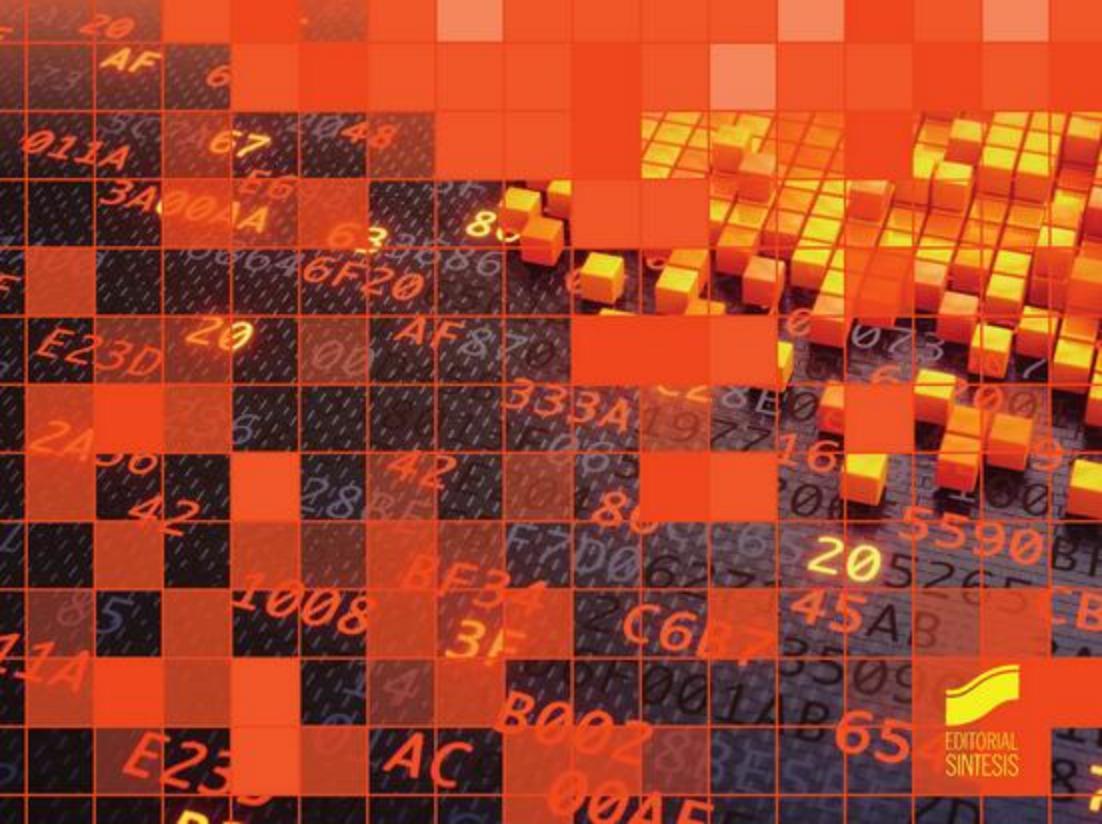




Acceso a datos

Carlos Alberto Cortijo Bon



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: <code>hibernate.cfg.xml</code>	115
4.4.4. Fichero de ingeniería inversa <code>hibernate.reveng.xml</code>	116
4.4.5. POJO (clases) y ficheros de correspondencia	117
4.4.6. <code>HibernateUtil.java</code>	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 <code>hbm</code> 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 <code>save()</code> , <code>saveOrUpdate()</code> y <code>persist()</code>	135
4.10.2. Obtención de un objeto persistente con <code>get()</code> y <code>load()</code>	136
4.10.3. De persistente a eliminado con <code>delete()</code>	137
4.10.4. De persistente a separado con <code>evict()</code> , <code>close()</code> y <code>clear()</code>	137
4.10.5. De separado a persistente con <code>update()</code> , <code>saveOrUpdate()</code> , <code>lock()</code> y <code>merge()</code>	137
4.11. Lenguajes de consulta HQL y JPQL	138
4.11.1. La interfaz <code>Query</code>	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

Presentación

Este libro se plantea como una completa introducción a la persistencia de datos en diversos soportes: ficheros, bases de datos relacionales, bases de datos de objetos y XML, tanto documentos individuales como bases de datos de XML. Va dirigido en especial al alumnado del ciclo superior de Desarrollo de Aplicaciones Multiplataforma que cursan el módulo Acceso a Datos. Pero es de interés también para estudiantes universitarios o para profesionales interesados en estas tecnologías.

Por encima de las particularidades de cada uno de estos tipos de sistemas diversos para persistencia de datos, existen aspectos comunes que sirven de hilo conductor y permiten un tratamiento unificado y coherente de todos ellos, de manera que este libro sea algo más que una exposición secuencial de tecnologías, técnicas y herramientas diversas, y haga hincapié en los aspectos fundamentales y, en gran medida, comunes. Los lenguajes de definición de datos para definir un esquema de almacenamiento apropiado y efectivo, la introducción de restricciones de integridad para evitar redundancias, inconsistencias e incorrecciones en los datos, los lenguajes de consulta para obtener la información que en cada momento o para cada aplicación se necesite, los mecanismos de iteración para obtener los resultados de las consultas realizadas, y las transacciones para garantizar la consistencia e integridad de los datos cuando se realizan modificaciones en ellos. Todos estos aspectos se cubren para cada uno de los tipos de sistemas para persistencia de datos.

A parte de eso, se dedica un primer capítulo de introducción a proporcionar una perspectiva general y una breve introducción a estos aspectos fundamentales y recurrentes. La persistencia de datos es un campo muy amplio y diverso, pero en resumen consiste en utilizar programas que pueden almacenar y recuperar datos en sistemas de almacenamiento de datos. Sugerido así, y teniendo en cuenta el planteamiento eminentemente práctico y pragmático de este libro, tienen una importancia capital los lenguajes orientados a objetos y las bases de datos relacionales. Estas últimas tienen, desde hace décadas (desde los años ochenta del siglo xx), un predominio incuestionable como sistemas de almacenamiento de datos. Se asume un conocimiento básico

por parte del lector de ambos, y en particular de Java para programación de aplicaciones y del lenguaje SQL para manejo de bases de datos relacionales. En consonancia con esto, se dedica mucha atención a ORM, la correspondencia objeto-relacional, y a otros planteamientos existentes para el almacenamiento en bases de datos relacionales de objetos y de XML, facilitados además por la inclusión en sucesivas revisiones del estándar SQL de soporte para objetos y para documentos de XML. El planteamiento es eminentemente práctico, pero sin renunciar a una consistente y sólida base teórica.

Cada capítulo comienza con una breve introducción histórica y una explicación de la importancia actual del planteamiento en cuestión para persistencia de datos, lo que permite hacerse una idea propia de la importancia que es de esperar que tengan en el futuro. Después, se explican brevemente los estándares disponibles y los lenguajes para definición, consulta y modificación de datos, y las API que permiten utilizarlos desde programas de aplicación. Se presentan programas de ejemplo que ilustran todos los aspectos teóricos planteados. Los programas de ejemplo suelen desarrollarse siempre sobre varios escenarios comunes que se plantean en todos los temas. Se plantea, pues, la persistencia del mismo modelo conceptual con los distintos sistemas y con los esquemas de almacenamiento que cada uno proporciona (fundamentalmente el relacional, de objetos y de XML). Con ello se persigue fomentar el espíritu crítico y la propia capacidad para evaluar la aplicabilidad de distintos planteamientos en persistencia de datos con problemas concretos, tras analizar objetivamente las ventajas e inconvenientes de cada uno.

También se intercalan actividades, que en general consisten en la realización de programas similares a –o basados en, o ampliaciones de– los programas de ejemplo. El objetivo es fomentar la capacidad de comprender programas ya existentes, como paso previo para modificarlos según las propias necesidades. Para ello es necesario consultar y buscar información, formular hipótesis y experimentar diversos planteamientos para la resolución de problemas concretos. Todas estas son habilidades fundamentales en el desempeño profesional en el sector del desarrollo de aplicaciones. La capacidad de investigar de forma autónoma y de evaluar críticamente diversas soluciones alternativas, una vez asimilados los conceptos teóricos sobre los que se fundamentan, hace posible la necesaria actualización constante de conocimientos y habilidades en un campo como el desarrollo de aplicaciones, en el que es patente que todo cambia continuamente, pero en el que, a la vez, y aunque muchas veces no parezca tan evidente, los principios fundamentales son, en gran medida, y en lo primordial, siempre los mismos, y en el que es precisamente un entendimiento de estos conceptos fundamentales lo que permite comprender los continuos cambios que se suceden y formarse una idea propia de su importancia e implicaciones reales.

En cada capítulo se añade un resumen y unas actividades de autoevaluación, que permiten hacerse una idea del grado de asimilación de los conceptos teóricos fundamentales, así como una lista de ejercicios que requieren un mayor grado de reflexión y elaboración previa, y en ocasiones de investigación, presentados en orden creciente de dificultad.

Además, en la página web de la editorial (www.sintesis.com) se incluye un archivo con los programas de ejemplo y varios recursos adicionales para el libro. Estos contenidos están especificados en recuadros llamados *Recurso digital* dentro del libro.

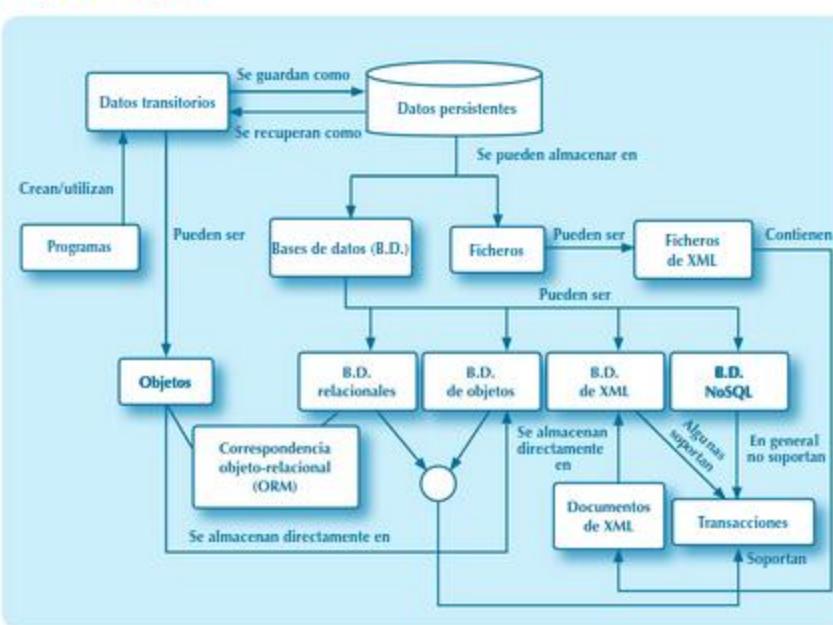
Dedico este libro a mis padres, por todas las oportunidades que me han dado y por hacer posible este libro con su esfuerzo y cariño. A Ana, que siempre me ayuda a encontrar el camino. Y a Celia, cuya sonrisa ilumina el mundo.

Introducción y conceptos básicos

Objetivos

- ✓ Proporcionar una breve introducción a los contenidos del libro.
- ✓ Realizar un breve repaso a conceptos preliminares, algunos probablemente ya conocidos.
- ✓ Introducir el concepto de *persistencia de datos*, los medios de almacenamiento que sirven de soporte para datos persistentes, y los mecanismos que pueden utilizar los programas de aplicación para trabajar de manera efectiva con datos persistentes.

Mapa conceptual



Glosario

API (Application Programming Interface). Interfaz de programación de aplicaciones. Conjunto de funciones disponibles para su uso por parte de programas de aplicaciones, y que estos pueden utilizar para acceder a determinados servicios.

Base de datos de objetos o base de datos orientada a objetos. Base de datos que almacena directamente objetos, en el sentido que tiene el término *objeto* en los lenguajes de programación orientados a objetos.

Base de datos de XML. Base de datos que almacena documentos en formato XML.

Base de datos NoSQL. Base de datos que no es de ninguno de los tipos anteriores, que normalmente tiene almacenamiento basado en estructuras básicas muy sencillas y flexibles, y que prima la disponibilidad y la eficiencia sobre el soporte de transacciones. Con frecuencia en este grupo se incluyen las bases de datos de XML.

Base de datos relacional. Base de datos que representa los datos y las relaciones entre ellos de acuerdo al modelo relacional, y que utiliza tablas como estructuras básicas para el almacenamiento de los datos.

Desfase objeto-relacional. Conjunto de dificultades que plantea la persistencia de objetos en las bases de datos relacionales.

Fichero. Secuencia de bytes almacenada en un medio de almacenamiento secundario, a la que se puede acceder indicando su nombre y su ubicación dentro de una jerarquía de directorios existente en dicho medio.

Persistencia de datos. Paso de datos de memoria principal a un medio de almacenamiento secundario, de manera que más adelante se puedan recuperar en memoria principal.

Transacción. Secuencia de operaciones de consulta y modificación de datos persistentes realizadas por un programa, que se realizan como un todo, y de manera aislada con cualquier otra operación sobre los mismos datos que pudiera realizar cualquier otro programa.

XML (eXtensible Markup Language). Lenguaje que tiene una sintaxis muy sencilla y que permite la representación de cualquier tipo de información.

1.1. Programas y datos

Se puede decir, a grandes rasgos, que los ordenadores ejecutan programas que gestionan información. La información se representa en forma de datos que pueden estar estructurados de distintas formas.

El concepto de *ordenador* es muy amplio. Hoy en día incluye no solo los servidores, ordenadores de sobremesa y portátiles tradicionales, sino también los teléfonos móviles, tabletas, televisores inteligentes, etc., sumándose cada día más dispositivos.

Los tipos de datos que manejan y los tipos de operaciones que realizan con ellos son muy diversos. Pero siempre cuentan con dos tipos de medios de almacenamiento de datos:

1. *Almacenamiento primario o memoria principal:* donde se almacenan los datos con los que en cada momento está trabajando el programa. Sus contenidos se borran cuando se apaga el ordenador. Tiene una capacidad relativamente baja y el tiempo de acceso a los datos es muy corto.
2. *Almacenamiento secundario:* donde se almacenan datos de manera permanente. Medios de almacenamiento secundario son discos duros, memorias *flash* de distintos tipos (*pen drives* o tarjetas de memoria), etc. Los datos que se almacenan en estos medios son *datos persistentes*, no se borran cuando se apaga el ordenador. Tienen una capacidad de almacenamiento relativamente alta y el tiempo de acceso a los datos es relativamente largo.

1.2. Persistencia de datos

Los programas solo pueden consultar directamente datos almacenados en memoria principal. Un programa puede crear datos en memoria principal y guardarlo en almacenamiento secundario. De esa manera, estos datos se convierten en datos persistentes. Y a la inversa, un programa puede recuperar datos persistentes desde un medio de almacenamiento secundario para pasarlo

a memoria principal. Una vez ahí se convierten en datos transitorios que el programa puede consultar, modificar y, llegado el caso, volver a almacenar en almacenamiento secundario, es decir, volver a hacer persistentes.



Figura 1.1
Persistencia de datos

Es muy habitual que el medio de almacenamiento secundario esté en un ordenador distinto a aquel en que se ejecuta el programa que los utiliza. Puede ser un servidor que proporciona servicios de persistencia de datos a múltiples aplicaciones que se ejecutan en distintos ordenadores y que se comunican con el servidor mediante protocolos estándares de red. Los programas de aplicación suelen acceder a los servicios para almacenamiento y consulta de datos persistentes mediante funciones de alto nivel proporcionadas por diversas API (interfaces de programación de aplicaciones).

1.3. Sistemas de persistencia de datos

En última instancia, los datos persistentes se guardan siempre en ficheros dentro de un sistema de ficheros en un medio de almacenamiento secundario. Un sistema de ficheros consiste en una jerarquía de directorios o carpetas, y en cada carpeta de esta jerarquía puede haber ficheros que consisten básicamente en una secuencia de bytes.

Sobre este sistema básico de almacenamiento se pueden organizar sistemas de almacenamiento de datos más o menos sofisticados.

Dirección: /var/lib/mysql		
Directorio		Nombre
		Tamaño
mysql	..	56 bytes
	cb_synt_corpus	
	mysql	
	performance_schema	
	phpmyadmin	
	sys	
	mysql-files	
	mysql-keyring	
	mysql-upgrade	
	NetworkManager	
	nssdb	
	ntpdate	
	ofono	
openvpn	auto.cnf	56 bytes
	debian-5.5.flag	0 bytes
	debian-5.7.flag	0 bytes
	ib_buffer_pool	893 bytes
	ib_logfile0	50.3 MB
	ib_logfile1	50.3 MB
	ibdata1	15.6 GB
	ibtmp1	12.6 MB
	mysql_upgrade_info	6 bytes

Figura 1.2
Contenidos de un sistema de ficheros

dos. De hecho, en la figura 1.2 se pueden ver los ficheros en los que se almacenan los datos de una base de datos de MySQL.

En este capítulo se hace una revisión general de los principales sistemas disponibles en la actualidad para persistencia de datos. En los siguientes capítulos se explicarán con más detalle. Cada sistema puede ser más o menos apropiado y tendrá sus ventajas e inconvenientes para cada aplicación. Hay que tenerlas en cuenta y evaluarlas críticamente para decidir la mejor solución.

En los siguientes apartados se describirán varios aspectos fundamentales de los sistemas de persistencia de datos. Por último, se describirán brevemente los tipos más importantes.

TOMA NOTA



No solo hay que tener en cuenta las necesidades actuales, sino también las futuras. Un aspecto muy importante es la escalabilidad, es decir, la posibilidad de hacer frente a mayores volúmenes de datos y cargas de trabajo. También son importantes aspectos como la estandarización, el grado de madurez o consolidación, y si se puede esperar soporte y esfuerzo de desarrollo en el futuro para las tecnologías y productos utilizados.

1.4. Almacenamiento de datos

Cada sistema proporciona una estructura básica de almacenamiento para los datos.

1.4.1. Ficheros

Los ficheros proporcionan por sí mismos una organización secuencial de los datos. Un fichero no es otra cosa que una secuencia de bytes.

Sobre este soporte elemental se puede representar cualquier tipo de información. De hecho, como ya se ha comentado, todos los sistemas de almacenamiento de datos almacenan los datos en ficheros. Pero cuando se habla de sistemas basados en ficheros, se suelo hacer referencia a un tipo particular ampliamente utilizado hasta la irrupción de las bases de datos relacionales en los años ochenta. En este tipo, la información dentro de cada fichero se almacena en una secuencia de registros de longitud fija, y cada registro está compuesto por varios campos de longitud fija. Por ejemplo, un registro podría tener los datos de un cliente, almacenados en campos tales como DNI, nombre, dirección, etc. A un fichero así se le llama *fichero secuencial*, porque está formado por una secuencia de registros.

1100	0001	0101	0001	0001	0001
1011	0110	0110	0000		1110	0110

Figura 1.3
Fichero

89012345E	23456789D	12345678Z	56789012B	78901234X	45678901G	67890123C	34567890V
ROJAS	DORCE	ARCOS	SAMPER	NADALES	JUÁREZ	GAMBOA	NADAL

Figura 1.4
Fichero secuencial

Para acelerar las consultas sobre un fichero secuencial se pueden crear ficheros de índice, que permiten acceder a sus contenidos en un orden determinado. Para el fichero secuencial anterior se podría crear un fichero de índice para el campo *nombre*. Al existir un índice para él, pasa a ser un fichero secuencial indexado. Nada impide crear tantos índices como se quiera sobre un fichero secuencial. Nótese que un fichero de índice no deja de ser un tipo particular de fichero secuencial que incluye un solo campo del fichero, con sus registros ordenados por ese campo, y un campo que indica la posición en el fichero secuencial.

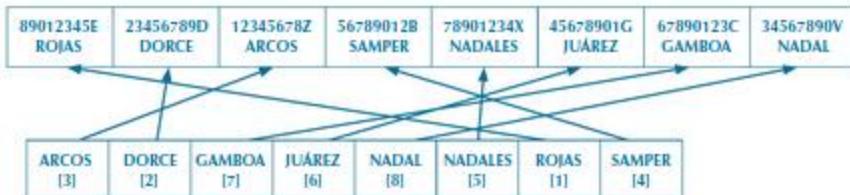
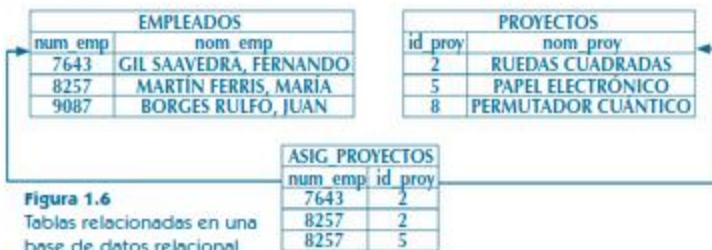


Figura 1.5
Fichero secuencial indexado

1.4.2. Bases de datos relacionales

Las bases de datos relacionales organizan los datos en tablas, y permiten especificar las relaciones entre dichas tablas.



Hoy en día es abrumador el predominio de las bases de datos relacionales, que en los años ochenta desplazaron a los antiguos sistemas basados en ficheros. SQL es el lenguaje estándar para bases de datos relacionales. Por ello, frecuentemente, se denomina *base de datos NoSQL* a cualquier base de datos no relacional. Dentro de esta categoría, de todas formas, cabe diferenciar algunos tipos particulares, como las bases de datos de XML y las bases de datos de objetos. Excluyendo estos tipos quedan diversas bases de datos, muy diferentes entre sí, pero que comparten algunas características generales y que, además, desde hace un tiempo, se vienen utilizando cada vez más. También se usa con frecuencia el término NoSQL, con un sentido más restrictivo, para referirse a ellas. Y es este último el sentido con el que se utilizará, en general, en este libro.

1.4.3. Documentos de XML

En los documentos de XML la información se organiza de manera jerárquica, es decir, en forma de árbol. El modelo DOM es un modelo estándar para representar los contenidos de un documento XML como una jerarquía de nodos.

```

<?xml version="1.0" encoding="UTF-8"?>
<clientes>
  <cliente DNI="78901234X">
    <apellidos>NADELES</apellidos>
    <CP>44126</CP>
  </cliente>
  <cliente DNI="89012345E">
    <apellidos>ROJAS</apellidos>
    <valides estado="borrado" timestamp="1528286082" />
  </cliente>
  <cliente DNI="56789012B">
    <apellidos>SAMPER</apellidos>
    <CP>29730</CP>
  </cliente>
</clientes>

<clientes>
  <cliente DNI="78901234X">
    <apellidos>NADELES</apellidos>
    <CP>44126</CP>
  </cliente>
  <cliente DNI="89012345E">
    <apellidos>ROJAS</apellidos>
    <valides estado="borrado" timestamp="1528286082" />
  </cliente>
  <cliente DNI="56789012B">
    <apellidos>SAMPER</apellidos>
    <CP>29730</CP>
  </cliente>
</clientes>

```

Figura 1.7

Fichero XML y organización jerárquica de los datos que contiene (modelo DOM)

Un documento de XML se puede almacenar en un fichero de texto. Una colección de ficheros de XML se puede organizar en una jerarquía de directorios dentro de un sistema de ficheros, junto con ficheros de otros tipos. Pero también se puede almacenar en una base de datos de XML. Las bases de datos de XML organizan los datos en una jerarquía de colecciones, y cada colección puede contener múltiples documentos de tipo XML, e incluso a veces de otros tipos.

1.4.4. Bases de datos de objetos

Las bases de datos de objetos almacenan objetos. Un objeto complejo puede incluir referencias a otros objetos y a colecciones de objetos relacionados con él. Una colección de objetos complejos tiene estructura de grafo.

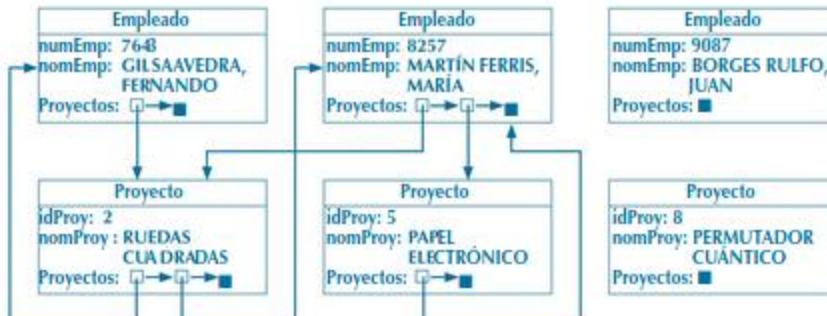


Figura 1.8. Representación de una colección de objetos complejos en forma de grafo

1.4.5. Bases de datos NoSQL

Las diversas bases de datos NoSQL representan la información de manera muy diversa y, en general, diferente de todos los modelos vistos hasta ahora. Pero en general utilizan estructuras muy sencillas y a la vez muy flexibles.

1.5. Restricciones de integridad

Las restricciones de integridad son condiciones que siempre deben cumplir los datos almacenados. Se pueden definir para datos particulares (por ejemplo, no se permite que el nombre de un cliente no esté indicado), y para datos relacionados (por ejemplo, no se permite que el cliente para una factura no exista, y eso significa que esté indicado en la factura, y además que exista como entidad independiente, por supuesto cumpliendo las restricciones de integridad que se hayan especificado para los clientes, como que el nombre esté indicado). Si el resultado de una modificación sobre los datos viola alguna restricción de integridad, no se permitirá. Por ejemplo, no se permitirá dar de alta un cliente si no se indica su nombre, y no se permitirá borrar un cliente para el que exista alguna factura.

1.6. Acceso a los datos con iteradores

Se utilizan iteradores (a veces también se les llama *cursors*) para obtener los resultados de consultas a diversos sistemas de persistencia de datos. En algunos casos es posible utilizar los iteradores no solo para consultar los datos, sino también para modificarlos.

Pasar a memoria principal todos los resultados de una consulta de una vez no es conveniente en general, porque podrían ser muchos. Un iterador se crea para una consulta y permite obtener uno a uno sus resultados. En el caso de bases de datos relacionales, por ejemplo, un iterador permite obtener los resultados fila a fila; en el caso de bases de datos de XML, nodo a nodo; y en el caso de bases de datos de objetos, objeto a objeto. Como las operaciones de recuperación de datos desde almacenamiento secundario son relativamente lentas, se suelen recuperar bloques de datos en cada una, aunque después el iterador los proporcione uno a uno.

1.7. Control de accesos concurrentes y transacciones

Los datos almacenados pueden compartirse entre múltiples aplicaciones, y estas pueden realizar lecturas y modificaciones concurrentes (simultáneas) sobre ellos. Por ello, debe arbitrarse el acceso a los datos para evitar problemas que puedan surgir cuando distintos programas realizan consultas y modificaciones simultáneas sobre los mismos datos. Es además muy habitual que un grupo de operaciones sobre datos relacionados formen un todo que debe llevarse a cabo conjuntamente y de manera aislada con respecto a otras operaciones simultáneas sobre esos datos. En ese caso se consideran una *transacción*.

Ejemplo

Si se hace una transferencia de dinero de una cuenta a otra, el importe se debe restar del saldo de la cuenta de origen y sumar al de la cuenta de destino. Si, por la razón que fuera, no se pudiera sumar el saldo a la cuenta de destino, no debe hacerse efectiva la resta del saldo a la cuenta de origen, sino que debe deshacerse para que todo quede como al principio. Las dos operaciones de las que consta una transferencia constituyen una transacción.

Además, no se pueden realizar simultáneamente dos transferencias que involucren a la misma cuenta. En todo caso, una de ellas debería esperar a que termine la otra.

Si un programa consulta el saldo de cualquiera de las dos cuentas involucradas en una transferencia que se está llevando a cabo, no obtendrá este saldo hasta que la transferencia haya concluido. Si se completa con éxito, obtendrá el nuevo importe, y si no, el importe antiguo, como si la transferencia nunca se hubiera intentado. Si solicita modificar el importe de cualquiera de las dos cuentas, esta modificación no se hará antes de que la transferencia haya concluido.

La sincronización de operaciones de lectura y escritura y las transacciones aseguran que los programas de aplicación siempre tengan una vista consistente, actualizada y correcta de los datos almacenados.

Una transacción es una secuencia de operaciones de lectura y actualización de datos que forman una unidad lógica y que tienen que ejecutarse como un todo. Las bases de datos relacionales proporcionan, en general, un completo y excelente soporte para transacciones, mientras que las bases de datos NoSQL no proporcionan, en general, soporte para transacciones.

Las características que debe satisfacer una transacción se suelen resumir con el acrónimo ACID (*atomic, consistent, isolated, durable*):

1. *Atomic* (atómica). Una transacción debe ejecutarse completamente y sin errores. Si ocurre algún error, los cambios que haya hecho deben deshacerse, de manera que todo quede como si nunca se hubiera iniciado la transacción.
2. *Consistent* (consistente). Las restricciones de integridad definidas para los datos deben cumplirse tras cada operación incluida en la transacción y también al finalizar esta.
3. *Isolated* (aislada). Una transacción está aislada de otras transacciones que se ejecutan simultáneamente. Las distintas transacciones tienen una visión consistente del conjunto de los datos, y no ven las modificaciones hechas por otras transacciones en curso pero no concluidas. Otra manera de expresar esto mismo es que las transacciones son serializables. En un momento dado puede haber varias transacciones ejecutándose simultáneamente. Pero el resultado final de la ejecución de todas ellas debe ser equivalente a su ejecución secuencial, una detrás de otra, en algún orden determinado, uno cualquiera de entre todos los posibles. Se entiende que alguna o varias de ellas podrían no completarse, lo que, al ser atómicas, equivale a que no se ejecuten en absoluto. Por lo demás, todas las transacciones podrán iniciarse inmediatamente, y no se pondrá ninguna restricción a la ejecución en paralelo de múltiples transacciones, siempre que no entren en conflicto. Por ejemplo, si una transacción en curso consulta un dato que ha sido modificado por otra en curso, quedará en suspenso hasta que esta última haya finalizado. Es posible que se produzca lo que se llama *interbloqueo*, es decir, que dos o más transacciones queden bloqueadas esperando cada una de ellas a que termine alguna de las otras. En este caso, el sistema devolvería un código de error a alguna o a varias de ellas y desharía sus cambios, y las demás podrían así continuar.
4. *Durable* (duradera). Una vez completadas todas las operaciones que forman la transacción (y no antes), se confirman los cambios, con lo que quedan grabados de forma permanente.

Todas las ventajas que proporcionan las transacciones tienen como contrapartida un menor rendimiento, aunque los sistemas gestores de bases de datos relacionales de hoy en día están muy

optimizados y son muy escalables, lo que significa que el rendimiento se puede mejorar añadiendo los recursos de *hardware* y *software* necesarios y realizando cambios en la configuración del sistema.

En cualquier caso, las ventajas que ofrecen las transacciones pueden no ser estrictamente necesarias y no compensar la merma en el rendimiento para algunas aplicaciones. Las bases de datos NoSQL, por ejemplo, renuncian a un soporte completo de transacciones a cambio de una mayor disponibilidad y rendimiento.

1.8. Persistencia de datos en ficheros

Los ficheros son el medio de almacenamiento más elemental. Un fichero es, en esencia, una secuencia de *bytes*, con lo que en principio puede almacenar cualquier tipo de información. Un fichero tiene un nombre y está situado en un directorio determinado dentro de una jerarquía de directorios. En última instancia, todos los sistemas de almacenamiento de datos, por sofisticados que sean, utilizan como medio de almacenamiento los ficheros.

El uso de ficheros sencillos de texto con una organización sencilla puede ser suficiente para algunas aplicaciones. Si las consultas que se vayan a realizar son complejas o requieren 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 varias aplicaciones realicen a la vez operaciones de consulta y actualización requiere elaborados mecanismos de control de acceso que añaden complejidad al sistema, y mucho más permitir transacciones.

En sistemas de este tipo se evidencian enseguida sus limitaciones intrínsecas para evitar la redundancia e inconsistencia de los datos, así como para definir y preservar restricciones de integridad. Además, no son apropiados para dar servicio a muchas aplicaciones concurrentes, como no sea que las operaciones que realicen sean, en general, solo de lectura y muy ocasionalmente de modificación de datos.

Estos son los motivos por los que, desde hace décadas, han sido relegados para la mayor parte de aplicaciones en favor de las bases de datos relacionales. Antes de ellas, la manera más habitual de almacenar la información era en ficheros indexados, para los que el lenguaje COBOL, aún hoy ampliamente utilizado en determinados ámbitos (por ejemplo, el sector bancario), ha proporcionado desde su creación, en 1959, un excelente soporte.

De todas formas, los ficheros siguen teniendo sus aplicaciones importantes. Los documentos de XML se suelen almacenar en ficheros, muchas veces agrupados en colecciones organizadas sobre una sencilla jerarquía de directorios. Los sistemas de correo electrónico suelen mantener sus datos en ficheros con un formato relativamente sencillo. Y para algunos tipos de procesos es suficiente un programa que lea secuencialmente la información almacenada en un fichero. Los ficheros secuenciales se suelen utilizar para procesos masivos que se ejecutan periódica o puntualmente, y para copias de seguridad.

1.9. Persistencia de datos en bases de datos relacionales

Los sistemas de bases de datos relacionales son, con muchísima diferencia, los más utilizados en la actualidad para el almacenamiento de datos. Su gran fortaleza viene de sus sólidos fundamentos matemáticos, al estar basados en un elegante modelo formal: el modelo relacional. Pero a la vez son sencillos e intuitivos. La estructura básica de almacenamiento que proporcionan es la tabla. Todo se almacena en tablas. Para su uso existe un lenguaje estándar y soportado universalmente: SQL.

Mediante SQL se pueden crear, de manera sencilla, esquemas relacionales, que consisten en una colección de tablas. Se pueden definir restricciones de integridad para tablas individuales y para pares de tablas relacionadas (para evitar, por ejemplo, que se borre un cliente si existen facturas para ese cliente). Los esquemas relacionales normalizados son simples y concisos, y permiten evitar redundancias e inconsistencias de datos, a la vez que se optimiza el espacio de almacenamiento utilizado.

SQL es un lenguaje declarativo, de muy alto nivel, que permite especificar la consulta que se quiere realizar para que sea el propio sistema el que decida la manera más eficiente de realizarla, y saque el máximo partido de los índices disponibles. El propio sistema crea automáticamente algunos índices de acuerdo a las restricciones de integridad definidas. Pero se pueden crear manualmente índices adicionales para optimizar determinados tipos de consultas.

Las actuales bases de datos relacionales son muy escalables, lo que significa que son apropiadas para gestionar bases de datos pequeñas con un pequeño volumen de operaciones, pero también para bases de datos enormes que soportan un continuo y elevado volumen de operaciones simultáneas de consulta y modificación de datos. Tienen un magnífico soporte para transacciones. Tienen mecanismos sofisticados de copia de seguridad y de recuperación que permiten recuperar los datos ante fallos de cualquier tipo.

Existen API para bases de datos relacionales para todos los lenguajes de programación ampliamente utilizados. Por ejemplo, JDBC para el lenguaje Java.



PARA SABER MÁS

Actualmente, las bases de datos relacionales son imbatibles como medio de almacenamiento para la mayoría de las aplicaciones. Relegaron a los sistemas basados en ficheros, y alternativas como bases de datos de objetos o de XML nunca han pasado de tener un uso muy restringido. Solo recientemente ha surgido con frecuencia la necesidad de buscar alternativas para algunas aplicaciones que requieren trabajar con gigantescos volúmenes de datos heterogéneos, lo que se conoce como *big data*. Para ello han surgido multitud de bases de datos alternativas, conocidas en general como bases de datos *NoSQL*.

1.9.1. Persistencia de datos de XML en bases de datos relacionales

XML es un formato muy utilizado actualmente para el intercambio de datos, y las principales bases de datos relacionales se han dotado de diversas funcionalidades y mecanismos para facilitar la persistencia de datos en formato XML. El estándar SQL incluye, desde SQL:2003, SQL/XML. SQL/XML incluye un tipo de datos XML. Permite, además, realizar consultas en SQL tanto sobre tablas como sobre documentos de XML, y a la vez sobre ambos, y puede obtener el resultado en forma de tabla o como XML. Las bases de datos relacionales que implementan SQL/XML se califican como *XML-enabled* o con capacidades para XML.

1.9.2. Persistencia de objetos en bases de datos relacionales

El inconveniente de utilizar el modelo relacional con lenguajes orientados a objetos es que almacenar objetos en un esquema relacional no es en principio sencillo. No desde luego cuando

se trata de objetos complejos. Un objeto complejo puede contener referencias a otros objetos y a colecciones de objetos relacionados con él. Al representar gráficamente una colección de objetos complejos, resulta evidente que tiene estructura de grafo, mientras que el modelo relacional ofrece almacenamiento basado en tablas. El término *desfase objeto-relacional* (en inglés, *object-relational impedance mismatch*), o *desajuste de impedancia objeto-relacional*, se utiliza para hacer referencia al conjunto de problemas que plantea la persistencia de objetos utilizando bases de datos relacionales.

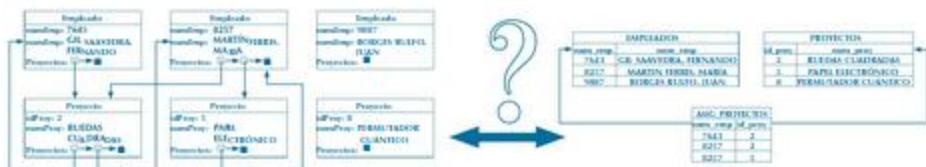


Figura 1.9
Desfase objeto-relacional

Ante esta problemática, han surgido varios planteamientos para una solución, cada uno con sus ventajas e inconvenientes:

1. *Bases de datos objeto-relacionales*. Son bases de datos relacionales con capacidades para gestionar objetos. En SQL:99 se introdujeron tipos estructurados definidos por el usuario, entre ellos los tipos objetos (clases). Las bases de datos relacionales que permiten almacenar objetos se califican como bases de datos objeto-relacionales. Entre ellas cabe destacar Oracle y PostgreSQL.
2. *Correspondencia objeto-relacional*. O, en inglés, *ORM (object-relational mapping)*, frecuentemente traducido como *mapeo objeto-relacional*. Es una solución más flexible, con la ventaja, además, de proporcionar soporte para múltiples bases de datos. Existen múltiples herramientas, bibliotecas o frameworks para ORM, entre las que cabe destacar Hibernate.

1.10. Persistencia de datos en bases de datos de objetos

Las bases de datos de objetos (o bases de datos orientadas a objetos) permiten almacenar directamente objetos. La persistencia de objetos en bases de datos de objetos es en principio la solución natural frente a soluciones de compromiso, tales como bases de datos objeto-relacionales o correspondencia objeto-relacional (ORM). Existen unas cuantas bases de datos de objetos, entre las que cabe citar Matisse y db4o. Pero hoy por hoy su uso es muy limitado.

En contraposición a la ventaja obvia de permitir almacenar directamente objetos, las bases de datos de objetos tienen como principales inconvenientes:

- La falta de un modelo formal y ampliamente aceptado en el que basarse, a diferencia de las bases de datos relacionales, basadas sobre el modelo relacional.
- La falta de estándares ampliamente adoptados, como en el caso de las bases de datos relacionales con el lenguaje SQL. El grupo ODMG (Object Database Management

Group) se disolvió en 2001 tras publicar la última versión de su estándar, ODMG 3.0. Las distintas bases de datos suelen proporcionar implementaciones limitadas del estándar ODMG, y ofrecer como alternativa prácticas lenguajes y mecanismos propios.

Las bases de datos de objetos soportan, en general, transacciones.

1.11. Persistencia de datos en bases de datos de XML nativas

La creciente importancia que ha adquirido XML como medio de representación de datos en múltiples aplicaciones ha hecho que, aparte de soluciones de compromiso como las bases de datos de XML con capacidades para XML o *XML-enabled*, se desarrollen bases de datos de XML nativas. Estas tienen estructuras y mecanismos de almacenamiento de datos diseñados y optimizados específicamente para XML, además de soporte para lenguajes estándares que permiten realizar diversos tipos de operaciones sobre documentos de XML: consulta (XPath y XQuery), actualización (diversas extensiones de XQuery) y transformación (XSL), así como validación (XML Schema y DTD). Estas bases de datos suelen proporcionar implementaciones de API estándares para bases de datos de XML, tales como XML:DB y XQJ (XQuery for Java), que son a estas bases de datos lo que JDBC es a las bases de datos relacionales.

Estas bases de datos cuentan con un amplio cuerpo de estándares sobre el que basarse, pero hay algunos aspectos para los que no existen estándares, y en los que hay importantes diferencias entre unas y otras, en particular la organización de documentos de XML en colecciones y la indexación.

Las bases de datos de XML nativas suelen organizar los documentos en colecciones. Algunas permiten una jerarquía de colecciones. Otras permiten almacenar no solo documentos de XML, sino de otros tipos.

Existen diversas extensiones de XQuery para operaciones de modificación de datos. Una de ellas, XQUF o XQuery Update Facility, es un estándar de W3C, pero surgió cuando ya existían otras propuestas anteriores, que se siguen utilizando ampliamente.

No todas las bases de datos nativas de XML soportan transacciones, ni mucho menos. El soporte para transacciones en este tipo de bases de datos plantea distintos tipos de problemas que en el caso de las bases de datos relacionales, objeto-relacionales o de objetos, porque las bases de datos de XML almacenan documentos enteros, no registros u objetos individuales.



Actividades propuestas

- 1.1.** Haz un esquema con las distintas posibilidades disponibles para almacenar tanto objetos como documentos de XML en diversos tipos de bases de datos, al menos: relacionales, de objetos, objeto-relacionales y de XML nativas. Si existe un nombre específico para la tecnología que hace posible el almacenamiento de un tipo particular de datos en un tipo particular de base de datos, indícalo.
- 1.2.** Para cada tipo de base de datos visto hasta ahora, haz una lista, por una parte, de estándares y, por otra parte, de API, e indica en pocas palabras su propósito o utilidad. Con frecuencia, una API proporciona la implementación de uno o varios estándares. Indica, siempre que se pueda y sea relevante, los estándares con los que se relaciona, se implementa o donde se basa una API. Por ejemplo: la API XQJ (XQuery for Java) es una API que permite ejecutar sentencias del lenguaje estándar XQuery.

1.12. Persistencia de datos en bases de datos NoSQL

Con frecuencia se incluyen, dentro de las bases de datos NoSQL, las bases de datos de objetos y de XML. En este libro, en general, se excluirán de esta categoría.

El auge de las bases de datos NoSQL, en la actualidad, se debe principalmente a la necesidad de recopilar, gestionar y analizar gigantescos conjuntos de datos heterogéneos, que crecen continuamente a una velocidad cada vez mayor. Esto es lo que se conoce como *big data*. A este enorme aumento de la cantidad de información contribuye el *internet de las cosas* o, por sus siglas en inglés, IoT (*internet of things*), es decir, la conexión a internet de un creciente número de dispositivos. Este tipo de bases de datos da también respuesta a las necesidades de aplicaciones que ofrecen servicio, a través de la web, a un enorme y creciente número de usuarios que no solo consultan información, sino que la añaden y la modifican de manera continua.

Las tradicionales bases de datos relacionales no pueden dar respuesta a estas necesidades, ni tampoco las de objetos o las de XML. Por ello han surgido soluciones alternativas muy diversas, conocidas en conjunto como bases de datos NoSQL.

NoSQL se interpreta a veces como *lo que no es SQL*, y a veces como *not only SQL*, es decir, *no solo SQL*, lo que no se limita a la funcionalidad que proporciona SQL. Más bien suele significar lo primero, y las bases de datos NoSQL no dan soporte para SQL ni para nada que se le parezca. En un sentido amplio, según esta interpretación, NoSQL es todo aquello que no se adhiere 100% al modelo relacional y a las características que se dan por sentadas en las bases de datos relacionales, a saber:

1. Almacenamiento basado en tablas.
2. SQL como lenguaje para interactuar con la base de datos.
3. Soporte para restricciones de integridad y transacciones.

Si NoSQL se entiende como todo lo que no es relacional, entonces incluye todo lo que no satisface la primera condición anterior, y dentro de esta categoría podrían incluirse, y a menudo se hace, las bases de objetos y de XML. Pero para estos tipos de bases de datos sí existen lenguajes estándares inspirados en SQL, y muchas veces muy similares. A saber: OQL para bases de datos de objetos y XQuery para bases de datos de XML. Y si no en tablas, sí almacenan los datos en estructuras relativamente regulares. Además, las bases de datos de objetos suelen proporcionar buen soporte para transacciones, y también algunas bases de datos de XML. Aquí se utiliza el concepto NoSQL con un sentido más restrictivo, para referirse a bases de datos que no cumplen ninguna de las tres condiciones anteriores. Cada una utiliza un tipo particular de estructuras para el almacenamiento de datos. Cada una tiene su propio lenguaje particular para consulta y modificación de datos. Y no ponen tanto énfasis en estrictas restricciones de integridad ni en todas las características de las transacciones que impone el modelo relacional, con el objetivo fundamental de primar la disponibilidad, es decir, que la base de datos pueda dar respuesta lo antes posible a peticiones de consulta y actualización de datos.

Algunas características comunes a estas bases de datos son:

- El almacenamiento no está basado en tablas. En general, utilizan estructuras de almacenamiento muy sencillas, pero, sobre todo, muy flexibles. Pueden ser *arrays* asociativos o de tipo clave-valor (como Redis), o documentos (como MongoDB), o basadas en columnas (como Casandra), o basadas en grafos, como otras bases de datos, por poner solo algunos ejemplos.

- No se manejan con SQL ni nada que se le parezca. Cada una tiene su propio lenguaje, en general no declarativo. SQL es un lenguaje declarativo, es decir, que especifica el resultado que se quiere obtener, pero no los pasos a seguir para obtenerlo. Estos lenguajes están pensados para el tipo particular de estructuras utilizadas para el almacenamiento de los datos. Para consultas o modificaciones no triviales, suele ser necesario escribir programas con instrucciones muy detalladas.
- En cuanto a restricciones de integridad y transacciones, se presta a confusión razoñar sobre estas bases de datos utilizando términos y conceptos que habitualmente se aplican al modelo relacional. En resumen, se puede decir que, en lugar de primar las características de las transacciones ACID (atomicidad, consistencia, aislamiento y durabilidad), prima la disponibilidad, lo que a veces se resume con el acrónimo BASE (*basic availability, soft state, eventual consistency*), es decir: disponibilidad básica, estado flexible y consistencia con el tiempo. *Soft state* significa que, aun sin recibir nuevas peticiones de actualización, el valor de algunos datos puede cambiar, debido a peticiones de cambios recibidos pero no llevadas a cabo. Consistencia con el tiempo significa que, pasado un periodo de tiempo, en general corto, sin que se reciban nuevas peticiones de actualización, el valor de un dato tomará un valor definitivo. Por cierto, ya que se ha contrapuesto el tradicional planteamiento ACID de los sistemas relationales al planteamiento BASE de los nuevos sistemas NoSQL, hay que recalcar que el sentido de *consistency* en ACID y en BASE (*eventual consistency*) es muy diferente.



Actividad propuesta 1.3

Haz un cuadro resumen en forma de tabla incluyendo filas para los distintos tipos de sistemas de persistencia de datos, y columnas para diversos aspectos relevantes tales como: estructura básica para almacenamiento de datos, soporte para transacciones, grado de estandarización (es decir, existencia de estándares establecidos y ampliamente adoptados), madurez (es decir, si las tecnologías se han venido utilizando durante un periodo largo de tiempo, aunque no sea de manera muy extendida, con lo que ha habido tiempo para corregir errores, perfilar características, hacer ajustes, optimizar, adaptar a nuevas situaciones), grado de implantación (uso extendido en la actualidad), y cualquier otro que consideres relevante. Se trata de dar una idea aproximada y muy sintética, con lo que para algunos de estos criterios basta indicar posibles valores como "alta", "media", "baja", o bien "siempre", "a veces".

Resumen

- Los programas de aplicación trabajan con datos transitorios almacenados en memoria principal, y con datos persistentes almacenados en medios de almacenamiento secundario.

- Existen diversos sistemas de persistencia de datos que permiten a los programas de aplicación grabar datos transitorios como datos persistentes, y recuperar datos persistentes en memoria principal como datos transitorios.
- Los sistemas de persistencia de datos suelen funcionar en servidores que proporcionan servicios a las aplicaciones a través de protocolos estándares de red. Las aplicaciones suelen utilizar estos servicios a través de API (interfaces de programación de aplicaciones).
- Hay distintos tipos de sistemas de persistencia de datos. Entre ellos los basados en ficheros, las bases de datos relacionales y objeto-relacionales, las bases de datos de objetos, las bases de datos de XML y de otros tipos, denominadas en conjunto bases de datos NoSQL. Las bases de datos de XML se suelen considerar con frecuencia como NoSQL.
- Cada tipo de sistema proporciona una estructura básica para el almacenamiento de datos: las bases de datos relacionales, tablas; las bases de datos de XML, documentos XML con estructura jerárquica (en forma de árbol); las bases de datos de objetos, objetos complejos, con referencias a otros objetos y a colecciones de objetos; las diversas bases de datos NoSQL, diversas estructuras; y los ficheros son simples secuencias de bytes.
- Las API que permiten a los programas de aplicación acceder a datos persistentes suelen proporcionar iteradores para obtener uno a uno los resultados de las consultas.
- Los sistemas de persistencia de datos suelen proporcionar control y sincronización de accesos concurrentes y transacciones. Las bases de datos relacionales suelen tener un excelente soporte para transacciones. Las bases de datos NoSQL –sin incluir entre ellas las bases de datos de objetos ni las de XML– no suelen ofrecer soporte para transacciones porque prima la disponibilidad sobre la consistencia e integridad de los datos.
- Las bases de datos relacionales son, con diferencia, las más utilizadas. Están basadas en un sólido modelo formal: el modelo relacional. Se manejan mediante SQL, un lenguaje estándar declarativo y de muy alto nivel. Suelen utilizarse incluso para almacenar objetos y documentos de XML, introducidos en sucesivas revisiones del estándar SQL.
- El almacenamiento de objetos en bases de datos relacionales plantea una serie de problemas, conocidos en conjunto como *desfase objeto-relacional*. Pero existen técnicas y herramientas de ORM o correspondencia objeto-relacional que lo hacen posible.
- Las bases de datos de objetos permiten almacenar directamente objetos complejos que contienen referencias a otros objetos y a colecciones de objetos. Muchas se basan en o dan soporte al estándar ODMG 3.0, que incluye los lenguajes ODL para definición de datos y OQL para consulta de datos.
- Las bases de datos de XML nativas suelen organizar los documentos en una jerarquía de colecciones, en la que cada una puede contener documentos XML y de otros tipos. Estas bases de datos proporcionan, además, soporte para lenguajes estándares del W3C, como XQuery, XML Schema y XSL, y para API estándares tales como XML:DB y XQJ.

7. Las bases de datos relacionales:
- a) Almacenan los datos en tablas.
 - b) Tienen, en general, un excelente soporte para transacciones.
 - c) Se manejan con el lenguaje SQL.
 - d) Todas las respuestas anteriores son correctas.
8. Las bases de datos de objetos:
- a) Permiten almacenar directamente objetos, pero no objetos complejos.
 - b) No suelen tener soporte para transacciones.
 - c) Son las únicas que permiten almacenar directamente objetos.
 - d) Ninguna de las opciones anteriores es correcta.
9. Las bases de datos de XML nativas:
- a) Incluyen soporte para colecciones de acuerdo a estándares del W3C.
 - b) Son las únicas que permiten almacenar directamente documentos de XML, a la vez que proporcionan soporte para el estándar XQuery de W3C.
 - c) Suelen tener soporte para transacciones.
 - d) No tienen un lenguaje estándar universalmente adoptado para operaciones de modificación de datos.
10. Las bases de datos NoSQL:
- a) Priman la disponibilidad sobre la consistencia e integridad de los datos.
 - b) Son importantes para las aplicaciones de *big data*.
 - c) Suelen utilizar estructuras de almacenamiento muy sencillas y flexibles.
 - d) Todas las opciones anteriores son correctas.

SOLUCIONES:

1. a b c d2. a b c d3. a b c d4. a b c d5. a b c d6. a b c d7. a b c d8. a b c d9. a b c d10. a b c d

ACTIVIDADES DE AUTOEVALUACIÓN

1. La persistencia de datos:

- a) Se utiliza para datos cuyo valor muy rara vez cambia. El resto se mantiene siempre en memoria principal.
- b) Comprende un conjunto de técnicas que permiten transferir datos de memoria principal a almacenamiento secundario y viceversa.
- c) Solo es posible para algunos tipos de datos.
- d) Permite utilizar siempre las mismas operaciones para manejar los datos, tanto si están en memoria principal como si están en almacenamiento secundario.

2. Los programas de aplicación acceden a datos persistentes:

- a) Bien directamente si están almacenados en el mismo ordenador, bien mediante API si están almacenados en otro.
- b) Mediante API, que permiten utilizar los servicios de persistencia.
- c) Mediante protocolos de red, que dan acceso a los servicios de persistencia.
- d) Previo volcado binario en memoria principal desde almacenamiento secundario.

3. Un documento de XML:

- a) Se almacena normalmente en varios directorios organizados jerárquicamente.
- b) Tiene estructura jerárquica.
- c) Al tener estructura jerárquica, no se puede almacenar en una estructura lineal como un fichero.
- d) Se puede almacenar en un fichero, pero debe ser necesariamente binario.

4. Las restricciones de integridad:

- a) Solo se pueden definir para bases de datos relacionales.
- b) Solo se pueden preservar en una base de datos utilizando transacciones.
- c) Se pueden definir para bases de datos de cualquier tipo.
- d) Se definen para garantizar la consistencia de datos relacionados entre sí.

5. Los iteradores:

- a) Permiten recuperar datos persistentes, pero no modificarlos.
- b) Solo permiten recorrer estructuras lineales o tabulares. No documentos de XML, por ejemplo.
- c) Obtiene los datos siempre uno a uno del almacenamiento secundario.
- d) Permiten obtener uno a uno los resultados de una consulta.

6. Una transacción:

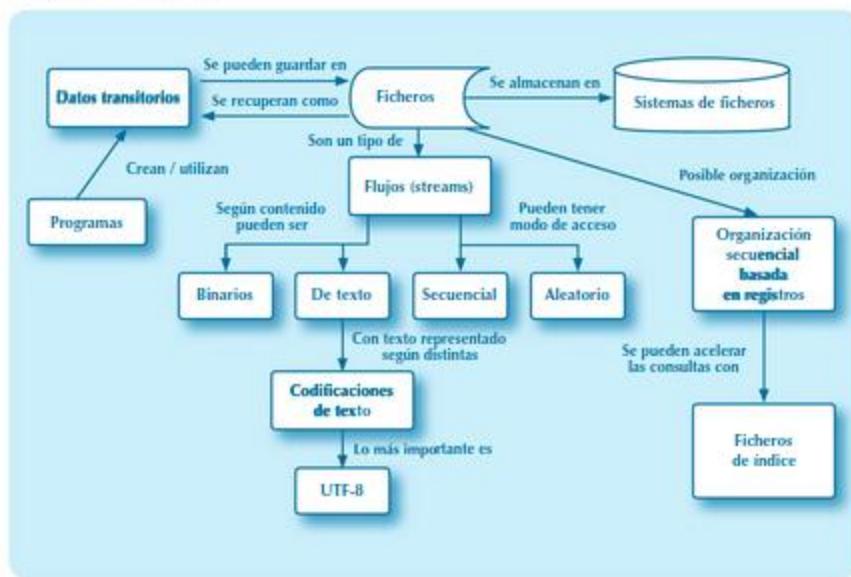
- a) Podría fallar en caso de interbloqueo, es decir, cuando esa transacción y otras estén paradas esperando a que alguna de ellas termine.
- b) Solo terminará con éxito si lo hacen todas las operaciones que incluye.
- c) Puede ejecutarse en paralelo con otras transacciones.
- d) Todas las respuestas anteriores son correctas.

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 únicamente 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.

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,



Figura 2.1
Ficheros

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.

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

Recurso web



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

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.ArithmaticException: / 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, *ArithmaticException*. 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 (ArithmaticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 0; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmaticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 3; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmaticException 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	<code>printStackTrace()</code>	Muestra información técnica muy detallada acerca de la excepción y el contexto en que se produjo. Lo hace en la salida de error, <code>System.err</code> . 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	<code>getMessage()</code>	Proporciona un mensaje detallado acerca de la excepción.
String	<code>getLocalizedMessage()</code>	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(ArithméticaException 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());
        }
    }
}
```

2.5.4. Excepciones, inicialización y liberación de recursos: bloques `finally` y `try` con recursos

Es muy frecuente que un bloque de programa de Java esté estructurado de la siguiente forma:

```
Inicialización y asignación de recursos
Cuerpo
Finalización y liberación de recursos
```

Este bloque puede estar dentro de un método de clase, como por ejemplo, el método `main()`, o de un bucle.

La primera y última parte deben ejecutarse siempre, independientemente de los errores que puedan suceder durante la ejecución del cuerpo. El bloque anterior, con gestión de excepciones, podría quedar de la siguiente manera:

```
Inicialización y asignación de recursos
try {
    Cuerpo
} catch(Excepcion_Tipo_1 e1) {
    Gestión de excepción de tipo 1
} catch(Excepcion_Tipo_2 e2) {
    Gestión de excepción de tipo 2
} catch(Exception e) {
    Gestión del resto de tipos de excepciones
}
Finalización y liberación de recursos
```

Pero es muy frecuente que, cuando sucede algún error en mitad del cuerpo, se quiera terminar inmediatamente la ejecución, bien sea con una sentencia `return` (en un método de clase), o con `break` o `continue` (dentro de un bucle, para salir de él o para saltar a la siguiente iteración). De esta manera no se ejecuta la parte final para finalización y liberación de recursos. Pero si se pone esta parte dentro de un bloque `finally {}`, se ejecutará justo antes de abandonar.

```
Inicialización y asignación de recursos
try {
    Cuerpo
} catch(Excepcion_Tipo_1 e1) {
    Gestión de excepción de tipo 1
} catch(Excepcion_Tipo_2 e2) {
    Gestión de excepción de tipo 2
} catch(Exception e) {
    Gestión del resto de tipos de excepciones
}
finally {
    Finalización y liberación de recursos
}
```

Los bloques `try` con recursos son de utilidad para simplificar la gestión de recursos de clases que implementan una de las interfaces `Closeable` o `AutoCloseable`. Para estos se invocará automáticamente el método `close()` en el bloque `finally`. Si no hay un bloque `finally`, se puede entender que existe uno vacío.

