UNIVERSIDAD
POLITECNICA
DE VALENCIA

Manejo de Ficheros
y
Trabajo en RED

Lenin G. Lemus

Despacho: A-6 en la EUI 2ª planta (DISCA)
Horario consultas:
Martes de 13:00 a 14:00
Viernes de 11:30 a 13:30
Teléfono: 96.387.97.02
Extensión UPV: 9702
E-mail: lemus@disca.upv.es

David de Andrés

Despacho: Lab. GSTF 1ª planta (DISCA)
Horario consultas:
Martes de 13:00 a 14:00
Viernes de 11:30 a 13:30
E-mail: ddandres@disca.upv.es

1. Clases Útiles

1.1. Clase StringTokenizer

En muchas ocasiones es necesario dividir una cadena de caracteres en subcadenas, identificando dónde empiezan y acaban las mismas mediante el uso de separadores. Un ejemplo es la división de una frase en las palabras que la constituyen.

Empleando la terminología utilizada en campo de los compiladores, a las partes se les llama ***tokens*** y a los caracteres intermedios ***delimitadores***. La acción de extraer una secuencia de ***tokens*** a partir de la cadena recibe la denominación de ***análisis lexicográfico*** o ***scanning***.

La clase ***StringTokenizer*** simplifica en gran medida este trabajo. En su constructor se le pasan dos ***Strings***, siendo el primero la cadena a analizar. El segundo ***String***, opcional, contiene el conjunto de caracteres que actuarán como delimitadores; en caso de no pasar este segundo parámetro, se consideran como delimitadores los caracteres espacio, tabulador, salto de línea y retomo de carro.

Una vez creado el objeto, se puede llamar al método ***nextToken( )*** que devuelve un ***String*** conteniendo el siguiente ***token*** y lo extrae de la cadena. El método ***hasMoreTokens( )*** devuelve *true* mientras queden ***tokens*** por extraer.

1.2. Clase Date

La clase ***Date*** está concebida para facilitar, en gran medida, el manejo de fechas y horas y cada objeto instanciado a partir de ella contiene información de día, mes, año, día de la semana, horas, minutos y segundos.

Esta clase tiene varios constructores, dependiendo de la cantidad de datos que se pasan para inicializar el objeto:

- ?? Date( ): inicializa el objeto con los datos actuales de fecha y hora, tomándolos del sistema operativo.
- ?? Date(int año, int mes, int dia_mes): inicializa la fecha del objeto a los valores especificados y pone la hora a medianoche: 00h00m00s.
- ?? Date(int año, int mes, int dia_mes, int horas, int minutos): inicializa el objeto con los valores indicados, poniendo a cero los segundos.
- ?? Date(int año, int mes, int dia_mes, int horas, int minutos, int segundos): inicializa el objeto con los valores indicados.
- ?? Date(String FechaHora): analiza el string, que deberá contener la fecha y la hora, por ejemplo "Thu, 12 Jan 1997 23:24:21".

Además de inicializarlos en el constructor, los valores se pueden leer y cambiar en cualquier momento, como es lógico. Los métodos de lectura son: getYear( ), getMonth( ), getDate( ), getDay( ), getHours( ), getMinutes( ) y getSeconds( ). Cada uno de ellos devuelve un int y corresponden al año, al mes, al día del mes, al día de la semana, a la hora, a los minutos y a los segundos, respectivamente.

Los métodos de escritura son setYear( ), setMonth( ), setDate( ), setDay( ), setHours( ), setMinutes( ) y setSeconds( ). Cada uno de ellos tiene un único parámetro entero que contiene el nuevo valor.

Existen tres métodos que permiten comparar dos objetos de tipo Date:

- ?? *boolean* before(Date DeCuando): devuelve *true* si el objeto sobre el que se invoca contiene una fecha y hora anteriores a las del objeto que se le pasa como parámetro.
- ?? *boolean* after(Date DeCuando): devuelve *true* si el objeto sobre el que se invoca contiene una fecha y hora posteriores a las del objeto que se le pasa , como parámetro.
- ?? *boolean* equals(Date OtraFecha): devuelve *true* si los contenidos de ambos objetos coinciden.

Por último, el método toString( ) entrega un string con la fecha en inglés, en mismo formato aceptado por el último de los constructores que antes he enumerado. Para disponer de esta funcionalidad en otros idiomas, habrá que implementar una clase derivada y redefinir este método.

1.3. Manejo de ficheros: java.io

La mayor parte de los dispositivos de entrada/salida se pueden tratar como si fueran simples secuencias de *byte*s, independientemente de si se trata de dispositivos en modo bloque o en modo carácter, e incluso de simulaciones software de un dispositivo que no está físicamente presente. Sobre una de secuencias o ***streams*** se pueden efectuar operaciones de lectura o escritura (según sea de entrada o de salida) y de posicionamiento dentro de la secuencia.

Esta abstracción es la que se ha adoptado en Java como base conceptual de todas las clases relacionadas con la entrada/salida, motivo por el cual lo que vamos a ver en esta sección sigue siendo válido al hablar de comunicaciones en red, que estudiaremos posteriormente. Pese a la potencia del concepto de ***stream***, las peculiaridades de cada dispositivo hacen necesarias clases adicionales para manejarlas, que como es lógico son muy diferentes entre sí.

Clase File

Esta es la clase que sirve para manejar información relativa a las propiedades de los ficheros como tales dejando el contenido para las clases ***stream*** correspondientes. La clase File admite tres posibles constructores:

- ?? File(String path): crea un objeto File en el directorio especificado.
- ?? File(String path, String nombreFichero): crea un objeto File referido al fichero cuyo nombre y ***path*** se especifican.
- ?? File(File objeto, String nombre): crea un objeto de tipo File con el nombre especificado y en el mismo directorio que el otro objeto que se le pasa como parámetro.

La mayoría de los métodos de la clase File sirven para examinar las propiedades de un fichero; veamos algunos de ellos:

- ?? getName( ): devuelve el nombre del fichero en el disco
- ?? getPath( ): devuelve el path completo del fichero, incluido su nombre
- ?? getParent( ): devuelve el path del directorio en el que se encuentra el fichero
- ?? canWrite( ): devuelve *true* si se puede escribir en el fichero
- ?? canRead( ): devuelve *true* si se puede leer del fichero
- ?? exists( ): devuelve *true* si el objeto corresponde a un fichero que existe realmente en el disco
- ?? isDirectory( ): devuelve *true* si el objeto representa un directorio
- ?? isFile( ): devuelve *true* si el objeto representa un fichero normal (no es, por ejemplo, un named pipe)
- ?? lastModified( ): devuelve el momento de la última modificación
- ?? length( ): devuelve la longitud en *byte*s del fichero

Los siguientes métodos permiten cambiar propiedades de un fichero:

- ?? renameTo(File destino): pone al fichero el mismo nombre que el del objeto que se le pasa como parámetro
- ?? delete( ): borra el fichero representado por el objeto, siempre que se trate de un fichero simple y no de un directorio, incluso aunque éste no contenga ficheros

Directorios

Si al crear un objeto de tipo ***File*** pasamos al constructor el ***path*** de un directorio, éste quedará representado por aquél y su método isDirectory( ) devolverá *true*. En este caso, se puede invocar sobre el objeto el método list(), que devuelve un ***array*** de String conteniendo los nombres de todos los ficheros contenidos en el directorio. En caso de tratar de invocar list() sobre un objeto File que no corresponde a un directorio, se producirá una excepción de tipo NullPointerException.

Hay que hacer notar que para los paths se acepta, además de "\\" la convención Unix de separar los nombres con la barra normal (“/”), aunque se trabaje en entorno Windows.

Así, los nombres "\\ficheros\\datos.txt" y "/ficheros/datos.txt hacen referencia al mismo fichero.

Interface FilenameFilter

Cuando se quiere especificar el tipo de ficheros que se desea obtener al llamar al método list(), es necesario indicar unos criterios de aceptación o rechazo, que serán aplicados, uno por uno, a cada uno de los ficheros de un directorio.

Para facilitar este filtrado se ha creado el *int**erface FilenameFilter*** cuyo único método, accept( ), recibe como parámetros un objeto de tipo File, que representa el directorio, y un ***String*** que indica el nombre del fichero particular para el que se debe tomar la decisión:

?? accept(***File*** directorio, ***String*** fichero) devuelve *true* si el fichero es aceptado y *false* si es rechazado.

Este método es llamado para cada fichero y sólo los nombres de los ficheros aceptados serán colocados en el ***array*** entregado por list(). Para conseguir el filtrado, se crea un objeto que implemente el *int**erface***, es decir, que defina el método accept( ), y se pasa como parámetro de list( ):

```
1.  File MiDirectorio = new File(NombreDirectorio);
2.  ……..
3.  class Filtro implements Filename Filter{
4.          public boolean accept(File directorio, String nombre){
5.                  ……..
6.                  //aquí se devuelve true o false, según se acepte o no el nombre fichero
7.          }//fin de Filtro
8.      ……
9.      FilenameFilter MiFiltro = new Filtro();
10.     String nombres() = MiDirectorio.list(MiFiltro);
```

Clase Input*Stream*

Esta es la clase abstracta que define el modo de acceder a todos los ***streams*** de entrada de datos. Contiene los siguientes métodos:

?? read( ): devuelve un entero con el valor del siguiente *byte* de entrada.

?? read(*byte* buffer[]): intenta leer el número de *byte*s dado por la longitud del ***array*** (buffer. length) y devuelve el número de *byte*s leídos realmente.

?? read(*byte* buffer[], int posidon, int número): intenta leer el número de *byte*s especificado y colocarlos a partir de la posición indicada del ***array***; devuelve el número de *byte*s leídos.

?? skip(long numero): se salta el número de *byte*s especificado.

?? available( ): devuelve el número de *byte*s disponibles para lectura en el momento de la llamada.

?? close( ): cierra la entrada de datos; cualquier intento de lectura posterior producirá una excepción.

?? mark(Int hastaCuando): pone una marca en el punto actual para poder volver a ella más adelante. El parámetro indica cuántos *byte*s se pueden leer antes de que se pierda la marca.

?? reset( ): deja la posición de lectura en el último punto especificado con mark( ).

?? markSupported( ): devuelve *true* si la funcionalidad dada por mark( )/reset( ) está disponible para el ***stream*** sobre el que se invoca.

Los errores dentro de estos métodos lanzan excepciones de tipo IOException.

Clase *OutputStream*

Esta clase abstracta determina cómo se va a acceder a los ***streams*** de salida. Sus métodos no tienen valor de retomo y son:

?? write(int b): escribe en el ***stream*** el valor que se le pasa como parámetro, previa conversión a tipo *byte*.

?? write(*byte* buffer[]): escribe por orden los *byte*s del ***array*** en el ***stream***.

?? write(*byte* buffer[], int comienzo, int numero): escribe en el ***stream*** el número de *byte*s especificado a partir de la posición en el ***array*** indicada por comienzo.

?? flush( ): hace que se escriba en la salida toda la información que esté contenida en los buffers internos.

?? close( ): cierra la salida de datos; cualquier intento de escritura posterior producirá una excepción.

Cualquier error en uno de estos métodos dará lugar a una excepción de tipo ***IOException***.

