Acceso a datos

Consulte nuestra página web: **www.sintesis.com**
En ella encontrará el catálogo completo y comentado

Queda prohibida, salvo excepción prevista en la ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de la propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (arts. 270 y sigs. Código Penal). El Centro Español de Derechos Reprográficos (www.cedro.org) vela por el respeto de los citados derechos.

Acceso a datos

Carlos Alberto Cortijo Bon

EDITORIAL
SÍNTESIS

Asesor editorial:
Juan Carlos Moreno Pérez

© Carlos Alberto Cortijo Bon

© EDITORIAL SÍNTESIS, S. A.
Vallehermoso, 34. 28015 Madrid
Teléfono 91 593 20 98
http://www.sintesis.com

ISBN: 978-84-9171-356-2
Depósito Legal: M-15.176-2019

Impreso en España - Printed in Spain

Reservados todos los derechos. Está prohibido, bajo las sanciones
penales y el resarcimiento civil previstos en las leyes, reproducir,
registrar o transmitir esta publicación, íntegra o parcialmente,
por cualquier sistema de recuperación y por cualquier medio,
sea mecánico, electrónico, magnético, electroóptico, por fotocopia
o por cualquier otro, sin la autorización previa por escrito
de Editorial Síntesis, S. A.

Índice

PRESENTACIÓN 11

1. INTRODUCCIÓN Y CONCEPTOS BÁSICOS 13

Objetivos 13
Mapa conceptual 14
Glosario 14
1.1. Programas y datos 15
1.2. Persistencia de datos 15
1.3. Sistemas de persistencia de datos 16
1.4. Almacenamiento de datos 17
1.4.1. Ficheros 17
1.4.2. Bases de datos relacionales 18
1.4.3. Documentos de XML 18
1.4.4. Bases de datos de objetos 19
1.4.5. Bases de datos NoSQL 20
1.5. Restricciones de integridad 20
1.6. Acceso a los datos con iteradores 20
1.7. Control de accesos concurrentes y transacciones 20
1.8. Persistencia de datos en ficheros 22
1.9. Persistencia de datos en bases de datos relacionales 22
1.9.1. Persistencia de datos de XML en bases de datos relacionales 23
1.9.2. Persistencia de objetos en bases de datos relacionales 23
1.10. Persistencia de datos en bases de datos de objetos 24
1.11. Persistencia de datos en bases de datos de XML nativas 25
1.12. Persistencia de datos en bases de datos NoSQL 26
Resumen 27
Actividades de autoevaluación 29

2. FICHEROS ... 31

Objetivos ... 31
Mapa conceptual ... 32
Glosario ... 32
2.1. Persistencia de datos en ficheros ... 33
2.2. Tipos de ficheros según su contenido ... 34
2.3. Codificaciones para texto ... 34
2.4. La clase File de Java ... 35
2.5. Gestión de excepciones en Java ... 38
2.5.1. Captura y gestión de excepciones ... 38
2.5.2. Gestión diferenciada de distintos tipos de excepciones ... 40
2.5.3. Declaración de excepciones lanzadas por un método de clase ... 42
2.5.4. Excepciones, inicialización y liberación de recursos: bloque `finally` y `try` con recursos ... 43
2.6. Formas de acceso a los ficheros ... 44
2.7. Operaciones sobre ficheros con Java ... 45
2.7.1. Operaciones de lectura ... 45
2.7.2. Operaciones de escritura ... 46
2.8. Acceso secuencial a ficheros en Java ... 46
2.8.1. Clases relacionadas con flujos de datos ... 46
2.8.2. Clases para recodificación ... 49
2.8.3. Clases para *buffering* ... 50
2.8.4. Operaciones de lectura para flujos de entrada ... 51
2.8.5. Operaciones de escritura para flujos de salida ... 54
2.9. Operaciones con ficheros de acceso aleatorio en Java ... 57
2.10. Organizaciones de ficheros ... 61
2.10.1. Organización secuencial ... 61
2.10.2. Organización secuencial indexada ... 62
Resumen ... 63
Ejercicios propuestos ... 64
Actividades de autoevaluación ... 65

3. BASES DE DATOS RELACIONALES ... 69

Objetivos ... 69
Mapa conceptual ... 70
Glosario ... 70
3.1. Conectores ... 71
3.2. Conectores para bases de datos relacionales ... 71
3.3. Acceso a resultados de consultas sobre bases de datos relacionales mediante conectores ... 73
3.4. Desfase objeto-relacional ... 74
3.5. Java Database Connectivity ... 74
3.6. Operaciones básicas con JDBC ... 75
3.6.1. Apertura y cierre de conexiones ... 75
3.6.2. La interfaz Statement ... 77
3.6.3. Ejecución de sentencias de DDL ... 78
3.6.4. Ejecución de sentencias para modificar contenidos de la base de datos ... 79
3.6.5. Ejecución de consultas y manejo de `ResultSet` ... 80
3.7. Sentencias preparadas ... 83

