Unidad 5

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 9

Corrientes de E / S

Corrientes de E / S

En Este Capítulo

- Argumentos desde la línea de comando
- Propiedades del sistema
- Fundamentos de las E/S
- Salidas por consola
- Escribiendo en el estándar output
- Leyendo del estándar input
- ¿Qué es una ruta? (Y otros datos del sistema de archivos)
- Creación de un objeto del tipo File
- E/S de Corrientes de Archivos
- Clases básicas previas a la versión 7 para el manejo de corrientes
- Corrientes clásicas
- Corrientes Nodales
- Decoración de corrientes de E/S
- Creando Archivos de Acceso Aleatorio
- Serialización
- Extensión del manejo de excepciones

Argumentos desde la línea de comando

Una aplicación puede recibir valores en formato de caracteres desde el sistema operativo. Estos valores se los denomina "argumentos desde la línea de comandos", puesto que es el interprete de comandos el encargado de pasarlos a la aplicación.

En el caso de Java, es la máquina virtual la que interacciona con el sistema operativo, sin embargo, a través de ella se pueden recibir dichos argumentos porque el espacio virtual de ejecución que genera emula esta interacción.

Los puede utilizar cualquier aplicación Java siempre y cuando respete el formato definido para pasar los mismos.

Cada argumento de la línea de comando se coloca en el vector args y es pasado a main:

```
public static void main(String[] args)
```

Los argumentos, se colocan en la línea de comando cuando se invoca al intérprete luego del nombre de la clase:

```
java TestArgs arg1 arg2 "otro arg"
```

Ejemplo

```
package consola;

public class VerificarArgumentos {
	public static void main(String[] args) {
		for (int i = 0; i < args.length; i++) {
			System.out.println("El argumento[" + i + "] es: " + args[i]);
		}
	}
}
```

Se obtiene la salida:

```
El argumento[0] es: arg1
El argumento[1] es: arg2
El argumento[2] es: otro arg
```

Propiedades del sistema

Como la máquina virtual genera su propio espacio de ejecución para las aplicaciones, existen una serie de valores de entorno para dicho ambiente. La forma de sustituir los valores de entorno que se pueden definir para una aplicación en un sistema operativo, Java las maneja como una serie de propiedades que pueden ser consultadas.

A estas propiedades se las denomina propiedades del sistema y es un concepto que se utiliza para remplazar las variables de entorno (las cuales son específicas de la plataforma que las define).

Como todo en Java, los valores se almacenarán en un objeto, el cual es del tipo Properties. Este objeto lo define la máquina virtual al arrancar y se puede obtener una referencia por el método System.getProperties(), el cual retorna un objeto de este tipo con los valores antes mencionados.

Una vez obtenida la referencia al objeto que almacena los valores, se puede utilizar un método de servicio, el cual está definido en este para recuperar el valor asociado al nombre de una propiedad que le sea especificado como argumento. El método getProperty() retorna un String el cual representa el valor de la propiedad cuyo nombre se especifica.

Se puede utilizar también la opción -D al ejecutar un programa para incluir una nueva propiedad, como si esta fuese del sistema, y su valor se puede recuperar como el de cualquier otra propiedad. El siguiente ejemplo muestra el código que realiza la acción antes descripta. Notar como se recupera primero el nombre de la propiedad para luego utilizarlo para saber su valor.

Ejemplo

```
package propiedades;

import java.util.Properties;
import java.util.Enumeration;

public class PruebaPropiedades {
	public static void main(String[] args) {
		Properties props = System.getProperties();
		Enumeration<?> prop_names = props.propertyNames();
		while (prop_names.hasMoreElements()) {
			String prop_name = (String) prop_names.nextElement();
			String property = props.getProperty(prop_name);
			System.out.println("propiedad '" + prop_name + "' es '" +
					property + "'");
		}
	}
}
```