Clases para *streams* en fichero

Las clases ***FileInputStream*** y ***FileOutputStream*** heredan de las clases abstractas ***InputStream*** y ***OutputStream*** , respectivamente, y permiten emplear sus métodos sobre ***streams*** asociados a ficheros.

La clase ***FileInputStream*** admite los siguientes constructores:

?? ***FileInputStream***(String Nombre): crea un ***stream*** a partir del fichero cuyo nombre y ***path*** se le pasan como parámetro (Ej. New ***FileInputStream*** ("c:\\contig.sys")).

?? ***FileInputStream***(File fichero): crea un ***stream*** a partir del fichero asociado al objeto de tipo ***File*** que se le pasa como parámetro. En caso de que el fichero especificado no exista, se produce una excepción del tipo ***FileNotFoundException***.

La clase ***FileOutputStream*** admite las mismas dos variantes de constructor:

?? ***FileQutputStream***(String Nombre): crea un ***stream*** a partir del fichero cuyo nombre y path se le pasan como parámetro.

?? ***FileOutputStream*** (File fichero): crea un ***stream*** a partir del fichero asociado al objeto de tipo File que se le pasa como parámetro.

Estos constructores se ejecutarán correctamente aunque el fichero referenciado aún no exista.

Clases para *streams* en memoria

La clase *Byte****ArraylnputStream*** permite crear ***streams*** cuya fuente de datos en un ***array*** de *byte*s en memoria. Sus constructores son:

?? *Byte****ArrayInputStream***(*byte* fuente[]): crea un ***stream*** a partir del array pasado como parámetro.

?? *Byte****ArraylnputStream***(*byte* fuente[], int posición, int numero): crea un ***stream*** a partir de la parte del ***array*** que comienza en la posición especificada y ocupa número *byte*s.

La clase *Byte****ArrayOutputStream*** es la complementaria de la anterior y permite emplear un ***array*** como destino de los datos de un ***stream***. Los constructores admitidos son:

?? *Byte****ArrayOutputStream*** ( ): crea un ***stream*** con un buffer asociado de 32 *byte*s.

?? *Byte****ArrayOutputStream*** (int tamaño): crea un ***stream***, dando al buffer el tamaño especificado.

El método to*Byte****Array***( ) permite asignar el contenido del ***stream*** a un ***array*** de *byte*s.

1.4. Comunicaciones en red: java.net

Clase lnetAddress

Cualquier máquina conectada a Intemet, lo que se conoce también como ***host***, ha de tener una dirección de 32 bits, denominada ***dirección IP***, que la identifica de forma única, salvo ocasionales conflictos transitorios de direcciones. Por otra parte, dentro de un mismo ***host*** puede haber diferentes procesos comunicándose de manera independiente con otros procesos en otras máquinas, o incluso con diferentes ***threads*** dentro de un mismo proceso en otra máquina. Por ejemplo, es frecuente que un ordenador tenga establecidas simultáneamente varias sesiones FTP con distintos servidores, y a la vez esté recibiendo correo mediante SMTP y leyendo las ***news*** por NNTP. Lo que se hace para poner de acuerdo a las distintas entidades que se van a comunicar y repartir la información entre ellas es asignar a cada servicio un identificador, también de 32 bits, denominado número de puerto (***port number***). De esta forma, una conexión queda completamente especificada por los números de puerto que emplean los interlocutores y las direcciones IP de las máquinas en que residen.

Un proceso o ***thread*** puede ponerse a emitir o recibir por un puerto cualquiera en cualquier momento, siempre que no esté ocupado, pero por convenio se emplean unos números de puerto determinados para algunos servicios: por ejemplo, FTP utiliza normalmente el puerto 25.

Como alternativa a las direcciones IP, existen unos identificadores simbólicos para las máquinas, basados en el servicio de nombres de dominio. De esta forma, podemos emplear nombres como www.upv.es o www.intel.com en vez de las direcciones IP de estos servidores. La clase ***InetAddress*** permite encapsular una dirección IP y su nombre de servidor correspondiente, de un modo que facilita su uso. Esta clase no tiene constructores accesibles para el código de usuario, sino que para instanciarla hay que invocar a unos métodos estáticos especiales, denominados métodos factoría (factory methods). Existen tres de estos métodos:

- getLocalHost( ): devuelve un objeto ***InetAddress*** conteniendo la dirección de la máquina sobre la que se está ejecutando el programa.
- getByName(String nombre): devuelve un objeto ***InetAddress*** con la dirección de la máquina cuyo nombre se le pasa como parámetro. Si no se encuentra ninguna máquina con ese nombre, se produce una excepción de tipo ***UnknownHostException***.
- getAllByName(String nombre): devuelve un ***array*** de objetos ***InetAddress*** conteniendo las direcciones de todas las máquinas que tienen el nombre pasado como parámetro. Si no se encuentra ninguna se produce una excepción de tipo ***UnknownHostException***.

Además de los métodos factoría que sirven para instanciar la clase, existen algunos métodos normales (de instancia):

- getHostName(): devuelve un ***String*** que contiene el nombre de la máquina a la que corresponde la dirección.
- getAddress(): devuelve un ***array*** de 4 *byte*s con la dirección IP.
- toString(): devuelve un ***String*** con el nombre y la dirección IP.

Manejo de datagramas UDP

A diferencia de las redes basadas en circuitos virtuales (redes X.25, por , ejemplo), Internet es una red de datagramas. La información a transmitir se divide en paquetes, denominados datagramas IP (siglas de Intenet Protocol), y cada uno de ellos es tratado de forma independiente por los nodos de la red.

De hecho, es posible y hasta probable que dos datagramas que van de un mismo origen a un mismo destino sigan caminos diferentes en la red. Además, al seguir diferentes caminos puede ocurrir que uno llegue y el otro no, o que lleguen en un orden distinto de aquel en que fueros transmitidos.

Lo normal es que, en vez de emplear los paquetes IP directamente, se emplee una capa de software que proporciona, a partir de los ***datagramas***, un servicio fiable de transferencia continua, lo que se conoce como TCP, siglas de “Transmission Control Protocol”. Para emplear este servicio se suele usar el interfaz de ***Sockets***, que veremos en el siguiente apartado.

Pese a la comodidad que supone el uso de ***Sockets***, a veces es conveniente, por motivos de eficiencia y/o velocidad, prescindir de su uso. Para esto se proporciona el protocolo UDP (User Datagram Protocol) que da la posibilidad de enviar y recibir directamente ***datagramas*** IP.

En Java, el protocolo UDP está, soportado por las clases ***DatagramPacket*** y ***DatagramSocket***. La primera de ellas está destinada a contener la información que va a ir dentro del datagrama IP.

Esta clase tiene dos constructores, según se vaya a usar para la recepción o para el envío:

- ***DatagramPacket***(*byte* buffer[], int longitud): construye un objeto Datagram-Packet para recepción; los datos recibidos se guardarán en el array, con un número máximo de *byte*s dado por el segundo parámetro.
- ***DatagramPacket***(*byte* buffer[], int longitud, InetAddress direccion, int puerto): construye un objeto ***DatagramPacket*** para envío. Además del buffer y la longitud, se proporcionan un objeto ***InetAddress*** correspondiente a la máquina destino y el número de puerto concreto al que se desea enviar el ***datagrama***.

Los métodos utilizables para acceder al estado interno de un objeto DatagramPacket son:

?? getAddress( ): devuelve un objeto de tipo ***InetAddress*** con la dirección de destino.

?? getPort( ): devuelve el número de puerto al que va dirigido el datagrama.

?? getData( ): devuelve un ***array*** de *byte*s con los datos recibidos

?? getLengtb( ): entrega la cantidad de *byte*s válidos contenidos en el ***array*** devuelto por getData().

Para poder enviar o recibir los datos encapsulados en un DatagramPacket hay que emplear un objeto de la clase ***DatagramSocket***. Esta clase tiene también dos constructores, uno para envío y otro para recepción:

?? Datagram***Socket***( ): crea un objeto Datagram***Socket*** para envío.

?? Datagram***Socket***(int puerto): crea un objeto Datagram***Socket*** para recibir por el puerto especificado.

Una vez creado un Datagram***Socket***, para enviar y recibir datos se invocan sus métodos:

?? send(DatagramPacket datagrama): envía un datagrama

?? receive(DatagramPacket datagrama): recibe un datagrama

No hay que confundir los ***Sockets*** para Datagrama con los ***Sockets*** TCP, que veremos a continuación. Siempre que en la literatura técnica se habla de "***Sockets***" a secas, de lo que se habla es de ***Sockets TCP*** y nosotros vamos a seguir también ese convenio.

*Socket*s del lado del cliente

El mecanismo de ***Sockets*** se emplea para establecer una comunicación bidireccional fiable entre dos procesos. Estos procesos pueden residir en la misma máquina o en máquinas diferentes conectadas en red. Un ***Socket*** es un canal de comunicación simétrico, una vez creado, pero si se va a implementar un servicio basado en el modelo cliente-servidor, aparecen requisitos distintos en un extremo de la conexión y en el otro. Mientras que un servidor debe esperar, cuando crea un ***Socket***, a que se conecte al mismo un cliente, éste deberá tratar la no disponibilidad del servidor como un fallo y actuar en consecuencia. Esto hace que en un servidor, además de la clase ***Socket*** que vamos a ver aquí, se emplee la clase ***ServerSocket***, que examinaremos en el siguiente apartado. La creación de un objeto de tipo ***Socket*** también establece la conexión. Se pueden usar los siguientes constructores:

?? ***Socket***(String host, int port): crea un ***Socket*** que enlaza el host local con el host y el puerto especificados. Puede producir las excepciones UnknownHost Exception e IOException.

?? ***Socket***(InetAddress direccion, int port): análogo al anterior, sólo que en este caso el host se especifica pasando un objeto InetAddress. En caso de haber problemas, se producirá una excepción de tipo IOException.

Se pueden conocer en cualquier momento los atributos de un objeto de tipo ***Socket*** mediante los siguientes métodos:

?? geInetAddress0: devuelve un objeto InetAddress con la dirección del host remoto al que está conectado el ***Socket***.

?? getPort( ): devuelve el número de puerto al que está conectado el ***Socket*** en el host remoto.

?? getLocalPort0: devuelve el número del puerto local al que está conectado el ***Socket***.

Una vez creado un ***Socket***, el envío y recepción de datos a través del mismo son exactamente iguales que la escritura y la lectura de los datos de un fichero. Para obtener y cerrar los streams de entrada y salida asociados al ***Socket***, se invocan los siguientes métodos:

?? getInputStream( ): devuelve el objeto de tipo ***InputStream*** asociado con el ***Socket***.

?? get*OutputStream* ( ): devuelve el objeto de tipo ***OutputStream*** asociado con el ***Socket***.

?? close( ): cierra tanto el ***stream*** de entrada como el de salida.

*Socket*s del lado del servidor

Como ya adelantábamos en el apartado anterior, si se desea que un programa actúe como servidor, es necesario algún mecanismo que le permita esperar a recibir conexiones por tiempo indefinido. Esta funcionalidad de espera con bloqueo, junto con las de ***multithreading*** son las que permiten crear servidores que atiendan a varias conexiones simultáneas de un modo muy eficiente.