3.8. Transacciones 85
3.9. Valores de claves autogeneradas 88
3.10. Llamadas a procedimientos y funciones almacenados 91
3.11. Actualizaciones sobre los resultados de una consulta 94
3.12. Ejecución de *scripts* 96
3.13. Ejecución de sentencias por lotes 96
Resumen 98
Ejercicios propuestos 99
Actividades de autoevaluación 102

4. CORRESPONDENCIA OBJETO-RELACIONAL 105

Objetivos 105
Mapa conceptual 106
Glosario 106
4.1. Correspondencia objeto-relacional 107
4.2. Hibernate 108
4.3. Iniciación a la correspondencia objeto-relacional con Hibernate 109
4.4. Correspondencia objeto-relacional a partir de las tablas 111
4.4.1. Creación de la conexión con la base de datos 113
4.4.2. Creación del proyecto 114
4.4.3. Fichero de configuración de Hibernate: `hibernate.cfg.xml` 115
4.4.4. Fichero de ingeniería inversa `hibernate.reveng.xml` 116
4.4.5. POJO (clases) y ficheros de correspondencia 117
4.4.6. HibernateUtil.java 118
4.5. Descarga y uso de una versión reciente de Hibernate 118
4.6. Programa de ejemplo para persistencia de objetos con Hibernate 119
4.7. Ficheros hbm o de correspondencia de Hibernate 121
4.7.1. Correspondencia para las clases y atributos de clase 124
4.7.2. Correspondencia para las relaciones 125
4.8. Manejo de relaciones de uno a muchos entre objetos persistentes 130
4.9. Manejo de relaciones de uno a uno entre objetos persistentes 132
4.10. Sesiones y estados de los objetos persistentes 133
4.10.1. De transitorio a persistente con `save(), saveOrUpdate() y persist()` 135
4.10.2. Obtención de un objeto persistente con `get() y load()` 136
4.10.3. De persistente a eliminado con `delete()` 137
4.10.4. De persistente a separado con `evict(), close() y clear()` 137
4.10.5. De separado a persistente con `update(), saveOrUpdate(), lock() y merge()` 137
4.11. Lenguajes de consulta HQL y JPQL 138
4.11.1. La interfaz `Query` 139
4.12. Correspondencia de la herencia 144
4.12.1. Eliminación de subtipos (una tabla para la jerarquía) 148
4.12.2. Una tabla por subclase (eliminación de la jerarquía) 151
4.13. Consultas con SQL 152
Resumen 153
Ejercicios propuestos 154
Actividades de autoevaluación 158

5. BASES DE DATOS DE OBJETOS Y OBJETO-RELACIONALES 161

Objetivos 161
Mapa conceptual 162
Glosario 162
5.1. Bases de datos de objetos y objeto-relacionales, y correspondencia objeto-relacional 163
5.2. Características de las bases de datos de objetos 163
5.3. El estándar ODMG 164
5.4. ODL 165
5.4.1. Modelo de objetos de ODL 165
5.4.2. Clases e interfaces 166
5.4.3. Relaciones 167
5.5. OQL 168
5.6. Consulta y manipulación de datos con el Java binding 170
5.7. La base de datos de objetos Matisse 170
5.7.1. Creación de una base de datos mediante ODL con Matisse 171
5.7.2. Utilización de la base de datos mediante el Java binding de Matisse 174
5.7.3. Consultas mediante el SQL de Matisse 181
5.8. SQL:99 y bases de datos objeto-relacionales 183
5.9. Características objeto-relacionales de Oracle 183
5.9.1. Tipos de objetos 184
5.9.2. Herencia 184
5.9.3. Objetos de fila y objetos de columna 185
5.9.4. Tipos de objetos con referencias a otros tipos de objetos 186
5.9.5. Tipos de datos de colección: VARRAY y tablas anidadas 186
Resumen 188
Ejercicios propuestos 188
Actividades de autoevaluación 189

6. XML 193

Objetivos 193
Mapa conceptual 194
Glosario 194
6.1. El lenguaje XML 195
6.2. Estructura de un documento de XML 195
6.3. DOM 197
6.4. *Parsers* o analizadores sintácticos, serialización y deserialización 197
6.5. DOM con Java 198
6.5.1. Parsing DOM 200
6.5.2. Creación de documentos con DOM 203
6.5.3. Serialización de documentos DOM 204
6.6. SAX 206
6.7. Validación de documentos de XML 209
6.7.1. Validación con DTD 209
6.7.2. Validación con esquemas de XML 210
6.8. *Parsing* con validaciones con Java 211
6.9. *Binding* con JAXB 217
6.9.1. Esquemas de XML para *binding* 218
6.9.2. Compilador de *binding* 219

6.9.3. Clases generadas por el compilador de *binding* ... 220
6.9.4. *Unmarshalling* (deserialización) con JAXB ... 221
6.9.5. *Marshalling* (serialización) con JAXB ... 223
6.10. El lenguaje de consulta XPath ... 225
6.11. El lenguaje XSL ... 228
Resumen ... 231
Ejercicios propuestos ... 232
Actividades de autoevaluación ... 235