Se obtiene la salida:

```
propiedad 'java.runtime.name' es 'Java(TM) SE Runtime Environment'
propiedad 'sun.boot.library.path' es 'C:\Program Files\Java\jre7\bin'
propiedad 'java.vm.version' es '21.0-b17'
propiedad 'user.country.format' es 'AR'
propiedad 'java.vm.vendor' es 'Oracle Corporation'
propiedad 'java.vendor.url' es 'http://java.oracle.com/'
propiedad 'path.separator' es ';'
propiedad 'java.vm.name' es 'Java HotSpot(TM) 64-Bit Server VM'
propiedad 'file.encoding.pkg' es 'sun.io'
propiedad 'user.script' es ''
propiedad 'user.country' es 'US'
propiedad 'sun.java.launcher' es 'SUN_STANDARD'
propiedad 'sun.os.patch.level' es 'Service Pack 1'
propiedad 'java.vm.specification.name' es 'Java Virtual Machine Specification'
```

```
propiedad 'user.dir' es 'C:\Educación\Java\Carrera básica - Desarrollador
Java\Versión 2\Trabajo\Programación en
Java\Acomodo\Ejemplos\Eclipse\Módulo9\Modulo9\EntradasSalidas'
propiedad 'java.runtime.version' es '1.7.0-b147'
propiedad 'java.awt.graphicsenv' es 'sun.awt.Win32GraphicsEnvironment'
propiedad 'java.endorsed.dirs' es 'C:\Program Files\Java\jre7\lib\endorsed'
propiedad 'os.arch' es 'amd64'
propiedad 'java.io.tmpdir' es 'C:\Users\Marcelo\AppData\Local\Temp\'
propiedad 'line.separator' es '
'
propiedad 'java.vm.specification.vendor' es 'Oracle Corporation'
propiedad 'user.variant' es ''
propiedad 'os.name' es 'Windows 7'
propiedad 'sun.jnu.encoding' es 'Cp1252'
propiedad 'java.library.path' es 'C:\Program
Files\Java\jre7\bin;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:\Pr
ogram Files\Common Files\Microsoft Shared\Windows Live;C:\Program Files
(x86)\Common Files\Microsoft Shared\Windows Live;C:\Program Files (x86)\NVIDIA
Corporation\PhysX\Common;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem
;C:\Windows\System32\WindowsPowerShell\v1.0\;c:\Program Files (x86)\Microsoft
SQL Server\100\Tools\Binn\;c:\Program Files\Microsoft SQL
Server\100\Tools\Binn\;c:\Program Files\Microsoft SQL
Server\100\DTS\Binn\;C:\Program Files (x86)\VisualSVN\bin;C:\Program
Files\TortoiseSVN\bin;C:\Program Files (x86)\Microsoft SQL
Server\100\Tools\Binn\VSShell\Common7\IDE\;C:\Program Files (x86)\Microsoft SQL
Server\100\DTS\Binn\;C:\Program Files (x86)\Microsoft Visual Studio
9.0\Common7\IDE\PrivateAssemblies\;C:\Program Files (x86)\WinMerge;C:\Program
Files\Java\jdk1.7.0\bin;C:\Program Files (x86)\QuickTime\QTSystem\;C:\Program
Files (x86)\Windows Live\Shared;C:\Sun\AppServer\bin;.'
propiedad 'java.specification.name' es 'Java Platform API Specification'
propiedad 'java.class.version' es '51.0'
propiedad 'sun.management.compiler' es 'HotSpot 64-Bit Tiered Compilers'
propiedad 'os.version' es '6.1'
propiedad 'user.home' es 'C:\Users\Marcelo'
propiedad 'user.timezone' es ''
propiedad 'java.awt.printerjob' es 'sun.awt.windows.WPrinterJob'
propiedad 'file.encoding' es 'Cp1252'
propiedad 'java.specification.version' es '1.7'
propiedad 'user.name' es 'Marcelo'
propiedad 'java.class.path' es 'C:\Educación\Java\Carrera básica - Desarrollador
Java\Versión 2\Trabajo\Programación en
Java\Acomodo\Ejemplos\Eclipse\Módulo9\Modulo9\EntradasSalidas\bin'
propiedad 'java.vm.specification.version' es '1.7'
propiedad 'sun.arch.data.model' es '64'
propiedad 'java.home' es 'C:\Program Files\Java\jre7'
propiedad 'sun.java.command' es 'propiedades.PruebaPropiedades'
propiedad 'java.specification.vendor' es 'Oracle Corporation'
propiedad 'user.language' es 'en'
propiedad 'user.language.format' es 'es'
propiedad 'awt.toolkit' es 'sun.awt.windows.WToolkit'
propiedad 'java.vm.info' es 'mixed mode'
propiedad 'java.version' es '1.7.0'
propiedad 'java.ext.dirs' es 'C:\Program
Files\Java\jre7\lib\ext;C:\Windows\Sun\Java\lib\ext'
```

```
propiedad 'sun.boot.class.path' es 'C:\Program
Files\Java\jre7\lib\resources.jar;C:\Program
Files\Java\jre7\lib\rt.jar;C:\Program
Files\Java\jre7\lib\sunrsasign.jar;C:\Program
Files\Java\jre7\lib\jsse.jar;C:\Program Files\Java\jre7\lib\jce.jar;C:\Program
Files\Java\jre7\lib\charsets.jar;C:\Program Files\Java\jre7\classes'
propiedad 'java.vendor' es 'Oracle Corporation'
propiedad 'file.separator' es '\'
propiedad 'java.vendor.url.bug' es 'http://bugreport.sun.com/bugreport/'
propiedad 'sun.cpu.endian' es 'little'
propiedad 'sun.io.unicode.encoding' es 'UnicodeLittle'
propiedad 'sun.desktop' es 'windows'
propiedad 'sun.cpu.isalist' es 'amd64'
```

