11: El sistema de E/S de Java

Crear un buen sistema de entrada/salida (E/S) es una de las tareas más difíciles para el diseñador de un lenguaje.

Esto es evidente con sólo observar la cantidad de enfoques diferentes. El reto parece estar en cubrir todas las posibles eventualidades. No sólo hay distintas fuentes y consumidores de información de E/S con las que uno desea comunicarse (archivos, la consola, conexiones de red), pero hay que comunicarse con ellos de varias maneras (secuencial, acceso aleatorio, espacio de almacenamiento intermedio, binario, carácter, mediante líneas, mediante palabras, etc.).

Los diseñadores de la biblioteca de Java acometieron este problema creando muchas clases. De hecho, hay tantas clases para el sistema de E/S de Java que puede intimidar en un principio (irónicamente, el diseño de la E/S de Java evita una explosión de clases). También hubo un cambio importante en la biblioteca de Java después de la versión 1.0, al suprimir la biblioteca original orientada a **byte**s por clases de E/S orientadas a **char** basadas en Unicode. Como resultado hay que aprender un número de clases aceptable antes de entender suficientemente un esbozo de la E/S de Java para poder usarla adecuadamente. Además, es bastante importante entender la historia de la biblioteca de E/S, incluso si tu primera reacción es: "¡No me aburras con esta historia, simplemente dime cómo usarla!" El problema es que sin la perspectiva histórica es fácil confundirse con algunas de las clases, y no comprender cuándo deberían o no usarse.

Este capítulo presentará una introducción a la variedad de clases de E/S contenidas en la biblioteca estándar de Java, y cómo usarlas.

La clase **File**

Antes de comenzar a ver las clases que realmente leen y escriben datos en flujos, se echará un vistazo a una utilidad proporcionada por la biblioteca para manejar aspectos relacionados con directorios de archivos.

La clase **File** tiene un nombre engañoso —podría pensarse que hace referencia a un archivo, pero no es así. Puede representar, o bien el *nombre* de un archivo particular, o los *nombres* de un conjunto de archivos de un directorio. Si se trata de un conjunto de archivos, se puede preguntar por el conjunto con el método **list( )**, que devuelve un array de **String**s. Tiene sentido devolver un array en vez de una de las clases contenedoras flexibles porque el número de elementos es fijo, y si se desea listar un directorio diferente basta con crear un objeto **File** diferente. De hecho, "FilePath" habría sido un nombre mejor para esta clase. Esta sección muestra un ejemplo de manejo de esta clase, incluyendo la **interfaz FilenameFilter** asociada.

Un generador de listados de directorio

Suponga que se desea ver el contenido de un directorio. El objeto **File** puede listarse de dos formas. Si se llama a **list( )** sin parámetros, se logrará la lista completa de lo que contiene el objeto **File**. Sin embargo, si se desea una lista restringida —por ejemplo, si se desean todos los archivos de extensión **.java**— se usará un "filtro de directorio", que es una clase que indica cómo seleccionar los objetos **File** a mostrar.

He aquí el código para el ejemplo. Nótese que el resultado se ha ordenado sin ningún tipo de esfuerzo, de forma alfabética, usando el método **java.utils.Array.sort( )** y el **ComparadorAlfabetico** definido en el Capítulo 9:

```
//: c11:ListadoDirectorio.java
// Muestra listados de directorios.
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class ListadoDirectorio {
  public static void main(String[] args) {
    File ruta = new File(".");
    String[] lista;
    if(args.length == 0)
      lista = ruta.list();
    else
      lista = ruta.list(new FiltroDirectorio(args[0]));
    Arrays.sort(lista,
      new ComparadorAlfabetico());
    for(int i = 0; i < lista.length; i++)
      System.out.println(lista[i]);
  }
}

class FiltroDirectorio implements FilenameFilter {
  String afn;
  FiltroDirectorio(String afn) { this.afn = afn; }
  public boolean accept(File dir, String name) {
    // Retirar información de ruta:
    String f = new File(name).getName();
    return f.indexOf(afn) != -1;
  }
} ///:~
```

La clase **FiltroDirectorio** "implementa" la **interfaz FilenameFilter**. Es útil ver lo simple que es la **interfaz FilenameFilter**:

```
public interface FilenameFilter {
  boolean accept (File dir, String name);
}
```

Dice que este tipo de objeto proporciona un método denominado **accept( )**. La razón que hay detrás de la creación de esta clase es proporcionar el método **accept( )** al método **list( )** de forma que **list( )** pueda "retrollamar" a **accept( )** para determinar qué nombres deberían ser incluidos en la lista. Por consiguiente, a esta técnica se le suele llamar *retrollamada* o a veces *functor* (es decir, **FiltroDirectorio** es un *functor* porque su único trabajo es albergar un método) o *Patrón Comando*. Dado que **list( )** toma un objeto **FilenameFilter** como parámetro, se le puede pasar un objeto de cualquier clase que implemente **FilenameFilter** para elegir (incluso en tiempo de ejecución) cómo se comportará el método **list( )**. El propósito de una retrollamada es proporcionar flexibilidad al comportamiento del código.

FiltroDirectorio muestra que, justo porque una **interfaz** contenga sólo un conjunto de métodos, uno no está restringido a escribir sólo esos métodos. (Sin embargo, al menos hay que proporcionar definiciones para todos los métodos de la interfaz.) En este caso, se crea también el constructor **FiltroDirectorio**.

El método **accept( )** debe aceptar un objeto **File** que represente el directorio en el que se encuentra un archivo en particular, y un **String** que contenga el nombre de ese archivo. Se podría elegir entre utilizar o ignorar cualquiera de estos parámetros, pero probablemente se usará al menos el nombre del archivo. Debe recordarse que el método **list( )** llama a **accept( )** por cada uno de los nombres de archivo del objeto directorio para ver cuál debería incluirse —lo que se indica por el resultado **boolean** devuelto por **accept( )**.

Para asegurarse de que el elemento con el que se está trabajando es sólo un nombre de archivo sin información de ruta, todo lo que hay que hacer es tomar el **String** y crear un objeto **File** a partir del mismo, después llamar a **getName( )**, que retira toda la información relativa a la ruta (de forma independiente de la plataforma). Después, **accept( )** usa el método **indexOf( )** de la clase **String** para ver si la cadena de caracteres a buscar **afn** aparece en algún lugar del nombre del archivo. Si se encuentra **afn** en el string, el valor devuelto es el índice de comienzo de **afn**, mientras que si no se encuentra, se devuelve el valor -1. Hay que ser conscientes de que es una búsqueda de cadenas de caracteres simple y que no tiene expresiones de emparejamiento de comodines —como por ejemplo "for?.b?*"— lo cual sería más difícil de implementar.

El método **list( )** devuelve un array. Se puede preguntar por la longitud del mismo y recorrerlo seleccionando sus elementos. Esta habilidad de pasar un array hacia y desde un método es una gran mejora frente al comportamiento de C y C++.

Clases internas anónimas

Este ejemplo es ideal para reescribirlo utilizando una clase interna anónima (descritas en el Capítulo 8). En principio, se crea un método **filtrar( )** que devuelve una referencia a **FilenameFilter**:

```
//: c11:ListadoDirectorio2.java
// Usa clases internas anónimas.
import java.io.*;
```

```
import java.util.*;
import com.bruceeckel.util.*;

public class ListadoDirectorio2 {
  public static FilenameFilter
  filtrar(final String afn) {
    // Creación de la clase interna anónima:
    return new FilenameFilter() {
      String fn = afn;
      public boolean accept(File dir, String n) {
        // Retirar información de ruta:
        String f = new File(n).getName();
        return f.indexOf(fn) != -1;
      }
    }; // Fin de la clase interna anónima
  }
  public static void main(String[] args) {
    File ruta = new File(".");
    String[] lista;
    if(args.length == 0)
      lista = ruta.list();
    else
      lista = ruta.list(filtro(args[0]));
    Arrays.sort(lista,
      new ComparadorAlfabetico());
    for(int i = 0; i < lista.length; i++)
      System.out.println(lista[i]);
  }
} ///:~
```

Nótese que el parámetro que se pase a **filtrar( )** debe ser **final**. Esto es necesario para que la clase interna anónima pueda usar un objeto de fuera de su ámbito.

El diseño es una mejora porque la clase **FilenameFilter** está ahora firmemente ligada a **ListadoDirectorio2**. Sin embargo, es posible llevar este enfoque un paso más allá y definir la clase interna anónima como un argumento de **list( )**, en cuyo caso es incluso más pequeña:

```
//: c11:ListadoDirecctorio3.java
// Construyendo la clase interna anónima "en el lugar".
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class ListadoDirectorio3 {
  public static void main(final String[] args) {
    File ruta = new File(".");
    String[] lista;
```

```
    if(args.length == 0)
      lista = ruta.list();
    else
      lista = ruta.list(new FilenameFilter() {
        public boolean
        accept(File dir, String n) {
          String f = new File(n).getName();
          return f.indexOf(args[0]) != -1;
        }
      });
    Arrays.sort(lista,
      new ComparadorAlfabetico());
    for(int i = 0; i < lista.length; i++)
      System.out.println(lista[i]);
  }
} ///:~
```

El argumento del **main( )** es ahora **final**, puesto que la clase interna anónima usa directamente **args[0]**.

Esto muestra cómo las clases anónimas internas permiten la creación de clases rápida y limpiamente para solucionar problemas. Puesto que todo en Java se soluciona con clases, ésta puede ser una técnica de codificación útil. Un beneficio es que mantiene el código que soluciona un problema en particular aislado y junto en el mismo sitio. Por otro lado, no es siempre fácil de leer, por lo que hay que usarlo juiciosamente.

Comprobando y creando directorios

La clase **File** es más que una simple representación de un archivo o un directorio existentes. También se puede usar un objeto **File** para crear un nuevo directorio o una trayectoria de directorio completa si ésta no existe. También se pueden mirar las características de archivos (tamaño, fecha de la última modificación, lectura/escritura), ver si un objeto **File** representa un archivo o un directorio, y borrar un archivo. Este programa muestra algunos de los otros métodos disponibles con la clase **File** (ver la documentación HTML de *http://java.sun.com* para obtener el conjunto completo):

```
//: c11:CrearDirectorios.java
// Demuestra el uso de la clase File para
// crear directorios y manipular archivos.
import java.io.*;

public class CrearDirectorios {
  private final static String uso =
    "Uso:CrearDirectorios ruta1 ...\n" +
    "Crea cada ruta\n" +
    "Uso:CrearDirectorios -d ruta1 ...\n" +
```

```
    "Borra cada ruta\n" +
    "Uso:CrearDirectorios -r ruta1 ruta2\n" +
    "Renombra ruta1 a ruta2\n";
  private static void uso() {
    System.err.println(uso);
    System.exit(1);
  }
  private static void datosArchivo(File f) {
    System.out.println(
      "Ruta absoluta: " + f.getAbsolutePath() +
      "\n Puede leer: " + f.canRead() +
      "\n Puede escribir: " + f.canWrite() +
      "\n Conseguir el nombre: " + f.getName() +
      "\n Conseguir su padre: " + f.getParent() +
      "\n Conseguir ruta: " + f.getPath() +
      "\n Longitud: " + f.length() +
      "\n Ultima modificacion: " + f.lastModified());
    if(f.isFile())
      System.out.println("Es un archivo");
    else if(f.isDirectory())
      System.out.println("Es un directorio");
  }
  public static void main(String[] args) {
    if(args.length < 1) uso();
    if(args[0].equals("-r")) {
      if(args.length != 3) uso();
      File
        viejo = new File(args[1]),
        nuevo = new File(args[2]);
      viejo.renameTo(nuevo);
      datosArchivo(viejo);
      datosArchivo(nuevo);
      return; // Salir del main
    }
    int contador = 0;
    boolean borrar = false;
    if(args[0].equals("-d")) {
      contador++;
      borrar = true;
    }
    for( ; contador < args.length; contador++) {
      File f = new File(args[contador]);
      if(f.exists()) {
        System.out.println(f + " existe");
        if(borrar) {
```

```
            System.out.println("borrando..." + f);
            f.delete();
          }
        }
        else { // No existe
          if(!borrar) {
            f.mkdirs();
            System.out.println("creado " + f);
          }
        }
        datosArchivo(f);
      }
    }
} ///:~
```

En **datosArchivo( )** se pueden ver varios métodos de investigación que se usan para mostrar la información sobre la trayectoria de un archivo o un directorio.

El primer método ejercitado por el método **main( )** es **renameTo( )**, que permite renombrar (o mover) un archivo a una ruta totalmente nueva representada por un parámetro, que es otro objeto **File**. Esto también funciona con directorios de cualquier longitud.

Si se experimenta con el programa, se verá que se pueden construir rutas de cualquier complejidad, pues **mkdirs( )** hará todo el trabajo.

Entrada y salida

Las bibliotecas de E/S usan a menudo la abstracción del flujo, que representa cualquier fuente o consumidor de datos como un objeto capaz de producir o recibir fragmentos de código. El flujo oculta los detalles de lo que ocurre con los datos en el dispositivo de E/S real.

Las clases de E/S de la biblioteca de Java se dividen en entrada y salida, como ocurre con los datos en el dispositivo de E/S real. Por herencia, todo lo que deriva de las clases **InputStream** o **Reader** tiene los métodos básicos **read( )** para leer un simple byte o un array de bytes. Asimismo, todo lo que derive de las clases **OutputStream** o **Writer** tiene métodos básicos denominados **write( )** para escribir un único byte o un array de bytes. Sin embargo, generalmente no se usarán estos métodos; existen para que otras clases puedan utilizarlos —estas otras clases proporcionan una interfaz más útil. Por consiguiente, rara vez se creará un objeto flujo usando una única clase, sino que se irán apilando en capas diversos objetos para proporcionar la funcionalidad deseada. El hecho de crear más de un objeto para crear un flujo resultante único es la razón primaria por la que la biblioteca de flujos de Java es tan confusa.

Ayuda bastante clasificar en tipos las clases en base a su funcionalidad. En Java 1.0, los diseñadores de bibliotecas comenzaron decidiendo que todas las clases relacionadas con entrada heredarían de **InputStream**, y que todas las asociadas con la salida heredarían de **OutputStream**.

Tipos de **InputStream**

El trabajo de **InputStream** es representar las clases que producen entradas desde distintas fuentes. Éstas pueden ser:

1. Un array de bytes.
2. Un objeto **String**.
3. Un archivo.
4. Una "tubería", que funciona como una tubería física: se ponen elementos en un extremo y salen por el otro.
5. Una secuencia de otros flujos, de forma que se puedan agrupar todos en un único flujo.
6. Otras fuentes, como una conexión a Internet (esto se verá al final del capítulo).

Cada una de éstas tiene asociada una subclase de **InputStream**. Además, el **FilterInputStream** es también un tipo de **InputStream**, para proporcionar una clase base para las clases "decoradoras" que adjuntan atributos o interfaces útiles a los flujos de entrada. Esto se discutirá más tarde.

Tabla 11-1. Tipos de InputStream

Clase	Función	Parámetros del constructor Cómo usarla
ByteArray-InputStream	Permite usar un espacio de almacenamiento intermedio de memoria como un **InputStream**	El intermedio del que extraer los bytes. Espacio de almacenamiento. Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar un interfaz útil.
StringBuffer-InputStream	Convierte un **String** en un **InputStream**	Un **String**. La implementación subyacente usa, de hecho, un **StringBuffer**.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
File-InputStream	Para leer información de un archivo	Un **String** que represente al nombre del archivo, o un objeto **File** o **FileDescriptor**.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.

Clase	Función	Parámetros del constructor / Cómo usarla
PipedInputStream	Produce los datos que se están escribiendo en el **PipedOutputStream** asociado. Implementa el concepto de "entubar".	**PipedOutputStream.** Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
SequenceInputStream	Convierte dos o más objetos **InputStream** en un **InputStream** único.	Dos objetos **InputStream** o una **Enumeration** para contener objetos **InputStream.** Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
FilterInputStream	Clase abstracta que es una interfaz para los decoradores que proporcionan funcionalidad útil a otras clases **InputStream.** Ver Tabla 11-3.	Ver Tabla 11-3. Ver Tabla 11-3.

Tipos de **OutputStream**

Esta categoría incluye las clases que deciden dónde irá la salida: un array de bytes (sin embargo, no **String**; presumiblemente se puede crear uno usando un array de bytes), un fichero, o una "tubería".

Además, el **FilterOutputStream** proporciona una clase base para las clases "decorador" que adjuntan atributos o interfaces útiles de flujos de salida. Esto se discute más tarde.

Tabla 11-2. Tipos de OutputStream

Clase	Función	Parámetros del constructor / Cómo usarla
ByteArrayOutputStream	Crea un espacio de almacenamiento intermedio en memoria. Todos los datos que se envían al flujo se ubican en este espacio de almacenamiento intermedio.	Tamaño opcional inicial del espacio de almacenamiento intermedio. Para designar el destino de los datos. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.

Clase	Función	Parámetros del constructor Cómo usarla
File-OutputStream	Para enviar información a un archivo.	Un **String**. que representa el nombre de archivo, o un objeto **File** o un objeto **FileDescriptor.** Para designar el destino de los datos. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.
Piped-OutputStream	Cualquier información que se desee escribir aquí acaba automáticamente como entrada del **PipedInputStream** asociado. Implementa el concepto de "entubar".	**PipedInputStream** Para designar el destino de los datos para multihilo. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.
Filter-OutputStream	Clase abstracta que es una interfaz para los decoradores que proporcionan funcionalidad útil a las otras clases **OutputStream.** Ver Tabla 11-4.	Ver Tabla 11-4. Ver Tabla 11-4.