7. BASES DE DATOS DE XML ... 237

Objetivos ... 237
Mapa conceptual ... 238
Glosario ... 238
7.1. XML como soporte para almacenamiento e intercambio de datos ... 239
7.2. Alternativas para el almacenamiento de documentos de XML ... 239
7.3. Almacenamiento de XML en SGBD relacionales ... 240
7.4. Características de las bases de datos de XML nativas ... 241
7.5. Gestores comerciales y libres ... 242
7.6. Instalación y configuración del SGBD de XML nativo eXist ... 243
7.7. API para gestión de bases de datos nativas de XML ... 245
7.8. La API XML:DB ... 245
7.8.1. Establecimiento de conexiones y acceso a servicios con XML:DB ... 246
7.8.2. Creación y borrado de colecciones con XML:DB ... 248
7.8.3. Creación y borrado de documentos con XML:DB ... 249
7.9. El lenguaje XQuery ... 251
7.9.1. Consultas con XQuery ... 252
7.9.2. Sentencias de modificación de datos con XQuery ... 255
7.10. La API XQJ ... 256
7.10.1. Establecimiento de conexiones con XQJ ... 257
7.10.2. Consultas con XQJ ... 260
7.10.3. Modificaciones de documentos con XQJ ... 261
7.10.4. Transacciones con XQJ ... 262
Resumen ... 264
Ejercicios propuestos ... 265
Actividades de autoevaluación ... 267

8. COMPONENTES PARA EL ACCESO A DATOS ... 271

Objetivos ... 271
Mapa conceptual ... 272
Glosario ... 272
8.1. Componentes de software ... 273
8.2. Modelos de componentes ... 273
8.3. La plataforma Java: Java SE y Java EE ... 274
8.4. JavaBeans ... 275
8.5. El modelo MVC para desarrollo de aplicaciones web con Java ... 276
8.6. JSP (JavaServer Pages) ... 277
8.6.1. Directivas de JSP ... 278
8.6.2. Scriptlets ... 278

8.6.3. Variables implícitas ... 278
8.6.4. Referencias a variables de Java ... 279
8.7. Servlets ... 279
8.8. Desarrollo de una aplicación web MVC basada en JavaBeans ... 280
8.8.1. Creación de la aplicación web ... 280
8.8.2. Persistencia de objetos con Hibernate ... 281
8.8.3. Creación del *servlet* controlador ... 281
8.8.4. Uso de JavaBeans asociado a formularios HTML con JSP ... 284
8.8.5. Creación de los JSP ... 284
8.9. Enterprise JavaBeans ... 289
8.10. Desarrollo de una aplicación web MVC para Java EE con EJB ... 289
8.10.1. Creación de la aplicación web ... 290
8.10.2. Creación de la unidad de persistencia ... 290
8.10.3. Creación de las clases de entidad ... 291
8.10.4. Creación de los EJB de sesión para las clases de entidad ... 292
8.10.5. Creación del *servlet* controlador ... 295
8.10.6. Configuración básica inicial de la aplicación web ... 297
8.10.7. Creación de los JSP ... 297
8.10.8. Incluir los ficheros jar de una versión reciente de Hibernate ... 301
8.10.9. Despliegue de la aplicación ... 301
8.10.10. Solución de errores adicionales en tiempo de ejecución ... 303
Resumen ... 304
Ejercicios propuestos ... 304
Actividades de autoevaluación ... 306

