8. EXCEPCIONES

A diferencia de otros lenguajes de programación orientados a objetos como C/C++, ***Java*** incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje ***Java***, una ***Exception*** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas ***excepciones*** son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado).

Un buen programa debe gestionar correctamente todas o la mayor parte de los errores que se pueden producir. Hay dos "estilos" de hacer esto:

1. ***A la "antigua usanza"***: los métodos devuelven un código de error. Este código se chequea en el entorno que ha llamado al método con una serie de ***if elseif*** …, gestionando de forma diferente el resultado correcto o cada uno de los posibles errores. Este sistema resulta muy complicado cuando hay varios niveles de llamadas a los métodos.
2. ***Con soporte en el propio lenguaje***: En este caso el propio lenguaje proporciona construcciones especiales para gestionar los errores o ***Exceptions***. Suele ser lo habitual en lenguajes modernos, como C++, Visual Basic y ***Java***.

En los siguientes apartados se examina cómo se trabaja con los bloques y expresiones ***try***, ***catch***, ***throw***, ***throws*** y ***finally***, cuándo se deben lanzar excepciones, cuándo se deben capturar y cómo se crean las clases propias de tipo ***Exception***.

8.1 EXCEPCIONES ESTÁNDAR DE JAVA

Los errores se representan mediante dos tipos de clases derivadas de la clase ***Throwable***: ***Error*** y ***Exception***. La siguiente figura muestra parcialmente la jerarquía de clases relacionada con ***Throwable***:

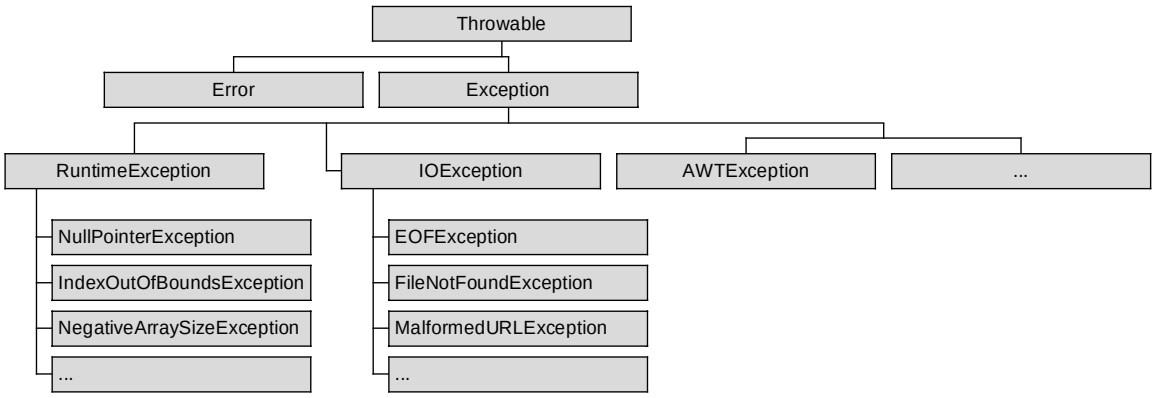

Figura 8.1: Jerarquía de clases derivadas de Throwable.

La clase ***Error*** está relacionada con errores de compilación, del sistema o de la JVM. De ordinario estos errores son ***irrecuperables*** y no dependen del programador ni debe preocuparse de capturarlos y tratarlos.

La clase ***Exception*** tiene más interés. Dentro de ella se puede distinguir:

1. ***RuntimeException***: Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar ***excepciones implícitas***.
2. Las demás clases derivadas de ***Exception*** son ***excepciones explícitas***. ***Java*** obliga a tenerlas en cuenta y chequear si se producen.

El caso de ***RuntimeException*** es un poco especial. El propio ***Java*** durante la ejecución de un programa chequea y lanza automáticamente las excepciones que derivan de ***RuntimeException***. El programador no necesita establecer los bloques ***try***/***catch*** para controlar este tipo de excepciones. Representan dos casos de errores de programación:

1. Un error que normalmente no suele ser chequeado por el programador, como por ejemplo recibir una referencia ***null*** en un método.
2. Un error que el programador debería haber chequeado al escribir el código, como sobrepasar el tamaño asignado de un array (genera un *ArrayIndexOutOfBoundsException* automáticamente).

En realidad sería posible comprobar estos tipos de errores, pero el código se complicaría excesivamente si se necesitara chequear continuamente todo tipo de errores (que las ***referencias*** son distintas de ***null***, que todos los argumentos de los métodos son correctos, y un largo etcétera).

Las clases derivadas de ***Exception*** pueden pertenecer a distintos packages de ***Java***. Algunas perenecen a ***java.lang*** (*Throwable*, *Exception*, *RuntimeException*, …); otras a ***java.io*** (*EOFException*, *FileNotFoundException*, ...) o a otros packages. Por heredar de ***Throwable*** todos los tipos de excepciones pueden usar los métodos siguientes:

1. String ***getMessage()*** Extrae el mensaje asociado con la excepción.
2. String ***toString()*** Devuelve un String que describe la excepción.
3. void ***printStackTrace()*** Indica el método donde se lanzó la excepción.

8.2 LANZAR UNA EXCEPTION

Cuando en un método se produce una situación anómala es necesario lanzar una excepción. El proceso de lanzamiento de una excepción es el siguiente:

1. Se crea un objeto ***Exception*** de la clase adecuada.
2. Se lanza la excepción con la sentencia ***throw*** seguida del objeto ***Exception*** creado.

```
// Código que lanza la excepción MyException una vez detectado el error
MyException me = new MyException("MyException message");
throw me;
```

Esta excepción deberá ser capturada (***catch***) y gestionada en el propio método o en algún otro lugar del programa (en otro método anterior en la ***pila*** o ***stack*** de llamadas), según se explica en el Apartado 8.3.

Al lanzar una excepción el método termina de inmediato, sin devolver ningún valor. Solamente en el caso de que el método incluya los bloques ***try/catch/finally*** se ejecutará el bloque ***catch*** que la captura o el bloque ***finally*** (si existe).

Todo método en el que se puede producir uno o más tipos de excepciones (y que no utiliza directamente los bloques ***try/catch/finally*** para tratarlos) debe ***declararlas*** en el encabezamiento de la función por medio de la palabra ***throws***. Si un método puede lanzar varias excepciones, se ponen detrás de ***throws*** separadas por comas, como por ejemplo:

```
public void leerFichero(String fich) throws EOFException, FileNotFoundException {…}
```

Se puede poner únicamente una ***superclase de excepciones*** para indicar que se pueden lanzar excepciones de cualquiera de sus clases derivadas. El caso anterior sería equivalente a:

```
public void leerFichero(String fich) throws IOException {…}
```

Las excepciones pueden ser lanzadas directamente por ***leerFichero()*** o por alguno de los métodos llamados por ***leerFichero()***, ya que las clases ***EOFException*** y ***FileNotFoundException*** derivan de ***IOException***.