```

Inicialización y asignación de recursos
try (T1 r1=new T1(); T2 r2=new T2()) {
    // T1, T2 implementan Closeable o AutoCloseable

    Cuerpo

} catch(Excepcion_Tipo_1 e1) {
    Gestión de excepción de tipo 1
} catch(Excepcion_Tipo_2 e2) {
    Gestión de excepción de tipo 2
} catch(Exception e) {
    Gestión del resto de tipos de excepciones
}
finally {
    Finalización y liberación de recursos
    // No es necesario r1.close()
    // No es necesario r2.close()
}

```

Recurso digital



En el anexo web 2.2 encontrarás un ejemplo que simula un programa que abre para lectura dos ficheros: f1.dat y f2.dat, y que crea dos ficheros temporales: f1.info.tmp y f2.info.tmp, que borra al final.

El anexo está disponible en www.sintesis.com y es accesible con el código indicado en la primera página del libro.

2.6. Formas de acceso a los ficheros

Existen dos formas de acceder a los contenidos de los ficheros. A saber:

1. *Acceso secuencial*. Se accede comenzando desde el principio del fichero. Para llegar a cualquier parte del fichero, hay que pasar antes por todos los contenidos anteriores, empezando desde el principio del fichero.
2. *Acceso aleatorio*. Se accede directamente a los datos situados en cualquier posición del fichero.

Ambas formas de acceso se pueden utilizar para operaciones tanto de lectura como de escritura. La mayoría de los medios de almacenamiento permiten el acceso secuencial y el acceso aleatorio a los ficheros que almacenan. Pero en algunos casos podría ser posible solo el acceso secuencial. Este sería el caso si el fichero está almacenado en una cinta magnética, si bien este medio de almacenamiento está hoy en desuso. Si se considera el concepto más general de flujo de datos o *stream*, del que un fichero sería un caso particular, existen más casos en los que solo es posible el acceso secuencial. Cuando dos aplicaciones se comunican mediante una conexión de red, por ejemplo, la aplicación de origen envía los datos secuencialmente, y la de destino los recibe secuencialmente. En muchos lenguajes de programación, entre ellos Java, se pueden utilizar las mismas funciones para enviar y recibir datos por una conexión de red que para escribir y leer datos en ficheros.

2.7. Operaciones sobre ficheros con Java

Independientemente del tipo de fichero (binario o de texto) y del tipo de acceso (secuencial o aleatorio), las operaciones básicas sobre ficheros son en lo esencial iguales, y se explicarán en este apartado. En los apartados siguientes se introducirán las particularidades de las clases que proporciona Java para diversos tipos de operaciones con diversos tipos de ficheros, y las operaciones adicionales que cada una permite realizar.

El mecanismo de acceso a un fichero está basado en un puntero y en una zona de memoria que se suele llamar *buffer*. El puntero siempre apunta a un lugar del fichero, o bien a una posición especial de fin de fichero, que a veces se denomina EOF (del inglés *end of file*). Esta se puede entender que está situada inmediatamente a continuación del último *byte* del fichero. Todas las clases para operar con ficheros disponen de las siguientes operaciones básicas:

- Apertura.** Antes de hacer nada con un fichero, hay que abrirlo. Esto se hace al crear una instancia de una clase que se utilizará para operar con él.
- Lectura.** Mediante el método `read()`. Consiste en leer contenidos del fichero para volverlos a memoria y poder trabajar con ellos. El puntero se sitúa justo después del último carácter leído.
- Salto.** Mediante el método `skip()`. Consiste en hacer avanzar el puntero un número determinado de bytes o caracteres hacia delante.
- Escritura.** Mediante el método `write()`. Consiste en escribir contenidos de memoria en un lugar determinado del fichero. El puntero se sitúa justo después del último carácter escrito.
- Cierre.** Mediante el método `close()`. Para terminar, hay que cerrar el fichero.

La diferencia entre el acceso secuencial y el acceso aleatorio es que, con el último, se puede, en cualquier momento, situar el puntero en cualquier lugar del fichero. En cambio, con el acceso secuencial solo se mueve el cursor tras realizar operaciones de lectura, escritura o salto.

2.7.1. Operaciones de lectura

Cuando se lee desde el fichero, hay que indicar el *buffer*, que recibirá los datos que se lean desde él. Si no se indica el número de bytes o de caracteres para leer, se leerá hasta llenar el *buffer*. Lógicamente, no se leerá nada si el puntero apunta a la posición EOF (final del fichero).

Los ejemplos que siguen, por claridad, son para un fichero de texto en el que cada carácter ocupa un byte. Pero la operación de lectura puede realizarse para un fichero tanto binario como de texto. En el primer caso, se leerán bytes, y en el segundo, caracteres, cada uno de los cuales puede estar formado por un número variable de bytes, dependiendo del carácter y de la codificación empleada para el texto.

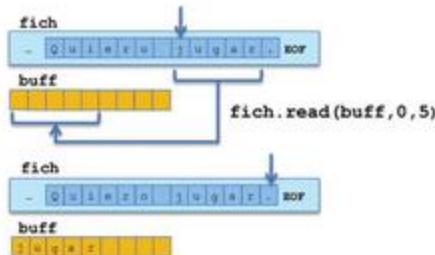


Figura 2.2
Lectura desde fichero

2.7.2. Operaciones de escritura

Cuando se escribe en el fichero, hay que indicar el *buffer* y el número de *bytes* para leer. Desde el *buffer* se transfieren los datos a la posición a la que apunta el puntero.

Si el puntero apunta al final del fichero, se añaden los datos al final de este.

Para concluir con las operaciones básicas, poco más se puede hacer que lo visto hasta ahora. No existe, por ejemplo, ninguna forma directa de insertar datos en medio de un fichero, de eliminar un fragmento de un fichero, ni de sustituir los contenidos de un fragmento de un fichero por otros, como no sea que tengan la misma longitud. Esto es así en Java y en prácticamente todos los lenguajes de programación, y se debe a que las implementaciones de los sistemas de ficheros más habituales no proporcionan ninguna manera directa para hacer operaciones de este tipo. Pero se puede hacer utilizando ficheros auxiliares, como se verá más adelante.

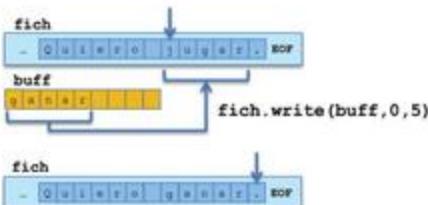


Figura 2.3

Escritura a un fichero cuando el puntero apunta en medio del fichero

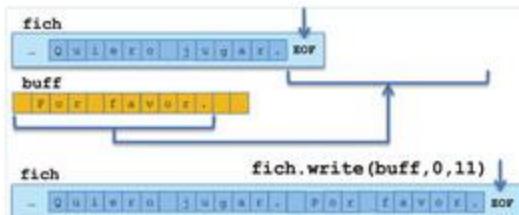


Figura 2.4

Escritura a un fichero cuando el puntero apunta al final del fichero

2.8. Acceso secuencial a ficheros en Java

Las operaciones con ficheros secuenciales en Java se realizan utilizando flujos (*streams*, en inglés) mediante clases del paquete `java.io`. Un flujo es una abstracción de alto nivel para cualquier secuencia de datos. Los ficheros secuenciales son flujos, lo mismo que los datos enviados a través de una conexión de red, o la información contenida en una secuencia de posiciones consecutivas de memoria, o la entrada y la salida estándar y de error de un programa en ejecución. Este paquete saca partido de la herencia, y proporciona clases con funcionalidad genérica de entrada y salida, común para cualquier tipo de flujo, y clases con funcionalidad específica para los distintos tipos de flujos.

Existe una jerarquía de clases derivada de cuatro clases, correspondientes a las cuatro combinaciones posibles, considerando, por una parte, flujos binarios y de texto y, por otra, flujos de entrada y de salida.

2.8.1. Clases relacionadas con flujos de datos

Para leer de un fichero o escribir en un fichero hay que crear un flujo (*stream*) asociado al mismo. El tipo de flujo para crear será distinto según se trate de un fichero binario o de texto.

Los flujos binarios no plantean mayor problema: se leen *bytes* de un flujo binario hacia una variable de tipo `byte[]`, o se escriben *bytes* de una variable de tipo `byte[]` a un flujo binario. Cuando se lee o escribe en flujos de texto, en cambio, todo es más complicado, porque entran en juego las codificaciones de texto. Todo funcionará bien mientras se trabaje con ficheros de texto con la codificación que Java asume por defecto. Normalmente será así, pero puede no serlo.

La codificación que utiliza Java por defecto para flujos de texto la toma de la configuración de la máquina virtual de Java, o en su defecto de la configuración del sistema operativo, o asume UTF-8. En la práctica, normalmente será UTF-8, que es la más habitual hoy en día. Esta codificación utiliza de uno a cuatro *bytes* para representar un carácter. Uno para todo lo que esté en el antiguo código ASCII. Dos para todo lo que no existe en inglés, como vocales acentuadas, la letra ñ, etc. Y hasta cuatro con algunos sistemas de escritura no basados en el alfabeto latino.

En cambio, para almacenar texto en memoria, Java utiliza UTF-16. Por tanto, las lecturas y escrituras con ficheros de texto suelen llevar implícito un proceso de recodificación. UTF-16 utiliza dos o cuatro *bytes* para representar un carácter o, lo que es lo mismo, uno o dos `char` (que en Java son dos *bytes*). Una cadena de texto en Java se almacena, pues, en un `array char[]` o en un `String` codificado en UTF-16 como una secuencia de caracteres, cada uno representado por dos *bytes* (normalmente) o por cuatro (con algunos sistemas de escritura no basados en el alfabeto latino).

RECUERDA

- ✓ Java codifica el texto en memoria en UTF-16, en variables de tipo `String` o `char[]`. La mayoría de los ficheros de texto están codificados en UTF-8, y normalmente Java utiliza esta codificación por defecto para leer y escribir en ellos. Por lo tanto, las operaciones de lectura o escritura de texto con ficheros de texto llevan implícito, generalmente, un proceso de recodificación.

Para lectura y escritura en flujos, Java proporciona dos jerarquías de clases: una para flujos binarios y otra para flujos de texto.



Figura 2.5
Clases para flujos binarios (de bytes)



TOMA NOTA

La clase `StringBufferInputStream` está obsoleta (`deprecated` según la terminología de Java). En su lugar habría que usar `StringReader`.

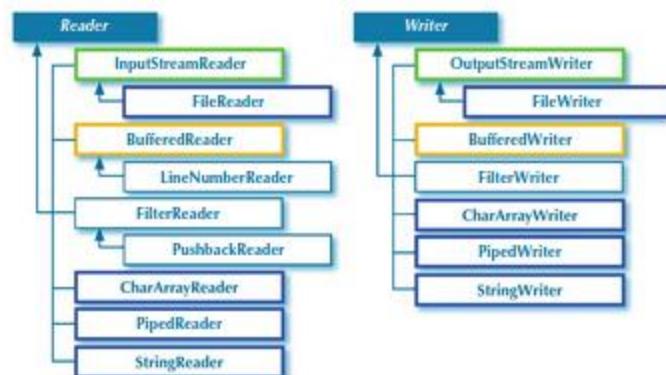


Figura 2.6
Clases para flujos de texto

En estas dos jerarquías, algunas clases ofrecen la funcionalidad básica de lectura y escritura en flujos asociados a distintos tipos de fuentes de datos. Otras ofrecen recodificación de texto. Otras ofrecen *buffering*, que permite aumentar la velocidad de las operaciones de lectura y escritura, y hace posible leer y escribir líneas de texto para los ficheros de texto. Todas estas clases se pueden componer unas sobre otras, de manera que cada una añade funcionalidad nueva sobre la proporcionada por las anteriores.

- Las clases que ofrecen la funcionalidad básica de lectura y escritura son las representadas con borde azul marino. Java usa una nomenclatura sistemática para ellas. El nombre indica primero el tipo de fuente (`File` para fichero, `ByteArray` para memoria, `Piped` para tubería). Después, si es de entrada (`Input` o `Reader`) o de salida (`Output` o `Writer`). Si acaba con `Stream` es para flujos de *bytes*, y si no, para flujos de texto.
- Las clases que ofrecen recodificación de texto son las representadas con borde verde claro.
- Las clases que ofrecen *buffering* son las representadas con borde verde oscuro.

CUADRO 2.3

Clases básicas para entrada y salida a flujos

	Fuente de datos	Lectura	Escritura
Flujo binario	Ficheros	<code>FileInputStream</code>	<code>FileOutputStream</code>
	Memoria (<code>byte[]</code>)	<code>ByteArrayInputStream</code>	<code>ByteArrayOutputStream</code>
	Tuberías	<code>PipedInputStream</code>	<code>PipedOutputStream</code>
Flujo de texto	Ficheros	<code>FileReader</code>	<code>FileWriter</code>
	Memoria (<code>char[]</code>)	<code>CharArrayReader</code>	<code>CharArrayWriter</code>
	Memoria (<code>String</code>)	<code>StringReader</code>	<code>StringWriter</code>
	Tuberías	<code>PipedReader</code>	<code>PipedWriter</code>

A continuación, se dan algunos ejemplos de cómo se crean objetos de estas clases. Para más información acerca de ellas, se puede consultar la documentación de Java.

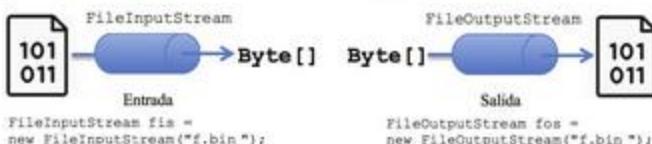


Figura 2.7. Lectura y escritura en ficheros binarios

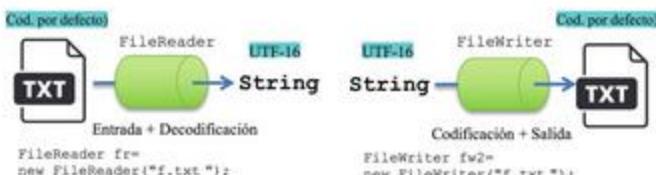


Figura 2.8. Lectura y escritura en ficheros de texto con la codificación por defecto

2.8.2. Clases para recodificación

Las clases `InputStreamReader` y `OutputStreamWriter` sirven de enlace entre ambas jerarquías: la de flujos binarios y la de flujos de texto. Dado que convierten flujos binarios en flujos de texto y viceversa, es necesario especificar una codificación a su constructor. Se pueden entender como clases que permiten recodificar texto, es decir, leer o escribir texto en ficheros con una codificación diferente a la codificación por defecto (que, como ya se ha comentado, suele ser UTF-8).

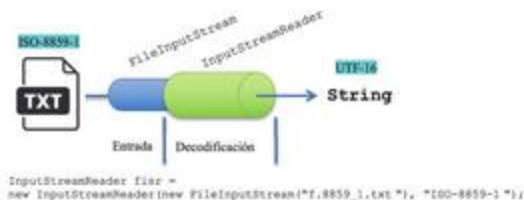


Figura 2.9.
Lectura desde
ficheros de
texto con
codificación
distinta a la
codificación
por defecto

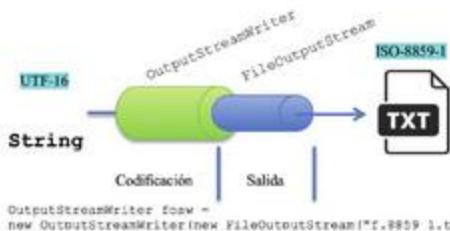


Figura 2.10.
Escritura a ficheros
de texto con
codificación
distinta a la
codificación
por defecto

2.8.3. Clases para buffering

Existen algunas clases que proporcionan *buffering* sobre el servicio proporcionado por las clases anteriores. El *buffering* es una técnica que permite acelerar las operaciones de lectura o escritura utilizando una zona de intercambio en memoria llamada *buffer*. Estas clases permiten además leer líneas de texto y escribir líneas de texto en ficheros de texto.



TOMA NOTA

En general conviene utilizar las clases que proporcionan *buffering*. El rendimiento aumenta para ficheros grandes y cuando se realizan lecturas o escrituras consecutivas en posiciones contiguas, que es lo habitual. Cuando no es así, la merma en el rendimiento no es significativa.

El *buffer* siempre representa el contenido actual de una sección del fichero. Cuando se lee de un fichero, la lectura no se limita a los datos solicitados, sino que se traen datos suficientes para llenar el *buffer*. Si la siguiente lectura del fichero es en posiciones consecutivas, lo que es muy habitual, los datos ya estarán en el *buffer*, y así se ahorra un nuevo acceso al fichero. Se puede utilizar un *buffer* de manera análoga para las operaciones de escritura. En lugar de escribir directamente en el fichero, se escribe en el *buffer*. Solo se vuelcan los contenidos del *buffer* en el fichero cuando una operación de escritura afecta a una parte del fichero fuera de la sección representada en el *buffer*, o cuando otro programa lee información de esa sección. A la actualización del fichero con los contenidos del *buffer* se le llama en inglés *flushing*.

CUADRO 2.4

Clases que proporcionan buffering para entrada y salida a flujos

	Lectura	Escritura
Flujo binario	BufferedInputStream	BufferedOutputStream
Flujo de texto	BufferedReader	BufferedWriter

Cada una de las clases anteriores proporciona *buffering* para objetos de una clase cuyo nombre viene después de `Buffered`, y además es subclase de ella, por lo que se puede usar en su lugar. Se podrían cambiar algunos de los ejemplos anteriores para utilizar *buffering*:

CUADRO 2.5

Flujo sin buffering y con buffering

Flujo sin buffering	Flujo con buffering
<code>new FileInputStream("f.bin")</code>	<code>new BufferedInputStream(new FileInputStream("f.bin"))</code>
<code>new FileOutputStream("f.bin")</code>	<code>new BufferedOutputStream(new FileOutputStream("f.bin"))</code>
	[.../...]

CUADRO 2.5 (CONT.)

<code>new FileReader("f.txt")</code>	<code>new BufferedReader(new FileReader("f.txt"))</code>
<code>new FileWriter("f.txt")</code>	<code>new BufferedWriter(new FileWriter("f.txt"))</code>

Como ya se ha dicho, las clases que proporcionan *buffering* para ficheros de texto permiten además leer y escribir líneas de texto.

CUADRO 2.6

Métodos para lectura y escritura de líneas en clases para *buffering* con ficheros de texto

Clase	Método	Funcionalidad
<code>BufferedReader</code>	<code>String readLine()</code>	Lee hasta el final de la línea actual.
<code>BufferedWriter</code>	<code>void newLine()</code>	Escribe un separador de líneas. El separador de líneas puede depender del sistema operativo, y suele ser distinto en Linux y en Windows. El método <code>readLine()</code> de <code>BufferedReader</code> tiene en cuenta estas particularidades.

2.8.4. Operaciones de lectura para flujos de entrada

Son similares para todas las clases que implementan flujos (*streams*) de entrada en Java. Las funciones `read` de clases que heredan de `InputStream` leen *bytes*, mientras que las que heredan de `Reader` leen caracteres. Según la variante, leen un *byte* o un carácter hasta llenar el *buffer* o el número de *bytes* o caracteres indicados, que se copiarán en la posición indicada (*offset*) dentro del *buffer*. Devuelven el número de *bytes* o caracteres leídos, o -1 si no se pudo leer nada porque el puntero estaba al final del fichero. Las funciones `skip` saltan el número indicado de *bytes* o caracteres, aunque podrían ser menos si se alcanza el final del fichero. En cualquier caso, devuelven el número de *bytes* o de caracteres que se han saltado.

Con ficheros de texto, para leer líneas se puede utilizar el método `readLine()` de un `BufferedReader` construido sobre un `FileReader`.

CUADRO 2.7

Métodos para lectura de las clases para gestión de flujos de entrada

InputStream	Reader
<code>int read()</code>	<code>int read()</code>
<code>int read(byte[] buffer)</code>	<code>int read(char[] buffer)</code>
<code>int read(byte[] buffer, int offset, int longitud)</code>	<code>int read(char[] buffer, int offset, int longitud)</code>
<code>long skip(long n)</code>	<code>long skip(long n)</code>
<code>BufferedReader</code>	
<code>String readLine()</code>	

Como ejemplo, el siguiente programa muestra los contenidos de un fichero de texto línea a línea, numerando las líneas. Para leer líneas de texto se usa el método `readLine()` de la clase `BufferedReader`. En este programa, y en todos a partir de ahora, se utilizarán bloques `try` con recursos para crear distintos tipos de flujos (`stream`), con lo que `close()` se ejecutará automáticamente al final.

```
// Uso de readLine() de BufferedReader
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class EscribeConNúmeroDeLineas {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Indicar por favor nombre de fichero.");
            return;
        }
        String nomFich = args[0];
        try(BufferedReader fbr = new BufferedReader(new FileReader(nomFich))) {
            int i = 0;
            String linea = fbr.readLine();
            while (linea != null) {
                System.out.format("[%5d] %s", i++, linea);
                System.out.println();
                linea = fbr.readLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("No existe fichero " + nomFich);
        } catch (IOException e) {
            System.out.println("Error de E/S: " + e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Actividades propuestas



- 2.3.** Crea un programa que busque un texto dado en un fichero de texto, y que muestre para cada aparición la línea y la columna. Se recomienda leer el fichero línea a línea y, dentro de cada línea, buscar las apariciones del texto utilizando un método apropiado de la clase `String`. Se puede consultar la documentación de dicha clase en la API de Java (<http://docs.oracle.com/javase/8/docs/api>).
- 2.4.** Crea un programa que, a partir de un fichero de texto codificado en UTF-8, genere un fichero de texto codificado en ISO-8859-1 y otro en UTF-16. El fichero codificado en UTF-8 debe crearse con un editor de texto, y debe incluir al menos vocales acentuadas. Puedes leer el fichero línea a línea con `readLine()`. Para generar el fichero de salida, puedes utilizar un `BufferedWriter` (para escribir línea a línea) construido sobre un `OutputStreamWriter` (para recodificar el texto) construido sobre un `OutputStream` (para escribir a un fichero). Busca una manera de verificar la codificación de los

ficheros de texto en el sistema operativo que estés utilizando mediante algún comando del sistema operativo o algún programa de utilidad. Puedes utilizar los programas de ejemplo para volcado binario para verificar qué caracteres se codifican de manera distinta. Puedes crear otro programa que haga la conversión inversa, para comprobar que se vuelve a obtener un fichero igual al inicial, utilizando las clases `InputStreamReader` y `FileInputStream`.

Ahora un ejemplo con flujos binarios. El siguiente programa hace un volcado binario de un fichero indicado desde línea de comandos. Los contenidos del fichero se leen en bloques de 32 bytes, y el contenido de cada bloque se escribe en una linea de texto. Los bytes se escriben en hexadecimal (base 16) y, por tanto, cada byte se escribe utilizando dos caracteres. El programa muestra como máximo los primeros 2 kilobytes (MAX_BYTES=2048). Por supuesto, este programa se puede utilizar tanto con ficheros binarios como con ficheros de texto. Hacer notar que esta clase permite hacer el volcado binario de un `InputStream`, y un `FileInputStream` es un caso particular. Siempre que sea posible, debemos hacer que las clases que desarrollemos funcionen con *streams* en general, y no solo con ficheros en particular.

```
// Volcado hexadecimal de un fichero con FileInputStream
package volcadobinario;

import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class VolcadoBinario {

    static int TAM_FILA=32;
    static int MAX_BYTES=2048;
    InputStream is=null;

    public VolcadoBinario(InputStream is) {
        this.is=is;
    }

    public void volcar() throws IOException {
        byte buffer[]=new byte[TAM_FILA];
        int bytesLeidos;
        int offset=0;
        do { bytesLeidos=is.read(buffer);
            System.out.format("[%5d]", offset);
            for(int i=0; i<bytesLeidos; i++) {
                System.out.format(" %2x", buffer[i]);
            }
            offset+=bytesLeidos;
            System.out.println();
        } while (bytesLeidos==TAM_FILA && offset<MAX_BYTES);
    }

    public static void main(String[] args) {
        if(args.length<1) {
            System.out.println("No se ha indicado ningún fichero");
            return;
        }
    }
}
```

```

        String nomFich=args[0];
        try (FileInputStream fis = new FileInputStream(nomFich)) {
            VolcadoBinario vb = new VolcadoBinario(fis);
            vb.volcar();
        }
        catch(FileNotFoundException e) {
            System.err.println("ERROR: no existe fichero "+nomFich);
        }
        catch(IOException e) {
            System.err.println("ERROR de E/S: "+e.getMessage());
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

Actividad propuesta 2.5



Modifica la clase `VolcadoBinario` para que pueda hacer el volcado a cualquier `PrintStream`, en lugar de siempre a `System.out`. Modifica el método `main()` para que realice el volcado hacia un fichero.

2.8.5. Operaciones de escritura para flujos de salida

Son similares para todas las clases que implementan flujos (*streams*) de salida en Java. Las funciones `write` de clases que heredan de `OutputStream` escriben `bytes`, mientras que las que heredan de `Writer` escriben caracteres. Según la variante, escriben un `byte` o un carácter, o todos los contenidos del `buffer`, o el número de `bytes` o caracteres indicados a partir de la posición indicada. Si se alcanza el fin del fichero, siguen escribiendo. Las que heredan de `Writer` tienen métodos para escribir los contenidos de un `String`, y métodos `append` para añadir al final del fichero.

Una característica muy útil de las clases `FileOutputStream` y `Writer` es que disponen de constructores con un parámetro que permite abrir ficheros para añadir contenidos al final (`append` en inglés). También disponen de constructores sin ese parámetro.

```

OutputStream(File file, boolean append)
OutputStream(String nombreFichero, boolean append)
Writer(File file, boolean append)
Writer(String nombreFichero, boolean append)

```

CUADRO 2.8

Métodos para escritura de clases para gestión de flujos de salida

OutputStream	Writer
<code>void write(int b)</code>	<code>void write(int c)</code>
<code>void write(byte[] buffer)</code>	<code>void write(char[] buffer)</code>
<code>void write(byte[] buffer, int offset, int longitud)</code>	<code>void write(char[] buffer, int offset, int longitud)</code>
	<code>void write(String str)</code>
	<code>void write(String str, int offset, int longitud)</code>

[.../...]

CUADRO 2.8 (CONT.)

```

        Writer append(char c)
        Writer append(CharSequence csq)
        Writer append(CharSequence csq, int offset, int
                      longitud)
    
```

BufferedWriter

```

void newLine()

```

El siguiente programa escribe un texto en un fichero. Después lo cierra y lo vuelve a abrir en modo *append* para añadir nuevos contenidos al final. A menos que el fichero ya exista, en cuyo caso no hace nada. Si se hicieran estas mismas operaciones sobre un fichero existente, se perderían los contenidos del fichero. Se añaden saltos de línea con *newLine()*.

```

// Añadir contenidos al final de un fichero de texto
package escribeenflujosalida;

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

public class EscribeEnFlujoSalida {
    public static void main(String[] args) {
        String nomFichero="f_texto.txt";
        File f=new File(nomFichero);
        if(f.exists()) {
            System.out.println("Fichero "+nomFichero+" ya existe. No se hace
                               nada");
            return;
        }
        try {
            BufferedWriter bfw=new BufferedWriter(new FileWriter(f));
            bfw.write(" Este es un fichero de texto. ");
            bfw.newLine();
            bfw.write(" quizás no está del todo bien.");
            bfw.newLine();
            bfw.close();
            bfw=new BufferedWriter(new FileWriter(f, true));
            bfw.write(" Pero se puede arreglar.");
            bfw.newLine();
            bfw.close();
        }
        catch(IOException e) {
            System.out.println(e.getMessage());
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

No es posible eliminar o reemplazar contenidos de un fichero directamente utilizando solo flujos, porque para ello es necesario leer y escribir en el mismo fichero. Se puede hacer utilizando ficheros auxiliares, que se pueden crear con `createTempFile()` de la clase `File`.

El siguiente programa realiza diversos cambios en los contenidos de un fichero de texto tales como eliminar secuencias de espacios al principio de línea, sustituir secuencias de espacios en otros lugares por un solo espacio, y hacer que todas las líneas empiecen por mayúsculas. Se puede probar con el fichero generado por el programa anterior.

Para las transformaciones en el texto se utiliza funcionalidad de la clase `Character`. Lo más importante no es entender en detalle lo que hace dentro del bucle `while` para cada línea, sino la técnica que emplea para modificar los contenidos de un fichero, leyendo de un `BufferedReader`, escribiendo en un `BufferedWriter`, utilizando ficheros temporales, y renombrando ficheros. Antes que nada, y por si acaso, se hace una copia del fichero en uno nuevo en cuyo nombre aparece una marca de tiempo incluyendo la fecha y hora exactas. Los nuevos contenidos del fichero se escriben en un fichero temporal. Al terminar, se borra el fichero original y se renombra el fichero temporal con el nombre del fichero original. El renombrado de ficheros consiste no solo en un cambio de nombre, sino también de ubicación, si se especifica un directorio distinto a aquel en que está ubicado el fichero. Es recomendable, y se puede ver que es muy sencillo, hacer que los programas realicen copias de seguridad de todos aquellos ficheros que vayan a modificar, al menos hasta que se hayan probado lo suficiente, después de lo cual se puede eliminar esta parte del programa.

```
// Cambio de contenidos de ficheros utilizando flujos de lectura y escritura y
// ficheros temporales

package arreglaficherotexto;

import java.io.File;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Date;
import java.text.SimpleDateFormat;

public class ArreglaFicheroTexto {

    public static void main(String[] args) {
        String nomFichero = "f_texto.txt";
        File f = new File(nomFichero);
        if (!f.exists()) {
            System.out.println("Fichero " + nomFichero + " no existe.");
            return;
        }
        try (BufferedReader bfr = new BufferedReader(new FileReader(f))) {
            File fTemp = File.createTempFile(nomFichero, "");
            System.out.println("Creado fich. temporal "+fTemp.
                getAbsolutePath());
            BufferedWriter bfw = new BufferedWriter(new FileWriter(fTemp));
            String linea = bfr.readLine();
            while (linea != null) { // En resumen, lee de bfr y escribe en bfw
                boolean principioLinea = true, espacios = false,
                primerAlfab=false;
```

```

        for (int i = 0; i < linea.length(); i++) {
            char c = linea.charAt(i);
            if (Character.isWhitespace(c)) {
                if (!espacios && !principioLinea) {
                    bfw.write(c);
                }
                espacios = true;
            } else if (Character.isAlphabetic(c)) {
                if(!primerAlfab) {
                    bfw.write(Character.toUpperCase(c));
                    primerAlfab=true;
                }
                else bfw.write(c);
                espacios = false;
                principioLinea = false;
            }
        }
        bfw.newLine();
        linea = bfr.readLine();
    }
    bfw.close();
    f.renameTo( // Copia de seguridad
        new File(nomFichero+
            ".+"+new SimpleDateFormat("yyyyMMddHHmmss").format(new
            Date())+".bak"
        )
    );
    fTemp.renameTo(new File(nomFichero));
} catch (IOException e) {
    System.out.println(e.getMessage());
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

Los ficheros temporales creados con `createTempFile()` normalmente se crean en un directorio especial para ficheros temporales del sistema operativo (`/tmp` en Linux). Es conveniente borrarlos al final si no se necesiten más (no es el caso en el ejemplo anterior, porque se renombra para pasar a ser un fichero definitivo). Si no se hace, debería ser el propio sistema operativo el que los elimine más adelante, por ejemplo, la próxima vez que se reinicie el sistema o pasado un tiempo. Pero eso depende del sistema operativo y de cómo esté configurado. Algunos ordenadores, como por ejemplo servidores, no se reinician durante períodos muy largos de tiempo.

2.9. Operaciones con ficheros de acceso aleatorio en Java

Para el acceso aleatorio a ficheros se utiliza la clase `RandomAccessFile`.

Las novedades fundamentales que aporta respecto a lo ya visto hasta ahora son:

1. En todo momento el cursor se puede situar en cualquier posición mediante la función `seek()`. En eso consiste el acceso aleatorio.
 2. Sobre el fichero se pueden realizar operaciones tanto de lectura como de escritura.

A pesar de estas nuevas posibilidades, las operaciones con ficheros siguen teniendo muchas limitaciones que no vienen del lenguaje Java ni del paquete `java.io`, sino de los sistemas de ficheros habituales. A saber, no es posible eliminar o insertar bloques de *bytes* o de caracteres en mitad de un fichero, ni reemplazar un bloque de un fichero por otro, a no ser que tengan exactamente el mismo tamaño en *bytes*. Para ello habrá que utilizar ficheros temporales auxiliares, de manera similar a como se ha visto en el ejemplo anterior.

En la siguiente tabla se proporciona un resumen de los principales métodos de esta clase. El resto de los métodos se puede consultar en la documentación del paquete `java.io`.

CUADRO 2.9

Principales métodos de la clase `RandomAccessFile`

Método	Funcionalidad
<code>RandomAccessFile(File file, String mode)</code> <code>RandomAccessFile(String name, String mode)</code>	Constructor. Abre el fichero en el modo indicado, si se dispone de permisos suficientes. <i>r</i> : modo de solo lectura. <i>rw</i> : modo de lectura y escritura. <i>rwd, rws</i> : Como <i>rw</i> pero con escritura síncrona. Esto significa que todas las operaciones de escritura (de datos con <i>rwd</i> y de datos y metadatos con <i>rws</i>) deben haberse completado cuando termina la función. Esto puede hacer que la llamada a la función tarde más, pero asegura que no se pierde información crítica ante una caída del sistema.
<code>void close()</code>	Cierra el fichero. Es conveniente hacerlo siempre al final.
<code>void seek(long pos)</code>	Posiciona el puntero en la posición indicada.
<code>int skipBytes(int n)</code>	Intenta avanzar el puntero el número de <i>bytes</i> indicado. Se devuelve el número de <i>bytes</i> que se ha avanzado. Podría ser menor que el solicitado, si se alcanza el fin del fichero.
<code>int read()</code> <code>int read(byte[] buffer)</code> <code>int read(byte[] buffer, int offset, int longitud)</code>	Lee del fichero. Según la variante, un <i>byte</i> o hasta llenar el <i>buffer</i> , o el número de <i>bytes</i> indicados, que se copiarán en la posición indicada (<i>offset</i>) del <i>buffer</i> . Devuelve el número de <i>bytes</i> leídos, o -1 si no se pudo leer nada porque el puntero estaba al final del fichero.
<code>void readFully(byte[] buffer)</code> <code>readFully(byte[] buffer, int offset, int longitud)</code>	Como <code>int read(byte[] b)</code> , pero si no se puede leer hasta llenar el <i>buffer</i> , o el número de <i>bytes</i> indicado, porque se llega al final del fichero, se lanza la excepción <code>IOException</code> . Útil cuando se sabe que se podrá leer hasta llenar el <i>buffer</i> o el número de <i>bytes</i> indicados. En cualquier caso, la eventualidad de que no se pueda completar la lectura se puede gestionar capturando la excepción.
<code>String readLine()</code>	Lee hasta el final de la línea de texto actual.
<code>void write(int b)</code> <code>void write(byte[] buffer)</code> <code>void write(byte[] buffer, int offset, int longitud)</code>	Escribe en el fichero. Según la variante, un <i>byte</i> , o todos los contenidos del <i>buffer</i> , o el número de <i>bytes</i> indicados a partir de la posición indicada (<i>offset</i>). Si alcanza el fin del fichero, siguen escribiendo.

La clase desarrollada en el siguiente ejemplo permite almacenar registros con datos de clientes en un fichero de acceso aleatorio. Los datos de cada cliente se almacenan en un registro, que es una estructura de longitud fija dividida en campos de longitud fija. En este caso, los campos son DNI, nombre y código postal. El constructor de la clase toma una lista con la definición del registro. Cada elemento de la lista contiene la definición de un campo en un par <nombre, longitud>. Los valores de los campos para un registro se almacenan en un `HashMap`, que contiene pares <nombre, valor>, cada uno de los cuales contiene el valor para un campo.

Al constructor se le proporciona un nombre de fichero. Si el fichero no existe, se crea. Si el fichero existe, se calcula el número de registros que contiene, dividiendo la longitud del fichero en `bytes` por la longitud de cada registro.

El método más interesante es `insertar()`. Tiene dos variantes. Si no se le indica la posición, añade el registro al final del fichero. Si no, en la posición que se le indique. La posición del primer registro es 0, no 1. Los textos se almacenan siempre codificados en UTF-8. Como es relativamente habitual en los métodos, no gestionan las excepciones que puede generar (`throws IOException`), y dejan esto para el programa principal.

```
// Almacenamiento de registros de longitud fija en fichero acceso aleatorio
package ficheroaccesoaleatorio;

import java.io.File;
import java.io.RandomAccessFile;
import java.io.IOException;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import javafx.util.Pair;

public class FicheroAccesoAleatorio {

    private File f;
    private List<Pair<String, Integer> campos;
    private long longReg;
    private long numReg = 0;

    FicheroAccesoAleatorio(String nomFich, List<Pair<String, Integer> campos)
        throws IOException {
        this.campos = campos;
        this.f = new File(nomFich);
        longReg = 0;
        for (Pair<String, Integer> campo: campos) {
            this.longReg += campo.getValue();
        }
        if(f.exists()) {
            this.numReg=f.length()/this.longReg;
        }
    }

    public longgetNumReg() {
        return numReg;
    }

    public void insertar(Map<String, String> reg) throws IOException {
        insertar(reg, this.numReg++);
    }
}
```

```

public void insertar(Map<String, String> reg, long pos) throws IOException {
    {
        try(RandomAccessFile faa = new RandomAccessFile(f, "rw")) {
            faa.seek(pos + this.longReg);
            for (Pair<String, Integer> campo: this.campos) {
                String nomCampo=campo.getKey();
                Integer longCampo = campo.getValue();
                String valorCampo = reg.get(nomCampo);
                if (valorCampo == null) {
                    valorCampo = "";
                }
                String valorCampoForm = String.format("%" + longCampo + "s",
                                                       valorCampo);
                faa.write(valorCampoForm.getBytes("UTF-8"), 0, longCampo);
            }
        }
    }

    public static void main(String[] args) {
        List campos = new ArrayList();
        campos.add(new Pair("DNI", 9));
        campos.add(new Pair("NOMBRE", 32));
        campos.add(new Pair("CP", 5));
        try {
            FicheroAccesoAleatorio faa = new FicheroAccesoAleatorio("fic_acceso_"
                + aleat.dat", campos);
            Map reg = new HashMap();
            reg.put("DNI", "56789012B");
            reg.put("NOMBRE", "SAMPER");
            reg.put("CP", "29730");
            faa.insertar(reg);
            reg.clear();
            reg.put("DNI", "89012345E");
            reg.put("NOMBRE", "ROJAS");
            faa.insertar(reg);
            reg.clear();
            reg.put("DNI", "23456789D");
            reg.put("NOMBRE", "DORCE");
            reg.put("CP", "13700");
            faa.insertar(reg);
            reg.clear();
            reg.put("DNI", "78901234X");
            reg.put("NOMBRE", "NADALES");
            reg.put("CP", "44126");
            faa.insertar(reg,1);
        } catch (IOException e) {
            System.err.println("Error de E/S: " + e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```



Actividades propuestas

- 2.6.** ¿Qué crees que pasaría si se intenta usar la clase `FicheroAccesoAleatorio` para almacenar un registro en una posición mayor que el número de registros que contiene el fichero? Compruébalo modificando el método `main()` para hacer las pruebas oportunas.
- 2.7.** Completa la clase `FicheroAccesoAleatorio` con un método que permita obtener el valor de un campo de un registro, dada la posición del registro y el nombre del campo. Por ejemplo: se podría acceder al valor del campo "Nombre" del registro situado en la posición 5 con `obtenerValorCampo(4, "Nombre")`. Por coherencia con la manera en que se ha implementado el método `insertar()`, la posición del primer registro debe entenderse que es 0, no 1. `String` debe tener el mismo formato que en la línea siguiente. Se recomienda utilizar el constructor `String(byte[] bytes, Charset charset)` de `String`.

2.10. Organizaciones de ficheros

Una organización de ficheros es una manera de organizar y estructurar los datos dentro de los ficheros, de manera que se puedan interpretar correctamente sus contenidos. El último programa de ejemplo para acceso aleatorio a ficheros almacena los datos de clientes en registros de longitud fija, compuestos por campos de longitud fija. Esta es una organización muy habitual. También es muy habitual que exista un campo, o un conjunto de campos, que identifique cada registro, de manera que no pueda existir más de un registro con los mismos valores para la clave. Se denomina *clave* a este campo o conjunto de campos. En el ejemplo anterior, en el que cada registro almacena los datos de un cliente, la clave sería el DNI del cliente.

Algunas organizaciones de ficheros pueden incluir estructuras complementarias en el propio fichero o en ficheros auxiliares. Por ejemplo, se podría incluir en el propio fichero un bloque de cabecera antes de la secuencia de campos o un bloque de cierre después. En dichos bloques se podría incluir el número de registros que contiene, la longitud de cada registro, la definición de los campos, etc. Algunas organizaciones de ficheros pueden incluir estructuras complementarias en ficheros auxiliares, como, por ejemplo, índices para acelerar las operaciones de consulta.

En definitiva, hay muchísimas posibles organizaciones de los contenidos de un fichero. Pero este apartado se centrará en dos muy sencillas, útiles y representativas: la organización secuencial y la organización secuencial indexada, basadas ambas en registros de longitud fija.

Con todo lo que se ha aprendido en este capítulo, con los ejemplos proporcionados y con la documentación de Java SE 8, se está en perfectas condiciones para desarrollar clases que implementen estas organizaciones de ficheros.

2.10.1. Organización secuencial

Los registros que forman el fichero se almacenan uno tras otro y no están ordenados de ninguna manera. No hay ningún mecanismo para localizar directamente un registro dado o para agilizar las búsquedas. La única manera es leer los registros uno a uno hasta en-



Actividades propuestas

- 2.6.** ¿Qué crees que pasaría si se intenta usar la clase `FicheroAccesoAleatorio` para almacenar un registro en una posición mayor que el número de registros que contiene el fichero? Compruébalo modificando el método `main()` para hacer las pruebas oportunas.
- 2.7.** Completa la clase `FicheroAccesoAleatorio` con un método que permita obtener el valor de un campo de un registro, dada la posición del registro y el nombre del campo. Por ejemplo: se podría acceder al valor del campo "Nombre" del registro situado en la posición 5 con `obtenerValorCampo(4, "Nombre")`. Por coherencia con la manera en que se ha implementado el método `insertar()`, la posición del primer registro debe entenderse que es 0, no 1. `String` debe tener el mismo formato que en la línea siguiente. Se recomienda utilizar el constructor `String(byte[] bytes, Charset charset)` de `String`.

2.10. Organizaciones de ficheros

Una organización de ficheros es una manera de organizar y estructurar los datos dentro de los ficheros, de manera que se puedan interpretar correctamente sus contenidos. El último programa de ejemplo para acceso aleatorio a ficheros almacena los datos de clientes en registros de longitud fija, compuestos por campos de longitud fija. Esta es una organización muy habitual. También es muy habitual que exista un campo, o un conjunto de campos, que identifique cada registro, de manera que no pueda existir más de un registro con los mismos valores para la clave. Se denomina *clave* a este campo o conjunto de campos. En el ejemplo anterior, en el que cada registro almacena los datos de un cliente, la clave sería el DNI del cliente.

Algunas organizaciones de ficheros pueden incluir estructuras complementarias en el propio fichero o en ficheros auxiliares. Por ejemplo, se podría incluir en el propio fichero un bloque de cabecera antes de la secuencia de campos o un bloque de cierre después. En dichos bloques se podría incluir el número de registros que contiene, la longitud de cada registro, la definición de los campos, etc. Algunas organizaciones de ficheros pueden incluir estructuras complementarias en ficheros auxiliares, como, por ejemplo, índices para acelerar las operaciones de consulta.

En definitiva, hay muchísimas posibles organizaciones de los contenidos de un fichero. Pero este apartado se centrará en dos muy sencillas, útiles y representativas: la organización secuencial y la organización secuencial indexada, basadas ambas en registros de longitud fija.

Con todo lo que se ha aprendido en este capítulo, con los ejemplos proporcionados y con la documentación de Java SE 8, se está en perfectas condiciones para desarrollar clases que implementen estas organizaciones de ficheros.

2.10.1. Organización secuencial

Los registros que forman el fichero se almacenan uno tras otro y no están ordenados de ninguna manera. No hay ningún mecanismo para localizar directamente un registro dado o para agilizar las búsquedas. La única manera es leer los registros uno a uno hasta en-

contrar el que se está buscando o hasta llegar al final del fichero. La figura 1.4 muestra los contenidos de un fichero con datos de clientes y con registros de longitud fija, siendo la clave el DNI.

La principal ventaja que tiene esta organización es su sencillez. Según el uso que se le vaya a dar al fichero, esto podría compensar sus inconvenientes, a saber:

- Las búsquedas son muy inefficientes. Para buscar cualquier registro, es necesario recorrer secuencialmente el fichero desde el principio hasta que se encuentre el registro que se busca o hasta llegar al final del fichero sin encontrarlo. Ordenar los registros evitaría tener que recorrer todos los ficheros cuando no está el registro que se busca. Pero ordenar por DNI no serviría de nada si se busca por nombre, por ejemplo. Además, mantener el fichero siempre ordenado complica las operaciones de inserción, borrado o modificación de registros (esto último en el caso en que se modifique el campo por el que está ordenado el fichero).
- El borrado de un registro es muy inefficiente porque obliga a correr una posición hacia atrás todos los registros siguientes. Una posible mejora podría ser marcar el registro como borrado mediante alguna marca especial. Por ejemplo, un carácter determinado en la primera posición, o dejar en blanco los campos que forman la clave, en este caso el DNI.
- La inserción de registros sería muy eficiente. Bastaría añadir el nuevo registro al final. A no ser, como ya se ha visto, que el fichero estuviera ordenado.

2.10.2. Organización secuencial indexada

Esta organización permite búsquedas muy eficientes por cualquier campo que se quiera, a cambio de crear y mantener un fichero de índice para ese campo.

Un fichero de índice no es más que un fichero secuencial ordenado cuyos registros contienen dos campos: uno para un valor del campo para el que se crea el índice, y otro que indica la posición en el fichero secuencial (véase figura 1.5).

Se pueden crear todos los índices que se quiera. Podría crearse uno por DNI, otro por nombre, o por el campo que se quiera. Podrían crearse también índices compuestos por más de un campo.

Utilizando esta organización, no es necesario reorganizar el fichero principal cuando se añaden nuevos registros o se modifican los que ya hay; solo es necesario reorganizar los índices, que son ficheros mucho más pequeños que el fichero principal. El marcar los registros como borrados en lugar de eliminarlos puede hacer innecesario reorganizar los índices cuando se elimina un registro.

El beneficio para las consultas que suponen los índices se consigue a cambio de espacio de almacenamiento y de tener que reorganizar todos los índices cuando se realizan operaciones de inserción, borrado o modificación. Hay que tenerlo en cuenta para evitar crear índices si los beneficios no compensan los inconvenientes.

Resumen

- Un fichero consiste en una secuencia de *bytes*. Un fichero se identifica por su nombre y el directorio donde está situado dentro de la jerarquía de directorios de un sistema de ficheros.
- Todos los sistemas para persistencia de datos almacenan los datos, en última instancia, en ficheros.
- Los sistemas para persistencia de datos basados en ficheros fueron relegados desde la década de los ochenta por los sistemas de bases de datos relacionales.
- Los ficheros pueden ser de texto o binarios. Los ficheros de texto contienen solo texto representado mediante una secuencia de *bytes*. Una codificación de texto es un método que permite representar un texto como una secuencia de *bytes*. La codificación de texto más ampliamente utilizada, y cada vez más, es UTF-8.
- El lenguaje de programación Java proporciona un completo soporte para operaciones con ficheros y directorios en el paquete `java.io`. Cuando se utiliza Java para operaciones con ficheros, y en general para cualquier cosa, hay que gestionar apropiadamente las excepciones que puedan producirse.
- La clase `File` permite acceder a propiedades de directorios y ficheros, así como crearlos, borrarlos, copiarlos y cambiar su ubicación dentro de la jerarquía de directorios de un sistema de ficheros.
- Hay dos formas fundamentales de acceso a ficheros: acceso secuencial y acceso aleatorio. Con acceso secuencial, para leer información en cualquier posición dentro del fichero, es necesario leer antes la información en todas las posiciones anteriores. Con acceso aleatorio es posible leer directamente la información presente en cualquier posición del fichero.
- Para el acceso secuencial en Java se utilizan *streams*. Un fichero es un tipo particular de *stream*. Java proporciona cuatro jerarquías de clases para *streams*. Corresponden a las cuatro combinaciones de valores posibles entre *streams*, por una parte, de entrada y de salida y, por otra, binarios y de texto. Para *streams* binarios las clases de origen de las jerarquías son `InputStream` y `OutputStream`, y para *streams* de texto, `Reader` y `Writer`. Las clases `BufferedInputStream` y `BufferedOutputStream` proporcionan *buffering* para *streams* binarios, lo que permite acelerar las operaciones de entrada y salida, respectivamente. Las clases `BufferedReader` y `BufferedWriter` hacen lo propio para *streams* de texto, y permiten, además, leer y escribir, respectivamente, líneas de texto. Las clases `InputStreamReader` y `OutputStreamWriter` proporcionan recodificación de texto y sirven de enlace entre las jerarquías para *streams* binarios y de texto.
- Para el acceso aleatorio a ficheros, Java proporciona la clase `RandomAccessFile`. Un `RandomAccessFile` solo proporciona operaciones para lectura y escritura de *bytes*, lo que no significa que no se puedan utilizar para el acceso aleatorio a ficheros de texto.
- Las organizaciones más sencillas para almacenar datos en ficheros están basadas en registros de longitud fija, en los que cada registro está a su vez formado por varios campos de longitud fija. En cada registro puede existir un campo clave, de manera que no pueden existir dos registros en el fichero con el mismo valor para el campo clave. La organización más sencilla es la organización secuencial. Para acelerar las búsquedas se pueden utilizar ficheros de índice externos, que permiten acceder a los registros contenidos en el fichero en un orden determinado.



Ejercicios propuestos

Los ejercicios propuestos a continuación se centran en las organizaciones de ficheros y las operaciones para cada una de ellas: inserción, borrado, modificación de ficheros, indexación y ordenación. Para la realización de estas actividades podría ser necesario utilizar clases, métodos u opciones no vistos en el capítulo, por lo que se recomienda consultar la documentación de Java SE 8 (<http://docs.oracle.com/javase/8/docs/api/>).

Nota: cuando se desarrolle una clase, debe crearse un método `main()` que pruebe suficientemente la funcionalidad de la clase. Si se piden modificaciones de una clase existente, las pruebas deberán incidir sobre la funcionalidad añadida o modificada y sobre cualquier otra funcionalidad que haya podido verse afectada por el cambio.

1. Crea una clase que implemente las operaciones para añadir, recuperar y modificar registros de un fichero con organización secuencial. Los nuevos registros se añadirán siempre al final. El fichero debe tener un campo clave, de manera que no se permita que existan dos registros con el mismo valor para el campo clave. Para recuperar un registro se proporcionará el valor de su campo clave. Para modificarlo se proporcionará el valor del campo clave para buscar el registro que se va a modificar, el nombre de un campo y el nuevo valor de ese campo. Ten en cuenta que el hecho de que el registro tenga organización secuencial no significa que no se pueda utilizar acceso aleatorio para él, si esto supone un beneficio para realizar algunas operaciones puedes implementar la clase con una estructura fija de fichero para que sea más sencillo. Por ejemplo: campos DNI y nombre de cliente, campo clave DNI.
2. Basándote en el ejercicio anterior, crea una nueva clase, más genérica, que permita definir la estructura del registro al crear un nuevo fichero, como se hace en un programa de ejemplo para ficheros de acceso aleatorio. Todo funcionará igual, solo que la clase valdrá para cualquier estructura de registro, en lugar de para una estructura fija.
3. Añade un método para borrar registros a la clase anterior. Pero los registros realmente no se borrarán, sino que se marcarán como borrados. Se puede, sencillamente, poner un carácter especial al principio del registro. Cuando se inserte un registro, se insertará, como antes, al final del fichero. Se pueden explorar otras posibilidades para marcar los registros como borrados. Por ejemplo, asignar todo espacios o asignar un valor cero a todos los bytes.
4. Añade a la clase anterior un método para compactar el fichero, es decir, para eliminar los registros marcados como borrados. Habrá que utilizar un fichero temporal para construir el nuevo fichero y, finalmente, sustituir el antiguo por el nuevo.
5. Crea una clase que implemente un fichero secuencial que contenga registros de longitud fija, compuestos por campos de longitud fija, y que siempre esté ordenado por el valor del campo clave. Cuando se inserta un nuevo registro hay que hacerlo en la posición que deba ocupar en el fichero para preservar su ordenación. Se sugiere generar el nuevo fichero en un fichero temporal y sustituir el antiguo por este al final. En el fichero temporal se irán copiando los registros del fichero hasta llegar a uno con un valor mayor para el campo clave. Entonces, se insertará el nuevo registro en el fichero temporal, y después se copiará el resto de los registros en el fichero temporal.

Por último, el fichero actual se sustituirá por el fichero temporal. En este capítulo se han visto programas de ejemplo que utilizan ficheros temporales como ayuda para modificar los contenidos de un fichero.

6. Crea una clase que implemente operaciones para añadir, buscar, borrar y modificar registros de un fichero con organización secuencial indexada. Existirá un único índice para el campo clave del fichero. El índice será un fichero adicional cuyo nombre indique el fichero principal (de datos) y el campo de indexación (que, en este caso, y según se ha dicho, es el campo clave). El índice lo creará el constructor de la clase, junto con el fichero principal. Los registros borrados no se deben borrar, sino marcarse como borrados tanto en el fichero principal como en el fichero de índice. En este último se podría poner un número negativo como posición para indicar que el registro está borrado. Los nuevos registros se añadirán siempre al final del fichero principal. La operación de inserción debe recomponer el índice, para ello hay que insertar una nueva entrada en el lugar apropiado. El índice no es más que un tipo especial de fichero secuencial ordenado, y el fichero principal un fichero secuencial no ordenado, por lo que se pueden utilizar las clases desarrolladas en ejercicios anteriores.

ACTIVIDADES DE AUTOEVALUACIÓN

1. Un fichero:
 - a) Es una secuencia *de bytes*.
 - b) Puede almacenar cualquier tipo de información
 - c) Está situado en un lugar dentro de una jerarquía de directorios en un sistema de ficheros.
 - d) Todas las respuestas anteriores son correctas.
2. La codificación de texto UTF-8:
 - a) Utiliza el mismo número *de bytes* para representar cualquier carácter.
 - b) Es compatible con ASCII.
 - c) No es una codificación de Unicode
 - d) Es compatible con ISO 8859-1.
3. Un método de clase:
 - a) No tiene permitido generar excepciones, por lo que cualquiera que el compilador detecte que podría generar debe capturarla y gestionarla.

- b) Puede generar excepciones, pero no con la sentencia `throw()`, y debe declararlo en su definición con `throws`.
- c) Debe declarar con `throws` cualquier tipo de excepción que el compilador detecte que puede generar, y por supuesto cualquiera que lance con `throw()`.
- d) Ninguna de las respuestas anteriores es correcta.
4. Se puede inicializar un recurso en la parte de inicialización de recursos de un bloque `try`:
- a) Siempre, pero si la clase del recurso no tiene definido el método `close()`, dará un error en tiempo de ejecución al cerrar el recurso.
- b) Siempre que en el bloque `finally` se le asigne valor `null`.
- c) Solo si el recurso es algún tipo de stream.
- d) Siempre que la clase del recurso implemente `Closeable` o `AutoCloseable`.
5. La clase `File`:
- a) Como su nombre indica, permite obtener información acerca de ficheros, pero no de directorios.
- b) Tiene métodos estáticos que permiten crear ficheros y directorios temporales.
- c) Permite consultar información acerca de ficheros y directorios, pero no hacer ningún cambio en ellos.
- d) Tiene un método `createNewFile` para crear ficheros y otro `createNewDir` para crear directorios.
6. Con acceso secuencial a ficheros:
- a) Solo se puede avanzar hacia delante, empezando desde el principio del fichero.
- b) Se puede avanzar y retroceder por el fichero, pero siempre *byte a byte*.
- c) No se puede acceder a cualquier posición del fichero, pero sí se puede acceder directamente al final
- d) Solo se puede acceder a medios de almacenamiento secuenciales, para el resto hay que acceder obligatoriamente con acceso aleatorio.
7. Con acceso aleatorio a ficheros:
- a) Es posible insertar un bloque de *bytes* en cualquier posición del fichero, de manera que los contenidos que vienen después se desplacen hacia delante.
- b) Es posible escribir un bloque de *bytes* en cualquier posición del fichero, pero, a menos que se haga al final, se sobrescribirá la información presente en el fichero.
- c) Es posible insertar un bloque de *bytes* al final o al principio del fichero, pero no en medio del fichero
- d) No se puede utilizar *buffering*.
8. Se puede usar `readLine()` para leer hasta el final de la línea de texto actual:
- a) Con ficheros de acceso aleatorio (clase `RandomAccessFile`).
- b) Con la clase `BufferedReader`.

- c) Con ficheros de texto con cualquier codificación.
 d) Todas las respuestas anteriores son correctas.
9. El puntero utilizado para acceso a un fichero:
- a) Puede apuntar a un byte del fichero, a una posición especial antes del primer byte o a una posición especial después del último byte.
 b) Siempre apunta a un byte del fichero.
 c) Puede apuntar a un byte de fichero o a una posición especial después del último byte.
 d) Ninguna de las anteriores respuestas es correcta.
10. La recodificación de texto al leer de o escribir a un stream:
- a) Se puede hacer con `InputStreamReader` y `OutputStreamWriter`.
 b) Solo es necesario cuando se lee de o se escribe a un fichero con una codificación que no es una codificación de Unicode.
 c) Solo es posible en Java para cuando se lee un texto con una codificación distinta de UTF-8 o se escribe en una codificación distinta de UTF-8.
 d) Solo se hace cuando el programa indica explícitamente que se haga.

SOLUCIONES:

1. a b c d
2. a b c d
3. a b c d
4. a b c d

5. a b c d
6. a b c d
7. a b c d
8. a b c d

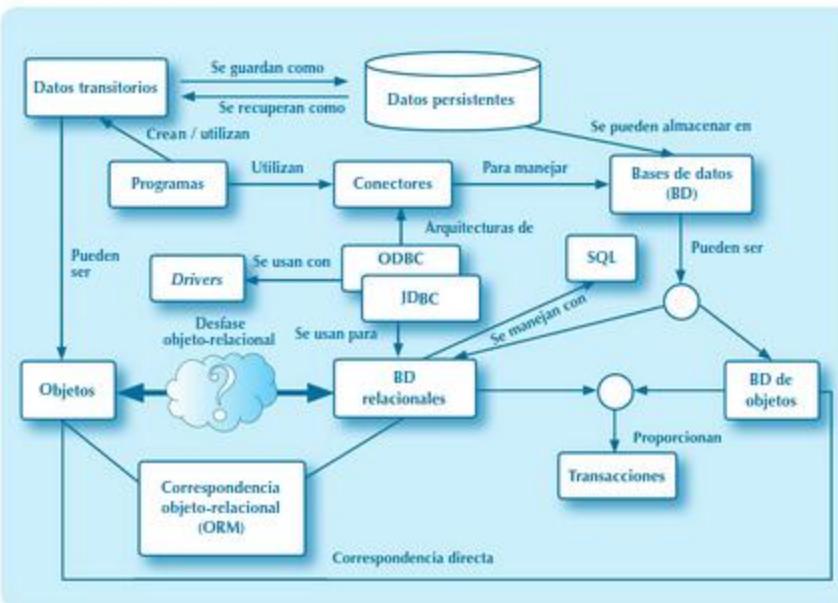
9. a b c d
10. a b c d

Bases de datos relacionales

Objetivos

- ✓ Conocer las características fundamentales de las API (*application programming interface*) que proporcionan conectores a bases de datos y, en particular, de JDBC (Java Database Connectivity) para bases de datos relacionales.
- ✓ Comprender el desfase objeto-relacional y, en particular, las dificultades para el almacenamiento de objetos complejos en estructuras de almacenamiento basadas en tablas.
- ✓ Abrir una conexión a una base de datos con JDBC utilizando un *driver* de JDBC para ella.
- ✓ Escribir programas en Java utilizando JDBC para ejecutar todo tipo de sentencias de SQL: de definición de datos (DDL) y de modificación y consulta de datos (DML).
- ✓ Utilizar sentencias preparadas para ejecutar sentencias de SQL de manera segura y eficiente.
- ✓ Trabajar con transacciones para ejecutar atómicamente un grupo de sentencias de SQL.