El ***thread*** principal del servidor puede estar permanentemente en un bucle, dentro del cual se queda bloqueado esperando a que una máquina remota establezca una conexión con él. En el momento en que comienza a existir dicha conexión, el servidor queda desbloqueado y crea un ***thread*** hijo que atiende al ***Socket*** establecido con ese cliente concreto. Una vez hecho esto, el ***thread*** principal vuelve a quedar en espera de la siguiente conexión y el ***thread*** hijo atiende a las peticiones del cliente hasta que se cierra el ***Socket*** que le enlaza con él, momento en el cual se autodestruye. Esta arquitectura permite conseguir de un modo muy sencillo un reparto de carga equitativo entre los distintos clientes que intentan acceder al servicio.

Para implementar esta funcionalidad se usa la clase ***ServerSocket***, que admite los siguientes constructores:

- ***ServerSocket***(int puerto): crea un ***ServerSocket*** sobre el puerto especificado.
- ***ServerSocket***(int puerto, int timeout): crea un ***ServerSocket*** sobre el puerto especificado. En caso de que ese puerto esté ocupado, espera un tiempo de hasta ***timeout*** milisegundos a que quede libre. En caso de no quedar libre en este tiempo, se produce una excepción.

Ambos constructores generan una excepción ***IOException*** en caso de problemas.

Para esperar a que un cliente "llame" y establezca una conexión, se invoca el método ***accept***( ). Este método deja bloqueado el ***thread*** que lo llama hasta que se produce la conexión. Una vez que ésta queda establecida, la llamada a ***accept***( ) devuelve un objeto de tipo ***Socket***, que se puede usar para la comunicación de la forma descrita en el apartado anterior.

Clases URL y URLConnection

Los ***URLs*** ofrecen una sintaxis uniforme para especificar una serie de servicios y los ***hosts*** que van a proporcionar esos servicios.

Los ***Sockets*** proporcionan un procedimiento genérico para establecer y mantener una conexión TCP/IP, pero obligan a implementar los protocolos de más alto nivel como parte del código de aplicación. Para evitar en lo posible esta situación, Java proporciona dos clases URL y URLConnection, que permiten usar estos protocolos de una manera transparente.

La clase URL encapsula un URL y tiene los siguientes constructores:

- URL(String cadena): construye un objeto URL a partir de una cadena que contiene un URL en el mismo formato que se introduciría manualmente en un navegador.
- URL(String Protocolo, String ***host***, int puerto, String fichero): permite acceder a un fichero dándole las partes constituyentes de su URL, e indicando el puerto con el que se quiere establecer la conexión.
- URL(String Protocolo, String ***host***, String fichero): permite acceder a un fichero dándole las partes constituyentes de su URL; se diferencia del anterior en que toma el número de puerto por defecto para el protocolo especificado.

Cualquiera de estos constructores genera una excepción ***MalformedURLException*** en caso de que el URL que se le especifica o alguno de sus constituyentes no sean sintácticamente correctos.

Una vez que disponemos de un objeto URL que hace referencia al recurso “X” al que queremos acceder, podemos acceder a sus propiedades o a su contenido mediante la clase ***URLConnection***. Los objetos de esta clase se obtienen como valor de retorno del método ***openConnection***( ) de un objeto de la clase URL. Por ejemplo, podríamos hacer:

```
URL localizador = new URL("http://www.sun.coml");
URLConnection conexion = localizador openConnectiono;
```

Cuando ya tenemos el objeto de tipo URLConnection, podemos acceder a a las propiedades del recurso referenciado mediante los métodos ***getDate( ), getContentType( ), getExpiration( ), getLastModified( )*** y ***GetContentLength***( ) que entregan, respectivamente, la fecha, el tipo de contenido, la fecha de expiración, la fecha de la última modificación y la longitud del recurso.

El método ***getInputStream( )*** devuelve un objeto de tipo ***InputStream*** que puede emplear para leer el contenido del recurso de forma idéntica a como lee de un fichero o de un ***Socket***.

2. Entrada/Salida en Java

Este capítulo está destinado a presentar las herramientas que ofrece Java para trabajar con datos externos, principalmente provenientes de archivos en disco, otros procesos en ejecución, o (como veremos en detalle en el próximo capítulo) de recursos de red.

Es fundamental que el lector ya se encuentre familiarizado con los aspectos básicos de Java y de la Programación Orientada a Objetos, ya que en este capítulo se hace un uso extensivo de dichos conceptos, aunque no nos detengamos a comentarlos. Deben ser ya conocidos los temas relacionados con la sintaxis básica, la declaración de clases, la creación y manejo de objetos y la gestión de excepciones.

Comenzaremos con la presentación del concepto de flujo ("stream" en inglés), muy común en otros lenguajes de programación, y que sirve para englobar todos los sistemas de entrada/salida, independientemente del origen de los datos.

A continuación seguiremos con aplicaciones básicas de los flujos, como el acceso a archivos.

Por último terminaremos presentando toda una serie de clases derivadas de los flujos básicos, útiles para filtrar los datos, cambiarlos de formato, etc.

Nota: Aunque ciertos conceptos (como la clase java.io.PrintStream) no sean introducidos hasta el final de este capítulo, nos será necesario utilizarlos a lo largo de la explicación cada vez que queramos escribir algo en la pantalla. Supondremos por ahora que cada vez que hagamos una referencia a construcciones del tipo System.out.println("Cadena de texto") o System.err.println("Mensaje de error"), simplemente estamos escribiendo texto en la pantalla, sin preocuparnos del hecho de que en realidad estemos trabajando con objetos de entrada/salida relativamente complejos.

2.1. El concepto de flujo

Podemos imaginar un flujo como un tubo donde podemos leer o escribir *byte*s. No nos importa lo que pueda haber en el otro extremo del tubo: puede ser un teclado, un monitor, un archivo, un proceso, una conexión TCP/IP o un objeto Java.

Todos los flujos que aparecen en Java (englobados generalmente en el paquete java.io) pertenecen a dos clases abstractas comunes:

java.io.InputStream para los flujos de entrada (aquellos de los que podemos leer) y java.io.*OutputStream* para los flujos de salida (aquellos en los que podemos escribir).

Estos flujos, como hemos dicho antes, pueden tener orígenes diversos (un archivo, un ***Socket*** TCP, etc.), pero una vez que tenemos una referencia a ellos, podemos trabajar siempre de la misma forma:

leyendo datos mediante los métodos de la familia read() o escribiendo datos con los métodos write().

2.1.1. Trabajo con flujos

Vamos a ver en detalle las herramientas que nos proporcionan los flujos básicos, junto con algunos ejemplos. Supondremos que los objetos de tipo ***InputStream*** y ***OutputStream*** nos vienen dados, sin preocuparnos de cómo han sido creados.

De hecho, en los ejemplos que aparezcan a continuación nunca concretaremos la forma en que esos objetos son creados, dejándolo indicado mediante puntos suspensivos. Para fijar ideas, podemos imaginar que, cuando tengamos un ***InputStream*** en realidad estamos leyendo de la entrada estándar (el teclado, generalmente), y cuando escribamos en un ***OutputStream*** estamos usando la salida estándar.

Lectura de *byte*s individuales

Mediante código como:

```
1.  InputStream is = ...;
2.  int b = is.read();
```

podemos obtener el siguiente *byte* del InputStream. Es importante darse cuenta de que el *byte* (8 bits) se devuelve como un dato de tipo int (32 bits), con un valor entre 0 y 255. En caso de que se haya alcanzado el final del archivo, read() devuelve un valor de -1.

Lectura de varios *byte*s

Primero creamos una matriz de *byte*s del tamaño adecuado. El tamaño de esta matriz es lo que le indica al método read() cuántos *byte*s debe leer como máximo. Veamos el código para leer 1024 *byte*s de un flujo de entrada:

```
1.  byte[] miArray = new byte[1024];
2.  InputStream is = ...;
3.  int leidos = is.read(miArray);
```

La variable leídos almacena el número de *byte*s que se han leído en realidad. Si se ha alcanzado el fin de archivo, devuelve el valor -1. Es importante ver que no hay ninguna garantía de que vayamos a leer exactamente el número de *byte*s especificado. El número de *byte*s leídos puede ser menor por varias razones: porque estamos leyendo de un archivo que se ha acabado, porque los datos de una conexión de red tardan en llegar o por cualquier otra razón.

De cualquier forma, mientras el método read() devuelva un valor distinto de -1, podemos seguir leyendo mediante sucesivas llamadas a read(). Otra variante de este método es la siguiente:

```
1.  byte[] miArray = new byte[2048];
2.  InputStream is = ...;
3.  int origen = 512;
4.  int longitud = 1024;
5.  int leidos = is.read(miArray, origen, longitud);
```

En este caso especificamos el número de *byte*s a leer mediante la variable longitud, y la posición dentro del array donde se deben almacenar los datos la indica la variable origen. El código anterior lee hasta 1024 *byte*s del flujo de entrada y los coloca en los elementos 512, 513, 514, etc. de la matriz miArray.

¿Cuántos *byte*s hay disponibles?

Los métodos anteriores bloquean la ejecución hasta que existen nuevos datos para ser leídos. Si estos datos vienen, por ejemplo, desde una conexión de red, pueden pasar varios segundos (o más tiempo, dependiendo de la situación) hasta que lleguen nuevos datos. Si estamos interesados en leer desde un *InputStream* pero no queremos arriesgarnos a que nuestro programa se pare durante un tiempo indefinido, podemos usar el método *available*( ), que devuelve el número de *byte*s disponibles que podemos leer sin bloquearnos. Estos *byte*s disponibles serán los que el sistema operativo tenga almacenados en sus *buffers* de entrada.

Si, por ejemplo, tenemos varias conexiones en red abiertas, podemos usar el método *available*( ) para determinar a cuál de ellas debemos dedicar nuestra atención. Por ejemplo, el siguiente código recorre cíclicamente una matriz de referencias ***InputStream*** y lee los datos a medida que le van llegando:

```
1.  InputStream[] is = new InputStream[10];
2.  ...
3.  int n=0;
4.  for(;;){
5.  if (is[n].available()>0){
6.  /* Leemos los datos y los procesamos */
7.  }
8.  n=(++n)%10;
9.  }
```

Cerrar el flujo de entrada

Cuando ya no necesitamos leer datos de un flujo de entrada, debemos liberar los recursos asociados mediante el método close(). Si no cerramos el ***InputStream*** explícitamente, el flujo asociado se cierra cuando se destruye el objeto.

Escritura de *byte*s individuales

La clase ***OutputStream*** dispone de varios métodos write(). Veamos un ejemplo en el que escribimos un solo *byte* en un flujo de salida:

```
1.  OutputStream os = ... ;
2.  int dato = 123;
3.  os.write(dato);
```

Como en el método read() de la clase InputStream, el método write() recibe un *byte* dentro de una variable de tipo int (32 bits).

Escritura de varios *byte*s

El código siguiente apenas merece explicación:

```
1.  byte[] matriz= { 65,66,67,68,69};
2.  OutputStream os = ... ;
3.  os.write(matriz); /* Escribe bytes 65,66,67,68,69 */
4.  os.write(matriz,1,3); /* Escribe bytes 66,67,68 */
```

Cerrar el flujo de salida

Los recursos asignados al ***OutputStream*** se liberan con el método close(). Este método, además, garantiza que los datos que hayamos escrito en el flujo pero que aun no hayan sido enviados (a un archivo, por ejemplo) se manden a su destino. En los casos en que queramos asegurarnos de que los datos han sido enviados, pero no queramos cerrar el flujo, podemos usar el método flush(), que vacía los buffers de salida.