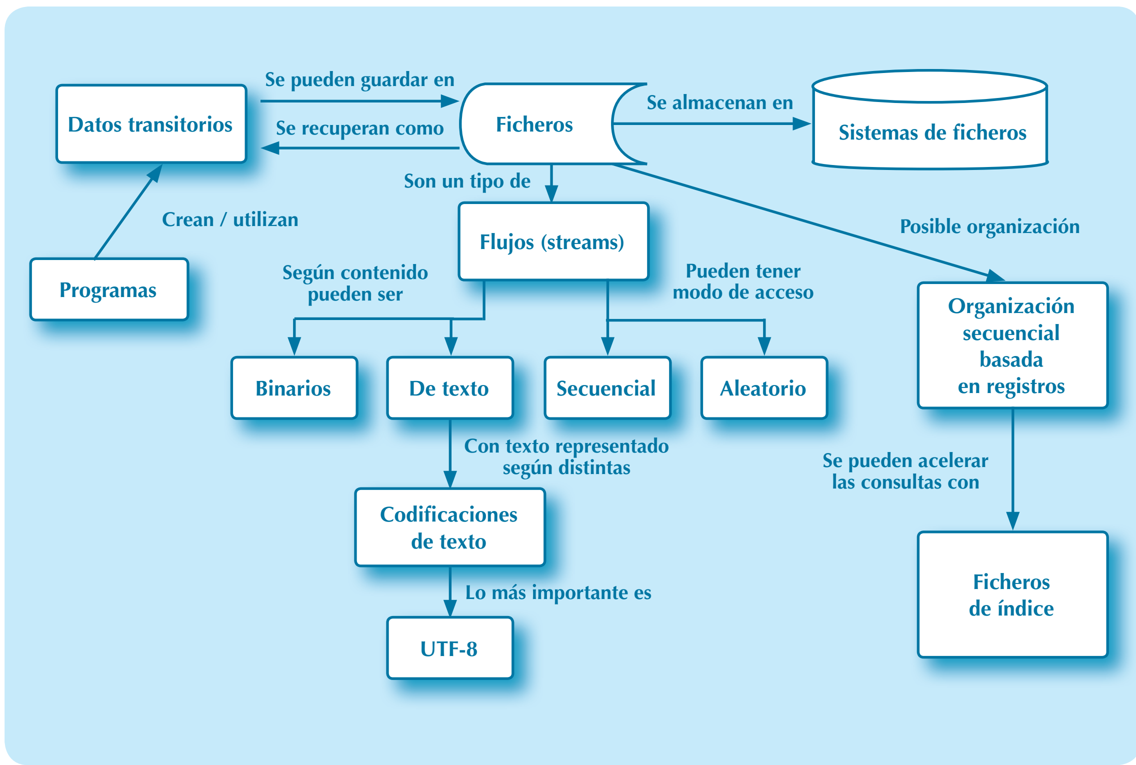
2

Ficheros

Objetivos

- ✓ Conocer las diferencias en la gestión de ficheros de texto y ficheros binarios.
- ✓ Diferenciar entre acceso secuencial y aleatorio a ficheros.
- ✓ Aprender las principales clases de Java para manejo de ficheros (y flujos en general) y su uso.
- ✓ Comprender el mecanismo de *buffering,* cómo permite acelerar las operaciones de lectura y escritura en ficheros tanto binarios como de texto, y cómo permite la lectura y escritura por líneas en ficheros de texto.
- ✓ Acceder correctamente a los contenidos de ficheros de texto con distintas codificaciones.
- ✓ Analizar algunas organizaciones sencillas de ficheros y la manera en que se pueden utilizar para la persistencia de datos.

Mapa conceptual



Glosario

Acceso aleatorio. Tipo de acceso a un fichero que permite acceder directamente a los datos situados en cualquier posición del fichero.

Acceso secuencial. Tipo de acceso a un fichero con el que la única manera de acceder a los datos situados en una posición determinada es leer todos los contenidos desde el principio hasta dicha posición.

Codificación de texto. Una manera particular de representar una secuencia de caracteres de texto mediante una secuencia de *bytes*.

Fichero de texto. Fichero que contiene texto.

Fichero. Unidad fundamental de almacenamiento. Consiste en una secuencia de *bytes*. Con una adecuada organización, se puede utilizar para almacenar cualquier tipo de información.

Índice. Fichero que permite recorrer los contenidos de otro fichero en un orden determinado.

Registro. Estructura para representar información. Consta de una serie de campos, en cada uno de los cuales se puede almacenar un dato particular.

2.1. Persistencia de datos en ficheros

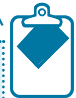
Los ficheros son el método de almacenamiento de información más elemental. De hecho, en última instancia, todos los métodos de almacenamiento, por sofisticados que sean, almacenan los datos en ficheros.

Hasta que fueron relegados en los años ochenta por las bases de datos relacionales, los ficheros fueron el principal medio de almacenamiento de datos. El nombre *fichero* se utilizó por analogía con los antiguos ficheros que contenían fichas de papel, todas con la misma estructura, consistente en un conjunto fijo de campos. En el caso de los libros de una biblioteca, por ejemplo, los campos podían ser el título, nombre del autor, tema, etc. Los ficheros que contenían fichas de papel fueron reemplazados por ficheros de ordenador que contenían registros, equivalentes a las antiguas fichas de papel. Un registro contiene un conjunto de campos de longitud fija. Para acelerar las búsquedas se empezaron a utilizar ficheros auxiliares de índice que permitían acceder a los registros según un orden determinado. IBM desarrolló un avanzado sistema de gestión de ficheros llamado ISAM *(indexed sequential access method)*. El lenguaje COBOL, creado en 1959, pero aún hoy ampliamente utilizado en determinados ámbitos (por ejemplo, el sector bancario), proporciona un excelente soporte para ficheros indexados.

Los ficheros como medio de almacenamiento masivo de datos fueron relegados progresivamente en favor de las bases de datos relacionales. En realidad, las bases de datos relacionales utilizan internamente sofisticados sistemas de gestión de ficheros para el almacenamiento de los datos.

En este capítulo se presentarán algunas organizaciones sencillas de ficheros, y se explicará todo lo necesario para escribir sencillos programas en Java para consultar, añadir, borrar y modificar la información contenida en ellos.

TOMA NOTA



El almacenamiento de datos en ficheros de texto con organizaciones sencillas puede ser una solución perfectamente válida para algunas aplicaciones, pero nunca hay que perder de vista sus limitaciones intrínsecas y su limitada escalabilidad. Si hay que realizar consultas complejas o que requiere relacionar mucha información diversa, será difícil escribir un programa para realizarlas. Si el volumen de datos para manejar es muy grande, o si es necesario realizar con mucha frecuencia operaciones de borrado o modificación de datos, el rendimiento será muy pobre. Permitir que varios programas realicen a la vez operaciones de consulta y actualización puede introducir inconsistencias en los datos e incluso dañar los ficheros. Para evitar estos problemas son necesarios elaborados mecanismos de control de acceso que añaden mucha complejidad al sistema, y mucho más complicado es añadir soporte para transacciones. Es difícil establecer y hacer que se cumplan restricciones de integridad sobre los datos. Y también es difícil evitar redundancias en los datos que, además de desperdiciar espacio de almacenamiento, puede hacer que surjan inconsistencias cuando se añaden o modifican datos. Porque si la misma información está en más de un lugar, puede acabar teniendo un valor distinto en cada uno.