La clase Properties implementa un mapa de nombres respecto de sus valores (un mapa de String a String) donde se debe interpretar el primer String como una clave y el segundo como el valor asociado a dicha clave.

El método propertyNames() retorna una Enumeration (enumeración) de todos los nombres de propiedades. Este tipo de objeto sirve para recorrer las propiedades y obtener sus valores.

El método getProperty() retorna un String representando el valor que corresponde al nombre indicado en el argumento del método.

Una colección de propiedades se puede leer y escribir a un archivo utilizando load y store para recuperar de esta manera la información o “persistirla”.

Fundamentos de las E / S

Una corriente 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 threads o procesos

Salidas por 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)
- La salida estándar (Standard Output)
- La corriente de errores estándar (Standard Error)

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).

En Java existen objetos que permiten manejarlas y según su tipo se asocian a los dispositivos por defecto que las definen, como ser:

1. System.out permite escribir en la "standard output" (salida estándar).
    a. Es un objeto del tipo PrintStream (asociado a la salida por consola).
2. System.in permite leer de la "standard input" (entrada estándar).
    a. Es un objeto del tipo InputStream (asociado al ingreso por consola).
3. System.err permite escribir en la "standard error" (error estándar).
    a. Es un objeto del tipo PrintStream(asociado a la salida por consola).

Cada una de las corrientes pueden manejarse como archivos, es decir, se las puede abrir, escribir, leer y cerrar según sea el caso y la posibilidad de hacerlo (no se puede escribir en el ingreso estándar, pero si se puede leer de él).

Escribiendo en el estándar output

Java define una serie de clases que permiten un cómodo manejo unificado de todas las corrientes en un sistema operativo (entre las que se encuentran las estándar y las propias de los archivos en disco, canales de comunicación, etc…).

Para llevar a cabo esa unificación, se definen una serie de clases que reciben como parámetros a las corrientes, por ejemplo, y definen métodos en su interfaz que permiten un mejor manejo de las mismas. Esta funcionalidad a nivel de diseño se la denomina "decorador" y es un patrón de diseño conocido.

Así, a través de los decoradores, se pueden utilizar métodos conocidos para escribir o leer de una corriente (los detalles de esta operatoria se explicarán posteriormente). Se puede definir entonces la funcionalidad de algunos métodos conocidos que permiten interaccionar con las corrientes:

- Los métodos println escriben el argumento y una nueva línea.
- Los métodos print imprimen el argumento sin la nueva línea

Ambos métodos están sobrecargados para todos los tipos primitivos y además para char[], Object y String.

Los métodos print(Object) y println(Object) llaman al método toString() del argumento para permitir al programador que coloque de la forma que quiera un mensaje que identifique al objeto tan sólo sobrescribiendo toString.

Leyendo del estándar input

Para demostrar la interacción con la corriente estándar, el siguiente código lee los ingresos por teclado hasta que se cierra la corriente con un carácter de fin de archivo.

Ejemplo

```
package corrientes;
import java.io.*;

public class IngresoPorTeclado {
	public static void main(String args[]) {
		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.

El siguiente programa no crea ningún archivo porque no se especifica ningún método que produzca dicho efecto.

Ejemplo

```
package archivos;
import java.io.File;

public class ArchivosAnteriores1 {

	public static void main(String[] args) {
		File miArchivo;
		miArchivo = new File("miArchivo.txt");
		File miArchivo2 = new File("MisDocs", "miArchivo.txt");
	}
}
```

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. El siguiente ejemplo muestra como crear un archivo, pero cuando se intenta crearlo en un directorio llamado MisDocs, se produce un error de E/S si el directorio no existe-

Ejemplo