Mapa conceptual



Glosario

Clave autogenerada. Columna numérica de una tabla que se define como clave primaria y para la que no se especifica valor cuando se inserta una nueva fila, de manera que el propio sistema gestor de bases de datos (SGBD) le asigna automáticamente un valor.

Conector. API que permite a los programas de aplicación trabajar con bases de datos.

Desfase objeto-relacional. Conjunto de dificultades que plantea el almacenamiento de objetos complejos en bases de datos relacionales, con estructuras de almacenamiento basadas en tablas.

Driver de JDBC. Biblioteca de software que proporciona una implementación para una base de datos particular de las interfaces definidas en la especificación JDBC, de manera que permite a JDBC interactuar con esa base de datos.

Iterador. Mecanismo que permite acceder a los resultados de una consulta. Tiene operaciones para navegar por el conjunto de resultados y para recuperar los resultados uno a uno.

JDBC (Java Database Connectivity). Conector a bases de datos relacionales para Java.

Pool de conexiones. Conjunto de conexiones a una base de datos que se mantienen abiertas y a disposición de los procesos que puedan necesitarlas, lo que evita el retraso y la sobrecarga que supone la apertura de una nueva conexión cada vez que un proceso necesita una.

Procedimientos y funciones almacenados. Procedimientos y funciones escritos en un lenguaje procedural que es una extensión de SQL, y que se ejecutan en el propio SGBD.

Sentencia preparada. Sentencia de SQL parametrizada que, una vez precompilada en el SGBD, permite su ejecución de manera segura y eficiente.

Transacción. Conjunto de sentencias de SQL que forma una unidad lógica y que se ejecuta de manera atómica y aislada de otras transacciones u operaciones con la base de datos.

3.1. Conectores

Los sistemas gestores de bases de datos (SGBD) de distintos tipos (relacionales, de XML, de objetos o de otros tipos) tienen sus propios lenguajes especializados para operar con los datos que almacenan. En cambio, los programas de aplicación se escriben con lenguajes de programación de propósito general, como por ejemplo Java. Para que los programas de aplicación puedan interactuar con los SGBD, se necesitan mecanismos que permitan a los programas de aplicación comunicarse con las bases de datos en estos lenguajes. Estos se implementan en API y se denominan *conectores*.



Figura 3.1
Interacción con bases de datos utilizando conectores

3.2. Conectores para bases de datos relacionales

Los sistemas de bases de datos más utilizados hoy en día, con mucha diferencia, son los relacionales. Para trabajar con ellos se utiliza SQL. SQL es un lenguaje estándar. Pero existen multitud de bases de datos relacionales distintas, y cada una tiene su propia versión de SQL con sus propias particularidades. Aparte de eso, cada base de datos tiene sus propias interfaces de bajo nivel.

El uso de *drivers* permite desarrollar una arquitectura genérica en la que el conector tiene una interfaz común tanto para las aplicaciones como para las distintas bases de datos, y los *drivers* se ocupan de las particularidades de las distintas bases de datos.

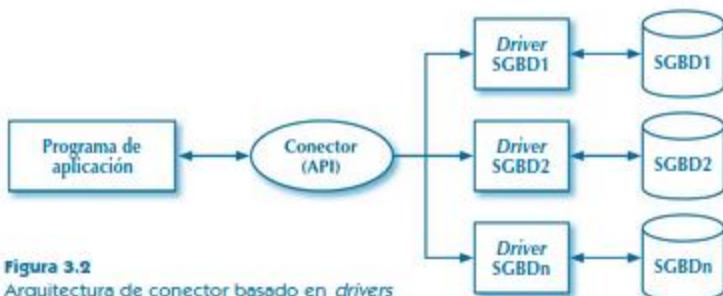


Figura 3.2
Arquitectura de conector basado en *drivers*

De esta manera, el conector no es solo una simple API, sino una arquitectura, porque especifica unas interfaces que los distintos *drivers* tienen que implementar para acceder a las bases de datos particulares. La primera arquitectura de conectores que surgió fue ODBC (Open DataBase Connectivity), desarrollada por Microsoft para Windows a principios de los noventa. ODBC es una API para el lenguaje C, y se usa ampliamente hoy en día tanto en entornos Windows como en Linux y Unix. Con el tiempo surgieron otras arquitecturas como sucesoras de ODBC, tales como OLE-DB y ADO (ActiveX Data Objects). Existen *drivers* para ellas, no solo para bases de datos relacionales, sino también para formatos de ficheros tabulares como hojas de cálculo y ficheros CSV, e incluso para fuentes de datos no relacionales como documentos XML. Estos sucesores de ODBC son API nativas para Windows, y en entornos Windows se utilizan hoy como alternativa o evolución de ODBC. Pero en entornos Linux o Unix, ODBC sigue siendo la principal solución para acceso a bases de datos relacionales desde C.



PARA SABER MÁS

A finales de los años noventa surgió JDBC como el equivalente a ODBC para Java, y es similar en muchos aspectos. De hecho, según se indica en la propia especificación de JDBC, ambos están basados en el estándar X/Open SQL CLI que especifica la manera en que un programa debe enviar sentencias de SQL a un SGBD y operar con los recordsets (conjuntos de registros o filas) obtenidos. Este estándar se definió a principios de los noventa y solo para los lenguajes C y COBOL.

Todas las bases de datos importantes proporcionan hoy en día *drivers* de ODBC y de JDBC. Existe un *driver* de JDBC para ODBC que se puede utilizar cuando no se dispone de un *driver* de JDBC específico para una base de datos, pero sí de uno de ODBC.

Existen *drivers* de JDBC no solo para bases de datos relacionales, sino para sistemas de almacenamiento basados en ficheros que almacenan los datos en forma más o menos tabular, como por ejemplo ficheros CSV (ya vistos en capítulos anteriores) y hojas de cálculo, e incluso para XML, que almacena los datos no de forma tabular, sino jerárquica.

Los beneficios que proporcionan los conectores basados en *drivers* –principalmente, independencia de la base de datos– se consiguen a cambio de una mayor complejidad y, en algunos casos, de un menor rendimiento. Existen también API que proporcionan acceso directo a determinadas bases de datos, y están ganando importancia con las aplicaciones web, cada vez más frecuentes, que se ejecutan en un entorno de servidor, que tienen una interfaz de usuario basada en HTML, y a las que se accede a través de un navegador web. Un ejemplo es el lenguaje PHP, que se ejecuta en un módulo de un navegador web Apache, y que tiene API para acceder directamente a bases de datos MySQL, todo funcionando en Linux. Este entorno se denomina con las siglas LAMP (Linux, Apache, MySQL, PHP). Para el acceso a bases de datos MySQL desde PHP se dispone de la extensión MySQLi, que proporciona una API con dos interfaces distintas: una procedural y otra orientada a objetos. También existe para PHP una API de conectores con una interfaz orientada a objetos y basada en *drivers*, llamada PDO (Portable Data Objects), con *drivers* para MySQL, Oracle y PostgreSQL, entre otras.

Recursos web



Más información acerca de las API MySQLi y PDO para PHP en la web PHP.net:

<http://php.net/manual/es/book.mysql.php>
<http://php.net/manual/es/book pdo.php>

En cualquier caso, el uso de un conector para una base de datos relacional es siempre igual en lo esencial, y conforme al estándar X/Open SQL CLI.

3.3. Acceso a resultados de consultas sobre bases de datos relacionales mediante conectores

Los conectores permiten realizar todo tipo de operaciones sobre una base de datos relacional. Pero este apartado se centrará en las operaciones de consulta.

La cuestión fundamental que se plantea con los conectores es la correspondencia entre las estructuras de datos utilizadas para el almacenamiento en la base de datos y las estructuras de datos de las que dispone el lenguaje de programación. En el caso de las bases de datos relacionales, la estructura de datos fundamental para el almacenamiento de la información es la tabla. Cada tabla tiene un conjunto fijo de columnas, cada una con un tipo de datos determinado. Una consulta de SQL devuelve un conjunto de filas o *recordset*. Los conectores permiten recuperar estos resultados fila a fila, mediante un objeto que actúa como *iterador* o *cursor*. A continuación, se muestra la manera de realizar una consulta y obtener sus resultados utilizando conectores. Se ha utilizado la sintaxis del lenguaje Java y las clases de JDBC, pero en esencia es igual para cualquier base de datos relacional y para cualquier lenguaje de programación.

```

Connection c = getConnection(datos de conexión)
Statement s = c.createStatement();
ResultSet rs = s.executeQuery("SELECT... ");
while(rs.next()) { //En rs están disponibles más resultados de la consulta
    String dato1 = rs.getString(1); // obtener String de primera columna
    int dato2 = rs.getInt(2); // obtener int de segunda columna
}
s.close();
c.close();

```

Figura 3.3

Obtención de resultados de una consulta mediante iteradores

El objeto de tipo `ResultSet` actúa como iterador sobre los resultados de la consulta, que son un conjunto de filas. Una vez recuperada una fila, se puede acceder a cada dato indicando su posición (como en el ejemplo anterior) o su nombre (como se verá más adelante).

Este modelo de acceso es válido, con algunas diferencias, para otros tipos de bases de datos, tales como bases de datos de objetos y de XML. Se trata siempre de abrir una conexión, realizar una consulta y utilizar un iterador o cursor para obtener uno a uno los resultados. Según el tipo de base de datos, habrá diferentes operaciones para hacer avanzar el cursor, y se utilizarán diferentes estructuras de datos para recuperar los resultados individuales.

3.4. Desfase objeto-relacional

Hacer a la inversa que en el apartado anterior, es decir, almacenar los resultados de variables de memoria en una base de datos relacional, puede ser más complicado. Especialmente cuando se trabaja con un lenguaje orientado a objetos y con objetos complejos, es decir, con objetos que contienen referencias a otros objetos, y referencias a colecciones de objetos relacionados. Una colección de objetos complejos tiene estructura de grafo, y no es sencillo almacenar esta información en tablas con filas y columnas. Al conjunto de dificultades que eso plantea se le conoce como *desfase objeto-relacional*, del inglés *object-relational impedance mismatch* o *desajuste de impedancia objeto-relacional* (véase figura 1.9).

Para la persistencia de objetos complejos hay dos posibilidades. Una es utilizar bases de datos de objetos, que permiten almacenar directamente objetos. La otra, utilizar técnicas o herramientas de correspondencia objeto-relacional (ORM). Ambas se verán en temas posteriores.

3.5. Java Database Connectivity

La arquitectura de conectores JDBC para Java está basada en *drivers*, como ya se ha explicado, y su API está disponible en el paquete `java.sql`. Los *drivers* de JDBC proporcionan clases que implementan las interfaces de la API JDBC para una base de datos particular. Como ya se ha comentado también, puede haber *drivers* de JDBC no para una base de datos, sino para otro conector. En particular, existe un *driver* de JDBC para ODBC que se puede utilizar si, para una base de datos, no existe un *driver* de JDBC, pero sí de ODBC.

Recurso web**www**

El siguiente enlace proporciona acceso a un sitio web de Oracle con información general, una breve introducción y tutoriales de JDBC:

<http://docs.oracle.com/javase/tutorial/jdbc/index.html>

3.6. Operaciones básicas con JDBC

En este apartado se verá en detalle cómo se pueden realizar distintos tipos de operaciones sobre bases de datos relacionales con JDBC. Se trata de ejecutar los principales tipos de sentencias de SQL. Se presupone un conocimiento básico del lenguaje SQL.

En SQL se pueden diferenciar varios sublenguajes, y a cada uno de ellos pertenecen varios tipos de sentencias. Se puede diferenciar entre DML (*data manipulation language* o lenguaje de manipulación de datos) y DDL (*data definition language* o lenguaje de definición de datos). Dentro de DML se pueden diferenciar operaciones de consulta y de modificación de datos. Las sentencias de consulta (`SELECT`) se ejecutan con `executeQuery()`, que devuelve una lista de filas en un `ResultSet`, sobre el que se puede iterar para obtener los resultados uno a uno. El resto de las sentencias de DML (`UPDATE`, `DELETE`, `INSERT`) se ejecutan con `executeUpdate()`, que devuelve el número de filas afectadas por la operación. Las sentencias de DDL se ejecutan con `execute()`. En el siguiente esquema se resume todo lo necesario para ejecutar cualquier sentencia de SQL con JDBC.

<code>Class.forName(nombre del driver); // No necesario desde JDBC 4.0 (Java SE 6)</code>		Cargar driver
<code>Connection c = DriverManager.getConnection(datos de conexión);</code>		Crear conexión
<code>Statement s = c.createStatement();</code>		Crear sentencia
<code>ResultSet rs = s.executeQuery(consulta);</code> <code>while(rs.next()) {</code> ... <code>}</code> <code>rs.close();</code>	<code>int res = s.executeUpdate(sent. DML);</code> (o bien) <code>boolean res = s.execute(sent. DDL);</code>	Ejecutar sentencia Si consulta, obtener resultados fila a fila y cerrar lista de resultados
<code>s.close();</code>		Cerrar sentencia
<code>c.close();</code>		Cerrar conexión

3.6.1. Apertura y cierre de conexiones

Los *drivers* de JDBC están disponibles en ficheros de tipo `.jar`. Se puede establecer una conexión mediante la clase `DriverManager`, con el método `getConnection(String URL conexión)`. La URL de conexión contiene un identificador del tipo de base de datos y los datos

necesarios para establecer una conexión con ella. Para MySQL, por ejemplo, tiene la forma `jdbc:mysql:host:puerto/basedatos`, donde host es localhost si está en el mismo host y puerto suele ser 3306. Este método carga automáticamente en memoria las clases para los drivers de JDBC de versión 4.0 (incluida en Java SE).

6) o posteriores disponibles. Una vez cargados, selecciona el apropiado para la base de datos indicada en la URL de conexión y le pasa esta para que establezca la conexión. Si el driver lo consigue, devuelve como resultado una `Connection` y desde entonces se encargará de todas las operaciones realizadas con ella.

El driver para MySQL 8.0, por ejemplo, lo proporciona la clase `com.mysql.cj.jdbc.Driver`.

Para añadir un driver de JDBC en un IDE como NetBeans o Eclipse, se puede añadir el fichero `.jar` al proyecto. Con el driver para MySQL 8.0 quedaría como en la figura 3.4.

Si el driver no se carga automáticamente en memoria, debe cargarlo el programa a la manera antigua:

```
Class.forName(nombre de la clase);
```

La clase que implementa el driver estará en el fichero `.jar`. En caso de duda, hay que consultar la documentación del driver, que explicará también el formato de la cadena de conexión.

El siguiente programa abre una conexión a una base de datos de MySQL y luego la cierra. El servidor de base de datos es un proceso que escucha en un puerto TCP de un host. Para la conexión hace falta indicar el servidor (`host` y puerto), el nombre de la base de datos y los datos de autenticación (usuario y contraseña). Se pueden proporcionar en la URL parámetros de conexión adicionales. En el siguiente programa se utilizan variables para todos los datos de conexión, y con ellos se compone la URL de conexión. Para algunas no se indica un valor, o se indica el habitual. Los valores exactos dependen del entorno donde se ejecute el programa.

Para abrir y cerrar los recursos necesarios y poder realizar operaciones con JDBC se utiliza un bloque `try` con recursos. Esto evita tener que cerrar explícitamente recursos con `close()`, a la vez que asegura que se cierran de manera apropiada aunque ocurra alguna excepción.

Las excepciones que se puedan producir en las operaciones con bases de datos utilizando JDBC serían generalmente de la clase `SQLException`. El método `muestraErrorSQL` muestra toda la información relativa a una `SQLException`, y se usará sin cambios en los siguientes programas de ejemplo, por lo que su código se omitirá en ellos.



Figura 3.4
Driver de MySQL en proyecto de NetBeans

```
// Apertura y cierre de conexión con JDBC
package jdbc_connection;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.SQLException;
```

```

public class JDBC_Connection {
    public static void muestraErrorSQL(SQLException e) {
        System.out.println("SQL ERROR mensaje: " + e.getMessage());
        System.out.println("SQL Estado: " + e.getSQLState());
        System.out.println("SQL código específico: " + e.getErrorCode());
    }
    public static void main(String[] args) {
        String basedatos = "...";
        String host = "localhost";
        String port = "3306";
        String parAdic = "...";
        String urlConnection = "jdbc:mysql://" + host + ":" + port + "/" + basedatos
            + parAdic;
        String user = "...";
        String pwd = "...";

        //Class.forName("com.mysql.jdbc.Driver"); // No necesario desde SE 6.0
        //Class.forName("com.mysql.cj.jdbc.Driver"); // para MySQL 8.0, no necesario

        try (Connection c = DriverManager.getConnection(urlConnection, user,
                pwd)) {
            System.out.println("Conexión realizada.");
        } catch (SQLException e) {
            muestraErrorSQL(e);
        } catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }
}

```

Recurso digital

En el anexo web 3.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás algunas recomendaciones.

3.6.2. La interfaz Statement

La interfaz **Statement** se utiliza para ejecutar cualquier tipo de sentencia SQL. Se puede obtener un **Statement** mediante el método **getStatement()** de **Connection**. En los siguientes apartados se explica cómo ejecutar distintos tipos de sentencias de SQL utilizando esta interfaz y, si se trata de una consulta, cómo recuperar los resultados en un **ResultSet**.

CUADRO 3.1
Métodos de Statement

Método	Funcionalidad
<code>ResultSet executeQuery(String sql)</code>	Ejecuta una consulta (sentencia SELECT de SQL), y devuelve un ResultSet que permite acceder a sus resultados.

[.../...]

CUADRO 3.1 (CONT.)

<code>ResultSet getResultSet()</code>	Obtiene el conjunto de resultados de una consulta SELECT y de otros tipos de sentencias que se verán más adelante, como procedimientos almacenados.
<code>int executeUpdate(String sql)</code>	Se utiliza para realizar operaciones que modifican los contenidos de la base de datos. A saber, sentencias INSERT, UPDATE y DELETE. Devuelve el número de filas afectadas.
<code>boolean execute(String sql)</code>	Se puede utilizar para ejecutar cualquier tipo de consulta. Es el método que hay que utilizar preferentemente para sentencias de DDL (sublenguaje de SQL para definición de datos), tales como CREATE, ALTER, DROP. El valor devuelto depende de la sentencia ejecutada y de sus resultados. Si se trata de una sentencia de DDL, devuelve <code>false</code> . Para ejecutar sentencias de DDL se podría utilizar también el método <code>executeUpdate</code> , y devolvería cero.
<code>void close()</code>	Cierra el Statement.

3.6.3. Ejecución de sentencias de DDL

DDL es el lenguaje de definición de datos. Incluye sentencias para crear, modificar y borrar tablas, vistas y el resto de los objetos que pueden existir en una base de datos relacional.

Las sentencias de DDL se pueden ejecutar con el método `execute()`.

Como ejemplo, el siguiente programa crea, utilizando SQL, una tabla para almacenar datos de clientes. Por supuesto, si se ejecuta una segunda vez, se producirá una excepción. No es necesario llamar al método `close()` ni de `Statement` ni de `Collection`, porque se crean en la parte de inicialización del bloque `try`.

```
// Ejecución de sentencias de DDL con execute()
package JDBC_create_table;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.SQLException;

public class JDBC_create_table {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try {
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            Statement s = c.createStatement();
            s.execute("CREATE TABLE CLIENTES (DNI CHAR(9) NOT NULL, APELLIDOS
                VARCHAR(32) NOT NULL, CP CHAR(5), PRIMARY KEY(DNI))");
            [...]
        }
    }
}
```

```
    } catch (SQLException e) {
        muestraErrorSQL(e);
    } catch (Exception e) {
        e.printStackTrace(System.err);
    }
}
```

3.6.4. Ejecución de sentencias para modificar contenidos de la base de datos

Estas sentencias se pueden ejecutar con el método `executeUpdate()`, que devolverá el número de filas afectadas por la operación, ya se trate de una sentencia `INSERT`, `UPDATE` o `DELETE`.

Como ejemplo, el siguiente programa añade varias filas con datos de clientes con una sentencia `INSERT`.

```

// Ejecución de sentencias de modificación de datos con executeUpdate()
package JDBC_insert;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.SQLException;

public class JDBC_insert {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión

        try {
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            Statement s = c.createStatement();
            int nFil = s.executeUpdate(
                "INSERT INTO CLIENTES (DNI,APELLIDOS,CP) VALUES "
                + "('78901234X','NADALES','44126'),"
                + "('89012345E','HOJAS', null),"
                + "('56789012B','SAMPER','29730'),"
                + "('09876543K','LAMIQUIZ', null');");
            System.out.println(nFil + " Filas insertadas.");
        } catch (SQLException e) {
            muestraErrorSQL(e);
            System.err.println("SQL código específico: " + e.getErrorCode());
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}

```

Haz un programa que haga los cambios necesarios para que los contenidos de la tabla CUENTES sean los siguientes: ('78901234X', 'NADALES', '44126'), ('89012345E', 'ROJAS', null), ('56789012B', 'SAMPER', '29730'), partiendo de los contenidos de la tabla resultantes de la ejecución del programa anterior. El programa debe utilizar sentencias UPDATE y DELETE.

3.6.5. Ejecución de consultas y manejo de ResultSet

Las consultas son sentencias SELECT. Se pueden ejecutar con `executeQuery()`, que devolverá un `ResultSet` con sus resultados. Un `ResultSet` contiene los resultados de una consulta como un conjunto de filas, y mantiene internamente un cursor o puntero a la fila actual. Hay métodos de `ResultSet` para obtener los datos de las distintas columnas de la fila actual, tanto por posición (por ejemplo, tercera columna) como por nombre de la columna.

`Sentence` tiene un constructor sin parámetros. Los `ResultSet` que se obtienen de las `Sentence` creadas con este constructor solo se pueden recorrer empezando por el primer resultado y avanzando al siguiente con el método `next()`. El siguiente cuadro muestra los métodos disponibles para `ResultSet` de este tipo:

CUADRO 3.2

Métodos de `ResultSet` para consultar contenidos

Método	Funcionalidad
<code>boolean next()</code>	El cursor del <code>ResultSet</code> puede apuntar a cualquier fila suya. Puede, además, apuntar también a una posición especial justo antes de la primera fila y a otra posición especial justo después de la última fila. El cursor apunta inicialmente a la posición especial de antes de la primera fila. Este método mueve el cursor a la siguiente posición y devuelve <code>true</code> , a menos que el cursor esté en la última fila o en la posición especial tras ella, en cuyo caso devuelve <code>false</code> .
<code>getXXX(int)</code> <code>getXXX(String)</code>	Obtiene el contenido de la columna especificada de la fila actual. Se puede especificar una columna por su posición o por su nombre. Hay distintas funciones para distintos tipos: <code>getInt()</code> , <code>getString()</code> , <code>getDate()</code> , etc.
<code>close()</code>	Cierra el <code>ResultSet</code> . Debería hacerse siempre.

Como ejemplo, el siguiente programa muestra los datos de todos los clientes en la tabla. No hace falta utilizar el método `close()` ni del `Statement` ni del `ResultSet`, porque se han creado en la parte de inicialización de recursos del bloque `try`.

```
// Ejecución de una consulta con executeQuery()
package JDBC_select;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JDBC_select {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try {
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            Statement s = c.createStatement();

```

```
ResultSet rs = s.executeQuery("SELECT * FROM CLIENTES");
int i=1;
while (rs.next()) {
    System.out.println("[ " + (i++) + " ]");
    System.out.println("DNI: " + rs.getString("DNI"));
    System.out.println("Apellidos: " + rs.getString("APELLODOS"));
    System.out.println("CP: " + rs.getString("CP"));
}
} catch (SQLException e) {
    muestraErrorSQL(e);
} catch (Exception e) {
    e.printStackTrace(System.err);
}
```



Actividad propuesta 3.2

El código postal (columna CP) está definido con tipo CHAR(5) en la tabla CLIENTES, pero es siempre un número entero. ¿Se podría utilizar `getInt()` en lugar de `getString()` para recuperar su valor? Cambia el programa y verifica tu hipótesis, o justifica los resultados si no son los que esperabas.

Es posible obtener `ResultSet` que permitan una mayor libertad para navegar por sus contenidos. Este tipo de `ResultSet` se llama *scrollable* (que en inglés significa *enrollable* o *desplazable*). Para ello hay que utilizar constructores para `Statement` con parámetros adicionales. Todo lo que se explica a continuación es aplicable igualmente a `PreparedStatement`, que se verá en el siguiente apartado.

CUADRO 3.3

```
Statement createStatement(int tipo, int concurrencia)
PreparedStatement prepareStatement(String sql, int tipo, int concurrencia).
```

Para el parámetro `tipo` se pueden especificar los siguientes valores:

- `ResultSet.TYPE_FORWARD_ONLY`: el tipo por defecto, utilizado hasta ahora.
 - `ResultSet.TYPE_SCROLL_INSENSITIVE`: crea un `ResultSet` de tipo *scrollable* o desplazable, y en el que no se reflejan los cambios realizados por otros procesos en la base de datos.
 - `ResultSet.TYPE_SCROLL_SENSITIVE`: crea un `ResultSet` de tipo *scrollable* o desplazable, y sensible a cambios realizados por otros procesos en la base de datos. Quiere esto decir que si algún otro proceso modifica en la base de datos algunos de los datos recuperados en el `ResultSet`, las modificaciones se reflejarán automáticamente en los contenidos del `ResultSet`.

La cuestión es que, si se quiere un `ResultSet` de tipo *scrollable*, hay que decidir si se necesita que sea sensible a cambios realizados por otros procesos en la base de datos. Si no se necesita, lo mejor es seleccionar `ResultSet.TYPE_SCROLL_INSENSITIVE`. Pero, además, hay que decidir un valor para el parámetro concurrencia. Más adelante se verán sus posibles valores. Entre tanto, se puede usar el valor `ResultSet.CONCUR_READ_ONLY`.

CUADRO 3.4

Métodos disponibles adicionalmente para consultar contenidos de `ResultSet scrollable`

Método	Funcionalidad
<code>boolean previous()</code> <code>boolean first()</code> <code>boolean last()</code> <code>void beforeFirst()</code> <code>void afterLast()</code> <code>boolean absolute(int pos)</code> <code>boolean isFirst()</code> <code>boolean isLast()</code> <code>boolean isBeforeFirst()</code> <code>boolean isAfterLast()</code> <code>int getRow()</code>	Movimiento del cursor a distintas posiciones y funciones que proporcionan información acerca de la posición del cursor. <code>previous()</code> es la reciproca de <code>next()</code> , es decir, mueve el cursor a la anterior posición y devuelve <code>true</code> , a menos que esté en la primera fila o en la posición especial anterior a ella. <code>first()</code> y <code>last()</code> mueven el cursor a la primera y última fila, y devuelven <code>true</code> a menos que no haya ninguna fila en el <code>ResultSet</code> . Los dos métodos siguientes mueven el cursor a las posiciones especiales antes de la primera fila y después de la última. <code>absolute()</code> mueve el cursor a la posición indicada. Las funciones cuyo nombre empieza por <code>is</code> indican si el cursor está en determinadas posiciones. <code>getRow()</code> devuelve la posición actual del cursor.
<code>getXXX(int)</code> <code>getXXX(String)</code>	Obtiene el contenido de la columna especificada de la fila actual, especificada por posición (<code>int</code>) o por nombre (<code>String</code>). Hay distintas funciones para distintos tipos: <code>getInt()</code> , <code>getString()</code> , <code>getDate()</code> , etc.
<code>close()</code>	Cierra el <code>ResultSet</code> . Debería hacerse siempre.

Actividades propuestas



- 3.3. Haz un programa que muestre los resultados de la misma consulta de SQL del programa anterior pero en orden inverso, del último al primero. La consulta de SQL debe ser la misma, sin ningún cambio.
- 3.4. ¿Cómo se podría averiguar el número de filas obtenidas por una consulta utilizando los métodos de `ResultSet`, pero sin un recorrer sus contenidos para contarlas? Escribe un programa que lo haga. Puedes emplear cualquier consulta.

Con lo visto hasta ahora se está en condiciones de ejecutar prácticamente cualquier sentencia de SQL y, en el caso de consultas, obtener todos sus resultados. Pero falta alguna cosa, como, por ejemplo, las funciones y procedimientos almacenados, que se verán más adelante.

En otro orden de cosas, faltan funcionalidades fundamentales de JDBC tales como sentencias preparadas, transacciones y alguna cosa más, que se verán a continuación. Por último, se verán algunas características algo más avanzadas de JDBC.

3.7. Sentencias preparadas

En los ejemplos vistos hasta ahora las sentencias de SQL eran fijas, estaban en cadenas de caracteres constantes. En una aplicación real suele ser necesario ejecutar sentencias de SQL en las que intervengan variables, bien porque dependan de valores introducidos desde la interfaz de usuario de una aplicación (por ejemplo, un formulario de consulta para clientes con diversos criterios de búsqueda, tales como DNI, nombre, etc.), bien porque dependan de un dato obtenido desde una fuente de datos (como, por ejemplo, un fichero o una consulta en SQL).

Se podría crear la sentencia en un `String`, y asignarle una expresión en la que intervengan variables. Por ejemplo: `String cons= "SELECT * FROM CLIENTES WHERE DNI= '"+dni+"'";`. Pero este planteamiento plantea graves problemas que obligan a descartarlo:

- Seguridad.* Una sentencia de SQL construida en tiempo de ejecución utilizando variables está expuesta a ataques mediante técnicas de inyección de SQL. Estas son técnicas para lograr que código SQL hábilmente introducido como parte del contenido de estas variables (`dni` en el ejemplo anterior) se ejecute. Esto permite al atacante consultar datos e incluso modificarlos y borrarlos. El riesgo es especialmente grande cuando los valores se introducen desde una interfaz de usuario, y mayor cuanto más amplia sea la posible base de usuarios. El peor escenario posible sería el acceso universal desde la web. Estos ataques se podrían evitar verificando el contenido de las variables, pero esto resulta engorroso y complejo, y no elimina completamente el riesgo.
- Rendimiento.* Una consulta que se envía al SGBD se compila antes de ejecutarse, es decir, se analiza y se crea un plan de ejecución para ella. Si entre una consulta y otra solo cambia el valor de algunas variables, se compilará cada consulta, para obtener siempre el mismo plan de ejecución.

Las sentencias preparadas permiten evitar estos problemas. Una sentencia preparada permite incluir marcadores (*placeholders* en inglés) en determinados lugares de la sentencia donde van valores que se proporcionarán en el momento de ejecutar la sentencia. Una sentencia preparada se proporciona al servidor de base de datos solo una vez, y este la prepara o precompila. A partir de ahí, solo hay que pasar un valor para cada marcador cada vez que se ejecuta la consulta.

Para hacer consultas con sentencias preparadas se utiliza `PreparedStatement`. Se obtiene un `PreparedStatement` con el método `getPreparedStatement` de `Connection`. A este método se le pasa como parámetro la sentencia SQL, y se utiliza el carácter "?" como marcador (*placeholder*).

En el cuadro 3.5 se incluyen los métodos importantes de `PreparedStatement` que no están en `Statement` o que tienen distintos parámetros.

CUADRO 3.5
Métodos de `PreparedStatement`

Método	Funcionalidad
<code>ResultSet executeQuery()</code>	Estos tres métodos no tienen como parámetro la consulta, esta se le pasó al constructor.
<code>int executeUpdate()</code>	
<code>boolean execute()</code>	
	[.../...]

CUADRO 3.5 (CONT.)

<code>setXXX(int pos, YYY valor)</code>	Se utilizan para asignar un valor a un placeholder determinado, dado por su posición, siendo 1 la posición del primero. Los hay con distintos nombres, dependiendo del tipo.
<code>setNull(int pos, int tiposql)</code>	Asigna valor NULL a una columna. Debe indicarse el tipo. Los posibles tipos están definidos en la clase <code>java.sql.Types</code> .

El siguiente programa de ejemplo inserta en una tabla CLIENTES1 los datos de tres clientes utilizando una sentencia preparada. Para la inserción de los últimos registros, se ha utilizado un planteamiento que puede ser útil para hacer frente al problema que supone, para la mantenibilidad del código, el identificar por posición los parámetros cuando hay muchos. Si más adelante hay que añadir uno, esto obliga a cambiar sentencias de Java (y podrían ser muchas) solo para incrementar un número. Se evita esto utilizando una variable que se va incrementando. La operación de posincremento `i++` en la última línea no es necesaria, pero evita el riesgo de que se añada un marcador después y se olvide añadir esta operación. Esto podría pasar, e incluso el programa podría funcionar aparentemente bien, y este error podría pasar desapercibido por un tiempo, durante el cual se introducirían datos incorrectos.

La tabla CLIENTES1 se puede crear en MySQL con: `CREATE TABLE CLIENTES1 (DNI CHAR(9) NOT NULL, APELLIDOS VARCHAR(32) NOT NULL, CP INTEGER, PRIMARY KEY(DNI));`

```
// Ejecución de varias sentencias INSERT de SQL con una sentencia preparada
package JDBC_prepared_statement;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Types;

public class JDBC_prepared_statement {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try {
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            PreparedStatement sInsert = c.prepareStatement("INSERT INTO
                CLIENTES1(DNI,APELLIDOS,CP) VALUES (?, ?, ?)");
            sInsert.setString(1, "78901234X");
            sInsert.setString(2, "NADALES");
            sInsert.setInt(3, 44126);
            sInsert.executeUpdate();
            int i = 1;
            sInsert.setString(i++, "89012345E");
            sInsert.setString(i++, "ROJAS");
            sInsert.setNull(i++, Types.INTEGER);
            sInsert.executeUpdate();
        }
    }
}
```

```

        i = 1; sInsert.setString(i++, "56789012B");
        sInsert.setString(i++, "SAMPER");
        sInsert.setInt(i++, 29730);
        sInsert.executeUpdate();
    } catch (SQLException e) {
        muestraErrorSQL(e);
    } catch (Exception e) {
        e.printStackTrace(System.err);
    }
}

```



Actividad propuesta 3.5

Escribe un programa que muestre los datos de varios clientes, o todos, de la tabla CLIENTES1. El programa debe utilizar una sentencia preparada para la consulta `SELECT * FROM CLIENTES1 WHERE DNI=?` Debe realizarse una consulta para cada cliente, especificando su DNI, y obtener los datos del `ResultSet` resultante, que solo tendrá una fila, al ser el acceso por clave primaria.

3.8. Transacciones

En este apartado se explica cómo utilizar JDBC para ejecutar varias sentencias de SQL como una transacción y gestionar los errores que se puedan producir durante su ejecución.

Una transacción es un conjunto de operaciones que se ejecutan conjuntamente como un todo, y de manera aislada de otras operaciones que pudiera realizar en paralelo otro proceso sobre los mismos datos. Las operaciones que componen una transacción se ejecutan todas completamente, con lo que todos sus cambios se confirman en la base de datos, o bien, si por cualquier motivo no se puede completar, la base de datos queda como si nunca se hubiera empezado a realizar la transacción. Las características de una transacción se resumen en inglés con el acrónimo ACID (*atomic, consistent, isolated, durable*). En el primer capítulo se explicaron en detalle todas estas características, porque se pueden aplicar a cualquier tipo de base de datos, no solo relacionales, y de hecho existen para otros tipos de bases de datos.

Las transacciones son muy importantes y se utilizan muy frecuentemente. Todos los SGBD relacionales proporcionan soporte para transacciones, pero cada uno tiene sus particularidades. Algunos, como Oracle, las tienen habilitadas por defecto, de manera que ningún cambio se confirma en la base de datos hasta que no se ejecuta una sentencia que marca el fin de una transacción (normalmente COMMIT). Otros, como MySQL, las tienen deshabilitadas por defecto, de manera que los cambios realizados por cualquier sentencia se confirman automáticamente aunque no se ejecute una sentencia COMMIT. Además, cada uno utiliza distintas sentencias, o con distinta sintaxis, para la gestión de transacciones. JDBC proporciona una interfaz común para todas las bases de datos. Por supuesto, también es posible ejecutar directamente las sentencias de SQL que controlan las transacciones, pero esto da como resultado una aplicación que solo funciona para una base de datos en particular.

Una transacción se realiza en SQL como sigue (se usa una sintaxis que no tiene por qué corresponder con la del SQL de ninguna base de datos en particular):

```

START TRANSACTION
Operación 1
Operación 2
...
COMMIT

```

Se puede abortar una transacción con la sentencia **ROLLBACK**. Con ello, se descartan todos los cambios, y todo queda como si la transacción nunca se hubiera iniciado.

La propia sentencia **COMMIT** podría fallar, aunque esto es muy inusual. Entonces se descartarían también todos los cambios, como si nunca se hubiera iniciado la transacción. También podría fallar la sentencia **ROLLBACK**.

La interfaz **Connection** tiene varios métodos para realizar estas operaciones y algunos más relacionados con transacciones. Aquí se verá lo necesario para una gestión básica de transacciones que, por otra parte, es más que suficiente en la inmensa mayoría de los casos.

CUADRO 3.6

Métodos de **connection** relacionados con transacciones

Método	Funcionalidad
<code>void setAutoCommit(boolean autoCommit)</code>	Con <code>autoCommit=false</code> inicia una transacción. En este sentido, equivale a START TRANSACTION .
<code>void commit()</code>	Equivalente a una sentencia COMMIT de SQL. Si después de utilizar este método se quiere ejecutar más sentencias de SQL pero no en una transacción, se puede hacer <code>setAutoCommit(true)</code> .
<code>void rollback()</code>	Descarta todos los cambios realizados por la transacción actual. Se hará normalmente en respuesta a cualquier excepción de tipo SQLException . Esto significa que alguna sentencia de SQL no se ha ejecutado correctamente y entonces, normalmente, se querrá estar seguro de que se aborta la transacción, descartando todos los cambios.

Un fallo durante la ejecución de una sentencia de SQL provoca una excepción de tipo **SQLException**. En ese caso, normalmente se querrá abortar la transacción con **rollback()**.

Como ejemplo se muestra un programa que ejecuta varias sentencias de SQL como una transacción. No se puede agrupar, como en ejemplos anteriores, la creación de la conexión con la creación de la sentencia preparada en el mismo bloque de inicialización de recursos de un bloque **try**, porque estos no están disponibles en el bloque **catch** correspondiente, y entonces la conexión no estaría disponible para hacer `c.rollback()`. Además, hay que hacer `c.rollback()` en un bloque **try ... catch**, porque puede lanzar una **SQLException**.

```

// Ejecución de varias sentencias en una transacción
package JDBC_transacciones;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

```

```

import java.sql.SQLException;
public class JDBC_transacciones {
    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try {
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd));
            try {
                PreparedStatement sInsert = c.prepareStatement("INSERT INTO
                    CLIENTES1(DNI,APELLIDOS,CP) VALUES (?, ?, ?);");
                c.setAutoCommit(false);

                int i = 0;
                sInsert.setString(++i, "54320198V");
                sInsert.setString(++i, "CARVAJAL");
                sInsert.setString(++i, "10109");
                sInsert.executeUpdate();

                sInsert.setString(i - 1, "765432108");
                sInsert.setString(++i, "MARQUEZ");
                sInsert.setString(++i, "46987");
                sInsert.executeUpdate();

                sInsert.setString(i - 1, "90123456A");
                sInsert.setString(++i, "MOLINA");
                sInsert.setString(++i, "35153");
                sInsert.executeUpdate();

                c.commit();
            } catch (SQLException e) {
                muestraErrorSQL(e);
                try {
                    c.rollback();
                    System.err.println("Se hace ROLLBACK");
                } catch (SQLException er) {
                    System.err.println("ERROR haciendo ROLLBACK");
                    muestraErrorSQL(er);
                }
            }
        } catch (Exception e) {
            System.err.println("ERROR de conexión");
            e.printStackTrace(System.err);
        }
    }
}

```



Actividad propuesta 3.6

Comprueba las diferencias entre el programa anterior, en el que se agrupan todas las inserciones de registros con transacciones, y otro programa igual pero sin transacciones. Primero hay que hacer una copia del programa y eliminar el código que gestiona las transacciones (`setAutoCommit`, `commit` y `rollback`). Después se trata de probar ambos programas con un mismo conjunto de datos inicial en la tabla `CLIENTES1`, para comprobar la manera distinta en que se comportan.

El conjunto de datos inicial y el programa podrían ser tales que las dos primeras inserciones se realizaran sin problemas, pero la última no, por haber ya en la tabla un cliente con ese DNI. El programa con transacciones no insertaría ningún registro. El programa sin transacciones insertaría registros hasta que se produjera un error. La creación del conjunto de datos inicial debe automatizarse. Se puede hacer mediante una secuencia de sentencias SQL, que se pueden guardar en un fichero de texto, y ejecutar con el intérprete de SQL, o mediante un programa. En esas secuencias podrían borrarse todos los contenidos de la tabla con una sentencia DELETE y añadirse varias filas con sentencias INSERT.

3.9. Valores de claves autogeneradas

Es una práctica muy habitual crear una tabla de manera que la clave primaria sea una columna numérica y se deje al sistema gestor de base de datos que proporcione un nuevo valor para cada nueva fila que se inserta. Un buen ejemplo es una aplicación que trabaja con facturas. Cada factura debe tener un identificador único, que normalmente es un simple número. El número asignado a una factura no tiene ninguna significación especial. Lo importante es que cada factura tenga un número distinto, y se deja al sistema que asigne un nuevo número consecutivo cada vez que crea una nueva factura. Las claves de este tipo se suelen llamar claves autoincrementales o, en JDBC, claves autogeneradas.

El mecanismo es esencialmente igual en casi todas las bases de datos, con pequeños cambios en la sintaxis del SQL utilizado. Oracle ha tenido históricamente una particularidad al respecto, y es el uso de secuencias para la generación de este tipo de identificadores. Las secuencias son objetos especiales que se crean en la base de datos para este fin. Pero en la versión 12 Oracle introdujo un mecanismo similar al habitual en otras bases de datos. A continuación, se muestra la definición de una tabla FACTURAS con clave autogenerada en MySQL y Oracle. Se define, además, una clave foránea sobre la columna DNI_CLIENTE para que solo se puedan crear facturas para clientes que existen en la tabla CLIENTES.

CUADRO 3.7

Definición de tabla FACTURAS con claves autogeneradas en MySQL y Oracle

```

CREATE TABLE FACTURAS(
    NUM_FACTURA INTEGER AUTO_INCREMENT
        NOT NULL,
    DNI_CLIENTE CHAR(9) NOT NULL,
    PRIMARY KEY(NUM_FACTURA),
    FOREIGN KEY FK_FACT_DNI_CLIENTES
        (DNI_CLIENTE) REFERENCES
        CLIENTES(DNI)
    );
    
```

```

CREATE TABLE FACTURAS(
    NUM_FACTURA INTEGER NOT NULL
        GENERATED BY DEFAULT AS
        IDENTITY,
    DNI_CLIENTE CHAR(9) NOT NULL,
    CONSTRAINT PK_FACTURAS PRIMARY
        KEY(NUM_FACTURA),
    CONSTRAINT FK_FACT_DNI_CLIENTES
        FOREIGN KEY(DNI_CLIENTE)
        REFERENCES CLIENTES(DNI)
    );
    
```

Aparte de la tabla FACTURAS, se utilizará en el ejemplo otra tabla LINEAS_FACTURA.

CUADRO 3.8
Definición de tabla LINEAS_FACTURA en MySQL y Oracle

MySQL	Oracle 12
<pre>CREATE TABLE LINEAS_FACTURA(NUM_FACTURA INTEGER NOT NULL, LINEA_FACTURA SMALLINT NOT NULL, CONCEPTO VARCHAR(32) NOT NULL, CANTIDAD SMALLINT NOT NULL, PRIMARY KEY(NUM_FACTURA, LINEA_FACTURA), FOREIGN KEY FK_LINEAFACT_ NUM_FACTURA(NUM_FACTURA) REFERENCES FACTURAS (NUM_FACTURA));</pre>	<pre>CREATE TABLE LINEAS_FACTURA(NUM_FACTURA INTEGER NOT NULL, LINEA_FACTURA SHORTINTEGER NOT NULL, CONCEPTO VARCHAR2(32) NOT NULL, CANTIDAD SHORTINTEGER NOT NULL, CONSTRAINT PK_LINEAS_FACTURA PRIMARY KEY(NUM_FACTURA, LINEA_FACTURA), CONSTRAINT FK_LINEAFACT_NUM_FACTURA FOREIGN KEY(NUM_FACTURA) REFERENCES FACTURAS(NUM_FACTURA));</pre>

Ahora falta saber cómo se puede recuperar, con JDBC, el valor asignado a la clave autogenerada una vez que se ha insertado una nueva fila en una tabla que tiene una clave de este tipo. Esto se puede hacer utilizando algunos métodos de `Statement` y de `Connection` (para obtener `PreparedStatement`).

CUADRO 3.9
Métodos de `Statement` (y de `PreparedStatement`) para claves autogeneradas

Método	Funcionalidad
<pre>int executeUpdate(String sql, int autoGeneratedKeys)</pre>	<p>autoGeneratedKeys puede tomar valores:</p> <ul style="list-style-type: none"> • <code>Statement.RETURN_GENERATED_KEYS</code> • <code>Statement.NO_GENERATED_KEYS</code> <p>Con el primero se pueden recuperar las claves autogeneradas como se explica a continuación. Normalmente será solo una. Este método no se puede utilizar con <code>PreparedStatement</code>. Una sentencia preparada debe crearse utilizando un método de <code>Connection</code>, como se explica más abajo</p>
<pre>ResultSet getGeneratedKeys()</pre>	<p>Devuelve un <code>ResultSet</code> con los valores de las claves autogeneradas. Normalmente será solo uno. Se puede utilizar el mismo método con <code>PreparedStatement</code>.</p>

CUADRO 3.10
Métodos de `Connection` relacionados con claves autogeneradas

Método	Funcionalidad
<pre>PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)</pre>	<p>Prepara una sentencia. Para poder recuperar valores para claves autogeneradas, hay que indicar el valor <code>PreparedStatement.RETURN_GENERATED_KEYS</code> para <code>autoGeneratedKeys</code>.</p>

El siguiente programa de ejemplo crea una factura con varias líneas. Este ejemplo funciona sin cambios en MySQL, en Oracle 12 y en otras bases de datos que dispongan de un mecanismo similar para claves autogeneradas (todas en general). Si hay alguna diferencia, está en la creación de las tablas, y eso es otra cuestión. Una vez insertada una nueva factura en FACTURAS, inserta sus líneas en LINEAS_FACTURAS, e indica como valor para el número de factura el valor para la clave autogenerada que se acaba de crear para el número de factura. La creación de la factura y de todas sus líneas se realiza conjuntamente como una transacción.

```
// Recuperación de valores para claves autogeneradas. Creación de factura.

package JDBC_claves_autogeneradas;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JDBC_claves_autogeneradas {
    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try {
            Connection c=DriverManager.getConnection(urlConnection, user, pwd));
            try {
                PreparedStatement sInsertFact = c.prepareStatement("INSERT INTO
                    FACTURAS(DNI_CLIENTE) VALUES (?)",PreparedStatement.RETURN_GENERATED_KEYS);
                PreparedStatement sInsertLineaFact = c.prepareStatement("INSERT
                    INTO LINEAS_FACTURA(NUM_FACTURA,LINEA_FACTURA,CONCEPTO,CANTIDAD)
                    VALUES (?,?,?,?)");
                c.setAutoCommit(false);

                int i = 1;
                sInsertFact.setString(i++, "78901234X");
                sInsertFact.executeUpdate();
                ResultSet rs=sInsertFact.getGeneratedKeys();
                rs.next();
                int numFact=rs.getInt(1);

                int lineaFact = 1;
                i = 1;
                sInsertLineaFact.setInt(i++, numFact);
                sInsertLineaFact.setInt(i++, lineaFact++);
                sInsertLineaFact.setString(i++, "TUERCAS");
                sInsertLineaFact.setInt(i++, 25);
                sInsertLineaFact.executeUpdate();

                i = 1;
                sInsertLineaFact.setInt(i++, numFact);
                sInsertLineaFact.setInt(i++, lineaFact++);
                sInsertLineaFact.setString(i++, "TORNILLOS");
                sInsertLineaFact.setInt(i++, 250);
                sInsertLineaFact.executeUpdate();

                c.commit();
            }
        }
    }
}
```

3.10. Llamadas a procedimientos y funciones almacenados

El estándar SQL incluye extensiones procedurales del lenguaje SQL, es decir, lenguajes de programación estructurados basados en SQL, pero que incluyen sentencias condicionales (IF), de asignación e iterativas (bucles). Los procedimientos y funciones almacenados son bloques de código escrito en un lenguaje de este tipo que pueden tener parámetros de entrada, de salida y de entrada-salida, y que, en el caso de las funciones, pueden devolver un valor. Son análogos a los procedimientos y funciones de cualquier lenguaje de programación estructurado (como Java). Se pueden invocar desde un intérprete de SQL y, como no, desde JDBC. Cada SGBD tiene su propio lenguaje de este tipo y su propia sintaxis para definir procedimientos y funciones almacenados, pero son muy similares. JDBC proporciona una interfaz única e independiente de la base de datos para utilizarlos: `CallableStatement`. En este apartado se explicará cómo utilizar esta interfaz, y se desarrollará un ejemplo completo para llamar a un procedimiento almacenado de MySQL.

El procedimiento que se va a utilizar se puede crear de la siguiente manera desde el intérprete de SQL. No es necesario entender los detalles de su implementación. Se trata de entender qué hace, cómo se le pasan valores para sus parámetros y cómo se obtienen los resultados.

```

DELIMITER //
CREATE PROCEDURE listado_parcial_clientes
(IN in_dni CHAR(9), INOUT inout_long INT)
BEGIN
    DECLARE apell VARCHAR(32) DEFAULT NULL;
    SELECT APELLIDOS FROM CLIENTES WHERE DNI=in_dni INTO apell;
    SET inout_long = inout_long + LENGTH(apell);
    SELECT DNI, APELLIDOS FROM CLIENTES
        WHERE APELLIDOS<=apell ORDER BY APELLIDOS;
END //
DELIMITER;

```

A este procedimiento se le pasa un DNI en el parámetro de entrada `in_dni`, y devuelve un conjunto de filas con DNI y APELLIDOS de los clientes por orden alfabético de APELLIDOS, hasta el valor de APELLIDOS para el cliente con el DNI proporcionado. Al valor pasado en el parámetro de entrada y salida `inout long` se le suma la longitud del valor de APELLIDOS para el cliente con el

DNI pasado en `in_dni`. Este ejemplo tiene todo lo que puede tener un procedimiento almacenado: parámetros de entrada y de entrada-salida, y que devuelva una lista de filas. Lo único que podría faltarle sería un parámetro de solo salida, pero esto no supondría ninguna complicación adicional. Se puede utilizar este procedimiento desde el intérprete de SQL de la siguiente manera:

```
SET @long=0;
CALL listado_parcial_clientes('78901234X', @long);
SELECT @long;
```

Primero, se asigna un valor a la variable `@long`, que se pasa como parámetro de entrada y salida `inout_long`. La llamada al procedimiento con `CALL` muestra las filas devueltas por el procedimiento, y la sentencia `SELECT` del final muestra el valor devuelto en `@long`.

La secuencia de pasos para utilizar el procedimiento con JDBC es la misma, como se verá en breve en un programa de ejemplo. Lo primero es obtener un `CallableStatement` con el método `prepareCall(String patron_llamada)` de `Connection`. `patron_llamada` incluye el nombre del procedimiento o de la función almacenada. Su sintaxis tiene variantes según se trate de un procedimiento o de una función, y según tenga o no parámetros.

CUADRO 3.11

Sintaxis para `patron_llamada` en método `prepareCall`
(`String patron_llamada`) de `CallableStatement`

	Con parámetros	Sin parámetros
Procedimiento	{ call procedimiento(?, ?,...) }	{ call procedimiento }
Función	{ ? = call función(?, ?,...) }	{ call función }

Los métodos de `CallableStatement` que permiten realizar la llamada y obtener los resultados se detallan en el cuadro 3.12.

CUADRO 3.12

Métodos de `CallableStatement` para uso de procedimientos y funciones almacenados

Método	Funcionalidad
<code>setXXX(int pos, YYYY valor)</code>	De manera similar a <code>PreparedStatement</code> , se utilizan para asignar un valor de diversos tipos (las hay con distintos nombres dependiendo del tipo) a un parámetro determinado, identificado por su posición, siendo 1 el primero.
<code>void registerOutParameter(int pos, int sqlType)</code>	Registra un parámetro de salida para poder obtener el valor devuelto en él.
<code>getXXX(int pos)</code>	Obtiene el valor de un parámetro de salida.
<code>ResultSet getResultSet()</code>	Obtiene el <code>ResultSet</code> resultado del procedimiento o función.

En este programa de ejemplo se llama al procedimiento almacenado anterior.

```
// Llamada a procedimiento almacenado en base de datos de MySQL
package JDBC_callable_statement;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.CallableStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class callable_statement {
    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try {
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            CallableStatement s =
                c.prepareCall("{call listado_parcial_clientes(?,?)}");
            s.setString(1, "78901234X");
            s.setInt(2, 0);
            s.registerOutParameter(2, java.sql.Types.INTEGER);
            s.execute();
            ResultSet rs = s.getResultSet();
            int inout_long = s.getInt(2);
            System.out.println(">> inout_long: "+inout_long);
            int nCli=0;
            while (rs.next()) {
                System.out.println("[ " + (++nCli) + " ]");
                System.out.println("DNI: " + rs.getString("DNI"));
                System.out.println("Apellidos: " + rs.getString("APELLODOS"));
            }
        } catch (SQLException e) {
            muestraErrorSQL(e);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.7

Crea una función almacenada con MySQL (o con Oracle o con otra base de datos) a la que se le pase el DNI de un cliente y que devuelva sus apellidos. Crea un programa en Java que realice una llamada a ella utilizando JDBC y escriba el valor devuelto por la función. Será necesario consultar documentación o investigar cómo crear funciones almacenadas en la base de datos utilizada, pero se hará de manera muy similar a un procedimiento almacenado.

3.11. Actualizaciones sobre los resultados de una consulta

La interfaz `ResultSet` no solo tiene métodos para consultar los contenidos de una base de datos, sino también para modificar esos contenidos. Con ellos se pueden eliminar, insertar y modificar filas en una tabla. Pero para ello es necesario crear `ResultSet` actualizables que permitan estas operaciones, y especificar algunos parámetros adicionales al crearlos.

Los `ResultSet` actualizables no se utilizan con mucha frecuencia. Normalmente, para hacer cambios en las bases de datos, se emplean sentencias de SQL `UPDATE`, `DELETE` e `INSERT`, y con `UPDATE` y `DELETE` las filas se seleccionan con una cláusula `WHERE`. Pueden ser útiles cuando las condiciones para seleccionar las filas para actualizar o borrar, o los cálculos para los nuevos valores que asignar a los atributos son muy complejos, o si no es posible, o es muy difícil, seleccionar las filas con SQL. Pero casi siempre es posible —y recomendable— utilizar sentencias de SQL separadas para modificaciones en bases de datos relacionales.

CUADRO 3.13

Métodos de `Connection` para obtener `Statement` y `PreparedStatement` actualizables

```
Statement createStatement(int tipo, int concurrencia)
PreparedStatement prepareStatement(String sql, int tipo, int concurrencia)
```

El parámetro `tipo` permite obtener un `ResultSet` de tipo `scrollable`, como ya se ha visto en un apartado previo, en el que se dejaron para más adelante las explicaciones sobre el parámetro `concurrencia`. Pues bien, este es el momento. Sus posibles valores son:

- `ResultSet.CONCUR_READ_ONLY`: el `ResultSet` no es actualizable, es decir, no permite que los cambios realizados en él se graben en la base de datos.
- `ResultSet.CONCUR_UPDATABLE`: el `ResultSet` es actualizable, es decir, los cambios realizados en él se pueden grabar en la base de datos.

En el cuadro 3.14 se muestran los métodos disponibles para los `ResultSet` actualizables.

CUADRO 3.14

Métodos disponibles para actualizaciones con `ResultSet` actualizables

Método	Funcionalidad
<code>void updateXXX(int pos, YYY valor)</code> <code>void updateXXX(String nombre, YYY valor)</code>	Asigna el valor indicado a la columna indicada, que se puede identificar, bien por posición, bien por nombre. XXX es un tipo de JDBC, YYY es un tipo de Java.
<code>void updateNull(int pos)</code> <code>void updateNull(String nombre)</code>	Asigna valor NULL a la columna indicada.
<code>void updateRow()</code>	Graba en la base de datos los contenidos de la fila actual del <code>ResultSet</code> .
<code>deleteRow()</code>	Borra de la base de datos la fila actual del <code>ResultSet</code> .
[.../...]	

CUADRO 3.14 (CONT.)

<code>moveToInsertRow()</code>	Mueve el cursor a una posición especial, la fila de inserción (<i>insert row</i>), que sirve para guardar los contenidos de una nueva fila que insertar en la base de datos. Una vez ejecutado este método, se asignan valores a las columnas con <code>updateXXX()</code> y finalmente se inserta la fila en la base de datos con <code>insertRow()</code> . Una vez hecho esto, se puede volver a la fila anterior con <code>moveToCurrentRow()</code> .
<code>void insertRow()</code>	Inserta los contenidos de la fila de inserción en el <code>ResultSet</code> y en la base de datos.

El siguiente programa de ejemplo utiliza las funciones anteriores para modificar el código postal del último cliente recuperado por una consulta, borrar el penúltimo cliente e insertar un nuevo cliente, todo dentro de una transacción.