Se siguen utilizando ficheros para la persistencia de datos en muchas aplicaciones, y muy importantes. Las bases de datos relacionales han reemplazado los sistemas basados en ficheros para grandes colecciones de datos con una estructura muy regular, lo que es muy importante, pero no lo es todo. Hoy en día se utilizan mucho los ficheros en formato XML (al que se dedicará un capítulo posterior) para el almacenamiento de todo tipo de datos. Los sistemas de

correo electrónico suelen mantener sus datos en ficheros con un formato relativamente sencillo. Muchos procesos masivos que se ejecutan periódica o puntualmente se realizan basándose en datos proporcionados en ficheros de texto con una estructura muy sencilla. También se usan ficheros de texto sencillos para exportación e importación de datos entre sistemas, y también para copias de seguridad. Aparte de eso, está la infinidad de aplicaciones que almacenan los datos con los que trabajan en ficheros con infinidad de formatos diferentes.

2.2. Tipos de ficheros según su contenido

Un fichero es simplemente una secuencia de *bytes,* con lo que en principio puede almacenar cualquier tipo de información (véase figura 1.3).

Un fichero se identifica por su nombre y su ubicación dentro de una jerarquía de directorios.

Los ficheros pueden contener información de cualquier tipo, pero a grandes rasgos cabe distinguir entre dos tipos: los ficheros de texto y los ficheros binarios:

1. *Ficheros de texto:* contienen única y exclusivamente una secuencia de caracteres. Estos pueden ser caracteres visibles tales como letras, números, signos de puntuación, etc., y también espacios y separadores tales como tabuladores y retornos de carro. Su contenido se puede visualizar y modificar con cualquier editor de texto, como por ejemplo el bloc de notas en Windows o gedit en Linux.
2. *Ficheros binarios:* son el resto de los ficheros. Pueden contener cualquier tipo de información. En general, hacen falta programas especiales para mostrar la información que contienen. Los programas también se almacenan en ficheros binarios.

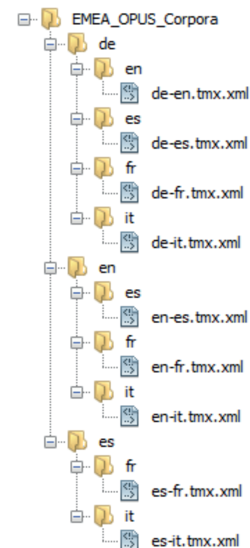


Figura 2.1
Ficheros en una jerarquía de directorios

2.3. Codificaciones para texto

Aunque la cuestión de las codificaciones para texto se ha incluido en este capítulo dedicado a los ficheros, es relevante para cualquier medio de almacenamiento de datos porque, allá donde se almacene un texto, debe hacerse con una codificación determinada.

Un texto es una secuencia de caracteres. Un texto, como cualquier tipo de información, se almacena en memoria o en cualquier dispositivo de almacenamiento como una secuencia de *bytes.* Una codificación es un método para representar cualquier texto como una secuencia de *bytes.* El mismo texto, según la codificación empleada, se puede representar como una secuencia de *bytes* distinta.

La variedad de codificaciones es un problema cada vez menos importante en la práctica, debido a la implantación general de Unicode y su codificación UTF-8, pero todavía es relativamente frecuente encontrar textos con otras codificaciones, sobre todo en entornos Windows.

Las más frecuentes son ISO 8859-1 y Windows-1252, que se puede considerar una variante no estándar de Microsoft del estándar ISO 8859-1. UTF-8 es compatible con el código ASCII,

lo que significa que cualquier texto codificado en ASCII se representa exactamente igual en UTF-8. Este ha sido un motivo fundamental para la adopción generalizada de UTF-8.

Para las nuevas aplicaciones, siempre hay que utilizar UTF-8 para almacenar texto, a no ser que haya alguna razón de peso para utilizar otra, que normalmente no la hay. A veces hay que importar u obtener textos de otras fuentes, desde donde podrían venir con otras codificaciones. Hay que identificar estas situaciones y hacer la conversión necesaria para recodificar los textos.

Recurso digital

En el anexo web 2.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás información sobre la codificación del contenido de ficheros.

Recuerda

Cualquier editor de texto debería permitir visualizar correctamente un fichero de texto independientemente de su codificación. Por ejemplo, Notepad en Windows y gedit en Linux. Normalmente, con la opción "Guardar como…" se puede seleccionar la codificación que se va a utilizar, y UTF-8 suele aparecer como una de las opciones.

La manera más fiable y segura de confirmar el tipo de un fichero en Linux es con el comando `file`. Este analiza los contenidos del fichero e indica su tipo. Si es de texto, indica la codificación utilizada para el texto, típicamente: ASCII, UTF-8 o iso-8859-1.

