterReader FilterWriter
Conversiones entre bytes y caracteres		InputStreamReader OuptutStreamWriter
Serialización de Objetos	ObjectInputStream ObjectOutputStream	
Conversiones de datos	DataInputStream DataOutputStream	
Conteo	LineNumberInputStream	LineNumberReader
Observación	PushbackInputStream	PushbackReader

Impresión	PrintStream	PrintWriter

FileInputStream y FileOutputStream

Leen o escriben datos en un archivo del sistema de archivos nativo.

PipedInputStream y PipedOutputStream

Implementan los componentes de entrada y salida de una tubería. Las tuberías se utilizan para canalizar la salida de un programa hacia la entrada de otro. Un PipedInputStream debe ser conectado a un PipedOutputStream y un PipedOutputStream debe ser conectado a un PipedInputStream .

ByteArrayInputStream y ByteArrayOutputStream

Leen o escriben datos en un vector de la memoria.

SequenceInputStream

Concatena varios canales de entrada dentro de un sólo canal de entrada.

StringBufferInputStream

Permite a los programas leer desde un StringBuffer como si fuera un canal de entrada.

DataInputStream y DataOutputStream

Lee o escribe datos primitivos de Java en una máquina independiente del formato.

BufferedInputStream y BufferedOutputStream

Almacena datos mientras los lee o escribe para reducir el número de accesos requeridos a la fuente original. Los canales con buffer son más eficientes que los canales similares sin buffer.

LineNumberInputStream

Tiene en cuenta los números de línea mientras lee.

PushbackInputStream

Un canal de entrada con un buffer de un byte hacia atrás. Algunas veces cuando se leen bytes desde un canal es útil chequear el siguiente caracter para poder decir lo que hacer luego. Si se chequea un carácter del canal, se necesitará volver a ponerlo en su sitio para que pueda ser leído y procesado normalmente.

PrintStream

Un canal de salida con los métodos de impresión convenientes.

Procesando las corrientes como decoradores

Un decorador es un patrón de diseño que permite a un objeto (el decorador) envolver a otro. Algunas llamadas de métodos simplemente se procesan en el objeto “envuelto” derivándole el procesamiento desde la clase que lo envuelve. A esto se lo denomina delegación. Sin embargo, otros métodos son sobrescritos para mejorar su funcionalidad. Este tipo de procesamiento se utiliza mucho en la API de java.io.
Por ejemplo, un BufferedReader se puede utilizar para “decorar” un FileReader , ya que esta última sólo implementa métodos de lectura de bajo nivel y en cambio la otra permite leer líneas enteras en una cadena.

Las clases del tipo FilterXXX proveen la base para heredar y obtener un procesamiento personalizado de una corriente de entrada o salida.
Por ejemplo, se puede escribir un par de clases como RegistroDeCorrienteDeEntrada y RegistroDeCorrienteDeSalida que lea y escriba registros de una base de datos en una corriente. Un programa puede entonces decorar una corriente de entrada de un archivo con un registro de entrada, para leer registros de una base de datos. Notar que los constructores reciben como parámetro una corriente de datos, el cual es el objeto decorado por la clase supuestamente creada.

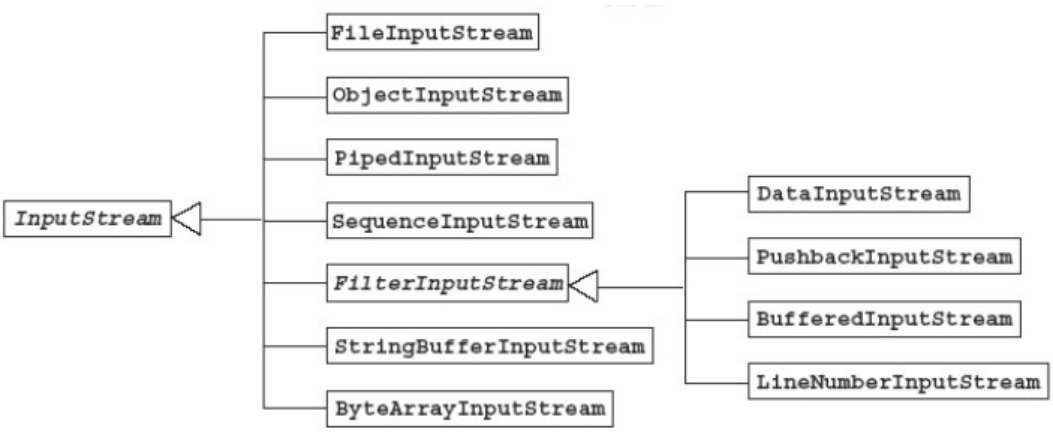
Las clases DataInputSteam y DataOutputStream