```
package archivos;
import java.io.File;
import java.io.IOException;

public class ArchivosAnteriores2 {

	public static void main(String[] args) {
		File miArchivo;
		miArchivo = new File("miArchivo.txt");
		File miArchivo2 = new File("MisDocs", "miArchivo.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();
                  }
      }

}
```

Al ejecutar el programa se produce la siguiente salida (suponiendo que el directorio no existe)

```
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)
```

Como se puede apreciar, el manejo de directorios y archivos puede ser un poco confuso.

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 archive esta en un directorio, retorna el nombre. Si no puede resolver el nombre del directorio retorna un nulo.
- **boolean** renameTo(File newName)
  - Ronombra 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 esta siendo representado por el que esta 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

Para entender mejor el proceso de lectura de un archivo de disco se presenta un ejemplo de lectura con almacenamiento intermedio (buffer).

Ejemplo

```
package archivos;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class LeeArchivo {
	public static void main(String[] args) {
		File file = new File(args[0]);
		try { // Crear un lector
			// con buffer para leer cada línea del archivo
			BufferedReader in = new BufferedReader(new FileReader(file));
			String s;
			s = in.readLine();
			// leer cada línea del archivo y mostrarlo en pantalla
			while (s != null) {
				System.out.println(" Leyó : " + s);
				s = in.readLine();
			}
			in.close();
			// Cerrar el lector con buffer, que también
			// cierra el lector de archivo
		} catch (FileNotFoundException e1) {
			// Si no existe el archivo
			System.err.println("Archivo no encontrado : " + file);
		} catch (IOException e2) {
			e2.printStackTrace();//Para cualquier otra excepción de E/S
		}
	  }
	}
```


Ejemplo de Salida

Análogamente al caso anterior, para comprender el proceso de salida a un archivo en disco, se presenta el siguiente código.

Ejemplo

```
package archivos;
import java.io.BufferedReader;
```

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class EscribeArchivo {
	public static void main(String[] args) {
		File file = new File(args[0]); // 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();
		}
	}
}
```


Clases básicas previas a la versión 7 para el manejo de corrientes

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.

Corrientes clásicas

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 esta 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).
- **Lector y Escritor**: se refieren a corrientes de caracteres (Unicode).

	Corrientes de byte	Corrientes de Caracteres
Corrientes Fuente	InputStream	Reader
Corrientes Receptoras	OutputStream	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 el los sistemas operativos tipo Unix solo es un caracter
- `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 se 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 esta listo para leer datos
- **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 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 string)
  - Escribe el String string en la corriente de salida
- **void** write(String string, **int** desplazamiento, **int** longitud)
  - Escribe el String string 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 se 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 strings) y tuberías (pipes: un canal de comunicación entre dos procesos o threads). Es posible crear nuevas clases nodales de corrientes, pero requiere lidiar con llamados a métodos nativos (JNI – Java Native Interface) y esto es, evidentemente, no portable.

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

Corrientes de bytes

FileInputStream

Esta clase se utiliza para leer bytes desde un archivo en el sistema de archivos de la máquina local: Esta clase 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 esta 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 thread y los bytes de datos son escritos a la correspondiente PipedOutputStream por otro thread. Intentar utilizar ambos tipos de objetos sobre un mismo thread 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 thread y los bytes de datos son leídos de la correspondiente PipedInputStream al que esta conectado por otro thread. Intentar utilizar ambos tipos de objetos sobre un mismo thread 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 esta 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 string, 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

Corrientes sin buffer

El siguiente ejemplo lee caracteres de un archivo cuyo nombre se pasa como primer argumento por línea de comandos y escribe dichos caracteres en el archivo de salida indicado como segundo argumento por línea de comandos, con lo cual, como resultado final, se obtiene una copia del primer archivo en el segundo.

Ejemplo

```
package archivos;
import java.io.*;

public class VerificaCorrientesNodales {
	public static void main(String[] args) {
		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

Este ejemplo es básicamente igual al anterior. La diferencia fundamental radica en 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().

Ejemplo

```
package archivos;
import java.io.*;

public class VerificaCorrientesConBuffer {
  public static void main(String[] args) {
    try {
      FileReader     entrada     = new FileReader(args[0]);
      BufferedReader bufEntrada  = new BufferedReader(entrada);
      FileWriter     salida      = new FileWriter(args[1]);
      BufferedWriter bufSalida = new BufferedWriter(salida);
      String line;

      // leer la primera línea
      line = bufEntrada.readLine();

      while ( line != null ) {
       // escribir la linea en el archivo de salida
       bufSalida.write(line, 0, line.length());
       bufSalida.newLine();

       // leer la próxima línea
       line = bufEntrada.readLine();
      }

      bufEntrada.close();
      bufSalida.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Tuberías

Como se mencionó anteriormente, las clases de tuberías están diseñadas para trabajar en conjunto y en threads o procesos 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 thread), 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 thread y una de escritura en otro.