El comando `iconv` de Linux permite recodificar ficheros de texto. El siguiente comando, por ejemplo, se podría utilizar para recodificar a UTF-8 un fichero codificado en ISO 8859-1.

```
iconv -f iso8859-1 -t utf-8 fichero_8859-1.txt > fichero_utf8.txt
```

2.4. La clase File de Java

La versión de Java utilizada para este libro es Java SE 8, una versión LTS (*long term support*, es decir, con soporte a largo plazo). En los servidores web de Oracle se puede consultar la documentación de la biblioteca estándar de clases de Java SE 8.

Las clases que permiten trabajar con ficheros están en el paquete `java.io`.

Recurso web www

Se recomienda consultar en http://docs.oracle.com/javase/8/docs/api la documentación de Java SE8 para más información acerca de cualquier clase utilizada en este capítulo. Arriba, a la izquierda, aparece la lista de paquetes por orden alfabético. En ella se puede localizar el paquete `java.io`. Si se pulsa sobre él, aparece debajo la lista de interfaces y clases que contiene el paquete.

La clase `File` permite obtener información relativa a directorios y ficheros dentro de un sistema de ficheros y realizar diversas operaciones con ellos tales como borrar, renombrar, etc. En el siguiente cuadro se proporciona un resumen de la funcionalidad de esta clase, mostrando solo los principales métodos agrupados por categorías.

CUADRO 2.1
Métodos de la clase `File`

Categoría	Modificador/tipo	Método(s)	Funcionalidad
Constructor		`File(String ruta)`	Crea objeto `File` para la ruta indicada, que puede corresponder a un directorio o a un fichero.

Consulta de propiedades	`boolean`	`canRead()`
		`canWrite()`
		`canExecute()`
			Comprueban si el programa tiene diversos tipos de permisos sobre el fichero o directorio, tales como de lectura, escritura y ejecución (si se trata de un fichero). Para un directorio, `canExecute()` significa que se puede establecer como directorio actual.

	`boolean`	`exists()`	Comprueba si el fichero o directorio existe.
	`boolean`	`isDirectory()` `isFile()`	Comprueban si se trata de un directorio o de un fichero.
	`long`	`length()`	Devuelve longitud del fichero.
	`File`	`getParent()` `getParentFile()`	Devuelven el directorio padre.
	`String`	`getName()`	Devuelve nombre del fichero.
Enumeración	`String[]`	`list()`	Devuelve un *array* con los nombres de los directorios y ficheros dentro del directorio.
	`File[]`	`listFiles()`	Devuelve un *array* con los directorios y ficheros dentro del directorio.
Creación, borrado y renombrado	`boolean`	`createNewFile()`	Crea nuevo fichero.
	`static File`	`createTempFile()`	Crea nuevo fichero temporal y devuelve objeto de tipo `File` asociado, para poder trabajar con él.
	`boolean`	`delete()`	Borra fichero o directorio.
	`boolean`	`renameTo()`	Renombra fichero o directorio.
	`boolean`	`mkdir()`	Crea un directorio.
Otras	`java.nio.file.Path`	`toPath()`	Devuelve un objeto que permite acceder a información y funcionalidad adicional proporcionada por el paquete `java.nio`.

El siguiente programa de ejemplo muestra por defecto un listado de los ficheros y directorios que contiene el directorio desde el que se ejecuta el programa. Pero si se le pasa la ruta de un directorio o fichero, muestra información acerca de él y, si se trata de un directorio, muestra los ficheros y directorios que contiene.

```
// Uso de la clase File para mostrar información de ficheros y directorios
package listadodirectorio;
import java.io.File;
public class ListadoDirectorio {
   public static void main(String[] args) {
      String ruta=".";
      if(args.length>=1) ruta=args[0];
      File fich=new File(ruta);
      if(!fich.exists()) {
         System.out.println("No existe el fichero o directorio ("+ruta+").");
      }
      else {
         if(fich.isFile()) {
            System.out.println(ruta+" es un fichero.");
         }
         else {
            System.out.println(ruta+" es un directorio. Contenidos: ");
            File[] ficheros=fich.listFiles(); // Ojo, ficheros o directorios
            for(File f: ficheros) {
               String textoDescr=f.isDirectory() ? "/":
               f.isFile() ? "_": "?";
               System.out.println("("+textoDescr+") "+f.getName());
            }
         }
      }
   }
}
```

PARA SABER MÁS

Se pueden pasar argumentos desde la línea de comandos a cualquier programa. Cualquier programa Java debe tener un método `main(String args [])` que se invoque en el momento de ejecutar el programa. El parámetro `String args[]` proporciona los argumentos de línea de comandos. Para pasar argumentos de línea de comando a un programa que se está desarrollando utilizando un IDE (entorno integrado de desarrollo), lo más cómodo suele ser utilizar las opciones que este IDE proporcione para ello dentro de su interfaz de usuario.