Se recuerda que no hace falta avisar de que se pueden lanzar objetos de la clases ***Error*** o ***RuntimeException*** (excepciones implícitas).

8.3 CAPTURAR UNA EXCEPTION

Como ya se ha visto, ciertos métodos de los packages de ***Java*** y algunos métodos creados por cualquier programador producen ("lanzan") excepciones. Si el usuario llama a estos métodos sin tenerlo en cuenta se produce un error de compilación con un mensaje del tipo: "… *Exception java.io.IOException must be caugth or it must be declared in the throws clause of this method*". El programa no compilará mientras el usuario no haga una de estas dos cosas:

1. ***Gestionar la excepción*** con una construcción del tipo ***try {…} catch {…}***.
2. ***Re-lanzar la excepción*** hacia un método anterior en el ***stack***, declarando que su método también lanza dicha excepción, utilizando para ello la construcción ***throws*** en el header del método.

El compilador obliga a capturar las llamadas ***excepciones explícitas***, pero no protesta si se captura y luego no se hace nada con ella. En general, es conveniente por lo menos imprimir un mensaje indicando qué tipo de excepción se ha producido.

8.3.1 Bloques try y catch

En el caso de las excepciones que no pertenecen a las ***RuntimeException*** y que por lo tanto ***Java*** obliga a tenerlas en cuenta habrá que utilizar los bloques ***try***, ***catch*** y ***finally***. El código dentro del bloque ***try*** está "vigilado": Si se produce una situación anormal y se lanza por lo tanto una excepción el control salta o sale del bloque ***try*** y pasa al bloque ***catch***, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques ***catch*** como sean necesarios, cada uno de los cuales tratará un tipo de excepción.

Las excepciones se pueden capturar individualmente o en grupo, por medio de una superclase de la que deriven todas ellas.

El bloque ***finally*** es opcional. Si se incluye sus sentencias se ejecutan siempre, sea cual sea la excepción que se produzca o si no se produce ninguna. El bloque ***finally*** se ejecuta aunque en el bloque ***try*** haya un ***return***.

En el siguiente ejemplo se presenta un método que debe "controlar" una ***IOException*** relacionada con la lectura ficheros y una ***MyException*** propia:

```
void metodo1(){
    ...
    try {
        // Código que puede lanzar las excepciones IOException y MyException
    } catch (IOException e1) {// Se ocupa de IOException simplemente dando aviso
        System.out.println(e1.getMessage());
    } catch (MyException e2) {
        // Se ocupa de MyException dando un aviso y finalizando la función
        System.out.println(e2.getMessage()); return;
    } finally { // Sentencias que se ejecutarán en cualquier caso
        ...
    }
    ...
} // Fin del metodo1
```

8.3.2 Relanzar una Exception

Existen algunos casos en los cuales el código de un método puede generar una ***Exception*** y no se desea incluir en dicho método la gestión del error. ***Java*** permite que este método pase o relance (***throws***) la ***Exception*** al método desde el que ha sido llamado, sin incluir en el método los bucles ***try/catch*** correspondientes. Esto se consigue mediante la adición de ***throws*** más el nombre de la ***Exception*** concreta después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques ***try/catch*** o volver a pasar la ***Exception***. De esta forma se puede ir pasando la ***Exception*** de un método a otro hasta llegar al último método del programa, el método ***main()***.

El ejemplo anterior (***metodo1***) realizaba la gestión de las excepciones dentro del propio método. Ahora se presenta un nuevo ejemplo (***metodo2***) que relanza las excepciones al siguiente método:

```
void metodo2() throws IOException, MyException {
    ...
    // Código que puede lanzar las excepciones IOException y MyException
    ...
} // Fin del metodo2
```

Según lo anterior, si un método llama a otros métodos que pueden lanzar excepciones (por ejemplo de un package de ***Java***), tiene 2 posibilidades:

1. ***Capturar*** las posibles excepciones y gestionarlas.
2. Desentenderse de las excepciones y ***remitirlas*** hacia otro método anterior en el ***stack*** para éste se encargue de gestionarlas.

Si no hace ninguna de las dos cosas anteriores el compilador da un error, salvo que se trate de una ***RuntimeException***.

8.3.3 Método finally {...}

El bloque ***finally {...}*** debe ir detrás de todos los bloques ***catch*** considerados. Si se incluye (ya que es opcional) sus sentencias se ejecutan siempre, sea cual sea el tipo de excepción que se produzca, o ***incluso si no se produce ninguna***. El bloque ***finally*** se ejecuta incluso si dentro de los bloques ***try/catch*** hay una sentencia ***continue***, ***break*** o ***return***. La forma general de una sección donde se controlan las excepciones es por lo tanto:

```
try {
    // Código "vigilado" que puede lanzar una excepción de tipo A, B o C
} catch (A a1) {
    // Se ocupa de la excepción A
} catch (B b1) {
    // Se ocupa de la excepción B
} catch (C c1) {
    // Se ocupa de la excepción C
} finally {
    // Sentencias que se ejecutarán en cualquier caso
}
```

El bloque ***finally*** es necesario en los casos en que se necesite recuperar o devolver a su situación original algunos elementos. No se trata de liberar la memoria reservada con ***new*** ya que de ello se ocupará automáticamente el *garbage collector*.

Como ejemplo se podría pensar en un bloque ***try*** dentro del cual se abre un fichero para lectura y escritura de datos y se desea cerrar el fichero abierto. El fichero abierto se debe cerrar tanto si produce una excepción como si no se produce, ya que dejar un fichero abierto puede provocar problemas posteriores. Para conseguir esto se deberá incluir las sentencias correspondientes a cerrar el fichero dentro del bloque ***finally***.

8.4 CREAR NUEVAS EXCEPCIONES

El programador puede crear sus propias excepciones sólo con heredar de la clase ***Exception*** o de una de sus clases derivadas. Lo lógico es heredar de la clase de la jerarquía de ***Java*** que mejor se adapte al tipo de excepción. Las clases ***Exception*** suelen tener dos constructores:

1. Un ***constructor*** sin argumentos.
2. Un ***constructor*** que recibe un ***String*** como argumento. En este ***String*** se suele definir un mensaje que explica el tipo de excepción generada. Conviene que este constructor llame al constructor de la clase de la que deriva ***super(String)***.

Al ser clases como cualquier otra se podrían incluir variables y métodos nuevos. Por ejemplo:

```
class MiExcepcion extends Exception {
    public MiExcepcion() {                  // Constructor por defecto
        super();
    }
    public MiExcepción(String s) {          // Constructor con mensaje
        super(s);
    }
}
```

8.5 HERENCIA DE CLASES Y TRATAMIENTO DE EXCEPCIONES

Si un método redefine otro método de una super-clase que utiliza ***throws***, el método de la clase derivada no tiene obligatoriamente que poder lanzar todas las mismas excepciones de la clase base. Es posible en el método de la subclase lanzar ***las mismas excepciones o menos***, pero no se pueden lanzar más excepciones. No puede tampoco lanzar nuevas excepciones ni excepciones de una clase más general.