Ejemplo

```
package archivos;
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();
		} finally {
		}
	}
}
```

La clase VerificaTuberiasBytes permite crear dos threads por medio de la clase ThreadDeESBytes e indicarle cual es la corriente de ingreso y salida a utilizar. 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.

Ejemplo

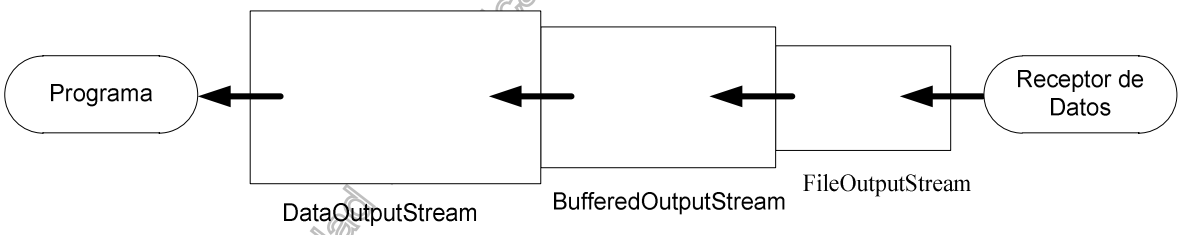
```
package archivos;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class VerificaTuberiasBytes {

	public static void main(String[] args) {
		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. La siguiente figura muestra un ejemplo para una corriente de ingreso en la cual se quiere leer un archivo, donde el mismo se leerá con buffer y su contenido se convertirá en datos primitivos de Java para su procesamiento

Análogamente, para la salida se realiza el mismo tipo de proceso pero a la inversa, con lo que se genera una cadena para una corriente de salida



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. Esto fundamentalmente permite convertir datos en bruto en formatos más amigables para las aplicaciones.

Tipo	Corrientes de bytes	Corrientes de Caracteres
Con Buffer	BufferedInputStream	BufferedReader

	BufferedOutputStream	BufferedWriter
Filtrado	FilterInputStream	FilterReader
	FilterOutputStream	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