Gestión de excepciones de entrada/salida

Los ejemplos que hemos visto hasta ahora han sido "simplificados" para resaltar los elementos más importantes. En una situación real, es necesario tener algo más de precaución a la hora de trabajar con flujos.

Todos los métodos *read*(), *write*(), *available*(), etc. que hemos visto antes lanzan excepciones del tipo java.io.IOException. Estas excepciones es obligatorio capturarlas, o aparecerán errores en tiempo de compilación.

En general, la partes de nuestros programas que trabajen con flujos deben estar dentro de una cláusula ***try ... catch***. Por ejemplo:

```
1.  try{
2.  InputStream is = ... ;
3.  while(...){
4.  /* Leemos y procesamos los datos */
5.  }
6.  } catch (IOException ioe){
7.  System.err.println("Error al abrir el flujo tal y tal...");
8.  ioe.printStackTrace();
9.  }
```

En programas pequeños en los que no queramos complicarnos con estructuras de este tipo, podemos tomar el camino fácil de mandar la excepción "hacia arriba". Por ejemplo:

```
1.  public static void main(String args[]) throws IOException
2.  {
3.  /* Programa creado por un programador vago */
4.  }
```

Flujos de acceso a archivos

En esta sección vamos a empezar a trabajar con flujos "reales", creados por nosotros mismos. El primer tipo de flujo va a ser *FileInputStream.* Esta clase, heredera directa de *InputStream*, es una clase no abstracta, destinada a la lectura de archivos en disco.

Posteriormente veremos la clase complementaria, *FileOutputStream* .

Utilización de la clase FileInputStream

Para crear un objeto de tipo *FileInputStream*, necesitamos indicarle al constructor de la clase el nombre del archivo que queremos leer.

Esto creará un objeto que, a todos los efectos, se comportará como un objeto de tipo *InputStream*, es decir, podremos considerarlo como un "tubo" del cual extraemos *byte*s de uno en uno. No nos importa el hecho de que en el otro extremo del "tubo" haya un archivo del que vienen los datos. Podremos trabajar con el "tubo" usando las técnicas vistas en el caso de *InputStream.*

La forma más directa de abrir un archivo para leer su contenido es la siguiente. En este ejemplo, abrimos el archivo c:\dir\subdir\archivo y realizamos la suma de sus *byte*s.

```
1.  try
2.  {
3.    FileInputStream fis = new
4.    FileInputStream("c:\\dir\\subdir\\archivo");
5.    int parcial,total=0;
6.    while ((parcial=fis.read())!=-1) total += parcial;
7.  }
8.  catch(FileNotFoundException fnfe)
9.  {
10.   /* Archivo no encontrado */
11. }
12. catch (IOException ioe)
13. {
14.   /* Error al leer */
15. }
```

En este fragmento de código tenemos que comprobar también que no se produzca una excepción porque el archivo no exista. Si no lo comprobáramos, el control pasaría al catch de *IOException*, que es la clase "madre" de *FileNotFoundException*. También hay que resaltar las dobles barras hacia atrás, necesarias por la forma que tiene Java de tratar los códigos de escape en las cadenas de texto.

Es importante ver que, aunque el objeto *fis* sea de tipo *FileInputStream*, podemos trabajar con él como si fuera un objeto de tipo *InputStream*, debido a la herencia. Esto es común a casi todos los flujos.

La clase File

En la sección anterior hemos visto uno de los posibles constructores de la clase *FileInputStream*: el que recibe como parámetro un nombre de archivo para leer. Existe otras dos variantes: una que recibe un objeto de tipo File y otra que recibe un objeto de tipo *FileDescriptor*. Veamos en qué consiste la clase File.

La clase File sirve para encapsular la interacción de nuestros programas con el sistema de archivos. Mediante la clase File no nos limitamos a leer el contenido de un archivo, como ocurría con la clase *FileInputStream*, sino que podemos obtener información adicional, como el tamaño del archivo, su tipo, su fecha de creación, los permisos de acceso que tenemos sobre él, etc.

Además, la clase File es la única forma que tenemos de trabajar con directorios (crearlos, ver bs archivos que contienen, cambiar el nombre o borrar los archivos, etc.)

La forma más sencilla de crear un objeto File es:

File miArchivo = new File("c:\\temp\\archivo.txt");

Es muy importante darse cuenta de la diferencia entre un objeto de tipo File y el archivo o directorio al que se refiere. Por ejemplo, el archivo c:\temp\archivo.txt que aparece en el fragmento de código anterior no tiene por qué existir. Para saber si un objeto File se refiere a un archivo existente podemos usar el método exists():

```
1.  File miArchivo = new File("c:\\temp\\archivo.txt");
2.  if (miArchivo.exists())
3.  {
4.    /*    El archivo existe      */
5.  }
6.  else
7.  {
8.    /*    El archivo no existe   */
```

```
9.      }
```

Podemos obtener más información de un objeto File:

```
1.      File f = new File("c:\\temp\\archivo.txt");
2.      long = f.length();                  /* Tamaño del archivo */
3.      boolean lectura = f.canRead()   /* ¿Podemos leer el archivo?
4.      */
5.      boolean escritura = f.canWrite() /* ¿Podemos escribir el
6.      archivo? */
7.      if (f.deleted( ))
8.        {     /* Archivo borrado con éxito          */     }
9.      else
10.       {     /* El archivo no se ha podido borrar    */     }
11.     String nombre = f.getName()  /*  Nombre (sin directorio) del
12.     archivo */
13.     String dir = f.getParent()       /* Directorio del archivo */
14.     if (f.isDirectory())
15.       {     /*     Es un directorio              */     }
16.     else
17.       {     /*     No es un directorio           */     }
18.
19.     if (f.isFile())
20.       {     /*     Es un archivo normal          */     }
21.     else
22.       {     /*     No es un archivo normal       */     }
23.
24.     long modificado = f.lastModified(); /* Fecha de última
25.     modificación */
26.
27.     if (f.renameTo(new File("c:\\temp\\otroArchivo.txt")))
28.       {     /*     Nombre de archivo modificado          */     }
29.     else
30.       {     /*     Nombre de archivo no modificado       */     }
```

Las siguientes líneas de código muestran algunos ejemplo sobre como trabajar con directorios mediante la clase File:

```
1.      File f = new File("c:\\temp");
2.      String[] lista = f.list();  /* Obtenemos la lista de archivos */
3.      for(int n=0;n<lista.length;n++) /* Los imprimimos */
4.        System.out.println("Archivo nº " + n + " : " + lista[n]);
5.
6.      File g = new File("c:\\temp\\dir1");
7.      if (g.mkdir())
8.        {     /* Hemos creado el directorio
9.                dir1 en el directorio c:\temp,
10.               previamente existente. */
11.     }
12.     else
13.       {     /*     No lo hemos podido crear       */     }
14.
15.     File h = new File("c:\\temp\\dirA\\dirB\\dirC\\dirD");
16.     if (h.mkdirs())
17.       {     /* Hemos creado el directorio dirD,
18.               junto con los directorio intermedios
19.               necesarios. */
20.     }
21.     else
22.       {     /*     No lo hemos podido crear      */     }
```

Utilización de la clase File*OutputStream*

El próximo paso es el esperado: abrir un archivo y escribir en él. Hay varias formas de realizar esto, todas prácticamente simétricas a las utilizadas para leer archivos. Por ejemplo:

```
1.      /* Creamos un stream de salida sobre el archivo */
2.      File OutputStream  fos = new
3.      File OutputStream ("c:\\temp\\archivo.dat");
4.      /* Vamos a escribir los números del 1 al 10 en el archivo  */
```

```
5.      for(int n=0;n<10;n++)
6.      {
7.        fos.write(n);
8.      }
9.      fos.close();
```

Otro ejemplo podría ser el siguiente programa, que pretende ser una versión simplificada del comando *copy* de MS-DOS:

```
1.      import java.io.*;
2.      class copy
3.      {
4.        public static void main(String args[])
5.        {
6.          String origen = args[0];
7.          String destino = args[1];
8.          byte buffer[]= new byte[1024];
9.          int cuenta;
10.         try
11.         {
12.           FileInputStream fis = new FileInputStream(origen);
13.           File OutputStream  fos = new File OutputStream (destino);
14.           while((cuenta=fis.read(buffer))>0)
15.             fos.write(buffer,cuenta);
16.         }
17.         catch(IOException ioe)
18.         {
19.           System.err.println("Se ha producido un error");
20.           ioe.printStackTrace();
21.         }
22.       }
23.     }
```

Si ejecutamos este programa mediante

```
java copy c:\temp\archivo1.txt "c:\mis documentos\archivo2.txt"
```

el programa copiará el contenido de un archivo en el otro. Si el archivo destino ya existe, será borrado. Si el archivo origen no existe, el *catch* del programa nos avisará y volcará en pantalla el contenido de la pila de llamadas (aunque en este programa tan simple no sea de ninguna utilidad).

Consultando la documentación del JDK podemos ver que también podemos crear un objeto *FileOutputStrem* mediante un objeto File. Su utilización sería similar a la que hemos visto antes para *FileInputStream*.

Entrada/Salida formateada

Hasta ahora nos hemos dedicado simplemente a escribir *byte*s en un flujo de datos. Aunque con únicamente estas técnicas podríamos conseguir leer y escribir cualquier cosa en un archivo, esta forma de trabajar es relativamente pesada, ya que cada vez que quisiéramos escribir, por ejemplo, un entero de 32 bits en un archivo, tendríamos que dividir los 32 bits en cuatro paquetes de 8 bits cada uno, e ir transmitiéndolos a través de flujo de datos. Para leer ese dato, el proceso sería el inverso: leer cuatro *byte*s del flujo y combinarlos para obtener el número de 32 bits.

Como veremos, el paquete java.io nos proporciona varias herramientas para facilitarnos este trabajo.

La clase Data*OutputStream*

La clase *DataOutputStream*, heredera indirecta de *OutputStream*, añade a ésta última la posibilidad de escribir datos "complejos" en un flujo de salida. Cuando hablamos de datos "complejos", en realidad nos referimos a tipo de datos primitivos, pero no restringidos únicamente a *byte*s y a matrices de *byte*s, como en el caso de *OutputStream.*

Mediante la clase *DataOutputStream* podemos escribir datos de tipo *int*, *float*, *double*, *char*, etc. Incluso podemos escribir algunos objetos, como datos de tipo String, en una gran cantidad de formatos.

La forma general de trabajar con objetos de tipo *DataOutputStream* será la siguiente: obtenemos un objeto *OutputStream* (cuyo origen puede ser cualquiera: un archivo, un *Socket*, una matriz en memoria, la salida standard, etc) y lo "envolvemos" en un objeto *DataOutputStream* , de forma que podamos usar la interfaz

que nos proporciona este último. Para crear este *DataOutputStream*, le pasamos como parámetro el *OutputStream* a su constructor.