Se trata de una restricción muy útil ya que como consecuencia de ello el código que funciona con la clase base podrá trabajar automáticamente con referencias de clases derivadas, incluyendo el tratamiento de excepciones, concepto fundamental en la *Programación Orientada a Objetos* (***polimorfismo***).

9. ENTRADA/SALIDA DE DATOS EN JAVA 1.1

Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar, como para devolver los resultados obtenidos.

La manera de representar estas entradas y salidas en ***Java*** es a base de ***streams*** (flujos de datos). Un ***stream*** es una conexión entre el programa y la fuente o destino de los datos. La información se traslada ***en serie*** (un carácter a continuación de otro) a través de esta conexión. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.

Por ejemplo, cuando se quiere imprimir algo en pantalla, se hace a través de un ***stream*** que conecta el monitor al programa. Se da a ese ***stream*** la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de Internet o la lectura de la información de un sensor a través del puerto en serie.

9.1 CLASES DE JAVA PARA LECTURA Y ESCRITURA DE DATOS

El package ***java.io*** contiene las clases necesarias para la comunicación del programa con el exterior. Dentro de este package existen dos familias de jerarquías distintas para la entrada/salida de datos. La diferencia principal consiste en que una opera con ***bytes*** y la otra con ***caracteres*** (el carácter de ***Java*** está formado por dos bytes porque sigue el código ***Unicode***). En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.

Desde ***Java 1.0***, la entrada y salida de datos del programa se podía hacer con clases derivadas de ***InputStream*** (para lectura) y ***OutputStream*** (para escritura). Estas clases tienen los métodos básicos ***read()*** y ***write()*** que manejan ***bytes*** y que no se suelen utilizar directamente. La Figura 9.1 muestra las clases que derivan de ***InputStream*** y la Figura 9.2 las que derivan de ***OutputStream***.

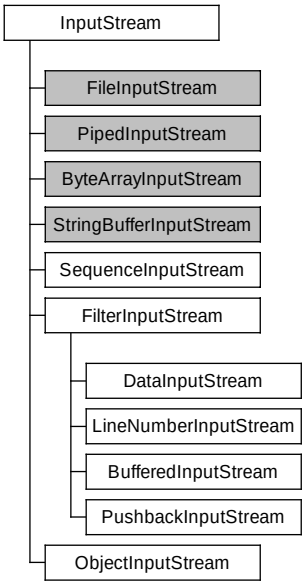

Figura 9.1. Jerarquía de clases InputStream.

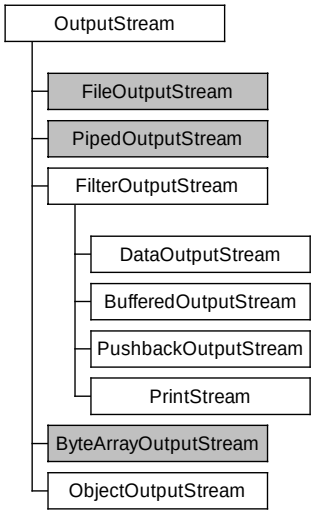

Figura 9.2. Jerarquía de clases OutputStream.

En ***Java 1.1*** aparecieron dos nuevas familias de clases, derivadas de ***Reader*** y ***Writer***, que manejan ***caracteres*** en vez de ***bytes***. Estas clases resultan más prácticas para las aplicaciones en las

que se maneja texto. Las clases que heredan de ***Reader*** están incluidas en la Figura 9.3 y las que heredan de ***Writer*** en la Figura 9.4.

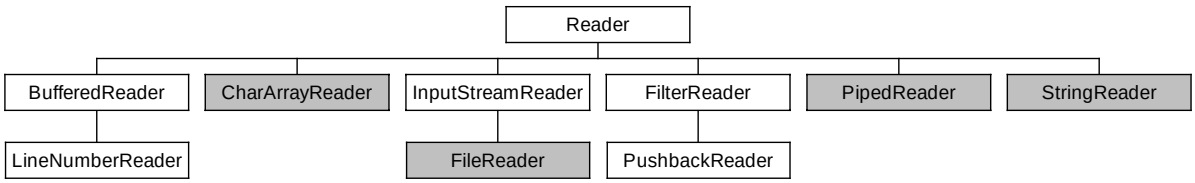

Figura 9.3. Jerarquía de clases Reader.

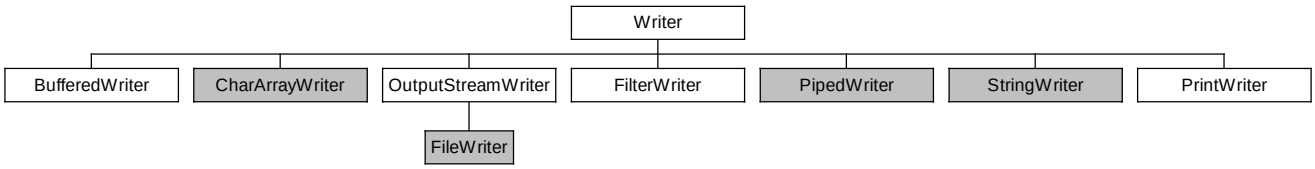

Figura 9.4. Jerarquía de clases Writer.