Estos filtros de corrientes permiten leer y escribir tipos de datos primitivos de Java y algunos formatos especiales utilizando corrientes. Poseen una serie de métodos para los distintos tipos manejados.
Los tipos primitivos que se pueden leer con una clase tienen su par para escritura en la otra. Los métodos para escribir Strings fueron depreciados y se utilizan con otras clases

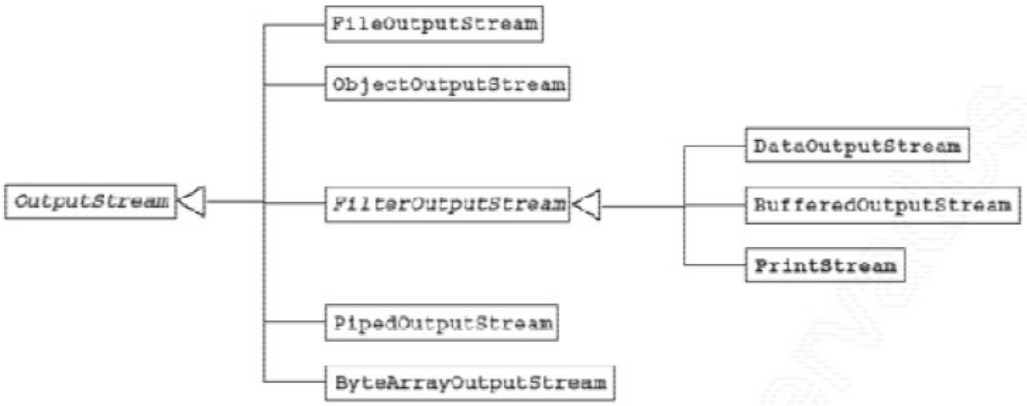
DataInputStream
- byte **readByte**()
- long **readLong**()
- double **readDouble**()

DataOutputStream
- void **writeByte**(byte)
- void **writeLong**(long)
- void **writeDouble**(double)