Añadir atributos e interfaces útiles

Al uso de objetos en capas para añadir dinámica y transparentemente responsabilidades a objetos individuales se le denomina patrón *Decorador*. (Los Patrones[1] son el tema central de *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com.*) El patrón decorador especifica que todos los objetos que envuelvan el objeto inicial tengan la misma interfaz. Así se hace uso de la transparencia del decorador —se envía el mismo mensaje a un objeto esté o no decorado. Éste es el motivo de la existencia de clases "filtro" en la biblioteca E/S de Java: la clase abstracta "filter" es la clase base de todos los decoradores. (Un decorador debe tener la misma interfaz que el objeto que decora, pero el decorador también puede extender la interfaz, lo que ocurre en muchas de las clases "filter".)

Los decoradores se han usado a menudo cuando se generan numerosas subclases para satisfacer todas las combinaciones posibles necesarias —tantas clases que resultan poco prácticas. La biblio-

[1] *Desing Patterns*, Erich Gamma *et al.*, Addison-Wesley 1995.

teca de E/S de Java requiere muchas combinaciones distintas de características, siendo ésta la razón del uso del patrón decorador. Sin embargo este patrón tiene un inconveniente. Los decoradores dan mucha más flexibilidad al escribir un programa (puesto que se pueden mezclar y emparejar atributos fácilmente), pero añaden complejidad al código. La razón por la que la biblioteca de E/S es complicada de usar es que hay que crear muchas clases —los tipos de E/S básicos más todos los decoradores— para lograr el único objeto de E/S que se quiere.

Las clases que proporcionan la interfaz decorador para controlar un **InputStream** o un **Output Stream** particular son **FilertInputStream** y **FilterOuputStream** —que no tienen nombres muy intuitivos. Éstas dos últimas clases son abstractas, derivadas de las clases base de la biblioteca de E/S, **InputStream** y **OutputStream**, el requisito clave del decorador (de forma que proporciona la interfaz común a todos los objetos que se están decorando).

Leer de un **InputStream** con un **FilterInputStream**

Las clases **FilterInputStream** llevan a cabo dos cosas significativamente diferentes. **DataInput Stream** permite leer distintos tipos de datos primitivos, además de objetos **String**. (Todos los métodos empiezan por "read", como **readByte( )**, **readFloat( )**, etc.) Esto, junto con su compañero **DataOutputStream**, permite mover datos primitivos de un sitio a otro vía un flujo. Estos "lugares" vendrán determinados por las clases de la Tabla 11-1.

Las clases restantes modifican la forma de comportarse internamente de **InputStream**: haga uso o no de espacio de almacenamiento intermedio, si mantiene un seguimiento de las líneas que lee (permitiendo preguntar por el número de líneas, o establecerlo) o si se puede introducir un único carácter. Las dos últimas clases se parecen mucho al soporte para la construcción de un compilador (es decir, se añadieron para poder construir el propio compilador de Java), por lo que probablemente no se usen en programación en general.

Probablemente se necesitará pasar la entrada por un espacio de almacenamiento intermedio casi siempre, independientemente del dispositivo de E/S al que se esté conectado, por lo que tendría más sentido que la biblioteca de E/S fuera un caso excepcional (o simplemente una llamada a un método) de entrada sin espacio de almacenamiento intermedio, en vez de una entrada con espacio de almacenamiento intermedio.

Tabla 11-3. Tipos de FilterInputStream

Clase	Función	Parámetros del constructor Cómo usarla
Data-InputStream	Usado junto con **DataOutputStream**, de forma que se puedan leer datos primitivos (**int**, **char**, **long**, etc.) de un flujo de forma portable.	**InputStream** Contiene una interfaz completa que permite leer tipos primitivos.

Clase	Función	Parámetros del constructor Cómo usarla
BufferedInputStream	Se usa para evitar una lectura cada vez que se soliciten nuevos datos. Se está diciendo "utiliza un espacio de almacenamiento intermedio".	**InputStream**, con tamaño de espacio de almacenamiento intermedio opcional. No proporciona una interfaz *per se*, simplemente el requisito de que se use un espacio de almacenamiento intermedio. Adjuntar un objeto interfaz.
LineNumberInputStream	Mantiene un seguimiento de los números de línea en el *flujo* de entrada; se puede llamar a **getLineNumber( )** y a **setLineNumber(int)**.	**InputStream** Simplemente añade numeración de líneas, por lo que probablemente se adjunte un objeto interfaz.
PushbackInputStream	Tiene un espacio de almacenamiento intermedio de un byte para el último carácter a leer.	**InputStream.** Se usa generalmente en el escáner de un compilador y probablemente se incluyó porque lo necesitaba el compilador de Java. Probablemente prácticamente nadie la utilice.

Escribir en un **OutputStream** con **FilterOutputStream**

El complemento a **DataInputStream** es **DataOutputStream**, que da formato a los tipos primitivos y objetos **String**, convirtiéndolos en un flujo de forma que cualquier **DataInputStream**, de cualquier máquina, los pueda leer. Todos los métodos empiezan por "write", como **writeByte( )**, **writeFloat( )**, etc.

La intención original para **PrintStream** era que imprimiera todos los tipos de datos primitivos así como los objetos **String** en un formato visible. Esto es diferente de **DataOutputStream**, cuya meta es poner elementos de datos en un flujo de forma que **DataInputStream** pueda reconstruirlos de forma portable.

Los dos métodos importantes de **PrintStream** son **print( )** y **println( )**, sobrecargados para imprimir todo los tipos. La diferencia entre **print( )** y **println( )** es que la útlima añade una nueva línea al acabar.

PrintStream puede ser problemático porque captura todas las **IOExpections**. (Hay que probar explícitamente el estado de error con **checkError( )**, que devuelve **true** si se ha producido algún error.) Además, **PrintStream** no se internacionaliza adecuadamente y no maneja saltos de línea independientemente de la plataforma (estos problemas se solucionan con **PrintWriter**).

BufferedOutputStream es un modificador que dice al flujo que use espacios de almacenamiento intermedio, de forma que no se realice una lectura física cada vez que se escribe en el flujo. Probablemente siempre se deseará usarlo con archivos, y probablemente la E/S de consola.

Tabla 11-4. Tipos de FilterOutputStream

Clase	Función	Parámetros del constructor Cómo usarla
Data-OutputStream	Usado junto con **DataInputStream**, de forma que se puedan escribir datos primitivos (int, char, long, etc.) de un flujo de forma portable.	**OutputStream** Contiene una interfaz completa que permite escribir tipos de datos primitivos.
PrintStream	Para producir salida formateada. Mientras que **DataOutputStream** maneja el *almacenamiento* de datos, **PrintStream** maneja su *visualización*.	**OutputStream** con un boolean opcional que indica que se vacía el espacio de almacenamiento intermedio con cada nueva línea. Debería ser la envoltura "final" del objeto **OutputStream.** Probablemente se usará mucho.
Buffered-OutputStream	Se usa para evitar una escritura física cada vez que se envía un fragmento de datos. Se está diciendo "Usar un espacio de almacenamiento intermedio". Se puede llamar a **flush( )** para vaciar el espacio de almacenamiento intermedio.	**OutputStream** con tamaño del espacio de almacenamiento intermedio. No proporciona una interfaz *per se*, simplemente pide que se use un espacio de almacenamiento intermedio. Adjuntar un objeto interfaz.

Readers & Writers

Java 1.1. hizo algunas modificaciones fundamentales a la biblioteca de flujos de E/S fundamental de Java (sin embargo, Java 2 no aportó modificaciones significativas). Cuando se observan las clases **Reader** y **Writer**, en un principio se piensa (como hicimos) que su intención es reemplazar las clases **InputStream** y **OutputStream**. Pero ése no es el caso. Aunque se desecharon algunos aspectos de la biblioteca de flujos original (si se usan estos aspectos se recibirá un aviso del compilador), las clases **InputStream** y **OutputStream** siguen proporcionando una funcionalidad valiosa en la forma de E/S orientada al **byte**, mientras que las clases **Reader** y **Writer** proporcionan E/S compatible Unicode basada en caracteres. Además:

1. Java 1.1 añadió nuevas clases a la jerarquía **InputStream** y **OutputStream**, por lo que es obvio que no se estaban reemplazando estas clases.

2. Hay ocasiones en las que hay que usar clases de la jerarquía "byte" *en combinación con* clases de la jerarquía "carácter". Para lograr esto hay clases "puente": **InputStreamReader** convierte un **InputStream** en un **Reader**, y **OutputStreamWriter** convierte un **OutputStream** en un **Writer**.

La razón más importante para la existencia de las jerarquías **Reader** y **Writer** es la internacionalización. La antigua jerarquía de flujos de E/S sólo soporta flujos de 8 bits no manejando caracteres Unicode de 16 bits correctamente. Puesto que Unicode se usa con fines de internacionalización (y el **char** nativo de Java es Unicode de 16 bits), se añadieron las jerarquías **Reader** y **Writer** para dar soporte Unicode en todas las operaciones de E/S. Además, se diseñaron las nuevas bibliotecas de forma que las operaciones se llevaran a cabo de forma más rápida que antiguamente.

En la práctica y en este libro, intentaremos proporcionar un repaso de las clases, pero asumimos que se usará la documentación *en línea* para concretar todos los detalles, así como una lista exhaustiva de los métodos.

Fuentes y consumidores de datos

Casi todas las clases de flujos de E/S de Java originales tienen sus correspondientes clases **Reader** y **Writer** para proporcionar manipulación Unicode nativa. Sin embargo, hay algunos lugares en los que la solución correcta la constituyen los **InputStreams** y **OutputStreams** orientados a **byte**; concretamente, las bibliotecas **java.util.zip** son orientadas a **byte** en vez de orientadas a **char**. Por tanto, el enfoque más sensato es *intentar* usar las clases **Reader** y **Writer** siempre que se pueda, y se descubrirán posteriormente aquellas situaciones en las que hay que usar las otras bibliotecas, ya que el código no compilará.

He aquí una tabla que muestra la correspondencia entre fuentes y consumidores de información (es decir, de dónde y a dónde van físicamente los datos) dentro de ambas jeraquías:

Fuentes & Cusumidores: Clase Java 1.0	**Clase Java 1.1 correspondiente**
InputStream	**Reader** convertidor: **InputStreamReader**
OutputStream	**Writer** convertidor: **OutputStreamWriter**

FileInputStream	**FileReader**
FileOutputStream	**FileWriter**
StringBufferInputStream	**StringReader**
(sin clase correspondiente)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

En general, se descubrirá que las interfaces de ambas jerarquías son similares cuando no idénticas.

Modificar el comportamiento del flujo

En el caso de **InputStreams** y **OutputStreams** se adaptaron los flujos para necesidades particulares utilizando subclases "decorador" de **FilterInputStream** y **FilterOutputStream**. Las jerarquías de clases **Reader** y **Writer** continúan usando esta idea —aunque no exactamente.

En la tabla siguiente, la correspondencia es una aproximación más complicada que en la tabla anterior. La diferencia se debe a la organización de las clases: mientras que **BufferedOutputStream** es una subclase de **FilterOuputStream**, **BufferedWriter** *no* es una subclase de **FilterWriter** (que, aunque es **abstract**, no tiene subclases, por lo que parece haberse incluido como contenedor o simplemente de forma que nadie la busque sin fruto). Sin embargo, las interfaces de las clases coinciden bastante:

Filtros Clase Java 1.0	**Clase Java 1.1 correspondiente**
FilterInputStream	**FilterReader**
FilterOutputStream	**FilterWriter** (clase abstracta sin subclases)
BufferedInputStream	**BufferedReader** (también tiene **readLine( )**)
BufferedOutputStream	**BufferedWriter**
DataInputStream	Usar **DataInputStream** (Excepto cuando se necesite usar **readLine( )**, caso en que debería usarse un **BufferedReader**)
PrintStream	**PrintWriter**

LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer (usar en vez de ello el constructor que toma un Reader)
PushBackInputStream	PushBackReader

Hay algo bastante claro: siempre que se quiera usar un **readLine( )**, no se debería hacer con un **DataInputStream** nunca más (se mostrará en tiempo de compilación un mensaje indicando que se trata de algo obsoleto), sino que debe usarse en su lugar un **BufferedReader**. **DataInputStream**, sigue siendo un miembro "preferente" de la biblioteca de E/S.

Para facilitar la transición de cara a usar un **PrintWriter**, éste tiene constructores que toman cualquier objeto **OutputStream**, además de objetos **Writer**. Sin embargo, **PrintWriter** no tiene más soporte para dar formato que el proporcionado por **PrintStream**; las interfaces son prácticamente las mismas.

El constructor **PrintWriter** también tiene la opción de hacer vaciado automático, lo que ocurre tras todo **println( )** si se ha puesto a uno el *flag* del constructor.

Clases no cambiadas

Java 1.1 no cambió algunas clases de Java 1.0:

Clases de Java 1.0 sin clases correspondientes Java 1.1
DataOutputStream
File
RandomAccessFile
SequenceInputStream

DataOutputStream, en particular, se usa sin cambios, por tanto, para almacenar y recuperar datos en un formato transportable se usan las jerarquías **InputStream** y **OutputStream**.

Por sí mismo: RandomAccessFile

RandomAccessFile se usa para los archivos que contengan registros de tamaño conocido, de forma que se puede mover de un registro a otro utilizando **seek( )**, para después leer o modificar los registros. Éstos no tienen por qué ser todos del mismo tamaño; simplemente hay que poder determinar lo grandes que son y dónde están ubicados dentro del archivo.

En primera instancia, cuesta creer que **RandomAccessFile** no es parte de la jeraquía **InputStream** o **OutputStream**. Sin embargo, no tiene ningún tipo de relación con esas jerarquías con la excepción de que implementa las interfaces **DataInput** y **DataOutput** (que también están implementados por **DataInputStream** y **DataOutputStream**). Incluso no usa ninguna funcionalidad de las clases **InputStream** u **OutputStream** existentes —es una clase totalmente independiente, escrita de la nada, con métodos exclusivos (en su mayoría nativos). La razón para ello puede ser que **RandomAccessFile** tiene un comportamiento esencialmente distinto al de otros tipos de E/S, puesto que se puede avanzar y retroceder dentro de un archivo. Permanece aislado, como un descendiente directo de **Object**.

Fundamentalmente, un **RandomAccessFile** funciona igual que un **DataInputStream** unido a un **DataOutputStream**, junto con los métodos **getFilePointer( )** para averiguar la posición actual en el archivo, **seek( )** para moverse a un nuevo punto del archivo, y **length( )** para determinar el tamaño máximo del mismo. Además, los constructores requieren de un segundo parámetro (idéntico al de **fopen( )** en C) que indique si se está simplemente leyendo ("**r**") al azar, o leyendo y escribiendo ("**rw**"). No hay soporte para archivos de sólo escritura, lo cual podría sugerir que **RandomAccessFile** podría haber funcionado bien si hubiera heredado de **DataInputStream**.

Los métodos de búsqueda sólo están disponibles en **RandomAccessFile**, que sólo funciona para archivos. **BufferedInputStream** permite **mark( )** una posición (cuyo valor se mantiene en una variable interna única) y hacer un **reset( )** a esa posición, pero no deja de ser limitado y, por tanto, no muy útil.

Usos típicos de flujos de E/S

Aunque se pueden combinar las clases *de flujos* de E/S de muchas formas, probablemente cada uno sólo haga uso de unas pocas combinaciones. Se puede usar el siguiente ejemplo como una referencia básica; muestra la creación y uso de las configuraciones de E/S típicas. Nótese que cada configuración empieza con un número comentado y un título que corresponde a la cabecera de la explicación apropiada que sigue en el texto.

```
//: c11:DemoFlujoES.java
// Configuraciones típicas de flujos de E/S.
import java.io.*;

public class DemoFlujoES {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    // 1. Leyendo de la entrada línea a línea:
    BufferedReader entrada =
      new BufferedReader(
        new FileReader("DemoflujoES.java"));
    String S, S2 = new String();
    while((S = entrada.readLine())!= null)
```

```
    S2 += S + "\n";
entrada.close();

// 1b. Leyendo de la entrada estándar:
BufferedReader entradaEstandar =
    new BufferedReader(
      new InputStreamReader(System.in));
System.out.print("Introduce una linea:");
System.out.println(entradaEstandar.readLine());

// 2. Entrada desde memoria
StringReader entrada2 = new StringReader(S2);
int c;
while((c = entrada2.read()) != -1)
    System.out.print((char)c);

// 3. Entada con formato desde memoria
try {
    DataInputStream entrada3 =
      new DataInputStream(
        new ByteArrayInputStream(s2.getBytes()));
    while(true)
      System.out.print((char)entrada3.readByte());
} catch(EOFException e) {
    System.err.println("Fin del flujo");
}

// 4. Salida de archivo
try {
    BufferedReader entrada4 =
      new BufferedReader(
        new StringReader(s2));
    PrintWriter salida1 =
      new PrintWriter(
        new BufferedWriter(
          new FileWriter("DemoES.out")));
    int contadorLineas = 1;
    while((s = entrada4.readLine()) != null )
      salida1.println(contadorLineas++ + ": " + s);
    salida1.close();
} catch(EOFException e) {
    System.err.println("Fin del flujo");
}

// 5. Almacenando & recuperando datos
```

```
try {
  DataOutputStream salida2 =
    new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Datos.txt")));
  salida2.writeDouble(3.14159);
  salida2.writeChars("Eso era pi\n");
  salida2.writeBytes("Eso era pi\n");
  salida2.close();
  DataInputStream entrada5 =
    new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Datos.txt")));
  BufferedReader entrada5br =
    new BufferedReader(
      new InputStreamReader(entrada5));
  // Hay que usar DataInputStream para datos:
  System.out.println(entrada5.readDouble());
  // Ahora se puede usar el readLine() "apropiado":
  System.out.println(entrada5br.readLine());
  // Pero la línea resulta divertida.
  // La creada con writeBytes es correcta:
  System.out.println(entrada5br.readLine());
} catch(EOFException e) {
  System.err.println("Fin del flujo");
}

// 6. Leyendo/escribiendo archivos de acceso directo
RandomAccessFile rf =
  new RandomAccessFile("rprueba.dat", "rw");
for(int i = 0; i < 10; i++)
  rf.writeDouble(i*1.414);
rf.close();

rf =
  new RandomAccessFile("rprueba.dat", "rw");
rf.seek(5*8);
rf.writeDouble(47.0001);
rf.close();

rf =
  new RandomAccessFile("rprueba.dat", "r");
for(int i = 0; i < 10; i++)
  System.out.println(
    "Valor " + i + ": " +
```

```
            rf.readDouble());
        rf.close();
    }
} ///:~
```

He aquí las descripciones de las secciones numeradas del programa:

Flujos de entrada

Las Secciones 1-4 demuestran la creación y uso de flujos de entrada. La Sección 4 muestra también el uso simple de un flujo de salida.