En las cuatro últimas figuras las clases con ***fondo gris*** definen de dónde o a dónde se están enviando los datos, es decir, el dispositivo con que conecta el ***stream***. Las demás (***fondo blanco***) añaden características particulares a la forma de enviarlos. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader in = new BufferedReader(new FileReader("autoexec.bat"));
```

Con esta línea se ha creado un ***stream*** que permite leer del archivo ***autoexec.bat***. Además, se ha creado a partir de él un objeto ***BufferedReader*** (que aporta la característica de utilizar ***buffer***[6]). Los caracteres que lleguen a través del ***FileReader*** pasarán a través del ***BufferedReader***, es decir utilizarán el ***buffer***.

A la hora de definir una comunicación con un dispositivo siempre se comenzará determinando el origen o destino de la comunicación (***clases en gris***) y luego se le añadiran otras características (***clases en blanco***).

Se recomienda utilizar siempre que sea posible las clases ***Reader*** y ***Writer***, dejando las de ***Java 1.0*** para cuando sean imprescindibles. Algunas tareas como la ***serialización*** y la ***compresión*** necesitan las clases ***InputStream*** y ***OutputStream***.

9.1.1 Los nombres de las clases de java.io

Las clases de ***java.io*** siguen una nomenclatura sistemática que permite deducir su función a partir de las palabras que componen el nombre, tal como se describe en la Tabla 9.1.

---

[6] Un ***buffer*** es un espacio de memoria intermedia que actúa de "colchón" de datos. Cuando se necesita un dato del disco se trae a memoria ese dato y sus datos contiguos, de modo que la siguiente vez que se necesite algo del disco la probabilidad de que esté ya en memoria sea muy alta. Algo similar se hace para escritura, intentando realizar en una sola operación de escritura física varias sentencias individuales de escritura.

Palabra	**Significado**
InputStream, OutputStream	Lectura/Escritura de bytes
Reader, Writer	Lectura/Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

Tabla 9.1. Palabras identificativas de las clases de java.io.

9.1.2 Clases que indican el origen o destino de los datos

La Tabla 9.2 explica el uso de las clases que definen el lugar con que conecta el ***stream***.

Clases	**Función que realizan**
FileReader, FileWriter, FileInputStream y FileOutputStream	Son las clases que leen y escriben en **archivos** de disco. Se explicarán luego con más detalle.
StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream	Estas clases tienen en común que se comunican con la **memoria** del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara carácter a carácter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa.
PipedReader, PipedWriter, PipedInputStream, PipedOutputStream	Se utilizan como un "tubo" o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro el PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. Tambíen puede comunicar a dos programas distintos.

Tabla 9.2. Clases que indican el origen o destino de los datos.

9.1.3 Clases que añaden características

La Tabla 9.3 explica las funciones de las clases que alteran el comportamiento de un ***stream*** ya definido.

Clases	**Función que realizan**
BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream	Como ya se ha dicho, añaden un **buffer** al manejo de los datos. Es decir, se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. ***BufferedReader*** por ejemplo tiene el método ***readLine()*** que lee una línea y la devuelve como un String.
InputStreamReader, OutputStreamWriter	Son clases puente que permiten **convertir** streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la transformación inversa.
ObjectInputStream, ObjectOutputStream	Pertenecen al mecanismo de la **serialización** y se explicarán más adelante.
FilterReader, FilterWriter, FilterInputStream, FilterOutputStream	Son clases base para aplicar diversos **filtros** o procesos al stream de datos. También se podrían extender para conseguir comportamientos a medida.
DataInputStream, DataOutputStream	Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible , pero independiente de plataforma y se usan por tanto para **almacenaje** o para **transmisiones** entre ordenadores de distinto funcionamiento.
PrintWriter, PrintStream	Tienen métodos adaptados para **imprimir** las variables de Java con la apariencia normal. A partir de un boolean escriben “true” o “false”, colocan la coma de un número decimal, etc.

Tabla 9.3. Clases que añaden características.

9.2 ENTRADA Y SALIDA ESTÁNDAR (TECLADO Y PANTALLA)

En ***Java***, la entrada desde teclado y la salida a pantalla están reguladas a traves de la clase ***System***. Esta clase pertenece al package ***java.lang*** y agrupa diversos métodos y objetos que tienen relación con el sistema local. Contiene, entre otros, tres objetos ***static*** que son:

System.in: Objeto de la clase ***InputStream*** preparado para recibir datos desde la entrada estándar del sistema (habitualmente el teclado).

System.out: Objeto de la clase ***PrintStream*** que imprimirá los datos en la salida estándar del sistema (normalmente asociado con la pantalla).

System.err: Objeto de la clase ***PrintStream***. Utilizado para mensajes de error que salen también por pantalla por defecto.

Estas clases permiten la comunicación alfanumérica con el programa a través de lo métodos incluidos en la Tabla 9.4. Son métodos que permiten la entrada/salida a un nivel muy elemental.

Métodos de System.in	**Función que realizan**
int read()	Lee un carácter y lo devuelve como int.
Métodos de System.out y System.err	**Función que realizan**
int print(cualquier tipo)	Imprime en pantalla el argumento que se le pase. Puede recibir cualquier tipo primitivo de variable de Java.
int println(cualquier tipo)	Como el anterior, pero añadiendo '\n' (nueva línea) al final.

Tabla 9.4. Métodos elementales de lectura y escritura.

Existen tres métodos de ***System*** que permiten sustituir la entrada y salida estándar. Por ejemplo, se utiliza para hacer que el programa lea de un archivo y no del teclado.

```
System.setIn(InputStream is);
System.setOut(PrintStream ps);
System.setErr(PrintStream ps);
```

El argumento de ***setIn()*** no tiene que ser necesariamente del tipo ***InputStream***. Es una referencia a la clase base, y por tanto puede apuntar a objetos de cualquiera de sus clases derivadas (como ***FileInputStream***). Asimismo, el constructor de ***PrintStream*** acepta un ***OutputStream***, luego se puede dirigir la salida estándar a cualquiera de las clases definidas para salida.

Si se utilizan estas sentencias con un compilador de ***Java 1.1*** se obtiene un mensaje de método obsoleto (***deprecated***) al crear un objeto ***PrintStream***. Al señalar como obsoleto el constructor de esta clase se pretendía animar al uso de ***PrintWriter***, pero existen casos en los cuales es imprescindible un elemento ***PrintStream***. Afortunadamente, ***Java 1.2*** ha reconsiderado esta decisión y de nuevo se puede utilizar sin problemas.

9.2.1 Salida de texto y variables por pantalla

Para imprimir en la pantalla se utilizan los métodos ***System.out.print()*** y ***System.out.println()***. Son los primeros métodos que aprende cualquier programador. Sus características fundamentales son:

1. Pueden imprimir valores escritos directamente en el código o cualquier tipo de variable primitiva de ***Java***.

   ```
   System.out.println("Hola, Mundo!");
   System.out.println(57);
   double numeroPI = 3.141592654;
   System.out.println(numeroPI);
   String hola = new String("Hola");
   System.out.println(hola);
   ```

2. Se pueden imprimir varias variables en una llamada al método correspondiente utilizando el operador + de concatenación, que equivale a convertir a ***String*** todas las variables que no lo sean y concatenar las cadenas de caracteres (el primer argumento debe ser un ***String***).

   ```
   System.out.println("Hola, Mundo! " + numeroPI);
   ```

Se debe recordar que los objetos ***System.out*** y ***System.err*** son de la clase ***PrintStream*** y aunque imprimen las variables de un modo legible, no permiten dar a la salida un formato a medida. El programador no puede especificar un formato distinto al disponible por defecto.

9.2.2 Lectura desde teclado

Para leer desde teclado se puede utilizar el método ***System.in.read()*** de la clase ***InputStream***. Este método lee un carácter por cada llamada. Su valor de retorno es un ***int***. Si se espera cualquier otro tipo hay que hacer una conversión explícita mediante un ***cast***.