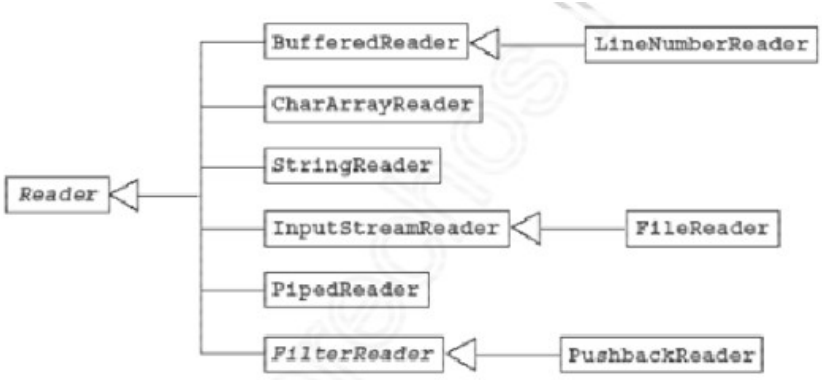
Cadenas de herencia para InputStream



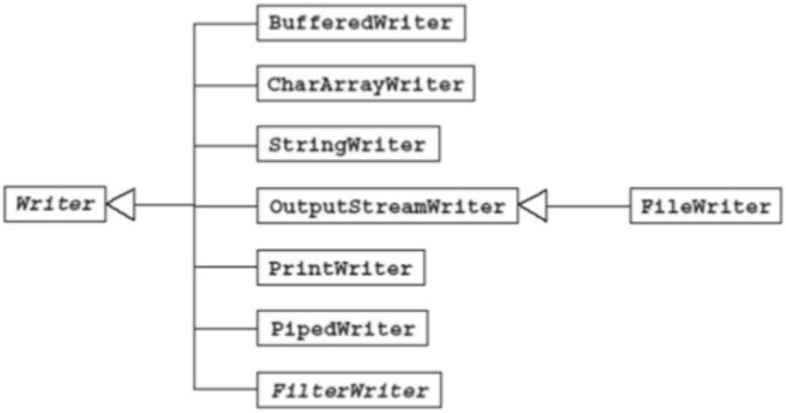
Cadenas de herencia para OutputStream



Cadenas de herencia para Reader



Cadenas de herencia para Writer



Acceso Aleatorio a Archivos

Para lectura y escritura se pueden utilizar todos los mismos métodos que existen para DataInputStream y DataOutputStream de la misma forma que se utilizaron para archivos secuenciales.
Los métodos novedosos son aquellos que permiten moverse dentro de un archivo como los siguientes:

long getFilePointer(): Sirve para tener información acerca de la posición actual del puntero del archivo.

void seek(long pos): Ubica el puntero del archive en una posición absoluta especificada en el argumento.

long length(): Retorna el largo del archivo hasta la marca de finalización del mismo.

Por otra parte, si en un archivo de acceso aleatorio se mueve su puntero al final del mismo, toda información que se escriba se agregará cambiando su tamaño (modo “append”).

Serialización de objetos

Las máquinas virtuales modernas de Java soportan serializar información en una corriente. Sólo son serializados los datos de los objetos (esto es, las variables de instancia). Cuando una clase es serializable, se debe declarar explícitamente aquellas variables de instancia que no se quieran serializar con la palabra clave transient .
La serialización se utiliza para almacenar el estado de un objeto en una corriente, generalmente un archivo en disco, y a este acto se lo llama persistencia. Por lo tanto, un objeto es “capaz de ser persistente” cuando se lo puede almacenar en un medio masivo.

Cuando un objeto es serializado, sólo los datos se preservan y ni los métodos ni los constructores son parte de la corriente de serialización. Cuando una variable de instancia es un objeto, las variables de instancia de dicho objeto también son serializadas y son parte de la corriente de serialización.
Los modificadores de tipo (private , protected , public o por defecto) no tienen efecto sobre los datos que van a ser serializados. Los datos son escritos en la corriente con un formato de composición de bytes y cadenas representados como caracteres UTF (Unicode Transformation Format).
Existe otro uso común de la serialización: cuando la corriente que se utiliza va, por ejemplo, a un puerto serial u otro dispositivo de entrada y salida de comunicaciones. En estos casos el concepto es válido de la misma manera que si se guardarán los datos en un medio de almacenamiento masivo. En el otro extremo de la comunicación se recibe la información serializada y con un proceso inverso se puede acceder a estos objetos.
Ciertos tipos de objetos, como los subprocesos o los tipo File no se pueden serializar cuando están asociados a un objeto que si puede, por lo tanto deben declararse obligatoriamente como transient para que no se genere una excepción del tipo NotSerializableException , ya que este tipo de objetos sólo tienen sentido cuando se encuentran en la memoria de la JVM que los crea.

```
//Ejemplo de Serialización
Date d = new Date();
try {
      FileOutputStream f = new FileOutputStream("fecha.ser");
      ObjectOutputStream s = new ObjectOutputStream(f);
      s.writeObject(d);
      s.close();
} catch (IOException e) {
      e.printStackTrace();
}

//Ejemplo de DesSerialización
Date d = null;
try {
      FileInputStream f = new FileInputStream("fecha.ser");
      ObjectInputStream s = new ObjectInputStream(f);
      d = (Date) s.readObject();
      s.close();
```

```
} catch (Exception e) {
      e.printStackTrace();
}
System. out .println("Fecha: " + d);
```