Cada vez que escribamos un datos "complejo" en un objeto *DataOutputStream*, éste lo traducirá a *byte*s individuales, y los escribirá en el *OutputStream* subyacente, sin que nosotros tengamos que preocuparnos de la forma en que lo hace.

Veamos algunos ejemplos:

```
1.  OutputStream os = ...;
2.  DataOutputStream dos = new DataOutputStream (os);
3.  int a = 3;
4.  float b = 3.56754;
5.  double c = -456.876345;
6.  dos.writeInt(a);       // Escribimos un entero en el stream (4 bytes)
7.  dos.writeFloat(b);     // Escribimos un nº de precisión simple (4 bytes)
8.  dos.writeDouble(c); // Escribimos un nº de precisión doble (8 bytes)
9.  dos.close();
10. os.close();
```

Los nombres de los métodos usados son bastante descriptivos por sí mismos. En la documentación del JDK podemos ver que existen otros métodos, uno para cada tipo de dato primitivo: writeBoolean(), write*Byte*(), writeChar(), writeLong() y writeShort().

Si seguimos investigando en el JDK, descubrimos algunos métodos de los que aún no hemos hablado: *writeBytes( ), writeChars( ) y writeUTF( ).* Estos tres métodos reciben como parámetros un objeto de tipo String, y lo escriben en el *OutputStream* subyacente usando diferentes formatos.

writeBytes( ) descompone la cadena de texto en *byte*s individuales (obtiene su código ASCII) y los escribe en el flujo de salida. Si la cadena consta de n letras, escribe n *byte*s, sin añadir ningún delimitador ni de principio ni de fin de cadena.

writeChars() descompone la cadena de texto en chars individuales (obtiene su código Unicode) y los escribe en el flujo de salida. Si la cadena consta de n letras, escribe *n chars*, sin añadir ningún delimitador ni de principio ni de fin de cadena.

writeUTF() escribe la cadena en un formato conocido como UTF-8. Este formato incluye información sobre la longitud exacta de la cadena. La conclusión importante que debemos extraer de estos tres últimos métodos es que solo el último nos permite recuperar la cadena con facilidad. Los otros dos métodos no incluyen información sobre la longitud de la cadena, por lo que si otro programa necesita leer los datos que nosotros hemos escrito, es necesario conocer "a priori" la longitud de la cadena. Si no, es imposible saber el número de *byte*s o de *chars* que debemos leer.

Un ejemplo de utilización:

```
1.  ...
2.  String cad1 = "Me voy a convertir en bytes";
3.  String cad2 = "Me voy a convertir en chars";
4.  String cad3 = "Me voy a convertir en formato UTF";
5.
6.  dos.writeBytes(cad1);
7.  dos.writeChars(cad2);
8.  dos.writeUTF(cad3);
```

La clase DataInputStream

Escribir datos formateados no vale de nada si luego no podemos leerlos cómodamente. Para esta función disponemos de la clase *DataInputStream*.

La clase *DataInputStream* está preparada para leer datos generados por un objeto *DataOutputStream* . La especificación garantiza que cualquier archivo escrito por un *DataOutputStream* , sobre cualquier plataforma y sistema operativo, será legible correctamente por un *DataInputStream*, sin que nos tengamos que preocupar de si las máquinas son "*little-endian*" o "*big-endian*".

Supongamos que estamos intentando leer los datos escritos por el programa de ejemplo anterior (donde escribíamos un *int*, *un float* y *un double* en un flujo de salida):

```
1.  InputStream is = ...;
2.  DataInputStream dis = new DataInputStream(is);
3.   int x;
4.  float y;
5.  double z;
6.  x = dis.readInt();
7.  y = dis.readFloat();
8.  z = dis.readDouble();
9.   dis.close();
10. is.close();
```

¿Qué pasa si queremos leer las cadenas de texto que hemos escrito? Es inmediato leer la cadena escrita en formato *UTF*. En cambio, leer las otras dos cadenas nos va a costar más trabajo, ya que debemos leer los *byte*s o los *chars* individuales, juntarlos de forma adecuada y construir la cadena resultante.

En este caso podemos hacer trampa, ya que sabemos que las cadenas cad1 y cad2 tienen una longitud de 27 letras exactamente. En una situación normal, puede que no tengamos esta información.

```
1.  int tam = 27;          // Hacemos trampas...
2.  InputStream is = ...;
3.  DataInputStream dis = new DataInputStream(is);
4.   byte miNuevoArray[] = new byte[tam];
5.  dis.readFully(miNuevoArray); /* Este método es nuevo */
6.  String cadenaConvertida = new String(miNuevoArray,0);
7.   // Ahora tenemos que leer un montón de chars (16 bits)
8.  // que forman el siguiente String escrito.
9.  // Hay que hacer un bucle para ir leyendo los chars uno a uno
10. char otroArrayMas[] = new char[tam];
11. for(int n=0;n<tam;n++)
12.       otroArrayMas[n]=dis.readChar();
13.  String otraCadenaConvertida = new String(otroArrayMas);
14. // Queda leer el String en formato UTF-8
15. // Basta con llamar a readUTF(), ya que la longitud
16. // del String está indicada en el propio archivo.
17.  String ultimaCadena = dis.readUTF();
```

En el ejemplo anterior hemos usado un método nuevo, DataInputStream.readFully(*byte*[]), que es básicamente equivalente a InputStream.read(*byte*[]), con la única diferencia de que no retorna hasta que hayan sido leídos exactamente el número de *byte*s que caben en la matriz.

Es fácil apreciar las ventajas de usar los métodos writeUTF() y readUTF().

Entrada/Salida en memoria

Existen ocasiones en que nos puede interesar acceder a cierta información en memoria como si estuviéramos leyendo desde una flujo de datos. Por ejemplo, podemos tener un objeto capaz de leer un archivo MPEG (vídeo comprimido) y mostrarlo posteriormente en pantalla. Supongamos que ese objeto está diseñado para leer los datos desde un InputStream, pero nosotros queremos que lea los datos desde una array de *byte*s que tenemos en memoria, cuyo contenido hemos generado nosotros de alguna forma.

Una forma de resolver el problema sería escribir el array de *byte*s en un archivo, y hacer que nuestro objeto reproductor de MPEG leyera el contenido.

Otra forma, más elegante y más eficiente que la anterior, sería crear un flujo que extrajera sus datos directamente desde nuestro array en memoria. Cualquier objeto que leyera datos de este flujo, en realidad estaría sacando los datos secuencialmente de nuestro array.

La clase que nos permite hacer esto es DataArrayInputStream. Este clase tiene una constructor al que se le pasa como parámetro la matriz de *byte*s de la que debe leer:

```
1.  int tam = 20;
2.  byte[] buffer = new byte[tam];
3.   for(int n=0;n<tam;n++)
4.       buffer[n] = n;
5.  ByteArrayInputStream bais = new ByteArrayInputStream(buffer);
6.  ...
7.   int c;
8.  while((c=bais.read())!=-1)     /* Mientras leamos algo */
```

```
9.                  System.out.println("Hemos leído el valor " + c);
10.             ...
```

Existe otra clase de este tipo muy útil: *ByteArrayOutputStream* . Mediante esta clase podemos escribir datos en un flujo, sabiendo que estos datos se almacenan internamente en una matriz de *byte*s.

Esta matriz crece dinámicamente a medida que escribimos datos en ella.

Una vez escritos los datos, podemos acceder a la matriz de *byte*s mediante el método *toByteArray*(), que nos devuelve una copia de la matriz original.

Veamos un ejemplo:

```
1.          ByteArrayOutputStream  baos = new ByteArrayOutpuStream();
2.          boolean condicion = false;
3.           while(!condicion){
4.                 int dato = ...;
5.                baos.write(dato);
6.                condicion = ...;
7.           }
8.           byte[] bufferSalida = baos.toByteArray();
```

Una vez ejecutado el código anterior obtenemos una matriz de *byte*s con todos los datos que hemos ido introduciendo a lo largo del bucle.

2.1.2. Trabajar con la red

En esta sección vamos a explorar una de las características más interesantes de Java: su capacidad para integrarse en redes TCP/IP. Esto permite usar este lenguaje para construir aplicaciones distribuidas en muy poco tiempo.

A lo largo de esta sección asumiremos que el lector ya conoce suficientemente los aspectos de Java relacionados con Entrada/Salida, trabajo con flujos de datos, manejo de excepciones, etc.

Los pasos que seguiremos serán los siguientes: primero presentaremos los conceptos básicos de los protocolos de Internet, necesarios para la discusión posterior; a continuación veremos la forma que tiene Java de implementar dichos conceptos; finalmente analizaremos pequeños programas que nos muestren la mejor forma de utilizar las herramientas que Java pone a nuestra disposición.

Advertencia:

Este capítulo no pretende ser una introducción genérica al funcionamiento de la pila de protocolos TCP/IP. Por ello recomendamos la lectura de obras de referencia que cumplen este objetivo. Para el lector interesado exclusivamente en el funcionamiento interno de TCP/IP, sugerimos "Internetworking with TCP/IP", de Douglas E. Comer. Para el que busque un texto más didáctico y que cubra el tema desde un punto de vista más generalista, es interesante "Computer Networks", de Andrew S. Tanenbaum.

Toda la documentación original sobre los protocolos utilizados se encuentran disponibles como documentos RFC y STD en Internet.

Uno de los "*mirrors*" españoles se encuentra en RedIris.

Conceptos básicos

Direcciones IP

Como el lector sabrá, todas las máquinas conectadas a una red IP (Internet Protocol), bien sea la red pública Internet o una red privada, se distinguen por su dirección IP. Esta dirección IP es un número de 32 bits, que por comodidad suele expresarse en forma de 4 números decimales separados por puntos. Cada uno de estos números se corresponde con 8 bits de la dirección IP. Por ejemplo: 209.41.57.70 es una dirección IP.

Esta dirección IP puede ser fija o puede ser distinta cada vez que la máquina se conecta a la red. Esto es lo que ocurre a casi todos los usuarios que se conectan a Internet a través de la línea telefónica.

Nombres de dominio

Para facilitar todavía más el trabajo con direcciones IP, existen los nombres de dominio. Estos nombres de dominio son cadenas alfanuméricas, más fáciles de recordar, y que suelen tener una única dirección IP asociada. Siguiendo con el ejemplo anterior, akal.com es el nombre de dominio asociado con la dirección IP 209.41.57.70.

Los encargados de traducir los nombres de dominios en dirección IP son los servidores DNS (Domain Name Server). Estos servidores DNS mantienen unas tablas de correspondencias entre direcciones y dominios. Estas tablas se actualizan periódicamente a medida que los distintos servidores DNS intercambian sus datos entre sí.

Puertos

La forma general de establecer una comunicación a través de Internet es: 1) Indicar la dirección IP de la máquina con la que queremos conectar y 2) especificar el número de puerto dentro de esa máquina a través del cual queremos establecer la comunicación.

Para que la comunicación se pueda establecer debe haber un proceso en esa máquina "escuchando" en el puerto especificado.

Generalmente, cada máquina tiene una serie de servicios escuchando en ciertos puertos standard: el servidor HTTP (servidor Web) en el puerto 80, el servidor FTP en el puerto 21, el puerto 25 para SMTP (correo electrónico), etc. Estos puertos están asignados en el standard RFC 1700.