```
char c;
c=(char)System.in.read();
```

Este método puede lanzar la excepción ***java.io.IOException*** y siempre habrá que ocuparse de ella, por ejemplo en la forma:

```
try {
    c=(char)System.in.read();
}
catch(java.io.IOException ioex) {
    // qué hacer cuando ocurra la excepción
}
```

Para leer datos más largos que un simple carácter es necesario emplear un bucle ***while*** o ***for*** y unir los caracteres. Por ejemplo, para leer una línea completa se podría utilizar un bucle ***while*** guardando los caracteres leídos en un ***String*** o en un ***StringBuffer*** (más rápido que ***String***):

```
char c;
String frase = new String("");          // StringBuffer frase=new StringBuffer("");
try {
    while((c=System.in.read()) != '\n')
    frase = frase + c;                  // frase.append(c);
}
catch(java.io.IOException ioex) {}
```

Una vez que se lee una línea, ésta puede contener números de coma flotante, etc. Sin embargo, hay una manera más fácil de conseguir lo mismo: utilizar adecuadamente la librería ***java.io***.

9.2.3 Método práctico para leer desde teclado

Para facilitar la lectura de teclado se puede conseguir que se lea una línea entera con una sola orden si se utiliza un objeto ***BufferedReader***. El método ***String readLine()*** perteneciente a ***BufferReader*** lee todos los caracteres hasta encontrar un '\n' o '\r' y los devuelve como un ***String*** (sin incluir '\n' ni '\r'). Este método también puede lanzar ***java.io.IOException***.

System.in es un objeto de la clase ***InputStream***. ***BufferedReader*** pide un ***Reader*** en el constructor. El puente de unión necesario lo dará ***InputStreamReader***, que acepta un ***InputStream*** como argumento del constructor y es una clase derivada de ***Reader***. Por lo tanto si se desea leer una línea completa desde la entrada estándar habrá que utilizar el siguiente código:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
// o en una línea:
// BufferedReader br2 = new BufferedReader(new InputStreamReader(System.in));

String frase = br2.readLine();    // Se lee la línea con una llamada
```

Así ya se ha leido una línea del teclado. El thread que ejecute este código estará parado en esta línea hasta que el usuario termine la línea (pulse ***return***). Es más sencillo y práctico que la posibilidad anterior.

¿Y qué hacer con una línea entera? La clase ***java.util.StringTokenizer*** da la posibilidad de separar una cadena de carácteres en las “palabras” (***tokens***) que la forman (por defecto, conjuntos de caracteres separados por un espacio, '\t', '\r', o por '\n'). Cuando sea preciso se pueden convertir las “palabras” en números.

La Tabla 9.5 muestra los métodos más prácticos de la clase ***StringTokenizer***.

Métodos	Función que realizan
StringTokenizer(String)	Constructor a partir de la cadena que hay que separar
boolean hasMoreTokens()	¿Hay más palabras disponibles en la cadena?
String nextToken()	Devuelve el siguiente token de la cadena
int countTokens()	Devuelve el número de tokens que se pueden extraer de la frase

Tabla 9.5. Métodos de StringTokenizer.

La clase ***StreamTokenizer*** de ***java.io*** aporta posibilidades más avanzadas que ***StringTokenizer***, pero también es más compleja. Directamente separa en ***tokens*** lo que entra por un ***InputStream*** o ***Reader***.

Se recuerda que la manera de convertir un ***String*** del tipo "3.141592654" en el valor ***double*** correspondiente es crear un objeto ***Double*** a partir de él y luego extraer su valor ***double***:

```
double pi = (Double.valueOf("3.141592654")).doubleValue();
```

El uso de estas clases facilita el acceso desde teclado, resultando un código más fácil de escribir y de leer. Además tiene la ventaja de que se puede generalizar a la lectura de archivos.

9.3 LECTURA Y ESCRITURA DE ARCHIVOS

Aunque el manejo de archivos tiene características especiales, se puede utilizar lo dicho hasta ahora para las entradas y salidas estándar con pequeñas variaciones. ***Java*** ofrece las siguientes posibilidades:

Existen las clases ***FileInputStream*** y ***FileOutputStream*** (extendiendo ***InputStream*** y ***OutputStream***) que permiten leer y escribir ***bytes*** en archivos. Para archivos de texto son preferibles ***FileReader*** (desciende de ***Reader***) y ***FileWriter*** (desciende de ***Writer***), que realizan las mismas funciones. Se puede construir un objeto de cualquiera de estas cuatro clases a partir de un ***String*** que contenga el nombre o la dirección en disco del archivo o con un objeto de la clase ***File*** que representa dicho archivo. Por ejemplo el código

```
FileReader fr1 = new FileReader("archivo.txt");
```

es equivalente a:

```
File f = new File("archivo.txt");
FileReader fr2 = new FileReader(f);
```

Si no encuentran el archivo indicado, los constructores de ***FileReader*** y ***FileInputStream*** pueden lanzar la excepción ***java.io.FileNotFoundException***.

Los constructores de ***FileWriter*** y ***FileOutputStream*** pueden lanzar ***java.io.IOException***. Si no encuentran el archivo indicado, lo crean nuevo. Por defecto, estas dos clases comienzan a escribir al comienzo del archivo. Para escribir detrás de lo que ya existe en el archivo ("append"), se utiliza un segundo argumento de tipo ***boolean*** con valor ***true***:

```
FileWriter fw = new FileWriter("archivo.txt", true);
```

Las clases que se explican a continuación permiten un manejo más fácil y eficiente que las vistas hasta ahora.

9.3.1 Clases File y FileDialog

Un objeto de la clase ***File*** puede representar un ***archivo*** o un ***directorio***. Tiene los siguientes constructores:

```
File(String name)
File(String dir, String name)
File(File dir, String name).
```

Se puede dar el nombre de un archivo, el nombre y el directorio, o sólo el directorio, como ***path*** absoluto y como ***path*** relativo al directorio actual. Para saber si el archivo existe se puede llamar al método ***boolean exists().***

```
File f1 = new File("c:\\windows\\notepad.exe");  // La barra '\' se escribe '\\'
File f2 = new File("c:\\windows");               // Un directorio
File f3 = new File(f2, "notepad.exe");           // Es igual a f1
```

Si ***File*** representa un archivo que existe los métodos de la Tabla 9.6 dan información de él.

Métodos	**Función que realizan**
boolean isFile()	true si el archivo existe
long length()	tamaño del archivo en bytes
long lastModified()	fecha de la última modificación
boolean canRead()	true si se puede leer
boolean canWrite()	true si se puede escribir
delete()	borrar el archivo
RenameTo(File)	cambiar el nombre

Tabla 9.6. Métodos de File para archivos.

Si representa un directorio se pueden utilizar los de la Tabla 9.7:

Métodos	**Función que realizan**
boolean isDirectory()	true si existe el directorio
mkdir()	crear el directorio
delete()	borrar el directorio
String[] list()	devuelve los archivos que se encuentran en el directorio

Tabla 9.7. Métodos de File para directorios.

Por último, otros métodos incluidos en la Tabla 9.8 devuelven el ***path*** del archivo de distintas maneras.

Métodos	**Función que realizan**
String getPath()	Devuelve el path que contiene el objeto File
String getName()	Devuelve el nombre del archivo
String getAbsolutePath()	Devuelve el path absoluto (juntando el relativo al actual)
String getParent()	Devuelve el directorio padre

Tabla 9.8. Métodos de File que devuelven el path.

Una forma típica de preguntar por un archivo es presentar un caja de diálogo. La clase ***java.awt.FileDialog*** presenta el diálogo típico de cada sistema operativo para guardar o abrir ficheros. Sus constructores son:

```
FileDialog(Frame fr)
FileDialog(Frame fr, String title)
FileDialog(Frame fr, String title, int type)
```

donde ***type*** puede ser ***FileDialog.LOAD*** o ***FileDialog.SAVE*** según la operación que se desee realizar.