Uso del API Stream en archivos de texto

Para mostrar un ejemplo de uso del api Stream con archivos de texto vamos a generar el siguiente archivo:

```
try (BufferedWriter out=new BufferedWriter(new FileWriter("texto.txt"))) {
      out.write("Lunes.\n");
      out.write("Martes.\n");
      out.write("Martes.\n");
      out.write("Miércoles.\n");
      out.write("Jueves.\n");
      out.write("Viernes.\n");
      out.write("Sábado.\n");
      out.write("Domingo.\n");
  } catch (IOException ex) { ex.printStackTrace(); }
```

Ahora vamos a consultarlo usando BufferedReader y el API Stream

```
try (BufferedReader in=new BufferedReader(new FileReader("texto.txt"))) {
      Stream<String> stream=in.lines();

      //Descomentarear la sentencia que se desea usar.
      //Las funciones de Strea automaticamente autoClosean
      //por esa razon solo podemos usar una por ambito.

      System.out.println("Cantidad de filas :"+stream.count());

      //System.out.println("Listado sin duplicados.");
      //stream.distinct().forEach(System.out::println);

      //System.out.println("Los primeros 5 Elementos.");
      //stream.limit(5).forEach(System.out::println);

      //System.out.println("Listado Ordenado");
      //stream.sorted().forEach(System.out::println);

      //System.out.println("Filas que inician con M");
      //stream.filter(p->p.startsWith("M")).forEach(System.out::println);

      //System.out.println("Filas con mas de 8 caracteres");
      //stream.filter(p->p.length()>=8).forEach(System.out::println);

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

Uso del API Stream en archivos con objetos Serializados

Para mostrar un ejemplo de uso del api Stream con archivos de serializados vamos a generar la siguiente clase y archivo:

```
class Persona implements Comparable<Persona>, Serializable{
      private int id;
      private String nombre;
      private int edad;
```

```
        private String estadoCivil;
        public Persona(int id, String nombre, int edad,String estadoCivil) {
            this.id = id;
            this.nombre = nombre;
            this.edad = edad;
            this.estadoCivil = estadoCivil;
        }
        @Override public String toString() {
            return "Persona{ nombre=" + nombre + ", edad=" + edad + '}';
        }
        public int getId()                      { return id;           }
        public void setId(int id)               { this.id = id;        }
        public String getNombre()               { return nombre;       }
        public void setNombre(String nombre)    { this.nombre = nombre; }
        public int getEdad()                    { return edad;         }
        public void setEdad(int edad)           { this.edad = edad;    }
        public String getEstadoCivil()          { return estadoCivil;  }
        public void setEstadoCivil(String estadoCivil){
            this.estadoCivil=estadoCivil;
        }
        @Override public int compareTo(Persona p) {
            return this.toString().compareTo(p.toString());
        }
    }

    try ( ObjectOutputStream out=new ObjectOutputStream(new
        FileOutputStream("datos.dat")) ){
        out.writeObject(new Persona(1,"Ana",32,"Soltero"));
        out.writeObject(new Persona(2,"Javier",41,"Casado"));
        out.writeObject(new Persona(3,"Carlos",22,"Viudo"));
        out.writeObject(new Persona(4,"Estela",55,"Casado"));
        out.writeObject(new Persona(5,"Raul",27,"Soltero"));
    } catch (Exception e ){ e.printStackTrace(); }
```

Ahora vamos a consultarlo usando ObjectOutputStream y el API Stream

```
try ( ObjectInputStream in=new ObjectInputStream(
      new FileInputStream("datos.dat")) ){
            List<Persona>lista=new ArrayList();
            try{ while(true) { lista.add((Persona)in.readObject()); } }
                        catch(EOFException e) {}

                System.out.println("Listado de personas: ");
                lista
                    .stream()
                    .forEach(System.out::println);

                System.out.println("Listado de personas ordenado por edad:");
                lista
                    .stream()
                    .sorted(Comparator.comparingInt(Persona::getEdad))
                    .forEach(System.out::println);