1. Archivo de entrada utilizando espacio de almacenamiento intermedio

Para abrir un archivo para entrada de caracteres se usa un **FileInputReader** junto con un objeto **File** o **String** como nombre de archivo. Para lograr mayor velocidad, se deseará que el archivo tenga un espacio de almacenamiento intermedio de forma que se dé la referencia resultante al constructor para un **BufferedReader**. Dado que **BufferedReader** también proporciona el método **readLine( )**, éste es el objeto final y la interfaz de la que se lee. Cuando se llegue al final del archivo, **readLine( )** devuelve **null**, por lo que es éste el valor que se usa para salir del bucle **while**.

El **String s2** se usa para acumular todo el contenido del archivo (incluyendo las nuevas líneas que hay que añadir porque **readLine( )** las quita). Después se usa **s2** en el resto de porciones del programa. Finalmente, se invoca a **close( )** para cerrar el archivo. Técnicamente, se llamará a **close( )** cuando se ejecute **finalize( )**, cosa que se supone que ocurrirá (se active o no el recolector de basura) cuando se acabe el programa. Sin embargo, esto se ha implementado inconsistentemente, por lo que el único enfoque seguro es el de invocar explícitamente a **close( )** en el caso de manipular archivos.

La sección 1b muestra cómo se puede envolver **System.in** para leer la entrada de la consola. **System.in** es un **DataInputStream** y **BufferedReader** necesita un parámetro **Reader**, por lo que se hace uso de **InputStreamReader** para llevar a cabo la traducción.

2. Entrada desde memoria

Esta sección toma el **String s2**, que ahora contiene todos los contenidos del archivo y lo usa para crear un **StringReader**. Después se usa **read( )** para leer cada carácter de uno en uno y enviarlo a la consola. Nótese que **read( )** devuelve el siguiente byte como un **int** por lo que hay que convertirlo en **char** para que se imprima correctamente.

3. Entrada con formato desde memoria

Para ler datos "con formato", se usa un **DataInputStream**, que es una clase de E/S orientada a **byte** (en vez de orientada a **char**). Por consiguiente se deben usar todas las clases **InputStream** en vez de clases **Reader**. Por supuesto, se puede leer cualquier cosa (como un archivo) como si de bytes se tratara, usando clases **InputStream**, pero aquí se usa un **String**. Para convertir el **String** en un array de

bytes, apropiado para un **ByteArrayInputStream, String** tiene un método **getBytes( )** que se encarga de esto. En este momento, se tiene un **InputStream** apropiado para manejar **DataInputStream**.

Si se leen los caracteres de un **DataInputStream** de uno en uno utilizando **readByte( )**, cualquier valor de byte constituye un resultado legítimo por lo que no se puede usar el valor de retorno para detectar el final de la entrada. En vez de ello, se puede usar el método **available( )** para averiguar cuántos caracteres más quedan disponibles. He aquí un ejemplo que muestra cómo leer un archivo byte a byte:

```
//: c11:PruebaEOF.java
// Probando el fin de archivo
// al leer de byte en byte.
import java.io.*;

public class PruebaEOF {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    DataInputStream entrada =
      new DataInputStream(
       new BufferedInputStream(
        new FileInputStream("PruebaEOF.java")));
    while(entrada.available() != 0)
      System.out.print((char)entrada.readByte());
  }
} ///:~
```

Nótese que **available( )** funciona de forma distinta en función del tipo de medio desde el que se esté leyendo; literalmente es "el número de bytes que se pueden leer *sin bloqueo*". Con archivos, esto equivale a todo el archivo pero con un tipo de flujo distinto podría no ser así, por lo que debe usarse con mucho cuidado.

También se podría detectar el fin de la entrada en este tipo de casos mediante una excepción. Sin embargo, el uso de excepciones para flujos de control se considera un mal uso de esta característica.

4. Salida a archivo

Este ejemplo también muestra cómo escribir datos en un archivo. En primer lugar, se crea un **FileWriter** para conectar con el archivo. Generalmente se deseará pasar la salida a través de un espacio de almacenamiento intermedio, por lo que se genera un **BufferedWriter** (es conveniente intentar retirar este envoltorio para ver su impacto en el rendimiento —el uso de espacios de almacenamiento intermedio tiende a incrementar considerablemente el rendimiento de las operaciones de E/S). Después, se convierte en un **PrintWriter** para hacer uso de las opciones de dar formato. El archivo de datos que se cree así es legible como un archivo de texto normal y corriente.

A medida que se escriban líneas al archivo, se añaden los números de línea. Nótese que no se *usa* **LineNumberInputStream**, porque es una clase estúpida e innecesaria. Como se muestra en este caso, es fundamental llevar a cabo un seguimiento de los números de página.

Cuando se agota el flujo de entrada, **readLine( )** devuelve **null**. Se verá una llamada **close( )** explícita para **salida1**, porque si no se invoca a **close( )** para todos los archivos de salida, los espacios de almacenamiento intermedio no se vaciarán, de forma que las operaciones pueden quedar inacabadas.

Flujos de salida

Los dos tipos primarios de flujos de salida se diferencian en la forma de escribir los datos: uno lo hace de forma comprensible para el ser humano, y el otro lo hace para pasárselos a **DataInput Stream**. El **RandomAccessFile** se mantiene independiente, aunque su formato de datos es compatible con **DataInputStream** y **DataOutputStream**.

5. Almacenar y recuperar datos

Un **PrintWriter** da formato a los datos de forma que sean legibles por el hombre. Sin embargo, para sacar datos de manera que puedan ser recuperados por otro flujo, se usa un **DataOutputStream para escribir los datos** y un **DataInputStream** para la recuperación. Por supuesto, estos flujos podrían ser cualquier cosa, pero aquí se usa un archivo con espacios de almacenamiento intermedio tanto para la lectura como para la escritura. **DataOutputStream** y **DataInputStream** están orientados a **byte**, por lo que requieren de **InputStreams** y **OutputStreams**.

Si se usa un **DataOutputStream** para escribir los datos, Java garantiza que se pueda recuperar el dato utilizando eficientemente un **DataInputStream** —independientemente de las plataformas sobre las que se lleven a cabo las operaciones de lectura y escritura. Esto tiene un valor increíble, pues nadie sabe quién ha invertido su tiempo preocupándose por aspectos de datos específicos de cada plataforma. El problema se desvanece simplemente teniendo Java en ambas plataformas[2].

Nótese que se escribe la cadena de caracteres haciendo uso tanto de **writeChars( )** como de **writeBytes( )**. Cuando se ejecute el programa, se observará que **writeChars( )** saca caracteres Unicode de 16 bits. Cuando se lee la línea haciendo uso de **readLine( )** se verá que hay un espacio entre cada carácter, que es debido al byte extra introducido por Unicode. Puesto que no hay ningún método "readChars" complementario en **DataInputStream**, no queda más remedio que sacar esos caracteres de uno en uno con **readChar( )**. Por tanto, en el caso de ASCII, es más sencillo escribir los caracteres como bytes seguidos de un salto de línea; posteriormente se usa **readLine( )** para leer de nuevo esos bytes en una línea ASCII tradicional.

El **writeDouble( )** almacena el número **double** en el flujo y el **readDouble( )** complementario lo recupera (hay métodos semejantes para hacer lo mismo en la escritura y lectura de otros tipos). Pero para que cualquiera de estos métodos de lectura funcione correctamente es necesario conocer la ubicación exacta del elemento de datos dentro del flujo, puesto que sería igualmente posible

[2] XML es otra solución al mismo problema: mover datos entre plataformas de computación diferentes, que en este caso no depende de que haya Java en ambas plataformas. Además, existen herramientas Java para dar soporte a XML.

leer el **double** almacenado como una simple secuencia de bytes, o como un **char**, etc. Por tanto, o bien hay que establecer un formato fijo para los datos dentro del archivo, o hay que almacenar en el propio archivo información extra que será necesario analizar para determinar dónde se encuentra ubicado el dato.

6. Leer y escribir archivos de acceso aleatorio

Como se vio anteriormente, el **RandomAccessFile** se encuentra casi totalmente aislado del resto de la jerarquía de E/S, excepto por el hecho de que implementa las interfaces **DataInput** y **DataOutput**. Por tanto, no se puede combinar con ninguno de los aspectos de las subclases **InputStream** y **OutputStream**. Incluso aunque podría tener sentido tratar un **ByteArrayInputStream** como un elemento de acceso aleatorio, se puede usar **RandomAccessFile** simplemente para abrir un archivo. Hay que asumir que un **RandomAccessFile** tiene sus espacios de almacenamiento intermedio, así que no hay que añadírselos.

La opción que queda es el segundo parámetro del constructor: se puede abrir un **RandomAccessFile** para leer ("**r**"), o para leer y escribir ("**rw**").

La utilización de un **RandomAccessFile** es como usar un **DataInputStream** y un **DataOutputStream** combinados (puesto que implementa las interfaces equivalentes). Además, se puede ver que se usa **seek( )** para moverse por el archivo y cambiar algunos de sus valores.

¿Un error?

Si se echa un vistazo a la Sección 6, se verá que el dato se escribe *antes* que el texto. Esto se debe a un problema que se introdujo con Java 1.1 (y que persiste en Java 2) que parece un error, pero informamos de él y la gente de JavaSoft que trabaja en errores nos informó de que funciona exactamente como se desea que funcione (sin embargo, el problema *no* ocurría en Java 1.0, y eso nos hace sospechar). El problema se muestra en el ejemplo siguiente:

```
//: c11:ProblemaES.java
// Problema de E/S de Java 1.1 y superiores.
import java.io.*;

public class ProblemaES {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    DataOutputStream salida =
      new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("Datos.txt")));
    salida.writeDouble(3.14159);
    salida.writeBytes("Este es el valor de pi\n");
    salida.writeBytes("El valor de pi/2 es:\n");
    salida.writeDouble(3.14159/2);
```

```
        salida.close();

        DataInputStream entrada =
          new DataInputStream(
            new BufferedInputStream(
              new FileInputStream("Datos.txt")));
        BufferedReader entradabr =
          new BufferedReader(
            new InputStreamReader(entrada));
        // Se escriben los doubles ANTES de la lectura correcta
        // de la línea de texto:
        System.out.println(entrada.readDouble());
        // Leer las líneas de texto:
        System.out.println(entradabr.readLine());
        System.out.println(entradabr.readLine());
        // Intentar leer los doubles después de la línea
        // produce una excepción de fin-de-fichero:
        System.out.println(entrada.readDouble());
      }
    } ///:~
```

Parece que todo lo que se escribe tras una llamada a **writeBytes( )** no es recuperable. La respuesta es aparentemente la misma que en el viejo chiste: "Doctor, ¡cuando hago esto, me duele!" "¡Pues no lo haga!"

Flujos entubados

PipedInputStream, **PipedOutputStream**, **PipedReader** y **PipedWriter** ya se han mencionado anteriormente en este capítulo, aunque sea brevemente. No se pretende, no obstante, sugerir que no sean útiles, pero es difícil descubrir su verdadero valor hasta entender el multihilo, puesto que este tipo de flujos se usa para la comunicación entre hilos. Su uso se verá en un ejemplo del Capítulo 14.

E/S estándar

El término *E/S estándar* proviene de Unix (si bien se ha reproducido tanto en Windows como en otros muchos sistemas operativos). Hace referencia al flujo de información que utiliza todo programa. Así, toda la entrada a un programa proviene de la *entrada estándar*, y su salida "fluye" a través de la *salida estándar*, mientras que todos sus mensajes de error se envían a la *salida de error estándar*. El valor de la E/S estándar radica en la posibilidad de encadenar estos programas de forma sencilla de manera que la salida estándar de uno se convierta en la entrada estándar del siguiente. Esta herramienta resulta extremadamente poderosa.

Leer de la entrada estándar

Siguiendo el modelo de E/S estándar, Java tiene **Sytem.in**, **System.out** y **System.err**. A lo largo de todo este libro, se ha visto cómo escribir en la salida estándar haciendo uso de **System.out**, que ya viene envuelto como un objeto **PrintStream**. **System.err** es igual que un **PrintStream**, pero **System.in** es como un **InputStream** puro, sin envoltorios. Esto significa que, si bien se pueden utilizar **System.out** y **System.err** directamente, es necesario envolver de alguna forma **System.in** antes de poder leer de él.

Generalmente se desea leer una entrada línea a línea haciendo uso de **readLine( )**, por lo que se deseará envolver **System.in** en un **BufferedReader**. Para ello hay que convertir **System.in** en un **Reader** haciendo uso de **InputStreamReader**. He aquí un ejemplo que simplemente visualiza toda línea que se teclee:

```
//: c11:Eco.java
// Como leer de la entrada estándar.
import java.io.*;

public class Eco {
  public static void main(String[] args)
  throws IOException {
    BufferedReader entrada =
         new BufferedReader(
           new InputStreamReader(System.in));
    String s;
    while((s = entrada.readLine()).length() != 0)
      System.out.println(s);
    // El programa acaba con una línea vacía.
  }
} ///:~
```

La razón que justifica la especificación de la excepción es que **readLine( )** puede lanzar una **IOException**. Nótese que **System.in** debería utilizar un espacio de almacenamiento intermedio, al igual que la mayoría de flujos.

Convirtiendo **System.out** en un **PrintWriter**

System.out es un **PrintStream**, que es, a su vez, un **OutputStream**. **PrintWriter** tiene un constructor que toma un **OutputStream** como parámetro. Por ello, si se desea es posible convertir **System.out** en un **PrintWriter** haciendo uso de ese constructor:

```
//: c11:CambiarSistemOut.java
// Convertir System.out en un PrintWriter.
import java.io.*;

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

```
    PrintWriter salida =
      new PrintWriter(System.out, true);
    salida.println("Hola, mundo");
  }
} ///:~
```

Es importante usar la versión de dos parámetros del constructor **PrintWriter** y poner el segundo parámetro a **true** para habilitar el vaciado automático, pues, si no, puede que no se vea la salida.

Redirigiendo la E/S estándar

La clase **System** de Java permite redirigir los flujos de entrada, salida y salida de error estándares simplemente haciendo uso de las llamadas a métodos estáticos:

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

Redirigir la salida es especialmente útil si, de repente, se desea comenzar la creación de mucha información de salida a pantalla, y el desplazamiento de la misma es demasiado rápido como para leer[3]. El redireccionamiento de la entrada es útil en programas de línea de comandos en los que se desee probar repetidamente una secuencia de entrada de usuario en particular. He aquí un ejemplo simple que muestra cómo usar estos métodos:

```
//: c11:Redireccionar.java
// Demuestra el redireccionamiento de la E/S estándar.
import java.io.*;

class Redireccionar {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    BufferedInputStream entrada =
      new BufferedInputStream(
        new FileInputStream(
          "Redireccionar.java"));
    PrintStream salida =
      new PrintStream(
        new BufferedOutputStream(
          new FileOutputStream("prueba.out")));
    System.setIn(entrada);
    System.setOut(salida);
    System.setErr(salida);
```

[3] El Capítulo 13 muestra una solución aún más adecuada para esto: un programa de IGU con un área de desplazamiento de texto.