Circuitos y paquetes

La forma más común de transmitir información a través de Internet es mediante los protocolos de transporte *TCP* (Transport Control Protocol) y *UDP* (User Datagram Protocol). Estos protocolos se diferencian principalmente en que *TCP* (definido en RFC 793) está orientado a la conexión (es decir, necesita el establecimiento de una conexión entre ambos extremos y realiza un complejo control de errores), mientras que *UDP* (definido en RFC 768) se basa en el envío de paquetes individuales, sin establecimiento previo de una conexión y sin control de errores.

Obviamente, cada tipo de comunicación tiene sus ventajas e inconvenientes, y será necesario decidirse por un protocolo u otro en función de las necesidades de cada aplicación.

Comunicación mediante el protocolo UDP

La clase DatagramSocket

Un objeto java.net.Datagram***Socket*** es un "conector" a través del cual enviamos y recibimos paquetes UDP. En la literatura técnica estos paquetes se denominan *Datagramas.*

La forma usual de crear un *Datagram**Socket*** para recibir paquetes es especificando un número de puerto en el constructor. De esta forma, este *Datagram**Socket*** estará "escuchando" en el puerto especificado, preparado para recibir cualquier paquete entrante.

Si queremos construir un *Datagram**Socket*** para enviar paquetes, no es necesario especificar el número de puerto, ya que nos es indiferente. No ocurre lo mismo en el caso anterior, ya que siempre querremos usar un puerto fijo conocido por el resto de aplicaciones.

```
1.  DatagramSocket ds1 = new DatagramSocket(123);
2.  /* Aquí usamos este DatagramSocket para recibir datos... */
3.  /* ...                                                   */
4.  /* Hemos terminado, cerramos el Socket                   */
5.  ds1.close();
6.  DatagramSocket ds2 = new DatagramSocket();
7.  /* Aquí lo usamos para transmitir datos...               */
8.  /* ...                                                   */
9.  /* Hemos terminado, cerramos el Socket                   */
10. ds2.close();
```

La clase DatagramPacket

Esta clase representa a los paquetes de datos que vamos a recibir o transmitir a través de los objetos Datagram***Socket***. Estos paquetes constan de una cabecera (que incluye la dirección de origen y destino del paquete, el puerto, la longitud del paquete, un checksum, etc.) y un cuerpo (donde se encuentra el contenido real del paquete).

En Java accedemos a las distintas partes de un *datagrama* mediante los métodos de la clase java.net.DatagramPacket.

La forma de construir *datagramas* es distinta dependiendo de si queremos enviar o recibir datos. En caso de que únicamente queramos recibir, debemos especificar un array de *byte*s donde almacenar los datos y un número entero con la longitud máxima que queremos recibir.

Si queremos transmitir, debemos especificar el buffer de datos que queremos enviar, la longitud máxima de datos, la dirección y el puerto de destino del *datagrama*. La dirección de destino se especifica mediante un objeto de tipo *InetAddress*, mientras que el puerto se indica mediante un número entero.

Veamos un ejemplo, donde enviamos un datagrama a una determinada dirección, suponiendo que tenemos un objeto InetAddress correctamente creado. Nótese el uso del método send() de la clase

Datagram***Socket*** para enviar el datagrama:

```
1.  int tam = 1024;
2.  InetAddress direcc = ...;
3.  byte[] datos = new byte[tam];
4.  int puerto = 543;
5.  for (int n=0;n<tam;n++){
6.      /* Generamos los datos que vamos a enviar */
7.      datos[n] = ...;
8.      }
9.  DatagramSocket ds = new DatagramSocket();
```

```
10.     DatagramPacket dp = new DatagramPacket(datos, tam, direcc, puerto);
11.     ds.send(dp);        /* Aquí enviamos el paquete    */
```

Como se ve, la clase *DatagramPacket* solo nos da herramientas para enviar matrices de *byte*s a través de UDP. Si queremos transmitir datos más complejos (cadenas de texto, enteros largos, números en coma flotante, objetos Java, etc.) debemos ser nosotros los encargados de codificar esa información dentro de un array de *byte*s.

Si lo que queremos es recibir datos, solo necesitamos reservar espacio para la información entrante (un array de *byte*s), poner un objeto Datagram***Socket*** escuchando en un puerto y esperar a recibir un paquete mediante el método receive().

```
1.      int tam = 1024;
2.      byte[] buffer = new byte[tam];
3.      int puerto = 987;
4.      DatagramSocket ds = new DatagramSocket(puerto);
5.      DatagramPacket dp = new DatagramPacket(buffer,tam);
6.              ds.receive(dp);
7.      // Ahora tenemos en buffer la información que nos interesa
```

La clase InetAddress

¿Cómo se usa la clase java.net.InetAddress que aparecía en uno de los ejemplos anteriores?.

La forma de crear un objeto InetAddress es mediante el método estático InetAddress.getByName(String), que recibe un nombre de host en notación alfanumérica (por ejemplo "www.etsit.upv.es" o "209.41.57.70" y devuelve un objeto InetAddress con esa dirección.

Si la dirección no existe o no puede ser encontrada, este método lanza una UnknownHostException.

Por ejemplo, si queremos mandar una matriz de *byte*s al puerto 90 de la dirección "www.upv.es", tenemos que escribir:

```
1.      int tam = ...;
2.      int puerto = 90;
3.      String maquina = "www.upv.es";
4.      byte[] buffer = new byte[tam];
5.      // ...
6.      // Generamos el contenido del buffer
7.      // ...
8.      InetAddress direcc = InetAddress.getByName(maquina);
9.      DatagramSocket ds = new DatagramSocket();
10.     DatagramPacket dp = new DatagramPacket(buffer, tam, direcc, puerto);
11.     ds.send(dp);        /* Aquí enviamos el paquete    */
```

Si queremos enviar paquetes a nuestra propia máquina hay que usar como nombre de host la dirección "*localhost*" ó "127.0.0.1". También podemos usar el método InetAddress.getLocalHost(), que devuelve un objeto *InetAddress* que "direcciona" a la máquina local.

Un ejemplo cliente/servidor completo usando UDP

A continuación se incluye una aplicación completa, formada por un cliente y un servidor que se comunican mediante UDP. El cliente envía números enteros (32 bits) al servidor y este se los devuelve después de procesarlos (simplemente los eleva al cuadrado), en forma de enteros largos (64 bits).

En ambos programas (cliente y servidor) se utilizan varias clases explicadas con anterioridad (es fundamental haber leído previamente el capítulo de Entrada/Salida) y se realiza un extensivo control de excepciones.

Los datos que el cliente envía al servidor los obtiene de la línea de comandos.

Veamos un ejemplo en el que el cliente manda los números 43, 56 y 2 al servidor. Si ejecutamos el servidor en una máquina con dirección IP 194.140.47.1 y ejecutamos el cliente en otra máquina con dirección IP distinta, tendríamos que teclear la siguiente en la línea de comandos:

Servidor:

```
java servidor
```

Cliente:

java cliente 194.140.47.1 43 56 2

Como cada paquete enviado es independiente de los demás, podríamos tener varios clientes comunicándose con el servidor al mismo tiempo, con lo que los distintos paquetes se intercalarían unos con otros.

Si ejecutamos ambos programas en la misma máquina tendríamos que abrir dos sesiones distintas y teclear las instrucciones anteriores en cada sesión. En el caso del cliente tendríamos que poner:

java cliente localhost 43 56 2

Veamos el código del programa:

```
1.  //código del servidor
2.  import java.net.*;
3.      import java.io.*;
4.      class servidor{
5.        public static void main(String args[]){
6.
7.          // Primero indicamos la dirección IP local
8.          try{
9.            System.out.println("LocalHost = " + InetAddress.getLocalHost().toString());
10.           } catch (UnknownHostException uhe){
11.             System.err.println("No puedo saber la dirección IP local : " + uhe);
12.             }
13.         // Abrimos un Socket UDP en el puerto 1234.
14.         // A través de este Socket enviaremos datagramas del tipo DatagramPacket
15.         DatagramSocket ds = null;
16.         try{
17.           ds = new DatagramSocket(1234);
18.           } catch(SocketException se){
19.             System.err.println("Se ha producido un error al abrir el Socket : " + se);
20.             System.exit(-1);
21.             }
22.
23.         // Bucle infinito
24.         while(true){
25.           try{
26.           // Nos preparamos a recibir un número entero (32 bits = 4 bytes)
27.           byte bufferEntrada[] = new byte[4];
28.            // Creamos un "contenedor" de datagrama, cuyo buffer
29.           // será el array creado antes
30.           DatagramPacket dp = new DatagramPacket(bufferEntrada,4);
31.            // Esperamos a recibir un paquete
32.           ds.receive(dp);
33.           // Podemos extraer información del paquete
34.           // Nº de puerto desde donde se envió
35.           int puerto = dp.getPort();
36.          // Dirección de Internet desde donde se envió
37.           InetAddress direcc = dp.getAddress();
38.           // "Envolvemos" el buffer con un ByteArrayInputStream...
39.           ByteArrayInputStream bais = new ByteArrayInputStream(bufferEntrada);
40.           // ... que volvemos a "envolver" con un DataInputStream
41.           DataInputStream dis = new DataInputStream(bais);
42.           // Y leemos un número entero a partir del array de bytes
43.           int entrada = dis.readInt();
44.           long salida = (long)entrada*(long)entrada;
45.          // Creamos un ByteArrayOutputStream sobre el que podamos escribir
46.           ByteArrayOutputStream baos = new ByteArrayOutputStream ();
47.           // Lo envolvemos con un DataOutputStream
48.           DataOutputStream dos = new DataOutputStream (baos);
49.           // Escribimos el resultado, que debe ocupar 8 bytes
50.           dos.writeLong(salida);
51.           // Cerramos el buffer de escritura
52.           dos.close();
53.
54.           // Generamos el paquete de vuelta, usando los datos
55.           // del remitente del paquete original
56.           dp = new DatagramPacket(baos.toByteArray(),8,direcc,puerto);
57.           // Enviamos
58.           ds.send(dp);
59.           // Registramos en salida estándar
60.            System.out.println(  "Cliente = " + direcc + ":" + puerto +     "\tEntrada = " + entrada +                    "\tSalida = " +
    salida );
61.           } catch(Exception e){
62.             System.err.println("Se ha producido el error " + e);
63.             }
```

```
64.                 }
65.               }
66.           }
```

```
1.    //Código del cliente
2.    class cliente {
3.          public static void main(String args[]){
4.            // Leemos el primer parámetro, donde debe ir la dirección
5.            // IP del servidor
6.            InetAddress direcc = null;
7.            try{
8.              direcc = InetAddress.getByName(args[0]);
9.              } catch(UnknownHostException uhe){
10.               System.err.println("Host no encontrado : " + uhe);
11.               System.exit(-1);
12.               }
13.           // Puerto que hemos usado para el servidor
14.           int puerto = 1234;
15.           // Creamos el Socket
16.           DatagramSocket ds = null;
17.           try{
18.             ds = new DatagramSocket();
19.             } catch(SocketException se){
20.               System.err.println("Error al abrir el Socket : " + se);
21.               System.exit(-1);
22.               }
23.           // Para cada uno de los argumentos...
24.           for (int n=1;n<args.length;n++){
25.             try{
26.             // Creamos un buffer para escribir
27.             ByteArrayOutputStream  baos = new ByteArrayOutputStream ();
28.             DataOutputStream  dos = new DataOutputStream (baos);
29.             // Convertimos el texto en número
30.             int numero = Integer.parseInt(args[n]);
31.             // Lo escribimos
32.             dos.writeInt(numero);
33.             // y cerramos el buffer
34.             dos.close();
35.             // Creamos paquete
36.             DatagramPacket dp = new DatagramPacket(baos.toByteArray(),4,direcc,puerto);
37.             // y lo mandamos
38.             ds.send(dp);
39.             // Preparamos buffer para recibir número de 8 bytes
40.             byte bufferEntrada[] = new byte[8];
41.             // Creamos el contenedor del paquete
42.             dp = new DatagramPacket(bufferEntrada,8);
43.             // y lo recibimos
44.             ds.receive(dp);
45.             // Creamos un stream de lectura a partir del buffer
46.             ByteArrayInputStream bais = new ByteArrayInputStream(bufferEntrada);
47.             DataInputStream dis = new DataInputStream(bais);
48.             // Leemos el resultado final
49.             long resultado = dis.readLong();
50.             // Indicamos en pantalla
51.             System.out.println(  "Solicitud = " + numero +
52.                                  "\tResultado = " +resultado );
53.             } catch (Exception e){
54.               System.err.println("Se ha producido un error : " + e);
55.               }
56.             }
57.           }
58.       }
```