                System.out.println("Listado de nombres: ");
                lista
                        .stream()
                        .map(Persona::getNombre)
                        .forEach(System.out::println);


                System.out.println("Listado de nombres de personas menores de
                        30: ");
                lista
                    .stream()
```

```
    .filter(p -> p.getEdad()<30)
    .map(Persona::getNombre)
    .forEach(System.out::println);


System.out.println("Listado de nombres de personas menores de 30
        ordenado por edad: ");
lista
    .stream()
    .filter(p -> p.getEdad()<30)
    .sorted(Comparator.comparingInt(Persona::getEdad))
    .map(Persona::getNombre)
    .forEach(System.out::println);

System.out.println("Listado de nombres de personas menores de 30
        ordenado por edad desc: ");
lista
    .stream()
    .filter(p -> p.getEdad()<30)
    .sorted(Comparator.comparingInt(Persona::getEdad).reversed()
            )
    .map(Persona::getNombre)
    .forEach(System.out::println);

System.out.println("Listado agrupado por estado civil y
        cantidades:");
lista
    .stream()
    .collect(
        Collectors.groupingBy(
                Persona::getEstadoCivil, Collectors.counting()
        )
    )
        .forEach((s, c) -> System.out.printf("Estado Civil: %s:
                    cantidad: %s \n", s,c));

System.out.println("Listado agrupado por estado civil y
                    cantidades de menores de 30:");
lista
    .stream()
    .filter(p -> p.getEdad() < 30)
    .collect(
        Collectors.groupingBy(
                Persona::getEstadoCivil, Collectors.counting()
        )
    )
.forEach((s, c) -> System.out.printf("Estado Civil: %s:
        cantidad: %s \n", s,c));


System.out.println("Listado agrupado por estado civil y suma de
        edades de menores de 30:");
lista
    .stream()
    .collect(
        Collectors.groupingBy(
                Persona::getEstadoCivil,
                Collectors.summingInt(Persona::getEdad)
        )
)
.forEach((s, c) -> System.out.printf("Estado Civil: %s:
        cantidad: %s \n", s,c));

System.out.println("Similar al Having de SQL.");
```

```
            lista
                .stream()
                .collect(
                    Collectors.groupingBy(
                            Persona::getEstadoCivil,
                            Collectors.summingInt(Persona::getEdad)
                    )
                )
                .entrySet()
                .stream() //volvemos a generar un stream
                .filter(p -> p.getValue() > 50) //filtramos (simula el
                                                //having)
                .collect(Collectors.toList())
                .forEach(list -> System.out.printf("Estado Civil: %s:
                   cantidad: %s \n", list.getKey(),list.getValue()));


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

Extensión del manejo de excepciones

Algunos recursos en Java deben ser cerrados de forma manual, como InputStream , Writer , etc... , lo cual implica llamar a un método específico de cada objeto para realiza un cierre limpio y no depender del recolector de basura para liberar los recursos tomados. Esto se ve claramente en las operaciones de entrada y salida, razón por la cual este tema se explica en este capítulo.
Esta característica nueva del lenguaje permite que la sentencia try defina por sí misma una visibilidad en el bloque a la que está asociada. La consecuencia directa es que si se define dentro del espacio de su visibilidad un recurso determinado, cuando el bloque finaliza, el mismo se libera automáticamente. Estos recursos están en el ámbito del bloque try y se cierran sin necesidad de llamar explícitamente a ningún método de cierre.

```
Try (
      FileOutputStream f = new FileOutputStream("fecha.ser");
      ObjectOutputStream s = new ObjectOutputStream(f);
) {
      s.writeObject(d);
} catch (IOException e) {
      e.printStackTrace();
}
```

Notar que el bloque try ahora incluye paréntesis y en su interior se declaran los objetos que toman recursos de salida en una corriente a disco. Notar además que se omite la llamada al método de cierre de la corriente porque el mismo ya no es necesario, ya que cuando finalice el bloque de sentencias asociadas a la instrucción, estos se liberarán automáticamente.