Es muy fácil conectar este diálogo con un ***File***, utilizando los métodos ***String getFile()*** y ***String getDirectory()***. Por ejemplo:

```
FileDialog fd = new FileDialog(f, "Elija un archivo");
fd.show();
File f = new File(fd.getDirectory(), fd.getFile());
```

9.3.2 Lectura de archivos de texto

Se puede crear un objeto ***BufferedReader*** para leer de un archivo de texto de la siguiente manera:

```
BufferedReader br = new BufferedReader(new FileReader("archivo.txt"));
```

Utilizando el objeto de tipo ***BufferedReader*** se puede conseguir exactamente lo mismo que en las secciones anteriores utilizando el método ***readLine()*** y la clase ***StringTokenizer***. En el caso de archivos es muy importante utilizar el ***buffer*** puesto que la tarea de escribir en disco es muy lenta respecto a los procesos del programa y realizar las operaciones de lectura de golpe y no de una en una hace mucho más eficiente el acceso. Por ejemplo:

```
// Lee un archivo entero de la misma manera que de teclado
String texto = new String();
try {
    FileReader fr = new FileReader("archivo.txt");
    entrada = new BufferedReader(fr);
    String s;
    while((s = entrada.readLine()) != null)
        texto += s;
    entrada.close();
}
catch(java.io.FileNotFoundException fnfex) {
    System.out.println("Archivo no encontrado: " + fnfex);}
catch(java.io.IOException ioex) {}
```

9.3.3 Escritura de archivos de texto

La clase ***PrintWriter*** es la más práctica para escribir un archivo de texto porque posee los métodos ***print***(cualquier tipo) y ***println***(cualquier tipo), idénticos a los de ***System.out*** (de clase ***PrintStream***).

Un objeto ***PrintWriter*** se puede crear a partir de un ***BufferedWriter*** (para disponer de ***buffer***), que se crea a partir del ***FileWriter*** al que se la pasa el nombre del archivo. Después, escribir en el archivo es tan fácil como en pantalla. El siguiente ejemplo ilustra lo anterior:

```
try {
   FileWriter fw = new FileWriter("escribeme.txt");
   BufferedWriter bw = new BufferedWriter(fw);
   PrintWriter salida = new PrintWriter(bw);
   salida.println("Hola, soy la primera línea");
   salida.close();
   // Modo append
   bw = new BufferedWriter(new FileWriter("escribeme.txt", true));
   salida = new PrintWriter(bw);
   salida.print("Y yo soy la segunda. ");
   double b = 123.45;
   salida.println(b);
   salida.close();
}
cacth(java.io.IOException ioex) { }
```

9.3.4 Archivos que no son de texto

DataInputStream y ***DataOutputStream*** son clases de ***Java 1.0*** que no han sido alteradas hasta ahora. Para leer y escribir datos primitivos directamente (sin convertir a/de ***String***) siguen siendo más útiles estas dos clases.

Son clases diseñadas para trabajar de manera conjunta. Una puede leer lo que la otra escribe, que en sí no es algo legible, sino el dato como una secuencia de ***bytes***. Por ello se utilizan para

almacenar datos de manera independiente de la plataforma (o para mandarlos por una red entre ordenadores muy distintos).

El problema es que obligan a utilizar clases que descienden de ***InputStream*** y ***OutputStream*** y por lo tanto algo más complicadas de utilizar. El siguiente código primero escribe en el fichero *prueba.dat* para después leer los datos escritos:

```
// Escritura de una variable double
DataOutputStream dos = new DataOutputStream(
   new BufferedOutputStream(
      new FileOutputStream("prueba.dat")));
double d1 = 17/7;
dos.writeDouble(d);
dos.close();
// Lectura de la variable double
DataInputStream dis = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream("prueba.dat")));
double d2 = dis.readDouble();
```

9.4 SERIALIZACIÓN

La ***serialización*** es un proceso por el que un objeto cualquiera se puede convertir en una ***secuencia de bytes*** con la que más tarde se podrá reconstruir dicho objeto manteniendo el valor de sus variables. Esto permite guardar un objeto en un archivo o mandarlo por la red.

Para que una clase pueda utilizar la serialización, debe implementar la interface ***Serializable***, que no define ningún método. Casi todas las clases estándar de ***Java*** son serializables. La clase MiClase se podría serializar declarándola como:

```
public class MiClase implements Serializable { }
```

Para escribir y leer objetos se utilizan las clases ***ObjectInputStream*** y ***ObjectOutputStream***, que cuentan con los métodos ***writeObject()*** y ***readObject()***. Por ejemplo:

```
ObjectOutputStream objout = new ObjectOutputStream(
                                 new FileOutputStream("archivo.x"));
String s = new String("Me van a serializar");
objout.writeObject(s);
ObjectInputStream objin = new ObjectInputStream(new FileInputStream("archivo.x"));
String s2 = (String)objin.readObject();
```

Es importante tener en cuenta que ***readObject()*** devuelve un ***Object*** sobre el que se deberá hacer un ***casting*** para que el objeto sea útil. La reconstrucción necesita que el archivo ****.class*** esté al alcance del programa (como mínimo para hacer este ***casting***).

Al serializar un objeto, automáticamente se serializan todas sus variables y objetos miembro. A su vez se serializan los que estos objetos miembro puedan tener (todos deben ser serializables). También se reconstruyen de igual manera. Si se serializa un ***Vector*** que contiene varios ***Strings***, todo ello se convierte en una serie de ***bytes***. Al recuperarlo la reconstrucción deja todo en el lugar en que se guardó.

Si dos objetos contienen una referencia a otro, éste no se duplica si se escriben o leen ambos del mismo ***stream***. Es decir, si el mismo ***String*** estuviera contenido dos veces en el ***Vector***, sólo se guardaría una vez y al recuperarlo sólo se crearía un objeto con dos referencias contenidas en el vector.

9.4.1 Control de la serialización

Aunque lo mejor de la serialización es que su comportamiento automático es bueno y sencillo, existe la posibilidad de especificar cómo se deben hacer las cosas.

La palabra clave ***transient*** permite indicar que un objeto o varible miembro no sea serializado con el resto del objeto. Al recuperarlo, lo que esté marcado como ***transient*** será ***0***, ***null*** o ***false*** (en esta operación no se llama a ningún constructor) hasta que se le dé un nuevo valor. Podría ser el caso de un ***password*** que no se quiere guardar por seguridad.

Las variables y objetos ***static*** no son serializados. Si se quieren incluir hay que escribir el código que lo haga. Por ejemplo, habrá que programar un método que serialice los objetos estáticos al que se llamará después de serializar el resto de los elementos. También habría que recuperarlos explícitamente después de recuperar el resto de los objetos.