En el código anterior se usan métodos no vistos hasta ahora, como:

- ?? DatagramPacket.getAddress( ), que devuelve la dirección IP del remitente del mensaje.
- ?? DatagramPacket.getPort( ), que devuelve el puerto.

Existen otros métodos importantes, como:

- ?? DatagramPacket.getLength( ), que indican la longitud de datos recibidos.

Consideraciones sobre UDP

UDP es un protocolo no orientado a la conexión. Esto significa que la comunicación es más rápida (porque no hay que establecer conexiones ni circuitos virtuales entre máquinas) pero también menos segura.

Nadie nos garantiza que un paquete vaya a llegar a su destino.

Tampoco está garantizado que dos paquetes lleguen en el mismo orden en que se enviaron. Si nuestros programas usan una red local privada para comunicarse, es muy improbable que estos problemas aparezcan. Pero si deben usar una red pública como Internet, con múltiples pasarelas y enlaces distintos entre los punto de origen y destino, cualquiera de estos errores puede ocurrir (y ocurrirá, seguro).

En estos casos es responsabilidad del programador el comprobar que los paquetes llegan a sus destino y que la información se transmite en orden. Existen multitud de mecanismos para conseguir esto, pero la complejidad de su implementación hace más recomendable la utilización de un protocolo orientado a la conexión como TCP.

Comunicación mediante el protocolo TCP

La clase Socket

Un objeto java.net.***Socket*** es un "conector" a través del cual enviamos y recibimos datos mediante el protocolo TCP. A diferencia de los "conectores" java.net.Datagram***Socket***, que eran usados para enviar paquetes sueltos, estos "conectores" TCP sirven para enviar o recibir datos de forma continua, como si trabajáramos con un flujo *InputStream* o *OutputStream* .

El hecho de que el protocolo subyacente sea TCP nos permite olvidarnos de detalles relacionados con la pérdida de datos, ya que es el propio protocolo el encargado de hacerlo por nosotros. A todos los efectos, podemos tratar un ***Socket*** TCP como un canal carente de errores.

¿Cómo se usa un objeto ***Socket***? La inicialización de estos objetos es más compleja que en el caso de Datagram***Socket***, ya que es necesario que previamente haya alguien "escuchando" en el extremo receptor.

Suponiendo que, de alguna forma, hay un programa "escuchando" en el puerto 1234 de la máquina con dirección IP 209.41.57.70, la inicialización de nuestro ***Socket*** sería:

```
1.  InetAddress d = InetAddress.getByName("209.41.57.70");
2.  Socket s = new Socket(d,1234);
3.  /* Utilizacion del Socket */
4.  ...
5.  /* Cerramos el Socket */
6.  s.close();
```

Una vez tenemos un ***Socket*** abierto con otra máquina, podemos obtener un flujo de entrada o de salida para poder recibir o transmitir datos. Esto se hace con los métodos ***Socket***.getInputStream() y ***Socket***.get*OutputStream* ():

Veamos un ejemplo donde abrimos un ***Socket***, leemos los *byte*s que nos transmitan desde el otro extremo y los imprimimos en pantalla:

```
1.  InetAddress d = InetAddress.getByName("209.41.57.70");
2.  Socket s = new Socket(d, 1234);
3.  InputStream is = s.getInputStream();
4.  while((int dato=is.read())!=-1){
5.        System.out.println("Recibido " + dato);
6.        }
7.  is.close();
8.  s.close();
```

La clase ServerSocket

La clase java.net.***ServerSocket*** es el mecanismo mediante el cual nuestros programas pueden quedarse "escuchando" en un puerto, esperando conexiones entrantes. La forma general de trabajar con ***Socket***s será entonces: Un programa (lo llamaremos "servidor") crea un ***ServerSocket*** en un determinado puerto conocido por el resto de programas. El servidor queda esperando a que algún cliente intente conectar con él. En el

momento en que se establece la conexión, ambos programas (el cliente y el servidor) obtienen un objeto ***Socket***. Mediante objetos ***InputStream*** y ***OutputStream*** obtenidos a través de los objetos ***Socket***, el cliente y el servidor intercambian datos. Uno de los dos programas cierra la conexión.

La parte del cliente ya la hemos visto en los apartados anteriores.

Veamos como se realiza la parte del servidor.

La forma más sencilla de crear un objeto ***ServerSocket*** es indicando el número de puerto al constructor:

```
1.  ServerSocket ss = new ServerSocket(1234);
2.  /* Utilizamos el objeto ServerSocket */
```

Una vez creado, tenemos que quedarnos esperando a que alguien intente realizar la conexión. Esto se consigue mediante la función ***ServerSocket***.accept(). Esta función espera una conexión entrante, y devuelve un objeto de tipo ***Socket***.

```
1.  ServerSocket ss = new ServerSocket(1234);
2.  Socket s = ss.accept();
3.  /* Utilizamos el objeto Socket */
4.  s.close();
```

Una vez el servidor tiene el objeto ***Socket***, puede realizar las mismas acciones que el cliente (extraer los flujos de entrada/salida, cerrar la conexión, etc.)

En el siguiente ejemplo, nuestro servidor espera la conexión entrante y responde con un mensaje de bienvenida:

```
1.  ServerSocket ss = new ServerSocket(1234);
2.  Socket s = ss.accept();
3.  OutputStream  os = s.getOutputStream ();
4.  String mensaje = "¡Conéctate con otro sitio y déjame en paz, pesado!";
5.  byte[] matriz = mensaje.getBytes();
6.
7.  os.write(matriz);
8.  os.close();
9.  s.close();
```

De un objeto ***ServerSocket*** se pueden obtener muchos objetos ***Socket*** diferentes, cada uno independiente de los demás. Por ejemplo, podemos tener un programa que trabaje en el puerto 80 y que asigne cada nueva conexión a un hilo de ejecución distinto. No es necesario que se cierren los objetos ***Socket*** previos antes de poder aceptar una nueva conexión. Esto es lo que hace que los servidores Web puedan atender a varias personas al mismo tiempo, sin tener que esperar a terminar con cada cliente antes de atender al siguiente.

Por ejemplo, supongamos que tenemos una clase de objetos "MiniServidor", que implementan la interfaz Runnable y que están programados para responder a las peticiones que les llegan a través de un ***Socket***. Una posible implementación para el servidor sería:

```
1.  ServerSocket ss = new ServerSocket(1234);
2.  while(true){
3.      Socket s = ss.accept();
4.      MiniServidor m = new MiniServidor(s);
5.      Thread t = new Thread(m);
6.      t.start();
7.      }
```

Gestión de excepciones

Como en todos los casos en que tengamos que trabajar con protocolos de red o sistemas de entrada/salida, tenemos que encargarnos de gestionar las posibles excepciones. Las más comunes son java.io.IOException (para los casos en los que haya problemas con la conexión) y java.net.UnknownHostException (cuando especificamos una dirección IP desconocida o incorrecta).

Un ejemplo cliente/servidor completo usando TCP

Con el fin de comparar ambos métodos, vamos a realizar un par de programas que realicen la misma función que el ejemplo anterior, pero usando el protocolo TCP en vez de UDP.

Veremos una implementación en la que para cada nuevo número transmitido el cliente abre una conexión TCP distinta. Aunque este sistema es perfectamente válido, no es muy eficiente, ya que pierde mucho tiempo en el proceso de establecimiento de las conexiones.

Sería más deseable una implementación en la que todas las peticiones fueran a través del mismo *Socket TCP*.

El funcionamiento del programa es muy similar al del que usaba *Datagramas*. Se realiza un completo control de errores.

```
1.  import java.io.*;
2.  import java.net.*;
3.  class servidor{
4.    public static void main(String args[]){
5.
6.      // Primero indicamos la dirección IP local
7.      try{
8.        System.out.println("LocalHost = " + InetAddress.getLocalHost().toString());
9.        } catch (UnknownHostException uhe){
10.         System.err.println("No puedo saber la dirección IP local : " + uhe);
11.         }
12.
13.
14.     // Abrimos un "Socket de Servidor" TCP en el puerto 1234.
15.     ServerSocket ss = null;
16.     try{
17.       ss = new ServerSocket(1234);
18.       } catch (IOException ioe){
19.         System.err.println("Error al abrir el Socket de servidor : " + ioe);
20.         System.exit(-1);
21.          }
22.
23.     int entrada;
24.     long salida;
25.     // Bucle infinito
26.     while(true){
27.       try{
28.       // Esperamos a que alguien se conecte a nuestro Socket
29.
30.       Socket sckt = ss.accept();
31.
32.       // Extraemos los Streams de entrada y de salida
33.       DataInputStream dis = new
34. DataInputStream(sckt.getInputStream());
35.       DataOutputStream  dos = new DataOutputStream (sckt.getOutputStream ());
36.
37.       // Podemos extraer información del Socket
38.       // Nº de puerto remoto
39.       int puerto = sckt.getPort();
40.
41.      // Dirección de Internet remota
42.       InetAddress direcc = sckt.getInetAddress();
43.
44.       // Leemos datos de la peticion
45.
46.       entrada = dis.readInt();
47.       // Calculamos resultado
48.
49.       salida = (long)entrada*(long)entrada;
50.       // Escribimos el resultado
51.
52.       dos.writeLong(salida);
53.       // Cerramos los streams
54.       dis.close();
55.       dos.close();
56.        sckt.close();
57.       // Registramos en salida estandard
58.       System.out.println(  "Cliente = " + direcc + ":" + puerto +
59.                            "\tEntrada = " + entrada +
60.                            "\tSalida = " + salida );
61.     } catch(Exception e){
62.       System.err.println("Se ha producido la excepción : " +e);
63.       }
64.     }
65.
66.   }
67.
68. }
```

```
1.  class cliente {
2.
3.    public static void main(String args[]){
4.
5.      // Leemos el primer parámetro, donde debe ir la dirección
6.      // IP del servidor
7.
8.      InetAddress direcc = null;
9.      try{
10.       direcc = InetAddress.getByName(args[0]);
11.     } catch(UnknownHostException uhe){
12.       System.err.println("Host no encontrado : " + uhe);
13.       System.exit(-1);
14.       }
15.
16.     // Puerto que hemos usado para el servidor
17.     int puerto = 1234;
18.
19.     // Para cada uno de los argumentos...
20.     for (int n=1;n<args.length;n++){
21.       Socket sckt = null;
22.       DataInputStream dis = null;
23.       DataOutputStream  dos = null;
24.       try{
25.
26.       // Convertimos el texto en número
27.       int numero = Integer.parseInt(args[n]);
28.
29.       // Creamos el Socket
30.
31.       sckt = new Socket(direcc,puerto);
32.
33.       // Extraemos los streams de entrada y salida
34.       dis = new DataInputStream(sckt.getInputStream());
35.
36.       dos = new DataOutputStream (sckt.getOutputStream ());
37.
38.
39.       // Lo escribimos
40.       dos.writeInt(numero);
41.
42.       // Leemos el resultado final
43.
44.       long resultado = dis.readLong();
45.
46.       // Indicamos en pantalla
47.       System.out.println(  "Solicitud = " + numero +
48.                            "\tResultado = " +resultado );
49.       // y cerramos los streams y el Socket
50.       dis.close();
51.       dos.close();
52.       } catch(Exception e){
53.         System.err.println("Se ha producido la excepción : " +e);
54.         }
55.
56.       try{
57.         if (sckt!=null) sckt.close();
58.         } catch(IOException ioe){
59.           System.err.println("Error al cerrar el Socket : " + ioe);
60.           }
61.
62.       }
63.
64.     }
65.   }
```