```
// Modificación de contenidos de una tabla con un ResultSet actualizable
package JDBC_ResultSet_actualizable;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBC_ResultSet_actualizable {
    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try {
            Connection c=DriverManager.getConnection(urlConnection, user, pwd));
            try {
                Statement sConsulta = c.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
                ResultSet rs = sConsulta.executeQuery(
                    "SELECT * FROM CLIENTES WHERE CP IS NOT NULL");
                c.setAutoCommit(false);
                rs.last(); // Modifica último cliente
                rs.updateString("CP", "02568");
                rs.updateRow();
                rs.previous(); // Borra penúltimo cliente
                rs.deleteRow();
                rs.moveToInsertRow(); // Inserta nuevo cliente
                rs.updateString("DNI", "24262486S");
                rs.updateString("APELIDOS", "ZURITA");
                rs.updateString("CP", "33983");
                rs.insertRow();
                c.commit();
            }
        }
    }
}
```

```
        } catch (SQLException e) {
            muestraErrorSQL(e);
            try {
                c.rollback();
                System.err.println("Se hace ROLLBACK");
            } catch (Exception er) {
                System.err.println("ERROR haciendo ROLLBACK");
                er.printStackTrace(System.err);
            }
        }
    } catch (Exception e) {
        System.err.println("ERROR de conexión");
        e.printStackTrace(System.err);
    }
}
```

Actividad propuesta 3.8



¿Existe alguna diferencia entre `updateXXX(1, null)` y `updateNull(1)`?

3.12. Ejecución de *scripts*

Un *script* de SQL es una secuencia de sentencias de SQL separadas por el carácter “;”. Para poder ejecutar *scripts* con MySQL debe establecerse la conexión indicando en la URL de conexión la opción `allowMultiQueries=true`. Con ello se puede ejecutar un *script*, utilizando un **Statement**, exactamente igual que una sentencia de SQL individual. Los *scripts* de SQL son muy utilizados para la instalación de aplicaciones, para crear y poner a punto la base de datos con la que trabajan. Normalmente se prepara en un fichero de texto una secuencia de sentencias de SQL que crean una base de datos, y dentro de ella tablas, vistas y en general todos los objetos necesarios, y añaden algunos datos básicos en determinadas tablas.

3.13. Ejecución de sentencias por lotes

Para ejecutar un número muy grande de sentencias de SQL, puede ser conveniente hacerlo por lotes. Un lote es un conjunto de sentencias de SQL que se envían todas juntas al servidor de bases de datos y se ejecutan todas juntas, en lugar de enviar y ejecutar una a una.

No hay métodos para crear un lote, se crean automáticamente al ejecutar sentencias con el método `addBatch(String sql)` de `Statement` y `addBatch()` de `PreparedStatement`. En un lote creado para `Statement` las sentencias pueden ser de distinto tipo. En un lote creado para `PreparedStatement` son del mismo, porque el lote se crea con una sentencia preparada, pero cada ejecución se hace con un conjunto de parámetros distinto. El lote se ejecuta con `executeBatch()`, que devuelve un `array` de tipo `int[]` con el número de filas afectadas por cada sentencia del lote. Existe también un método `executeLargeBatch()` que devuelve un `array` de tipo `long[]`. También `CallableStatement` admite ejecución por lotes.

El siguiente programa crea un lote para una `PreparedStatement` de tipo `INSERT`, añade varias sentencias y lo ejecuta, todo dentro de una transacción. Se obtienen los datos de un *array* para ilustrar el hecho de que, normalmente, los datos se obtendrán de una fuente de datos mediante un bucle, dado que la ejecución por lotes tiene sentido cuando se agrupa un número grande de sentencias en el lote.

```
// Ejecución de un lote de sentencias preparadas
package JDBC_prepared_statement_en_lote;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBC_prepared_statement_en_lote {
    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión

        String[][] datosClientes = {
            {"13579135G", "MOYA", null},
            {"24680246G", "SILVA", "25865"},
            {"96307418R", "TORRES", "19273"}
        }
        try {
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd));
        try {
            PreparedStatement sInsert = c.prepareStatement("INSERT INTO
                CLIENTES(DNI,APELLIDOS,CP) VALUES (?, ?, ?)");
            c.setAutoCommit(false);
            for (int nCli = 0; nCli < datosClientes.length; nCli++) {
                for (int i = 0; i < datosClientes[nCli].length; i++) {
                    sInsert.setString(i + 1, datosClientes[nCli][i]);
                }
                sInsert.addBatch();
            }
            sInsert.executeBatch();
            c.commit();
        } catch (SQLException e) {
            muestraErrorSQL(e);
        } try {
            c.rollback();
        } catch (Exception er) {
            System.err.println("ERROR haciendo ROLLBACK");
            er.printStackTrace(System.err);
        }
    }
} catch (Exception e) {
    System.err.println("ERROR de conexión");
    e.printStackTrace(System.err);
}
}
```

Actividad propuesta 3.9



No todas las bases de datos proporcionan soporte para lotes. Se puede saber con `soportaLotes = c.getMetaData().supportsBatchUpdates()`, siendo `c` una `Connection`. Crea un programa a partir de una copia del programa anterior que compruebe si la base de datos proporciona soporte para lotes y escriba esta información en pantalla. Si es el caso, agrupará los cambios en un lote, como hace el programa anterior. Si no, ejecutará directamente las sentencias preparadas. Si tu base de datos tiene soporte para lotes, prueba que funciona bien en ambos casos, por ejemplo, añadiendo solo para una prueba: `soportaLotes = false`.



Recurso digital

En el anexo web 3.2, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás información sobre *pool de conexiones*.

Resumen

- Un conector es una API que permite acceder, desde programas de aplicación, a los datos almacenados en SGBD.
- Cada tipo de base de datos tiene su propio mecanismo para almacenamiento de los datos y su propio lenguaje para trabajar con ella. Los conectores suelen ser para un tipo particular de bases de datos (relacional, de objetos, de XML). Pero tienen características comunes. Para empezar, proporcionan métodos para abrir una conexión a una base de datos. Una vez hecho esto, permiten ejecutar sentencias en el lenguaje propio de la base de datos (SQL en el caso de las bases de datos relacionales). Proporcionan iteradores para poder acceder a los resultados de una consulta. En concreto, para navegar por el conjunto de resultados y para obtener los datos de cada resultado individual. En el caso concreto de las bases de datos relacionales, las consultas devuelven un conjunto de filas, y los iteradores permiten navegar por el conjunto de filas y obtener los datos de cada fila particular. Si la base de datos proporciona soporte para transacciones, también proporcionan métodos para gestionarlas.
- La API JDBC para Java permite trabajar con bases de datos relacionales. Es más que una API, es una arquitectura, dado que proporciona una interfaz común para los programas de aplicaciones y para el acceso a bases de datos, y requiere *drivers* específicos para el acceso a las bases de datos particulares. Un *driver* hace posible la conexión a una base de datos particular utilizando sus propios protocolos de red e interfaces de bajo nivel, y proporciona la implementación para esa base de datos de todas las interfaces recogidas en la especificación JDBC.

- JDBC permite ejecutar cualquier sentencia de SQL mediante `execute`, `executeUpdate` y `executeQuery`. En el caso de consultas, permite obtener los resultados en un `ResultSet`.
- En el caso de que en la consulta intervengan variables, deben utilizarse siempre sentencias preparadas o `PreparedStatement` por seguridad (para evitar ataques de inyección de SQL). Deben utilizarse también, por eficiencia, siempre que se ejecuten muchas sentencias que son iguales salvo que cambien determinados valores de una a otra.
- JDBC permite ejecutar un conjunto de sentencias de SQL en una transacción de manera sencilla e independiente de la base de datos, y utiliza métodos estándares de JDBC en lugar de sentencias de SQL distintas según la base de datos.
- JDBC permite invocar procedimientos y funciones almacenados de manera independiente de la base de datos, utiliza métodos estándares, y hace abstracción de las diferencias entre distintas bases de datos en lo referente a procedimientos y funciones almacenados.
- JDBC permite ejecutar grupos de sentencias como *scripts*, la base de datos lo permite.
- JDBC permite ejecutar por lotes sentencias de SQL y sentencias de SQL preparadas, si la base de datos lo permite.

Ejercicios propuestos



Hay que utilizar transacciones para las modificaciones de datos siempre que sea apropiado. Para la realización de estos ejercicios puede ser necesario, como en los capítulos anteriores, consultar la documentación de Java SE 8 (<https://docs.oracle.com/javase/8/docs/api/>).

1. Haz un programa que permita navegar de forma interactiva por los contenidos de la tabla CLIENTES creada con un programa de ejemplo anterior. Primero, el programa debe realizar una consulta para obtener los contenidos de la tabla, y debe mostrar el mensaje "fila 1" y el contenido de la fila, indicando para cada columna el nombre de la columna y su valor. Después, se deben ejecutar los comandos que se vayan introduciendo por teclado. Si el comando es ".", debe terminar, por supuesto liberando todos los recursos. Si es "k", debe ir a la siguiente fila, indicar el número de la fila y mostrar sus contenidos, como al principio para la primera fila. El comando para ir a la fila anterior será "d". Si se introduce un número, se debe mostrar la fila en la posición indicada por el número. El programa debe mostrar mensajes apropiados en caso de que el comando que se ha introducido no se pueda realizar (por ejemplo, estando en la última fila se pide ir a la siguiente, o se introduce el número de una fila que no existe). La clase `Integer` tiene métodos que permiten determinar si un `String` representa un número entero.

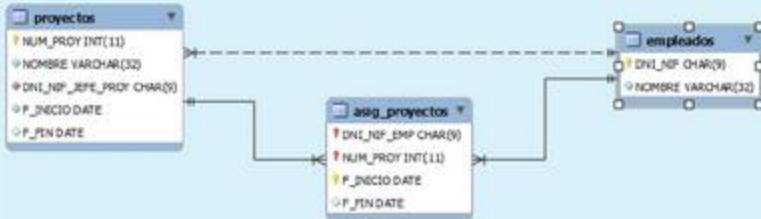
Nota: se puede leer una cadena de caracteres desde el teclado de la siguiente forma: `BufferedReader br=new BufferedReader(new InputStreamReader(System.in)); String comando=br.readLine();`

2. Mejora el programa anterior para que se pueda utilizar con cualquier tabla, cuyo nombre se pedirá al iniciarse el programa. Será necesario obtener información acerca de la tabla mediante el método `getMetaData()` de `ResultSet`.

Para algunos ejercicios que siguen, debe utilizarse un conjunto de tablas para representar los proyectos que desarrolla una empresa, sus empleados y las asignaciones de empleados a proyectos. Las tablas se pueden crear en MySQL con las siguientes sentencias (una posibilidad es ejecutarlas con un programa Java con opción de conexión `allowMultiQueries=true`).

```
CREATE TABLE EMPLEADOS(DNI_NIF CHAR(9) NOT NULL, NOMBRE VARCHAR(32) NOT
NULL, PRIMARY KEY(DNI_NIF));
CREATE TABLE PROYECTOS(NUM_PROY INTEGER AUTO_INCREMENT NOT NULL, NOMBRE
VARCHAR(32) NOT NULL, DNI_NIF_JEFE_PROY CHAR(9) NOT NULL, F_INICIO
DATE NOT NULL, F_FIN DATE, PRIMARY KEY(NUM_PROY), FOREIGN KEY
FK_PROY_JEFE(DNI_NIF_JEFE_PROY) REFERENCES EMPLEADOS(DNI_NIF));
CREATE TABLE ASIG_PROYECTOS(DNI_NIF_EMP CHAR(9), NUM_PROY INTEGER NOT
NULL, F_INICIO DATE NOT NULL, F_FIN DATE, PRIMARY KEY(DNI_NIF_EMP,
NUM_PROY, F_INICIO), FOREIGN KEY F_ASIG_EMP(DNI_NIF_EMP) REFERENCES
EMPLEADOS(DNI_NIF), FOREIGN KEY F_ASIG_PROY(NUM_PROY) REFERENCES
PROYECTOS(NUM_PROY));
```

Para mayor claridad, el siguiente diagrama, obtenido con MySQL Workbench, muestra los esquemas de las tablas, incluyendo campos y claves primarias, y las relaciones entre tablas (claves foráneas).



3. Crea una clase `GestorProyectos` que contenga métodos para almacenar datos de empleados, proyectos y asignaciones de empleados a proyectos. La clase debe tener métodos para:

- Crear un nuevo empleado (`nuevoEmpleado`). Debe devolver true si el empleado se creó correctamente. A este método se le pueden pasar todos los datos del empleado. Un valor null para alguno significa que no se especifica valor. No debe validar los datos que se le pasan. Si el valor indicado para alguno provoca que se lance alguna excepción, este método la propagará. Es decir, su definición debe incluir la opción `throws` con la clase de excepción correspondiente (al menos `SQLException`).

- b) Crear un nuevo proyecto (`nuevoProyecto`). Debe devolver el número de proyecto (`NUM_PROY`). Como el método anterior, tiene parámetros para todos los datos del proyecto (un valor `null` significa que no se especifica valor), no valida los valores proporcionados para ellos, y propaga excepciones. Si se especifica `null` para `F_INICIO`, debe asignarse la fecha actual como fecha de inicio del proyecto, que en MySQL se puede obtener con la función `now()`. Un valor `null` para `F_FIN` significa que no está informada, y debe asignarse un valor `NULL` en la base de datos.
- c) Asignar un empleado a un proyecto (`asignaEmpAProyecto`). Seguir para ello las mismas directrices que para los métodos anteriores, incluyendo las referentes a `F_INICIO` y `F_FIN`.
- Debe probarse esta clase mediante un programa de prueba en el método `main()` que cree varios empleados y proyectos, y realice la asignación de algunos empleados a proyectos. No es necesario realizar ninguna verificación de fechas. Por ejemplo: que la fecha de inicio de una asignación de un empleado a un proyecto no es anterior a la fecha de inicio del proyecto, y otras similares. Pero por supuesto se pueden incluir como mejora. Una posibilidad es realizar estas verificaciones en la clase de Java. Otra es realizarlas mediante restricciones de integridad definidas en la propia base de datos o mediante triggers.
4. Se trata de hacer algo similar a lo hecho en el anterior ejercicio, pero con un planteamiento algo distinto. Hay que crear clases `Empleado`, `Proyecto` y `AsignacionEmplAProyecto`. Cada una de ellas debe tener un constructor sin parámetros y campos correspondientes a los campos de las correspondientes tablas en la base de datos. Deben tener métodos `getXXX()` y `setXXX()` para cada propiedad. Por ejemplo, para "Empleado" serían `String getDNINIF()`, `void setDNINIF(String DNINIF)`, `String getNombre()` y `void setNombre(String nombre)`. Aparte del constructor sin parámetros, deben tener uno con los parámetros correspondientes a los campos de la clave primaria. Por ejemplo: `Empleado(String DNINIF)`, que lanzará una excepción `SQLException` si no existe en la base de datos una fila para los valores de atributos de la clave primaria proporcionados. Deben tener un método `save()` que guarde el objeto en la base de datos, con una sentencia `INSERT`, si no existe (en el caso de clientes, si no existe ninguno con el DNI), o que modifique los datos, con una sentencia `UPDATE`, si existe. Puedes considerar el uso de `INSERT... ON DUPLICATE KEY UPDATE` en MySQL, o sentencias similares en otras bases de datos.
5. Completa la clase `Proyecto` desarrollada en el ejercicio anterior con un método `getListAsigEmpleados()` que devuelva una lista con los empleados asignados actualmente al proyecto. Un empleado está asignado actualmente a un proyecto si existe para él una fila en `ASIG_PROYECTOS` con `F_INICIO` anterior a la fecha actual (que se puede recuperar en MySQL con `now()`) y con `F_FIN` no informada (es decir, con valor `NULL`) o posterior a la fecha actual.
6. Haz un programa para insertar en una tabla de clientes los datos leídos de un fichero en formato CSV, que debes preparar tú mismo. Los datos de cada cliente deben estar en una línea distinta, y como separador de campos puedes utilizar ";" o "|". Debe crearse una clase `LectorDatosClientes`, con un método `insertaDatosClientes(String nombreFichero, String nombreTabla, String separadorCampos)`.

La tabla debe tener la misma estructura que la tabla de ejemplo CLIENTES, y debe estar creada previamente a la ejecución del programa. El parámetro `separadorCampos` indica el separador de campos. Se puede leer el fichero línea a línea utilizando un `BufferedReader`. Se pueden obtener los campos de una línea utilizando la clase `StringTokenizer`, o bien con el método `split()` de `String`. Se debe utilizar una sentencia preparada (`PreparedStatement`) para las operaciones `INSERT`, y deben agruparse todas en una única transacción. Si el carácter separador aparece dos veces seguidas, eso significa que para un campo no se ha especificado un valor, y entonces se le debe asignar el valor `null`. Haz alguna prueba con ficheros cuyos contenidos puedan provocar errores, para asegurarte de que se da el mensaje de error apropiado y no se realiza ningún cambio, al estar todas las operaciones dentro de una transacción. Por ejemplo, especifica un valor nulo para DNI o valores incorrectos para algunos campos.

7. Haz una lista lo más completa posible con condiciones de prueba para el programa anterior. Para cada condición de prueba, indica el resultado esperado. Un ejemplo de condición de prueba sería: "Se especifica valor nulo para tal columna (que no admite valor nulo)", y resultado esperado: "Se muestra un mensaje de error apropiado, se termina la ejecución del programa y no se realiza ningún cambio en la base de datos". Otra sería: "Se especifica un valor alfabético para tal columna (que tiene valores numéricos)", y resultado esperado podría ser el mismo que para la anterior condición de prueba. Verifica cada condición de prueba con un fichero apropiado. No es necesario que el programa proporcione mensajes de error específicos, ni que en las condiciones de prueba se indique un error específico. Normalmente bastará con que el programa gestione las excepciones de tipo `SQLException` y muestre la información que proporciona esta clase de excepciones. Se recomienda redactar las condiciones de prueba sin tener el programa en mente, como lo haría alguien que no sabe cómo está hecho el programa, pero sí qué debe hacer.
8. Si para el ejercicio 6 no has utilizado lotes, modifica el programa para que los utilice. El lote se creará con el método `createBatch()` de `PreparedStatement`. Después se le añadirán todas las sentencias con `addBatch()` y por último se ejecutará el lote con `executeBatch()`.

ACTIVIDADES DE AUTOEVALUACIÓN

1. Un conector es una API que permite a las aplicaciones utilizar bases de datos, pero:
 - a) Solo utilizando el lenguaje Java.
 - b) Solo bases de datos relacionales.
 - c) Solo mediante un driver específico para cada base de datos.
 - d) Ninguna de las respuestas anteriores es correcta.
2. Parte de la problemática del desfase objeto-relacional consiste en que:
 - a) No siempre es fácil obtener los contenidos de una tabla desde un lenguaje orientado a objetos.

- b) No siempre es sencillo almacenar los datos contenidos en objetos complejos en un conjunto de tablas y, a la inversa, recuperar esos datos como objetos.
- c) Cuando se modifican objetos nunca se utilizan transacciones, pero cuando se modifican los contenidos de una base de datos relacional sí.
- d) Todas las respuestas anteriores son correctas.
3. Los *drivers* de JDBC:
- a) Proporcionan acceso a una base de datos relacional particular, pero a cambio de utilizar SQL estándar, es decir, sin ninguna característica propia de la base de datos.
 - b) Solo están disponibles para bases de datos relacionales.
 - c) Proporcionan clases que implementan las interfaces de la especificación JDBC para una base de datos particular.
 - d) Se pueden comunicar directamente con *drivers* de ODBC para complementar sus funcionalidades.
4. La carga de un *driver* JDBC con `Class.forName(...)`:
- a) No es necesaria a partir de Java SE 6.
 - b) Es imprescindible si se quiere que el programa funcione con versiones muy antigüas de Java.
 - c) Requiere que se indique el nombre de la clase que implementa el *driver*.
 - d) Todas las respuestas anteriores son correctas.
5. Para ejecutar una sentencia `INSERT` de SQL, lo más aconsejable es utilizar el método:
- a) `executeQuery()`, porque siempre hay que utilizar este método para cualquier sentencia de SQL que afecte a un conjunto de filas.
 - b) `execute()`, porque una sentencia `INSERT` solo afecta a una fila, y entonces basta con un booleano para saber si la sentencia insertó una fila o no.
 - c) `executeUpdate`, que devuelve el número de filas insertadas.
 - d) `executeQuery`, que devuelve un `ResultSet` con la fila recién insertada.
6. Un `ResultSet`:
- a) Permite obtener datos de la base de datos, pero no modificarlos.
 - b) Permite obtener datos de la base de datos, pero no modificarlos, a menos que sea de tipo *scrollable*, en cuyo caso solo se pueden recorrer secuencialmente empezando por la primera fila y avanzando una a una.
 - c) Permite obtener filas de tablas de la base de datos, modificar y borrar esas filas en la base de datos, y añadir nuevas filas en la base de datos.
 - d) Proporciona una imagen consistente de los contenidos de la base de datos en el momento de realizar la consulta. Esto significa que los cambios realizados por otros procesos en la base de datos nunca se reflejarán en sus contenidos.
7. A los valores para los campos de la fila actual de un `ResultSet`:
- a) Se puede acceder tanto por posición como por nombre.
 - b) Se puede acceder por posición nada más.
 - c) Se puede acceder por nombre nada más.
 - d) Se puede acceder por posición, por nombre o tanto por posición como por nombre, según se especifique en los parámetros del método `getResultSet()`.

8. Si una sentencia de SQL solo se va a ejecutar una vez:
- a) Da igual ejecutarla con un `Statement` o con un `PreparedStatement`, porque solo se ejecutará una vez y el rendimiento es igual.
 - b) Debe ejecutarse con `PreparedStatement` si para construir la sentencia con `Statement` se necesita utilizar alguna variable de programa. Ejecutar con `Statement` sentencias construidas utilizando variables de programa hace al programa vulnerable ante técnicas de inyección de SQL.
 - c) Debe ejecutarse siempre utilizando un `PreparedStatement` para evitar ataques por inyección de SQL.
 - d) Debe ejecutarse siempre utilizando un `PreparedStatement` si se trata de una sentencia que modifique los datos. Las consultas solo leen datos, por lo que no son vulnerables a ataques por inyección de SQL.
9. Un lote creado para un `PreparedStatement`:
- a) Permite ejecutar sentencias de SQL que sería imposible ejecutar con un lote creado para un `Statement`.
 - b) Puede tener sentencias SQL de cualquier tipo, siempre que solo tenga sentencias de un tipo (SELECT, INSERT, UPDATE o DELETE).
 - c) Es para una única sentencia preparada.
 - d) Puede tener sentencias SQL de cualquier tipo, siempre que sean del mismo tipo y tengan el mismo número de marcadores (*placeholders*).
10. Las claves autogeneradas:
- a) Pueden ser de tipo tanto alfabético como numérico, pero para que puedan ser de tipo alfabético hay que utilizar secuencias.
 - b) Solo pueden ser de tipo numérico.
 - c) No existen en Oracle, en su lugar se usan secuencias.
 - d) Son valores que se generan para un campo clave y que se pueden recuperar con un método de `Connection`.

SOLUCIONES:

1. a b c d
 2. a b c d
 3. a b c d
 4. a b c d

5. a b c d
 6. a b c d
 7. a b c d
 8. a b c d

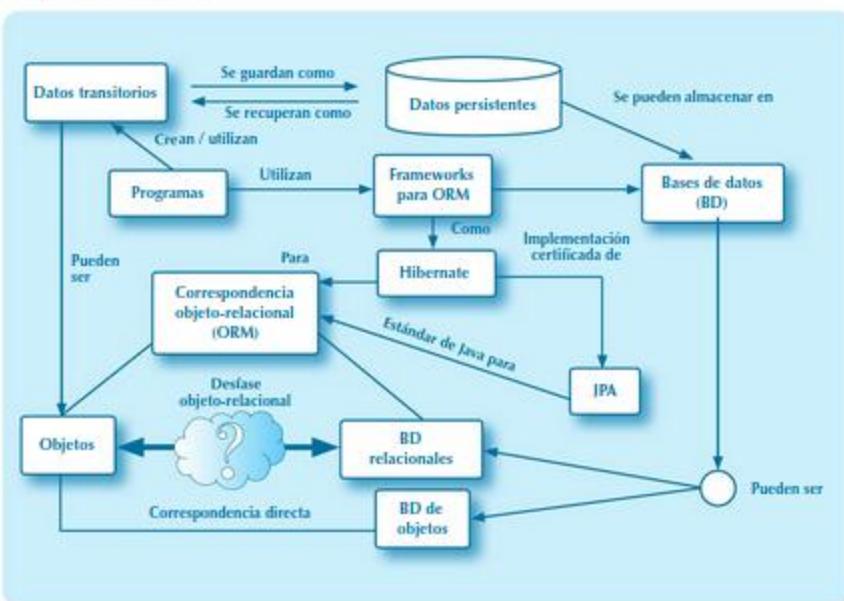
9. a b c d
 10. a b c d

Correspondencia objeto-relacional

Objetivos

- ✓ Comprender el propósito y los fundamentos de la correspondencia objeto-relacional (ORM).
- ✓ Utilizar Hibernate para persistencia de objetos en un esquema relacional mediante ficheros de correspondencia hbm, incluyendo relaciones de uno a uno, de uno a muchos y herencia.
- ✓ Crear POJO (ficheros de clases de Java) apropiados para ORM con Hibernate, crear un esquema relacional para almacenamiento de objetos persistentes, y crear ficheros de correspondencia para establecer la correspondencia entre POJO y tablas del esquema relacional.
- ✓ Analizar el ciclo de vida de un objeto persistente en Hibernate y crear programas en Java que utilicen las principales operaciones disponibles en cada estado.
- ✓ Conocer los fundamentos de los lenguajes HQL y JPQL y crear programas que utilicen HQL.

Mapa conceptual



Glosario

Contexto de persistencia. Lleva el control de los cambios realizados sobre objetos persistentes asociados a él durante una sesión mediante un gestor de entidades, para reflejarlos en la base de datos.

Desfase objeto-relacional. Conjunto de dificultades conceptuales y técnicas para llevar a cabo la persistencia de objetos sobre bases de datos relacionales.

Ficheros de correspondencia o ficheros hbm. Ficheros propios de Hibernate en los que se especifica la correspondencia entre una clase y un esquema relacional (es decir, un conjunto de tablas relacionadas). Es decir, en un fichero de correspondencia se especifica cómo la información contenida en un objeto de una clase determinada se almacena en un esquema relacional.

HQL (Hibernate Query Language), JPQL (Java Persistence Query Language). Lenguajes similares a SQL pero que manejan conjuntos de objetos y sus atributos en lugar de conjuntos de filas y columnas. JPQL es un subconjunto de HQL y es parte de la especificación JPA.

JPA (Java Persistence Arquitectura). API estándar de Java para ORM.

Objeto persistente. Objeto que tiene una representación persistente en la base de datos.

Un objeto persistente está asociado a un contexto de persistencia que detecta los cambios que se realizan sobre él para, en su momento, reflejarlos en la base de datos.

ORM (*Object-Relational Mapping*). Correspondencia objeto-relacional. Técnicas y herramientas que hacen posible la persistencia de objetos en un esquema relacional, basada en una correspondencia entre clases creadas con un lenguaje de programación orientado a objetos y tablas creadas en una base de datos relacional. La traducción correcta de *mapping* en español es *correspondencia*, y no *mapeo*, aunque esta última es, con mucho, la más habitual.

4.1. Correspondencia objeto-relacional

Los lenguajes orientados a objetos se popularizaron enormemente en los años noventa. C++ estaba disponible desde mediados de los ochenta como una extensión orientada a objetos del lenguaje C. Java se popularizó a mediados de los noventa, especialmente para aplicaciones de gestión empresarial con uso intensivo de bases de datos relacionales. Para entonces las bases de datos relacionales ya tenían un dominio absoluto como medio de almacenamiento de datos, que continúa en la actualidad. El lenguaje Java se ha seguido utilizando para este tipo de aplicaciones y ha sido dominante en el campo de las aplicaciones empresariales de gran envergadura, normalmente desplegadas sobre servidores de aplicaciones basados en Java EE (Java Enterprise Edition).

Bien es cierto que utilizar lenguajes orientados a objetos no significa necesariamente utilizar sus características orientadas a objetos. De hecho, gran parte de estas aplicaciones no lo hacen. Es decir, no utilizan objetos para representar los datos persistentes con los que trabajan, o bien no sacan partido de la herencia y otras posibilidades de estos lenguajes.

En cualquier caso, cada vez más se planteó la conveniencia o incluso la necesidad de almacenar objetos en bases de datos relacionales, incluso objetos complejos con referencias a otros objetos, o a colecciones de objetos, y objetos de clases sobre las que se definen otras subclases. La persistencia de objetos en bases de datos relacionales planteaba una serie de problemas, conocidos conjuntamente como *desfase objeto-relacional* (en inglés *object-relational impedance mismatch*) o *desajuste de impedancia objeto-relacional* (véase la figura 1.9).

Para resolver o para evitar estos problemas surgieron varios planteamientos:

1. *Bases de datos de objetos.* Almacenan directamente objetos. Esta parece en principio la solución ideal o la más natural, y esta idea surgió con fuerza en los noventa, durante el *boom* de la programación orientada a objetos. Pero el desarrollo de estas bases de datos planteaba problemas conceptuales y prácticos. Para empezar, la falta de un modelo formal ampliamente aceptado en el que basarse, al contrario que las bases de datos relacionales, basadas en el modelo relacional. También la falta de estándares ampliamente adoptados, como SQL para las bases de datos relacionales, a pesar del trabajo inicial de ODMG (Object Data Management Group). Este grupo se creó en 1991 y se disolvió en 2001. Hoy en día las bases de datos de objetos siguen teniendo un uso muy reducido.
2. *Bases de datos objeto-relacionales.* Son bases de datos relacionales con capacidades para gestionar objetos. En SQL:99 se introdujeron tipos estructurados definidos por el usuario,

entre ellos los tipos objetos (clases). Son una solución de compromiso y resuelven solo parcialmente el problema.

3. *ORM o correspondencia objeto-relacional.* Consiste en el establecimiento de una correspondencia entre clases definidas en un lenguaje de programación orientado a objetos, como Java, y tablas de una base de datos relacional, y en el uso de mecanismos para que las modificaciones sobre los objetos se registren en la base de datos y, a la inversa, para que se pueda recuperar en objetos la información registrada en la base de datos. Esto hace posible la persistencia de objetos en bases de datos puramente relacionales. La primera herramienta ORM importante que surgió fue Hibernate para el lenguaje Java en 2001, como alternativa a la persistencia de objetos proporcionada de Java EE, basada en EJB (Enterprise Java Beans). Después de Hibernate surgieron otras herramientas y frameworks ORM para Java, como Apache Cayenne, Apache OpenJPA, Oracle TopLink, etc. Utilizan en general JDBC para la interacción con la base de datos y funcionan con diferentes bases de datos. Tambien existen soluciones ORM para otros lenguajes orientados a objetos como C++, PHP, Python, etc.

4.2. Hibernate

Hibernate es un *framework* de ORM para Java, distribuido bajo licencia LGPL 2.1 de GNU. Al estar distribuido bajo licencia LGPL, Hibernate se puede utilizar en aplicaciones comerciales sin necesidad de hacer público su código fuente.



WWW

Recurso web

En el sitio web de Hibernate ORM se puede encontrar el software y abundante documentación: <http://hibernate.org/orm/>.

Hibernate se desarrolló en 2001 como una alternativa a la persistencia de objetos proporcionada en Java EE mediante EJB. Java EE es una plataforma muy potente pero compleja y que necesita muchos recursos, pensada para aplicaciones empresariales de envergadura. Como alternativa, se desarrolló Hibernate, un *framework* que proporciona persistencia de objetos basada en ORM, pero sin necesidad de un servidor Java EE. Tiene soporte para transacciones, y utiliza técnicas para mejorar el rendimiento, como por ejemplo *pool* de conexiones, técnicas de *caching* y también *batching* (agrupación en lotes) para las operaciones sobre las bases de datos.

En 2010, Hibernate 3 (versión 3.5.0 y posteriores) se convirtió en implementación certificada de JPA 2.0. JPA es la API estándar de Java para persistencia de objetos, y parte de la especificación Java EE. A partir de entonces, Hibernate siempre ha proporcionado dos API, tanto la propia como la de JPA.

Hibernate tiene su propio lenguaje para manejar objetos persistentes, HQL, inspirado en SQL. Las sentencias de SQL operan sobre filas y columnas de tablas, mientras que las de HQL lo hacen sobre conjuntos de objetos persistentes y sus atributos. El lenguaje estándar de JPA para manejo de objetos persistentes, JPQL, es un subconjunto de HQL.

La siguiente ilustración muestra la arquitectura de Hibernate y sus principales componentes. Como ya se ha comentado, Hibernate proporciona tanto una API propia como una implementación de la API de JPA 2.0. En esta ilustración se muestran las clases y componentes de la API propia de Hibernate, pero son muy similares en concepto a los de la API de JPA.

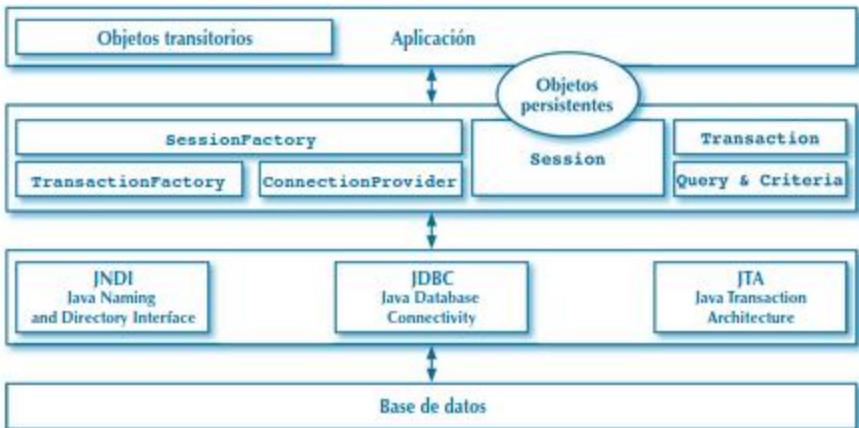


Figura 4.1
Arquitectura de Hibernate

Una aplicación que utiliza Hibernate trabaja con objetos tanto transitorios como persistentes. Los objetos persistentes están siempre asociados a una sesión (`Session`) creada por una `SessionFactory`. Una aplicación puede crear objetos transitorios y después convertirlos en persistentes para que después se almacenen en la base de datos. Puede también recuperar un objeto persistente desde la base de datos y realizar cambios sobre él, para que después se reflejen en la base de datos. Puede agrupar operaciones sobre los objetos persistentes dentro de transacciones (`Transaction`) que están siempre asociadas a una sesión. Puede también utilizar sentencias de HQL mediante la clase `Query`. Hibernate interactúa con la base de datos mediante JDBC. Hibernate puede utilizar internamente API de Java tales como JNDI (Java Naming and Directory Interface) para fuentes de datos (`DataSource`) y JTA (Java Transaction Arquitectura) para transacciones.

4.3. Iniciación a la correspondencia objeto-relacional con Hibernate

La correspondencia entre objetos de Java y datos en un esquema relacional (es decir, un conjunto de tablas) es compleja si se tienen en cuenta todos los posibles casos. Pero para empezar con lo más sencillo, en general se puede decir que para cada clase existe una tabla en el esquema relacional que permite almacenar los objetos de esa clase.

**Figura 4.2**

Correspondencia entre una clase de Java sencilla y una tabla de la base de datos

La correspondencia de otras características de las clases es menos directa. A saber:

- Colecciones de objetos, que representan relaciones de uno a muchos o de muchos a muchos entre objetos.
- Referencias a objetos, que representan relaciones de uno a uno o de muchos a uno entre objetos.
- Herencia.

Hibernate permite establecer una correspondencia objeto-relacional en la que basar la persistencia de objetos sobre bases de datos relacionales no solo cuando se parte de cero, sino también cuando se parte de una base de datos ya existente o cuando se quiere añadir persistencia para clases ya creadas. Esto es muy importante porque no siempre se desarrolla una aplicación desde cero, sino que a menudo debe construirse sobre algo ya existente.

Hibernate impone, eso sí, algunas condiciones tanto a las tablas como a las clases. Si no se crean de cero, sino que vienen dadas, y no las cumplen, puede ser necesario un trabajo previo de adecuación.

- Las tablas deben tener clave primaria. En una base de datos relacional no es obligatorio que todas las tablas la tengan, pero en la práctica casi siempre es así. Cuando no existe, se puede crear una nueva clave primaria autogenerada (se habló de ellas en el capítulo previo dedicado a bases de datos relacionales). Esto es sencillo y normalmente no obliga a hacer ningún cambio en las aplicaciones que utilizan la tabla. En correspondencia, las clases deben tener un atributo para almacenar un identificador único.

RECUERDA

- ✓ Un clave primaria autogenerada para una tabla se define sobre una columna numérica, de manera que el sistema gestor de base de datos asigna un nuevo valor para esa columna para cada nueva fila insertada en la tabla. En MySQL se crean con la opción **AUTO_INCREMENT** en la definición de la columna.

- Los valores de la columna o columnas que forman la clave primaria no pueden cambiar. Esto significa, por ejemplo, que si la clave primaria es el DNI, este no se puede cambiar, ni siquiera para corregir errores o para llenar con ceros por la izquierda, no al menos con una aplicación basada en Hibernate. En la práctica, lo más habitual es tener para

cada tabla una clave primaria autogenerada, lo que evita este tipo de inconvenientes. Si en este caso particular se quisieran evitar duplicidades de DNI, que podrían darse si no es clave primaria, se podría definir un índice único para ese campo, por ejemplo: `CREATE UNIQUE INDEX i_cliente_dni ON cliente(dni)`.

- Las clases deben implementar la interfaz `Serializable`. Para ello basta con incluir `implements Serializable` en la definición de la clase.
- Las clases deben cumplir una serie de convenciones tales como la existencia de método `getX` y `setX` (`getters` y `setters`) para obtener y asignar, respectivamente, el valor de cada atributo `X`.

TOMA NOTA

Hay que tener muy presentes los problemas que pueden surgir debido a la nomenclatura de las tablas y los campos cuando se usan algunos sistemas operativos. En Windows y MacOS pueden surgir problemas si se usan mayúsculas para los nombres de tablas o de campos. El motivo, en última instancia, es que los datos se almacenan en sistemas de ficheros y su nombre se puede convertir a minúsculas. Para evitar problemas, lo mejor es utilizar nombres sin mayúsculas para las tablas y sus campos y, en general, para cualquier objeto de la base de datos.



La correspondencia se puede establecer mediante ficheros de correspondencia de tipo hbm (*Hibernate mapping*). Estos son ficheros de XML en los que se especifica la correspondencia entre, por una parte, clases y sus atributos, y por otra parte, tablas y sus columnas. También, alternativamente, se puede establecer mediante JPA, por medio de anotaciones en las clases. Se puede incluso utilizar un mecanismo para unas clases y otro para otras.

4.4. Correspondencia objeto-relacional a partir de las tablas

En esta sección se explicará cómo con Hibernate se puede establecer la correspondencia para un conjunto de tablas relacionadas. Algunos IDE o entornos de desarrollo incluyen asistentes para establecer esta correspondencia, si bien puede ser necesario o conveniente ajustar posteriormente algunas cosas manualmente. Se explica-

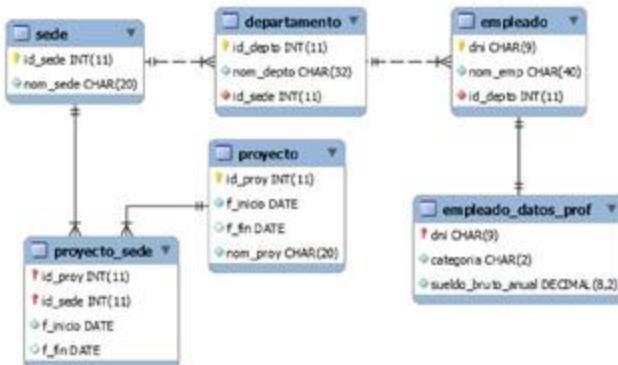


Figura 4.3
Esquema relacional para proyecto de ORM con Hibernate

rá cómo hacerlo en el IDE Netbeans. Se acabará con un pequeño programa de ejemplo que se basa en esta correspondencia para crear varios objetos relacionados y almacenarlos en la base de datos. Los nombres de las tablas se ponen en singular (**departamento** y no **departamentos**). Esto no es estrictamente necesario, pero hace que los nombres generados por Hibernate para la definición de las clases sean claros y descriptivos.

Con las siguientes sentencias de SQL se puede crear, en MySQL, el esquema relacional anterior y un usuario **libro_ad** con los permisos necesarios sobre él. Para ello se puede entrar con el usuario **root**.

```

create database proyecto_orm;
use proyecto_orm;
create table sede(
    id_sede integer auto_increment not null,
    nom_sede char(20) not null,
    primary key(id_sede)
);
create table departamento(
    id_depto integer auto_increment not null,
    nom_depto char(32) not null,
    id_sede integer not null,
    primary key(id_depto),
    foreign key fk_deptosede(id_sede) references sede(id_sede)
);
create table empleado(
    dni char(9) not null,
    nom_emp char(40) not null,
    id_deptosede integer not null,
    primary key(dni),
    foreign key fk_empleado_deptosede(id_deptosede) references departamento(id_deptosede)
);
create table empleado_datos_prof(
    dni char(9) not null,
    categoria char(2) not null,
    sueldo_bruto_anual decimal(8,2),
    primary key(dni),
    foreign key fk_empleado_datosprof_empl(dni) references empleado(dni)
);
create table proyecto (
    id_proy integer auto_increment not null,
    f_inicio date not null,
    f_fin date,
    nom_proy char(20) not null,
    primary key(id_proy)
);
create table proyecto_sede (
    id_proy integer not null,
    id_sede integer not null,
    f_inicio date not null,
    f_fin date,
    primary key(id_proy, id_sede),
    foreign key fk_proyosedeproy (id_proy) references proyecto(id_proy),
    foreign key fk_proyosedesede (id_sede) references sede(id_sede)
);
CREATE USER 'libro_ad'@'localhost' IDENTIFIED BY '(password)';
GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP,EXECUTE
ON proyecto_orm.* TO 'libro_ad'@'localhost';

```

4.4.1. Creación de la conexión con la base de datos

Lo primero es crear en NetBeans una conexión con la base de datos. Habrá que seleccionar uno de los *drivers* disponibles en la pestaña “Services”, o si ninguno de ellos permite la conexión, habrá que instalar antes un *driver* que la permita. En este caso se quiere establecer una conexión con un servidor MySQL 8.0. Se pueden consultar las propiedades de un *driver* pulsando sobre él con el botón derecho y seleccionando la opción “Customize”. En este ejemplo, se puede ver que el nombre del fichero **.jar** para el *driver* con nombre “MySQL (Connector/J driver)” incluye el número de versión 5.1.23.



Figura 4.4
Consulta de los *drivers* instalados y de sus propiedades en NetBeans

Ni este *driver* ni ningún otro disponible permite la conexión con MySQL 8.0. Hay que añadir el *driver* apropiado, que viene empaquetado en un fichero **.jar** disponible en la sección de descargas del sitio web de MySQL. Para ello, se pulsa con el botón derecho sobre “Drivers” y se selecciona la opción “New driver...”. Por último, se selecciona el fichero **.jar**. Hay que pulsar el botón “Find” para localizar la clase que implementa el *driver*. Conviene cambiarle el nombre para diferenciarlo del *driver* para versiones anteriores de MySQL. Se ha elegido “MySQL (Connector/J driver) [8.0]”.

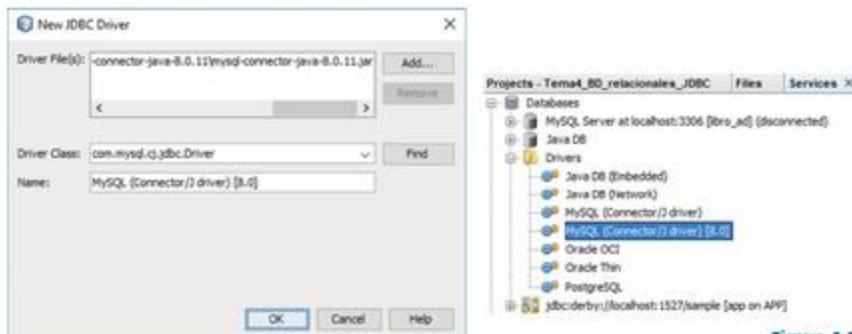


Figura 4.5
Instalación de un nuevo *driver* JDBC en NetBeans

Para crear la conexión a la base de datos, se pulsa con el botón derecho sobre “Databases” y se selecciona la opción “New connection...”. Después se selecciona el *driver* y se especifican las opciones de conexión. Para MySQL 8.0 es necesario añadir algunas opciones en la URL de conexión y esta quedaría así:

```
jdbc:mysql://localhost:3306/proyecto_ORM?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
```

Conviene verificar que la conexión se puede establecer pulsando el botón “Test Connection”.

Por último, se especifica el nombre de la conexión. En este caso se le pone el nombre `proyecto_ORM`.



Figura 4.6

Datos de conexión para nueva conexión

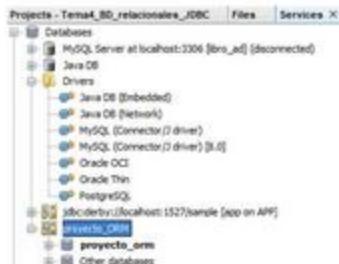


Figura 4.7

Nombre para nueva conexión

4.4.2. Creación del proyecto

Se trata de crear un proyecto en NetBeans como se ha venido haciendo hasta ahora, con nombre `ORM_conexion`.

Una vez creado el proyecto, se pueden utilizar distintos *wizards* o asistentes para automatizar la creación de todo lo necesario para que el proyecto pueda hacer uso de la persistencia de objetos con Hibernate.

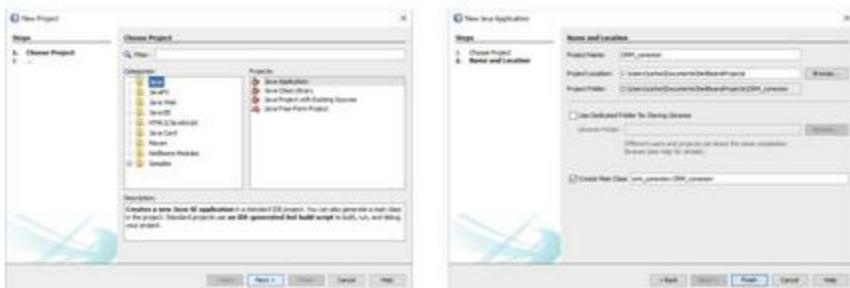


Figura 4.8
Nuevo proyecto en NetBeans

4.4.3. Fichero de configuración de Hibernate: `hibernate.cfg.xml`

Este fichero contiene todo lo necesario para establecer una conexión con la base de datos.

Para crear el fichero de configuración con Netbeans 8.2 se pulsa sobre el proyecto con el botón derecho del ratón y se selecciona la opción “New...”, después “Other...”, y después “Hibernate”. Entonces se muestra una lista con todos los asistentes para Hibernate (figura 4.9).

Se selecciona “Hibernate Configuration Wizard”. En los siguientes diálogos se indica `src` como directorio (`Folder`) para situar el fichero y como conexión la que se acaba de crear en el apartado anterior `proyecto_ORM` (figura 4.10). Eso es todo.

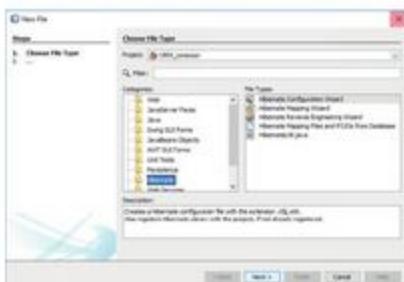


Figura 4.9
Asistentes para Hibernate en NetBeans 8.2

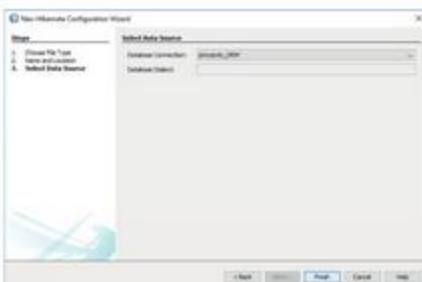


Figura 4.10
Asistente para creación de fichero de configuración `hibernate.cfg.xml`

El contenido del fichero de configuración recién creado es este. Como se puede ver, contiene los datos necesarios para establecer una conexión a la base de datos con JDBC.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
```

```

<session-factory> <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.
        Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/
        proyecto_orm?(... URL de conexión, conteniendo opciones
        adicionales)</property>
    <property name="hibernate.connection.username">libro_ad</property>
    <property name="hibernate.connection.password">(password)</property>
</session-factory>
</hibernate-configuration>

```

4.4.4. Fichero de ingeniería inversa hibernate.reveng.xml

En este fichero de configuración se especifica para qué clases y tablas se va a establecer la correspondencia, y suele tener por nombre `hibernate.reveng.xml` (figura 4.11).

Se puede crear este fichero con otro asistente para Hibernate. Es necesario especificar el fichero de configuración antes creado y las tablas para las que se va a establecer correspondencia. Como directorio para situar el fichero ("Folder") se indica `src`. Hay que elegir las tablas (figura 4.12).

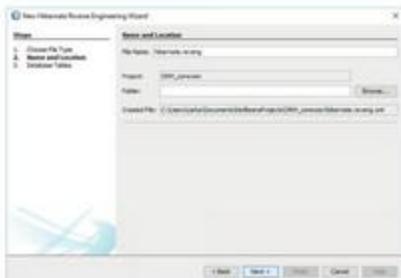


Figura 4.11
Asistente para creación del fichero de
ingeniería inversa `hibernate.reveng.xml`



Figura 4.12
Selección de tablas para el fichero de ingeniería
inversa

Si se muestra un mensaje de error indicando que los *drivers* no se han añadido al proyecto, hay que añadirlos como se explicó en el capítulo anterior. Si no se ven las tablas que cabría esperar ver, o bien si aparecen junto con el texto (*no primary key*), probablemente se está trabajando en Windows y no se ha tenido la precaución de no utilizar mayúsculas para nombres de tablas y atributos.

A continuación, se muestra el contenido del fichero de ingeniería inversa recién creado. En él se indica la base de datos y las tablas que se van a hacer corresponder con objetos.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate
    Reverse Engineering DTD 3.0//EN" "http://hibernate.sourceforge.net/
    hibernate-reverse-engineering-3.0.dtd">
<hibernate-reverse-engineering>
    <schema-selection match-catalog="proyecto_orm"/>
    <table-filter match-name="proyecto"/>

```

```
<table-filter match-name="empleado"/>
<table-filter match-name="empleado_datos_prof"/>
<table-filter match-name="proyecto_sede"/>
<table-filter match-name="departamento"/>
<table-filter match-name="sede"/>
</hibernate-reverse-engineering>
```

4.4.5. POJO (clases) y ficheros de correspondencia

Esta es la parte interesante. Finalmente se generan las clases y los ficheros de correspondencia con el asistente “Hibernate Mapping Files and POJOs from Database”. POJO es el acrónimo de *plain old Java file*, algo así como “un fichero de Java de toda la vida, sin más”. Son clases que incluyen un atributo por cada columna de la tabla y un método *getter* y *setter* para cada uno, es decir, un método para obtener su valor y otro para asignarlo.

Hay que indicar el fichero de configuración antes creado, el de ingeniería inversa, y el nombre del paquete para incluir los POJO y los ficheros de correspondencia. En este ejemplo se opta por crear un nuevo paquete llamado **ORM**. La opción “EJB 3 Annotations” se puede utilizar para generar los ficheros con anotaciones de JPA. No supone ningún problema generarlos con esas anotaciones aunque no se vaya a utilizar JPA para la correspondencia. Como se verá en breve, en el fichero de configuración **hibernate.cfg.xml** se indicarán los ficheros hbm para la correspondencia de cada clase, con lo que se ignorarán estas anotaciones.

Además de crear las clases (POJO) y los ficheros de correspondencia, este asistente añade las siguientes líneas al fichero de configuración **hibernate.cfg.xml**, para establecer la correspondencia de las clases mediante los correspondientes ficheros hbm. Si para alguna clase se quisiera utilizar anotaciones en lugar de ficheros hbm, habría que eliminar del fichero de configuración la linea correspondiente a su fichero de correspondencia. Más adelante se verá en detalle el contenido de estos ficheros de correspondencia.



Figura 4.13
Asistente para la creación de clases (POJO) y ficheros de correspondencia

```
<mapping resource="ORM/Departamento.hbm.xml"/>
<mapping resource="ORM/Empleado.hbm.xml"/>
<mapping resource="ORM/EmpleadoDatosProf.hbm.xml"/>
<mapping resource="ORM/ProyectoSede.hbm.xml"/>
<mapping resource="ORM/Proyecto.hbm.xml"/>
<mapping resource="ORM/Sede.hbm.xml"/>
```

4.4.6. HibernateUtil.java

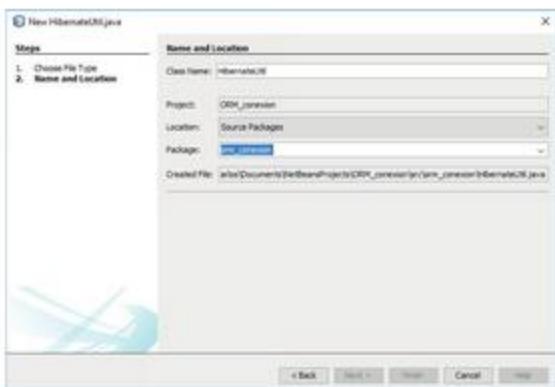


Figura 4.14
Asistente para la creación de HibernateUtil.java

Este es un fichero en el que se crea una clase que facilita enormemente la inicialización de Hibernate. Solo hay que indicar el nombre del fichero y el paquete en que se quiere crear.

Con esto se ha terminado de crear todo lo que necesita un programa que utilice Hibernate para persistencia de objetos. Solo falta escribir este programa en el fichero **ORM_Conexion.java**. Pero antes se explicará cómo descargar y utilizar una versión reciente de Hibernate.

4.5. Descarga y uso de una versión reciente de Hibernate

Se puede comprobar que los asistentes han añadido los ficheros jar necesarios para Hibernate, pero que son de la versión 4.3. Se puede descargar la última versión estable de Hibernate de su página web. Es muy importante verificar sus requisitos, especialmente en lo relativo a la versión de Java. La versión 5.3, que se va a emplear aquí, tiene como prerequisito Java SE 8.

Una vez descargado el fichero zip y descomprimido, aparece un directorio **lib**, y dentro de él los directorios **required**, **optional** y algunos más.



Figura 4.15
Cambios en el proyecto para usar una versión más reciente de Hibernate

Hay que reemplazar los ficheros `.jar` de Hibernate añadidos por el asistente al proyecto por los de la versión más reciente. En el proyecto deben estar al menos los que vienen en el directorio `required`. Aparte de eso, se incluyen todos los equivalentes en la nueva versión a alguno de los que están en el proyecto. También se ha eliminado el `driver` para versiones anteriores de MySQL, `mysql-connector-java-5.1.23-bin.jar`, porque se usa el nuevo para la versión 8.0. En la figura 4.15 se muestra el proyecto antes y después de hacer estos cambios.

Ahora se plantea un pequeño problema, y es que no compila `HibernateUtil.java`. No se admite `AnnotationConfiguration`. Como con cualquier API, cuando se trabaja con Hibernate conviene tener a mano sus Javadoc.

Recurso web



Los Javadoc de Hibernate están disponibles en la descarga y también en Internet en las direcciones que se muestran a continuación:

<http://docs.jboss.org/hibernate/orm/5.3/javadocs/>
<http://docs.jboss.org/hibernate/orm/4.3/javadocs>

En los Javadoc de la versión más reciente de Hibernate no aparece `AnnotationConfiguration`. Pero este código se creó para la versión 4.3 de Hibernate. En los Javadoc de esta versión dice que está obsoleta (*deprecated*), y que su funcionalidad se ha pasado a `Configuration`. El problema se resuelve cambiando `AnnotationConfiguration` por `Configuration`.

Los programas de ejemplo que se hagan a partir de ahora se harán en un proyecto como este, en el fichero `ORM_conexion.java` o en otro en la misma ubicación.

4.6. Programa de ejemplo para persistencia de objetos con Hibernate

Finalmente, un programa de ejemplo que crea una sede, un departamento de esa sede y un empleado de ese departamento. Todas las operaciones se realizan en una sesión y, dentro de ella, en una transacción. La clase `HibernateUtil`, definida en `HibernateUtil.java`, permite abrir una `Session` con suma facilidad. Las clases para objetos persistentes (POJO) están en el paquete con nombre `ORM`, por lo que se prefija con `ORM` su nombre. Para crear objetos se utilizan sus creadores sin parámetros. Para asignar valores a sus atributos se utilizan métodos `setter`, y también para relacionarlos con otros objetos (por ejemplo, para asociar un empleado a su departamento). Para guardar objetos en la base de datos se utiliza el método `save()` de la sesión. Baste esto para una primera aproximación. En las secciones siguientes se explicarán los detalles de la correspondencia objeto-relacional y otros aspectos de Hibernate.