Actividad propuesta 2.1

Modifica el programa anterior para que muestre más información acerca de cada fichero y directorio, al menos el tamaño (si es un fichero), los permisos de los que se dispone sobre el fichero o

directorio, y la fecha de última modificación (en cualquier formato). Los permisos hay que mostrarlos en el formato habitual de Linux, a saber, con tres caracteres seguidos: una *r* si hay permiso para lectura o un guion si no lo hay; una *w* si hay permiso para escritura o un guion si no lo hay; y una *x* si hay permiso para ejecución, en el caso de que se trate de un fichero, o para entrar en el directorio, en el caso de que se trate de un directorio, o un guion si no lo hay.

2.5. Gestión de excepciones en Java

Antes de seguir avanzando con el contenido del capítulo, se incluye un breve repaso a la gestión de excepciones en el lenguaje Java. Cualquier programa escrito en Java debe realizar una adecuada gestión de excepciones. En este apartado se explicará lo más importante de la gestión de excepciones en Java, o al menos todo lo que se vaya a necesitar para este libro.

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe su curso normal de ejecución. Una excepción podría producirse, por ejemplo, al dividir por cero, como se muestra en el siguiente ejemplo.

```
// Excepción no gestionada durante la ejecución de un programa.
package excepcionesdivporcero;
public class ExcepcionesDivPorCero {
    public int divide(int a, int b) {
        return a/b;
    }
    public static void main(String[] args) {
        int a,b;
        a=5; b=2; System.out.println(a+"/"+b+"="+a/b);
        b=0; System.out.println(a+"/"+b+"="+a/b);
        b=3; System.out.println(a+"/"+b+"="+a/b);
    }
}
```

Al ejecutar este programa se obtendrá algo similar a lo siguiente:

```
5/2=2
Exception in thread "main" java.lang.ArithmeticException: / by zero at excepcionesdivporcero.
ExcepcionesDivPorCero.main(ExcepcionesDivPorCero.java:25)
```

2.5.1. Captura y gestión de excepciones

Cuando tiene lugar una excepción no gestionada, como con el programa anterior, aparte de mostrarse un mensaje de error, se aborta la ejecución del programa. Por ese motivo, el programa anterior no muestra el resultado de la última división entera. Cualquier fragmento de programa que pueda generar una excepción debería capturarlas y gestionarlas.

La salida del programa anterior indica el tipo de excepción que se ha producido, a saber, `ArithmeticException`. Se puede capturar esta excepción con un sencillo bloque `try {} catch {}` y gestionarla, con lo que el programa anterior podría quedar así:

```
// Excepción gestionada durante la ejecución de un programa.
package excepcionesdivporcerogest;
public class ExcepcionesDivPorCeroGest {
    public int divide(int a, int b) {
        return a / b;
    }
    public static void main(String[] args) {
        int a, b;
        a = 5; b = 2;
        try {
            System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmeticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 0; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmeticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 3; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmeticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
    }
}
```

Al ejecutar este programa, se captura y gestiona cualquier excepción que se pueda producir, de manera que se muestra un mensaje apropiado y se continúa con la ejecución. La salida del anterior programa sería la siguiente:

```
5/2=2
Error al dividir: 5/0
5/3=1
```

RECUERDA

✓ Es recomendable escribir los mensajes de error en la salida de error `System.err` en lugar de en la salida estándar `System.out`.

Actividad propuesta 2.2

¿Cómo cambiaría la funcionalidad del programa anterior si, en lugar de haber un bloque `try … catch …` para cada división, hubiera uno solo para las tres divisiones?

Las excepciones son objetos de Java, pertenecientes a la clase `Exception` o a una subclase de ella. Esta clase tiene varios métodos interesantes para obtener información acerca de la excepción que se ha producido.

CUADRO 2.2
Métodos de la clase `Exception`

Modificador/tipo	Método(s)	Funcionalidad
`void`	`printStackTrace()`	Muestra información técnica muy detallada acerca de la excepción y el contexto en que se produjo. Lo hace en la salida de error, `System.err`. Al principio del desarrollo de un programa, y para programas de prueba, puede ser una buena opción utilizar esta función para mostrar información de todas las excepciones, y perfilar más adelante cómo se gestionan excepciones de tipos particulares.
`String`	`getMessage()`	Proporciona un mensaje detallado acerca de la excepción.
`String`	`getLocalizedMessage()`	Proporciona una descripción localizada (es decir, traducida a la lengua local) de la excepción.

2.5.2. Gestión diferenciada de distintos tipos de excepciones

En un bloque `try {} catch {}` se pueden gestionar por separado distintos tipos de excepciones. Es conveniente incluir un manejador para `Exception` al final para que ninguna excepción se quede sin gestionar.