Aparte de la introducción de los métodos ***Socket***.getPort() y ***Socket***.getInetAddress(), que nos devuelven, respectivamente, el puerto y la dirección IP de la máquina remota, el código anterior únicamente resume las ideas presentadas con anterioridad.

Se deja como "ejercicio para el lector" el cambiar el código anterior para que todas las peticiones que el cliente transmite al servidor vayan por el mismo ***Socket*** *TCP*.

Trabajo con URLs

Una URL (Uniform Resource Locator) es, a grandes rasgos, el nombre de un determinado recurso (archivos, bases de datos, ordenadores, impresoras, etc) en Internet. Por ejemplo, http://www.akal.com/index2.htm es una URL que "apunta" a una página dentro de un servidor Web.

El formato de una URL está definido en el standard RFC 1738, y suele seguir el esquema:

PROTOCOLO://MAQUINA/DIRECTORIO/SUBDIRECTORIO/ARCHIVO

Por ejemplo:

ftp://ftp.microsoft.com/public/file.txt

http://www.etsit.upv.es/iaeste/index.html

Una forma más general, que nos permite especificar el puerto de conexión, así como el login y el ***password*** es la siguiente:

http://login:password@maquina:puerto/dir/subdir/archivo

En Java, las URLs se representan mediante la clase java.net.URL.

Esta clase solo representa la dirección, no su contenido. Para acceder al contenido de un objeto URL necesitamos obtener un objeto java.net.URLConnection, extraído a partir del propio URL.

Por ejemplo, el siguiente código crea una URL que apunta a http://www.javasoft.com, posteriormente obtiene un objeto URLConnection y por último hace algo realmente útil: leer el contenido de la URL:

```
1.  URL direccion = new URL("http://www.javasoft.com");
2.  URLConnection conex = direccion.openConnection();
3.  InputStream entrada = conex.getInputStream();
4.  while((int dato=entrada.read())!=-1){
5.      /* Hacemos algo interesante con lo que leemos */
6.      }
```

El ejemplo anterior es una muestra muy simple de como utilizar las clases URL y URLConnection. En una aplicación real probablemente habría que usar el resto de métodos que nos proporciona la clase URLConnection (para conocer datos del recurso remoto, leer información sobre la comunicación y el tipo de protocolo usado, etc.)

Veamos un ejemplo completo. El siguiente programa simplemente se conecta con una URL especificada en la línea de comandos y guarda el contenido en un archivo.

```
1.  /* Este programa muestra como trabajar con URL's a través de Java */
2.  /* El programa se limita a acceder a una URL, extraer su stream de salida, y leer los datos byte a byte, guardándolos en el archivo especificado */
3.  import java.net.*;
4.  import java.io.*;
5.  class getURL{
6.    public static void main(String params[]){
7.
8.      /* El programa debe recibir dos parámetros:
9.          1.- la URL
10.         2.- el archivo local donde guardar el resultado */
11.     if (params.length<2){
12.       System.err.println("Necesito una URL y un nombre de archivo
13. local válidos");
14.       System.exit(-1);
15.       }
16.     /* Intentamos acceder a la URL */
17.
18.     try{
19.       /* Si la URL fuera incorrecta saltaría al catch de
20. MalformedURLException */
21.       URL miUrl = new URL(params[0]);
22.       /* Obtenemos una URLConnection, mediante openConnection(), y sacamos
23.         un InputStream mediante getInputStream() */
24.
25.       /* Si se produce algún error salta al catch de IOException */
26.       InputStream is = miUrl.openConnection().getInputStream();
27.
28.       /* Abrimos el archivo para escritura */
29.       FileOutputStream fos = new FileOutputStream (params[1]);
30.
31.       int dato;
32.       /* Vamos leyendo bytes hasta que read() nos devuelva -1 */
```

```
33.               while ((dato=is.read()) != -1)
34.                  fos.write(dato);
35.
36.               /* Cerramos todos los streams */
37.               fos.close();
38.               is.close();
39.               }
40.            catch(MalformedURLException errorURL){
41.               System.err.println("La URL " + params[0] + " es incorrecta");
42.               }
43.            catch(IOException errorIO){
44.               System.err.println("Error de entrada/salida : " + errorIO);
45.               }
46.         }   /* fin main */
47.      }   /*fin clase */
```

La forma de invocar el programa sería como sigue:

java getURL http://www.akal.com/index2.htm mifichero.htm

En el código anterior además aparecen las excepciones típicas que aparecen en estos casos (java.net.MalformedURLException, java.io.IOException, etc.)

2.2. Otras tecnologías Java

Sun tiene otros componentes enfocados a Java, al margen de la plataforma central, en varias etapas de especificaciones e implantaciones. Entre ellas cabe citar:

- ?? Java3D. Soporte para imágenes 3D.
- ?? Java Media Framework. Soporte de multimedia.
- ?? Java Servlets. Java para servidores web.
- ?? Java Cryptography Extensions. Un marco para criptografía de clave privada y pública.
- ?? JavaHelp. Un sistema completo de ayuda.
- ?? Jini. Un marco para crear comunidades de dispositivos "inteligentes', incluyendo configuración automática de redes y búsqueda de recursos.
- ?? JavaSpeech. Una API para síntesis y reconocimiento de voz.
- ?? Java 2 Enterprise Edition. Una colección de tecnologías, directorio, bases de datos, correo electrónico, mensajería, transacciones, etc., orientadas al despliegue en el entorno empresarial.

2.3. ¿Dónde se utiliza Java?

Algunos escenarios en los que Java ha encontrado un sitio (comenzando con los dos tradicionales) son los siguientes:

- ?? Aplicaciones Java independientes, gestionadas por un JRE bajo diversos sistemas operativos diferentes: Linux, NT, MacOS, las principales modalidades de UNIX, los SO de los mainframes de IBM, etcétera.
- ?? Los entornos applet JRE proporcionados por los navegadores web Netscape Navigator e Internet Explorer.
- ?? Servidores web, para generación programática de contenidos web.
- ?? Servidores de aplicaciones, integrando las actividades de las aplicaciones corporativas, bases de datos y actividades web.
- ?? Java PC. JavaOS de Sun es un sistema operativo adecuado para uso en computadoras y en sistemas de redes, en los cuales las clases de Java son el formato nativo de las aplicaciones.
- ?? Dentro de sistemas de gestión de bases de datos (DBMS, Database Management Systems) como Oracle y Sybase, soportando procedimientos almacenados para consultas inteligentes a bases de datos.
- ?? Cajas de televisión, set-top boxes, que ejecutan Java IV.
- ?? Tarjetas inteligentes (smart cards). Una máquina virtual de Java completa, más el soporte de datos en la tarjeta, que reside en un chip sobre una pequeña tarjeta de plástico.
- ?? Controladores integrados en dispositivos industriales y de consumo: impresoras, cámaras, robots, etc.

2.4. ¿Qué podemos hacer en Java?

Java es, por muchas razones, el sueño de cualquier ingeniero de computadoras. Incorpora la mayor parte de las tecnologías más interesantes de los últimos veinte años, desde recolección de residuos, a código independiente de la arquitectura, desde optimización sobre la marcha a validación en tiempo de ejecución o a OOP. Muchas de estas tecnologías no se han convertido en fundamentales, en el mundo real, debido a que resultan muy lentas.

Éste es precisamente el problema con Java: es lento. El asunto del rendimiento indudablemente mejorará, pero hay buenas razones para pensar que Java no podrá competir en términos de velocidad con las aplicaciones nativas compiladas. Entre los problemas que Java no puede manejar hoy cabe citar:

?? Problemas críticos de prestaciones. Éstos todavía requieren aplicaciones nativas o, al menos, componentes con código nativo en las aplicaciones Java.

?? Grandes problemas. Los problemas que implican gran cantidad de memoria o requisitos de E/S exigen que la aplicación tome un papel activo en la gestión de memoria o E/S (el afinado de las aplicaciones crea una gran diferencia entre el software utilizable e inservible en áreas tan exigentes como las simulaciones y SGBD). Java no es un entorno favorable para estos problemas.

?? Problemas específicos de la plataforma. Java asume considerables esfuerzos para lograr una independencia de la plataforma, hasta el punto de que no aprovecha muchas de las capacidades que ofrecen los lenguajes nativos o, incluso, los lenguajes de guiones independientes de la plataforma. Por ejemplo, no podemos, sin escribir componentes en código nativo, detectar o crear un enlace simbólico, implementar un gestor X Window, leer variables de entorno Unix, identificar el usuario de un fichero, cambiar los ajustes del TTY, etc.

?? GUI (por supuesto que Java tiene una GUI, una interfaz gráfica de usuario). Swing es una herramienta de primera clase. Pero el rendimiento de la GUI necesita que se le preste una gran atención si Java pretende convertirse en una plataforma GUI seria. Actualmente, Java está disfrutando de un éxito mucho mayor en entornos no GUI, como servidores, que en entornos GUI como los Applets.

Parece que Java está en todas partes..., bueno, ciertamente ha desatado la imaginación colectiva de los mundos de la computación y los sistemas de redes. En realidad, la tecnología Java es una mezcla compleja de software y marketing, y vive en un ambiente cargado de intensas peleas entre ***Sun*** Microsystems, sus competidores, sus socios, los tribunales y las comunidades de usuarios y desarrolladores. Java no es, ciertamente, la solución a todos los problemas, pero es (como Linux) un lugar interesante para trabajar, jugar y construir el futuro de Internet.

Temario