FilterInputStream y FilterOutputStream, por ejemplo, son subclases de InputStream y OutputStream, respectivamente, y también son clases abstractas. Estas clases definen la interfaz para los canales filtrados que procesan los datos que están siendo leídos o escritos. Por ejemplo, los canales filtrados BufferedInputStream y BufferedOutputStream almacenan datos mientras los leen o escriben para aumentar su velocidad.

Lo siguiente es una introducción a las clases no abstractas que descienden directamente desde InputStream y OutputStream y que conforman corrientes de procesamiento.

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

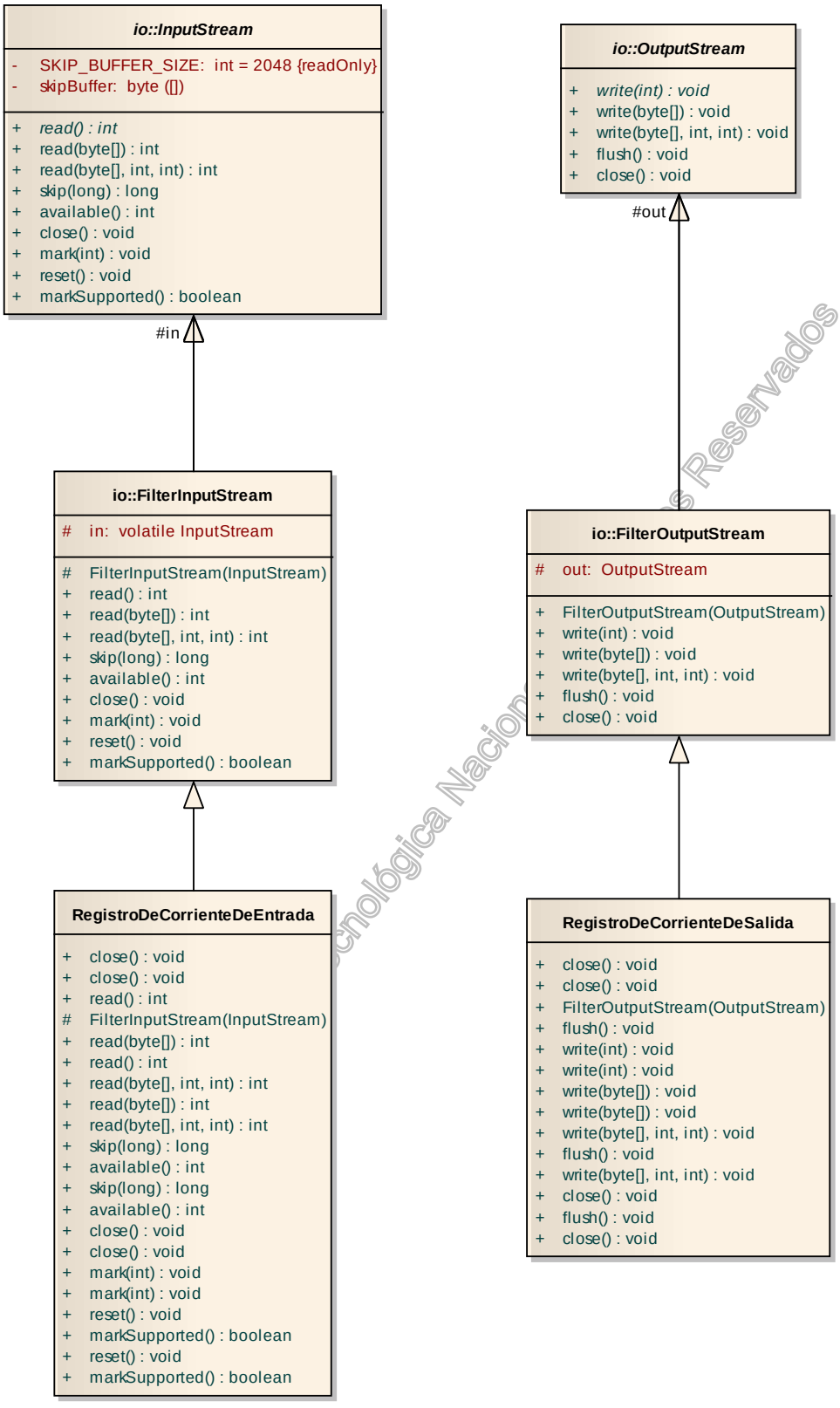
Un decorador es un patrón de diseño que permite a un objeto (el decorador) envolver a otro. Algunas llamadas de métodos simplemente se procesan en el objeto "envuelto" derivándole el procesamiento desde la clase que lo envuelve. A esto se lo denomina delegación. Sin embargo, otros métodos son sobrescritos 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

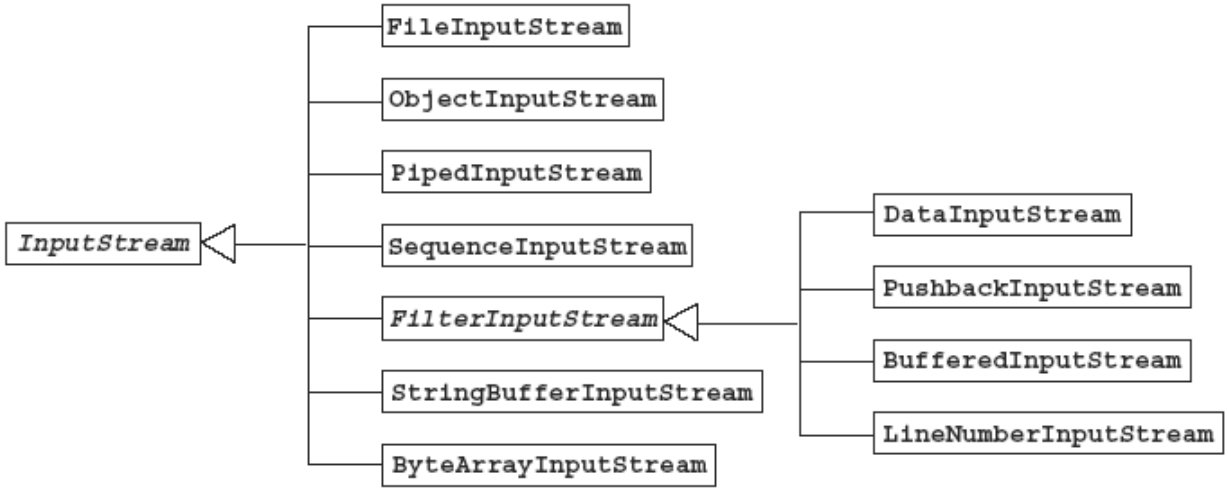
- `byte` readByte()
- `long` readLong()
- `double` readDouble()

DataOutputStream

- `void` writeByte(`byte`)
- `void` writeLong(`long`)
- `void` writeDouble(`double`)

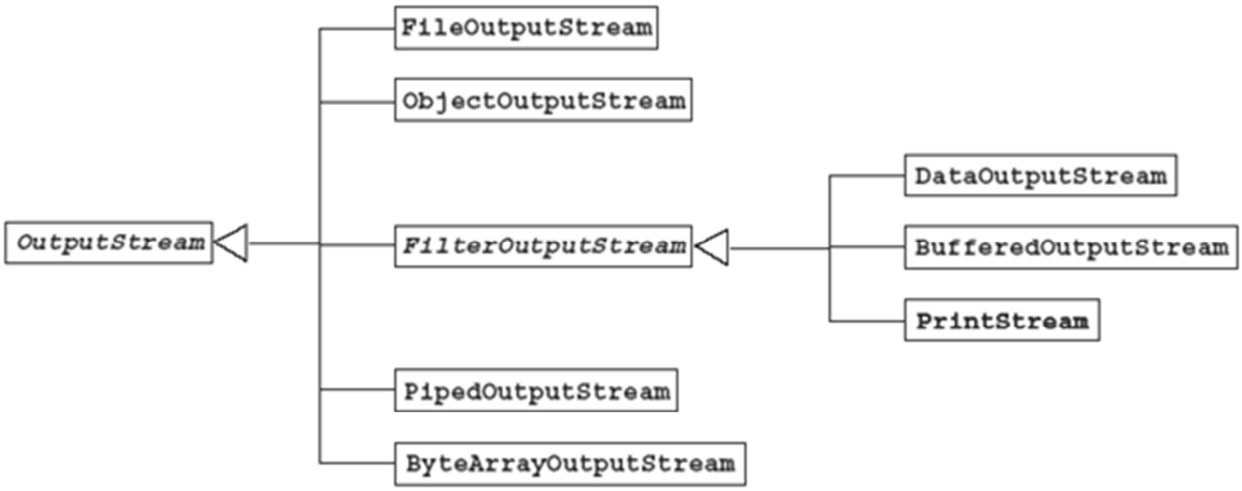