```
// Programa sencillo para ORM con Hibernate
package ormConexion;
import org.hibernate.Session;
import org.hibernate.Transaction;
```

```

public class ORM_conexion {
    public static void main(String[] args) {
        Transaction t = null;
        try (Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();
            ORM.Sede sede = new ORM.Sede();
            sede.setNomSede("MÁLAGA");
            s.save(sede);

            ORM.Departamento depto = new ORM.Departamento();
            depto.setNomDept("INVESTIGACIÓN Y DESARROLLO");
            depto.setSede(sede);
            s.save(depto);

            ORM.Empleado emp = new ORM.Empleado();
            emp.setDni("56789012B");
            emp.setNomEmp("SAMPER");
            emp.setDepartamento(depto);
            s.save(emp);

            t.commit();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}

```

Como resultado de la ejecución del programa, este será el contenido de las tablas. Para todos los valores definidos como `auto_increment` se ha asignado valor 1, al ser la primera fila que se añade a la tabla.

sede		departamento			empleado		
id_sede	nom_emp	id_dept	nom_dept	id_sede	dni	nom_emp	id_dept
1	MÁLAGA	1	I+D	1	56789012B	SAMPER	1

Actividades propuestas



- 4.1. Haz un programa que cree una nueva sede, dos departamentos para esta nueva sede y dos empleados para cada uno de estos departamentos. Verifica que los datos se crean correctamente comprobando el contenido de las tablas.
- 4.2. Ejecuta otra vez el programa de ejemplo, y verifica si se produce alguna excepción. Localiza el tipo de excepción. Cambia el programa para que este tipo de excepción se gestione de manera separada y se proporcione una información más concisa pero suficiente, en lugar de la muy prolífica proporcionada por `printStackTrace()`. No se puede capturar directamente una excepción del tipo `ConstraintViolationException`. Hay que utilizar repetidamente el método

`getCause()` de `Exception` y verificar el tipo de excepción con `instanceof ConstraintViolationException`.

Haz que no puedan existir dos sedes distintas con idéntico nombre, y que no puedan existir dos departamentos distintos con idéntico nombre en una misma sede. La manera más sencilla es con índices únicos (sentencia `CREATE UNIQUE INDEX` de SQL). Verifica tu solución utilizando el programa de ejemplo inicial o pequeñas variaciones de él. Hay que introducir esta restricción en la propia base de datos, y hay que verificar que (en el caso en que se intente crear una nueva sede con el mismo nombre que una ya existente, y en el caso en que se intente crear un nuevo departamento en una sede con el mismo nombre que un departamento ya existente en esa sede) se produce una excepción y el programa la gestiona adecuadamente.

4.7. Ficheros hbm o de correspondencia de Hibernate

En este apartado se verá en detalle el contenido de los ficheros hbm (*Hibernate mapping files* o ficheros de correspondencia de Hibernate), que especifican la correspondencia entre clases de Java y tablas de una base de datos relacional. Se pondrá como ejemplo el fichero de correspondencia `departamento.hbm.xml`, que especifica la correspondencia entre la clase de Java `Departamento` y la tabla `departamento`. Pero la explicación no se limitará a lo que aparece en este fichero, sino que se pondrá todo en un contexto más amplio para explicar otras opciones y características de Hibernate.

La tabla `departamento` se definió de la siguiente manera en MySQL:

```
create table departamento(
    id_depto integer auto_increment not null,
    nom_depto char(32) not null,
    id_sede integer not null,
    primary key(id_depto),
    foreign key fk_depto_sede(id_sede) references sede(id_sede)
);
```

El POJO generado para ella, `Departamento.java`, es el siguiente:

```
/**
 * Departamento generated by hbm2java
 */
@Entity
@Table(name="departamento",catalog="proyecto_orm")
public class Departamento implements java.io.Serializable {

    private Integer idDepto;
    private Sede sede;
    private String nomDepto;
    private Set empleados = new HashSet(0);

    public Departamento() {
    }
```

```

public Departamento(Sede sede, String nomDept) {
    this.sede = sede;
    this.nomDept = nomDept;
}
public Departamento(Sede sede, String nomDept, Set empleados) {
    this.sede = sede;
    this.nomDept = nomDept;
    this.empleados = empleados;
}

@Id @GeneratedValue(strategy=IDENTITY)
@Column(name="id_dept")
public Integer getIdDept() {
    return this.idDept;
}

public void setIdDept(Integer idDept) {
    this.idDept = idDept;
}

@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="id_sede", nullable=false)
@OneToMany(fetch=FetchType.LAZY, mappedBy="departamento")
public Sede getSede() {
    return this.sede;
}

public void setSede(Sede sede) {
    this.sede = sede;
}

@Column(name="nom_dept", nullable=false, length=32)
public String getNomDept() {
    return this.nomDept;
}

public void setNomDept(String nomDept) {
    this.nomDept = nomDept;
}

@OneToMany(fetch=FetchType.LAZY, mappedBy="departamento")
public Set getEmpleados() {
    return this.empleados;
}

public void setEmpleados(Set empleados) {
    this.empleados = empleados;
}
}

```

Hibernate requiere la opción `implements java.io.Serializable`. Los atributos se declaran como privados y tienen métodos *getter* y *setter*.

Los nombres de los atributos de la clase se obtienen a partir de los nombres de los atributos de las tablas siguiendo sencillas reglas, de manera que al atributo `nom_dept` de la tabla `departamento`, por ejemplo, le corresponde el atributo `nomDept` de la clase `Departamento`.

En la clase `Departamento` se reflejan las relaciones de uno a muchos en las que participa la tabla `departamento`, definidas mediante claves foráneas (`FOREIGN KEY` en SQL). La relación

de uno a muchos de un departamento con sus empleados se refleja en `Set empleados`, para los empleados del departamento. El nombre `empleados` es el resultado de añadir `s` al final del nombre de la tabla `empleado` para formar su plural.



PARA SABER MÁS

Reglas de nomenclatura de Hibernate

Hibernate utiliza reglas para la correspondencia entre nombres de tablas y nombres de columnas de tablas, por una parte, y nombres de clases y nombres de atributos de clases, por otra. Esto incluye la formación de los plurales de los nombres de clases para las colecciones, añadiendo `s` o `es`. El motivo para poner los nombres de las tablas en singular, si se tiene esa posibilidad, es que los plurales resultantes sean correctos. Que no lo sean, por supuesto, no impide que las clases funcionen como deben. Las reglas de nomenclatura se pueden cambiar, pero no es sencillo.

La relación de uno a muchos de una sede con sus departamentos se refleja en el atributo `Sede sede` en la clase `Departamento`, para la sede a la que pertenece el departamento. Este es el contenido del fichero de correspondencia `departamento.hbm.xml`.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated by Hibernate Tools 4.3.1 -->
<hibernate-mapping>
  <class name="ORM.Departamento" table="departamento" catalog="proyecto_orm"
    optimistic-lock="version">
    <id name="idDepto" type="java.lang.Integer">
      <column name="id_depto" />
      <generator class="identity" />
    </id>
    <many-to-one name="sede" class="ORM.Sede" fetch="select">
      <column name="id_sede" not-null="true" />
    </many-to-one>
    <property name="nomDept" type="string">
      <column name="nom_dept" length="32" not-null="true" />
    </property>
    <set name="empleados" table="empleado" inverse="true" lazy="true"
      fetch="select">
      <key>
        <column name="id_dept" not-null="true" />
      </key>
      <one-to-many class="ORM.Empleado" />
    </set>
  </class>
</hibernate-mapping>
```

4.7.1. Correspondencia para las clases y atributos de clase

En el elemento <class> se indica la clase (`ORM.Departamento`), la tabla con la que se corresponde (`departamento`) y la base de datos en que está dicha tabla (`proyecto_ORM`).

A continuación, vienen una serie de elementos que establecen la correspondencia de los distintos atributos de la clase. Tienen algunos atributos comunes, tales como:

`name`: el nombre del atributo en la clase.

`type`: el tipo del atributo en la clase de Java. Hibernate hace corresponder a cada posible tipo para un atributo de una clase un tipo estándar de SQL.

El atributo de la tabla de la base de datos se indica con el elemento <column>. Este puede tener algún atributo con información adicional para algunos tipos. Para un `string`, por ejemplo, `length` indica la longitud del campo en la base de datos.

Los tipos de elementos que aparecen son:

- a) <`id`> hace corresponder un atributo de la clase con uno que es clave primaria de la tabla:

<code>Departamento.java</code>	CREATE TABLE departamento
<code>private Integer idDepto;</code>	<code>id_depto integer auto_increment not null,</code> <code>primary key(id_depto)</code>
<code><id name="idDepto" type="java.lang.Integer"></code> <code><column name="id_depto" /></code> <code><generator class="identity" /></code> <code></id></code>	
<code>Departamento.hbm.xml</code>	

<code>Empleado.java</code>	CREATE TABLE empleado
<code>private String dni;</code>	<code>dni char(9) not null,</code> <code>primary key(dni)</code>
<code><id name="dni" type="string"></code> <code><column name="dni" length="9"/></code> <code><generator class="assigned"/></code> <code></id></code>	
<code>Empleado.hbm.xml</code>	

El valor del atributo `generator` hace referencia a una clase de Hibernate utilizada para generar valores para los campos de la clave primaria. Hay unas cuantas disponibles que deberían ser suficientes para cualquier situación. Pero si fuera necesario, se pueden crear nuevas a medida. Algunos de los valores disponibles son:

- `identity`: la columna se define en la base de datos como clave autogenerada, en el caso de MySQL con la opción `auto_increment`. La base de datos genera un valor para esta (`id_depto` de la tabla `departamento`), que se asigna al atributo correspondiente (`idDepto` de la clase `Departamento`).

- assigned:** la aplicación debe proporcionar un valor para el campo. Este es el caso, por ejemplo, para el atributo `DNI` de la clase `Empleado`, que se corresponde con el atributo `DNI` de la tabla `empleado`.
- sequence:** se puede utilizar con Oracle para generar un identificador numérico utilizando una secuencia de la base de datos.
- native:** puede comportarse como `identity`, `sequence` o `hil` (esta última opción no se verá aquí), dependiendo de la base de datos con la que se trabaje.
- foreign:** usa el identificador de otro objeto asociado. Usado para relaciones de uno a uno, que se verán en breve.

b) `<property>` hace corresponder un atributo de la clase con uno de la tabla, sin más:

<pre>Departamento.java private String nomDept;</pre>	<pre>CREATE TABLE departamento nom_dept char(32) not null</pre>
<pre><property name="nomDept" type="string"> <column name="nom_dept" length="32" not-null="true" /> </property></pre>	<pre>Departamento.hbm.xml</pre>
Departamento.hbm.xml	

El atributo `not-null` se utiliza para indicar que no puede tomar valor nulo, lo que se corresponde con la opción `not null` en SQL.

4.7.2. Correspondencia para las relaciones

También debe establecerse una correspondencia para las relaciones. Es decir, también es necesario especificar cómo las relaciones entre distintos objetos se reflejan en la base de datos. Existen relaciones de uno a muchos, de uno a uno y de muchos a muchos. Hibernate permite especificar la correspondencia para todos estos tipos.

Para el ejemplo desarrollado se ha partido de un esquema relacional. Para ilustrar las relaciones se recurre a un diagrama entidad-relación (o diagrama E-R).

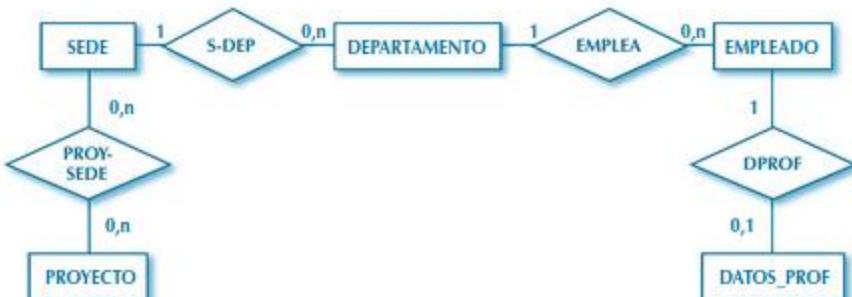


Figura 4.16
Diagrama E-R para el esquema relacional de ejemplo

RECUERDA

- ✓ En el módulo "Bases de Datos" de primer curso se estudian los diagramas E-R, su uso para el modelado conceptual de datos, y la manera de obtener un esquema relacional a partir de un diagrama E-R. En un diagrama E-R se pueden representar relaciones de muchos a muchos. No es así en un esquema relacional, por lo que, para poder representarlas en un esquema relacional, hay que descomponerlas previamente en dos relaciones de uno a muchos.

El diagrama E-R que se muestra en la figura 4.16 tiene relaciones de todos los tipos: de uno a muchos (entre SEDE y DEPARTAMENTO y entre DEPARTAMENTO y EMPLEADO), de uno a uno (entre EMPLEADO y DATOS_PROF) y de muchos a muchos (entre SEDE y PROYECTO).

Las relaciones entre dos clases deben definirse en Hibernate en los ficheros de correspondencia de ambas clases. Por este motivo, si a un objeto de la clase A le pueden corresponder varios de la clase B, la relación entre ambas clases se define de uno a muchos desde la clase A hacia la clase B en el fichero de correspondencia para la clase A, y de muchos a uno desde la clase B hacia la clase A en el fichero de correspondencia para la clase B.

A continuación, se explica la manera en que las relaciones de distintos tipos se han representado en los ficheros de correspondencia generados por Hibernate.

A) Relaciones de uno a muchos

<many-to-one> y <one-to-many> se usan para especificar dos relaciones recíprocas, la segunda de uno a muchos y la primera de muchos a uno. A una sede pueden corresponder varios departamentos (relación uno a muchos o *one-to-many*). Recíprocamente, varios departamentos pueden corresponder a una sede (relación muchos a uno o *many-to-one*). En última instancia, estas relaciones de uno a muchos, sus reciprocas de muchos a uno, sea entre instancias de objetos o entre filas de tablas, son una misma cosa, especificada en la tabla **departamento** con la clave foránea **fk_depto_sede(id_sede)** references **sede(id_sede)**, que establece que a cada fila de la tabla **departamento** le corresponde una en la tabla **sede**, determinada por el valor del atributo **id_sede**. Esto se refleja en la clase **ORM.Departamento** mediante un atributo **sede**, que indica la sede a la que pertenece el departamento, y en la clase **ORM.Sede** mediante una colección **departamentos**, que contiene los departamentos pertenecientes a la sede.

Departamento.java	CREATE TABLE departamento
private Sede sede;	id_sede integer not null, foreign key fk_depto_sede(id_sede) references sede(id_sede)
<many-to-one name="sede" class="ORM.Sede" fetch="select"> <column name="id_sede" not-null="true" /> </many-to-one>	
	Departamento.hbm.xml

Sede.java	CREATE TABLE sede
private Set departamentos = new HashSet(0);	id_sede integer auto_increment not null, primary key(id_sede)
<set name="departamentos" table="departamento" inverse="true" lazy="true">	fetch="select">
<key>	<column name="id_sede" not-null="true" />
</key>	<one-to-many class="ORM.Departamento" />
</set>	
	Sede.hbm.xml

El elemento `<set>` especifica el tipo de colección que se utilizará para almacenar los departamentos de una sede (`HashSet`) y cómo se pueden obtener en la base de datos (consultando la tabla `departamento` por `id_sede`, como se indica en `<key>`). `<one-to-many>` establece el tipo de objetos que se almacenarán en la colección (`ORM.Departamento`).

El atributo `lazy` toma por defecto valor `true`. Eso significa que, cuando se consulta la base de datos para obtener un objeto de tipo `ORM.Departamento`, no se obtienen los empleados del departamento en la colección `empleados`. En lugar de ello, Hibernate proporciona un proxy, que obtendrá los empleados de la base de datos solo cuando se consulte el contenido de la colección `empleados`.

El atributo `fetch` puede tomar los valores `select` y `join`. El valor elegido solo supone una diferencia cuando se consulta la base de datos para obtener varios departamentos. `select` significa que para cada departamento se hará una consulta de SQL para obtener sus empleados. `join` significa que en algunos casos (no siempre, depende del mecanismo utilizado para obtener los datos de los departamentos) se hará una única consulta de SQL con `left outer join`, lo que puede mejorar el rendimiento.

B) Relaciones uno a uno

Se especifica en el elemento `<one-to-one>` en los ficheros de correspondencia `Empleado.hbm.xml` y `EmpleadoDatosProf.hbm.xml`. Es una relación de uno a uno y no de uno a muchos porque la clave foránea se define desde la clave primaria de una tabla (`empleado_datos_prof`). No es completamente simétrica: a un objeto de la clase `EmpleadoDatosProf` le corresponde exactamente uno de la clase `Empleado` (cardinalidad 1), pero a uno de la clase `Empleado` puede no corresponderle ninguno de la clase `EmpleadoDatosProf` (cardinalidad 0,1). Esta asimetría se refleja en los ficheros de correspondencia, cuyos contenidos más relevantes se muestran a continuación:

Empleado.java	CREATE TABLE empleado
private EmpleadoDatosProf empleadoDatosProf;	dni char(9) not null, primary key(dni)
<id name="dni" type="string">	
<column name="dni" length="9" />	
<generator class="assigned" />	
</id>	
<one-to-one name="empleadoDatosProf" class="ORM.EmpleadoDatosProf">	
</one-to-one>	
	Empleado.hbm.xml

<pre>EmpleadoDatosProf.java</pre> <pre>private Empleado empleado;</pre> <pre><id name="dni" type="string"> <column name="dni" length="9" /> <generator class="foreign"> <param name="property">empleado</param> </generator> </id> <one-to-one name="empleado" class="ORM.Empleado" constrained="true"> </one-to-one></pre>	<pre>CREATE TABLE empleado_datos_prof</pre> <pre>dni char(9) not null, primary key(dni), foreign key fk_empleado_datosprof_ empl(dni) references empleado(dni)</pre>
	Empleado.hbm.xml

Las asimetrías se dan en `EmpleadoDatosProf.hbm.xml`, porque es en la tabla `Empleado_datos_prof` donde se define, desde su clave primaria, la clave foránea a la otra tabla `Empleado`. Esto significa que no puede existir un objeto de clase `EmpleadoDatosProf` sin un objeto correspondiente de clase `Empleado`, y se refleja con `constrained=true`. Además, el objeto de clase `EmpleadoDatosProf` debe tener el mismo identificador que el correspondiente de clase `Empleado`, y esto se refleja con `<generator class="foreign">`, lo que significa que el identificador (`dni`) se tomará del objeto de esta otra clase.

C) Relaciones de muchos a muchos

En los ficheros de correspondencia de Hibernate se pueden definir relaciones de este tipo mediante el elemento `<many-to-many>`. No se explica aquí cómo especificar la correspondencia para relaciones de muchos a muchos con Hibernate. Cuando se generan los ficheros de correspondencia a partir de un esquema relacional, no aparecen este tipo de relaciones porque, como se ha comentado, en un esquema relacional no se pueden representar directamente.

Cuando se obtiene un esquema relacional a partir de un diagrama E-R, una relación de muchos a muchos se descompone en dos relaciones de uno a muchos con una nueva entidad. En este caso, la relación de muchos a muchos `PROY-SEDE` entre `SEDE` y `PROYECTO` se descompone en dos relaciones de uno a muchos de `SEDE` y de `PROYECTO` a una nueva entidad, y en el esquema relacional se introducen tablas para `SEDE` (tabla `sede`), para `PROYECTO` (tabla `proyecto`) y para la nueva entidad (tabla `proyecto_sede`), y una restricción de clave foránea (`FOREIGN KEY`) para cada relación de uno a muchos.

Como recapitulación, se incluye un diagrama E-R similar al anterior, solo que con la relación de muchos a muchos entre `SEDE` y `PROYECTO` transformada en dos relaciones de uno a muchos con una nueva entidad (a la que corresponde la clase `ProyectoSede`). Se indican nombres de clases en lugar de entidades. Se indican junto a cada clase los atributos que se emplean para representar las relaciones con otras clases, los que son colecciones con letra en cursiva. Estos atributos son privados y no se puede acceder directamente a ellos, pero existe un método `getter` para recuperar cada uno (`getSede()` para `sede`, `getEmpleados()` para `empleados`, etc), y un método `setter` para modificarlos.

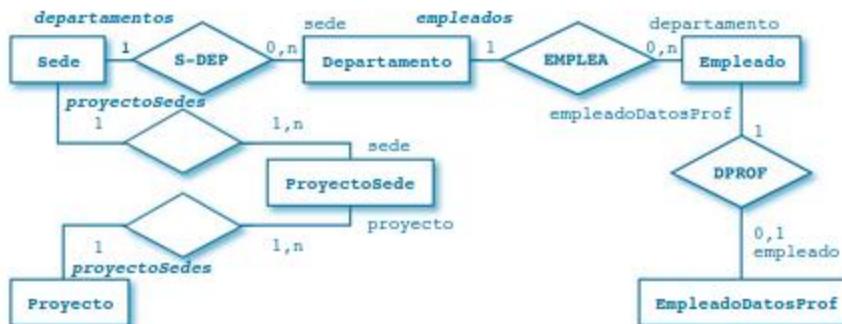


Figura 4.17
Atributos de clase para representación de las relaciones entre clases

Una vez recuperado un objeto persistente, estos atributos permiten acceder fácilmente a otros objetos relacionados, y crear nuevas asociaciones con otros, sin necesidad de operar con la base de datos, solo trabajando con clases de Java. Por ejemplo: a partir del DNI de un empleado se puede obtener su departamento con `getDepartamento()`, y a partir de él todos sus empleados con `getEmpleados()`, y para cada uno de ellos sus datos profesionales con `getEmpleadoDatosProf()`. También se podrían establecer nuevas relaciones entre objetos o modificar las que ya existen utilizando los correspondientes métodos *setter*.

TOMA NOTA

No es recomendable utilizar los métodos *setter* para asignar directamente un valor a las colecciones. Es mejor obtener la colección con el método *getter* y manejarla con los métodos propios de las colecciones tales como `add()` para añadir, etc. En breve se pondrán algunos ejemplos.

Si no se trabaja con Hibernate o una herramienta similar, habría que utilizar consultas SQL para obtener los departamentos de una sede, o los empleados de un departamento, y también para modificar estos datos. Para ello es necesario tener presente la estructura de la base de datos. Esta operativa no se presta a un desarrollo orientado a objetos. Los programas de este tipo tienden a utilizar muchas y a menudo complejas sentencias de SQL. La desventaja que puede tener utilizar Hibernate o frameworks similares es el menor rendimiento de las operaciones de consulta o modificación de datos. Para aplicaciones de envergadura que manejan bases de datos complejas y con grandes volúmenes de datos, es necesario optimizar las operaciones con la base de datos. Pero el margen de maniobra es menor, y para optimizar bien el rendimiento se necesita un conocimiento avanzado del framework. Lo que se gana en claridad y mantenibilidad, y las ventajas de utilizar la programación orientada a objetos, puede ir en detrimento del grado de control sobre la base de datos, de la flexibilidad y de la eficiencia en las operaciones con la base de datos. De todas formas, como se verá más adelante, Hibernate permite también realizar consultas directamente con SQL.

En los siguientes apartados se explica la manera de gestionar las relaciones de uno a muchos y de uno a uno.

4.8. Manejo de relaciones de uno a muchos entre objetos persistentes

Para reflejar este tipo de relaciones entre dos clases se utilizan atributos en ambas clases. En una de ellas el atributo es sencillo y en la otra es una colección.

El siguiente programa crea una nueva sede y tres departamentos de ella, y por último muestra los datos de la sede y de sus departamentos, e itera sobre la colección que en un objeto de tipo `Sede` permite acceder a sus departamentos. El programa no muestra los departamentos creados para la sede si no se utiliza `refresh()`, aunque estos se graban correctamente en la base de datos. La asignación de la sede a los departamentos en objetos de tipo `ORM.Departamento` no se refleja automáticamente en la colección de los departamentos para la sede en el objeto de tipo `ORM.Sede`. Con `s.refresh(sede)` se actualiza `sede` con los contenidos actuales de la base de datos. Es posible que los departamentos no se listeen en el mismo orden en el que se han creado. Con Hibernate se puede establecer un orden para los objetos de una colección, pero una vez establecido, no se pueden recuperar en otro. Para el segundo y tercer departamento, se usan constructores que permiten especificar directamente sus datos, en lugar de usar el constructor sin parámetros y después métodos `setter`, como para el primero. Se ha hecho de las dos formas para ilustrar ambas formas de hacer lo mismo.

```
// Programa que crea una sede y luego varios departamentos asociados
package orm_colecciones_e_iteradores;

import java.util.Iterator;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ORM_colecciones_e_iteradores {

    public static void main(String[] args) {
        Transaction t = null;
        try (Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();
            ORM.Sede sede = new ORM.Sede("TERUEL");
            s.save(sede);

            ORM.Departamento depto = new ORM.Departamento();
            depto.setNomDept("I+D");
            depto.setSede(sede);
            s.save(depto);

            depto = new ORM.Departamento(sede, "MARKETING");
            s.save(depto);

            depto = new ORM.Departamento(sede, "QA");
            s.save(depto);
            s.refresh(sede);

            Iterator itDeptos = sede.getDepartamentos().iterator();
            System.out.println("Nueva sede [" + sede.getIdSede() + "] (" +
                sede.getNomSede() + ")");
            System.out.println("-----");

            while(itDeptos.hasNext()) {
                ORM.Departamento unDept = (ORM.Departamento) itDeptos.
                    next();
            }
        }
    }
}
```

```
        System.out.println("Dept: [" + unDepto.getIdDept() + "] (" +
                           +
                           unDepto.getNomDept() + ")");
    }

    t.commit();
} catch (Exception e) {
    e.printStackTrace(System.err);
    if (t != null) {
        t.rollback();
    }
}
}
```

El siguiente programa hace lo mismo, pero de distinta manera. Primero crea una nueva sede. Después crea uno a uno los nuevos departamentos y los añade a la colección que almacena los departamentos para la nueva sede. Para este programa no hace falta usar `refresh()` para que se muestre correctamente toda la información.

```

// Programa que crea una sede y varios departamentos que añade a su lista

package orm_colecciones_e_iteradores_2;

import java.util.Iterator;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class ORM_colecciones_e_iteradores_2 {

    public static void main(String[] args) {

        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();

            ORM.Sede sede = new ORM.Sede("VALENCIA");
            s.save(sede);

            ORM.Departamento depto1=new ORM.Departamento(sede, "I+D");
            s.save(depto1);
            sede.getDepartamentos().add(depto1);

            ORM.Departamento depto2=new ORM.Departamento(sede, "MARKETING");
            s.save(depto2);
            sede.getDepartamentos().add(depto2);

            ORM.Departamento depto3=new ORM.Departamento(sede, "QA");
            s.save(depto3);
            sede.getDepartamentos().add(depto3);

            Iterator<ORM.Departamento> itDeptos =
                sede.getDepartamentos().iterator();

            System.out.println("Nueva sede [" + sede.getIdSede() + "] (" +
                sede.getNomSede() + ")");
            System.out.println("-----");

            while(itDeptos.hasNext()) {
                ORM.Departamento unDepto = (ORM.Departamento) itDeptos.next();
                System.out.println("Depto: [" + unDepto.getIdDepto() + "] (" +
                    unDepto.getNomDepto() + ")");
            }
        }
    }
}

```

```
        unDept.getNomDept() + "}") +  
    }  
  
    t.commit();  
} catch (Exception e) {  
    if (t != null) {  
        e.printStackTrace(System.err);  
        t.rollback();  
    }  
}  
}
```

4.9. Manejo de relaciones de uno a uno entre objetos persistentes

Para reflejar este tipo de relaciones entre dos clases se utilizan atributos sencillos en ambas clases. Hay una relación de este tipo entre las clases `Empleado` y `EmpleadoDatosProf`.

El siguiente programa de ejemplo crea un nuevo empleado y le asigna sus datos profesionales. El nuevo empleado se asigna a un departamento ya existente con identificador (`id_depart`) 1. Para obtener este departamento, se utiliza el método `get()`. Este método aún no se ha visto, pero su uso aquí es fácil de comprender. Es interesante que, para que se puedan recuperar los datos profesionales a partir del objeto de tipo `Empleado` mediante `getEmpleadosProf()`, debe antes confirmarse la transacción con `commit()`. Esto es porque, al estar definido el generador para el identificador de `Empleado` como `assigned`, se poserga la grabación de los datos hasta entonces, y al coincidir con el identificador de `EmpleadosProf` (generador de tipo `foreign`), solo se puede obtener cuando se haya grabado en la base de datos. Para obtener los datos profesionales a partir del objeto `emp` de tipo `Empleado` ha sido necesario `s.refresh(emp)`.

```
// Pro datos profesionales

package ORM_rel_uno_a_uno;

import java.math.BigDecimal;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ORM_rel_uno_a_uno {

    public static void main(String[] args) {

        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();

            ORM.Departamento depto = s.get(ORM.Departamento.class, 1);

            ORM.Empleado emp = new ORM.Empleado();
            emp.setDni("765432108");
            emp.setDepartamento(depto);
            emp.setNomEmp("SILVA");
            s.save(emp);
        }
    }
}
```

4.10. Sesiones y estados de los objetos persistentes

Ya se ha aprendido cómo se puede establecer la correspondencia objeto-relacional para clases de Java, y también para relaciones entre clases. Se han creado objetos transitorios y se han almacenado en la base de datos como objetos persistentes con Hibernate. Ahora es el momento de explicar en detalle el ciclo de vida de los objetos persistentes, los distintos estados en los que pueden estar, y las operaciones que permiten recuperar objetos persistentes de la base de datos, modificarlos y grabar estas modificaciones de nuevo en la base de datos.

Una clase para la que se establecen correspondencias mediante Hibernate se denomina *clase persistente*, y una instancia de una clase persistente, *objeto persistente*.

La sesión (interfaz `org.hibernate.Session`) es un componente esencial en Hibernate. Una sesión se construye sobre una conexión a la base de datos. Las transacciones creadas por los programas anteriores son transacciones de sesión, y pueden incluir varias transacciones de base de datos. Una sesión puede mejorar el rendimiento de las consultas en objetos persistentes utilizando una caché. La apertura de sesiones es muy rápida utilizando un *pool* de conexiones.

Una sesión, junto con un gestor de entidades asociado, constituye un *contexto de persistencia*. Un gestor de entidades (`javax.persistence.EntityManager`) lleva el control de los cambios que se realizan sobre objetos persistentes. La interfaz `Session` pertenece a la API propia de Hibernate, mientras que la interfaz `EntityManager` pertenece a la de JPA. Se puede obtener un `EntityManager` a partir de una `Session` y viceversa; existe, pues, un puente entre ambas API. Casi todo lo que se puede hacer utilizando una de estas API se puede hacer utilizando la otra, si bien es cierto que la interfaz de Hibernate ofrece más posibilidades, o al menos una manera más directa de hacer muchas cosas. Aquí, por brevedad y simplicidad, se explicará solo la interfaz `Session` de la API de Hibernate.

Los cambios que se realizan sobre objetos persistentes se reflejan, en última instancia, en la base de datos en tanto en cuanto están asociados a un contexto de persistencia.

Los objetos persistentes pueden estar en diferentes estados:

- Transitorio (transient)*. El objeto se acaba de crear y no está asociado con ningún contexto de persistencia. No está grabado en la base de datos. No tiene por qué tener un identificador único asignado, pero podría tenerlo si se ha especificado para su identificador el generador `assigned` (valor “`assigned`” para el atributo `generator` del elemento `id`), y se le ha asignado un valor.
- Gestionado (managed) o persistente (persistent)*. El objeto tiene un identificador asociado y está asociado con un contexto de persistencia, y está grabado en la base de datos. Cualquier cambio que se realice en el objeto se reflejará en la base de datos. No inmediatamente, sino en respuesta a determinadas operaciones sobre la sesión, típicamente cuando se cierra con `close()`.
- Separado (detached)*. El objeto tiene un identificador asociado, pero ya no está asociado con un contexto de persistencia, bien porque el contexto se ha cerrado, bien porque se desvinculó el objeto del contexto.
- Eliminado (removed)*. El objeto tiene un identificador asociado y está asociado con un contexto de persistencia, pero está pendiente de ser borrado de la base de datos.

Un objeto persistente puede cambiar de estado merced a distintas operaciones realizadas sobre él dentro de una sesión. Algunos de estos cambios de estado conllevan operaciones en la base de datos para grabar, modificar o borrar la representación del objeto en ella.

Los objetos en estados transitorio y separado, una vez que ya no están asociados a un contexto de persistencia, pueden ser eliminados por el recolector de basura de Java (*garbage collector*), desde el mismo momento en que no exista ninguna referencia a ellos.

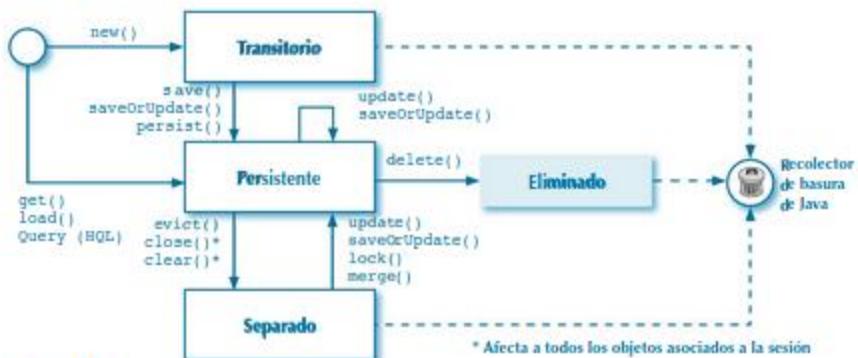


Figura 4.18
Diagrama de estados de un objeto persistente

En el resto de este apartado se ofrece una breve explicación de las operaciones que permiten crear objetos persistentes, cambiar su estado y realizar diversas operaciones con ellos, limitadas, como ya se ha comentado, a la API de Hibernate. Esta es una explicación general para proporcionar una primera introducción a las posibilidades de Hibernate. Por tanto, se omiten muchos detalles, muchas opciones de algunas operaciones, muchos casos particulares, muchas implicaciones que pueden ser relevantes en situaciones concretas, y muchas posibilidades de una herramienta tan potente y compleja como es Hibernate.

4.10.1. De transitorio a persistente con `save()`, `saveOrUpdate()` y `persist()`

Un objeto se puede pasar de estado transitorio a persistente con los métodos `save()`, `saveOrUpdate()` y `persist()` de `Session`. Si el identificador de la clase se genera automáticamente (es el caso con un valor `identity`, `sequence` o `native`, y algunos otros, para el atributo `generator`), no hay que asignar un valor al identificador único para la clase. En ese caso, `save()` y `saveOrUpdate()` devuelven el identificador una vez que se ha grabado el objeto en la base de datos, lo que se hace inmediatamente, mientras que `persist()` no devuelve este identificador, y en algunos casos se puede grabar el objeto posteriormente, lo que contribuye a mejorar el rendimiento. `persist()` solo puede utilizarse dentro de una transacción, al contrario que `save()` y `saveOrUpdate()`, que pueden utilizarse tanto dentro como fuera de una transacción.

```
ORM.Sede sede = new ORM.Sede();
sede.setNomSede("CUENCA");
Integer idSede = (Integer) s.save(sede);
```

En caso contrario, como es el caso con el valor `assigned` para el atributo `generator`, el programa debe asignar un valor al identificador, y la grabación en la base de datos no se realizará inmediatamente.

```
ORM.Empleado emp = new ORM.Empleado();
emp.setDni("78901234X");
emp.setNomEmp("NADALES");
emp.setDepartamento(depto);
s.save(emp);
```

Si se intenta hacer persistente un objeto transitorio creado con `new()` y ya existe un objeto persistente de la misma clase y con idéntico identificador, `save()` generará una excepción, dado que no puede haber dos objetos persistentes con el mismo identificador. De hecho, en alguna de las actividades propuestas anteriormente, se ha pedido verificar esto mismo, y gestionar la excepción que se producía.

El método `saveOrUpdate()` funciona igual que `save()` para un objeto transitorio creado con `new` si no hay ningún objeto persistente con el mismo identificador. Si lo hay, entonces modifica dicho objeto. Dicho de otra forma, `saveOrUpdate()` es a `save()` en Hibernate lo que `INSERT INTO ... ON DUPLICATE KEY UPDATE` es a `INSERT INTO` en SQL.

```
ORM.Empleado emp = new ORM.Empleado();
emp.setDni("78901234X");
emp.setNomEmp("NADALES");
emp.setDepartamento(depto);
s.saveOrUpdate(emp);
```



Actividad propuesta 4.3

Modifica el primer programa de ejemplo para que, si ya existe un empleado con el DNI, le asigne un nombre determinado, y si no existe, cree un empleado con dicho nombre. Utiliza `saveOrUpdate()`.

Ya se ha comentado que `persist()` no devuelve un identificador y que, en el caso de que el identificador de la clase se genere automáticamente, en algunos casos se podría postergar la grabación del objeto en la base de datos. Esto puede mejorar el rendimiento si esta operación se realiza dentro de una transacción con muchas operaciones o que se prolongue mucho en el tiempo. Si no es el caso, no hay ninguna diferencia en la práctica entre utilizar `save()` y `persist()`, aparte de que con la primera se podrá disponer siempre inmediatamente de los identificadores generados automáticamente.

```
ORM.Sede sede = new ORM.Sede();
sede.setNomSede("PALENCIA");
s.persist(sede);
// sede.getIdSede() podría devolver null porque todavía no se haya grabado
// la sede en la base de datos. En cambio, con s.save() la sede se habría
// grabado ya en la base de datos y getIdSede() siempre devolvería un valor.

ORM.Empleado emp = new ORM.Empleado();
emp.setDni("78901234X");
emp.setNomEmp("NADALES");
emp.setDepartamento(depto);
s.persist(emp);
// emp.getDni() devolvería el DNI: "78901234X"
```

4.10.2. Obtención de un objeto persistente con `get()` y `load()`

`get()` y `load()` permiten obtener un objeto persistente e indicar su clase y el valor para su identificador único. Se diferencian en que, si no existe un objeto persistente de la clase indicada y con el identificador indicado, `get()` devuelve un valor `null`, mientras que `load()` lanza una excepción de tipo `ObjectNotFoundException`.

Hasta ahora no se había visto cómo recuperar un objeto persistente grabado en la base de datos. Más adelante se verá cómo obtener objetos persistentes sin necesidad de conocer su identificador único, sino especificando criterios de selección.

Existen versiones de ambos métodos con parámetros adicionales para especificar diversas opciones, tales como tipo de bloqueo sobre el objeto persistente, tiempo de espera máximo antes de desistir de conseguir el bloqueo, y otras más, pero no se explican aquí.

```
ORM.Empleado emp = s.get(ORM.Empleado.class, "78901234X");
if(emp==null) {
    System.out.println("No existe empleado");
}
else {
    System.out.println("El nombre del empleado es "+emp.getNomEmp());
```

Se suele utilizar `load()` cuando se sabe con seguridad que el objeto existe, o solo puede no existir en una situación anómala. El siguiente fragmento de código muestra un escenario de uso típico, en el que no se gestionan excepciones de tipo `ObjectNotFoundException`.

```
ORM.Sede sede = s.load(ORM.Sede.class, 12);
System.out.println("El nombre de la sede es " + sede.getNomSede());
```

4.10.3. De persistente a eliminado con `delete()`

Se puede borrar un objeto persistente con `delete()`.

```
ORM.Sede sede = s.get(ORM.Sede.class, 15);
if(sede != null) {
    s.delete(sede);
}
```

4.10.4. De persistente a separado con `evict()`, `close()` y `clear()`

Estos métodos permiten separar objetos de sesiones, es decir, pasarlo a estado separado o *detached*. Con `evict(Object obj)` se puede separar un objeto. Con `close()` y `clear()` se separan todos los objetos asociados a la sesión. Con `close()` se grabarán los cambios realizados sobre ellos en la base de datos, mientras que con `clear()` no se grabarán. Se pueden seguir consultando y modificando los datos de un objeto en estado separado, pero las modificaciones no se reflejarán en la base de datos, a menos que se vuelva a asociar el objeto a una sesión, que se puede hacer.

```
s.evict(sede); // Separa objeto sede de sesión
```

```
s.close(); // Graba todos los objetos asociados a la sesión y los separa
```

```
s.clear(); // Separa todos los objetos asociados a la sesión, sin grabarlos
```

4.10.5. De separado a persistente con `update()`, `saveOrUpdate()`, `lock()` y `merge()`

Un objeto separado puede volver a hacerse persistente con `lock()`. Al llamar a este método hay que indicar un modo de bloqueo. Hay unos cuantos modos de bloqueo.

Recurso web



Modos de bloqueo con Hibernate:

https://docs.jboss.org/hibernate/stable/orm/userguide/html_single/Hibernate_User_Guide.html#locking-LockMode

Se puede también volver a hacer persistente un objeto separado con `update()` y `saveOrUpdate()`. También con `merge()`, que no hace persistente el objeto que se le pasa, sino que actualiza los contenidos de otro objeto persistente con el mismo identificador en la sesión si existe, o crea uno nuevo si no existe. El resultado, en cualquier caso, es que los datos del objeto que se le pasa se incorporan a un objeto persistente asociado a la sesión, y se devuelve este objeto. El objeto que se le pasa a `merge()` sigue siendo un objeto separado y se puede descartar.

4.11. Lenguajes de consulta HQL y JPQL

HQL son las siglas de Hibernate Query Language. Es un lenguaje inspirado en SQL. Tiene sentencias `SELECT`, `INSERT`, `UPDATE` y `DELETE`, similares a las del lenguaje SQL.

JPQL es un subconjunto de HQL incluido en la especificación de JPA.

www

Recurso web

El lenguaje HQL es similar al lenguaje SQL. Para más información sobre la sintaxis de los lenguajes HQL y JPQL se puede consultar el siguiente enlace, con numerosos ejemplos, y muy asequible con unos conocimientos básicos de SQL (sección de la guía de usuario de Hibernate).

https://docs.jboss.org/hibernate/stable/orm/userguide/html_single/Hibernate_User_Guide.html#hql-statement-types

Los IDE NetBeans y Eclipse tienen intérpretes de HQL que permiten ejecutar directamente sentencias de HQL. En Netbeans, se accede a él pulsando con el botón derecho del ratón sobre el fichero de configuración `hibernate.cfg.xml` y seleccionando la opción “Run HQL Query”. No es muy intuitivo, pero tiene su lógica que el intérprete de HQL se lance desde ahí, ya que en este fichero se define la conexión con la base de datos. En la imagen se puede ver el resultado de una sencilla consulta

Sección	Nombre Departamento	ID Departamento	Empleados
ORM.Sede@65d1621e	INVESTIGACIÓN Y DESARROLLO	11	ORM.Empleado@3817317f, ORM.Empleado@9b61ff9f
ORM.Sede@65d1621e	MARKETING	13	
ORM.Sede@65d1621e	RECURSOS HUMANOS	14	

Figura 4.19

Resultado de una consulta en el intérprete de HQL de Netbeans

Tiene un aspecto muy similar al resultado de una consulta con SQL, pero con algunas particularidades. La columna Sede muestra objetos en lugar de datos de un tipo elemental. Cada objeto viene identificado por el nombre de su clase (`ORM.Sede`) y un identificador único. La columna Empleados muestra listas de empleados, vacías todas excepto para un departamento con dos empleados, que se muestran de la misma manera: con el nombre de su clase (`ORM.Empleado`) y un identificador único.

4.11.1. La interfaz `Query`

La API de Hibernate permite utilizar sentencias de HQL mediante la interfaz `Query`. Esto abre muchas posibilidades. Con lo visto hasta ahora solo se podía acceder a objetos persistentes mediante sus identificadores únicos. Pero hay que tener cuidado con las operaciones de actualización y borrado masivo.

TOMA NOTA



Conviene tener muy en cuenta una advertencia al principio de la sección de la guía de usuario de Hibernate dedicada a HQL y a JPQL, cuyo enlace se ha proporcionado antes (se traduce del inglés).

Hay que tener cuidado cuando se ejecuta una sentencia UPDATE o DELETE.

"Debe actuarse con precaución cuando se ejecutan operaciones masivas de actualización o borrado porque pueden provocar inconsistencias entre la base de datos y las entidades dentro del contexto de persistencia activo. En general, las operaciones masivas de actualización y borrado solo deberían realizarse dentro de una transacción en un nuevo contexto de persistencia, o antes de recuperar o acceder a entidades cuyo estado pueda verse afectado por dichas operaciones".

— Sección 4.10 de la especificación JPA 2.0

La especificación JPA (http://download.oracle.com/otn-pub/jcp/persistence-2.0-fr-eval-oth-JSpec/persistence-2_0-final-spec.pdf) se refiere solo a JPQL. Pero se entiende que este aviso es de aplicación para HQL, dado que JPQL es un subconjunto suyo, y que está incluido en la documentación de Hibernate y en una sección dedicada conjuntamente a HQL y JPQL.

Existe una interfaz `Query` tanto en la API de Hibernate como en la de JPA. En este apartado se hablará solo de la API de Hibernate. Se puede obtener una instancia de `Query` a partir de una instancia de `Session`, mediante el método `createQuery(String sentenciaHQL)`.

RECUERDA

- ✓ `createQuery` devuelve una instancia de la interfaz `org.hibernate.Query`. No hay que confundirla con la interfaz `org.hibernate.Query`, que está obsoleta (`deprecated`) y que está previsto eliminar para la versión 6 de Hibernate. Hay que tener esto en cuenta cuando se consulte la documentación para `Query`, para hacerlo en el lugar correcto.

El funcionamiento de la interfaz `Query` es análogo al de `PreparedStatement` de JDBC. Una `query` puede tener parámetros que se pueden identificar por nombre además de por posición. Por ejemplo: `SELECT nomDept FROM Departamento WHERE idDept=:dep`.

La interfaz `Query`, de la que se ha venido hablando hasta ahora, es, hablando con propiedad, `Query<R>`, es decir, una interfaz genérica con un parámetro de tipo `R`, que es el tipo de los objetos con los que trabaja la `Query`.

En el siguiente cuadro se incluyen algunos de sus principales métodos. Muchos devuelven la propia `Query`, de manera que se pueden encadenar varias llamadas al mismo método, lo que da como resultado un código muy compacto. Por ejemplo:

```
Query q = s.createQuery(
    "FROM Departamento WHERE idSede=:idSede AND nomDept LIKE ':nomDept'"
).
setParameter("idSede", 12).
setParameter("nomDept", "%ADMIN%").
setReadOnly().
setFirstResult(10).setMaxResults(5);
List<ORM.Departamento> listaDep = q.getResultList();
```

CUADRO 4.1

Métodos de `Query <R>`

Método	Funcionalidad
<code>List<R> getResultList()</code>	Devuelve la lista de resultados de una consulta HQL.
<code>R getSingleResult()</code>	Devuelve un único resultado para una consulta. Este método puede lanzar varias excepciones, entre ellas <code>NoResultException</code> cuando la consulta no devuelve ningún resultado y <code>NonUniqueResultException</code> cuando devuelve más de uno.
<code>int executeUpdate()</code>	Ejecuta la sentencia de tipo <code>UPDATE</code> o <code>DELETE</code> y devuelve el número de objetos afectados.
<code>Query<R> setParameter(...)</code>	Asigna un valor a un parámetro de la sentencia. Este método tiene muchas variantes que se pueden consultar en los Javadocs de la API de Hibernate. En general, tienen un parámetro de tipo <code>Object</code> para especificar el valor.
<code><P> Query <R> setParameterList(...)</code>	Este método tiene diversas variantes que permiten asignar a un parámetro una lista de valores. Es útil cuando se especifica en una sentencia de HQL una restricción del tipo <code>valor IN :listaValores</code> . La lista de valores se puede proporcionar en un array o en diversos tipos de colecciones.
<code>Query<R> setReadOnly(boolean readOnly)</code>	Especifica que los resultados recuperados serán solo para lectura, y no para modificación. Conviene usar este método si ese es el caso.

[...]

CUADRO 4.1 (CONT.)

<code>Query<R> setFirstResult(int posInicial)</code>	Especifican el primer resultado que recuperar de entre los obtenidos por la consulta, siendo cero el primero, y el número máximo de resultados que recuperar.
<code>Query<R> setMaxResults(int maxResult)</code>	

En el resto de esta sección se verán varios programas de ejemplo que utilizan distintos tipos de sentencias de HQL.

A) Sentencias SELECT

El siguiente programa muestra los datos de los departamentos cuyo nombre contenga una cadena de caracteres introducida por teclado. La consulta tiene un parámetro `patNombre` al que se le asigna valor con una llamada a `setParameter()` justo antes de lanzar la consulta. Como no se van a modificar los objetos persistentes recuperados, se utiliza `setReadOnly()`. La consulta se realiza con `getResultSet()`, que devuelve la lista de resultados.

```
// Programa que recupera varios objetos persistentes con una consulta en HQL
package orm_query_consulta;

import java.util.List;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import org.hibernate.Session;
import org.hibernate.Query;

public class ORM_query_consulta {

    public static void main(String[] args) {
        try (Session s = HibernateUtil.getSessionFactory().openSession()) {
            System.out.println("Introducir texto para buscar por nombre: ");
            BufferedReader br =
                new BufferedReader(new InputStreamReader(System.in));
            String nomDept = br.readLine();

            Query q = s.createQuery(
                "FROM Departamento WHERE nomDept LIKE :patNombre"
            ).setParameter("patNombre", "%" + nomDept + "%");
            q.setReadOnly(true);

            List<ORM.Departamento> listaDep =
                (List<ORM.Departamento>) q.getResultSet();
            for (ORM.Departamento unDept: listaDep) {
                System.out.println("Dept: [" + unDept.getIdDept() + "] (" +
                    unDept.getNomDept() + ")");
            }
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 4.4



Escribe un programa que utilice el método `setParameterList()` para mostrar los departamentos de varias sedes, y que muestre para cada uno el identificador y el nombre de la sede y del departamento. Es necesario consultar documentación acerca de la interfaz `Query` de la API de Hibernate.

Por supuesto, se podrían modificar los objetos obtenidos mediante una consulta de HQL y grabar los cambios realizados, lo que debería hacerse dentro de una transacción.

El siguiente programa muestra los datos del empleado con mayor sueldo, que recupera mediante una consulta con `getSingleResult()`. Si hubiera más de uno, o si no existieran datos profesionales para ningún empleado, se muestra un mensaje de error específico.

```
// Programa que recupera un único objeto persistente con getSingleResult()
package ORM_Query_consulta_getSingleResult;

import org.hibernate.Session;
import org.hibernate.query.Query;
import javax.persistence.NoResultException;
import javax.persistence.NonUniqueResultException;

public class ORM_Query_consulta_getSingleResult {

    public static void main(String[] args) {
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            Query q = s.createQuery("FROM EmpleadoDatosProf dp WHERE
                dp.sueldoBrutoAnual>=ALL(SELECT sueldoBrutoAnual FROM
                EmpleadoDatosProf)".setReadOnly(true);
            ORM.EmpleadoDatosProf datosProf =
                (ORM.EmpleadoDatosProf) q.getSingleResult();
            System.out.println("Empleado [" + datosProf.getDni() + "] "
                + "(" + datosProf.getEmpleado().getNomEmp() + ")"
                + " de departamento: "
                + datosProf.getEmpleado().getDepartamento().getNomDepto()
                + " de sede: "
                + datosProf.getEmpleado().getDepartamento().getSede().getNomSede()
                + ", con sueldo: " + datosProf.getSueldoBrutoAnual());
        }
        } catch (NoResultException e) {
            System.err.println("ERROR: No hay datos profesionales para
                ningún empleado");
        } catch (NonUniqueResultException e) {
            System.err.println("ERROR: Hay más de un empleado con el salario
                máximo");
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

El siguiente programa de ejemplo obtiene objetos de tipo `Empleado` y los modifica. Si para un `Empleado` no existen datos profesionales, se crea un nuevo objeto de clase `EmpleadoDatosProf` y se asocian mutuamente ambos objetos, porque entre ellos hay una relación de uno a uno. Si existen, hay que modificar el objeto de clase `EmpleadoDatosProf` que ya existe. Como sueldo, se asigna una cantidad aleatoria entre 20.000 € y 80.000 €. El objeto de clase `EmpleadoDatosProf` se graba con `saveOrUpdate()` porque podría existir o no en la base de datos, y el de clase `Empleado`, con `update()`. Siempre que las modificaciones sobre los objetos se puedan realizar con una sentencia `UPDATE` de HQL, es preferible hacerlo así. Si no, habrá que recuperar los objetos con una sentencia `SELECT` de HQL y hacer las modificaciones en un bucle, como se hace en este programa.

```
// Programa que recupera objetos mediante una consulta de HQL y los modifica
package ORM_query_mas_modificacion;

import java.util.List;
import java.util.Random;
import java.math.BigDecimal;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.query.Query;

public class ORM_rel_uno_a_uno {
    public static void main(String[] args) {
        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();
            Query q = s.createQuery("FROM Empleado");
            List<ORM.Empleado> listaEmp = (List<ORM.Empleado>)
                q.getResultList();
            Random rand = new Random();
            for (ORM.Empleado unEmp: listaEmp) {
                ORM.EmpleadoDatosProf datosProf = unEmp.getEmpleadoDatosProf();
                if (datosProf == null) {
                    datosProf = new ORM.EmpleadoDatosProf();
                    datosProf.setEmpleado(unEmp);
                    unEmp.setEmpleadoDatosProf(datosProf);
                }
                datosProf.setCategoria("A");
                datosProf.setSueldoBrutoAnual(
                    BigDecimal.valueOf(2000000 +
                        rand.nextInt(6000000)).movePointLeft(2));
                s.saveOrUpdate(datosProf);
                s.update(unEmp);
            }
            t.commit();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

B) Sentencias UPDATE y DELETE

Se pueden ejecutar sentencias de tipo UPDATE de HQL con `executeUpdate()`. El siguiente programa cambia el nombre del departamento llamado Investigación y Desarrollo por I+D. Después, borra el departamento con nombre Actividades Paranormales. Además, escribe el número de objetos afectados por cada una de estas operaciones. Una característica de HQL es que el operador `LIKE` no distingue entre mayúsculas y minúsculas. Si hubiera algún empleado en uno de los departamentos que se quiere borrar, no se realizaría la operación. Por último, hay que insistir en que, como se indicó al principio, operaciones de este tipo que pueden realizar cambios con muchos objetos podrían dar como resultado inconsistencias si hubiera objetos afectados por ellas en el contexto de persistencia en el que se realizan. Y al igual que cuando se utiliza SQL, hay que extremar la precaución con este tipo de sentencias, y especificar la cláusula `WHERE` de manera apropiada para no modificar o borrar objetos que realmente no se quiere modificar o borrar, porque los resultados podrían ser catastróficos.

```
// Ejecución de sentencias UPDATE y DELETE de HQL con executeQuery()
package orm_executeUpdate;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.query.Query;
public class orm_executeUpdate {

    public static void main(String[] args) {
        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();
            Query q = s.createQuery("UPDATE Departamento SET nomDept='I+D'
                WHERE nomDept='Investigación y Desarrollo'");
            int numObj = q.executeUpdate();
            System.out.println(numObj + " objetos cambiados.");
            q = s.createQuery(
                "DELETE Departamento WHERE nomDept='ACTIVIDADES
                PARANORMALES'");
            numObj = q.executeUpdate();
            System.out.println(numObj + " objetos borrados.");
            t.commit();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

4.12. Correspondencia de la herencia

Una característica fundamental de los lenguajes orientados a objetos es la relación de jerarquía entre clases. Cuando una clase se define como subclase de otra, hereda sus atributos y métodos.

Por esta razón, a este tipo de relación se le denomina también relación de herencia. En Java, una clase se declara como subclase de otra mediante la palabra clave **extends**.

La principal dificultad que plantea la herencia para la persistencia es la vinculación dinámica (*late binding*). Esta consiste en que, si un objeto se define como perteneciente a una clase de la que existen subclases, solo se puede saber si ese objeto pertenece a la propia clase o a una de sus subclases en tiempo de ejecución, cuando se ejecuta el programa, y no antes, en tiempo de compilación. Cuando se quiera hacer persistente ese objeto, podría ser de esa misma clase o de una subclase suya. Por ejemplo, para esta sección se utiliza una superclase **Publicacion** y dos subclases suyas **Libro** y **Revista**. Si se declara un objeto de clase **Publicacion**, en el momento de almacenarlo en base de datos, podría ser un objeto de clase **Libro** o **Revista**, además de **Publicacion**, y en cada caso tendría un conjunto de atributos diferente. Tendrá los de **Publicacion**, pero podría, además, tener los propios de **Libro** o de **Revista**.

A continuación, se muestra la definición de estas clases, que han sido escritas y no generadas automáticamente, y cumplen los requisitos para los POJO que impone Hibernate. Como es práctica habitual, los constructores de las subclases invocan el constructor de su superclase con **super()**.

```
// Fichero Publicacion.java
package ORM;
public class Publicacion implements java.io.Serializable {
    private Integer idPub;
    private String nomPub;
    public Publicacion() {
    }
    public Publicacion(String nomPub) {
        this.nomPub = nomPub;
    }
    public Integer getIdPub() {
        return this.idPub;
    }
    public void setIdPub(Integer idPub) {
        this.idPub = idPub;
    }
    public String getNomPub() {
        return this.nomPub;
    }
    public void setNomPub(String nomPub) {
        this.nomPub = nomPub;
    }
}

// Fichero Libro.java
package ORM;
public class Libro extends Publicacion implements java.io.Serializable {
    private String isbn;
```

```

private String autor;
public Libro() {
}
public Libro(String titulo, String isbn, String autor) {
    super(titulo);
    this.isbn = isbn;
    this.autor = autor;
}
public String getIsbn() {
    return this.isbn;
}
public void setIsbn(String isbn) {
    this.isbn = isbn;
}
public String getAutor() {
    return this.autor;
}
public void setAutor(String autor) {
    this.autor = autor;
}
}

// Fichero Revista.java
package ORM;
public class Revista extends Publicacion implements java.io.Serializable {
    private String issn;
    public Revista () {
    }
    public Revista(String nombre, String issn) {
        super(nombre);
        this.issn = issn;
    }
    public String getIssn() {
        return this.issn;
    }
    public void setIssn(String issn) {
        this.issn = issn;
    }
}
}

```

Al igual que sucede con las relaciones de muchos a muchos, las relaciones jerárquicas:

- No se pueden representar directamente en un esquema relacional.
- Se pueden representar en un diagrama E-R.
- Se pueden hacer corresponder con un esquema relacional con Hibernate.

Por tanto, se ilustrará un ejemplo de correspondencia de relaciones jerárquicas entre clases mediante un diagrama E-R.

En el módulo Bases de Datos de primer curso se estudia la obtención de esquemas relacionales a partir de diagramas E-R. Las relaciones jerárquicas no se pueden representar en esquemas relacionales, pero existe un conjunto de posibles transformaciones que se pueden aplicar a los diagramas E-R para eliminarlas. No siempre se pueden aplicar todas. Cada una tiene ventajas e inconvenientes que hay que sopesar para tomar la decisión más adecuada para cada caso particular. No se entrará en más detalles aquí. Solo se explicarán dos posibles transformaciones y cómo se puede establecer la correspondencia mediante ficheros de correspondencia de Hibernate.



Figura 4.20 Diagrama E-R con herencia



Figura 4.21 Distintas formas de eliminar una relación de herencia en un diagrama E-R

En esencia es lo mismo hablar de entidades relacionadas por una relación jerárquica de generalización/especialización en un diagrama E-R que de una jerarquía de clases. Hibernate permite implementar fácilmente los planteamientos anteriores para la correspondencia entre una jerarquía de clases y un esquema relacional, que se puede obtener directamente del diagrama E-R que resulta de eliminar la jerarquía.

En los subapartados siguientes se mostrará cómo se puede establecer la correspondencia con Hibernate de cada una de las formas anteriores, y se indicará la definición de las tablas y los ficheros de correspondencia hbm.

El programa de ejemplo que se muestra a continuación funcionará sin cambios con cualquiera de los planteamientos anteriores para la correspondencia. Crea un objeto de cada una de las clases `Publicacion`, `Libro` y `Revista` que graba en la base de datos como persistentes. Para cada proyecto hay que indicar el nombre del paquete y de la clase apropiados, de acuerdo al nombre del proyecto.

```
// Programa que crea un objeto persistente de cada clase de una jerarquía
package (...);

import org.hibernate.Session;
import org.hibernate.Transaction;

public class (...) {
    public static void main(String[] args) {
        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();

            ORM.Publicacion pub = new ORM.Publicacion("FOLLETO INDEFINIDO");
            s.save(pub);
            Integer idPub = pub.getIdPub();

            ORM.Libro libro = new ORM.Libro("CORRESPONDENCIA OBJETO-"
                "RELACIONAL", "9789901234567", "LOMAS, ANTONIO");
            s.save(libro);
            Integer idLibro = libro.getIdPub();

            ORM.Revista rev = new ORM.Revista("PERSISTENCIA DE OBJETOS",
                "6789012x05");
            s.save(rev);
            Integer idRev = rev.getIdPub();

            pub = (ORM.Publicacion) s.load(ORM.Publicacion.class, idPub);
            libro = (ORM.Libro) s.load(ORM.Libro.class, idLibro);
            rev = (ORM.Revista) s.load(ORM.Revista.class, idRev);

            System.out.println("Publicación [" + pub.getIdPub() + "] " + pub.
                getNomPub());
            System.out.println("Libro [" + libro.getIdPub() + "] (" + libro.
                getNomPub() + "; " + libro.getIsbn() + ";" + libro.getAutor() +
                ")");
            System.out.println("Revista [" + rev.getIdPub() + "] (" + rev.
                getNomPub() + "; " + rev.getIsbn() + ")");

            t.commit();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

4.12.1. Eliminación de subtipos (una tabla para la jerarquía)

Se utiliza una sola tabla en la que se añaden todos los atributos del tipo y los subtipos y un campo discriminante llamado `tipo`. Todos los atributos de los subtipos deben admitir valores nulos

en la base de datos, pero, dependiendo del valor del discriminante, determinados atributos no deberían tenerlos. Para mayor seguridad, se añade una restricción adicional para controlar que `tipo` tiene un valor válido y que, para cada posible valor, no se admitan valores nulos en los atributos obligatorios para el subtipo correspondiente.