```
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
      System.out.println(s);
    salida.close(); // ¡No olvidarse de esto!
  }
} ///:~
```

Este programa adjunta la entrada estándar a un archivo y redirecciona la salida estándar y la de error a otro archivo.

El redireccionamiento de la E/S manipula flujos de bytes, en vez de flujos de caracteres, por lo que se usan **InputStreams** y **OutputStreams** en vez de **Readers** y **Writers**.

Compresión

La biblioteca de E/S de Java contiene clases que dan soporte a la lectura y escritura de flujos en un formato comprimido. Estas clases están envueltas en torno a las clases de E/S existentes para proporcionar funcionalidad de compresión.

Estas clases no se derivan de las clases **Reader** y **Writer**, sino que son parte de las jerarquías **InputStream** y **OutputStream**. Esto se debe a que la biblioteca de compresión funciona con bytes en vez de caracteres. Sin embargo, uno podría verse forzado en ocasiones a mezclar ambos tipos de flujos. (Recuérdese que se puede usar **InputStreamReader** y **OutputStreamWriter** para proporcionar conversión sencilla entre un tipo y el otro.)

Clase de Compresión	Función
CheckedInputStream	**GetCheckSum( )** produce una suma de comprobación para cualquier **InputStream** (no simplemente de descompresión).
CheckedOutputStream	**GetCheckSum( )** produce una suma de comprobación para cualquier **OutputStream** (no simplemente de compresión).
DeflaterOutputStream	Clase base de las clases de compresión.
ZipOutputStream	Una **DeflaterOutputStream** que comprime datos en el formato de archivos ZIP.
GZIPOutputStream	Una **DeflaterOutputStream** que comprime datos en el formato de archivos GZIP.
InflaterInputStream	Clase base de las clases de descompresión.
ZipInflaterStream	Una **InflaterInputStream** que descomprime datos almacenados en el formato de archivos ZIP.

Clase Compresión	Función
GZIPInputStream	Una **InflaterInputStream** que descomprime datos almacenados en el formato de archivos GZIP.

Aunque hay muchos algoritmos de compresión, los más comúnmente utilizados son probablemente ZIP y GZIP. Por consiguiente, se pueden manipular de forma sencilla los datos comprimidos con las muchas herramientas disponibles para leer y escribir en estos formatos.

Compresión sencilla con GZIP

La interfaz GZIP es simple y por consiguiente la más apropiada a la hora de comprimir un único flujo de datos (en vez de tener un contenedor de datos heterogéneos). He aquí un ejemplo que comprime un archivo:

```
//: c11:ComprimirGZIP.java
// Utiliza compresion GZIP para comprimir un archivo
// cuyo nombre se pasa en línea de comandos.
import java.io.*;
import java.util.zip.*;

public class ComprimirGZIP {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    BufferedReader entrada =
      new BufferedReader(
        new FileReader(args[0]));
    BufferedOutputStream salida =
      new BufferedOutputStream(
        new GZIPOutputStream(
          new FileOutputStream("prueba.gz")));
    System.out.println("Escribiendo el archivo");
    int c;
    while((c = entrada.read()) != -1)
      salida.write(c);
    entrada.close();
    salida.close();
    System.out.println("Leyendo el archivo");
    BufferedReader entrada2 =
      new BufferedReader(
        new InputStreamReader(
          new GZIPInputStream(
            new FileInputStream("prueba.gz"))));
```

```
    String s;
    while((s = entrada2.readLine()) != null)
      System.out.println(s);
  }
} ///:~
```

El uso de clases de compresión es directo —simplemente se envuelve el flujo de salida en un **GZIPOutputStream** o en un **ZipOutputStream,** y el flujo de entrada en un **GZIPInputStream** o en un **ZipInputStream**. Todo lo demás es lectura y escritura de E/S ordinarias. Éste es un ejemplo que mezcla flujos orientados a **byte** con flujos orientados a **char**: **entrada** usa las clases **Reader**, mientras que el constructor de **GZIPOutputStream** sólo puede aceptar un objeto **OutputStream**, y no un objeto **Writer**. Cuando se abre un archivo, se convierte el **GZIPInputStream** en un **Reader**.

Almacenamiento múltiple con ZIP

La biblioteca que soporta el formato ZIP es mucho más completa. Con ella, es posible almacenar de manera sencilla múltiples archivos, e incluso existe una clase separada para hacer más sencillo el proceso de leer un archivo ZIP. La biblioteca usa el formato ZIP estándar de forma que trabaja prácticamente con todas las herramientas actualmente descargables desde Internet. El ejemplo siguiente tiene la misma forma que el anterior, pero maneja tantos parámetros de línea de comandos como se desee. Además, muestra el uso de las clases **Checksum** para calcular y verificar la suma de comprobación del archivo. Hay dos tipos de **Checksum**: **Adler32** (que es más rápido) y **CRC32** (que es más lento pero ligeramente más exacto).

```
//: c11:ComprimirZip.java
// Utiliza compresion ZIP para comprimir cualquier
// número de archivos indicados en línea de comandos.
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ComprimirZip {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f =
      new FileOutputStream("Prueba.zip");
    CheckedOutputStream csum =
      new CheckedOutputStream(
        f, new Adler32());
    ZipOutputStream salida =
      new ZipOutputStream(
        new BufferedOutputStream(csum));
    salida.setComment("una prueba de compresión Zip con Java");
```

```
// Sin el getComment() correspondiente.
for(int i = 0; i < args.length; i++) {
  System.out.println(
    "Escribiendo el archivo " + args[i]);
  BufferedReader entrada =
    new BufferedReader(
      new FileReader(args[i]));
  salida.putNextEntry(new ZipEntry(args[i]));
  int c;
  while((c = entrada.read()) != -1)
    salida.write(c);
  entrada.close();
}
salida.close();
// ¡suma de chequeo válida únicamente una vez
// cerrado el archivo!
System.out.println("Suma de comprobacion: " +
  csum.getChecksum().getValue());
// Ahora extraer los archivos:
System.out.println("Leyendo el archivo");
FileInputStream fi =
    new FileInputStream("prueba.zip");
CheckedInputStream csumi =
  new CheckedInputStream(
    fi, new Adler32());
ZipInputStream entrada2 =
  new ZipInputStream(
    new BufferedInputStream(csumi));
ZipEntry ze;
while((ze = entrada2.getNextEntry()) != null) {
  System.out.println("Leyendo el archivo " + ze);
  int x;
  while((x = entrada2.read()) != -1)
    System.out.write(x);
}
System.out.println("Suma de comprobación: " +
  csumi.getChecksum().getValue());
entrada2.close();
// Forma alternativa de abrir y leer
// archivo zip:
ZipFile zf = new ZipFile("prueba.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
  ZipEntry ze2 = (ZipEntry)e.nextElement();
  System.out.println("Archivo: " + ze2);
```

```
        // ... extrayendo los datos como antes
      }
    }
} ///:~
```

Para cada archivo que se desee añadir al archivo, es necesario llamar a **putNextEntry( )** y pasarle un objeto **ZipEntry**. Este objeto contiene una interfaz que permite extraer y modificar todos los datos disponibles en esa entrada particular del archivo ZIP: nombre, tamaño comprimido y descomprimido, fecha, suma de comprobación CRC, datos de campos extra, comentarios, métodos de compresión y si es o no una entrada de directorio. Sin embargo, incluso aunque el formato ZIP permite poner contraseñas a los archivos, no hay soporte para esta faceta en la biblioteca ZIP de Java. Y aunque tanto **CheckedInputStream** como **CheckedOutputStream** soportan ambos sumas de comprobación, **Adler32** y **CRC32**, la clase **ZipEntry** sólo soporta una interfaz para CRC. Esto es una restricción del formato ZIP subyacente, pero podría limitar el uso de la más rápida **Adler32**.

Para extraer archivos, **ZipInputStream** tiene un método **getNextEntry( )** que devuelve la siguiente **ZipEntry** si es que la hay. Una alternativa más sucinta es la posibilidad de leer el archivo utilizando un objeto **ZipFile**, que tiene un método **entries( )** para devolver una **Enumeration** al **ZipEntries**.

Para leer la suma de comprobación hay que tener algún tipo de acceso al objeto **Checksum** asociado. Aquí, se retiene una referencia a los objetos **CheckedOutputStream** y **CheckedInputStream**, pero también se podría simplemente guardar una referencia al objeto **Checksum**.

Existe un método misterioso en los flujos Zip que es **setComment( )**. Como se mostró anteriormente, se puede poner un comentario al escribir un archivo, pero no hay forma de recuperar el comentario en el **ZipInputStream**. Parece que los comentarios están completamente soportados en una base de entrada por entrada, eso sí, sólamente vía **ZipEntry**.

Por supuesto, no hay un número de archivos al usar las bibliotecas **GZIP** o **ZIP** —se puede comprimir cualquier cosa, incluidos los datos a enviar a través de una conexión de red.

Archivos Java (JAR)

El formato ZIP también se usa en el formato de archivos JAR (Java ARchive), que es una forma de coleccionar un grupo de archivos en un único archivo comprimido, exactamente igual que el zip. Sin embargo, como todo lo demás en Java, los ficheros JAR son multiplataforma, por lo que no hay que preocuparse por aspectos de plataforma. También se pueden incluir archivos de audio e imagen, o archivo de clases.

Los archivos JAR son particularmente útiles cuando se trabaja con Internet. Antes de los archivos JAR, el navegador Web habría tenido que hacer peticiones múltiples a un servidor web para descargar todos los archivos que conforman un *applet*. Además, cada uno de estos archivos estaba sin comprimir. Combinando todos los archivos de un *applet* particular en un único archivo JAR, sólo es necesaria una petición al servidor, y la transferencia es más rápida debido a la compresión. Y cada entrada de un archivo JAR soporta firmas digitales por seguridad (consultar a la documentación de Java si se necesitan más detalles).

Un archivo JAR consta de un único archivo que contiene una colección de archivos ZIP junto con una "declaración" que los describe. (Es posible crear archivos de declaración; de otra manera el programa **jar** lo hará automáticamente.) Se puede averiguar algo más sobre declaraciones JAR en la documentación del JDK HTML.

La utilidad **jar** que viene con el JDK de Sun comprime automáticamente los archivos que se seleccionen. Se invoca en línea de comandos:

```
jar [opciones] destino [manifiesto] archivo(s)Entrada
```

Las opciones son simplemente una colección de letras (no es necesario ningún guión u otro indicador). Los usuarios de Unix/Linux notarán la semejanza con las opciones **tar**. Éstas son:

c	Crea un archivo nuevo o vacío.
t	Lista la tabla de contenidos.
x	Extrae todos los archivos.
x file	Extrae el archivo nombrado.
f	Dice: "Voy a darte el nombre del archivo." Si no lo usas, JAR asume que su entrada provendrá de la entrada estándar, o, si está creando un archivo, su salida irá a la salida estándar.
m	Dice que el primer parámetro será el nombre de un archivo de declaración creado por el usuario.
v	Genera una salida que describe qué va haciendo JAR.
O	Simplemente almacena los archivos; no los comprime (usarlo para crear un archivo JAR que se puede poner en el *classpath*).
M	No crear automáticamente un archivo de declaración.

Si se incluye algún subdirectorio en los archivos a añadir a un archivo JAR, se añade ese subdirectorio automáticamente, incluyendo también todos sus subdirectorios, etc. También se mantiene la información de rutas.

He aquí algunas formas habituales de invocar a **jar**:

```
jar cf miArchivoJar.jar *.class
```

Esto crea un fichero JAR llamado **miFicheroJar.jar** que contiene todos los archivos de clase del directorio actual, junto con un archivo declaración creado automáticamente.

```
jar cmf miArchivoJar.jar miArchivoDeclaracion.mf *.class
```

Como en el ejemplo anterior, pero añadiendo un archivo de declaración de nombre **miArchivo Declaracion.mf** creado por el usuario.

```
jar tf miArchivoJar.Jar
```

Añade el indicador que proporciona información más detallada sobre los archivos de **miArchivoJar.jar**.

```
jar cvf miAplicacion.jar audio clases imagen
```

Si se asume que **audio, clases** e **imagen** son subdirectorios, combina todos los subdirectorios en el archivo **miAplicacion.jar**. También se incluye el indicador que proporciona realimentación extra mientras trabaja el programa **jar**.

Si se crea un fichero JAR usando la opción **O**, el archivo se puede ubicar en el CLASSPATH:

```
CLASSPATH = "lib1.jar; lib2.jar"
```

Entonces, Java puede buscar archivos de clase en **lib1.jar** y **lib2.jar**.

La herramienta **jar** no es tan útil como una utilidad **zip**. Por ejemplo, no se pueden añadir o actualizar archivos de un archivo JAR existente; sólo se pueden crear archivos JAR de la nada. Además, no se pueden mover archivos a un archivo JAR, o borrarlos al moverlos. Sin embargo, un fichero JAR creado en una plataforma será legible transparentemente por la herramienta **jar** en cualquier otra plataforma (un problema que a veces se da en las utilidades **zip**).

Como se verá en el Capítulo 13, los archivos JAR también se utilizan para empaquetar JavaBeans.

Serialización de objetos

La *serialización de objetos* de Java permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bits que puede ser posteriormente restaurada para regenerar el objeto original. Esto es cierto incluso a través de una red, lo que significa que el mecanismo de serialización compensa automáticamente las diferencias entre sistemas operativos. Es decir, se puede crear un objeto en una máquina Windows, serializarlo, y enviarlo a través de la red a una máquina Unix, donde será reconstruido correctamente. No hay que preocuparse de las representaciones de los datos en las distintas máquinas, al igual que no importan la ordenación de los bytes y el resto de detalles.

Por sí misma, la serialización de objetos es interesante porque permite implementar *persitencia ligera*. Hay que recordar que la persistencia significa que el tiempo de vida de un objeto no viene determinado por el tiempo que dure la ejecución del programa —el objeto vive *mientras se den* invocaciones al mismo en el programa. Al tomar un objeto serializable y escribirlo en el disco, y luego restaurarlo cuando sea reinvocado en el programa, se puede lograr el efecto de la persistencia. La razón por la que se califica de "ligera" es porque simplemente no se puede definir un objeto utilizando algún tipo de palabra clave "persistent" y dejar que el sistema se encargue de los detalles

(aunque puede que esta posibilidad exista en el futuro). Por el contrario, hay que serializar y deserializar explícitamente los objetos.

La serialización de objetos se añadió a Java para soportar dos aspectos de mayor calibre. La invocación de Procedimientos Remotos (*Remote Method Invocation-RMI*) permite a objetos de otras máquinas comportarse como si se encontraran en la tuya propia. Al enviar mensajes a objetos remotos, es necesario serializar los parámetros y los valores de retorno. RMI se discute en el Capítulo 15.

La serialización de objetos también es necesaria en el caso de los JavaBeans, descritos en el Capítulo 13. Cuando se usa un Bean, su información de estado se suele configurar en tiempo de diseño. La información de estado debe almacenarse y recuperarse más tarde al comenzar el programa; la serialización de objetos realiza esta tarea.

La serialización de un objeto es bastante simple, siempre que el objeto implemente la interfaz **Serializable** (la interfaz es simplemente un flag y no tiene métodos). Cuando se añadió la serialización al lenguaje, se cambiaron muchas clases de la biblioteca estándar para que fueran serializables, incluidos todos los envoltorios y tipos primitivos, todas las clases contenedoras, y otras muchas. Incluso los objetos **Class** pueden ser serializados. (Véase el Capítulo 12 para comprender las implicaciones de esto.)

Para serializar un objeto, se crea algún tipo de objeto **OutputStream** y se envuelve en un **ObjectOutputStream**. En este momento sólo hay que invocar a **writeObject( )** y el objeto se serializa y se envía al **OutputStream**. Para invertir este proceso, se envuelve un **InputStream** en un **ObjectInputStream** y se invoca a **readObject( )**. Lo que vuelve, como siempre, es una referencia a un **Object**, así que hay que hacer una conversión hacia abajo para dejar todo como se debe.

Un aspecto particularmente inteligente de la serialización de objetos es que, no sólo salva la imagen del objeto, sino que también sigue todas las referencias contenidas en el objeto, y salva *esos* objetos, siguiendo además las referencias contenidas en cada uno de ellos, etc. A esto se le suele denominar la "telaraña de objetos" puesto que un único objeto puede estar conectado, e incluir arrays de referencias a objetos, además de objetos miembro. Si se tuviera que mantener un esquema de serialización de objetos propio, el mantenimiento del código para seguir todos estos enlaces sería casi imposible. Sin embargo, la serialización de objetos Java parece encargarse de todo haciendo uso de un algoritmo optimizado que recorre la telaraña de objetos. El ejemplo siguiente prueba el mecanismo de serialización haciendo un "gusano" de objetos enlazados, cada uno de los cuales tiene un enlace al siguiente segmento del gusano, además de un array de referencias a objetos de una clase distinta, **Datos**:

```
//: c11:Gusano.java
// Demuestra la serialización de objetos.
import java.io.*;

class Datos implements Serializable {
  private int i;
  Datos(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
```

```
  }
}

public class Gusano implements Serializable {
  // Generar un valor entero al azar:
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Datos[] d = {
    new Datos(r()), new Datos(r()), new Datos(r())
  };
  private Gusano siguiente;
  private char c;
  // Valor de i == número de segmentos
  Gusano(int i, char x) {
    System.out.println(" Constructor Gusano: " + i);
    c = x;
    if(--i > 0)
      siguiente = new Gusano(i, (char)(x + 1));
  }
  Gusano() {
    System.out.println("Constructor por defecto");
  }
  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
      s += d[i].toString();
    s += ")";
    if(siguiente != null)
      s += siguiente.toString();
    return s;
  }
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Gusano c = new Gusano(6, 'a');
    System.out.println("g = " + g);
    ObjectOutputStream salida =
      new ObjectOutputStream(
        new FileOutputStream("gusano.out"));
    salida.writeObject("Almacenamiento Gusano");
    salida.writeObject(w);
    salida.close(); // También vacía la salida
    ObjectInputStream entrada =
      new ObjectInputStream(
```