Cadenas de herencia para InputStream

Es conveniente entender bien las cadenas de herencia para saber que objetos utilizar y como se pueden asignar referencias, por ejemplo, para el caso de InputStream el gráfico muestra como están armadas.



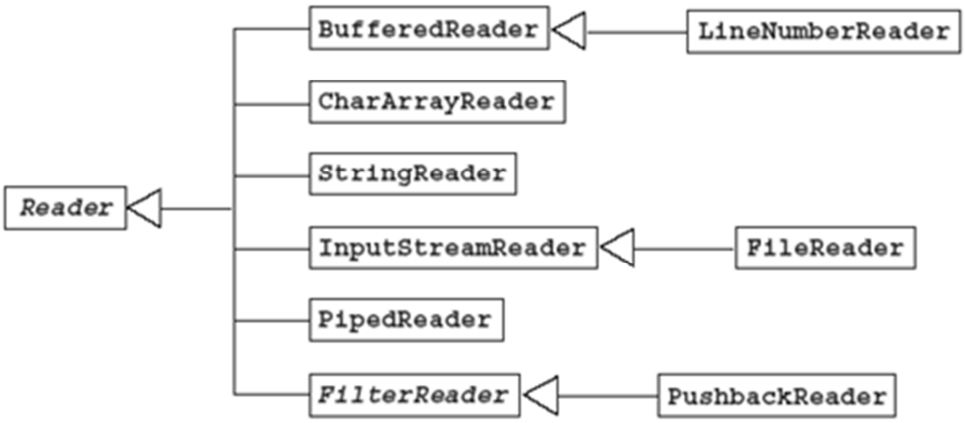
Cadenas de herencia para OutputStream

El siguiente gráfico muestra como están armadas las cadenas de herencia para OutputStream



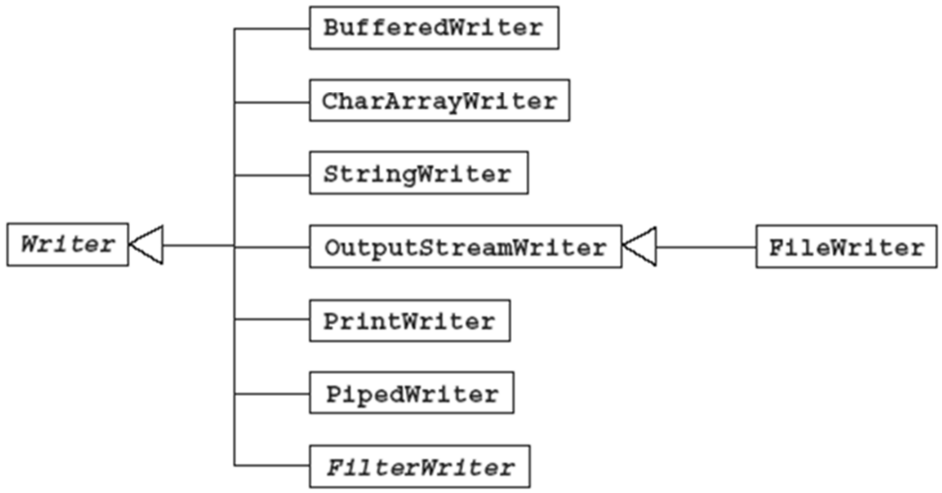
Cadenas de herencia para Reader

El siguiente gráfico muestra como están armadas las cadenas de herencia para Reader.



Cadenas de herencia para Writer

El siguiente gráfico muestra como están armadas las cadenas de herencia para Writer.



Creando Archivos de Acceso Aleatorio

En muchas oportunidades se desea tener acceso a un archivo o a parte de él sin recorrerlo desde el principio al final secuencialmente. Java provee una clase, RandomAccesFile para manejar este tipo de entradas y salidas.

Los posibles constructores son:

- Con un nombre de archivo:
  - miArchivoDeAccAl=**new** RandomAccesFile (String nombre, String modo)
- Con un objeto del tipo File
  - miArchivoDeAccAl=**new** RandomAccesFile (File nombre, String modo)

El modo de apertura indica si es de sólo lectura (“r”) o de lectura y escritura (“rw”). Por ejemplo, para actualizar un archivo:

```
miArchivoDeAccAl = new RandomAccesFile (“Prueba.txt”, “rw”)
```

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

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 threads o los tipo File no se pueden serializar cuando están asociados a un objeto que sí 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.

El siguiente es un ejemplo de cómo serializar un objeto de tipo fecha en el disco de la máquina local. Notar que se utiliza para esto el método writeObject de ObjectOutputStream. Esto permite escribir un objeto en la corriente asociada. Toda la operación se puede llevar a cabo porque la clase Date implementa la Interfaz Serializable, la cual sirve de marca para que la máquina virtual sepa que los objetos del tipo de esa clase se pueden persistir.

Ejemplo

```
package serializacion;
import java.io.*;
```

```
import java.util.Date;

public class SerializaFecha {

	SerializaFecha() {
		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();
		}
	}

	public static void main(String args[]) {
		new SerializaFecha();
	}
}
```

El proceso para leer un objeto que fue persistido en un archivo es a través del método readObject de la clase ObjectInputStream.

Ejemplo

```
package serializacion;

import java.io.*;
import java.util.Date;

public class DeSerializaFecha {

	DeSerializaFecha() {
		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(
			"Desearializa un objeto de tipo Date desde fecha.ser");
		System.out.println("Fecha: " + d);
	}

	public static void main(String args[]) {
		new DeSerializaFecha();
	}
}
```

Java 7

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 módulo.

Esta característica nueva del lenguaje permite que la sentencia **try** defina por sí misma una visibilidad en el bloque a la que esta 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. Esto se puede demostrar fácilmente rescribiendo el ejemplo anterior.

Ejemplo

```
package Excepciones;

import java.io.*;
import java.util.Date;

public class SerializaFecha {
	SerializaFecha() {
		Date d = new Date();
		try (
				FileOutputStream f = new
					FileOutputStream("fecha.ser");
				ObjectOutputStream s = new ObjectOutputStream(f);
				) {
			s.writeObject(d);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	public static void main(String args[]) {
		new SerializaFecha();
	}
}
```


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.