```
create table publicacion(
    id_pub integer auto_increment not null,
    nom_pub varchar(50) not null,
    tipo char(3) not null,
    isbn char(13),
    autor varchar(40) not null,
    issn char(10),
    primary key(id_pub),
    constraint check_subtipos check(tipo='pub' or (tipo='lib' and not
        isnull(isbn) and not isnull(autor)) or (tipo='rev' and not
        isnull(issn)))
);
```

El fichero de correspondencia `Publicacion.hbm.xml` sería el siguiente. Se especifica el campo discriminante con `<discriminator>`, y se especifica su valor para la superclase además de para las subclases. Las subclases se definen con `<subclass>`. Hibernate se encarga de asignar el valor apropiado a este campo, y de interpretarlo correctamente.

```
<?xml version="1.0" encoding="UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ORM.Publicacion" table="publicacion" catalog=" proyecto_orm">
        discriminator-value="pub">
        <id name="idPub" type="java.lang.Integer">
            <column name="id_pub"/>
            <generator class="identity"/>
        </id>
        <discriminator column="tipo" type="string"/>
        <property name="nomPub" type="string">
            <column name="nom_pub" length="50"/>
        </property>
        <subclass name="ORM.Libro" discriminator-value="lib">
            <property name="isbn" type="string">
                <column name="isbn" length="13"/>
            </property>
            <property name="autor" type="string">
                <column name="autor" length="40"/>
            </property>
        </subclass>
        <subclass name="ORM.Revista" discriminator-value="rev">
            <property name="issn" type="string">
                <column name="issn" length="10"/>
            </property>
        </subclass>
    </class>
</hibernate-mapping>
```

Debe añadirse la siguiente línea en el fichero de configuración `hibernate.cfg.xml`:

```
<mapping resource="ORM/Publicacion.hbm.xml"/>
```

El proyecto se puede crear en Netbeans de la siguiente forma:

1. Antes de nada, crear la tabla en la base de datos `proyecto_orm`.
2. Crear un proyecto para una aplicación de Java estándar (Java application).
3. Crear el fichero de configuración de Hibernate, `hibernate.cfg.xml`, utilizando el wizard o asistente, para la conexión `conexion_ORM`.
4. Crear un paquete (*Java package*) con nombre `ORM`, e incluir en él (*New Java Class...*) los POJO `Publicacion.java`, `Libro.java` y `Revista.java`.
5. Crear el fichero de correspondencia `Publicacion.hbm.xml` en el paquete `ORM`, junto a `Publicacion.java`. Validarlo, pulsando con el botón derecho del ratón y seleccionando la opción `Validate XML`.
6. Modificar el fichero de configuración `hibernate.cfg.xml` para añadir una linea `<mapping resource="ORM/Publicacion.hbm.xml"/>`. NetBeans no deja editarlo directamente, pero se puede añadir en la sección “Mappings”, con “Add...”, y especificando el valor `ORM/Publicacion.hbm.xml` para `resource`.
7. Crear un fichero `HibernateUtil.java` en el paquete que contiene el fichero creado por NetBeans para la clase principal (la que contiene el método `main()`). Este fichero debe ser igual que los que se han usado, con el mismo nombre, para todos los programas hasta ahora. Se puede copiar de un proyecto anterior.
8. Modificar el fichero para la clase principal, creado automáticamente por NetBeans, para incluir los contenidos del programa de ejemplo anterior.
9. Por último, eliminar del proyecto los ficheros JAR incluidos automáticamente para una versión antigua de Hibernate e incluir los de la nueva versión de Hibernate, y que se han venido incluyendo hasta ahora en todos los proyectos.

Una vez ejecutado el programa de ejemplo, asumiendo que la tabla `publicacion` estaba vacía inicialmente, sus contenidos serán:

<code>id_pub</code>	<code>nom_pub</code>	<code>tipo</code>	<code>isbn</code>	<code>autor</code>	<code>issn</code>
1	FOLLETO INDEFINIDO	pub	NULL	NULL	NULL
2	CORRESPONDENCIA OBJETO-RELACIONAL	lib	9789901234567	LOMAS, ANTONIO	NULL
3	PERSESTENCIA DE OBJETOS	rev	NULL	NULL	6789012x05

Debería, además, garantizarse la unicidad del ISBN para los libros y del ISSN para las revistas. Esto se podría hacer creando índices únicos para la tabla por estos campos:

```
CREATE UNIQUE INDEX i_publicaciones_isbn ON publicaciones(isbn);
CREATE UNIQUE INDEX i_publicaciones_issn ON publicaciones(issn);
```

4.12.2. Una tabla por subclase (eliminación de la jerarquía)

Se utiliza una tabla por clase, y las tablas para cada subclase tienen la misma clave primaria que la de la superclase, e incluyen una clave foránea hacia la tabla para la superclase. Es una solución muy flexible y elegante. Su único inconveniente, si acaso, es que se necesita una clave foránea en cada subclase, y para obtener los datos de un objeto de una subclase se necesita hacer una consulta en SQL relacionando las tablas para la subclase y la superclase.

```

create table publicacion(
    id_pub integer auto_increment not null,
    nom_pub varchar(50) not null,
    primary key(id_pub)
);

create table libro(
    id_pub integer not null,
    isbn char(13) not null,
    autor varchar(40),
    primary key(id_pub),
    constraint fk_libro_publicacion foreign key(id_pub) references
        publicacion(id_pub)
);

create table revista(
    id_pub integer not null,
    issn char(10) not null,
    primary key(id_pub),
    constraint fk_revista_publicacion foreign key(id_pub) references
        publicacion(id_pub)
);

```

En cuanto al fichero de correspondencia `Publicacion.hbm.xml`, las subclases se definen con `<joined-subclass>` en lugar de con `<subclass>`, y no hay campo discriminante. Por lo demás es prácticamente igual.

```

<?xml version="1.0" encoding="UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ORM.Publicacion" table="publicacion" catalog="proyecto_orm">
        <id name="idPub" type="java.lang.Integer">
            <column name="id_pub"/>
            <generator class="identity"/>
        </id>
        <property name="nomPub" type="string">
            <column name="nom_pub" length="50"/>
        </property>
        <joined-subclass name="ORM.Libro">
            <key column="id_pub"/>
            <property name="isbn" type="string">
                <column name="isbn" length="13"/>
            </property>
            <property name="autor" type="string">
                <column name="autor" length="40"/>
            </property>
        </joined-subclass>
    </class>
</hibernate-mapping>

```

```

</joined-subclass>
<joined-subclass name="ORM.Revista">
<key column="id_pub"/>
<property name="isbn" type="string">
  <column name="isbn" length="10"/>
</property>
</joined-subclass>
</class>
</hibernate-mapping>

```

Una vez ejecutado el programa de ejemplo, asumiendo que las tablas estaban vacías inicialmente, sus contenidos serán:

publicacion	
id_pub	nom_pub
1	FOLLETO INDEFINIDO
2	CORRESPONDENCIA OBJETO-RELACIONAL
3	PERSISTENCIA DE OBJETOS

libro			revista	
id_pub	isbn	autor	id_pub	isbn
2	9789901234567	LOMAS, ANTONIO	3	6789012X05

De nuevo, debería garantizarse la unicidad del ISBN para los libros y del ISSN para las revistas. Esto se podría hacer creando índices únicos:

```

CREATE UNIQUE INDEX i_libros_isbn ON libros(isbn);
CREATE UNIQUE INDEX i_revistas_issn ON revistas (isbn);

```

4.13. Consultas con SQL

Con Hibernate también se pueden realizar consultas directamente en SQL. Conviene, de todas formas, no abusar de esta posibilidad y recurrir a ella solo en casos muy justificados. Por ejemplo, cuando supone un aumento del rendimiento muy importante, cuando se requiere una ordenación particular de los resultados, o para consultas muy complejas que no es posible realizar en HQL, o si ya se dispone de la consulta en SQL y utilizarla ahorra tiempo.

El siguiente programa realiza una consulta en SQL y obtiene los resultados como una lista de *arrays* de objetos, que acto seguido muestra en pantalla. Hibernate ofrece muchas más posibilidades, algunas muy interesantes, pero no vale la pena entretenérse aquí en ellas.

```

// Programa que realiza una consulta en SQL y muestra los resultados
package ORM_Query_SQL;

import java.util.List;
import org.hibernate.Session;

```

```

public class ORM_Query_SQL {
    public static void main(String[] args) {
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            List<Object[]> empleados = s.createNativeQuery("SELECT e.dni, e.id_
                depto, e.nom_emp, dp.categoría FROM empleado e LEFT OUTER JOIN
                empleado_datos_prof dp ON dp.dni=e.dni").list();
            for (Object[] objetos: empleados) {
                System.out.println("Empleado [dni:" + (String) objetos[0]
                    + ", id_depto: " + (Integer) objetos[1]
                    + ", nom_emp: " + (String) objetos[2]
                    + ", categoría: " + (String) objetos[3]
                    + "]");
            }
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}

```

Hibernate realiza la consulta de SQL mediante JDBC y consulta los metadatos del `ResultSet` obtenido para averiguar el tipo de cada columna. Para evitar que tenga que hacer esto, o simplemente para explicitar los nombres y los tipos de cada una de las columnas que se espera obtener, lo que redunda en un código de programa más legible y mantenible, se puede utilizar `addScalar()`. Con ello, la consulta se haría de esta forma:

```

import org.hibernate.type.IntegerType;
import org.hibernate.type.StringType;
(...)
List<Object[]> empleados = s.createNativeQuery("SELECT e.dni, e.id_depto,
    e.nom_emp, dp.categoría FROM empleado e LEFT OUTER JOIN empleado_datos_
    prof dp ON dp.dni=e.dni")
    .addScalar("dni", StringType.INSTANCE)
    .addScalar("id_depto", IntegerType.INSTANCE)
    .addScalar("nom_emp", StringType.INSTANCE)
    .addScalar("categoría", StringType.INSTANCE).list()

```

Resumen

- La persistencia de objetos en bases de datos relacionales plantea un conjunto de problemas que se conocen en conjunto como “desfasaje objeto-relacional”.
- ORM (*object-relational mapping*), o “correspondencia objeto-relacional”, consiste en el establecimiento de una correspondencia entre clases definidas en un lenguaje de programación orientado a objetos y tablas de una base de datos relacional, y en el uso de mecanismos para que las modificaciones sobre los objetos se registren en la base de datos y, a la inversa, para que se pueda recuperar en objetos la información registrada en la base de datos. La correspondencia objeto-relacional hace posible la persistencia de objetos en bases de datos puramente relacionales.

- Hibernate es un *framework* para ORM en lenguaje Java distribuido bajo licencia LGPL de GNU. Fue la primera herramienta para correspondencia objeto-relacional que logró amplia aceptación y, hoy en día, sigue siendo la más importante. Existe un *porting* de Hibernate para .NET, llamado NHibernate.
- Además de tener su propia API nativa, Hibernate es una implementación certificada de JPA, el estándar de Java para ORM, que es parte de la especificación Java EE.
- Hibernate permite establecer la correspondencia no solo para objetos individuales, sino también para las relaciones entre objetos, ya sean de uno a muchos, de uno a uno o de muchos a muchos. Las relaciones se reflejan en los objetos mediante atributos sencillos y mediante colecciones.
- Hibernate proporciona varios mecanismos alternativos para establecer la ORM para un conjunto de tablas relacionadas por la relación de herencia.
- Con Hibernate, la correspondencia se puede establecer mediante ficheros hbm (*Hibernate mapping files* o "ficheros de correspondencia de Hibernate") o mediante anotaciones de JPA.
- Una clase para la que se ha definido la correspondencia objeto-relacional es una clase persistente.
- El concepto de sesión es fundamental en Hibernate. Los objetos persistentes lo son en tanto en cuanto están asociados a una sesión. Hibernate sigue la pista de los cambios realizados sobre objetos persistentes asociados a una sesión mediante un gestor de entidades asociado a la sesión y, llegado el momento, se encarga de que esos cambios se reflejen en la base de datos, de manera que se mantenga siempre la consistencia entre el contenido de los objetos persistentes en memoria y en la base de datos. Una sesión y un gestor de entidades asociados constituyen un contexto de persistencia.
- Hibernate tiene su propio lenguaje para manejar objetos persistentes, HQL, similar a SQL, con la diferencia de que maneja objetos y sus propiedades en lugar de filas y columnas de tablas de bases de datos relacionales. JPQL es un subconjunto de HQL y parte del estándar JPA.



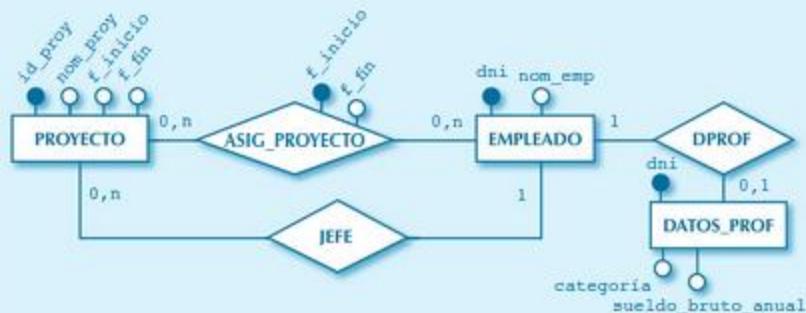
Ejercicios propuestos

Para la realización de estos ejercicios puede ser necesario consultar la documentación de la API de Java SE 8 (<https://docs.oracle.com/javase/8/docs/api/>) o de la de Hibernate (<http://docs.jboss.org/hibernate/orm/5.3/javadocs>).

Para algunos de los ejercicios debe utilizarse un esquema relacional para representar los proyectos que desarrolla una empresa, sus empleados y las asignaciones de empleados a proyectos. Se da el diagrama entidad-relación para describir las entidades y las relaciones entre ellas, y las sentencias SQL necesarias para crear las tablas en MySQL. No hay que crear todas las tablas y clases desde el principio, sino a medida que vayan siendo necesarias para realizar las actividades. De esa manera, se podrá desarrollar la correspondencia objeto-relacional paso a paso, y centrar la atención en cada momento en cada aspecto que desarrollear.

Para desarrollar los componentes de la aplicación hay que utilizar los asistentes siempre que sea posible, y revisar los ficheros generados, y hacer cambios manualmente si es preciso.

Se puede utilizar otra base de datos relacional distinta de MySQL, preferiblemente Oracle.



Precisiones adicionales:

- La fecha de inicio es obligatoria para proyectos y para asignaciones de empleados a proyectos. Si la fecha de fin es nula, significa que el proyecto no ha concluido. Si no es nula y está en el pasado, significa que el proyecto ha concluido.
- El campo dni puede representar también un NIE, es decir, un número de identificación para extranjeros. Por brevedad se le ha llamado sencillamente dni.

```

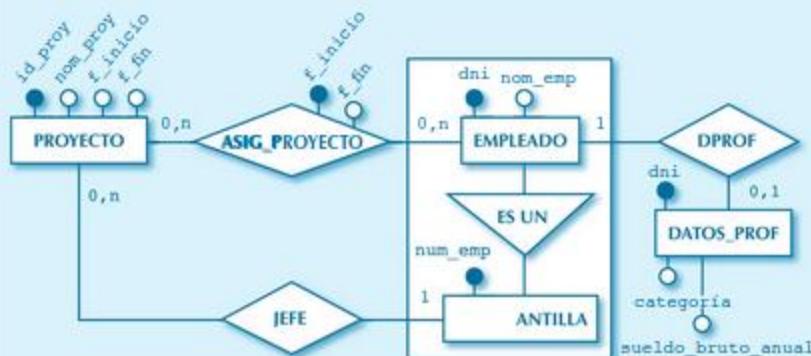
create table empleado(
    dni char(9) not null,
    nom_emp varchar(32) not null,
    primary key(dni)
);
create table proyecto(
    id_proy integer auto_increment not null,
    nom_proy varchar(32) not null,
    f_inicio date not null,
    f_fin date,
    dni_jefe_proy char(9) not null,
    primary key(id_proy),
    foreign key fk_proy_jefe(dni_jefe_proy) references
        empleado(dni)
);
create table asig_proyecto(
    dni_emp char(9),
    id_proy integer not null,
    f_inicio date not null,
    f_fin date,
    primary key(dni_emp, id_proy, f_inicio),
    foreign key fk_asig_proy_dni(dni_emp) references
        empleado(dni),
    foreign key fk_asig_proy_id_proy(id_proy) references
        proyecto(id_proy)
);
  
```

```

        foreign key f_asig_emp(dni_emp) references empleado(dni),
        foreign key f_asig_proy(id_proy) references proyecto(id_proy)
    );
create table datos_prof(
    dni char(9) not null,
    categoria char(2) not null,
    sueldo_bruto_anual decimal(8,2),
    primary key(dni),
    foreign key fk_datosprof_empl(dni) references empleado(dni)
);

```

1. Crea una base de datos en un servidor de bases de datos para las siguientes actividades, llamada `orm_gestion_proyectos`. Crea un usuario sobre la misma base de datos, al que hay que otorgarle los permisos apropiados, como se ha hecho en ejemplos anteriores. Crea una conexión a la base de datos desde el IDE, y verifica que funciona correctamente.
2. Crea las tablas `proyecto` y `empleado`. Crea los POJO y los ficheros de correspondencia utilizando los asistentes que proporciona el IDE. Crea un programa de prueba que cree varios empleados y varios proyectos, y que asigne a cada proyecto su jefe de proyecto. Verifica que no se puede crear un proyecto si no está informado su jefe de proyecto. Verifica que no se puede crear un proyecto si no está informada su fecha de inicio.
3. Crea todo lo necesario para la correspondencia de la relación de muchos a muchos entre proyectos y empleados (incluyendo –y empezando por– la tabla `asig_proyecto`). Crea un programa de prueba que cree varios proyectos y varios empleados y asigne al menos a un empleado a varios proyectos y al menos a un proyecto varios empleados. Verifica que no se puede crear una asignación de un empleado a un proyecto si no se asigna una fecha de inicio.
4. Crea la tabla `datos_prof` y todo lo necesario para la correspondencia de uno a uno entre empleados y datos profesionales. Crea un programa de prueba que genere un nuevo empleado y le asigne sus datos profesionales, y a un empleado ya existente, identificado por su DNI.
5. Crea un programa que, para varias asignaciones cualesquiera de empleados a proyectos, les asigne como fecha de fin una fecha en el pasado (por ejemplo, el día anterior a la fecha actual).
6. Crea un programa que, utilizando una sentencia de HQL, muestre los detalles de todas las asignaciones de empleados a proyectos.
7. Lo mismo que en la actividad anterior, pero solo para asignaciones vigentes. Es decir, no se deben mostrar asignaciones cuya fecha de inicio esté en el futuro ni asignaciones cuya fecha de fin esté informada y esté en el pasado.
8. Considera la distinción de empleados en plantilla frente a empleados no en plantilla, que se contratan para proyectos determinados. Solo los empleados en plantilla pueden ser jefes de proyecto. El siguiente esquema E-R describe este cambio.



Se creará una nueva tabla `emp_plantilla` para los empleados en plantilla, incluyendo un campo `dni` y con una clave foránea por este campo hacia `empleado`. A los empleados en plantilla se les asigna un número de empleado que sirve como identificación para ellos, pero no será una tabla autogenerada. La relación entre empleados y empleados en plantilla es una relación jerárquica, que se implementa en Java con una nueva clase `EmpPlantilla` subclase de la actual `Empleado`. Se pide hacer una copia de la base de datos y una copia del proyecto para ORM desarrollado hasta ahora, que funcionará con la nueva base de datos, y desarrollar la actividad a partir de esta copia. Se pide hacer los cambios necesarios en la estructura de la base de datos, hacer las conversiones de datos necesarias, hacer los cambios necesarios en el proyecto, tanto en las clases como en los ficheros de correspondencia, y crear un programa de prueba en el método `domain()` de una nueva clase.

El primer paso será hacer los cambios de estructura y la conversión de datos en la nueva base de datos. Todos los empleados existentes se considerarán empleados en plantilla. Una vez creada la nueva tabla `emp_plantilla`, se tiene que insertar en ella una fila por empleado, con una sentencia `INSERT INTO emp_plantilla(...) SELECT ... FROM empleado`. La clave foránea de `proyecto` a `empleado` debe cambiarse a `emp_plantilla`. Para ello debe borrarse la antigua clave foránea y crear la nueva. Eso es todo en lo que respecta a la conversión de datos. Después habrá que crear la nueva clase `EmpPlantilla` como subordinada de `Empleado`, y la nueva correspondencia para la clase `Empleado` en conjunto con `EmpleadoPlantilla`.

El programa de prueba debe: crear nuevos empleados en plantilla y no en plantilla, crear proyectos nuevos y asignarles como jefe de proyecto empleados en plantilla tanto existentes como empleados creados en el mismo programa, y por último asignar empleados a los nuevos proyectos creados, tanto de plantilla como no de plantilla, y tanto existentes como creados en el mismo programa.

ACTIVIDADES DE AUTOEVALUACIÓN

1. El desfase objeto-relacional es:

- a) El conjunto de dificultades conceptuales y técnicas que plantean las bases de datos de objetos.
- b) La sobrecarga del sistema que supone la persistencia de objetos en bases de datos relacionales, que se evita con la persistencia en bases de datos de objetos.
- c) El conjunto de dificultades que plantea la persistencia de objetos en bases de datos relacionales.
- d) El conjunto de correspondencias que es necesario establecer entre clases y tablas de un esquema relacional para hacer posible la persistencia de objetos en bases de datos relacionales.

2. ORM es:

- a) El conjunto de extensiones que se pueden añadir a una base de datos relacional para convertirla en una base de datos objeto-relacional, en las que es posible, con ciertas limitaciones, la persistencia de objetos.
- b) Un conjunto de extensiones que permiten implementar una base de datos de objetos sobre una base de datos relacional.
- c) Un estándar para persistencia de objetos en bases de datos relacionales.
- d) La correspondencia objeto-relacional, que hace posible la persistencia de objetos en bases de datos puramente relacionales.

3. Hibernate es:

- a) Una implementación de JPA (Java Persistence Architecture).
- b) Un framework para ORM para el lenguaje Java.
- c) Una base de datos objeto-relacional.
- d) Nada de lo anterior.

4. Las tablas utilizadas para ORM con Hibernate:

- a) No es necesario que tengan clave primaria.
- b) Deben tener una clave primaria simple, es decir, consistente en una única columna.
- c) Deben tener una clave en la que solo intervengan valores numéricos y auto-generados.
- d) Deben tener clave primaria.

5. El fichero de configuración de Hibernate (`hibernate.cfg.xml`):

- a) Permite establecer la correspondencia objeto-relacional para un conjunto de clases, independientemente de que para otras clases se pueda establecer mediante ficheros hbm.
- b) Guarda todos los datos necesarios para establecer una conexión con el sistema gestor de bases de datos relacional en el que se van a almacenar los objetos persistentes.
- c) Normalmente hace referencia a un fichero de ingeniería inversa, cuyo nombre suele ser `hibernate.reveng.xml`.
- d) Debe almacenarse en la base de datos.

6. Cuando se hace `s.save(obj)`, siendo obj un objeto persistente:
- a) Hibernate asigna siempre un identificador único automáticamente al objeto.
 - b) Hibernate asigna un identificador único al objeto solo en el caso en que se haya especificado para el atributo `generator` el valor `assigned`.
 - c) Hibernate asigna un identificador único al objeto solo si se ha especificado para el atributo `generator` el valor `identity`.
 - d) Hibernate asigna un identificador único al objeto si se ha especificado para el atributo `generator` el valor `identity`.
7. Las relaciones de uno a muchos entre dos clases:
- a) Se representan mediante colecciones en las dos clases.
 - b) Se representan mediante una colección de tipo `HashSet` en una de las clases.
 - c) Se representan mediante un atributo sencillo en una clase y mediante una colección en la otra clase.
 - d) No se pueden representar directamente en Hibernate.
8. JPA es:
- a) Un estándar para el que Hibernate incluye una implementación.
 - b) Un estándar de ORM para Java.
 - c) Un estándar que incluye anotaciones en clases de Java para ORM y también el lenguaje JPQL.
 - d) Todas las opciones anteriores son correctas.
9. Un objeto de una clase persistente puede no tener un identificador único asignado:
- a) Si está en estado transitorio (*transient*).
 - b) Si está en estado gestionado (*managed*).
 - c) Justo después de pasar a estado separado (*detached*).
 - d) Justo después de pasar a estado eliminado (*removed*).
10. Para establecer la correspondencia objeto-relacional para varias clases entre las cuales hay una relación de herencia:
- a) Es necesario introducir un campo adicional que funcione como discriminante.
 - b) Es necesario crear una tabla en la base de datos para todas y cada una de las clases relacionadas mediante la relación de herencia.
 - c) Si no se crean tablas para las subclases, es necesario crear uno o varios índices únicos en la tabla para la superclase, para evitar duplicidades.
 - d) Ninguna de las respuestas anteriores es válida.

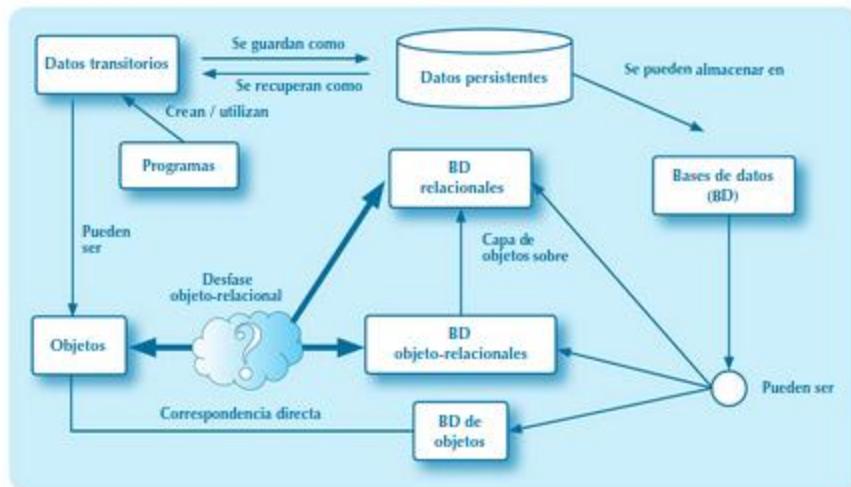
SOLUCIONES:1. a b c d2. a b c d3. a b c d4. a b c d5. a b c d6. a b c d7. a b c d8. a b c d9. a b c d10. a b c d

Bases de datos de objetos y objeto-relacionales

Objetivos

- ✓ Comprender el propósito y los fundamentos de las BDO.
- ✓ Conocer el estándar ODMG 3.0 para persistencia transparente de objetos desde lenguajes orientados a objetos.
- ✓ Analizar los lenguajes ODL para definición de objetos y OQL para consulta de objetos.
- ✓ Estudiar la BDO Matisse y utilizar su ODL para crear esquemas de objetos.
- ✓ Utilizar el Java *binding* de Matisse para crear, consultar, modificar y borrar objetos persistentes.
- ✓ Entender el propósito y los fundamentos de las BDOR.
- ✓ Asimilar las características objeto-relacionales de la base de datos Oracle.
- ✓ Crear esquemas objeto-relacionales y guardar datos en ellos utilizando el SQL de Oracle.

Mapa conceptual



Glosario

Base de datos de objetos (BDO). Base de datos que puede almacenar directamente objetos complejos.

Base de datos objeto-relacional (BDOR). Base de datos que, aun basándose en el modelo relacional para el almacenamiento de datos, permite trabajar también con objetos.

Language binding. Conjunto de mecanismos implementados en un lenguaje orientado a objetos específico para conseguir persistencia transparente de objetos.

ODL (Object Definition Language). Lenguaje para definición de esquemas de objetos en BDO, parte del estándar ODMG.

ODMG (Object Data Management Group). Grupo formado en 1991 para desarrollar estándares para persistencia de objetos en BDO. También se utiliza para hacer referencia al estándar desarrollado por dicho grupo, cuya última versión es ODMG 3.0.

OQL (Object Query language). Lenguaje para consulta de objetos, parte del estándar ODMG.

Persistencia transparente. Posibilidad de trabajar de la misma forma con objetos persistentes y transitorios, de manera que los cambios realizados en estos últimos durante una transacción se reflejen automáticamente en la base de datos cuando se confirme la transacción.

SQL:99. También llamado SQL3. Revisión del estándar SQL que introduce tipos estructurados definidos por el usuario, entre ellos tipos de objetos.

5.1. Bases de datos de objetos y objeto-relacionales, y correspondencia objeto-relacional

Con el auge de la programación orientada a objetos desde finales de los años ochenta se pusieron de manifiesto las dificultades, tanto conceptuales como técnicas, que planteaba el almacenamiento de objetos complejos en bases de datos relacionales. Para referirse a ellas en conjunto se acuñó el término *desfase objeto-relacional* (en inglés, *object-relational impedance mismatch*) o *desajuste de impedancia objeto-relacional*.

Como solución natural se plantearon bases de datos que permitieran almacenar directamente objetos, a las que se llamó BDO, o en inglés ODB (*object databases*), o también bases de datos orientadas a objetos (BDOO), o en inglés OODB (*object-oriented databases*).

Se intentó repetir la receta del éxito de las bases de datos relacionales unos años antes: el estar basadas en un modelo formal y el temprano desarrollo de estándares, principalmente el lenguaje SQL. Así pues, los primeros esfuerzos se desarrollaron en estos dos ámbitos. En el primero, destaca el manifiesto de Atkinson y otros (1989), con una propuesta para los requisitos que debería cumplir una BDO. Para desarrollar estándares sobre los que basar las BDO, se creó en 1991 el grupo ODMG (Object Database Management Group, en un principio). Este grupo fue creado por R. Cattell (de Sun) y representantes de varios fabricantes de BDO, justo cuando Sun acababa de desarrollar el lenguaje Java. En 1998, ODMG cambió el significado de sus siglas a Object Data Management Group, para reflejar que su objetivo era la persistencia de objetos en general, no necesariamente en BDO.

A pesar del entusiasmo inicial, los resultados a medio y largo plazo no fueron los esperados. No llegó a desarrollarse ningún modelo formal ampliamente aceptado. ODMG desarrolló estándares como ODL para definición de datos, OQL para consulta de datos y, en lugar de un OML o lenguaje específico de manipulación de datos, *language bindings* o vinculaciones con lenguajes orientados a objetos de propósito general ya existentes. La última versión del estándar es ODMG 3.0, publicada en 2000. ODMG se disolvió en 2001, y las compañías que lo integraban centraron sus esfuerzos en JDO (Java data objects). En 2004 se cedieron los derechos para revisar la especificación ODMG 3.0 a OMG (Object Management Group).

Por otra parte, en SQL:99 se incluyeron tipos estructurados definidos por el usuario, entre ellos tipos de objetos, que se han implementado en algunas bases de datos relacionales como PostgreSQL y Oracle, que se pueden considerar por ello BDOR.

Las BDO siguen teniendo un uso muy limitado a fecha de hoy. Existen unas cuantas BDO de software libre y privativas, entre ellas Matisse, Objectivity/DB y db4o. Otros planteamientos alternativos para persistencia de objetos han tenido más éxito. Oracle es una BDOR dominante en el segmento de las aplicaciones empresariales, que implementa los objetos en una capa de software sobre una base de datos relacional. La correspondencia objeto-relacional (ORM) ha tenido mucho éxito con productos como Hibernate (2001), y el estándar JPA para ORM es parte integrante de Java EE desde Java EE 6 (2009).

5.2. Características de las bases de datos de objetos

Una BDO es una base de datos que puede trabajar directamente con objetos. El manifiesto de Atkinson y otros (1989) propone una lista de características obligatorias que debería cumplir cualquier SGBDO (sistema gestor de BDO), o en inglés ODBMS (*object database management system*). Es muy amplio, pero contiene las ideas fundamentales sobre las que se basan las BDO. En este resumen se omite, por conocido, lo que es similar en las bases de datos relacionales, y conceptos no estrictamente de bases de datos, sino en general de programación orientada a objetos.

1. *Objetos complejos.* Debe ser posible almacenar objetos complejos. Un objeto complejo contiene un conjunto de atributos, cada uno con un nombre y un valor, como una fila de una base de datos relacional. Pero en un objeto complejo los atributos pueden ser no solo de tipos elementales, como números y cadenas de caracteres, sino también:
 - a) Referencias a otros objetos.
 - b) Colecciones desordenadas (conjuntos) y colecciones ordenadas (listas).
2. *Identidad de objetos.* Los objetos tienen una existencia independiente de su valor. Cada objeto tiene un identificador único. Se distingue entre igualdad (dos objetos tienen el mismo valor, o mejor dicho, los mismos valores para sus atributos) e identidad (dos objetos son el mismo objeto, es decir, tienen el mismo identificador único). Identidad implica igualdad, pero no a la inversa.
3. *Tipos o clases.* Un tipo comprende un conjunto de objetos con características comunes, a saber, un conjunto de atributos y un conjunto de operaciones. Una clase es similar, pero es algo más que un conjunto de objetos. Una clase es a su vez un objeto sobre el que se pueden realizar algunas operaciones. Una clase podría mantener su extensión (*extent*), es decir, el conjunto de todos los objetos de la clase, y proporcionar mecanismos para acceder a ellos y realizar operaciones sobre ellos. En adelante se hablará solo de clases, pero todo lo que se diga de ellas se puede hacer extensivo a los tipos.
4. *Extensibilidad.* El sistema de bases de datos tendrá un conjunto de tipos y clases predefinidos. Debe poderse definir nuevos tipos y clases a partir de otros ya existentes, y estos deben poderse utilizar para cualquier cosa para la que se puedan usar los predefinidos.
5. *Complejidad computacional.* Debe poderse realizar cualquier posible cálculo utilizando el DML (lenguaje de manipulación de datos) de la base de datos. Esta es una diferencia fundamental con las bases de datos relacionales. El sublenguaje DML de SQL no es computacionalmente completo. Para realizar muchos o la mayoría de los cálculos con los datos hay que obtener los datos en el contexto de un lenguaje de programación de propósito general. Este requisito se puede satisfacer mediante *bindings* o vinculaciones con lenguajes orientados a objetos de propósito general ya existentes, como por ejemplo Java, en lugar de con un DML separado.
6. *Métodos de consulta sencillos.* Idealmente de alto nivel, eficientes e independientes de la aplicación, es decir, utilizables con cualquier posible base de datos y lenguaje de programación.



Recurso digital

En el anexo web 5.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás el manifiesto de Atkinson, Bancilhon, DeWitt, Dittrich, Maier y Zdonik (1989).

5.3. El estándar ODMG

El estándar ODMG es un estándar para persistencia de objetos en bases de datos, cuya última versión es ODMG 3.0, del año 2000. Los principales componentes de ODMG 3.0 son:

- Modelo de objetos. Basado en el modelo de objetos de OMG.

- ODL o lenguaje de definición de objetos.
- OQL o lenguaje de consulta de objetos. Inspirado en SQL-92.
- *Language bindings* o vinculaciones con los lenguajes C++, Smalltalk y Java.

A diferencia de SQL para bases de datos relacionales, que incluye un DML independiente del lenguaje de programación desde el que se usa, ODMG 3.0 no incluye un DML, sino *language bindings* o vinculaciones con lenguajes de propósito general existentes, a saber: C++, Smalltalk y Java. Esta planteamiento se conoce como *persistencia transparente*. Desde el lenguaje de programación de propósito general se crean, modifican y borran los objetos persistentes de la misma forma que los no persistentes, y se utilizan los mecanismos del propio lenguaje. Es en el momento de confirmar los cambios realizados dentro de una transacción cuando se graban en la base de datos los cambios realizados sobre objetos persistentes. Un *language binding* también proporciona mecanismos para realizar consultas de OQL y obtener los resultados como objetos persistentes.

Recurso digital



En el anexo web 5.2, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás más información acerca de ODMG y del soporte del estándar ODMG 3.0 por parte de diversas BDO (2001).

5.4. ODL

ODL es un lenguaje formal para especificación de objetos. Un SGBDO almacena objetos de tipos definidos en un esquema mediante ODL. Se pueden establecer las analogías con el modelo relacional que se recogen en el cuadro 5.1.

CUADRO 5.1
Analogías entre el modelo ODMG y modelo relacional

Modelo de objetos de ODMG 3.0	Modelo relacional
ODL: lenguaje de definición de objetos	DDL: lenguaje de definición de datos, subconjunto de SQL
Esquema de objetos	Esquema relacional
Clase	Tabla
Instancia de clase	Fila de tabla

5.4.1. Modelo de objetos de ODL

ODL se basa sobre el modelo de objetos de OMG, cuyas principales características son:

1. El estado de un objeto está definido por los valores de sus propiedades. El valor de las propiedades, y por lo tanto el estado de un objeto, puede cambiar a lo largo del tiempo.

2. Las propiedades pueden ser atributos del objeto o relaciones con otros objetos. Las relaciones pueden ser de uno a uno, de uno a muchos o de muchos a muchos.
3. Un objeto es una instancia de una clase y tiene un OID (*object identifier*), que es un identificador único a nivel del sistema, y que no cambia a lo largo del tiempo. Se distingue entre identidad, determinada por el OID (dos objetos son idénticos si y solo si tienen el mismo OID), e igualdad (dos objetos son iguales si y solo si tienen el mismo estado, es decir, los mismos valores para sus atributos).
4. La conducta (*behavior*) de un objeto se define por el conjunto de operaciones que se pueden realizar sobre él. Cada una puede tener una lista de parámetros de entrada y de salida, cada uno de un tipo, y puede devolver un resultado de un tipo determinado.
5. Un tipo (que puede ser una clase o una interfaz) tiene una especificación externa y puede tener una o más implementaciones. La especificación incluye las características visibles externamente del tipo, a saber:
 - a) Operaciones: que se pueden invocar en las instancias del tipo.
 - b) Propiedades: o variables de estado, cuyo valor se puede consultar y cambiar.
 - c) Excepciones: que pueden lanzar sus operaciones.

5.4.2. Clases e interfaces

Un tipo puede ser:

- a) Una interfaz (*interface*). Una interfaz se define por su conducta abstracta, es decir, por sus operaciones.
- b) Una clase (*class*). Una clase se define por su conducta abstracta y por su estado abstracto, es decir, por sus atributos.

La distinción entre interfaces y clases, y su relación con los objetos, es similar a la que hay en Java.

- No se pueden crear instancias de una interfaz, pero sí de una clase. Un objeto es una instancia de una clase.
- Clases e interfaces pueden heredar de interfaces (relaciones de tipo “es un/una”).
- Las clases pueden extender otras clases (relaciones de tipo “extiende”). Una clase que extiende a otra, es decir, una subclase, hereda las operaciones y propiedades de la otra, es decir, de la superclase.

Para una clase se puede definir una extensión. Se define con la palabra clave **extent** y se le asigna un nombre. Una extensión proporciona acceso a todas las instancias de una clase. A una extensión (**extent**) se le

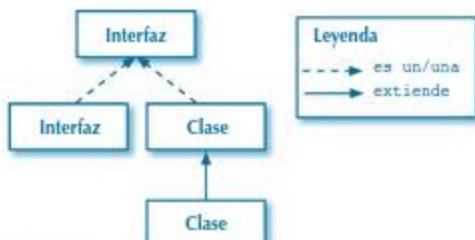


Figura 5.1

Relaciones “es un/una” y “extiende” entre interfaces y clases

puede asociar una clave (**key**), formada por uno o varios atributos. No puede haber dos objetos distintos dentro de la extensión con idéntico valor para la clave. Esta es una restricción de integridad que debe mantener el SGBDO.

5.4.3. Relaciones

Las relaciones son propiedades de las clases. Una relación entre dos clases consiste en la asociación de cada instancia de una clase con una o más instancias de la otra. ODMG 3.0 solo contempla relaciones binarias, es decir, relaciones entre dos clases.

- Pueden ser de uno a uno (1-1), uno a muchos (1-N) y muchos a muchos (N-M).
- Se definen en ambos extremos, es decir, en cada una de las dos clases, como un *traversal path*. La relación entre **Departamento** y **Empleado**, por ejemplo, es de uno a muchos, y está definida por dos *traversal paths* recíprocos el uno del otro: un departamento *emplea_a* uno o varios empleados, y un empleado *trabaja_en* un departamento.



Figura 5.2
Relaciones y *traversal paths* en ODL

- Según la cardinalidad máxima de la relación en un *traversal path*, este se puede implementar mediante un único objeto (cardinalidad máxima uno) o mediante una colección de objetos (cardinalidad máxima muchos). En este último caso, la colección puede ser ordenada (**list**) o desordenada (**set**).
- Las relaciones se gestionan mediante operaciones públicas de las clases involucradas. Si en un *traversal path* la cardinalidad máxima es 1, habrá operaciones para especificar el objeto relacionado, y en caso de que la mínima sea 0, para borrarlo. Si la cardinalidad máxima es mayor que 1, existirán operaciones para añadir y borrar objetos en el conjunto de objetos relacionados. En el ejemplo anterior, existirán operaciones para asignar a un empleado su departamento, así como para añadir un empleado a un departamento.

En ODMG 3.0 se usa una notación propia para diagramas de objetos que representan clases y relaciones entre ellas. Como ejemplo se incluye un diagrama E-R, que incluye los proyectos y los empleados de una empresa, y las relaciones entre ellos, y el correspondiente diagrama de objetos. La traducción de uno a otro es directa.

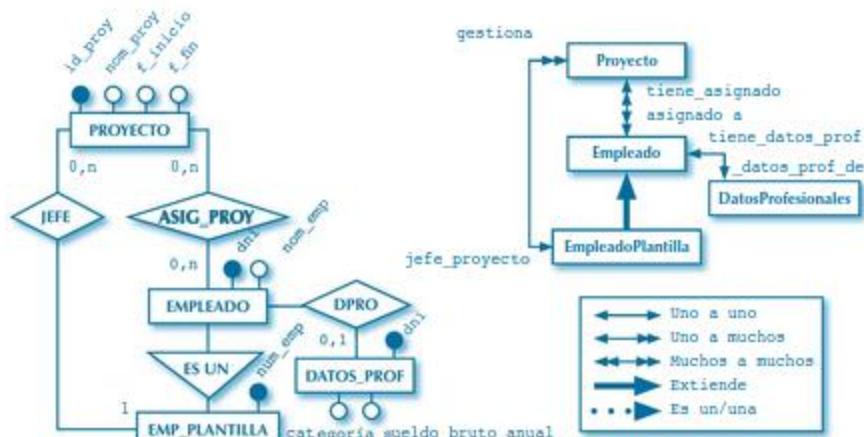
**Figura 5.3**

Diagrama E-R y de objetos correspondiente para proyectos y empleados de una empresa

Actividad propuesta 5.1



Crea un diagrama E-R, y el correspondiente diagrama de objetos, para representar las siguientes relaciones entre entidades. Una publicación puede ser un libro o una revista. En cualquier caso, tiene un nombre de publicación [nom_pub], que en el caso del libro representa el título y en el de la revista, el nombre. Un libro tiene un autor, del que interesa su nombre [nom_autor] y su nacionalidad (que se indica en un campo como texto libre). Un libro tiene como identificador un ISBN [isbn], y una revista, un ISSN [issn]. El diagrama E-R debe incluir las entidades PUBLICACIÓN, LIBRO, REVISTA y AUTOR.



Recurso digital

En el anexo web 5.3, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás una explicación detallada de algunos aspectos del ODL de ODMG 3.0.

5.5. OQL

OQL es un lenguaje de consulta incluido en el estándar ODMG 3.0. Tiene una sintaxis inspirada en la sentencia `SELECT` de SQL, pero adaptada para su uso con objetos. El *language binding* permite realizar consultas en OQL y recuperar los resultados mediante un iterador, como objetos que se pueden utilizar con el lenguaje de programación. OQL no incluye sentencias similares a las sentencias `INSERT`, `UPDATE` o `DELETE` de SQL. En lugar de ello, y según el principio de persistencia transparente de ODMG 3.0, las operaciones de creación, borrado y modificación de objetos persis-

tentes se realizan con el lenguaje de programación, cuyo *language binding* garantiza que los cambios realizados sobre ellos dentro de una transacción se reflejen en la base de datos al confirmarse esta.

OQL tiene el equivalente de casi todas las opciones de la sentencia SELECT de SQL y es, por tanto, un lenguaje de consulta muy potente. No se explicará aquí en detalle, sino que se incluirán ejemplos sencillos para ilustrar su funcionamiento general y sus posibilidades.

Las consultas sobre bases de datos se realizan utilizando puntos de entrada. Un punto de entrada da acceso a un objeto o una colección de objetos almacenados. Como punto de entrada puede servir la extensión (*extent*) de una clase que, como ya se ha comentado, permite acceder a todas sus instancias. A continuación, se indica cómo se incluiría en la definición de las clases del esquema de ejemplo la definición de una extensión asociada y de una clave para ella.

```
class Proyecto(extent proyectos key id_proy)
class Empleado(extent empleados key dni)
class EmpleadoPlantilla(extent empleados_plantilla key numEmp) extends
    Empleado
```

La siguiente consulta obtiene los jefes de proyectos con fecha de inicio desde 2017.

```
SELECT p.jefe_proyecto
FROM proyectos p
WHERE p.f_inicio >= date '2017-01-01';
```

Se utiliza como punto de entrada a la base de datos la extensión `proyectos` de la clase `Proyecto`, con alias `p`. El resultado es un objeto de tipo `bag<EmpleadoPlantilla>`. En un `bag` puede haber elementos repetidos. Para evitarlo se puede utilizar `SELECT DISTINCT` en lugar de `SELECT`, y entonces se obtendría un objeto de tipo `set<EmpleadoPlantilla>`.

Se puede acceder a una propiedad de un objeto añadiendo un punto y el nombre de la propiedad. Una propiedad puede ser de un tipo atómico, o un objeto o una relación con otros objetos, en cuyo caso se puede acceder a su vez a sus propiedades de la misma forma. Utilizando este mecanismo repetidamente se puede construir un *path expression* para navegar por la estructura de objetos complejos recuperados a partir del punto de entrada. El siguiente ejemplo recupera el número de empleado de los jefes de proyecto recuperados en el ejemplo anterior, y devolvería un objeto de tipo `set<String>`.

```
SELECT DISTINCT p.jefe_proyecto.num_emp
FROM proyectos p
WHERE p.f_inicio >= date '2017-01-01';
```

Si se quisiera recuperar, además, el nombre de los jefes de proyecto, habría que recuperar ambas cosas en un tipo `struct(String, String)`.

```
SELECT DISTINCT struct(num: p.jefe_proyecto.num_emp, nom: p.jefe_proyecto.
    nom_emp)
FROM proyectos p
WHERE p.f_inicio >= date '2017-01-01';
```

En cada consulta anterior el tipo de objeto devuelto es distinto. En la última hay que utilizar incluso una palabra reservada del OQL (`struct`) para especificar el tipo de objeto devuelto. Todo esto dificulta la composición de consultas, es decir, la creación de nuevas consultas basadas en consultas existentes. En SQL, en cambio, las consultas operan con relaciones y obtienen relaciones. Esto simplifica también el desarrollo de programas de aplicación.

OQL incluye cláusulas análogas a `GROUP BY`, `HAVING` y `ORDER BY` de SQL, y los operadores de agregación `min`, `max`, `count`, `sum`, `avg`.

5.6. Consulta y manipulación de datos con el Java binding

Una implementación del Java *binding*, conforme al estándar ODMG 3.0, debe proporcionar una clase que implemente la interfaz `org.odmg.Implementation`. Esta tiene métodos para crear y recuperar bases de datos (`Database`) y transacciones (`Transaction`). Tiene también un método que permite crear consultas de OQL (`OQLQuery`).

La manipulación de objetos persistentes se realiza según el principio de persistencia transparente. Eso significa que no hay un lenguaje específico para manipulación de objetos persistentes (OML), sino que estos se manipulan desde el lenguaje de programación, y de igual manera que los objetos no persistentes. El *language binding* garantiza que, al realizar una operación `commit` para confirmar los cambios realizados en la transacción actual, se graban en la base de datos todos los cambios realizados sobre los objetos persistentes. También se graban los cambios realizados sobre objetos alcanzables siguiendo las referencias desde cualquier objeto persistente. Esto se conoce como *persistence by reachability* o *transitive persistence* (persistencia por alcanzabilidad o persistencia transitiva).

Las clases persistentes o *persistent-capable* (habilitadas para persistencia) se pueden generar de diversas formas. Una es mediante un preprocesador de ODL que genere su código a partir de definiciones en ODL, lo que no impide que se puedan modificar posteriormente.

Para ilustrar todos estos aspectos se desarrollarán varios ejemplos con el Java *binding* de Matisse, que es en gran medida conforme a ODMG 3.0.

5.7. La base de datos de objetos Matisse

Matisse es una de las BDO más veteranas y con mejor soporte para ODMG 3.0. Es una base de datos de pago pero se puede descargar, previo registro, para su evaluación. Cuenta con una magnífica y muy completa documentación, enlazada desde la ayuda del propio programa.



WWW

Recursos web

Los siguientes enlaces proporcionan acceso a la descarga de software y a toda la documentación para administradores de bases de datos y desarrolladores.

<http://www.matisse.com/developers/downloads>

<http://www.matisse.com/developers/documentation>

La empresa que desarrolla Matisse la define como base de datos posrelacional. Pero, ante todo, y es lo que interesa especialmente aquí, es una potente BDO que soporta en gran medida los diversos aspectos del estándar ODMG 3.0.

- ODL: su implementación es, en gran medida, conforme al planteamiento de ODMG 3.0, aunque usa una sintaxis distinta y presenta algunas discrepancias.
- OQL: Matisse no implementa el OQL de ODMG 3.0. Proporciona como alternativa un *driver JDBC* que permite recuperar colecciones de objetos de la base de datos mediante su lenguaje SQL. Matisse llama SQL a un lenguaje propio que es una adaptación de SQL para BDO, y que incluye no solo una sentencia SELECT como OQL, sino también sentencias **INSERT**, **UPDATE** y **DELETE**.
- *Language bindings*: Matisse proporciona *language bindings* no solo para los incluidos en ODMG 3.0 (C++, Smalltalk y Java, este último con soporte para Java 8 en su versión 9.1.1), sino también para C, C#, Eiffel, Perl, PHP, Python y Visual Basic.

En los siguientes apartados se muestra cómo crear un esquema de objetos en una base de datos de Matisse mediante el ODL de Matisse, y cómo utilizar su Java *binding* para crear programas que se conectan a ella para crear, modificar y borrar tanto objetos como relaciones entre ellos. Para ello será necesario instalar el Enterprise Manager.

5.7.1. Creación de una base de datos mediante ODL con Matisse

Primero, se crea la base de datos y, después, el esquema de objetos en ella mediante ODL. Para crear la base de datos desde Enterprise Manager, se pulsa con el botón derecho del ratón sobre el servidor y se selecciona “New Database”. Como nombre se indica **AcDat_BDO**. Después, se arranca la base de datos pulsando sobre ella y con la opción “Start”. También están disponibles estas opciones en una barra de iconos en la parte de arriba.

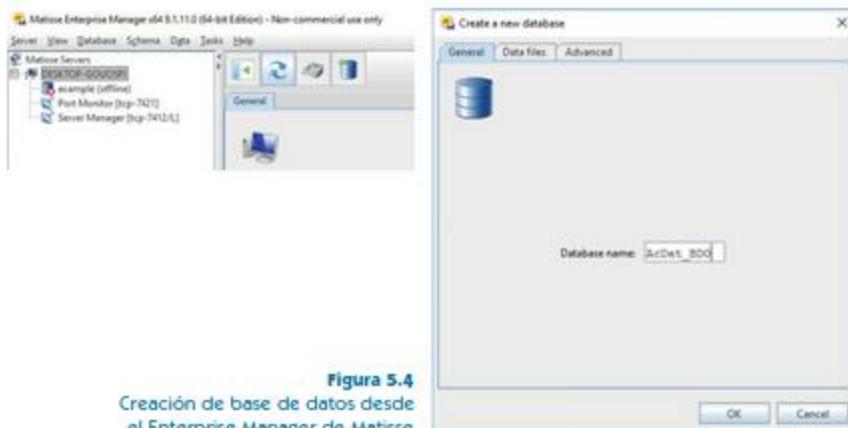


Figura 5.4
Creación de base de datos desde el Enterprise Manager de Matisse

www

Recurso web

En la siguiente dirección se puede encontrar un documento acerca del ODL de Matisse.

http://www.matisse.com/pdf/developers/matisse_odl.pdf

Las sentencias del ODL de Matisse que permiten crear el esquema de objetos para el modelo de objetos de ejemplo son las siguientes:

```
module gest_proyectos {
    interface Proyecto: persistent
    {
        attribute String<32> nom_proy;
        attribute Date f_inicio;
        attribute Date Nullable f_fin;
        relationship set<Empleado> tiene_asignado[0,-1]
            inverse Empleado::asignado_a;
        relationship EmpleadoPlantilla jefe_proyecto
            inverse EmpleadoPlantilla::gestiona;
    };
    interface Empleado: persistent {
        attribute String<9> dni;
        attribute String<60> nom_emp;
        relationship set<Proyecto> asignado_a[0,-1]
            inverse Proyecto::tiene_asignado;
        relationship DatosProfesionales tiene_datos_prof[0,1]
            inverse DatosProfesionales::datos_prof_de;
        mt_index Empleado_pk unique_key TRUE criteria { dni MT_ASCEND };
        mt_index Empleado_i_nom_emp criteria { nom_emp MT_ASCEND };
    };
    interface DatosProfesionales: persistent {
        attribute String<9> dni;
        attribute String<2> categoria;
        attribute Float sueldo_bruto_anual;
        relationship Empleado datos_prof_de
            inverse Empleado::tiene_datos_prof;
        mt_index Empleado_DatosProf_pk unique_key TRUE criteria { dni MT_ASCEND };
    };
    interface EmpleadoPlantilla: Empleado: persistent {
        attribute String<12> num_emp;
        relationship set<Proyecto> gestiona
            inverse Proyecto::jefe_proyecto;
        mt_index EmpleadoPlantilla_i_dni unique_key TRUE criteria { dni
            MT_ASCEND };
        mt_index EmpleadoPlantilla_i_nom_emp criteria { Empleado::nom_emp
            MT_ASCEND };
    };
}
```

Solo se admiten valores nulos para un atributo si se define con la opción `Nullable`. Por defecto, la cardinalidad mínima de una relación en cualquiera de sus dos extremos o *traversal paths* es 1, y no hay cardinalidad máxima si se define mediante un tipo de colección (`set` o `list`). Se puede cambiar cualquiera de las cardinalidades por defecto indicándolas con `[min, máx]`. Un valor -1 para `máx` indica que no hay cardinalidad máxima.

Matisse proporciona un identificador único (OID) para cada objeto, único globalmente para todos los objetos existentes en la base de datos y que, por tanto, puede hacer las veces de clave primaria para cualquier clase. Matisse permite definir índices (`mt_index`), sobre un atributo o un conjunto de ellos, que aceleran las búsquedas basadas en sus valores. Los índices pueden ser únicos (de tipo `unique_key`), y entonces sirven como clave primaria, lo que impide que haya más de un objeto con los mismos valores para ellos. En la clase `Empleado` se ha definido un índice único sobre `dni` porque cada persona tiene un DNI distinto, y otro para acelerar búsquedas por nombre. No se ha definido ningún índice único para `Proyecto`. En una base de datos relacional hubiera sido necesario crear en una tabla `proyectos` un atributo para guardar un identificador único, y utilizarlo para la clave primaria.

El esquema de objetos se puede crear importando un fichero con las declaraciones en ODL. El nombre del módulo (en este caso, `gest_proyectos`) es el *namespace* o espacio de nombres, que se creará en la base de datos. El esquema se crea con el botón derecho del ratón sobre la base de datos recién creada, seleccionando "Schema", "Import ODL Schema...". Una vez creado el esquema, se puede utilizar el Data Modeller para obtener diagramas de clases. Este es un programa independiente del Enterprise Manager, y hay que instalarlo aparte.

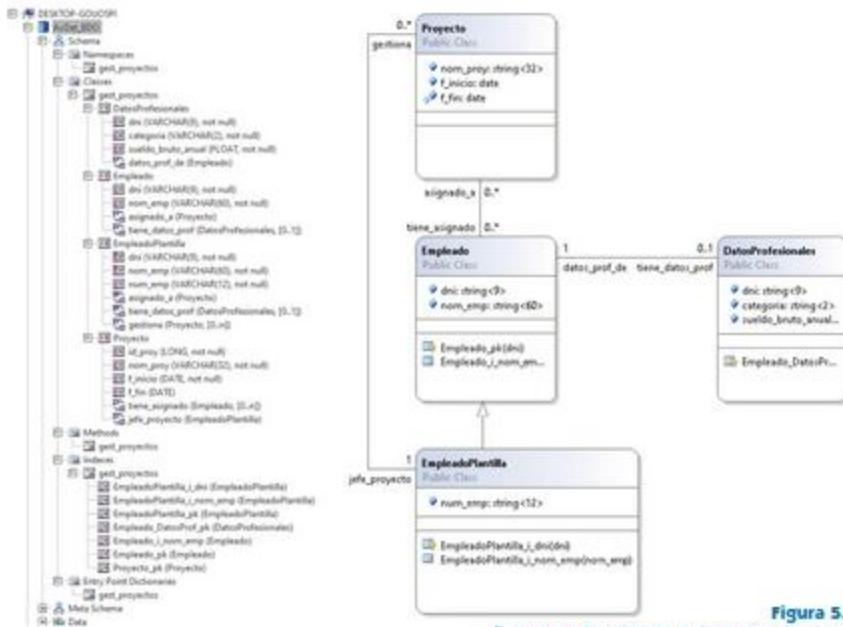


Figura 5.5
Esquema de objetos y diagrama de clases

Actividad propuesta 5.2



Define, utilizando el ODL de Matisse, el esquema de objetos para el modelo de objetos creado en la actividad anterior para publicaciones (libros y revistas), incluyendo todas las clases y los atributos indicados para ellas. Importa el esquema y visualízalo en el Enterprise Manager. Obtén el diagrama de clases con el Data Modeller.

5.7.2. Utilización de la base de datos mediante el Java binding de Matisse

Matisse tiene un Java *binding* muy bien documentado, cuya documentación se puede encontrar en la sección para desarrolladores (*developers*) de su página web, cuyo enlace se ha proporcionado anteriormente.



Recursos web

El primero de los siguientes enlaces corresponde a una guía de programación en Java para Matisse con numerosos ejemplos. El segundo proporciona el código fuente completo de estos ejemplos. El tercero es una aplicación para Java EE que se puede desplegar en el servidor de aplicaciones Tomcat, y que permite diversos tipos de consulta sobre los contenidos de una base de datos de Matisse. Se hablará más acerca de Java EE, y del despliegue de aplicaciones en esta plataforma, en un capítulo posterior dedicado a componentes para acceso a datos.

http://www.matisse.com/pdf/developers/java_pg.pdf
http://www.matisse.com/pdf/developers/java_examples.zip
http://www.matisse.com/pdf/developers/jsp_demo.zip



Matisse genera clases de Java para objetos persistentes (*stub classes*) a partir sus definiciones en ODL. Se hace pulsando con el botón derecho sobre la base de datos y seleccionando "Schema", "Generate Code".

Las clases generadas contienen código que no hay que modificar entre comentarios `// BEGIN Matisse SDL Generated Code` y `// END of Matisse SDL Generated Code`. Este código se encarga de la persistencia de objetos. Fuera de ahí, se pueden añadir métodos adicionales a las clases y recompilarlas, y seguirán funcionando.

Figura 5.6

Generación de código (*stub classes*)

perfectamente con la base de datos. A continuación, se indican los métodos más importantes disponibles en las *stub classes* generadas por Matisse para Java.

Matisse no proporciona una implementación de la interfaz `org.odmg.Implementation` conforme al estándar ODMG 3.0. En lugar de ello, proporciona una clase `com.matisse.MtDatabase` en `matisse.jar`.

CUADRO 5.2
Métodos más importantes de las *stub classes* generadas por Matisse para Java

	Método(s)	Funcionalidad
Para cada clase. Son de tipo <code>public static</code> . <code>Clase</code> es el nombre de la clase. Permiten crear objetos persistentes e iterar sobre los objetos persistentes de la clase	<code>Clase(MtDatabase db)</code> <code>getInstanceNumber(MtDatabase db)</code> <code>getOwnInstanceNumber (MtDatabase db)</code> <code>instanceIterator(MtDatabase db)</code> <code>ownInstanceIterator(MtDatabase db)</code>	Crea un objeto de la clase. <code>Clase</code> indica el nombre de la clase. Obtienen el número de instancias de la clase existente en la base de datos. Aquellos métodos en cuyo nombre aparece <code>Own</code> solo tienen en cuenta las instancias de la propia clase, pero no de las subclases. Obtienen iteradores para las instancias de la clase. El primero incluye instancias de subclases, el segundo no.
Para índices. Son de tipo <code>public static</code> . Permiten acceder directamente a objetos conociendo el valor de determinados atributos. <code>índice</code> es el nombre de un índice	<code>Clase lookupindice(MtDatabase db, TipoAtrib1 valorAtrib1, TipoAtrib2 valorAtrib2,...)</code> <code>Clase[] lookupObjectsíndice (MtDatabase db, TipoAtrib1 valorAtrib1, TipoAtrib2 valorAtrib2,...)</code> <code>índiceIterator(MtDatabase db, TipoAtrib1 valorAtrib1, TipoAtrib2 valorAtrib2,...)</code>	Recuperar objetos dados los valores para atributos incluidos en el índice. Los métodos del primer tipo permiten recuperar un objeto, y son especialmente útiles para índices únicos. Los del segundo tipo permiten recuperar varios objetos, y son de utilidad para índices no únicos. Devuelve un iterador sobre objetos. Existe otra función que permite especificar opciones adicionales, y que no se describe aquí.
Para cada atributo. <code>Atr</code> es el nombre del atributo	<code>getAtr()</code> <code>setAtr()</code> <code>removeAtr()</code> <code>isAtrNull()</code> <code>isAtrDefault()</code>	Métodos <code>getter</code> y <code>setter</code> . Elimina el valor del atributo y le asigna su valor por defecto. Comprueban si el valor del atributo es nulo (<code>null</code>) y si es el valor definido por defecto.

[.../...]

CUADRO 5.2 (CONT.)

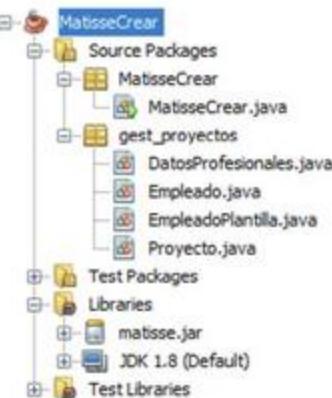
Para todas las relaciones. Rel es el nombre de la relación	<code>clearRel()</code>	Elimina la relación con todos los sucesores según la relación. Un sucesor de un objeto según una relación es un objeto con el que está relacionado de acuerdo a esa relación. No borra los sucesores, solo elimina la relación del objeto en cuestión con ellos.
Para relaciones con cardinalidad máxima 1. TipoSuc es el tipo de los sucesores según la relación	<code>getRel()</code> <code>setRel(TipoSuc succ)</code>	El primer método obtiene el sucesor según la relación. El segundo establece la relación con un objeto dado.
Para relaciones con cardinalidad máxima mayor que 1	<code>TipoSuc[] getRel()</code> <code>relIterator()</code> <code>getRelSize()</code> <code>setRel(TipoSuc[] sucesores)</code> <code>prependRel(TipoSuc sucesor)</code> <code>appendRel(TipoSuc sucesor)</code> <code>appendRel(TipoSuc[] sucesores)</code> <code>removeRel(TipoSuc sucesor)</code> <code>removeRel(TipoSuc [] sucesores)</code>	Obtiene en un array sucesores según la relación Rel . Obtiene iterador para sucesores según la relación Rel . Obtiene número de sucesores según la relación Rel . Asigna los sucesores según la relación Rel . Añade sucesor según relación Rel , en primera posición, es decir, antes que los ya existentes. Añade sucesor según relación Rel en última posición, es decir, después de todos los ya existentes. Añade múltiples sucesores según relación Rel después de todos los ya existentes. Elimina sucesor según la relación Rel . Elimina un sucesor según la relación Rel .

Para construir un programa que utilice el Java *binding* de Matisse hay que incluir en el proyecto **matisse.jar** y las *stub classes*, como se muestra en la figura.

A continuación, se proporcionan ejemplos del uso del Java *binding* de Matisse para abrir una conexión a una base de datos y, dentro de una transacción, y utilizando las funciones anteriores, crear, recuperar, modificar y borrar objetos, y gestionar las relaciones entre ellos.

El siguiente programa abre una conexión con la base de datos y crea un proyecto y varios empleados, tanto de plantilla como no de plantilla, que se asignan al proyecto. Como jefe de proyecto se asigna uno de los empleados de plantilla. Las cardinalidades de las relaciones se verifican al confirmar los cambios en la base de datos. Se produciría una excepción, por ejemplo, si no se asignara un jefe de proyecto a un proyecto. Se resaltan las sentencias para abrir la conexión a la base de datos y crear y confirmar una transacción, y también para gestionar las relaciones entre objetos.

Figura 5.7
Proyecto para Matisse con stub classes.