Las clases que implementan ***Serializable*** pueden definir dos métodos con los que controlar la serialización. No están obligadas a hacerlo porque una clase sin estos métodos obtiene directamente el comportamiento por defecto. Si los define serán los que se utilicen al serializar:

```
private void writeObject(ObjectOutputStream stream) throws IOException
private void readObject(ObjectInputStream stream) throws IOException
```

El primero permite indicar qué se escribe o añadir otras instrucciones al comportamiento por defecto. El segundo debe poder leer lo que escribe ***writeObject()***. Puede usarse por ejemplo para poner al día las variables que lo necesiten al ser recuperado un objeto. Hay que leer en el mismo orden en que se escribieron los objetos.

Se puede obtener el comportamiento por defecto dentro de estos métodos llamando a ***stream.defaultWriteObject()*** y ***stream.defaultReadObject()***.

Para guardar explícitamente los tipos primitivos se puede utilizar los métodos que proporcionan ***ObjectInputStream*** y ***ObjectOutputStream***, idénticos a los de ***DataInputStream*** y ***DataOutputStream*** (***writeInt()***, ***readDouble()***, ...) o guardar objetos de sus clases equivalentes (***Integer***, ***Double***...).

Por ejemplo, si en una clase llamada ***Tierra*** se necesita que al serializar un objeto siempre le acompañe la constante ***g*** (9,8) definida como ***static*** el código podría ser:

```
static double g = 9.8;

private void writeObject(ObjectOutputStream stream) throws IOException {
      stream.defaultWriteObject();
      stream.writeDouble(g);
}

private void readObject(ObjectInputStream stream) throws IOException {
      stream.defaultReadObject();
      g = stream.readDouble(g);
}
```

9.4.2 Externalizable

La interface ***Externalizable*** extiende ***Serializable***. Tiene el mismo objetivo que ésta, pero no tiene ningún comportamiento automático, todo se deja en manos del programador.

Externalizable tiene dos métodos que deben implementarse.

```
interface Externalizable {
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException,
                                                    ClassNotFoundException;
}
```

Al transformar un objeto, el método ***writeExternal()*** es responsable de todo lo que se hace. Sólo se guardará lo que dentro de éste método se indique.

El método ***readExternal()*** debe ser capaz de recuperar lo guardado por ***writeExternal()***. La lectura debe ser en el mismo orden que la escritura. Es importante saber que antes de llamar a este método se llama al constructor por defecto de la clase.

Como se ve el comportamiento de ***Externalizable*** es muy similar al de ***Serializable***.

9.5 LECTURA DE UN ARCHIVO EN UN SERVIDOR DE INTERNET

Teniendo la dirección de Internet de un archivo, la librería de ***Java*** permite leer este archivo utilizando un ***stream***. Es una aplicación muy sencilla que muestra la polivalencia del concepto de ***stream***.

En el package ***java.net*** existe la clase ***URL***, que representa una dirección de Internet. Esta clase tiene el método ***InputStream openStream(URL dir)*** que abre un ***stream*** con origen en la dirección de Internet.

A partir de ahí, se trata como cualquier elemento ***InputStream***. Por ejemplo:

```
//Lectura del archivo (texto HTML)
URL direccion = new URL("http://www1.ceit.es/subdir/MiPagina.htm");
String s = new String();
String html = new String();
try {
     BufferedReader br = new BufferedReader(
                              new InputStreamReader(
                                  direccion.openStream()));
     while((s = br.readLine()) != null)
          html += s + '\n';
     br.close();
}
catch(Exception e) {
     System.err.println(e);
}
```

10. OTRAS CAPACIDADES DE JAVA

A lo largo de este manual se han presentado algunos de los fundamentos del lenguaje ***Java***. Debido a que ***Java*** engloba en el propio lenguaje muchos de los conceptos de la informática moderna la inclusión de todos ellos sobrepasaría ampliamente el carácter introductorio de este manual.

No se puede sin embargo finalizar esta introducción al lenguaje ***Java*** sin una breve descripción de algunas de las capacidades más interesantes del lenguaje.

10.1 JAVA FOUNDATION CLASSES (JFC) Y JAVA 2D

Las ***JFC***, ***Java™ Foundation Classes*** son un conjunto de componentes y características para ayudar a construir los entornos gráficos de los programas o GUIs (*Graphical User Interfaces*). Incluye prácticamente todo tipo de elementos gráficos como botones, paneles, menús y ventanas, con muchas ventajas sobre el AWT.

Swing es una parte de las ***JFC*** que permite incorporar en las aplicaciones elementos gráficos de una forma mucho más versátil y con más capacidades que utilizando el **AWT** básico de ***Java***. Algunas de las características más interesantes son:

1. Cualquier programa que utiliza componentes de ***Swing*** puede elegir el aspecto que desea para sus ventanas y elementos gráficos: entorno ***Windows 95/98/NT***, entorno ***Motif*** (asociado a sistemas UNIX) o ***Metal*** (aspecto propio de ***Java***, común a todas las plataformas).
2. Cualquier componente gráfico de ***Swing*** presenta más propiedades que el correspondiente elemento del ***AWT***: Los botones pueden incorporan imágenes, hay nuevos ***layouts*** y paneles, menús, …
3. Posibilidad de ***Drag & Drop***, es decir de seleccionar componentes con el ratón y arrastrar a otro lugar de la pantalla.

En la versión ***JDK 1.2*** se incorpora como parte de las JFC el llamado ***Java 2D***, que permite a los desarrolladores incorporar texto, imágenes y gráficos en dos dimensiones de gran calidad. Además da soporte para poder imprimir documentos complejos.

A partir de la versión ***1.2*** de ***Java*** las ***JFC*** forman parte del propio ***JDK***. Si se desea utilizar desde la versión ***1.1*** es necesario instalar las ***JFC*** de forma independiente.

10.2 JAVA MEDIA FRAMEWORK (JMF)

El API ***JMF*** (***Java Media FrameWork***) especifica una arquitectura, un protocolo de transmisión de datos y unos elementos gráficos simples y unificados para la reproducción de contenidos ***multimedia***, esto es vídeo, audio y animaciones, principalmente.

Los distintos JDK aparecidos hasta la publicación de este manual no incorporan este API de JMF. Es necesario instalar un software que complementa el JDK.

10.3 JAVA 3D

El API de ***Java 3D™*** es un conjunto de clases para crear aplicaciones y ***applets*** con elementos ***3D***. Ofrece a los desarrolladores la posibilidad de manipular geometrías complejas en tres dimensiones.

La principal ventaja que presenta este API 3D frente a otros entornos de programación 3D es que permite crear aplicaciones gráficas 3D independientes del tipo de sistema.

Java 3D es un conjunto de clases, interfaces y librerías de alto nivel que permiten aprovechar la aceleración gráfica por hardware que incorporan muchas tarjetas gráficas, ya que las llamadas a los métodos de ***Java 3D*** son transformadas en llamadas a funciones de ***OpenGL*** o ***Direct3D***.

Aunque tanto conceptualmente como oficialmente ***Java 3D*** forma parte del API ***JMF***, se trata de unas librerías que se instalan independientemente del ***JMF***.