```
            new FileInputStream("gusano.out"));
        String s = (String)entrada.readObject();
        Gusano g2 = (Gusano)entrada.readObject();
        System.out.println(s + ", g2 = " + g2);
        ByteArrayOutputStream bsalida =
          new ByteArrayOutputStream();
        ObjectOutputStream salida2 =
          new ObjectOutputStream(bsalida);
        salida2.writeObject("Almacenamiento Gusano");
        salida2.writeObject(g);
        salida2.flush();
        ObjectInputStream entrada2 =
          new ObjectInputStream(
            new ByteArrayInputStream(
              bsalida.toByteArray()));
        s = (String)entrada2.readObject();
        Gusano g3 = (Gusano)entrada2.readObject();
        System.out.println(s + ", g3 = " + g3);
    }
} ///:~
```

Para hacer las cosas interesantes, el array de objetos **Datos** dentro de **Gusano** se inicializa con números al azar. (De esta forma no hay que sospechar que el compilador mantenga algún tipo de meta-información.) Cada segmento **Gusano** se etiqueta con un **char** que es automáticamente generado en el proceso de generar recursivamente la lista enlazada de **Gusanos**. Cuando se crea un **Gusano**, se indica al constructor lo largo que se desea que sea. Para hacer la referencia **siguiente** llama al constructor **Gusano** con una longitud uno menor, etc. La referencia **siguiente** final se deja a **null** indicando el final del **Gusano**.

El objetivo de todo esto era hacer algo racionalmente complejo que no pudiera ser serializado fácilmente. El acto de serializar, sin embargo, es bastante simple. Una vez que se ha creado el **ObjectOutputStream** a partir de otro flujo, **writeObject( )** serializa el objeto. Nótese que la llamada a **writeObject( )** es también para un **String**. También se pueden escribir los tipos de datos primitivos utilizando los mismos métodos que **DataOutputStream** (comparten las mismas interfaces).

Hay dos secciones de código separadas que tienen la misma apariencia. La primera lee y escribe un archivo, y la segunda escribe y lee un **ByteArray**. Se puede leer y escribir un objeto usando la serialización en cualquier **DataInputStream** o **DataOutputStream**, incluyendo, como se verá en el Capítulo 15, una red. La salida de una ejecución fue:

```
Constructor Gusano: 6
Constructor Gusano: 5
Constructor Gusano: 4
Constructor Gusano: 3
Constructor Gusano: 2
Constructor Gusano: 1
```

```
g = :a(262):b(100):c(396):d(480):e(316):f(398)
Almacenamiento Gusano, g2 =
:a(262):b(100):c(396):d(480):e(316):f(398)
Almacenamiento Gusano, g3
:a(262):b(100):c(396):d(480):e(316):f(398)
```

Se puede ver que el objeto deserializado contiene verdaderamente todos los enlaces que había en el objeto original.

Nótese que no se llama a ningún constructor, ni siquiera el constructor por defecto, en el proceso de deserialización de un objeto **Serializable**. Se restaura todo el objeto recuperando datos del **InputStream**.

La serialización de objetos está orientada al **byte**, y por consiguiente usa las jerarquías **InputStream** y **OutputStream**.

Encontrar la clase

Uno podría preguntarse qué debe tener un objeto para que pueda ser recuperado de su estado serealizado. Por ejemplo, supóngase que se serializa un objeto y se envía como un archivo o a través de una red a otro máquina. ¿Podría un programa de la otra máquina reconstruir el objeto simplemente con el contenido del archivo?

La mejor forma de contestar a esta pregunta es (como siempre) haciendo un experimento. El archivo siguiente se encuentra en el subdirectorio de este capítulo:

```
//: c11:Extraterrestre.java
// Una clase serializable.
import java.io.*;

public class Extraterrestre implements Serializable {
} ///:~
```

El archivo que crea y serializa un objeto **Extraterrestre** va en el mismo directorio:

```
//: c11:CongelarExtraterrestre.java
// Crear un archivo de salida serializado.
import java.io.*;

public class CongelarExtraterrestre {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException {
    ObjectOutput salida =
      new ObjectOutputStream(
        new FileOutputStream("Expediente.X"));
```

```
    Extraterrestre zorcon = new Extraterrestre();
    salida.writeObject(zorcon);
  }
} ///:~
```

Más que capturar y manejar excepciones, este programa toma el enfoque *rápido y sucio* de pasar las excepciones fuera del método **main( )**, de forma que serán reportadas en línea de comandos.

Una vez que se compila y ejecuta el programa, copie el fichero **Expediente.X** resultante al directorio denominado **expedientesx**, en el que se encuentra el siguiente código:

```
//: c11:expedientesx:DescongelarExtraterrestre.java
// Intentar recuperar un objeto serializado sin tener la
// clase de objeto almacenada en él.
import java.io.*;

public class DescongelarExtraterrestre {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    ObjectInputStream entrada =
      new ObjectInputStream(
        new FileInputStream("Expediente.X"));
    Object misterio = entrada.readObject();
    System.out.println(misterio.getClass());
  }
} ///:~
```

Este programa abre el archivo y lee el objeto **misterio** con éxito. Sin embargo, en cuanto se intenta averiguar algo del objeto —lo cual requiere el objeto **Class** de **Extraterrestre**— la Máquina Virtual Java (JVM) no puede encontrar **Extraterrestre.class** (a menos que esté en el Classpath, lo cual no ocurre en este ejemplo). Se obtendrá una **ClassNotFoundException**. (¡De nuevo, se desvanece toda esperanza de vida extraterrestre antes de poder encontrar una prueba de su existencia!)

Si se espera hacer mucho una vez recuperado un objeto serializado, hay que asegurarse de que la JVM pueda encontrar el archivo **.class** asociado en el path de clases locales o en cualquier otro lugar en Internet.

Controlar la serialización

Como puede verse, el mecanismo de serialización por defecto tiene un uso trivial. Pero ¿qué pasa si se tienen necesidades especiales? Quizás se tienen aspectos especiales relativos a seguridad y no se desea serializar algunas porciones de un objeto, o quizás simplemente no tiene sentido que se serialice algún subobjeto si esa parte necesita ser creada de nuevo al recuperar el objeto.

Se puede controlar el proceso de serialización implementando la interfaz **Externalizable** en vez de la interfaz **Serializable**. La interfaz **Externalizable** extiende la interfaz **Serializable** añadiendo dos métodos, **writeExternal( )** y **readExternal( )**, que son invocados automáticamente para el objeto

durante la serialización y la deserialización, de forma que se puedan llevar a cabo las operaciones especiales.

El ejemplo siguiente muestra la implementación simple de los métodos de la interfaz **Externalizable**. Nótese que **Rastro1** y **Rastro2** son casi idénticos excepto por una diferencia mínima (a ver si la descubres echando un vistazo al código):

```
//: c11:Rastros.java
// Uso simple de Externalizable & un truco.
import java.io.*;
import java.util.*;

class Rastro1 implements Externalizable {
  public Rastro1() {
    System.out.println("Constructor Rastro1");
  }
  public void writeExternal(ObjectOutput salida)
      throws IOException {
    System.out.println("Rastro1.writeExternal");
  }
  public void readExternal(ObjectInput entrada)
      throws IOException, ClassNotFoundException {
    System.out.println("Rastro1.readExternal");
  }
}

class Rastro2 implements Externalizable {
  Rastro2() {
    System.out.println("Constructor Rastro2");
  }
  public void writeExternal(ObjectOutput salida)
      throws IOException {
    System.out.println("Rastro2.writeExternal");
  }
  public void readExternal(ObjectInput entrada)
      throws IOException, ClassNotFoundException {
    System.out.println("Rastro2.readExternal");
  }
}

public class Rastros {
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Construyendo objetos:");
```

```
    Rastro1 r1 = new Rastro1();
    Rastro2 r2 = new Rastro2();
    ObjectOutputStream S =
      new ObjectOutputStream(
        new FileOutputStream("Rastros.salida"));
    System.out.println("Salvando objetos:");
    s.writeObject(r1);
    s.writeObject(r2);
    s.close();
    // Ahora recuperarlos:
    ObjectInputStream entrada =
      new ObjectInputStream(
        new FileInputStream("Rastros.salida"));
    System.out.println("Recuperando r1:");
    b1 = (Rastro1)entrada.readObject();
    // ¡OOPS! Lanza una excepción:
//! System.out.println("Recuperando r2:");
//! r2 = (Rastro2)entrada.readObject();
  }
} ///:~
```

La salida del programa es:

```
Construyendo objetos:
Constructor Rastro1
Constructor Rastro2
Salvando objetos:
Rastro1.writeExternal
Rastro2.writeExternal
Recuperando r1:
Constructor Rastro1
Rastro1.readExternal
```

La razón por la que el objeto **Rastro2** no se recupera es que intentar hacerlo causa una excepción. ¿Se ve la diferencia entre **Rastro1** y **Rastro2**? El constructor de **Rastro1** es **public**, mientras que el constructor de **Rastro2** no lo es, y eso causa la excepción en la recuperación. Puede intentarse hacer **public** el constructor de **Rastro2** y retirar los comentarios **//!** para ver los resultados correctos.

Cuando se recupera **r1**, se invoca al constructor por defecto de **Rastro1**. Esto es distinto a recuperar el objeto **Serializable**, en cuyo caso se construye el objeto completamente a partir de sus bits almacenados, sin llamadas al constructor. Con un objeto **Externalizable**, se da todo el comportamiento de construcción por defecto normal (incluyendo las inicializaciones del momento de la definición de campos), y *posteriormente*, se invoca a **readExternal( )**. Es necesario ser consciente de esto —en particular, del hecho de que siempre tiene lugar toda la construcción por defecto— para lograr el comportamiento correcto en los objetos **Externalizables**.

He aquí un ejemplo que muestra qué hay que hacer para almacenar y recuperar un objeto **Externalizable** completamente:

```
//: c11:Rastro3.java
// Reconstruyendo un objeto externalizable.
import java.io.*;
import java.util.*;

class Rastro3 implements Externalizable {
  int i;
  String s; // Sin inicialización
  public Rastro3() {
    System.out.println("Constructor Rastro3");
    // s, i sin inicializar
  }
  public Rastro3(String x, int a) {
    System.out.println("Rastro3(String x, int a)");
    s = x;
    i = a;
    // s & i inicializadas sólo en un constructor
    // distinto del constructor por defecto.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput salida)
  throws IOException {
    System.out.println("Rastro3.writeExternal");
    // Hay que hacer esto:
    salida.writeObject(s);
    salida.writeInt(i);
  }
  public void readExternal(ObjectInput entrada)
  throws IOException, ClassNotFoundException {
    System.out.println("Rastro3.readExternal");
    // Hay que hacer esto:
    s = (String)entrada.readObject();
    i = entrada.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Construyendo objetos:");
    Rastro3 r3 = new Rastro3("Una cadena ", 47);
    System.out.println(b3);
    ObjectOutputStream s =
      new ObjectOutputStream(
        new FileOutputStream("Rastro3.salida"));
```

```
    System.out.println("Salvando objeto:");
    s.writeObject(r3);
    s.close();
    // Ahora recuperarlo:
    ObjectInputStream entrada =
      new ObjectInputStream(
        new FileInputStream("Rastro3.salida"));
    System.out.println("Recuperando r3:");
    r3 = (Rastro3)entrada.readObject();
    System.out.println(r3);
  }
} ///:~
```

Los campos **s** e **i** se inicializan solamente en el segundo constructor, pero no en el constructor por defecto. Esto significa que si no se inicializan **s** e **i** en **readExternal( )**, serán **null** (puesto que el espacio de almacenamiento del objeto se inicializa a ceros en el primer paso de la creación del mismo). Si se comentan las dos líneas de código que siguen a las frases "Hay que hacer esto" y se ejecuta el programa, se verá que se recupera el objeto, **s** es **null**, e **i** es cero.

Si se está heredando de un objeto **Externalizable**, generalmente se invocará a las versiones de clase base de **writeExternal( )** y **readExternal( )** para proporcionar un almacenamiento y recuperación adecuados de los componentes de la clase base.

Por tanto, para que las cosas funcionen correctamente, no sólo hay que escribir los datos importantes del objeto durante el método **writeExternal( )** (no hay comportamiento por defecto que escriba ninguno de los objetos miembro de un objeto **Externalizable**), sino que también hay que recuperar los datos en el método **readExternal( )**. Esto puede ser un poco confuso al principio puesto que el comportamiento por defecto de la construcción de un objeto **Externalizable** puede hacer que parezca que tiene lugar automáticamente algún tipo de almacenamiento y recuperación. Esto no es así.

La palabra clave transient

Cuando se está controlando la serialización, puede ocurrir que haya un subobjeto en particular para el que no se desee que se produzca un salvado y recuperación automáticos por parte del mecanismo de serialización de Java. Éste suele ser el caso si ese objeto representa información sensible que no se desea serializar, como una contraseña. Incluso si esa información es **private** en el objeto, una vez serializada es posible que cualquiera acceda a la misma leyendo el objeto o interceptando una transmisión de red.

Una forma de evitar que partes sensibles de un objeto sean serializables es implementar la clase como **Externalizable**, como se ha visto previamente. Así, no se serializa automáticamente nada y se pueden serializar explícitamente sólo las partes de **writeExternal( )** necesarias.

Sin embargo, al trabajar con un objeto **Serializable**, toda la serialización se da automáticamente. Para controlar esto, se puede desactivar la serialización en una base campo-a-campo utilizando la palabra clave **transient**, que dice: "No te molestes en salvar o recuperar esto —me encargaré yo".

Por ejemplo, considérese un objeto **InicioSesion**, que mantiene información sobre un inicio de sesión en particular. Supóngase que, una vez verificado el inicio, se desean almacenar los datos, pero sin la contraseña. La forma más fácil de hacerlo es implementar **Serializable** y marcar el campo **contraseña** como **transient**. Debería quedar algo así:

```
//: c11:InicioSesion.java
// Demuestra la palabra clave "transient".
import java.io.*;
import java.util.*;

class InicioSesion implements Serializable {
  private Date fecha = new Date();
  private String usuario;
  private transient String contrasenia;
  InicioSesion(String nombre, String cont) {
    usuario = nombre;
    contrasenia = cont;
  }
  public String toString() {
    String cont =
      (contrasenia == null) ? "(n/a)" : contrasenia;
    return "Info inicio sesión: \n    " +
      "usuario: " + usuario +
      "\n    fecha: " + fecha +
      "\n    contrasenia: " + cont;
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    InicioSesion a = new InicioSesion("Hulk", "myLittlePony");
    System.out.println( "InicioSesion a = " + a);
      ObjectOutputStream s =
        new ObjectOutputStream(
          new FileOutputStream("InicioSesion.out"));
    s.writeObject(a);
    s.close();
    // Retraso:
    int segundos = 5;
    long t = System.currentTimeMillis()
            + seconds * 1000;
    while(System.currentTimeMillis() < t)
      ;
    // Ahora, recuperarlos:
    ObjectInputStream entrada =
      new ObjectInputStream(
        new FileInputStream("InicioSesion.out"));
```

```
      System.out.println(
        "Recuperando el objeto a las " + new Date());
      a = (InicioSesion)entrada.readObject();
      System.out.entrada( "InicioSesion a = " + a);
    }
} ///:~
```

Se puede ver que los campos **fecha** y **usuario** son normales (no **transient**) y, por consiguiente, se serializan automáticamente. Sin embargo, el campo **contraseña** es **transient**, así que no se almacena en el disco; además el mecanismo de serialización ni siquiera intenta recuperarlo. La salida es:

```
Info InicioSesion a = Info InicioSesion:
    usuario: Hulk
    fecha: Sun Mar 23 18:25:53 PST 1997
    contrasenia: myLittlePony
Recuperando objeto a las Sun Mar 23 18:25:59 PST 1997
InicioSesion a = Info InicioSesion:
    usuario: Hulk
    fecha: Sun Mar 23 18:25:53 PST 1997
    contrasenia: (n/a)
```

Cuando se recupera el objeto, el campo **contrasenia** es **null**. Nótese que **toString( )** debe comprobar si hay un valor **null** en **contrasenia** porque si se intenta ensamblar un objeto **String** haciendo uso del operador '+' sobrecargado, y ese operador encuentra una referencia a **null**, se genera una **NullPointerException**. (En versiones futuras de Java puede que se añada código que evite este problema.)

También se puede ver que el campo **fecha** se almacena y recupera a partir del disco en vez de regenerarse de nuevo.

Puesto que los objetos **Externalizable** no almacenan ninguno de sus campos por defecto, la palabra clave **transient** es para ser usada sólo con objetos **Serializable**.

Una alternativa a Externalizable

Si uno no es entusiasta de la implementación de la interfaz **Externalizable**, hay otro enfoque. Se puede implementar la interfaz **Serializable** y *añadir* (nótese que se dice "añadir", y no "superponer" o "implementar") métodos llamados **writeObject( )** y **readObject( ),** que serán automáticamente invocados cuando se serialice y deserialice el objeto, respectivamente. Es decir, si se proporcionan estos dos métodos, se usarán en vez de la serialización por defecto.

Estos métodos deben tener exactamente las signaturas siguientes:

```
private void
    writeObject(ObjectOutputStream flujo)
      trhrows IOException;

private void
```

```
    readObject(ObjectInputStream flujo)
      throws IOException, ClassNotFoundException
```

Desde el punto de vista del diseño, aquí todo parece un misterio. En primer lugar, se podría pensar que, debido a que estos métodos no son parte de una clase base o de la interfaz **Serializable**, deberían definirse en sus propias interface(s). Pero nótese que se definen como **private**, lo que significa que sólo van a ser invocados por miembros de esa clase. Sin embargo, de hecho no se les invoca desde otros miembros de esta clase, sino que son los métodos **writeObject( )** y **readObject ( )** de los objetos **ObjectOutputStream** y **ObjectInputStream** los que invocan a los métodos **writeObject( )** y **readObject( )** de nuestro objeto. (Nótese nuestro tremendo temor a no comenzar una larga diatriba sobre el nombre de los métodos a usar aquí. En pocas palabras: todo es confuso.) Uno podría preguntarse cómo logran los objetos **ObjectOutputStream** y **ObjectInputStream** acceso a los métodos **private** de la clase. Sólo podemos asumir que es parte de la magia de la serialización.

En cualquier caso, cualquier cosa que se defina en una **interface** es automáticamente **public,** por lo que si **writeObject( )** y **readObject( )** deben ser **private**, no pueden ser parte de una **interface**. Puesto que hay que seguir las signaturas con exactitud, el efecto es el mismo que si se está implementando una **interfaz**.

Podría parecer que cuando se invoca a **ObjectOutputStream.writeObject( )**, se interroga al objeto **Serializable** que se le pasa (utilizando sin duda la reflectividad) para ver si implementa su propio **writeObject( )**. Si es así, se salta el proceso de serialización normal, y se invoca al **writeObject( )**. En el caso de **readObject( )** ocurre exactamente igual.

Hay otra particularidad. Dentro de tu **writeObject( )** se puede elegir llevar a cabo la acción **writeObject( )** por defecto invocando a **defaultWriteObject( )**. De forma análoga, dentro de **readObject( )** se puede invocar a **defaultReadObject( )**. He aquí un ejemplo simple que demuestra cómo se puede controlar el almacenamiento y recuperación de un objeto **Serializable**:

```
//: c11:CtlSerial.java
// Controlando la serialización añadiendo métodos
// writeObject() y readObject() propios.
import java.io.*;

public class CtlSerial implements Serializable {
  String a;
  transient String b;
  public CtlSerial(String aa, String bb) {
    a = "No Transient: " + aa;
    b = "Transient: " + bb;
  }
  public String toString() {
    return a + "\n" + b;
  }
  private void
    writeObject(ObjectOutputStream flujo)
      throws IOException {
```

```
    flujo.defaultWriteObject();
    flujo.writeObject(b);
  }
  private void
    readObject(ObjectInputStream flujo)
      throws IOException, ClassNotFoundException {
    flujo.defaultReadObject();
    b = (String)flujo.readObject();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    SerialCtl cs =
      new SerialCtl("Prueba1", "Prueba2");
    System.out.println("Antes:\n" + cs);
    ByteArrayOutputStream buf =
      new ByteArrayOutputStream();
    ObjectOutputStream s =
      new ObjectOutputStream(buf);
    s.writeObject(cs);
    // Ahora, recuperarlo:
    ObjectInputStream entrada =
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf.toByteArray()));
    CtlSerial cs2 = (CtlSerial)entrada.readObject();
    System.out.println("Después:\n" + cs2);
  }
} ///:~
```

En este ejemplo, uno de los campos **String** es normal, y el otro es **transient**, para probar que se salva el campo no **transient** por parte del método **defaultWriteObject( )**, mientras que el campo **transient** se salva y recupera de forma explícita. Se inicializan los campos dentro del constructor en vez de definirlos para probar que no están siendo inicializados por ningún tipo de mecanismo automático durante la deserialización.

Si se va a usar el mecanismo por defecto para escribir las partes no **transient** del objeto, hay que invocar a **defaultWriteObject( )** como primera operación de **writeObject( )**, y a **defaultReadObject( )**, como primera operación de **readObject( )**. Éstas son llamadas extrañas a métodos. Podría parecer, por ejemplo, que está llamando al método **defaultWriteObject( )** de un **ObjectOutputStream** sin pasar argumentos, y así de algún modo convierte y conoce la referencia a su objeto y cómo escribir todas las partes no **transient**.

El almacenamiento y recuperación de objetos **transient** usa más código familiar. Y lo que es más: piense en lo que ocurre aquí. En el método **main( )**, se crea un objeto **CtlSerial**, y después se serializa a un **ObjectOutputStream**. (Nótese que en ese caso se usa un espacio de almacenamiento intermedio en vez de un archivo.) La serialización se realiza en la línea:

```
o.writeObject(cs);
```

El método **writeObject( )** debe examinar **cs** para averiguar si tiene su propio método **writeObject ( )**. (No comprobando la interfaz —pues no la hay— o el tipo de clase, sino buscando el método haciendo uso de la reflectividad.) Si lo tiene, se usa. Se sigue un enfoque semejante en el caso de **readObject( )**. Quizás ésta era la única forma, en la práctica, de solucionar el problema, pero es verdaderamente extraña.

Versionar

Es posible que se desee cambiar la versión de una clase serializable (por ejemplo, se podrían almacenar objetos de la clase original en una base de datos). Esto se soporta, pero probablemente se hará sólo en casos especiales, y requiere de una profundidad de entendimiento adicional que no trataremos de alcanzar aquí. La documentación JDK HTML descargable de *http://java.sun.com* cubre este tema de manera bastante detallada.

También se verá que muchos comentarios de la documentación JDK HTML comienzan por:

> ***Aviso**: Los objetos serializados de esta clase no serán compatibles con versiones futuras de Swing. El soporte actual para serialización es apropiado para almacenamiento a corto plazo o RMI entre aplicaciones...*

Esto se debe a que el mecanismo de versionado es demasiado simple como para que funcione correctamente en todas las situaciones, especialmente con JavaBeans. Actualmente se está trabajando en corregir su diseño, y por eso se presentan estas advertencias.

Utilizar la persistencia

Es bastante atractivo usar la tecnología de serialización para almacenar algún estado de un programa de forma que se pueda restaurar el programa al estado actual más adelante. Pero antes de poder hacer esto hay que resolver varias cuestiones. ¿Qué ocurre si se serializan dos objetos teniendo ambos una referencia a un tercero? Cuando se restauren esos dos objetos de su estado serializado ¿se obtiene sólo una ocurrencia del tercer objeto? ¿Qué ocurre si se serializan los dos objetos en archivos separados y se deserializan en partes distintas del código?

He aquí un ejemplo que muestra el problema:

```
//: c11:MiMundo.java
import java.io.*;
import java.util.*;

class Casa implements Serializable {}

class Animal implements Serializable {
  String nombre;
  Casa casaFavorita;
  Animal(String nm, Casa h) {
```

```
    nombre = nm;
    casaFavorita = h;
  }
  public String toString() {
    return nombre + "[" + super.toString() +
      "], " + casaFavorita + "\n";
  }
}

public class MiMundo {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    Casa casa = new Casa();
    ArrayList  animales = new ArrayList();
    animales.add(
      new Animal("Bosco el perro", casa));
    animales.add(
      new Animal("Ralph el hamster", casa));
    animales.add(
      new Animal("Fronk el gato", casa));
    System.out.println("animales: " + animales);

    ByteArrayOutputStream buf1 =
      new ByteArrayOutputStream();
    ObjectOutputStream s1 =
      new ObjectOutputStream(buf1);
    s1.writeObject(animales);
    s1.writeObject(animales); // Escribir un 2° conjunto
    // Escribir a un flujo distinto:
    ByteArrayOutputStream buf2 =
      new ByteArrayOutputStream();
    ObjectOutputStream s2 =
      new ObjectOutputStream(buf2);
    s2.writeObject(animales);
    // Ahora, recuperarlos:
    ObjectInputStream entrada1 =
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf1.toByteArray()));
    ObjectInputStream entrada2 =
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf2.toByteArray()));
    ArrayList animales1 =
      (ArrayList)entrada1.readObject();
```

```
      ArrayList animales2 =
         (ArrayList)entrada1.readObject();
      ArrayList animales3 =
         (ArrayList)entrada2.readObject();
      System.out.println("animales1: " + animales1);
      System.out.println("animales2: " + animales2);
      System.out.println("animales3: " + animales3);
   }
} ///:~
```

Una de las cosas interesantes aquí es que es posible usar la serialización de objetos para y desde un array de bytes logrando una "copia en profundidad" de cualquier objeto **Serializable**. (Una copia en profundidad implica duplicar la telaraña de objetos entera, en vez de simplemente el objeto básico y sus referencias). La copia se cubre en detalle en el Apéndice A.

Los objetos **Animal** contienen campos del tipo **Casa**. En el método **main( )**, se crea un **ArrayList** de estos **Animales** y se serializa dos veces en un flujo y de nuevo a otro flujo distinto. Cuando se deserializan e imprimen, se verán en la ejecución los resultados siguientes (en cada ejecución, los objetos estarán en distintas posiciones de memoria):

```
animales: [Bosco el perro[Animal@1cc76c], Casa@1cc769
, Ralph el hamster[Animal@1cc76d], Casa@1cc769
, Fronk el gato[Animal@1cc76e], Casa@1cc769
]
animales1: [Bosco el perro[Animal@1cca0c], Casa@1cca16
, Ralph el hamster[Animal@1cca17], Casa@1cca16
, Fronk el gato[Animal@1cca1b], Casa@1cca16
]
animales2: [Bosco el perro[Animal@1cca0c], Casa@1cca16
, Ralph el hamster[Animal@1cca17], Casa@1cca16
, Fronk el gato[Animal@1cca1b], Casa@1cca16
]
animales3: [Bosco el perro[Animal@1cca52], Casa@1cca5c
, Ralph el hamster[Animal@1cca5d], Casa@1cca5c
, Fronk el gato[Animal@1cca61], Casa@1cca5c
]
```

Por supuesto, se puede esperar que los objetos deserializados tengan direcciones distintas a la del original. Pero nótese que en **animales1** y **animales2** aparecen las mismas direcciones, incluyendo las referencias al objeto **Casa** que ambos comparten. Por otro lado, cuando se recupera **animales3** el sistema no puede saber que los objetos del otro flujo son alias de los objetos del primer flujo, por lo que construye una telaraña de objetos completamente diferente.

Mientras se serialice todo a un único flujo, se podrá recuperar la misma telaraña de objetos que se escribió, sin duplicaciones accidentales de los mismos. Por supuesto, se puede cambiar el estado de los objetos en el tiempo que transcurre entre la escritura del primero y el último, pero eso es res-

ponsabilidad de cada uno —los objetos se escribirán en el estado en el que estén (y con cualquier conexión que tengan con otros objetos) en el preciso momento de la serialización.

Lo más seguro si se desea salvar el estado de un sistema es hacer la serialización como una operación "atómica". Si se serializa una parte, se hacen otras cosas, luego se serializa otra parte, etc., no se estará almacenando el sistema de forma segura. Lo que hay que hacer es poner todos los objetos que conforman el estado del sistema en un único contenedor y simplemente se escribe este contenedor en una única operación. Después, es posible restaurarlo también con una única llamada a un método.

El ejemplo siguiente es un sistema de diseño asistido por ordenador (CAD) que demuestra el enfoque. Además, se introduce en el aspecto de los campos **static** —si se echa un vistazo a la documentación se verá que **Class** es **Serializable**, por lo que debería ser fácil almacenar campos **static** simplemente serializando el objeto **Class**. De cualquier forma, este enfoque parece sensato.

```
//: c11:EstadoCAD.java
// Almacenando y restaurando el estado de un
// sistema CAD aparente.
import java.io.*;
import java.util.*;

abstract class Figura implements Serializable {
  public static final int
    ROJO = 1, AZUL = 2, VERDE = 3;
  private int xPos, yPos, dimension;
  private static Random r = new Random();
  private static int contador = 0;
  abstract public void setColor(int nuevoColor);
  abstract public int getColor();
  public Figura(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass() +
      " color[" + getColor() +
      "] xPos[" + xPos +
      "] yPos[" + yPos +
      "] dim[" + dimension + "]\n";
  }
  public static Figura factoriaAleatoria() {
    int xVal = r.nextInt() % 100;
    int yVal = r.nextInt() % 100;
    int dim = r.nextInt() % 100;
    switch(contador++ % 3) {
```

```
        default:
        case 0: return new Circulo(xVal, yVal, dim);
        case 1: return new Cuadrado(xVal, yVal, dim);
        case 2: return new Linea(xVal, yVal, dim);
      }
    }
}

class Circulo extends Figura {
  private static int color = ROJO;
  public Circulo(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void estableceColor(int nuevoColor) {
    color = nuevoColor;
  }
  public int obtenerColor() {
    return color;
  }
}

class Cuadrado extends Figura {
  private static int color;
  public Cuadrado(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = ROJO;
  }
  public void establecerColor(int nuevoColor) {
    color = nuevoColor;
  }
  public int obtenerColor() {
    return color;
  }
}

class Linea extends Figura {
  private static int color = ROJO;
  public static void
  serializarEstadoEstatico(ObjectOutputStream os)
      throws IOException {
    os.writeInt(color);
  }
  public static void
  deserializarEstadoEstatico(ObjectInputStream os)
      throws IOException {
```

```
    color = os.readInt();
  }
  public Linea(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void establecerColor(int nuevoColor) {
    color = nuevoColor;
  }
  public int obtenerColor() {
    return color;
  }
}

public class EstadoCAD {
  public static void main(String[] args)
  throws Exception {
    ArrayList tiposFigura, figuras;
    if(args.length == 0) {
      tiposFigura = new ArrayList();
      figura = new ArrayList();
      // Añadir referencias a objetos de la clase:
      tiposFigura.add(Circulo.class);
      tiposFigura.add(Cuadrado.class);
      tiposFigura.add(Linea.class);
      // Construir algunas figuras:
      for(int i = 0; i < 10; i++)
        figuras.add(Figura.factoriaAleatoria());
      // Poner todos los colores estáticos a VERDE:
      for(int i = 0; i < 10; i++)
        ((Figura)figuras.get(i))
          .establecerColor(Poligono.VERDE);
      // Salvar el vector de estado:
      ObjectOutputStream salida =
        new ObjectOutputStream(
          new FileOutputStream("EstadoCAD.out"));
      salida.writeObject(Tiposfigura);
      Linea.serializacionEstadoEstatico(salida);
      salida.writeObject(figuras);
    } else { // Hay un parámetro de línea de comandos
      ObjectInputStream entrada =
        new ObjectInputStream(
          new FileInputStream(args[0]));
      // Leer de la misma forma en que se escribieron:
      tiposFigura = (ArrayList)entrada.readObject();
      Linea.deserializarEstadoEstatico(entrada);
```

```
        Figuras = (ArrayList)entrada.readObject();
      }
      // Mostrar los polígonos:
      System.out.println(figuras);
    }
} ///:~
```

La clase **Figura implementa Serializable**, por lo que cualquier cosa que herede de **Figura** es automáticamente también **Serializable**. Cada **Figura** contiene datos, y cada clase **Figura** derivada contiene un campo **static** que determina el color de todos esos tipos de **Figuras**. (Colocar un campo **static** en la clase base resultaría en un solo campo, mientras que los campos **static** no se duplican en las clases derivadas). Se pueden superponer los métodos de la clase base para establecer el color de los diversos tipos (los métodos **static** no se asignan dinámicamente, por lo que son método normales). El método **factoriaAleatoria( )** crea una **Figura** diferente cada vez que se le invoca, utilizando valores al azar para los datos **Figura**.

Círculo y **Cuadrado** son extensiones directas de **Figura**; la única diferencia radica en que **Círculo** inicializa **color** en el momento de su definición y **Cuadrado** lo inicializa en el constructor. Se dejará la discusión sobre **Línea** para un poco más adelante.

En el método **main( )**, se usa un **ArrayList** para guardar los objetos **Class** y otro para mantener las figuras. Si no se proporciona un parámetro de línea de comandos, se crea el **ArrayList tiposFigura** y se añaden los objetos **Class**, para posteriormente crear el **ArrayList figuras** y añadirle objetos **Figura**. A continuación, se ponen a **VERDE** todos los valores de **static color**, y todo se serializa al archivo **EstadoCAD.out**.

Si se proporciona un parámetro de línea de comandos (se supone que **EstadoCAD.out**) se abre ese archivo y se usa para restaurar el estado del programa. En ambas situaciones, se imprime el **ArrayList** de **Figuras**. Los resultados de una ejecución son:

```
<java EstadoCAD
[class Circulo color[3] xPos[-51] yPos[-99] dim[38]
, class Cuadrado color[3] xPos[2] yPos[61] dim[-46]
, class Linea color[3] xPos[51] yPos[73] dim[64]
, class Circulo color[3] xPos[-70] yPos][1] dim[16]
, class Cuadrado color[3] xPos[3] yPos[94] dim[-36]
, class Linea color[3] xPos[-80] yPos[-21] dim[-35]
, class Circulo color[3] xPos[-75] yPos[-43] dim[22]
, class Cuadrado color[3] xPos[81] yPos[30] dim[-45]
, class Linea color[3] xPos[-29] yPox[92] dim[17]
, class Circulo color[3] xPos[17] yPox[90] dim[-76]
]

<java EstadoCAD EstadoCAD.out
[class Circulo color[1] xPos[-51] yPos[-99] dim[38]
, class Cuadrado color[0] xPos[2] yPos[61] dim[-46]
, class Linea color[3] xPos[51] yPos[73] dim[64]
```