```
// Creación de empleados y proyectos, y asignación de empleados a proyectos
package MatisseCrear;

import com.matisse.MtDatabase;
import com.matisse.MtException;
import gest_proyectos.*;

public class MatisseCrear {
    public static void main(String[] args) {
        try(MtDatabase db = new MtDatabase("localhost", "AcDat_BDO")) {
            db.open();
            db.startTransaction();
            Proyecto pl = new Proyecto(db);
            pl.setNom_proy("PAPEL ELECTRÓNICO");
            pl.setF_inicio(new java.util.GregorianCalendar(2018,12,01));
            EmpleadoPlantilla jpl = new EmpleadoPlantilla(db);
            jpl.setDni("78901234X");
            jpl.setNom_emp("NADEALES");
            jpl.setNum_emp("604202");
            pl.setJefe_proyecto(jpl);
            Empleado el=new Empleado(db);
            el.setDni("56789012B");
            el.setNom_emp("SAMPER");
            pl.appendTiene_asignado(el);
            EmpleadoPlantilla e2=new EmpleadoPlantilla(db);
            e2.setDni("76543210S");
            e2.setNom_emp("SILVA");
            e2.setNum_emp("753014");
            DatosProfesionales dp2 = new DatosProfesionales(db);
            dp2.setDni("76543210S");
            dp2.setCategoria("B1");
            dp2.setSueldo_bruto_anual((float) 45200.00);
            e2.setTiene_datos_prof(dp2);
            pl.appendTiene_asignado(e2);
            Empleado e3=new Empleado(db);
```

```

        e3.setDni("89012345E");
        e3.setNom_emp("ROJAS");
        db.commit();
    }
    catch (MtException mte)
    {
        System.out.println("MtException: " + mte.getMessage());
    }
}
}

```

Actividades propuestas



- 5.3.** Añade un método `float subeSueldoBruto(float incr)` a la clase `Empleado`, al que se le pase el porcentaje de incremento. Si el sueldo bruto está definido, debe devolver el nuevo sueldo, y `null` en otro caso. Escribe y verifica un programa que actualice el sueldo bruto para un empleado utilizando el nuevo método.
- 5.4.** Piensa/investiga/experimenta/verifica: todos los objetos creados por el programa de ejemplo anterior se crean como objetos persistentes. ¿Cómo se crearían objetos transitorios (es decir, no persistentes) de las mismas clases? ¿Es posible que un objeto creado como transitorio se acabe almacenando en la base de datos? ¿En qué casos? Justifica tus respuestas y verificalas, en su caso, mediante un programa.
- 5.5.** Genera las *stub classes* para el esquema de objetos creado en una actividad anterior para publicaciones. Crea un programa en Java que cree al menos tres libros, dos de ellos del mismo autor, y dos revistas en la base de datos y que, para terminar, muestre toda la información acerca de todos los objetos creados.

El Object Browser del Enterprise Manager de Matisse permite visualizar los objetos creados y las relaciones entre ellos. Se pueden seleccionar objetos mediante consultas en SQL, visualizar sus propiedades y acceder a los objetos relacionados. Como se puede ver, desde un objeto se puede acceder a los objetos relacionados siguiendo los *traversal paths*.

Figura 5.8
Consulta
de objetos
en el Object
Browser
de Matisse

Object	Value	Type
selection	[Length=1]	Proyecto[]
[0]	[0x10a8 Proyecto]	Proyecto
nom_proy	JAPEL ELECTRÓNICO	MT_STRING
f_inicio	2019-01-01	MT_DATE
f_fin	0x_NULL	MT_DATE
[1] bene_asignado	[Length=2]	Empleado[]
[0]	[0x10ab Empleado]	Empleado
dni	56793012B	MT_STRING
nom_emp	SANTPER	MT_STRING
[1] asignado_a	[Length=1]	Proyecto[]
[2] tiene_datos_prof	[Length=0]	DatosProfesionales
[1]	[0x10ab EmpleadoPlantilla]	EmpleadoPlantilla
dni	745432105	MT_STRING
nom_emp	SSIVIA	MT_STRING
num_emp	783014	MT_STRING
[1] asignado_a	[Length=1]	Proyecto[]
[2] tiene_datos_prof	[Length=1]	DatosProfesionales
[3] gestiona	[Length=0]	Proyecto[]
[2] jefe_proyecto	[Length=1]	EmpleadoPlantilla
[0]	[0x10ab EmpleadoPlantilla]	EmpleadoPlantilla
dni	78901234X	MT_STRING
nom_emp	ROSALES	MT_STRING
num_emp	404202	MT_STRING
[1] asignado_a	[Length=0]	Proyecto[]
[2] tiene_datos_prof	[Length=0]	DatosProfesionales
[3] gestiona	[Length=1]	Proyecto[]

Figura 5.9
Navegación
por los objetos
con el Object Browser
de Matisse

El siguiente programa borra y modifica algunos objetos y elimina algunas relaciones entre objetos, y utiliza iteradores de varios tipos. Recupera empleados de plantilla a partir de su DNI con `lookupEmpleadoPlantilla_i_dni`, que se ha creado en `EmpleadoPlantilla` porque se ha definido el índice `EmpleadoPlantilla_i_dni`. Recupera empleados a partir de su DNI con `lookup_empleado_pk()`, que se ha creado porque DNI se ha definido como clave primaria de `Empleado` (`unique_key`). Antes de borrar un empleado, es necesario borrar, si existen, sus datos profesionales. Si no, se produce una excepción, dado que la cardinalidad mínima en el `traversal path datos_prof_de` es 1. También obtiene y utiliza iteradores. Primero, obtiene todos los proyectos con `instanceIterator()` de la clase `Proyecto`. Para cada proyecto obtiene los empleados asignados a él con un iterador obtenido con el método `tiene_asignadoIterator()` de la clase `Proyecto`, que se ha creado para el `traversal path tiene_asignado` de la clase `Proyecto`.

```
// Borrado objetos y relaciones entre ellos
import com.matisse.MtDatabase;
import com.matisse.MtException;
import com.matisse.MtObjectIterator;
import gest_proyectos.*;
public class MatisseModifBorrar {
    public static void muestraProyecto(Proyecto p) {
        System.out.println("Proyecto "+p.getNom_proy()+
            "[OID: "+p.getMtOidToHexString()+"]");
    }
}
```

```

        System.out.println("-----");
        System.out.println("Jefe proyecto: DNI: "+p.getJefe_proyecto().getDni()+
                           ", Nombre: "+p.getJefe_proyecto().getNom_emp());
        System.out.println("Empleados:");
        MtObjectIterator<Empleado> itEmp = p.tiene_asignadoIterator();
        while(itEmp.hasNext()) {
            Empleado e = itEmp.next();
            System.out.println("DNI: "+e.getDni()+" , Nombre: "+e.getNom_emp());
        }
    }

    public static void main(String[] args) {
        try(MtDatabase db = new MtDatabase("localhost", "AcDat_BDO")) {
            db.open();
            db.startTransaction();
            Proyecto p = new Proyecto(db);
            p.setNom_proy("TINTA HOLOGRÁFICA");
            p.setF_inicio(new java.util.GregororianCalendar(2018,12,28));
            EmpleadoPlantilla ep = // NADALES
            EmpleadoPlantilla.lookupEmpleadoPlantilla_i_dni(db,
                "78901234X");
            p.setJefe_proyecto(ep);
            Empleado el = Empleado.lookupEmpleado_pk(db, "89012345E"); //
            ROJAS
            el.setNom_emp("ROJAS");
            el.appendAsignado_s(p);
            Empleado e2 = Empleado.lookupEmpleado_pk(db, "765432108"); //
            SILVA
            e2.getTiene_datos_prof().remove();
            e2.remove();
            Empleado e3 = Empleado.lookupEmpleado_pk(db, "56789012B"); //
            SAMPER
            e3.clearAsignado_s();
            MtObjectIterator<Proyecto> itProy = Proyecto.
                instanceIterator(db);
            while(itProy.hasNext()) {
                Proyecto unProy = itProy.next();
                muestraProyecto(unProy);
            }
            db.commit();
        }
        catch (MtException mte)
        {
            System.out.println("MtException: " + mte.getMessage());
        }
    }
}

```

Actividades propuestas



- 5.6.** Crea un programa que cree un nuevo libro y le asigne como autor uno de los autores ya existentes. El programa debe también crear un nuevo autor, que no tendrá ningún libro

asociado. El programa debe borrar uno de los dos libros del autor para el que había dos libros. Por último, como comprobación, debe mostrar toda la información acerca de todos los objetos existentes. Se sugiere mostrar autor a autor, y para cada autor sus libros.

- 5.7.** Existe un método `deepRemove()` en cada clase que por defecto llama a `remove()`. Este se puede cambiar, cuando tenga sentido, para borrar objetos subordinados antes de llamar a `remove()` para borrar el propio objeto. Cambia `deepRemove()` en la clase `Empleado` para borrar antes los datos profesionales. Modifica el programa de ejemplo anterior para que utilice este método para borrar el empleado.

5.7.3. Consultas mediante el SQL de Matisse

Matisse no proporciona soporte para OQL conforme al estándar ODMG 3.0. En lugar de ello, tiene un lenguaje para consulta, definición y manejo de datos al que se llama SQL, similar al SQL estándar de las bases de datos relacionales, pero adaptado para BDO, y un *driver* JDBC. El lenguaje SQL de Matisse tiene sentencias `SELECT`, `INSERT`, `UPDATE` y `DELETE`. No hay que dejarse confundir por el nombre SQL ni por su similitud al SQL estándar para bases de datos relacionales, ni porque se denomine a Matisse base de datos posrelacional. Matisse es una BDO, y lo que se llama SQL de Matisse no es realmente SQL. En la parte para desarrolladores de la web de Matisse, para la que se ha proporcionado antes un enlace, se puede encontrar documentación acerca del SQL de Matisse, y acerca del Java *binding* de Matisse y programación en Java para Matisse en general, incluyendo el uso de su *driver* de JDBC.

Recursos web



En los siguientes enlaces se puede encontrar documentación detallada acerca del lenguaje SQL de Matisse y una guía de programación en Java para Matisse que dedica un capítulo específicamente a JDBC.

http://www.matisse.com/pdf/developers/sql_pg.pdf
http://www.matisse.com/pdf/developers/java_pg.pdf

La clase para el *driver* JDBC es `com.matisse.sql.MtDriver`. Se puede cargar el *driver* a la manera antigua con `Class.forName("com.matisse.sql.MtDriver")` o, lo más sencillo, obtener una conexión JDBC con `getJDBCConnection()` de `MtDatabase`. El siguiente programa crea un nuevo empleado y lo asigna a todos los proyectos de los que es jefe un determinado empleado, identificado por su DNI. Para obtener los proyectos como objetos hay que utilizar en la consulta `REF(p)`, que devuelve una referencia a cada objeto de clase `Proyecto`, que se obtiene con `getObject()`. El planteamiento con una base de datos estrictamente conforme a ODMG 3.0 sería análogo: recuperar los proyectos mediante una consulta en OQL, hacer los cambios pertinentes sobre ellos y confirmar los cambios.

```

// Recuperación de datos mediante JDBC y modificaciones sobre datos obtenidos
package MatisseConsultaConJDBCModifConJavaBinding;

import com.matisse.MtDatabase;
import com.matisse.MtException;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import gest_proyectos.*;

public class MatisseConsultaConJDBCModifConJavaBinding {

    public static void muestraErrorSQL(SQLException e) {
        System.err.println("SQL ERROR mensaje: " + e.getMessage());
        System.err.println("SQL Estado: " + e.getSQLState());
        System.err.println("SQL código específico: " + e.getErrorCode());
    }

    public static void main(String[] args) {
        try {
            MtDatabase db = new MtDatabase("localhost", "AcDat_BDO");
            db.open();
            db.startTransaction();
            Empleado e = new Empleado(db);
            e.setDni("654321098");
            e.setNom_emp("LUQUE");
            try {
                Connection jdbcCon = db.getJDBCConnection();
                Statement stmt = jdbcCon.createStatement();
                String commandText = "SELECT REF(p) FROM gest_proyectos."
                    + "Proyecto p WHERE p.jefe_proyecto.dni='78901234x'";
                ResultSet rset = stmt.executeQuery(commandText);
                Proyecto p;
                while (rset.next()) {
                    p = (Proyecto) rset.getObject(1);
                    System.out.println("Proyecto: "+p.getNom_proy()+
                        ", jefe: ["+p.getJefe_proyecto().getDni()+"] "+
                        p.getJefe_proyecto().getNom_emp());
                    if(p.getJefe_proyecto().getDni().equals("78901234x")) {
                        p.appendTiene_asignado(e);
                        System.out.println("Asignado nuevo empleado.");
                    }
                }
                rset.close();
                stmt.close();
                db.commit();
            } catch (SQLException sqle) {
                muestraErrorSQL(sqle);
            }
        } catch (MtException mte) {
            System.out.println("MtException: " + mte.getMessage());
        }
    }
}

```

5.8. SQL:99 y bases de datos objeto-relacionales

En SQL:99, también conocido como SQL3, la cuarta revisión del estándar SQL, se introdujeron los tipos estructurados definidos por el usuario. Esto significa que los atributos de una tabla pueden tener tipos no atómicos o estructurados, además de tipos atómicos como `CHAR`, `VARCHAR2`, `INTEGER`, etc. En una tabla puede haber columnas que contengan objetos, *arrays* o incluso tablas (tablas anidadas), y también referencias a objetos, de manera que en varios lugares diferentes de la base de datos puedan existir referencias al mismo objeto, almacenado en un único lugar en la base de datos. Estas características, por supuesto, se salen completamente del modelo relacional. Pero si no se utilizan en una base de datos, esta sigue siendo relacional.

Las bases de datos que implementan, en mayor o menor medida, estas características se pueden calificar como objeto-relacionales. Entre ellas cabe destacar Oracle.

Oracle implementa todas estas características en una capa de software sobre una base de datos relacional.

Recursos web



Las últimas versiones de la base de datos Oracle son 11g, 12c y 18c. Existen versiones XE de 11g y 18c. Son versiones simplificadas y limitadas que pueden utilizarse de manera gratuita y sin soporte técnico de Oracle. Para un primer contacto puede ser buena opción la versión 11g XE. El siguiente enlace dirige a la página de descargas para las distintas versiones de Oracle:

<https://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html>

El siguiente enlace proporciona la documentación más importante de la versión 12c:

<https://docs.oracle.com/en/database/oracle/oracle-database/12.2/books.html>

Es de especial interés la documentación de SQL, una vez que se haya completado el proceso de instalación y configuración inicial:

<http://docs.oracle.com/en/database/oracle/oracle-database/12.2/sqlqr/sql-language-quick-reference.pdf>

<http://docs.oracle.com/en/database/oracle/oracle-database/12.2/sqlrf/sql-language-reference.pdf>

5.9. Características objeto-relacionales de Oracle

Oracle permite definir tipos de objetos (clases), tablas de objetos (que almacenan un objeto de una clase determinada en cada fila), tablas con columnas de objetos (que contienen objetos de una clase determinada en una columna), y tablas con columnas que contienen colecciones, tanto *arrays* como tablas (tablas anidadas). Pero en última instancia se trata de una base de datos relacional y todos estos datos se almacenan en una base de datos puramente relacional.



Recurso digital

En el anexo web 5.4, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás la "Guía del desarrollador objeto-relacional".

A continuación, se explicará la forma de definir, utilizando el SQL de Oracle, tipos estructurados definidos por el usuario, tales como los introducidos en SQL:99, entre ellos tipos de objetos, y se utilizarán para implementar el modelo de objetos de ejemplo que se ha venido utilizando hasta ahora y para introducir algunos datos de ejemplo.

5.9.1. Tipos de objetos

Se crean con `CREATE TYPE ... AS OBJECT`. Un tipo de objeto es un tipo estructurado que tiene atributos y métodos, es decir, una clase. Hay que terminar las declaraciones de tipo con el carácter /. Se pueden crear tipos de objeto, que se utilizarán para la definición de las tablas, como sigue:

```
CREATE TYPE DatosProfesionales AS OBJECT {
    categoria CHAR(2),
    sueldo_bruto_anual NUMBER(8, 2)
};

CREATE TYPE Empleado AS OBJECT {
    dni CHAR(9),
    nom_emp VARCHAR2(60),
    datos_prof DatosProfesionales
} NOT FINAL;
```

El propósito de la opción `NOT FINAL` en la definición de `Empleado` se verá en breve.

En estas definiciones de tipos no tienen sentido restricciones sobre el valor que puedan tomar los atributos, como por ejemplo que no puedan tomar valor nulo. Estas se introducen como restricciones de integridad en tablas basadas en estos tipos, como se verá en breve.

5.9.2. Herencia

Se pueden definir clases como subclases de otras. No se admite herencia múltiple, solo simple. Solo se pueden definir subclases de una clase dada si se define como `NOT FINAL`. Se puede definir la clase `EmpleadoPlantilla` como subclase de `Empleado` como sigue:

```
CREATE TYPE EmpleadoPlantilla UNDER Empleado(
    num_emp CHAR(12)
);
```

Si no se hubiera definido la clase Empleado como `NOT FINAL`, se podría cambiar con:

```
ALTER TYPE Empleado NOT FINAL;
```

5.9.3. Objetos de fila y objetos de columna

Se puede definir una tabla con la misma estructura de una clase (es decir, de un *object type* o tipo de objeto). Cada fila de la tabla contendrá un objeto de la clase, lo que se conoce como *row object* u objeto de fila. La base de datos genera un OID o identificador único para cada objeto de fila.

Un atributo de una tabla puede ser de un tipo de objeto. Esto es lo que se conoce como *object column* u objeto de columna. La base de datos no genera OID para objetos de columna.

Por ejemplo, se puede definir una tabla `empleados` cuyas filas contengan objetos del tipo de objeto `Empleado`, cuyo atributo `datos_prof` es del tipo `DatosProfesionales`, por lo que cada uno de estos objetos de fila contendrá un objeto de columna en `datos_prof`.

En la definición de la tabla se puede incluir la definición de clave primaria y restricciones sobre los valores que puedan tomar los atributos.

```
CREATE TABLE empleados OF Empleado(
    dni PRIMARY KEY,
    nom_emp NOT NULL
);
```

Se pueden insertar filas en la tabla de empleados de la siguiente forma:

```
INSERT INTO empleados VALUES (Empleado('56789012B', 'SAMPER', null));
INSERT INTO empleados VALUES (Empleado('76543210S', 'SILVA',
    DatosProfesionales('B1', 45200.00)));
INSERT INTO empleados VALUES (EmpleadoPlantilla('78901234X', 'NADALES', null,
    '604202'));
```

El ejemplo anterior muestra cómo en la tabla `empleados` se pueden introducir objetos de la clase `Empleado` y de su subclase `EmpleadoPlantilla`.

El operador `REF` permite obtener referencias a objetos de fila, es decir, sus OID.

```
SELECT REF(e) FROM empleados e;
```

Para objetos de columna no se crean referencias. La siguiente consulta es errónea:

```
SELECT REF(e.datos_prof) FROM empleados e;
```

5.9.4. Tipos de objetos con referencias a otros tipos de objetos

Las referencias permiten utilizar un mismo objeto en la definición de varios objetos. Por ejemplo: un mismo empleado de plantilla puede ser el jefe de proyecto de varios proyectos. Para reflejar esto en la base de datos, un objeto de tipo `EmpleadoPlantilla` se almacena una sola vez en la base de datos, en la tabla `empleados`, y en la tabla `proyectos` se almacenan varios objetos de tipo `Proyecto` que contienen una referencia a él. Guardar copias del mismo objeto de tipo `EmpleadoPlantilla` en varios objetos de tipo `Proyecto` no solo es inefficiente, sino erróneo. Una copia de un objeto es igual, pero no idéntica, no es el mismo objeto. Una referencia contiene el OID del objeto referenciado. El tipo `Proyecto` y la tabla `proyectos` se podrían definir de esta manera:

```
CREATE TYPE Proyecto AS OBJECT {
    nom_proy VARCHAR2(32),
    f_inicio DATE,
    f_fin DATE,
    jefe_proy REF EmpleadoPlantilla
};

CREATE TABLE proyectos OF Proyecto(
    nom_proy NOT NULL,
    f_inicio NOT NULL,
    jefe_proy SCOPE IS empleados
);
```

El jefe de proyecto de un proyecto se almacena como una referencia (`REF`) a un objeto de tipo `EmpleadoPlantilla` almacenado en la tabla `empleados` (`SCOPE IS empleados`).

Ahora se puede introducir un proyecto, incluyendo la referencia a su jefe de proyecto.

```
INSERT INTO proyectos
SELECT Proyecto('PAPEL ELECTRÓNICO', to_date('01/12/2018','dd/mm/YYYY'),
    NULL, TREAT(REF(e) AS REF EmpleadoPlantilla)) FROM empleados e WHERE
    DNI='78901234X';
```

El operador `TREAT` hace posible tratar la referencia recuperada como una referencia a un objeto de tipo `EmpleadoPlantilla` en lugar de como un objeto de su superclase `Empleado`.

Se puede obtener un objeto a partir de una referencia a él mediante el operador `DEREF`.

```
SELECT DEREF (p.jefe_proy)
FROM proyectos p WHERE p.nom_proy='PAPEL ELECTRÓNICO';
```

5.9.5. Tipos de datos de colección: VARRAY y tablas anidadas

A parte de tipos de objeto, es posible definir distintos tipos de colecciones, y se pueden definir atributos con esos tipos. En particular, se pueden definir tipos de *arrays* y de tablas.

Los tipos de *array* (`VARRAY`) son similares a los *arrays* de Java. Los elementos guardados en un `VARRAY` se pueden recuperar indicando un índice, siendo 1 el índice del primer elemento. Por

ejemplo, para cada empleado se podría guardar hasta un máximo de tres teléfonos, modificando la definición del tipo `Empleado` como sigue:

```
CREATE TYPE TelefonosEmpleado AS VARRAY(2) OF CHAR(14);
/
ALTER TYPE Empleado ADD ATTRIBUTE telefonos TelefonosEmpleado CASCADE;
```

La palabra reservada `CASCADE` se utiliza para que los cambios realizados en el tipo se propaguen a todos los tipos y tablas en los que este se utiliza. Esto implica, aparte del cambio de los tipos, la actualización de los contenidos de las tablas para adaptar los datos existentes a la nueva definición de los tipos.

Se podría añadir un nuevo empleado en la tabla de objetos `empleados` definida anteriormente de la siguiente forma:

```
INSERT INTO empleados VALUES( Empleado('89012345E', 'ROJAS', null,
    TelefonosEmpleado('654321098','959456789'))
);
```

Se pueden definir tipos de tablas. Si se define una columna de una tabla con un tipo de tabla, cada fila de la tabla contendrá en esa columna una tabla (tabla anidada o *nested table*). Por ejemplo, se podría añadir a la tabla `proyectos` una columna para la lista de empleados asignados al proyecto.

```
CREATE TYPE TablaAsigEmpleados AS TABLE OF REF Empleado;
/
ALTER TYPE Proyecto ADD ATTRIBUTE asig_empleados TablaAsigEmpleados CASCADE;
```

Se puede añadir un nuevo proyecto con la lista de empleados asignados de la siguiente manera:

```
UPDATE proyectos
SET asig_empleados=TablaAsigEmpleados
(
(SELECT REF(e) FROM empleados e WHERE DNI='56789012B'),
(SELECT REF(e) FROM empleados e WHERE DNI='76543210B')
)
WHERE nom_proy='PAPEL ELECTRÓNICO';
```



Actividad propuesta 5.8

Se quiere tener para cada asignación de un empleado a un proyecto la fecha de inicio y de fin. Elimina la columna `asig_empleados` de la tabla `proyectos` y añade una nueva columna con igual nombre que contenga también una tabla anidada con los empleados asignados al proyecto, pero en cada fila de esta tabla, además de la referencia al empleado, debe haber columnas con las fechas de inicio y de fin de asignación al proyecto (`f_ini` y `f_fin`). La referencia al empleado no puede ser nula, ni la fecha de inicio. Su clave primaria debe estar compuesta por la referencia

al empleado y la fecha de inicio. Se trata de definir un nuevo tipo `AsigEmpleado` que incluya la referencia al empleado y las dos fechas, y utilizarlo para definir el tipo `TablaAsigEmpleados`.

Las características objeto-relacionales de Oracle ofrecen muchas posibilidades. Con esta breve introducción se ha tratado solo de introducir las características fundamentales de las BDOR, basadas en SQL:99, y de dar una idea de cómo se puede crear con Oracle un esquema objeto-relacional con diversos tipos de objetos y diversos tipos de relaciones entre ellos.

Resumen

- Como solución al desfase objeto-relacional, es decir, al conjunto de dificultades que plantea la persistencia de objetos en bases de datos relacionales, han surgido diversas soluciones. Entre ellas, las BDO, las BDOR y la correspondencia objeto-relacional.
- Las BDO permiten almacenar directamente objetos.
- La organización ODMG desarrolló estándares para BDO, entre los que están ODL y OQL. No se incluye entre ellos ningún lenguaje para manipulación de objetos, sino *language bindings* para lenguajes de programación orientados a objetos –entre ellos Java–, que hacen posible la persistencia transparente. Es decir, se gestionan de igual manera los objetos persistentes y transitorios, y en el momento de confirmar una transacción se reflejan los cambios realizados sobre los objetos en la base de datos.
- ODL permite especificar los atributos de los objetos, sus métodos y las relaciones entre objetos, que pueden ser de tipo uno a uno, uno a mucho o muchos a muchos.
- En SQL:99 se introdujeron tipos estructurados definidos por el usuario, entre los que están los tipos objetos (clases), además de tipos de colecciones, tablas anidadas y referencias.
- Las BDOR, como Oracle, se basan en SQL:99 y amplían la funcionalidad de una base de datos relacional con tipos estructurados definidos por el usuario, entre los que están los objetos.



Ejercicios propuestos

1. En el siguiente diagrama E-R se representan las relaciones entre clientes, facturas y productos incluidos en las facturas. Cada línea de factura viene identificada por la línea de factura y un número secuencial dentro de la propia factura. Interesa guardar ambos, dado que aparecen en las facturas impresas y el orden de las líneas es relevante. Los productos son de dos tipos: los que se venden por unidades, identificados por su EAN, y los que se venden a granel (por kilos, litros, etc.), identificados por un código que puede contener letras y números. Para estos últimos interesa guardar la unidad de medida a la que se aplica el precio unitario. Crea un esquema de objetos para él, la definición del esquema de objetos en ODL de Matisse, e importa el esquema en una nueva base de datos en Matisse.

ejemplo, para cada empleado se podría guardar hasta un máximo de tres teléfonos, modificando la definición del tipo `Empleado` como sigue:

```
CREATE TYPE TelefonosEmpleado AS VARRAY(2) OF CHAR(14);
/
ALTER TYPE Empleado ADD ATTRIBUTE telefonos TelefonosEmpleado CASCADE;
```

La palabra reservada `CASCADE` se utiliza para que los cambios realizados en el tipo se propaguen a todos los tipos y tablas en los que este se utiliza. Esto implica, aparte del cambio de los tipos, la actualización de los contenidos de las tablas para adaptar los datos existentes a la nueva definición de los tipos.

Se podría añadir un nuevo empleado en la tabla de objetos `empleados` definida anteriormente de la siguiente forma:

```
INSERT INTO empleados VALUES( Empleado('89012345E', 'ROJAS', null,
    TelefonosEmpleado('654321098','959456789'))
);
```

Se pueden definir tipos de tablas. Si se define una columna de una tabla con un tipo de tabla, cada fila de la tabla contendrá en esa columna una tabla (tabla anidada o *nested table*). Por ejemplo, se podría añadir a la tabla `proyectos` una columna para la lista de empleados asignados al proyecto.

```
CREATE TYPE TablaAsigEmpleados AS TABLE OF REF Empleado;
/
ALTER TYPE Proyecto ADD ATTRIBUTE asig_empleados TablaAsigEmpleados CASCADE;
```

Se puede añadir un nuevo proyecto con la lista de empleados asignados de la siguiente manera:

```
UPDATE proyectos
SET asig_empleados=TablaAsigEmpleados
(
(SELECT REF(e) FROM empleados e WHERE DNI='56789012B'),
(SELECT REF(e) FROM empleados e WHERE DNI='76543210B')
)
WHERE nom_proy='PAPEL ELECTRÓNICO';
```



Actividad propuesta 5.8

Se quiere tener para cada asignación de un empleado a un proyecto la fecha de inicio y de fin. Elimina la columna `asig_empleados` de la tabla `proyectos` y añade una nueva columna con igual nombre que contenga también una tabla anidada con los empleados asignados al proyecto, pero en cada fila de esta tabla, además de la referencia al empleado, debe haber columnas con las fechas de inicio y de fin de asignación al proyecto (`f_ini` y `f_fin`). La referencia al empleado no puede ser nula, ni la fecha de inicio. Su clave primaria debe estar compuesta por la referencia

al empleado y la fecha de inicio. Se trata de definir un nuevo tipo `AsigEmpleado` que incluya la referencia al empleado y las dos fechas, y utilizarlo para definir el tipo `TablaAsigEmpleados`.

Las características objeto-relacionales de Oracle ofrecen muchas posibilidades. Con esta breve introducción se ha tratado solo de introducir las características fundamentales de las BDOR, basadas en SQL:99, y de dar una idea de cómo se puede crear con Oracle un esquema objeto-relacional con diversos tipos de objetos y diversos tipos de relaciones entre ellos.

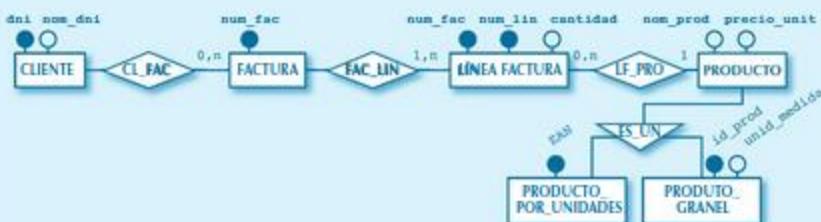
Resumen

- Como solución al desfase objeto-relacional, es decir, al conjunto de dificultades que plantea la persistencia de objetos en bases de datos relacionales, han surgido diversas soluciones. Entre ellas, las BDO, las BDOR y la correspondencia objeto-relacional.
- Las BDO permiten almacenar directamente objetos.
- La organización ODMG desarrolló estándares para BDO, entre los que están ODL y OQL. No se incluye entre ellos ningún lenguaje para manipulación de objetos, sino *language bindings* para lenguajes de programación orientados a objetos –entre ellos Java–, que hacen posible la persistencia transparente. Es decir, se gestionan de igual manera los objetos persistentes y transitorios, y en el momento de confirmar una transacción se reflejan los cambios realizados sobre los objetos en la base de datos.
- ODL permite especificar los atributos de los objetos, sus métodos y las relaciones entre objetos, que pueden ser de tipo uno a uno, uno a mucho o muchos a muchos.
- En SQL:99 se introdujeron tipos estructurados definidos por el usuario, entre los que están los tipos objetos (clases), además de tipos de colecciones, tablas anidadas y referencias.
- Las BDOR, como Oracle, se basan en SQL:99 y amplían la funcionalidad de una base de datos relacional con tipos estructurados definidos por el usuario, entre los que están los objetos.



Ejercicios propuestos

1. En el siguiente diagrama E-R se representan las relaciones entre clientes, facturas y productos incluidos en las facturas. Cada línea de factura viene identificada por la línea de factura y un número secuencial dentro de la propia factura. Interesa guardar ambos, dado que aparecen en las facturas impresas y el orden de las líneas es relevante. Los productos son de dos tipos: los que se venden por unidades, identificados por su EAN, y los que se venden a granel (por kilos, litros, etc.), identificados por un código que puede contener letras y números. Para estos últimos interesa guardar la unidad de medida a la que se aplica el precio unitario. Crea un esquema de objetos para él, la definición del esquema de objetos en ODL de Matisse, e importa el esquema en una nueva base de datos en Matisse.



- Crea un programa en Java que cree varios clientes y productos y varias facturas para varios clientes, cada una de ellas con varias líneas. Debe haber alguna factura que contenga líneas para productos tanto por unidades como a granel. Será necesario generar previamente las *stub classes*.
- Añade un método a la clase *Factura* que devuelva su importe neto, que será el resultado de sumar el importe para cada línea, que a su vez será el resultado de multiplicar el precio unitario del producto por la cantidad. Escribe un programa que lo utilice para mostrar todos los datos de todas las facturas, incluyendo el importe neto.
- Escribe un programa que borre una factura, que debe recuperar a partir de su número de factura. Antes de borrar la factura, debe borrar sus líneas.
- Cambia el método *deepRemove()* de la clase *Factura* para que borre una factura, borrando previamente todas sus líneas y, por último, la propia factura.
- Escribe un programa que, utilizando el driver JDBC de Matisse, obtenga todas las facturas de un cliente identificado por su DNI, y muestre para cada una su número de factura y su importe total, que se obtendrá con el método desarrollado en una actividad anterior.
- Crea un esquema objeto-relacional en Oracle para el diagrama E-R anterior. Las relaciones de uno a muchos se implementarán mediante referencias a objetos en lugar de mediante claves foráneas. Las líneas de factura se almacenarán en una tabla anidada dentro de la propia tabla de facturas, y en cada línea de factura habrá una referencia al producto.
- Crea para Oracle, mediante sentencias de SQL, los mismos datos que se crearon en actividades anteriores para Matisse.
Borra en Oracle, mediante sentencias de SQL, la misma o las mismas facturas que se borraron en una actividad anterior mediante un programa en Java en Matisse.

ACTIVIDADES DE AUTOEVALUACIÓN

- El objeto del estándar ODMG 3.0 es:
 - a) Definir las características que debe cumplir una BDO.
 - b) La persistencia de objetos utilizados en lenguajes orientados a objetos.
 - c) La persistencia de objetos en BDO u objeto-relacionales.
 - d) Ninguna de las opciones anteriores es correcta.

2. En SQL:99 se introdujeron:
- a) Tipos de objetos definidos por el usuario que son clases e interfaces.
 - b) Mecanismos para facilitar la transición de las actuales bases de datos relacionales a futuras BDO.
 - c) Tipos estructurados definidos por el usuario, entre ellos objetos, arrays y tablas anidadas.
 - d) Mecanismos para implementar la persistencia de objetos en bases de datos relacionales mediante el establecimiento automático de correspondencias objeto-relacionales.
3. Un *language binding* tal como los especificados en ODMG 3.0:
- a) Interactúa con un DML con unas funcionalidades mínimas.
 - b) Permite gestionar de la misma forma objetos persistentes y transitorios, utilizando los mecanismos ya existentes en un lenguaje de programación.
 - c) Hace innecesario un DML, pero requiere determinadas extensiones en la sintaxis de un lenguaje orientado a objetos de propósito general.
 - d) No garantiza la completitud computacional propuesta en el manifiesto de Atkinson y otros en 1989.
4. Un *traversal path* en ODL:
- a) Solo se define para relaciones de uno a muchos o de muchos a muchos.
 - b) Se define en extremos de una relación con multiplicidad mayor que 1.
 - c) Se definen en ambos extremos de una relación binaria.
 - d) Se implementan siempre mediante colecciones, ordenadas o desordenadas.
5. El lenguaje OQL:
- a) Tiene las sentencias UPDATE y DELETE de SQL, adaptadas para objetos.
 - b) Permite hacer consultas sobre objetos persistentes almacenados en una BDO.
 - c) Solo se puede utilizar desde un *language binding*.
 - d) Es completamente independiente de los *language binding*, porque estos solo sirven para crear, modificar y borrar objetos, pero no para consultarlos.
6. Una implementación del *language binding* de Java de ODMG 3.0 garantiza que, al realizarse una operación *commit* para confirmar una transacción:
- a) Se graben todos los cambios realizados en objetos persistentes, pero se ignoran los cambios realizados en cualquier objeto transitorio.
 - b) Se graben los cambios realizados sobre cualquier objeto persistente modificado, y también sobre cualquier objeto transitorio alcanzable siguiendo las referencias desde ellos.
 - c) Si siguiendo las referencias desde cualquier objeto persistente modificado se alcanza un objeto no persistente, se produzca un error y se deshagan todos los cambios realizados.
 - d) Se graben todos los cambios realizados durante ella, tanto en objetos persistentes como transitorios, y todos estos últimos se convierten en persistentes.
7. La base de datos Matisse:
- a) Es una base de datos posrelacional, y solo hasta cierto punto una BDO.

- b) Es en última instancia una base de datos relacional sobre la que se implementan los objetos en una capa de *software*.
- c) Es una BDO que tiene su propio *driver JDBC*, pero en una transacción no se pueden mezclar cambios hechos con JDBC y con los métodos de las *stub classes* del Java *binding*.
- d) Es una BDO que tiene además un *driver JDBC*.
8. El lenguaje SQL de Matisse:
- a) Proporciona una implementación completa de SQL:99.
- b) Solo proporciona operaciones de consulta, dado que las operaciones de creación, modificación y borrado se realizan con el Java *binding*.
- c) No es realmente SQL, aunque sí es similar a SQL en muchos aspectos.
- d) No es SQL, pero incluye una implementación de OQL y, al igual que OQL, es similar a SQL.
9. Oracle 12c:
- a) Es una BDOR, es decir, es a la vez una BDO y una base de datos relacional.
- b) Es una base de datos relacional sobre la que se implementan objetos en una capa de *software*.
- c) Es conforme a ODMG 3.0.
- d) Todo lo anterior es cierto.
10. La base de datos Oracle:
- a) Permite tablas de objetos y columnas de objetos.
- b) Permite tablas de objetos, pero no columnas de objetos, aunque sí columnas que contengan referencias a objetos.
- c) Permite tablas de objetos siempre que no contengan referencias a otros objetos.
- d) Permite tablas anidadas, siempre que no sean tablas de objetos.

SOLUCIONES:

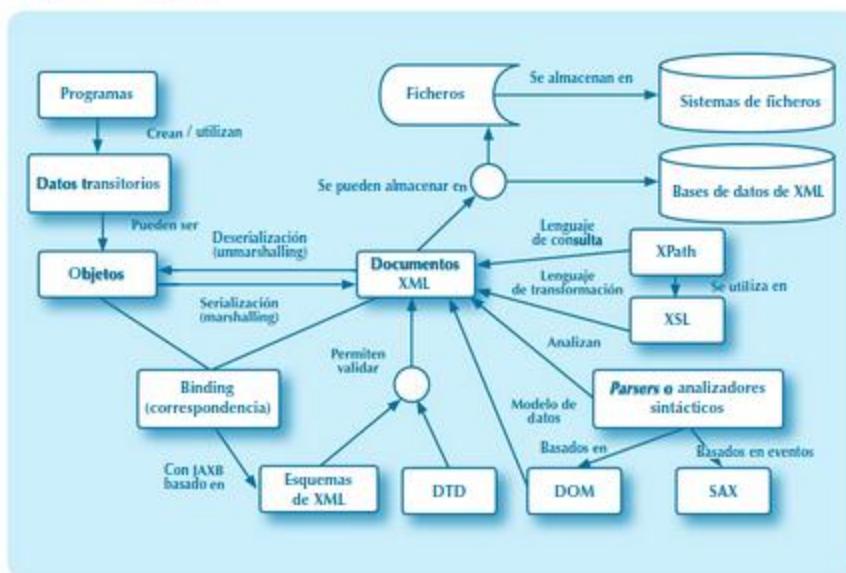
1. a b c d2. a b c d3. a b c d4. a b c d5. a b c d6. a b c d7. a b c d8. a b c d9. a b c d10. a b c d

XML

Objetivos

- ✓ Entender la sintaxis del lenguaje XML.
- ✓ Conocer el modelo DOM para almacenamiento en memoria de documentos de XML.
- ✓ Utilizar parsers o analizadores sintácticos para XML basados en DOM y SAX.
- ✓ Analizar los mecanismos de verificación y validación de XML tales como DTD y esquemas.
- ✓ Trabajar de forma efectiva con la funcionalidad de *binding* proporcionada por JAXB para serialización (*marshalling*) y deserialización (*unmarshalling*) de objetos con documentos de XML.
- ✓ Aprender la sintaxis de XPath y utilizarlo para realizar consultas en documentos de XML.
- ✓ Comprender XSL y desarrollar plantillas sencillas para obtener documentos de diverso tipo a partir de un documento de XML, al menos documentos en formato HTML y CSV.

Mapa conceptual



Glosario

DOM. Del inglés *document object model* o “modelo de objetos para documentos”. Modelo de datos para los contenidos de documentos de XML.

Esquema de XML. Documento que especifica la estructura y contenidos para un formato particular de documento de XML.

Parser o analizador sintáctico. Programa o componente de software que analiza un documento de texto y verifica que es sintácticamente correcto de acuerdo a la sintaxis de un lenguaje formal, como por ejemplo XML, y que permite acceder a sus contenidos, una vez analizado.

SAX. Del inglés *simple API for XML* o “analizador sencillo para XML”. Analizador o parser de XML basado en eventos, que no genera una estructura de datos, sino que va lanzando determinados eventos conforme va identificando componentes de un documento de XML.

Serialización y deserialización de documentos XML (unmarshalling y marshalling). Generación de una estructura de objetos a partir de un documento de XML y viceversa, basada en la correspondencia entre los elementos de un esquema de XML y una colección de clases.

Validación. Proceso que permite verificar si un documento de XML sintácticamente correcto es, además, válido, es decir, conforme a una especificación para su estructura y contenidos.

XML. Lenguaje de marcas que permite representar información en documentos de texto con una estructura jerárquica.

XPath. Lenguaje de consulta sobre los contenidos de documentos XML.

XSL. Lenguaje para transformación de documentos XML a otros formatos.

6.1. El lenguaje XML

El nombre XML viene de *extensible markup language* o “lenguaje de marcado extensible”. XML es un estándar del W3C (World Wide Web Consortium), que es la organización que tiene como objetivo fundamental desarrollar y mantener el amplio cuerpo de estándares que hacen posible la web. XML ha jugado un papel fundamental en el desarrollo de nuevos servicios y aplicaciones para internet desde su introducción en 1998.

El origen del lenguaje XML está ligado a la *world wide web*, WWW o simplemente “la web”. HTML (*hypertext markup language* o “lenguaje de marcado para hipertexto”) es el lenguaje de marcado (*markup language*, en inglés) que se inventó para especificar tanto los contenidos como el aspecto visual o presentación de las páginas webs. El significado original de *markup* en inglés hace referencia a anotaciones que se hacen sobre un texto para indicar qué formato debe tener una vez publicado. Un documento de HTML sería como un texto con anotaciones (*markups*) que indican el formato o la presentación, es decir, aspectos tales como el tamaño de letra, negritas, cursivas, etc. XML surgió como un lenguaje con sintaxis similar a HTML, pero para representar exclusivamente datos de cualquier tipo. Esto fue en paralelo con la tendencia creciente a la separación entre la presentación, formato o aspecto visual de las páginas webs, por una parte, y los contenidos o datos que mostraban, por otra. Finalmente, HTML quedó como un subconjunto de XML para presentación, denominado con frecuencia XHTML, y XML, como lenguaje universal para representar cualquier tipo de datos.

Entre los principios de diseño de XML está el que sea fácil de leer e interpretar tanto por humanos como por máquinas, y esta es una de las claves de su éxito. Esto se ha conseguido a cambio de una gran *verbalidad*. Los documentos de XML ocupan mucho espacio en relación con la cantidad efectiva de información que contienen.

6.2. Estructura de un documento de XML

Un documento de XML es un documento de texto que está escrito conforme a la sintaxis del lenguaje XML. De la misma forma, por ejemplo, un fichero de programa en Java es un fichero cuyos contenidos son conformes a la sintaxis del lenguaje Java. De un fichero así se dice que está bien formado, lo que significa simplemente que es sintácticamente correcto. El concepto de “XML bien formado” no deja de ser redundante. Si no estuviera bien formado, no sería XML.

Validación. Proceso que permite verificar si un documento de XML sintácticamente correcto es, además, válido, es decir, conforme a una especificación para su estructura y contenidos.

XML. Lenguaje de marcas que permite representar información en documentos de texto con una estructura jerárquica.

XPath. Lenguaje de consulta sobre los contenidos de documentos XML.

XSL. Lenguaje para transformación de documentos XML a otros formatos.

6.1. El lenguaje XML

El nombre XML viene de *extensible markup language* o “lenguaje de marcado extensible”. XML es un estándar del W3C (World Wide Web Consortium), que es la organización que tiene como objetivo fundamental desarrollar y mantener el amplio cuerpo de estándares que hacen posible la web. XML ha jugado un papel fundamental en el desarrollo de nuevos servicios y aplicaciones para internet desde su introducción en 1998.

El origen del lenguaje XML está ligado a la *world wide web*, WWW o simplemente “la web”. HTML (*hypertext markup language* o “lenguaje de marcado para hipertexto”) es el lenguaje de marcado (*markup language*, en inglés) que se inventó para especificar tanto los contenidos como el aspecto visual o presentación de las páginas webs. El significado original de *markup* en inglés hace referencia a anotaciones que se hacen sobre un texto para indicar qué formato debe tener una vez publicado. Un documento de HTML sería como un texto con anotaciones (*markups*) que indican el formato o la presentación, es decir, aspectos tales como el tamaño de letra, negritas, cursivas, etc. XML surgió como un lenguaje con sintaxis similar a HTML, pero para representar exclusivamente datos de cualquier tipo. Esto fue en paralelo con la tendencia creciente a la separación entre la presentación, formato o aspecto visual de las páginas webs, por una parte, y los contenidos o datos que mostraban, por otra. Finalmente, HTML quedó como un subconjunto de XML para presentación, denominado con frecuencia XHTML, y XML, como lenguaje universal para representar cualquier tipo de datos.

Entre los principios de diseño de XML está el que sea fácil de leer e interpretar tanto por humanos como por máquinas, y esta es una de las claves de su éxito. Esto se ha conseguido a cambio de una gran *verbalidad*. Los documentos de XML ocupan mucho espacio en relación con la cantidad efectiva de información que contienen.

6.2. Estructura de un documento de XML

Un documento de XML es un documento de texto que está escrito conforme a la sintaxis del lenguaje XML. De la misma forma, por ejemplo, un fichero de programa en Java es un fichero cuyos contenidos son conformes a la sintaxis del lenguaje Java. De un fichero así se dice que está bien formado, lo que significa simplemente que es sintácticamente correcto. El concepto de “XML bien formado” no deja de ser redundante. Si no estuviera bien formado, no sería XML.

Como ejemplo de documento de XML, valga el siguiente, con datos de diferentes clientes:

```
<?xml version="1.0" encoding="UTF-8"?>
<clientes>
    <cliente DNI="78901234X">
        <apellidos>NADELES</apellidos>
        <CP>44126</CP>
    </cliente>
    <cliente DNI="89012345E">
        <apellidos>ROJAS</apellidos>
        <validez estado="borrado" timestamp="1528286082"/>
    </cliente>
    <cliente DNI="56789012B">
        <apellidos>SAMPER</apellidos>
        <CP>29730</CP>
    </cliente>
</clientes>
```

A parte de la línea de cabecera, el documento está formado por bloques dentro de los cuales hay otros bloques, con lo que tiene una estructura jerárquica. A continuación, se describen sus distintos componentes, empleando la terminología habitual para XML:

1. *Cabecera.* En ella se identifica el documento como un documento de XML y se proporcionan algunos detalles como la versión de XML (1.0) y la codificación empleada para el texto (UTF-8). UTF-8 es la codificación por defecto, en caso de que no se indique.
2. *Elementos.* Aparte de la cabecera, todo el documento está comprendido entre una etiqueta de apertura `<clientes>` y una etiqueta de cierre `</clientes>`. Un bloque de este tipo es un elemento. Hay que tener en mente que, aunque la palabra *elemento* tiene un significado muy amplio y genérico, cuando se habla de documentos de XML es un término con un significado muy específico. Por lo tanto, en lo sucesivo se debe entender con este significado y se evitará utilizar esta palabra si no es con ese significado específico. Un elemento puede tener una única etiqueta, como es el caso de la etiqueta con nombre `validez`. Se sabe que no hay etiqueta de cierre porque al final de la etiqueta de apertura se encuentra `/>` y no `>`. Dentro de un elemento puede haber:

- Un texto. En el caso de la primera etiqueta con nombre `apellidos`, es `NADELES`.
- Otros elementos, y por eso los documentos de XML tienen estructura jerárquica.

En ese ejemplo no hay ningún elemento que contenga a la vez un texto y otros elementos, pero puede haberlos, como en el siguiente ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<comunidades>
    <comunidad>EXTREMADURA
        <provincia CAPITAL="CÁCERES">CÁCERES</provincia>
        <provincia CAPITAL="BADAJOZ">BADAJOZ</provincia>
    </comunidad>
    <comunidad>ASTURIAS
        <provincia CAPITAL="OVIEDO">ASTURIAS</provincia>
    </comunidad>
</comunidades>
```

3. *Atributos.* Aparecen en las etiquetas de apertura de los elementos, y consisten en la asignación de un valor a un nombre. En el elemento <cliente DNI="78901234X">, el atributo con nombre **DNI** tiene valor **78901234X**. El elemento **validez** tiene dos atributos con nombres **estado** y **timestamp**.

6.3. DOM

DOM es un modelo que define interfaces de programación para acceder a los contenidos de un documento de XML y modificarlos. En el modelo DOM, un documento de XML se representa como un árbol, y los distintos componentes de un documento de XML son nodos de dicho árbol. La figura 1.7 es el documento DOM para el documento de XML del ejemplo anterior.

En el árbol DOM hay distintos tipos de nodos. Los hay para los elementos (**clientes**, **cliente**, **apellidos**, **CP**, **validez**). También los hay para los textos, identificados como **#text**, que contienen el valor para los elementos, incluyendo los elementos para los que no hay ningún valor, a saber, **clientes**, **cliente** y **validez**. Los atributos, en el modelo DOM, no son nodos del árbol. Están en una lista asociada al nodo de su elemento. Cada entrada de esta lista contiene el nombre de un atributo y su valor.

Para concluir, existen nodos de tipo **#text** que parecen no contener ningún texto. En realidad, contienen saltos de línea y espacios en blanco, incluidos normalmente en los ficheros de XML para facilitar a las personas su lectura. Nada impediría que el documento de XML estuviera en una única línea de texto, aunque serían más incómodos de leer para las personas. Más adelante veremos la manera de ignorar estos nodos de texto innecesarios e *ignorables*.

En breve se verá cómo se puede obtener un documento DOM a partir de un documento de XML con un programa en Java.

6.4. Parsers o analizadores sintácticos, serialización y deserialización

Un *parser* o analizador sintáctico, para un lenguaje formal como por ejemplo XML, Java o C++, es un programa que permite verificar que un documento de texto es sintácticamente correcto, es decir, conforme a la sintaxis del lenguaje. En ese caso, además, una vez analizado el documento, permite acceder a sus contenidos. Existen *parsers* para XML como para cualquier otro lenguaje formal. Un *parser* DOM permite verificar que un documento de XML es sintácticamente correcto y si lo es, obtener un documento DOM.

El proceso de *parsing*, incluyendo la construcción de una estructura de datos que representa sus contenidos, se denomina a veces deserialización, porque a partir de un documento de texto, que es una serie o secuencia de caracteres, se obtiene una estructura de datos no secuencial. El proceso inverso, es decir, obtener una estructura secuencial, como por ejemplo un fichero, a partir de una no secuencial, se suele denominar serialización.



Figura 6.1
Parser DOM

Un *parser* DOM es muy sencillo de utilizar en Java. Con pocas líneas de código se obtiene un documento DOM a partir de un fichero XML. Disponer de todo el documento en memoria permite un acceso rapidísimo a sus contenidos. Las interfaces de programación de DOM permiten no solo consultar los contenidos de un documento, sino también modificarlos. También se puede serializar un árbol DOM, es decir, generar un fichero con sus contenidos.

Pero construir un documento DOM en memoria puede suponer un grave inconveniente para documentos muy grandes. XML se utiliza hoy en día para almacenar enormes volúmenes de datos. Por ejemplo, remesas con decenas de miles de recibos bancarios, o corpus lingüísticos con millones de entradas. Para aplicaciones que necesitan consultar los contenidos de documentos muy grandes, pero no modificarlos, conviene utilizar otro tipo de *parser* que permita acceder a ellos sin almacenar el documento en memoria, como por ejemplo un *parser* SAX.

Java incluye soporte para XML y varias tecnologías relacionadas en su API JAXP (Java API for XML Processing). La mayoría de ellas (DOM, SAX y XSLT) se verán en el resto de este capítulo.



Recurso web

Tutorial de Oracle sobre JAXP: <http://docs.oracle.com/javase/tutorial/jaxp/>.

6.5. DOM con Java

Todo lo necesario para trabajar con los distintos componentes del modelo DOM está en el paquete `org.w3c.dom`. Se puede ver que es de la organización W3C (su nombre empieza por `org.w3c`, no por `java` o `javax`).

Un documento DOM puede crearse a partir de un documento de XML, mediante un proceso de *parsing*. O también puede crearse directamente en memoria, empezando con un documento vacío y añadiendo elementos.



Recurso web

Se pueden consultar los JavaDocs del paquete `org.w3c.dom` en <http://docs.oracle.com/javase/8/docs/api/>.

Las principales interfaces para los distintos elementos del modelo DOM son las siguientes:

- `Document`: representa un documento de XML.
- `Node`: representa un nodo de un documento de XML. La interfaz `Document` implementa la interfaz `Node`. Eso representa el hecho de que un documento DOM está definido por su nodo raíz y algunas cosas más.
- `Element`: representa un elemento.
- `NodeList`: representa una lista de nodos. Es útil para recuperar la lista de nodos de un nodo, como por ejemplo la lista de elementos de un `Element`.