PARA SABER MÁS

En aplicaciones profesionales, la práctica habitual es utilizar herramientas de *logging* (de registro), y no solo para mensajes de error. Los mensajes se suelen registrar en ficheros. Estas herramientas permiten generar de forma diferenciada distintos tipos de mensaje, típicamente, y por orden de importancia, de mayor a menor: *error, warning* (de aviso), *info* (informativo) y *debug* (para depuración). Permiten configurar el nivel de *logging,* de manera que solo se registren los mensajes a partir del nivel de importancia establecido. Si este se establece como *warning,* se mostrarían los de tipo *warning* y *error*. Entre las herramientas más ampliamente utilizadas para Java están:

- Java Logging API (http://docs.oracle.com/javase/8/docs/technotes/guides/logging)
- Log4j (http://logging.apache.org/log4j)

El siguiente programa rellena un *array* con números y después realiza un cálculo aritmético para cada uno con ellos y muestra la segunda cifra del resultado. Esto no es realmente muy útil, como no sea para mostrar cómo se puede hacer saltar excepciones de diversos tipos, capturarlas y gestionarlas por separado. Solo para un elemento del *array* se realiza el cálculo sin que salte ninguna excepción.

```
// Gestión diferenciada de distintos tipos de excepciones
package excepcionesdiversas;
public class ExcepcionesDiversas {
    public static void main(String[] args) {
    // Rellenar array con números variados
    int nums[][] = new int[2][3];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
            nums[i][j] = i + j;
            }
        }
        // Realizar cálculo para cada posición del array.
        // Se producen excepciones de diversos tipos.
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                try {
                    System.out.print("Segunda cifra de 5*nums["+i+"]
                        ["+j+"]/"+j+": ");
                    System.out.println(String.valueOf(5 * nums[i][j] /
                        j).charAt(1));
                } catch(ArithmeticException e) {
                    System.out.println("ERROR: aritmético 5*"+nums[i][j]+"/"+j);
                } catch (ArrayIndexOutOfBoundsException e) {
                    System.out.println("ERROR: No existe nums["+i+"]["+j+"]");
                } catch (Exception e) {
                    System.out.println("ERROR: de otro tipo al calcular segunda
                        cifra de: 5*"+nums[i][j]+"/"+j);
                    System.out.println();
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Para poder interpretar más fácilmente la salida de este programa, todos los mensajes de error se han dirigido hacia `System.out` y no `System.err`. Su salida sería:

```
Segunda cifra de 5*nums[0][0]/0: ERROR: aritmético 5*0/0
Segunda cifra de 5*nums[0][1]/1: ERROR: de otro tipo al calcular segunda cifra de: 5*1/1

java.lang.StringIndexOutOfBoundsException: String index out of range: 1
at java.lang.String.charAt(String.java:658)
at excepcionesdiversas.ExcepcionesDiversas.main(ExcepcionesDiversas.java:31)
Segunda cifra de 5*nums[0][2]/2: ERROR: de otro tipo al calcular segunda cifra de: 5*2/2

java.lang.StringIndexOutOfBoundsException: String index out of range: 1
at java.lang.String.charAt(String.java:658)
at excepcionesdiversas.ExcepcionesDiversas.main(ExcepcionesDiversas.java:31)
Segunda cifra de 5*nums[1][0]/0: ERROR: aritmético 5*1/0
Segunda cifra de 5*nums[1][1]/1: 0
Segunda cifra de 5*nums[1][2]/2: ERROR: de otro tipo al calcular segunda cifra de: 5*3/2
```

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 1
at java.lang.String.charAt(String.java:658)
at excepcionesdiversas.ExcepcionesDiversas.main(ExcepcionesDiversas.java:31)
Segunda cifra de 5*nums[2][0]/0: ERROR: No existe nums[2][0]
Segunda cifra de 5*nums[2][1]/1: ERROR: No existe nums[2][1]
Segunda cifra de 5*nums[2][2]/2: ERROR: No existe nums[2][2]
```

2.5.3. Declaración de excepciones lanzadas por un método de clase

Si el compilador es capaz de determinar que un método de una clase puede originar un tipo de excepción, pero no lo gestiona mediante un bloque `catch(){}`, la compilación terminará con un error. Una posibilidad entonces es gestionar la excepción en el método mediante un bloque `catch(){}`. La otra es añadir el modificador `throws` seguido de la clase de excepción.

La siguiente clase tiene un método que crea un fichero temporal con un nombre que empieza por un prefijo dado, y escribe en él un carácter dado, un número dado de veces. Durante este proceso puede saltar la excepción `IOException` en varias ocasiones, pero no se quiere gestionarla en el método. Por ello se incluye `throws IOException` en su declaración. Por lo demás, el método es sencillo y, en cualquier caso, lo importante es comprender la gestión de excepciones. Las clases y procedimientos utilizados se verán más adelante en este capítulo.

```
// Declaración de excepciones lanzadas por método de clase con throws

package excepcionesconthrows;

import java.io.File;
import java.io.IOException;
import java.io.FileWriter;

public class ExcepcionesConThrows {

    public File creaFicheroTempConCar(String prefNomFich, char car, int
        numRep) throws IOException {
        File f = File.createTempFile(prefNomFich, "");
        FileWriter fw = new FileWriter(f);
        for (int i = 0; i < numRep; i++) fw.write(car);
        fw.close();
        return f;
    }

    public static void main(String[] args) {
        try {
            File ft = new ExcepcionesConThrows().
                creaFicheroTempConCar("AAAA_", 'A', 20);
            System.out.println("Creado fichero: " + ft.getAbsolutePath());
            ft.delete();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```