```
, class Circulo color[1] xPos[-70] yPos][1] dim[16]
, class Cuadrado color[0] xPos[3] yPos[94] dim[-36]
, class Linea color[3] xPos[-80] yPos[-21] dim[-35]
, class Circulo color[1] xPos[-75] yPos[-43] dim[22]
, class Cuadrado color[0] xPos[81] yPos[30] dim[-45]
, class Linea color[3] xPos[-29] yPox[92] dim[17]
, class Circulo color[1] xPos[17] yPox[90] dim[-76]
]
```

Se puede ver que los valores de **xPos**, **yPos** y **dim** se almacenaron y recuperaron satisfactoriamente, pero hay algo que no va bien al recuperar la información **static**. Se están introduciendo todo "3", pero no vuelve a visualizarse así. Los **Círculos** tienen valor 1 (**ROJO**, que es la definición), y los **Cuadrados** tienen valor 0 (recuérdese que se inicializan en el constructor). ¡Es como si, de hecho, los **static**s no se serializaran! Esto es así —incluso aunque la clase **Class** sea **Serializable**, no hace lo que se espera. Por tanto si se desea serializar **static**s, hay que hacerlo a mano.

Esto es para lo que sirven los métodos **static serializarEstadoEstatico( )** y **deserializar EstadoEstatico( )** de **Línea**. Se puede ver que se invocan explícitamente como parte del proceso de almacenamiento y recuperación. (Nótese que el orden de escritura al archivo serializado y el de lectura del mismo debe ser igual.) Por consiguiente, para que **EstadoCAD.java** funcione correctamente hay que:

1. Añadir un **serializarEstadoEstatico( )** y un **deserializarEstadoEstatico( )** a los polígonos.
2. Eliminar el **ArrayList tiposFigura** y todo el código relacionado con él.
3. Añadir llamadas a los nuevos métodos estáticos para serializar y deserializar en cada figura.

Otro aspecto a tener en cuenta es la seguridad, puesto que la serialización también almacena datos **private**. Si se tiene un problema de seguridad, habría que marcar los campos afectados como **transient**. Pero entonces hay que diseñar una forma segura de almacenar esa información, de forma que cuando se restaure, se puedan poner a cero esas variables **private**.

Identificar símbolos de una entrada

IdentificarSimbolos es el proceso de dividir una secuencia de caracteres en una secuencia de "símbolos", que son bits de texto delimitados por lo que se elija. Por ejemplo, los símbolos podrían ser palabras, pudiendo delimitarse por un espacio en blanco y signos de puntuación. Las dos clases proporcionadas por la biblioteca estándar de Java y que pueden ser usadas para poner símbolos son: **StreamTokenizer** y **StringTokenizer**.

StreamTokenizer

Aunque **StreamTokenizer** no deriva de **InputStream** ni de **OutputStream**, sólo funciona con objetos **InputStream**, por lo que pertenece a la parte de E/S de la biblioteca.

Considérese un programa que cuenta la ocurrencia de cada palabra en un archivo de texto:

```
//: c11:RecuentoPalabra.java
// Cuenta las palabras de un archivo, muestra los
// resultados de forma ordenada.
import java.io.*;
import java.util.*;

class Contador {
  private int i = 1;
  int leer() { return i; }
  void incrementar() { i++; }
}

public class RecuentoPalabra {
  private FileReader archivo;
  private StreamTokenizer st;
  // Un TreeMap mantiene las claves ordenadas:
  private TreeMap cuentas = new TreeMap();
  RecuentoPalabra(String nombreArchivo)
    throws FileNotFoundException {
    try {
      archivo = new FileReader(nombreArchivo);
      st = new StreamTokenizer(
        new BufferedReader(archivo));
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      System.err.println(
        "No se pudo abrir " + nombreArchivo);
      throw e;
    }
  }
  void limpieza() {
    try {
      archivo.close();
    } catch(IOException e) {
      System.err.println(
        "archivo.close() sin exito");
    }
  }
  void contarPalabras() {
    try {
      while(st.nextToken() !=
        StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
```

```
            case StreamTokenizer.TT_EOL:
              s = new String("EOL");
              break;
            case StreamTokenizer.TT_NUMBER:
              s = Double.toString(st.nval);
              break;
            case StreamTokenizer.TT_WORD:
              s = st.sval; // Es ya un String
              break;
            default: // sólo hay un carácter en ttype
              s = String.valueOf((char)st.ttype);
          }
          if(cuentas.containsKey(s))
            ((Contador)cuentas.get(s)).incrementar();
          else
            cuentas.put(s, new Contador());
        }
      } catch(IOException e) {
        System.err.println(
          "st.nextToken() sin exito");
      }
    }
    Collection valores() {
      return cuentas.values();
    }
    Set conjuntoClaves() { return cuentas.keySet(); }
    Contador obtenerContador(String s) {
      return (Contador)cuentas.get(s);
    }
    public static void main(String[] args)
    throws FileNotFoundException {
      RecuentoPalabra rp =
        new RecuentoPalabra(args[0]);
      rp.contarPalabras();
      Iterator claves = rp.conjuntoClaves().iterator();
      while(claves.hasNext()) {
        String clave = (String)claves.next();
        System.out.println(clave + ": "
                  + cp.obtenerContador(clave).read());
      }
      rp.limpieza();
    }
} ///:~
```

Es fácil presentar las palabras de forma ordenada almacenando los datos en un **TreeMap**, que organiza automáticamente sus claves en orden (véase Capítulo 9). Cuando se logra un conjunto de claves utilizando **keySet( )**, éstas también estarán en orden.

Para abrir el archivo, se usa un **FileReader**, y para convertir el archivo en palabras se crea un **StreamTokenizer** a partir del **FileReader** envuelto en un **BufferedReader**. En **StreamTokenizer**, hay una lista de separadores por defecto, y se pueden añadir más con un conjunto de métodos. Aquí, se usa **ordinaryChar( )** para decir: "Este carácter no tiene el significado en el que estoy interesado", por lo que el analizador no lo incluye como parte de las palabras que crea. Por ejemplo, decir **st.ordinaryChar('.')** quiere decir que no se incluirán los periodos como partes de las palabras a analizar. Se puede encontrar más información en la documentación JDK HTML de *http://java.sun.com*.

En **contarPalabras( )**, se sacan los símbolos de uno en uno desde el flujo, y se usa la información **ttype** para determinar qué hacer con cada símbolo, puesto que un símbolo puede ser un fin de línea, un número, una cadena de caracteres, o un único carácter.

Una vez que se encuentra un símbolo, se pregunta al **TreeMap cuentas** para ver si ya contiene el símbolo como clave. Si lo tiene, se incrementa el objeto **Contador** correspondiente, para indicar que se ha encontrado otra instancia de esa palabra. Si no, se crea un nuevo **Contador** —puesto que el constructor de **Contador** inicializa su valor a uno, esto también sirve para contar la palabra.

RecuentoPalabra no es un tipo de **TreeMap**, por lo que no heredó de éste. Lleva a cabo un tipo de funcionalidad especifico del tipo, por lo que incluso aunque hay que reexponer los métodos **keys( )** y **values( )**, eso sigue sin querer decir que debería usarse la herencia, puesto que utilizar varios métodos de **TreeMap** sería inadecuado. Además, se usan otros métodos como **obtenerContador( )**, que obtiene el **Contador** de un **String** en particular, y **sortedKeys( )**, que produce un **Iterator**, para finalizar el cambio en la forma de la interfaz de **RecuentoPalabra**.

En el método **main( )** se puede ver el uso de un **RecuentoPalabra** para abrir y contar las palabras de un archivo —simplemente ocupa dos líneas de código. Después se extrae un **Iterator** a una lista de claves (palabras) ordenadas, y se usa éste para extraer otra clave y su **Contador** asociado. La llamada a **limpieza( )** es necesaria para asegurar que se cierre el archivo.

StringTokenizer

Aunque éste no forma parte de la biblioteca de E/S, el **StringTokenizer** tiene funcionalidad lo suficientemente similar a **StreamTokenizer** como para describirlo aquí.

El **StringTokenizer** devuelve todos los símbolos contenidos en una cadena de caracteres de uno en uno. Estos símbolos son caracteres consecutivos delimitados por tabuladores, espacios y saltos de línea. Por consiguiente, los símbolos de la cadena "¿Dónde está mi gato?" son "¿Dónde", "está", "mi", y "gato?". Al igual que con **StreamTokenizer** se puede indicar a **StringTokenizer** que divida la entrada de la forma que desee, pero con **StringTokenizer** esto se logra pasando un segundo parámetro al constructor, que es un **String** con los delimitadores que se desea utilizar. En general, si se necesita más sofisticación, hay que usar un **StreamTokenizer**.

Para pedir a un **StringTokenizer** que te pase el siguiente token de la cadena se usa el método **nextToken( )** que o bien devuelve el símbolo, o bien una cadena de caracteres vacía para indicar que no quedan más símbolos.

A modo de ejemplo, el programa siguiente lleva a cabo un análisis limitado de una sentencia, buscando secuencias de frases clave para indicar si hay algún tipo de alegría o tristeza implicadas.

```
//: c11:AnalizarSentencia.java
// Buscar secuencias particulares en sentencias.
import java.util.*;

public class AnalizarSentencia {
  public static void main(String[] args) {
    analizar("Yo estoy contento por esto");
    analizar("Yo no estoy contento por esto");
    analizar("¡No lo estoy! Estoy contento");
    analizar("Yo no estoy triste por esto");
    analizar("Yo estoy triste por esto");
    analizar("¡No lo estoy! Estoy triste");
    analizar("¿Estas tu contento por esto?");
    analizar("¿Estas tu triste por esto?");
    analizar("¡Eres tu! Estoy contento");
    analizar("¡Eres tu! Estoy triste");
  }
  static StringTokenizer st;
  static void analizar(String s) {
    prt("\nnueva frase >> " + s);
    boolean triste = false;
    st = new StringTokenizer(s);
    while (st.hasMoreTokens()) {
      String símbolo = siguiente();
      // Buscar hasta encontrar uno de los dos
      // símbolos de inicio:
      if(!simbolo.equals("Yo") &&
          !simbolo.equals("¡Estas"))
        continue; // Parte de arriba del bucle while
      if(símbolo.equals("Yo")) {
        String s2 = siguiente();
        if(!s2.equals("estoy")) // Debe estar después de "yo"
          break; // Salir del bucle while
        else {
          String s3 = siguiente();
          if(s3.equals("triste")) {
            triste = true;
            break; // Salir del bucle while
```

```
            }
            if (s3.equals("no")) {
              String s4 = siguiente();
              if(s4.equals("triste"))
                break; // Dejar triste a false
              if(s4.equals("contento")) {
                triste = true;
                break;
              }
            }
          }
        }
        if(símbolo.equals("Estás")) {
          String s2 = siguiente();
          if(!s2.equals("tu"))
            break; // Debe ir después de éstas
          String s3 = siguiente();
          if(s3.equals("triste"))
            triste = true;
          break; // Salir del bucle while
        }
      }
      if(triste) prt("Tristeza detectada");
    }
    static String siguiente() {
      if(st.hasMoreTokens()) {
        String s = st.nextToken();
        prt(s);
        return s;
      }
      else
        return "";
    }
    static void prt(String s) {
      System.out.println(s);
    }
} ///:~
```

Por cada cadena de caracteres que se analiza, se entra en un bucle **while** y se extraen de la cadena los símbolos. Nótese la primera sentencia **if**, que dice que se **continúe** (volver al principio del bucle y comenzar de nuevo) si el símbolo no es ni "Yo" ni "Estás". Esto significa que cogerá símbolos hasta que se encuentre un "Yo" o un "Estás". Se podría pensar en usar == en vez del método **equals( )**, pero eso no funcionaría correctamente, pues == compara los valores de las referencias, mientras que **equals( )** compara contenidos.

La lógica del resto del método **analizar( )** es que el patrón que se está buscando es "Yo estoy triste", "Yo no estoy contento" o "¿Tú estás triste? Sin la sentencia **break**, el código habría sido aún más complicado de lo que ya es. Habría que ser conscientes de que un analizador típico (éste es un ejemplo primitivo de uno) normalmente tiene una tabla de estos símbolos y un fragmento de código que se mueve a través de los estados de la tabla a medida que se leen los nuevos símbolos.

Debería pensarse que **StringTokenizer** sólo es un atajo para un tipo simple y específico de **StreamTokenizer**. Sin embargo, si se tiene un **String** en el que se desean identificar símbolos, y **StringTokenizer** es demasiado limitado, todo lo que hay que hacer es convertirlo en un stream con **StringBufferInputStream** y después usarlo para crear **StreamTokenizer** mucho más potente.

Comprobar el estilo de escritura de mayúsculas

En esta sección, echaremos un vistazo a un ejemplo más completo del uso de la E/S de Java, que también hace uso de la identificación de símbolos. Este proyecto es directamente útil pues lleva a cabo una comprobación de estilo para asegurarse que el uso de mayúsculas se adecua al estilo de Java, tal y como se relata en *http://java.sun.com/docs/codeconv/index.html.* Abre cada archivo **.java** del directorio actual y extrae todos los nombres de archivos e identificadores, para mostrar después las que no utilicen el estilo Java.

Para que el programa funcione correctamente, hay que construir, en primer lugar, un repositorio de nombres de clases para guardar todos los nombres de clases de la biblioteca estándar de Java. Esto se logra moviendo todos los subdirectorios de código fuente de la biblioteca estándar de Java y ejecutando **ExploradorClases** en cada subdirectorio. Deben proporcionarse como argumentos el nombre del repositorio (usando la misma trayectoria y nombre siempre) y la opción de línea de comandos **–a** para indicar que deberían añadirse los nombres de clases al repositorio.

Al usar el programa para que compruebe el código de cada uno, hay que pasarle la trayectoria y el nombre del repositorio que debe usar. Comprobará todas las clases e identificadores en el directorio actual y dirá cuáles no siguen el ejemplo de uso de mayúsculas típico de Java.

Uno debería ser consciente de que el programa no es perfecto; pocas veces señalará algo que piensa que es un problema, pero que mirando al código se comprobará que no hay que cambiar nada. Esto es un poco confuso, pero sigue siendo más sencillo que intentar encontrar todos estos casos simplemente mirando cuidadosamente al código.

```
//: c11:ExploradorClases.java
// Busca clases e identificadores en todos los archivos
// de un directorio para comprobar el uso de mayúsculas.
// Asume listados de código que compilan adecuadamente.
// No lo hace todo bien pero es una ayuda
// útil.
import java.io.*;
import java.util.*;

class Mapa MultiCadena extends HashMap {
```

```
  public void add(String key, String value) {
    if(!containsKey(key))
      put(key, new ArrayList());
    ((ArrayList)get(key)).add(value);
  }
  public ArrayList getArrayList(String key) {
    if(!containsKey(key)) {
      System.err.println(
        "ERROR: no se puede encontrar la clave: " + key);
      System.exit(1);
    }
    return (ArrayList)get(key);
  }
  public void printValues(PrintStream p) {
    Iterator k = keySet().iterator();
    while(k.hasNext()) {
      String unaClave = (String)k.next();
      ArrayList val = getArrayList(oneKey);
      for(int i = 0; i < val.size(); i++)
        p.println((String)val.get(i));
    }
  }
}

public class ExploradorClases {
  private File ruta;
  private String[] listaArchivos;
  private Properties clases = new Properties();
  private MapaMultiCadena
    mapaClases = new MapaMultiCadena(),
    mapaIdent = new MapaMultiCadena();
  private StreamTokenizer entrada;
  public ExploradorClases() throws IOException {
    ruta = new File(".");
    listaArchivos = ruta.list(new FiltroJava());
    for(int i = 0; i < listaArchivos.length; i++) {
      System.out.println(listaArchivos[i]);
      try {
        explorarListado(listaArchivos[i]);
      } catch(FileNotFoundException e) {
        System.err.println("No se pudo abrir " +
          listaArchivos[i]);
      }
    }
  }
```

```
  void explorarListado(String nombreF)
  throws IOException {
    entrada = new StreamTokenizer(
         new BufferedReader(
           new FileReader(nombreF)));
    // Parece que no funciona:
    // entrada.slashStarComments(true);
    // entrada.slashSlashComments(true);
    entrada.ordinaryChar('/');
    entrada.ordinaryChar('.');
    entrada.wordChars('_', '_');
    entrada.eolIsSignificant(true);
    while(entrada.nextToken() !=
          StreamTokenizer.TT_EOF) {
      if(entrada.ttype == '/')
        comerComentarios();
      else if(entrada.ttype ==
                StreamTokenizer.TT_WORD) {
        if(entrada.sval.equals("class") ||
           entrada.sval.equals("interface")) {
          // Conseguir el nombre de la clase:
          while(entrada.nextToken() !=
                StreamTokenizer.TT_EOF
                && entrada.ttype !=
                StreamTokenizer.TT_WORD)
            ;
          clases.put(entrada.sval, entrada.sval);
          mapaClases.add(nombreF, entrada.sval);
        }
        if(entrada.sval.equals("import") ||
           entrada.sval.equals("package"))
          descartarLinea();
        else // Es un identificador o palabra clave
          mapaIdent.add(nombreF, entrada.sval);
      }
    }
  }
  void descartarLinea() throws IOException {
    while(entrada.nextToken() !=
          StreamTokenizer.TT_EOF
          && entrada.ttype !=
          StreamTokenizer.TT_EOL)
      ; // Lanza tokens al final de la línea
  }
  // La retirada del comentario de StreamTokenizer
```