10.4 JAVABEANS

El API de ***JavaBeans*** hace posible escribir "componentes de software" en el lenguaje ***Java***. Los ***componentes*** son elementos reutilizables que pueden incorporarse gráficamente a otros componentes como ***applets*** y aplicaciones utilizando herramientas gráficas de desarrollo.

Cada componente ofrece sus características concretas (por ejemplo sus métodos públicos y sus eventos) a los entornos gráficos de desarrollo permitiendo su manipulación visual. Son análogos a otros componentes de algunos entornos visuales, como por ejemplo los controles de Visual Basic.

El ***BDK*** (***Beans Developer Kit***) es un conjunto de herramientas para desarrollar ***JavaBeans***. Se trata de un kit no incorporado en los distintos ***JDK*** de ***Java***.

10.5 JAVA EN LA RED

A diferencia de otros lenguajes de programación, ***Java*** presenta de una forma estándar para todas las plataformas y sistemas operativos, un conjunto de clases que permiten la comunicación entre aplicaciones que se ejecutan en distintos ordenadores.

El package ***java.net*** del API de ***Java*** incluye las clases necesarias para establecer conexiones, crear servidores, enviar y recibir datos, y para el resto de operaciones utilizadas en las comunicaciones a través de redes de ordenadores. Existen además otros APIs independientes preparados especialmente para realizar unas determinadas tareas, como son ***Servlets, RMI*** y ***Java IDL***. Muchos de estos APIs utilizan internamente las clases presentes en ***java.net***.

10.6 JAVA EN EL SERVIDOR: SERVLETS

Los ***Servlets*** son módulos que permiten sustituir o utilizar el lenguaje ***Java*** en lugar de los programas ***CGI*** escritos en otros lenguajes como C/C++ o ***Perl***. Los programas ***CGI*** son aplicaciones que se ejecutan en un ***servidor Web*** en respuesta a una acción de un browser remoto (petición de una página HTML, envío de los datos de un formulario, etc.). Permiten generar páginas HTML dinámicas, esto es, páginas HTML cuyo contenido puede variar y que por lo tanto no pueden almacenarse en un fichero en el servidor.

Los ***Servlets*** no tienen entorno gráfico ya que se ejecutan en el servidor. Reciben unos datos y su salida o respuesta son principalmente ficheros de texto HTML.

Los ***servlets*** son desarrollados utilizando el API ***Java Servlet***, que es una extensión de ***Java*** que hasta la fecha no forma parte de ninguno de los JDK. Es necesario instalar el software específico de ***Java Servlet***.

10.7 RMI Y JAVA IDL

Tanto ***RMI*** (***Remote Method Invocation***) como ***Java IDL*** (***Java Interface Definition Language***) son herramientas para desarrollar ***aplicaciones distribuidas***. Estas aplicaciones presentan la característica de que una aplicación puede ejecutar funciones y métodos en varios ordenadores distintos. Utilizando una referencia a un objeto que se encuentra en un ordenador remoto, es posible ejecutar métodos de ese objeto desde una aplicación que se ejecuta en un ordenador distinto. ***RMI y Java IDL*** proporcionan los mecanismos mediante los cuales los distintos objetos distribuidos se comunican y se transmiten la información. Son por lo tanto tecnologías que permiten la creación y uso de ***objetos distribuidos***, esto es, objetos o programas que interactúan en diferentes plataformas y ordenadores a través de una red.

RMI es una solución basada íntegramente en ***Java***. Incorpora métodos para localizar los objetos remotos, comunicarse con ellos e incluso enviar un objeto de ***Java***, "por valor", de un objeto distribuido a otro.

Java IDL permite la conectividad entre objetos distribuidos utilizando ***CORBA (Common Object Request Broker Architecture***). ***CORBA*** es un estándar para la interconexión entre objetos distribuidos. Existen implementaciones de ***CORBA*** en varios lenguajes, lo que posibilita comunicar objetos realizados en distintos lenguajes como ***Java***, ***C/C++***, ***COBOL***, …

Tanto ***RMI*** como ***Java IDL*** están incluidos en el ***JDK 1.2*** de ***Sun***. En el caso de ***Java IDL*** se precisa de una utilidad adicional (llamada ***idltojava***) que genera el código necesario para comunicarse con cualquier implementación ***CORBA***.

10.8 SEGURIDAD EN JAVA

El espacio natural de trabajo de la Informática moderna y por lo tanto del lenguaje ***Java*** son las redes de ordenadores y en especial ***Internet***. En una red como ***Internet***, utilizada por millones de usuarios, la seguridad adquiere una importancia vital.

Desde su aparición ***Java*** ha ido incorporando elementos destinados a proporcionar un mayor control sobre la seguridad de los datos y programas enviados a través de una red. Los distintos ***JDK*** incorporan herramientas para añadir seguridad a las aplicaciones ***Java***: *firmas digitales*, *transmisión segura de datos*, ... ***Java*** permite establecer distintos niveles de seguridad, lo que posibilita una gran flexibilidad para asignar o denegar permisos. Existe abundante información sobre estas herramientas que el lector deberá consultar en caso necesario.

10.9 ACCESO A BASES DE DATOS (JDBC)

JDBC (***Java DataBase Connectivity***) es el estándar de ***Java*** para conectarse con bases de datos. Se estima que aproximadamente la mitad del software que se crea en la actualidad incorpora operaciones de lectura y escritura con bases de datos. ***JDBC*** está diseñado para ser independiente de la plataforma e incluso de la base de datos sobra la que se desee actuar.

Para conseguir esta independencia, ***JDBC*** ofrece un sistema estándar de interconexión con las bases de datos, muy similar al ***SQL*** (***Structured Query Language***). Los distintos vendedores de bases de datos crean los elementos necesarios que actúan como puente entre ***JDBC*** y la propia base de datos.

La versión ***JDBC 1.0*** forma parte del ***JDK 1.1***. Después de distintas revisiones, actualmente ha aparecido la versión ***JDBC 2.0***, incluida en el ***JDK 1.2***.

10.10 JAVA NATIVE INTERFACE (JNI)

JNI (***Java Native Interface***) es el interface de programación de ***Java*** para ejecutar código nativo, es decir código compilado al lenguaje binario propio de una plataforma o sistema de ordenador. Se incluye en el ***JDK*** las herramientas necesarias para su utilización.

JNI permite al código de ***Java*** que se ejecuta dentro de la ***JVM*** interactuar con aplicaciones y librerías escritas en otros lenguajes, como ***C/C++*** o incluso lenguaje ***ensamblador***. Incorpora a su vez las herramientas para ejecutar código ***Java*** desde aplicaciones desarrolladas en otros lenguajes. El entorno ***JNI*** ofrece por lo tanto a los métodos nativos utilizar objetos de ***Java*** de igual forma que el código ***Java*** puede utilizar estos objetos nativos. Tanto la parte de ***Java*** como la parte nativa de una aplicación pueden crear, actualizar y acceder a los objetos programados en ***Java*** y compartir dichos objetos.