```
// parecia rota. Esto los extrae:
void comerComentarios() throws IOException {
  if(entrada.nextToken() !=
      StreamTokenizer.TT_EOF) {
    if(entrada.ttype == '/')
      descartarLinea();
    else if(entrada.ttype != '*')
      entrada.pushBack();
    else
      while(true) {
        if(entrada.nextToken() ==
          StreamTokenizer.TT_EOF)
          break;
        if(entrada.ttype == '*')
          if(entrada.nextToken() !=
            StreamTokenizer.TT_EOF
            && entrada.ttype == '/')
            break;
      }
  }
}
public String[] nombresClases() {
  String[] resultado = new String[clases.size()];
  Iterator e = clases.keySet().iterator();
  int i = 0;
  while(e.hasNext())
    resultado[i++] = (String)e.next();
  return result;
}
public void comprobarNombresClases() {
  Iterator archivos = mapaClases.keySet().iterator();
  while(archivos.hasNext()) {
    String archivo = (String)archivos.next();
    ArrayList cls = mapaClases.getArrayList(archivo);
    for(int i = 0; i < cls.size(); i++) {
      String nombreClase = (String)cls.get(i);
      if(Character.isLowerCase(
            nombreClase.charAt(0)))
        System.out.println(
          "error de escritura de mayusculas, archivo: "
          + archivo + ", clase: "
          + nombreClase);
    }
  }
}
```

```
  public void comprobarNombreIdent() {
    Iterator archivos = mapaIdent.keySet().iterator();
    ArrayList conjuntoInformes = new ArrayList();
    while(archivos.hasNext()) {
      String archivo = (String)archivos.next();
      ArrayList ids = mapaIdent.getArrayList(archivo);
      for(int i = 0; i < ids.size(); i++) {
        String id = (String)ids.get(i);
        if(!clases.contains(id)) {
          // Ignorar identificadores de longitud 3 o
          // mayor que sean todo mayúsculas
          // (probablemente son valores static final):
          if(id.length() >= 3 &&
              id.equals(
                id.toUpperCase()))
            continue;
          // Comprobar para ver si el primer carácter es mayúscula:
          if(Character.isUpperCase(id.charAt(0))){
            if(conjuntoInformes.indexOf(archivo + id)
                == -1){ // No informado aun
              conjuntoInformes.add(archivo + id);
              System.out.println(
                "Error de mayúscula indentada en:"
                + archivo + ", ident: " + id);
            }
          }
        }
      }
    }
  static final String uso =
    "Uso: \n" +
    "ExploradorClases nombresClases -a\n" +
    "\tañade todos los nombres de clase de este \n" +
    "\tdirectorio al archivo repositorio \n" +
    "\tllamado 'nombresClases'\n" +
    "ExploradorClases nombresClases\n" +
    "\tComprueba todos los archivos java de este \n" +
    "\tdirectorio buscando errores de escritura de mayúsculas, \n" +
    "\tusando el archivo repositorio 'nombresClases'";
  private static void uso() {
    System.err.println(uso);
    System.exit(1);
  }
  public static void main(String[] args)
```

```
  throws IOException {
    if(args.length < 1 || args.length > 2)
      uso();
    ExploradorClases c = new ExploradorClases();
    File antiguo = new File(args[0]);
    if(antiguo.exists()) {
      try {
        // Intentar abrir un archivo de
        // propiedades existente:
        InputStream antiguolistado =
          new BufferedInputStream(
            new FileInputStream(antiguo));
        c.clases.load(antiguoListado);
        antiguoListado.close();
      } catch(IOException e) {
        System.err.println("No se pudo abrir "
          + antiguo + " en modo lectura");
        System.exit(1);
      }
    }
    if(args.length == 1) {
      c.comprobarNombresClases();
      c.comprobarNombresClases();
    }
    // Escribir los nombres de clase en un repositorio:
    if(args.length == 2) {
      if(!args[1].equals("-a"))
        uso();
      try {
        BufferedOutputStream salida =
          new BufferedOutputStream(
            new FileOutputStream(args[0]));
        c.clases.store(salida,
          "ExploradorClases.java encontro clases");
        salida.close();
      } catch(IOException e) {
        System.err.println(
          "No se pudo escribir " + args[0]);
        System.exit(1);
      }
    }
  }
}

class FiltroJava implements FilenameFilter {
```

```
  public boolean aceptar(File dir, String nombre) {
    // Eliminar información de trayectoria:
    String f = new File(nombre).getName();
    return f.trim().endsWith(".java");
  }
} ///:~
```

La clase **MapaMultiCadena** es una herramienta que permite establecer una correspondencia entre un grupo de cadenas de caracteres y su clave. Usa un **HashMap** (esta vez con herencia) con la clave como única cadena de caracteres con correspondencias sobre **ArrayList**. El método **add( )** simplemente comprueba si ya hay una clave en el **HashMap**, y si no, pone una. El método **getArrayList( )** produce un **ArrayList** para una clave en particular, y **printValues( )**, que es especialmente útil para depuración, imprime todos los valores de **ArrayList** en **ArrayList**.

Para que todo sea sencillo, se ponen todos los nombres de la biblioteca estándar de Java en un objeto **Properties** (de la biblioteca estándar de Java). Recuérdese que un objeto **Properties** es un **HashMap** que sólo guarda objetos **String,** tanto para las entradas de clave como para la de valor. Sin embargo, se puede salvar y restaurar a disco con una única llamada a un método, por lo que es ideal para un repositorio de nombres. De hecho, sólo necesitamos una lista de nombres, y un **HashMap** no puede aceptar **null,** ni para su entrada clave, ni para su entrada valor. Por tanto, se usará el mismo objeto tanto para la clave como para valor.

Para las clases e identificadores que se descubran para los archivos en un directorio en particular, se usan dos **MultiStringMaps**: **mapaClases** y **mapaIdent**. Además, cuando el programa empieza carga el repositorio de nombres de clase estándares en el objeto **Properties** llamado **clases**, y cuando se encuentra un nuevo nombre de clase en el directorio local se añade tanto a **clases** como a **mapaClases**. De esta forma, puede usarse **mapaClases** para recorrer todas las clases en el directorio local, y puede usarse **clases** para ver si el símbolo actual es un nombre de clase (que indica que comienza una definición de un objeto o un método).

El constructor por defecto de **ExploradorClases** crea una lista de nombres de archivo usando la implementación **FiltroJava** de **FilenameFilter**, mostrada al final del archivo. Después llama a **explorarListado( )** para cada nombre de archivo.

Dentro de **explorarListado( )** se abre el código fuente y se convierte en **StreamTokenizer**. En la documentación, se supone que pasar **true** a **slashStarComments( )** y **slashSlashComments( )** retira estos comentarios, pero parece un poco fraudulento, pues no funciona muy bien. En vez de ello, las líneas se marcan como comentarios que son extraídos por otro método. Para hacer esto, el "/" debe capturarse como un carácter normal, en vez de dejar a **StreamTokenizer** que lo absorba como parte de un comentario, y el método **ordinaryChar( )** dice al **StreamTokenizer** que lo haga así. Esto también funciona para los puntos ("."), puesto que se desea retirar las llamadas a métodos en forma de identificadores individuales. Sin embargo, el guión bajo, que suele ser tratado por **StreamTokenizer** como un carácter individual, debería dejarse como parte de los identificadores, pues aparece en valores **static final**, como **TT_EOF**, etc., usados en este mismo programa. El método **wordChars( )** toma un rango de caracteres que se desee añadir a los ya dejados dentro del símbolo a *analizar* como palabras. Finalmente, al analizar comentarios de una línea o descartar una lí-

nea, hay que saber cuándo se produce un fin de línea[4], por lo que se llama a **eolIsSignificant(true)** que mostrará los finales de línea en vez de dejar que sean absorbidos por el **StreamTokenizer**.

El resto de **explorarListado( )** lee y vuelve a actuar sobre los símbolos hasta el fin de fichero, que se encuentra cuando **nextToken( )** devuelva el valor **final static StreamTokenizer.TT_EOF**.

Si el símbolo es un "/" es potencialmente un comentario, por lo que se llama a **comerComentarios( )** para que lo maneje. Únicamente la otra situación que nos interesa en este caso es si es una palabra, donde pueden presentarse varios casos.

Si la palabra es **class** o **interfaz**, el siguiente símbolo representa un nombre de clase o interfaz, y se introduce en **clases** y **mapaClases**. Si la palabra es **import** o **package**, entonces no se desea el resto de la línea. Cualquier otra cosa debe ser un identificador (que nos interesa) o una palabra clave (que no nos interesa, pero que en cualquier caso se escriben con minúsculas, por lo que no pasa nada por incluirlas). Éstas se añaden a **mapaIdent**.

El método **descartarLínea( )** es una simple herramienta que busca finales de línea. Nótese que cada vez que se encuentre un nuevo símbolo, hay que comprobar los finales de línea.

El método **comerComentarios( )** es invocado siempre que se encuentra un "/" en el bucle de análisis principal. Sin embargo, eso no quiere decir necesariamente que se haya encontrado un comentario, por lo que hay que extraer el siguiente comentario para ver si hay otra barra diagonal (en cuyo caso se descarta toda la línea) o un asterisco. Si no estamos ante ninguna de éstas, ¡hay que volver a insertar el símbolo que se acaba de extraer! Afortunadamente, el método **pushBack( )** permite volver a introducir el símbolo actual en el flujo de entrada, de forma que cuando el bucle de análisis principal llame a **nextToken( )**, se obtendrá el que se acaba de introducir.

Por conveniencia, el método **nombresClases( )** produce un array de todos los nombres del contenedor **clases**. Este método no se usa en el programa pero es útil en procesos de depuración.

Los dos siguiente métodos son precisamente aquéllos en los que de hecho se realiza la comprobación . En **comprobarNombresClases( )**, se extraen los nombres de clase de **mapaClases** (que, recuérdese, contiene sólo los nombres de este directorio, organizados por nombre de archivo, de forma que se puede imprimir el nombre de archivo junto con el nombre de clase errante). Esto se logra extrayendo cada **ArrayList** asociado y recorriéndolo, tratando de ver si el primer carácter está en minúsculas. Si es así, se imprimirá el pertinente mensaje de error.

En **comprobarNombresIdent( )**, se sigue un enfoque similar: se extrae cada nombre de identificador de **mapaIdent**. Si el nombre no está en la lista **clases**, se asume que es un identificador o una palabra clave. Se comprueba un caso especial: si la longitud del identificador es tres o más *y* todos sus caracteres son mayúsculas, se ignora el identificador pues es probablemente un valor **static final** como **TT_EOF**. Por supuesto, éste no es un algoritmo perfecto, pero asume que generalmente los identificadores formados exclusivamente por letras mayúsculas se pueden ignorar.

---

[4] N. del traductor: en inglés *End Of Line* o *EOL*.

En vez de informar de todos los identificadores que empiecen con una mayúscula, este método mantiene un seguimiento de aquéllos para los que ya se ha generado un informe en un **ArrayList** denominado **conjuntoInformes( )**. Éste trata al **ArrayList** como un "conjunto" que indica si un elemento se encuentra o no en el conjunto. El elemento se genera concatenando el nombre de archivo y el identificador. Si el elemento no está en el conjunto, se añade y después se emite el informe.

El resto del listado se lleva a cabo en el método **main( )**, que se mantiene ocupado manejando los parámetros de línea de comandos y averiguando si se está o no construyendo un repositorio de nombres de clase a partir de la biblioteca estándar de Java o comprobando la validez del código escrito. En ambos casos hace un objeto **ExploradorClase**.

Se esté construyendo un repositorio o utilizando uno, hay que intentar abrir el repositorio existente. Haciendo un objeto **File** y comprobando su existencia, se puede decidir si abrir un archivo y **load( )** las **clases** de la lista de **Properties** dentro de **ExploradorClase**. (Las clases del repositorio se añaden, más que sobreescribirse, a las clases encontradas en el constructor **ExploradorClase**.) Si se proporciona sólo un parámetro en línea de comandos, se quiere llevar a cabo una comprobación de nombres de clase e identificadores, pero si se proporcionan dos argumentos (siendo el segundo "**-a**") se está construyendo un repositorio de nombres de clase. En este caso, se abre un archivo de salida y se usa el método **Properties.save( )** para escribir la lista en un archivo, junto con una cadena de caracteres que proporciona información de cabecera de archivo.

Resumen

La biblioteca de flujos de E/S de Java satisface los requisitos básicos: se puede llevar a cabo lectura y escritura con la consola, un archivo, un bloque de memoria o incluso a través de Internet (como se verá en el Capítulo 15). Con la herencia, se pueden crear nuevos tipos de objetos de entrada y salida. E incluso se puede añadir una extensibilidad simple a los tipos de objetos que aceptará un flujo redefiniendo el método **toString( )** que se invoca automáticamente cuando se pasa un objeto a un método que esté esperando un **String** (la "conversión automática de tipos" limitada de Java).

En la documentación y diseño de la biblioteca de flujos de E/S quedan cuestiones sin contestar. Por ejemplo, habría sido genial si se pudiese decir que se desea que se lance una excepción si se intenta sobreescribir un archivo cuando se abre como salida —algunos sistemas de programación permiten especificar que se desea abrir un archivo de salida, pero sólo si no existe aún. En Java, parece suponerse que uno usará un objeto **File** para determinar si existe un archivo, porque si se abre como un **FileOutputStream** o **FileWriter** siempre será sobreescrito.

La biblioteca de flujos de E/S trae a la mente sentimientos entremezclados; hace gran parte del trabajo y es portable. Pero si no se entiende ya el patrón decorador, el diseño no es intuitivo, por lo que hay una gran sobrecarga en lo que a aprendizaje y enseñanza de la misma se refiere. También está incompleta: no hay soporte para dar formato a la salida, soportado casi por el resto de paquetes de E/S del resto de lenguajes.

Sin embargo, una vez que *se entiende* el patrón decorador, y se empieza a usar la biblioteca en situaciones que requieren su flexibilidad, se puede empezar a beneficiar de este diseño, punto en el que su coste en líneas extra puede no molestar tanto.

Si no se encuentra lo que se estaba buscando en este capítulo (que no ha sido más que una introducción, que no pretendía ser comprensivo) se puede encontrar información en profundidad en *Java I/O*, de Elliote Rusty Harold (O'Reilly, 1999).

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Abrir un archivo de texto de forma que se pueda leer del mismo de línea en línea. Leer cada línea como un **String** y ubicar ese objeto **String** en un **LinkedList**. Imprimir todas las líneas del **LinkedList** en orden inverso.
2. Modificar el Ejercicio 1 de forma que el nombre del archivo que se lea sea proporcionado como un parámetro de línea de comandos.
3. Modificar el Ejercicio 2 para que también abra un archivo de texto de forma que se pueda escribir texto en el mismo. Escribir las líneas del **ArrayList**, junto con los números de línea (no intentar usar las clases "LineNumber"), fuera del archivo.
4. Modificar el Ejercicio 2 para forzar que todas las líneas de **ArrayList** estén en mayúsculas y enviar los resultados a **System.out**.
5. Modificar el Ejercicio 2 para que tome palabras a buscar dentro del archivo como parámetros adicionales de línea de comandos. Imprimir todas las líneas en las que casen las palabras.
6. Modificar **ListadoDirectorio.java** de forma que **FilenameFilter** abra cada archivo y acepte el archivo basado en la existencia de alguno de los parámetros de la línea de comandos en ese archivo.
7. Crear una clase denominada **ListadoDirectorioOrdenado** con un constructor que tome información de una ruta de archivo y construya un listado de directorio ordenado con todos los archivos de esa ruta. Crear dos métodos **listar( )** sobrecargados que, bien produzcan toda la lista, o bien un subconjunto de la misma basándose en un argumento. Añadir un método **tamanio( )** que tome un nombre de archivo y produzca el tamaño de ese archivo.
8. Modificar **RecuentoPalabra.java** para que produzca un orden alfabético en su lugar, utilizando la herramienta del Capítulo 9.
9. Modificar **RecuentoPalabra.java** de forma que use una clase que contenga un **String** y un valor de cuenta para almacenar cada palabra, y un **Set** de esos objetos para mantener la lista de palabras.

10. Modificar **DemoFlujoES.java** de forma que use **LineNumberInputStream** para hacer un seguimiento del recuento de líneas. Nótese que es mucho más fácil mantener el seguimiento desde la programación.

11. Basándonos en la Sección 4 de **DemoFlujoES.java,** escribir un programa que compare el rendimiento de escribir en un archivo al usar E/S con y sin espacios de almacenamiento intermedio.

12. Modificar la Sección 5 de **DemoFlujoES.java** para eliminar los espacios en la línea producida por la primera llamada a **entrada5br.readLine( )**. Hacerlo utilizando un bucle **while** y **readChar( )**.

13. Reparar el programa **EstadoCAD.java** tal y como se describe en el texto.

14. En **Rastros.java**, copiar el archivo y renombrarlo a **ComprobarRastro.java**, y renombrar la clase **Rastro2** a **ComprobarRastro** (haciéndola además **public** y retirando el ámbito público de la clase **Rastros** en el proceso). Eliminar las marcas **//!** del final del archivo y ejecutar el programa incluyendo las líneas que causaban ofensa. A continuación, marcar como comentario el constructor por defecto de **ComprobarRastro**. Ejecutarlo y explicar por qué funciona. Nótese que tras compilar, hay que ejecutar el programa con "**java Rastros**" porque el método **main( )** sigue en la clase **Rastros.**

15. En **Rastro3.java**, marcar como comentarios las dos líneas tras las frases "Hay que hacer esto:" y ejecutar el programa. Explicar el resultado y por qué difiere de cuando las dos líneas se encuentran en el programa.

16. (Intermedio) En el Capítulo 8, localizar el ejemplo **ControlesCasaVerde.java**, que consiste en tres archivos. En **ControlesCasaVerde.java**, la clase interna **Rearrancar( )** tiene un conjunto de eventos duramente codificados. Cambiar el programa de forma que lea los eventos y sus horas relativas desde un archivo de texto. (Desafío: utilizar un *método factoría* de patrones de diseño para construir los eventos —véase *Thinking in Patterns with Java*, descargable desde *http://www.BruceEckel.com*.)