- **NamedNodeMap:** al igual que `NodeList`, representa una lista de nodos, pero con algunas diferencias. Se utiliza para listas de atributos (interfaz `Attr`). En ellas, los atributos se identifican por el nombre. No puede haber dos con el mismo nombre, y el orden en que aparecen es irrelevante.

En lo sucesivo se describirán brevemente estas interfaces y sus principales métodos, para hacerse una idea general de su funcionalidad y comprender los programas de ejemplo. Para realizar algunas de las actividades propuestas será necesario consultar su documentación.

En el cuadro 6.1 se muestran algunos métodos de la interfaz `Document` para consulta.

CUADRO 6.1
Métodos de la interfaz `Document` para consulta

Método	Funcionalidad
<code>String getXmlEncoding()</code>	Devuelve la codificación. Podría ser, por ejemplo, "UTF-8".
<code>String getXmlVersion()</code>	Devuelve la versión de XML. Podría ser, por ejemplo, "1.0".

En el cuadro 6.2 se muestran algunos métodos de la interfaz `Node` que son útiles para consulta de un documento DOM.

CUADRO 6.2
Métodos de la interfaz `Node` para consulta

Método	Funcionalidad
<code>short getNodeType()</code>	Devuelve el tipo de nodo. Algunos de los posibles valores que puede devolver son: <code>DOCUMENT_NODE</code> : se trata del nodo raíz del documento. <code>ELEMENT_NODE</code> : se trata de un elemento. <code>TEXT_NODE</code> : se trata de un texto.
<code>String getNodeName()</code>	Devuelve el nombre del nodo. Si es un elemento, devuelve el nombre del elemento. Si es un texto, devuelve "#text". Si es un atributo, devuelve el nombre del atributo.
<code>String getNodeValue()</code>	Devuelve el valor asociado al nodo. Si es un texto, devuelve el texto. Si es un atributo, devuelve el valor del atributo.
<code>NodeList getChildNodes()</code>	Devuelve la lista de nodos hijos. Relevante para nodos de tipo <code>Element</code> .
<code>NamedNodeMap getAttributes()</code>	Si el nodo es de tipo <code>Element</code> , devuelve una lista con sus atributos. Si no, devuelve null. Devuelve un <code>NamedNodeMap</code> y no un <code>NodeList</code> , porque entre los atributos de un <code>Element</code> el orden no es relevante, y se identifican solo por su nombre.

[.../...]

CUADRO 6.2 (CONT.)

<code>Node getParentNode()</code>	Devuelve el nodo padre si existe, o <code>null</code> en caso contrario. Los nodos de algunos tipos de nodos, como <code>Document</code> o <code>Attribute</code> , no tienen nodo padre.
-----------------------------------	---

6.5.1. Parsing DOM

El paquete `javax.xml.parsers` proporciona toda la funcionalidad para el *parser* DOM y también para el *parser* SAX, que se verá más adelante.

La clase `DocumentBuilder` implementa un *parser* DOM. Se obtiene una instancia suya con la clase `DocumentBuilderFactory`, mediante su método de clase `newInstance()`. A su vez, se obtiene una instancia de `DocumentBuilderFactory` mediante su método de clase `newDocumentBuilder()`. En los cuadros siguientes se indican los métodos más importantes de ambas clases. Algunos métodos de `DocumentBuilderFactory` especifican opciones por defecto para los *parsers* que construye.

CUADRO 6.3

Algunos de los principales métodos de la clase `DocumentBuilderFactory`

Método	Funcionalidad
<code>static DocumentBuilderFactory newInstance ()</code>	Crea una instancia de la clase.
<code>DocumentBuilder newDocumentBuilder ()</code>	Devuelve un <i>parser</i> DOM.
<code>void setIgnoringComments (boolean ignoreComm)</code>	Ignora comentarios. A no ser que se ignoren, se incluyen como nodos dentro del árbol DOM.
<code>void setIgnoringElementContentWhitespace (boolean whitespace)</code>	Ignora textos vacíos en el contenido de los elementos si el <i>parser</i> está en modo de validación, que se verá más adelante. En caso contrario, no tiene ningún efecto.

En el cuadro 6.4 se explican los métodos más importantes de `DocumentBuilder`. Esta clase no solo sirve para obtener documentos DOM mediante el *parsing* de un documento DOM, sino también para crearlos desde cero. Existe un método para crear un documento DOM vacío y métodos para añadir distintos tipos de elementos, como se verá más adelante.

CUADRO 6.4

Métodos de la clase `DocumentBuilder`

Método	Funcionalidad
<code>abstract Document newDocument()</code>	Devuelve un nuevo documento DOM vacío inicialmente.
	[.../...]

CUADRO 6.4 (CONT.)

```
Document parse(File f)
Document parse(InputStream is)
Document parse(String uri)
```

Construye un documento DOM a partir de los contenidos de un fichero (`File`), a partir de un `InputStream`, o a partir de un URI. El URI puede ser, por ejemplo, una dirección desde la que obtener el documento mediante HTTP, y en ese caso empezaría por `http://` y tendría la forma habitual para ese tipo de URI.

El siguiente programa obtiene un documento DOM a partir de un fichero XML y muestra sus contenidos. El *parsing* en sí es sencillo, cinco líneas de código para crear y configurar un `DocumentBuilderFactory`, crear un `DocumentBuilder` e invocar su método `parse()`. El grueso del programa es la parte que muestra los contenidos del documento, a saber, el método `muestraNodo`. Para mostrar el árbol entero se le pasa el nodo raíz. El parámetro `nivel` (0 para el nodo raíz) se utiliza para mostrar los contenidos del nodo con la indentación apropiada. Una vez se han mostrado los detalles del nodo, se obtiene la lista de nodos hijos y se muestran sus contenidos llamando de nuevo a `muestraNodo` para cada uno de ellos, incrementando `nivel` en uno. Se hace una verificación previa para ignorar los nodos de texto que solo tienen espacios en blanco o saltos de línea. `muestraNodo` escribe a un `PrintStream` que se le pasa como parámetro. Esto permite, de manera muy sencilla, dirigir la salida a un fichero cualquiera, por ejemplo, en lugar de a `System.out`.

```
// Parsing DOM y visualización de documento DOM generado
package dom_parser;
import java.io.File;
import java.io.PrintStream;
import java.io.FileNotFoundException;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;
import org.xml.sax.SAXException;
public class DOM_Parser {
    private static final String INDENT_NIVEL = " "; // Para indentación
    public static void muestraNodo(Node nodo, int nivel, PrintStream ps) {
        if (nodo.getNodeType() == Node.TEXT_NODE) { // Ignora textos vacíos
            String text = nodo.getNodeValue();
            if (text.trim().length() == 0) {
                return;
            }
        }
        for (int i = 0; i < nivel; i++) { // Indentación
            ps.print(INDENT_NIVEL);
        }
        switch (nodo.getNodeType()) { // Escribe información de nodo según
            tipo
        case Node.DOCUMENT_NODE: // Documento
            Document doc=(Document) nodo;
            ps.println("Documento DOM, versión: "+doc.getXmlVersion()+
                      ", codificación: "+doc.getXmlEncoding());
            break;
```

```

case Node.ELEMENT_NODE: // Elemento
    ps.print("<" + nodo.getNodeName());
    NamedNodeMap listaAtr = nodo.getAttributes(); // Lista atributos
    for (int i = 0; i < listaAtr.getLength(); i++) {
        Node atr = listaAtr.item(i);
        ps.print(" @" + atr.getNodeName() + "[" + atr.getNodeValue()
            + "]");
    }
    ps.println(">");
    break;
case Node.TEXT_NODE: // Texto
    ps.println(nodo.getNodeName() + "[" + nodo.getNodeValue() +
        "]");
    break;
default: // Otro tipo de nodo
    ps.println("(nodo de tipo: " + nodo.getNodeType() + ")");
}
NodeList nodosHijos = nodo.getChildNodes(); // Muestra nodos hijos
for (int i = 0; i < nodosHijos.getLength(); i++) {
    muestraNodo(nodosHijos.item(i), nivel + 1, ps);
}
}

public static void main(String[] args) {
    String nomFich;
    if (args.length < 1) {
        System.out.println("Indicar por favor nombre de fichero");
        return;
    } else {
        nomFich = args[0];
    }
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    dbf.setIgnoringComments(true);
    dbf.setIgnoringElementContentWhitespace(true);
    try {
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document domDoc = db.parse(new File(nomFich));
        muestraNodo(domDoc, 0, System.out);
    } catch (FileNotFoundException | ParserConfigurationException |
        SAXException e) {
        System.err.println(e.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

A continuación, se muestra la salida del programa anterior al lado del fichero XML. Se puede ver que no se muestran nodos de texto que solo tengan espacios en blanco.

```

<?xml version="1.0" encoding="UTF-8"?>
<clientes>
    <cliente DNI="78901234X">
        <apellidos>NADALES</apellidos>
        <CP>44126</CP>
    </cliente>

```

<pre> <?xml version="1.0" encoding="UTF-8"?> <clientes> <cliente @DNI[78901234x]> <apellidos> #text[NADALES] </pre>	<p>Documento DOM, versión: 1.0, codificación: UTF-8</p>
---	---

```

<cliente DNI="89012345E">
    <apellidos>ROJAS</apellidos>
    <valides estado="borrado">
        timestamp="1528286082">
    </valides>
</cliente>
<cliente DNI="56789012B">
    <apellidos>SAMPER</apellidos>
    <CP>29730</CP>
</cliente>
</clientes>

```

```

<CP>
    #text[44126]
<cliente @DNI[89012345E]>
<apellidos>
    #text[ROJAS]
<valides @estado[borrado] @
    timestamp[1528286082]>
<cliente @DNI[56789012B]>
<apellidos>
    #text[SAMPER]
<CP>
    #text[29730]

```



Actividad propuesta 6.1

Modifica el programa anterior para que escriba la salida a un fichero con nombre `parsing_dom.txt`. Como mejora, puedes utilizar la clase `SimpleDateFormat` para hacer que en el nombre de fichero se incluyan la fecha y la hora, de manera similar a como se hacía en un programa de ejemplo del capítulo anterior dedicado a ficheros.

6.5.2. Creación de documentos con DOM

Ahora se mostrará cómo se puede crear desde cero un documento DOM, y cómo se puede escribir un documento DOM a un fichero de XML.

Se pueden crear nodos de distintos tipos con algunos métodos de la interfaz `Document`. Se usarán estos para crear los nodos, y métodos de la interfaz `Node` para añadirlos al documento.

CUADRO 6.5

Métodos de la interfaz `Document` para crear nuevos nodos para un documento DOM

Método	Funcionalidad
<code>Element createElement(String nombreEtiqueta)</code>	Crea nuevo elemento con el nombre dado.
<code>Text createTextNode(String texto)</code>	Crea nuevo nodo de tipo texto.

Los atributos se añaden con métodos de la interfaz `Element`.

CUADRO 6.6

Métodos de la interfaz `Element`

Método	Funcionalidad
<code>void setAttribute(String name, String value)</code>	Añade atributo con el valor dado.

A continuación, algunos métodos de la interfaz `Node` que permiten añadir elementos a un documento DOM. Existen más métodos para eliminar o reemplazar elementos, y para cambiar

valores y propiedades, que no se explican aquí. Si alguna de estas operaciones no se puede realizar, típicamente lanzará una excepción de tipo `DOMException`.

CUADRO 6.7

Métodos de la interfaz Node para añadir elementos a un documento DOM

Método	Funcionalidad
<code>Node appendChild(Node nuevoHijo)</code>	Añade nodo como último nodo hijo, y devuelve el nodo añadido.
<code>Node insertBefore(Node nuevoHijo, Node refHijo)</code>	Añade nuevo hijo antes del indicado.

Antes de poner un programa de ejemplo para construir un documento DOM, se verá cómo serializar un documento DOM para poder verificarlo fácilmente una vez creado. Serializar un documento DOM es generar un documento de XML textual a partir de él.

6.5.3. Serialización de documentos DOM

La clase `Transformer` permite serializar documentos DOM, es decir, generar documentos de texto XML a partir de ellos.

Se obtienen instancias de `Transformer` mediante el método `newTransformer()` de la clase `TransformerFactory`. A su vez, se obtienen instancias de `TransformerFactory` mediante su método de clase `newInstance()`.

Por lo demás, se pueden controlar las opciones para la serialización mediante el método `setOutputProperty`, y se realiza la serialización mediante el método `transform(Source xmlSource, Result outputTarget)`.

Las opciones para serialización son muy numerosas y se pueden consultar en la documentación de la clase. Son básicamente las recogidas en el estándar XSL. Más adelante en este capítulo se verá algo de XSL.

El siguiente programa crea un documento DOM con parte de los contenidos del documento de XML de ejemplo, y después lo serializa en un `String` mediante un `StringWriter`. Se indica codificación UTF-8 para la serialización.

```
// Creación de árbol DOM añadiendo elementos y serialización una vez creado
package creardocumentodom;

import java.io.StringWriter;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.OutputKeys;
```

valores y propiedades, que no se explican aquí. Si alguna de estas operaciones no se puede realizar, típicamente lanzará una excepción de tipo `DOMException`.

CUADRO 6.7

Métodos de la interfaz `Node` para añadir elementos a un documento DOM

Método	Funcionalidad
<code>Node appendChild(Node nuevoHijo)</code>	Añade nodo como último nodo hijo, y devuelve el nodo añadido.
<code>Node insertBefore(Node nuevoHijo, Node refHijo)</code>	Añade nuevo hijo antes del indicado.

Antes de poner un programa de ejemplo para construir un documento DOM, se verá cómo serializar un documento DOM para poder verificarlo fácilmente una vez creado. Serializar un documento DOM es generar un documento de XML textual a partir de él.

6.5.3. Serialización de documentos DOM

La clase `Transformer` permite serializar documentos DOM, es decir, generar documentos de texto XML a partir de ellos.

Se obtienen instancias de `Transformer` mediante el método `newTransformer()` de la clase `TransformerFactory`. A su vez, se obtienen instancias de `TransformerFactory` mediante su método de clase `newInstance()`.

Por lo demás, se pueden controlar las opciones para la serialización mediante el método `setOutputProperty`, y se realiza la serialización mediante el método `transform(Source xmlSource, Result outputTarget)`.

Las opciones para serialización son muy numerosas y se pueden consultar en la documentación de la clase. Son básicamente las recogidas en el estándar XSL. Más adelante en este capítulo se verá algo de XSL.

El siguiente programa crea un documento DOM con parte de los contenidos del documento de XML de ejemplo, y después lo serializa en un `String` mediante un `StringWriter`. Se indica codificación UTF-8 para la serialización.

```
// Creación de árbol DOM añadiendo elementos y serialización una vez creado
package creardocumentodom;

import java.io.StringWriter;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.OutputKeys;
```

```

import javax.xml.transform.stream.StreamResult;
public class CrearDocumentoDOM {
    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.newDocument();
            doc.setXmlVersion("1.0");

            Element elClientes = doc.createElement("clientes");
            doc.appendChild(elClientes);

            Element elCliente = doc.createElement("cliente");
            elCliente.setAttribute("DNI", "89012345E");
            Element elApell = doc.createElement("apellidos");
            elApell.appendChild(doc.createTextNode("ROJAS"));
            elCliente.appendChild(elApell);
            Element elValidez = doc.createElement("validez");
            elValidez.setAttribute("estado", "borrado");
            elValidez.setAttribute("timestamp", "1528286082");
            elCliente.appendChild(elValidez);
            elClientes.appendChild(elCliente);

            DOMSource domSource = new DOMSource(doc);
            Transformer transformer = TransformerFactory.newInstance().newTransformer();
            transformer.setOutputProperty(OutputKeys.METHOD, "xml");
            transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
            transformer.setOutputProperty(OutputKeys.INDENT, "yes");
            transformer.setOutputProperty(
                "{http://xml.apache.org/xslt}indent-amount", "4");

            StringWriter sw = new StringWriter();
            StreamResult sr = new StreamResult(sw);
            transformer.transform(domSource, sr);
            System.out.println(sw.toString());
        } catch (ParserConfigurationException e) {
            System.err.println(e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

La salida del programa anterior es la siguiente:

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<clientes>
    <cliente DNI="89012345E">
        <apellidos>ROJAS</apellidos>
        <validez estado="borrado" timestamp="1528286082"/>
    </cliente>
</clientes>

```

Actividad propuesta 6.2



Crea un programa que realice el *parsing DOM* del fichero de ejemplo con los datos de varios clientes. Una vez hecho esto, debe buscar un cliente cualquiera por DNI y cambiarle sus apellidos. Después, debe hacer una copia del fichero añadiendo al nombre ".bak". Seguidamente, debe utilizar la clase `Transformer` para serializar el documento DOM al fichero, sobreescribiendo sus contenidos. Tanto el DNI como el nuevo valor para `apellidos` se pueden asignar en variables del programa.

6.6. SAX

Ya se ha comentado que un *parser DOM* crea un documento DOM en memoria, y esto puede ser un inconveniente importante cuando se trabaja con documentos muy grandes.

En ese caso, puede ser más conveniente utilizar un *parser SAX*, que no construye una estructura de datos en memoria. En lugar de ello, cada vez que sucede un determinado tipo de evento durante el proceso de *parsing*, llama a determinados métodos de una clase especial manejadora de eventos. Estos se definen a conveniencia para la aplicación que se quiera desarrollar. También llama a métodos manejadores de errores y avisos cuando se da alguna situación que así lo requiera.

Se puede considerar que un *parser SAX* es un *parser* mínimo. Se limita a identificar eventos y, para cada evento, invocar un método manejador para el tipo de evento. Para utilizar un *parser SAX* hay que proporcionar una o varias clases que implementen distintas interfaces para manejar distintos tipos de eventos. Lo más sencillo es gestionarlos todos mediante un `DefaultHandler`, porque esta clase implementa todas estas interfaces, y esto es lo que se hará aquí. En `DefaultHandler` los métodos manejadores de eventos no hacen nada. Hay que redefinirlos a conveniencia en una subclase suya.

La clase `XMLReader` implementa un *parser SAX*. Se pueden construir instancias suyas mediante la clase `XMLReaderFactory`. Una vez creado un `XMLReader` se le asocian instancias de clases manejadoras de eventos y, por último, se invoca su método `parse()`. A continuación, se explican las clases y métodos necesarios para el siguiente ejemplo.



Figura 6.2
Parser SAX

CUADRO 6.8
Métodos de la clase XMLReader

Método	Funcionalidad
<code>void parse(InputSource input)</code> <code>void parse(String idSistema)</code>	Realiza el <i>parsing</i> de un documento de XML. Estos métodos permiten obtener fácilmente el documento a partir de un identificador de sistema, un <code>InputStream</code> o un <code>Reader</code> . Un identificador de sistema puede ser un nombre de fichero o un URI. En este último caso, se podría obtener, por ejemplo, de un servidor web mediante <code>http</code> .
<code>void setContentHandler(ContentHandler handler)</code>	Asocia al parser un manejador de eventos para contenidos. Hay varias clases que implementan <code>ContentHandler</code> , pero lo más sencillo es utilizar <code>DefaultHandler</code> para gestionar todos los eventos.

Los métodos de `DefaultHandler` suelen ir por parejas: uno para el inicio de un componente y otro para el final. Ya se ha dicho que en `DefaultHandler` no hacen nada.

CUADRO 6.9
Métodos de la clase DefaultHandler

Método	Funcionalidad
<code>void startDocument()</code> <code>void endDocument()</code>	Inicio y fin del documento (<code>Document</code>).
<code>void startElement(String uri, String nombreLocal, String nombreCualif, Attributes atributos)</code> <code>void endElement(String uri, String nombreLocal, String nombreCualif)</code>	Inicio y fin de un elemento (<code>Element</code>). Si no se usan namespaces, el nombre del elemento está en <code>localName</code> .
<code>void characters(char[] cars, int inicio, int longitud)</code>	Texto dentro de un elemento (<code>Element</code>). Se proporciona el texto del documento de XML, y la posición inicial y la posición final del texto. Se puede obtener el texto para el elemento con <code>String(cars, inicio, fin)</code> .

El siguiente programa de ejemplo utiliza un *parser SAX* para mostrar la misma información que mostraba el anterior programa de ejemplo que utilizaba un *parser DOM*. Los métodos que se han desarrollado son muy sencillos. Solo es necesario dar respuesta adecuada a cada evento redefiniendo los métodos apropiados en `GestorEventos`, clase hija de `DefaultHandler`. No hay nada en este programa realmente complicado, es incluso más sencillo que el programa anterior basado en DOM. Pero puede costar más entender cómo encajan todas las piezas en el funcionamiento global del *parser SAX*. Se utiliza la anotación `@override` para indicar que se redefine un método de la clase padre. Esto no es estrictamente necesario, pero sí aconsejable.

```

// Parsing con SAX
package sax_parser;
import java.io.PrintStream;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
class GestorEventos extends DefaultHandler {
    private static final String INDENT_NIVEL = "  "; // Para indentación
    private final PrintStream ps;
    private int nivel;
    private void indenta() {
        for (int i = 0; i < nivel; i++) { // Indentación
            ps.print(INDENT_NIVEL);
        }
    }
    public GestorEventos(PrintStream ps) {
        this.ps = ps;
    }
    @Override
    public void startDocument() {
        nivel = 0;
    }
    @Override
    public void startElement(String uri, String nombreLocal, String
        nombreCualif, Attributes atributos) {
        nivel++;
        indenta();
        ps.print("<" + nombreCualif);
        for(int i=0; i<atributos.getLength(); i++) {
            ps.print(" @" + atributos.getLocalName(i) + "[" + atributos.
                getValue(i)+ "]");
        }
        ps.println(">");
    }
    @Override
    public void endElement(String uri, String nombreLocal, String
        nombreCualif) {
        nivel--;
    }
    @Override
    public void characters(char[] cars, int inicio, int longitud) {
        String texto = new String(cars, inicio, longitud);
        if (texto.trim().length() == 0) {
            return;
        }
        nivel++;
        indenta();
        nivel--;
        ps.println("#text[" + texto + "]");
    }
}
public class SAX_Parser {
    public static void main(String[] args) {
        String nomFich;

```

```
if (args.length < 1) {
    System.out.println("Indicar por favor nombre de fichero");
    return;
} else {
    nomFich = args[0];
}
try {
    XMLReader parserSAX = XMLReaderFactory.createXMLReader();
    GestorEventos gestorEventos = new GestorEventos(System.out);
    parserSAX.setContentHandler(gestorEventos);
    parserSAX.parse(nomFich);
} catch (SAXException e) {
    System.err.println(e.getMessage());
} catch (Exception e) {
    e.printStackTrace();
}
```

6.7. Validación de documentos de XML

Un documento de XML bien formado es, además, válido si satisface un conjunto de restricciones adicionales sobre su estructura y contenidos. La estructura se refiere a qué tipo de elementos puede tener, dónde y con qué atributos. El contenido se refiere a qué valores puede tener el contenido de cada tipo de elemento y cada atributo. Hay distintos mecanismos para validación de documentos de XML, de los que los más importantes son:

1. *DTD*. Son las siglas de *data type definition* o definición de tipos de datos. Fue el primer mecanismo de validación para XML. De hecho, es previo a XML y se creó para SGML, un precedente de HTML y XML. Hoy en día su uso ha declinado en favor de los esquemas de XML.
 2. *Esquemas de XML*. Es un mecanismo mucho más potente. Los esquemas se almacenan en ficheros XSD (XML Schema Definition), que son documentos de XML.

6.7.1. Validación con DTD

DTD permite especificar la estructura y el contenido de un documento de XML. En lo referente a estructura, qué elementos (identificados por su nombre) pueden aparecer, dónde y cuántas veces, y qué atributos pueden tener. En lo referente a contenido, qué valores se admiten para cada elemento y atributo. Su funcionalidad es mucho más limitada que los esquemas de XML y tiene una sintaxis diferente, al contrario que los esquemas, que son documentos de XML.



En el anexo web 6.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás más información sobre DTD.

6.7.2. Validación con esquemas de XML

Los esquemas de XML son documentos de XML que permiten verificar o validar los contenidos de un documento de XML.

A continuación, se incluye un esquema de XML para el documento de ejemplo con datos de clientes. Al final del documento se puede ver cómo los esquemas de XML permiten especificar todo tipo de restricciones sobre los valores aceptables para cada elemento o atributo.

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema version="1.0"
  xmlns:xss="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xss:element name="clientes">
    <xss:complexType>
      <xss:sequence>
        <xss:element ref="cliente" minOccurs="0" maxOccurs="unbounded"/>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
  <xss:element name="cliente">
    <xss:complexType>
      <xss:sequence>
        <xss:element name="apellidos" type="xs:string"/>
        <xss:element name="CP" minOccurs="0" type="tipo_CP"/>
        <xss:element ref="validez" minOccurs="0"/>
      </xss:sequence>
      <xss:attribute name="DNI" use="required" type="tipo_DNI"/>
    </xss:complexType>
  </xss:element>
  <xss:element name="validez">
    <xss:complexType>
      <xss:attribute name="estado" use="required" type="tipo_estado"/>
      <xss:attribute name="timestamp" use="optional" type="tipo_timestamp"/>
    </xss:complexType>
  </xss:element>
  <xss:simpleType name="tipo_CP">
    <xss:restriction base="xs:string">
      <xss:pattern value="\d{5}"/>
    </xss:restriction>
  </xss:simpleType>
  <xss:simpleType name="tipo_timestamp">
    <xss:restriction base="xs:string">
      <xss:pattern value="\d{10}"/>
    </xss:restriction>
  </xss:simpleType>
  <xss:simpleType name="tipo_DNI">
    <xss:restriction base="xs:string">
      <xss:pattern value="[\d-9XYZ]\d{7}[\dA-Z]"/>
    </xss:restriction>
  </xss:simpleType>
  <xss:simpleType name="tipo_estado">
    <xss:restriction base="xs:string">
      <xss:enumeration value="borrado"/>
      <xss:enumeration value="vigente"/>
    </xss:restriction>
  </xss:simpleType>
</xss:schema>
```

Se puede especificar el esquema para validación dentro del propio documento de XML y añadir una referencia al esquema dentro de la etiqueta de apertura del elemento `clientes`, con lo que quedaría así:

```
<?xml version="1.0" encoding="UTF-8"?>
<clientes xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
           xsi:noNamespaceSchemaLocation='clientes.xsd'>
</clientes>
```

Esto puede ser necesario o cómodo para validar el XML con algunas herramientas, porque automáticamente se localiza y utiliza el esquema. Pero puede plantear problemas con otras herramientas, y en particular con los programas de ejemplo que se desarrollarán en breve. En un sistema real el esquema estaría normalmente situado en un servidor web y se accedería a él no por un nombre de fichero local, sino mediante un URI de tipo `http://`.



Actividad propuesta 6.3

Dado que un esquema de XML es un tipo de documento de XML, ¿piensas que puede tener sentido o utilidad un esquema de XML para verificar los contenidos de esquemas de XML? ¿En qué situaciones? Justifica tu respuesta.

6.8. Parsing con validaciones con Java

Las clases `DocumentBuilderFactory` para *parsing* DOM y `SAXParserFactory` para *parsing* SAX tienen métodos para que el *parser* valide los XML con un DTD o esquema. De esta manera, se producirán excepciones si el XML no es válido.

CUADRO 6.10
Métodos de la clase DocumentBuilderFactory y SAXParserFactory para validación

Método	Funcionalidad
<code>setValidating(boolean validating)</code>	Realiza validación utilizando un DTD si se llama con valor <code>true</code> para <code>validating</code> . El DTD debe estar referenciado desde o incluido en el mismo XML.
<code>void setSchema(Schema schema)</code>	Especifica el esquema que se va a utilizar para validar los documentos.
<code>void setIgnoringElementContentWhitespace(boolean whitespace)</code>	Solo en el caso de que se realice validación, hará que se ignoren los textos vacíos o ignorables en el contenido de los elementos.

A parte de utilizar estos métodos, debe hacerse una apropiada gestión de los errores de validación que se puedan producir. No hay que confundir estos con los errores de sintaxis, que se producen si el documento de XML no está bien formado, es decir, no es correcto sintácticamente. Aunque pueda parecer un tanto extraño, para gestionar los errores de validación hay que utilizar el sistema de gestión de eventos de SAX, aunque se utilice un *parser* DOM. Esto significa, por cierto, que para la implementación del *parser* DOM se utiliza internamente un *parser* SAX.

CUADRO 6.11

Métodos de la clase `DefaultHandler` para gestión de errores y avisos de validación

Método	Funcionalidad
<pre>void error(SAXParseException e) throws SAXException void fatalError(SAXParseException e) throws SAXException</pre>	Error recuperable o no recuperable durante el parsing. En el primer caso, se puede continuar el parsing. En el segundo, no se puede.
<pre>void warning(SAXParseException e) throws SAXException</pre>	Aviso durante el parsing.

A continuación se muestra un ejemplo de *parsing* DOM con validación. El fichero XML se le pasa mediante el primer parámetro de línea de comandos. Si se le pasa un fichero con un esquema de XML en el segundo parámetro, la validación se hace con ese esquema, y en otro caso, con un DTD que debe estar referenciado desde el propio fichero XML.



TOMA NOTA

Los atributos que se pueden añadir a elementos de un documento de XML relativos a esquemas, como los del siguiente ejemplo, no son necesarios para los programas de ejemplo que siguen, porque en ellos se valida el documento entero con un esquema de XML indicado en línea de comandos.

```
<clientes xmlns:xsi=
'http://www.w3.org/2001/XMLSchema-instance'
xsi:noNamespaceSchemaLocation='clientes.xsd'>
```

Además, harán que falle siempre la validación. Si están, hay que eliminarlos, y dejar:

```
<clientes>
```

La validación con DTD se consigue con `setValidating(true)`. Con esquemas de XML habría que utilizar, en cambio, `setSchema(Schema schema)`. Los errores y avisos se gestionan con un objeto de una clase manejadora de eventos, hija de `DefaultHandler`, la misma que se usó para el ejemplo de *parsing* SAX. A sus métodos manejadores de errores (y avisos) se les pasa un `SAXParseException` que se lanza de nuevo con `throws`. En el programa principal se capturan las excepciones de este tipo y no se hace nada con ellas. Para avisos y errores recuperables, podría no relanzarse la excepción para seguir con el *parsing*.

```

// Parsing DOM con validación basada en DTD
package DOMParserValid;
import java.io.File;
import java.io.PrintStream;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.validation.SchemaFactory;
import javax.xml.XMLConstants;
public class DOMParserValid {
    private static final String INDENT_NIVEL = " "; // Para indentación
    public static void muestraNodo(Node nodo, int nivel, PrintStream ps) {
        for (int i = 0; i < nivel; i++) {
            ps.print(INDENT_NIVEL);
        }
        switch (nodo.getNodeType()) { // Escribe información de nodo según
            tipo
            case Node.DOCUMENT_NODE: // Documento
                ps.println("DOCUMENTO");
                break;
            case Node.ELEMENT_NODE: // Elemento
                ps.println("ELEMENTO(" + nodo.getNodeName() + ")");
                break;
            case Node.TEXT_NODE: // Texto
                ps.println(nodo.getNodeName() + "[" + nodo.getNodeValue() +
                           "]");
                break;
        }
        NodeList nodosHijos = nodo.getChildNodes(); // Nodos hijos
        for (int i = 0; i < nodosHijos.getLength(); i++) {
            muestraNodo(nodosHijos.item(i), nivel + 1, ps);
        }
    }
    public static void main(String[] args) {
        File f = null, fEsq = null;
        if (args.length < 1) {
            System.out.println("Indicar por favor nombre de fichero");
            return;
        } else {
            String nomFich = args[0];
            f = new File(nomFich);
            if (!f.isFile()) {
                System.err.println("Fichero " + nomFich + " no existe.");
                return;
            }
            if (args.length >= 2) {
                String nomFichEsquema = args[1];
                fEsq = new File(nomFichEsquema);
                if (!fEsq.isFile())) {

```

```

        System.err.println("Fichero " + nomFichEsquema + " no
                           existe.");
    }
}

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setIgnoringComments(true);
dbf.setIgnoringElementContentWhitespace(true);
System.out.println("Fichero XML: " + f.getAbsolutePath());
if (fEsq != null) { // Validar con esquema
    System.out.println("Validación con esquema: " + fEsq.getAbsolutePath());
    try {
        dbf.setSchema(
            SchemaFactory.newInstance(
                XMLConstants.W3C_XML_SCHEMA_NS_URI).newSchema(fEsq)
            );
    } catch (SAXException e) {
        System.err.println(e.getMessage());
        return;
    }
} else { // Validar con DTD
    System.out.println("Validación con DTD");
    dbf.setValidating(true);
}
try {
    DocumentBuilder db = dbf.newDocumentBuilder();
    db.setErrorHandler(new GestorEventos());
    Document domDoc = db.parse(f);
    muestraNodo(domDoc, 0, System.out);
} catch (ParserConfigurationException e) {
    System.err.println(e.getMessage());
} catch (SAXParseException e) { // Ya gestionada en GestorEventos
} catch (Exception e) {
    e.printStackTrace();
}
}
}

class GestorEventos extends DefaultHandler {
    @Override
    public void error(SAXParseException e) throws SAXParseException {
        System.err.println("Error recuperable: " + e.toString());
        throw (e);
    }
    @Override
    public void fatalError(SAXParseException e) throws SAXParseException {
        System.err.println("Error no recuperable: " + e.toString());
        throw (e);
    }
    @Override
    public void warning(SAXParseException e) throws SAXParseException {
        System.err.println("Aviso: " + e.toString());
        throw (e);
    }
}

```

Para *parsing* SAX se haría de manera análoga, pero utilizando `SAXParserFactory` y `SAXParser` en lugar de `DocumentBuilderFactory` y `DocumentBuilder`. Hay, además, alguna otra pequeña variación. Por ejemplo, es necesario proporcionar dos clases manejadoras distintas: una para contenidos y otra para errores y avisos. Pero nada impide usar la misma, siempre que en ella estén definidos los métodos manejadores de errores y eventos. Así se ha hecho en el siguiente programa de ejemplo. Como en el anterior programa, el fichero XML se pasa mediante el primer argumento de línea de comandos. Si se pasa un fichero con un esquema de XML en el segundo, la validación se hace con ese esquema, y en otro caso con un DTD que debe estar referenciado desde el propio fichero XML.

```
// Parsing SAX con validación basada en DTD
package SAXParserValid;
import java.io.File;
import java.io.PrintStream;
import javax.xml.XMLConstants;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import javax.xml.validation.SchemaFactory;

class GestorEventos extends DefaultHandler {
    private static final String INDENT_NIVEL = " "; // Para indentación
    private final PrintStream ps;
    private int nivel;
    private void indenta() {
        for (int i = 0; i < nivel; i++) {
            ps.print(INDENT_NIVEL);
        }
    }
    public GestorEventos(PrintStream ps) {
        this.ps = ps;
    }
    @Override
    public void startDocument() {
        nivel = 0;
        ps.println("{DOCUMENTO}");
    }
    @Override
    public void startElement(String uri, String nombreLocal, String
        nombreCualif, Attributes atributos) {
        nivel++;
        indenta();
        ps.println("ELEMENTO(" + nombreCualif + ")");
    }
    @Override
    public void endElement(String uri, String nombreLocal, String
        nombreCualif) {
        nivel--;
    }
}
```

```

@Override
public void characters(char[] chars, int inicio, int longitud) {
    String texto = new String(chars, inicio, longitud);
    nivel++;
    indentar();
    nivel--;
    ps.println("TEXTO{" + texto + "}");
}
@Override
public void error(SAXParseException e) throws SAXParseException {
    System.err.println("Error recuperable: " + e.toString());
    throw (e);
}
@Override
public void fatalError(SAXParseException e) throws SAXParseException {
    System.err.println("Error no recuperable: " + e.toString());
    throw (e);
}
@Override
public void warning(SAXParseException e) throws SAXParseException {
    System.err.println("Aviso: " + e.toString());
    throw (e);
}
}
public class SAXParserValid {
    public static void main(String[] args) {
        File f = null, fEsq = null;
        if (args.length < 1) {
            System.out.println("Indicar por favor nombre de fichero");
            return;
        } else {
            String nomFich = args[0];
            f = new File(nomFich);
            if (!f.isFile()) {
                System.err.println("Fichero " + nomFich + " no existe.");
                return;
            }
            if (args.length >= 2) {
                String nomFichEsquema = args[1];
                fEsq = new File(nomFichEsquema);
                if (!fEsq.isFile()) {
                    System.err.println("Fichero " + nomFichEsquema + " no
                        existe.");
                    return;
                }
            }
        }
        try {
            SAXParserFactory spf = SAXParserFactory.newInstance();
            System.out.println("Fichero XML: " + f.getAbsolutePath());
            if (fEsq != null) { // Validar con esquemas
                System.out.println("Validación con esquema: " +
                    fEsq.getAbsolutePath());
            }
        }
    }
}

```

```

        try {
            spf.setSchema(
                SchemaFactory.newInstance(
                    XMLConstants.W3C_XML_SCHEMA_NS_URI).newSchema(fEsq)
                );
        } catch (SAXException e) {
            System.err.println(e.getMessage());
            return;
        }
    } else { // Validar con DTD
        System.out.println("Validación con DTD");
        spf.setValidating(true);
    }
    SAXParser parserSAX = spf.newSAXParser();
    XMLReader xmlReader = parserSAX.getXMLReader();
    GestorEventos gestorEventos = new GestorEventos(System.out);
    xmlReader.setContentHandler(gestorEventos);
    xmlReader.setErrorHandler(gestorEventos);
    xmlReader.parse(f.getAbsolutePath());
} catch (SAXException e) {
    System.err.println(e.getMessage());
} catch (Exception e) {
    e.printStackTrace();
}
}

```



Actividad propuesta 6.4

Prueba los dos programas anteriores (parsers DOM y SAX con validación). Utiliza la validación mediante esquema de XML. Puedes utilizar el documento de prueba proporcionado con datos de clientes o cualquier otro documento de XML, siempre que dispongas de su correspondiente esquema o que lo crees tú mismo. Cambia el documento de XML para introducir diversos errores de sintaxis y de validación en el documento de XML, y comprueba qué errores se detectan. Verifica qué sucede si se cambia el esquema de XML para introducir errores en él. Modifica los manejadores de eventos para que solo se relancen las excepciones en caso de aviso o de error recuperable, de manera que se pueda seguir validando para mostrar más avisos o errores. Recuerda: CTRL+Z para deshacer cambios, CTRL+Y para rehacer cambios.

6.9. Binding con JAXB

JAXB proporciona una API que permite realizar de manera muy sencilla la traducción de un documento de XML a una colección de objetos (*unmarshalling* o deserialización), y a la inversa, la traducción de una colección de objetos a un documento de XML (*marshalling* o serialización).

**Figura 6.3**

Binding con JAXB para persistencia de objetos en documentos de XML

El paso previo para utilizar JAXB es la generación —a partir de un esquema de XML y utilizando el compilador de *binding xjc*— de una colección de clases de Java que representan sus elementos y las relaciones entre ellos, y de interfaces y clases que implementan los procesos de *marshalling* y *unmarshalling* (incluyendo este último el *parsing* de los ficheros XML y su validación con el esquema de XML).

6.9.1. Esquemas de XML para *binding*

Para que se pueda navegar sin problemas desde cualquier objeto creado mediante el proceso de *unmarshalling* a cualesquier otros relacionados, es aconsejable que cualquier elemento (`<xs:element>`) que pueda contener otros elementos, o bien repetirse varias veces (en cualquiera de estos dos casos se define con `<xs:complexType>`), tenga una definición aparte y no incluida dentro de la de otro elemento. Siempre que se necesite se puede hacer referencia a su definición mediante `<xs:element ref="...">`. Así se garantiza que el compilador de *binding* crea las clases apropiadas para ellos. Se puede verificar que el esquema de ejemplo anterior es conforme a estas recomendaciones.

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema version="1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xsi:element name="clientes">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="cliente" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xsi:element>
  <xsi:element name="cliente">
    <xs:complexType>
      <xs:sequence>
        {...}
      </xs:sequence>
      <xs:attribute name="DNI" use="required" type="tipo_DNI"/>
    </xs:complexType>
  </xsi:element>
  {...}
</xs:schema>

```

Figura 6.4

Esquema para un documento de XML apropiado para JAXB

6.9.2. Compilador de *binding*

El compilador de *binding* es un programa llamado `xjc` que se puede usar directamente desde línea de comandos o indirectamente desde un IDE (entorno de desarrollo integrado) como NetBeans o Eclipse. Se desarrollará ahora un programa de ejemplo para el esquema de ejemplo `clientes.xsd`, empezando por la generación de las clases a partir del esquema mediante el compilador de *binding*.

Recurso digital

En el anexo web 6.2, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás un aviso para los usuarios de Windows.

- Desde línea de comandos:

```
xjc clientes.xsd
```

- Se pueden utilizar opciones para, entre otras cosas, indicar el nombre de paquete y el directorio donde se quiere que se generen las clases e interfaces.
- Desde el entorno de desarrollo NetBeans: se añade el esquema de XML (fichero xsd) al proyecto desde el que se quiere utilizar JAXB. Después se selecciona el menú “File”, opción “New File”, se selecciona “XML” y después “JAXB Binding”, o bien se pulsa con el botón derecho y se selecciona “New” y después “JAXB Binding”.

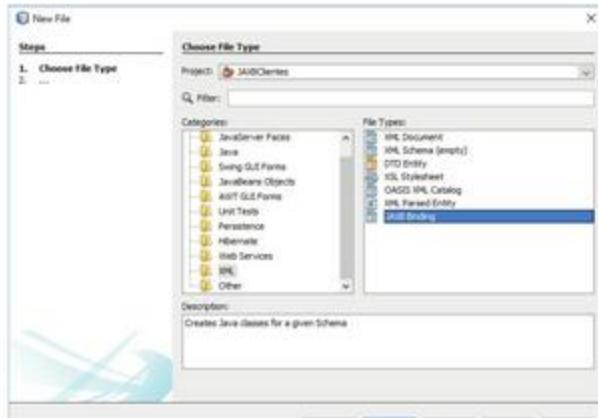


Figura 6.5

Creación de *binding* para esquema de XML con Netbeans

En el siguiente diálogo hay que indicar el fichero de esquema, el nombre para el contexto de *binding* y el nombre de un paquete para incluir las clases e interfaces creadas por el compi-

lador de *binding* `xjc`. Para estos dos últimos se ha especificado `clientes_JAXB`. Se muestran estas opciones y cómo queda el proyecto.

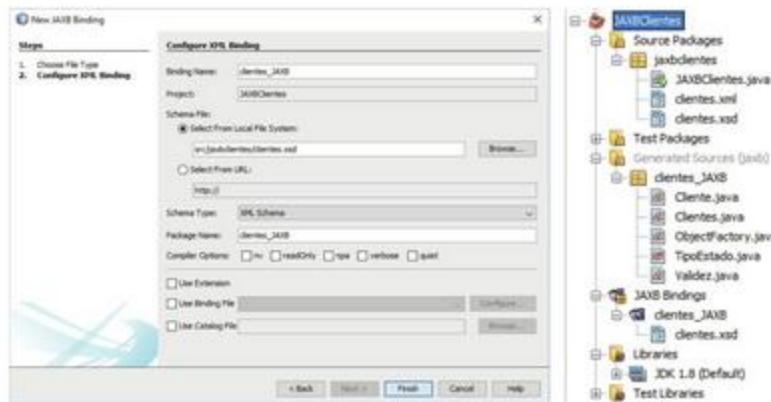


Figura 6.6
Configuración del *binding* y proyecto final

6.9.3. Clases generadas por el compilador de *binding*

Todas las clases generadas por el compilador de *binding* disponen de métodos `get` y `set` (`getters` y `setters`) para obtener y asignar los valores de cada atributo y de cada elemento que no sea de un tipo complejo `<xs:complexType>`. Si dentro del elemento `<xs:element>` al que está asociada una clase hay otro elemento `<xs:element>` que se puede repetir muchas veces (atributo `maxOccurs` de `<xs:element>` con valor mayor que uno o `unbounded`), habrá un método `get` que devuelve una lista de objetos de la clase correspondiente al otro elemento, no existirá el correspondiente método `set`, y los contenidos de la lista deberán modificarse con los métodos de la interfaz `List`. En el cuadro 6.12 se muestran todos los métodos.

CUADRO 6.12
Getters y setters generados por el compilador de *binding*

Clase	Métodos
Clientes	public List<Cliente> getCliente()
Cliente	public String getApellidos() public void setApellidos(String value) public String getCP() public void setCP(String value) public Validez getValidez() public void setValidez(Validez value) public String getDNI() public void setDNI(String value)
	[.../...]

CUADRO 6.19 (CONT.)

Se ha generado otra clase `TipoEstado` para el elemento `validez` de tipo simple (`<xs:simpleType>`) basado en una enumeración (`<xs:enumeration>`). Su uso es muy sencillo y se verá en un ejemplo posterior.

6.9.4. *Unmarshalling* (deserialización) con JAXB

Ahora se está en condiciones de escribir programas que utilicen la funcionalidad de *binding* para acceder a los datos en ficheros de XML conformes al esquema de XML. La API JAXB está disponible en el paquete `javax.xml.bind`.

A continuación, se muestra un programa de ejemplo para ilustrar la funcionalidad de JAXB. Primero se crea un `JAXBContext` asociado a la clase creada por el compilador de *binding* para el elemento al nivel más alto del esquema, `clientes_JAXB.Clientes.class` (`clientes_JAXB` es el paquete creado por el compilador de *binding*). A continuación, se crea un `Unmarshaller` asociado a este contexto. Después, se le asocia un esquema para validación y un manejador de eventos para errores y avisos. Por último, se invoca el método `unmarshal`, que devuelve un objeto de tipo `clientes_JAXB.Clientes` para todos los contenidos del fichero XML, a los que se puede acceder mediante los métodos `get` de las clases creadas por el compilador de *binding*. Los eventos que se van a gestionar pueden ser de tres tipos: error no recuperable, error recuperable o aviso. Todos se gestionan en un único método, y si devuelve `true`, se continúa con el proceso. En este programa se continua a menos que se trate de un error no recuperable. En caso de error recuperable o aviso, se devuelve `true` para poder continuar con el proceso e informar de más errores o avisos, en caso de que los haya.

```
// Unmarshalling con JAXB para mostrar contenidos de un documento de XML

package jaxbclientes;

import java.util.List;
import java.io.File;
import javax.xml.XMLConstants;
import javax.xml.validation.SchemaFactory;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.JAXBException;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.ValidationEvent;
import javax.xml.bind.ValidationEventHandler;

public class JAXBClientes {

    public static void main(String[] args) {
        File f = null, fEsq = null;
        if (args.length < 1) {
            System.out.println("Indicar por favor nombre de fichero");
            return;
        }
        SchemaFactory sf = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Schema schema = sf.newSchema(f);
        JAXBContext jc = JAXBContext.newInstance("jaxbclientes");
        Unmarshaller um = jc.createUnmarshaller();
        um.setSchema(schema);
        try {
            List<ValidationEvent> events = um.getValidationEvents();
            ValidationEvent event = events.iterator().next();
            System.out.println("Error en linea " + event.getLineNumber() + " columna " + event.getColumnNumber());
            System.out.println(event.getMessage());
        } catch (UnmarshalException e) {
            e.printStackTrace();
        }
    }
}
```

```

} else {
    String nomFich = args[0];
    f = new File(nomFich);
    if (!f.isFile()) {
        System.err.println("Fichero " + nomFich + " no existe.");
        return;
    }
    if (args.length >= 2) {
        String nomFichEsquema = args[1];
        fEsq = new File(nomFichEsquema);
        if (!fEsq.isFile()) {
            System.err.println("Fichero " + nomFichEsquema + " no
                existe.");
            return;
        }
    }
    System.out.println("Fichero XML: " + f.getAbsolutePath());
    if (fEsq != null) { // Validar con esquema
        System.out.println("Valid. con esquema: " + fEsq.
            getAbsolutePath());
    }
}

try {
    JAXBContext contextoClientes = JAXBContext.newInstance(
        clientes_JAXB.Clientes.class // Contexto para clase Clientes
    );
    // Unmarshaller para contexto
    Unmarshaller u = contextoClientes.createUnmarshaller();

    u.setSchema(SchemaFactory.newInstance( // Esquema para validación
        XMLConstants.W3C_XML_SCHEMA_NS_URI).newSchema(fEsq));
    u.setEventHandler(new GestorEventos()); // Gestor de eventos

    clientes_JAXB.Clientes clientes = (clientes_JAXB.Clientes)
        u.unmarshal(f); // unmarshalling, que incluye parsing y
        validación
    // Iteración sobre objeto para mostrar los contenidos
    List<clientes_JAXB.Cliente> listaClientes = clientes.getCliente();
    for (clientes_JAXB.Cliente cliente: listaClientes) {
        System.out.println("-> DNI: " + cliente.getDNI());
        System.out.println("Apellidos: " + cliente.getApellidos());
        String cp = cliente.getCP();
        if (cp != null) {
            System.out.println("CP: " + cp);
        }
        clientes_JAXB.Validez validez = cliente.getValidez();
        if (validez != null) {
            System.out.println("Validez: " + validez.getEstado());
            String timestamp = validez.getTimestamp();
            if (timestamp.length() > 0) {
                System.out.println("timestamp: " + timestamp);
            }
        }
    }
} catch (UnmarshalException e) {
    System.err.println(e.getMessage());
}

```

```

        } catch (JAXBException e) {
            System.err.println(e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class GestorEventos implements ValidationEventHandler {
    @Override
    public boolean handleEvent(ValidationEvent e) {
        System.err.println("Evento de validación (" + e.getSeverity() + ")"
+ "[línea: " + e.getLocator().getLineNumber() + ", "
+ "columna: " + e.getLocator().getColumnNumber() + "]: "
+ e.getMessage());
        return (e.getSeverity() != ValidationEvent.FATAL_ERROR);
    }
}

```



Actividad propuesta 6.5

Crea un nuevo esquema de XML a partir del esquema de ejemplo para clientes, y añade para cada cliente la posibilidad de especificar en un elemento opcional `max_descuento` una cantidad entera entre 1 y 60 que represente el máximo porcentaje de descuento para el cliente (si no sabes cómo, debes investigar acerca de XML Schema). Vuelve a generar las clases a partir del esquema y cambia el programa para que muestre este porcentaje para los clientes en los que esté informado.

6.9.5. Marshalling (serialización) con JAXB

A continuación se muestra un programa que utiliza JAXB para crear un fichero XML con los datos de dos clientes. Para construir la estructura de objetos se utilizan los *setters* de las clases generadas por el compilador de *binding*. Para añadir elementos en una secuencia se utilizan los métodos de la interfaz `List`, típicamente `add()`, para añadir al final de la lista. Para asignar los posibles valores del atributo `estado` se utilizan constantes definidas en `clientes_JAXB.TipoEstado`. Para serializar la estructura de objetos se crea un contexto `JAXBContext` para la clase `clientes_JAXB.Clientes`. Después, se crea un `Marshaller` para este contexto y, por último, se genera el fichero con el método `marshal()`.

```

// Marshalling con JAXB para generar un documento de XML
package jaxbelclientscrear;

import java.io.File;
import java.util.List;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.JAXBException;

public class JAXBClientesCrear {

    public static void main(String[] args) {

```

```

File f = null;
if (args.length < 1) {
    System.out.println("Indicar por favor nombre de fichero");
    return;
} else {
    String nomFich = args[0];
    f = new File(nomFich);
    if (f.isFile()) {
        System.err.println("Fichero "+nomFich+" existe, no se hace
            nada");
        return;
    }
}

clientes_JAXB.Clientes clientes = new clientes_JAXB.Clientes();
List<clientes_JAXB.Cliente> listaClientes = clientes.getCliente();

clientes_JAXB.Cliente cliente = new clientes_JAXB.Cliente();
cliente.setDNI("78901234X");
cliente.setApellidos("NADALES");
cliente.setCP("44126");
listaClientes.add(cliente);

cliente = new clientes_JAXB.Cliente();
cliente.setDNI("89012345E");
cliente.setApellidos("ROJAS");
clientes_JAXB.Validez validez = new clientes_JAXB.Validez();
validez.setEstado(clientes_JAXB.TipoEstado.VIGENTE);
validez.setTimestamp("1528286082");
cliente.setValidez(validez);
listaClientes.add(cliente);

try {
    JAXBContext contextoClientes = JAXBContext.newInstance(
        clientes_JAXB.Clientes.class // Crea contexto para objeto
        Clientes
    );
    Marshaller m=contextoClientes.createMarshaller(); // Crea
    marshaller

    m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    m.marshal(clientes, f);

    System.out.println("Generado fichero XML: " + f.getAbsolutePath());
} catch (JAXBException e) {
    System.err.println(e.getMessage());
} catch (Exception e) {
    e.printStackTrace();
}
}

```

Actividad propuesta 6.6



Modifica el programa anterior para que muestre el documento de XML en pantalla, en vez de crear un fichero.

6.10. El lenguaje de consulta XPath

XPath es un lenguaje de consulta para documentos de XML. Fue desarrollado inicialmente para el lenguaje XSLT, el cual se verá en el siguiente apartado. En él se basan otros estándares de XML que también se verán más adelante, como XQuery, que se estudiará en un capítulo posterior dedicado a bases de datos de XML.

Una expresión (o consulta) de XPath permite recuperar contenidos de un documento de XML representado en forma de árbol. Una expresión de XPath devuelve un conjunto de nodos. Esto recuerda al modelo DOM. La especificación de XPath incluye su propio modelo de datos, muy similar al modelo DOM, pero con algunas diferencias. Las implementaciones de XPath en Java y, en general, en todos los lenguajes de programación se basan en DOM.

Una expresión de XPath consta de una secuencia de pasos de búsqueda, separados por el carácter “/”. La búsqueda comienza por el nodo raíz del árbol, al que se le aplica el primer paso, lo que proporciona una primera lista de nodos como resultado. Cada paso sucesivo se aplica sobre la lista de resultados del paso anterior, lo que proporciona una nueva lista de resultados. Como resultado del último paso se obtiene la lista de resultados de la expresión de XPath.

Se muestra un ejemplo del funcionamiento de XPath. Supongamos que se quieren recuperar todos los DNI de clientes que tengan informado el código postal. Esto se podría hacer con la expresión /clientes/cliente/CP/../../@DNI. El paso “..” se utiliza para obtener el nodo padre. “@” se utiliza para obtener el valor de un atributo.

<pre><?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE clientes SYSTEM "clientes. dtd"> <clientes> <cliente DNI="78901234X"> <apellidos>NADALES</apellidos> <CP>44126</CP> </cliente> <cliente DNI="89012345E"> <apellidos>ROJAS</apellidos> <valores estado="borrado"> timestamp="1528286082"/> </cliente> <cliente DNI="56789012B"> <apellidos>SAMPER</apellidos> <CP>29730</CP> </cliente> </clientes></pre>	<pre>/clientes [<clientes>] /clientes/cliente [<cliente DNI="78901234X">...</cliente>, <cliente DNI="89012345E">...</cliente>, <cliente DNI="56789012B">...</cliente>] /clientes/cliente/CP [<CP>44126</CP>, <CP>29730</CP>] /clientes/cliente/CP/.. [<cliente DNI="78901234X">...</cliente>, <cliente DNI="56789012B">...</cliente>] /clientes/cliente/CP/../../@DNI ["78901234X", "56789012B"]</pre>
---	--

Figura 6.7
Resultados de una consulta de XPath, explicada paso a paso

A continuación se muestra un programa en Java que permite obtener los resultados de una expresión de XPath sobre un documento de XML. Después, se incluirán varios ejemplos de consultas de XPath, cuyos resultados se podrán obtener utilizando este programa.

A este programa se le pasa un nombre de fichero XML y una expresión de XPath como argumentos de línea de comandos, o bien desde consola (`BufferedReader` construido sobre

`System.in`), y muestra la lista de resultados obtenidos. Primero, se hace un *parsing* DOM del documento para obtener un `Document` con sus contenidos. Después se crea un `XPathExpression` a partir de un `String` que contiene la expresión de XPath. Por último, `evaluate` devuelve en un `NodeList` la lista de resultados de aplicar el XPath sobre el `Document`. Hay otras maneras de obtener los resultados, pero en este programa se ha seleccionado la opción `XPathConstants.NODESET`. Para visualizar el contenido de los nodos, incluyendo el de sus nodos hijos, se utiliza el método `getTextContent` de la clase `Node`.

```
// Consulta en un documento de XML mediante XPath
package xPathConsulta;

import java.io.File;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathException;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.xpath.XPathConstants;

public class XPathConsulta {

    public static void main(String[] args) {
        String nomFich, xPath = "";
        File fXML;

        if (args.length < 1) {
            System.out.println("Por favor, indicar nombre de fichero.");
            return;
        } else {
            nomFich = args[0];
            System.out.println("Fichero XML: " + nomFich);
            if (args.length < 2) {
                System.out.print("Introducir XPath: ");
                BufferedReader br = new BufferedReader(
                    new InputStreamReader(System.in));
                try {
                    xPath = br.readLine();
                } catch (Exception e) {}
            } else {
                xPath = args[1];
            }
        }
        fXML = new File(nomFich);
        if (!fXML.isFile()) {
            System.err.println("Fichero " + nomFich + " no existe.");
            return;
        }
        System.out.println("Fichero XML: " + fXML.getAbsolutePath() + " XPath:
                           " + xPath);
    }
}
```

```
try {
    Document docXML = DocumentBuilderFactory.newInstance().
        newDocumentBuilder().parse(fXML);
    XPathExpression xPathExp = XPathFactory.newInstance().newXPath();
    compile(xPath);
    NodeList resultados = (NodeList) xPathExp.evaluate(
        docXML, XPathConstants.NODESET);
    for (int i = 0; i < resultados.getLength(); i++) {
        System.out.printf("%td: %s\n", i + 1,
            resultados.item(i).getTextContent());
    }
    if (resultados.getLength() < 1) {
        System.out.println("No se obtuvo ningún resultado.");
    }
} catch (SAXException | XPathException e) {
    System.err.println(e.getMessage());
} catch (Exception e) {
    e.printStackTrace();
}
```

En el cuadro 6.13 se recogen algunos ejemplos más de consultas con XPath.

CUADRO 6.13 Varios ejemplos de XPath

//@DNI	El paso “vacío” obtiene todos los nodos por debajo del nodo actual. Este XPath muestra todos los valores de cualquier atributo con nombre “DNI”, en cualquier elemento.
/clientes/*/@DNI	El paso * obtiene todos los nodos por debajo del nodo actual.
/clientes/cliente[apellidos="SAMPER"]	Obtiene elemento cliente para cliente que tiene como apellidos “SAMPER”.
/clientes/cliente[@DNI="89012345E"]	Obtiene elemento cliente para el cliente con DNI 89012345E.
/clientes/cliente[CP=44126]/apellidos	Obtiene apellidos de cliente con código postal 44126.
//validez/@*	Obtiene valor de todos los atributos de cualquier elemento de nombre validez.
/clientes/cliente[CP>=29000 and CP<30000]	Devuelve elementos cliente de clientes con código postal mayor o igual que 29000 y menor que 30000.
/clientes/cliente[position()=2]	Devuelve el nombre del segundo cliente.
/clientes/cliente[position()=last()]/@DNI	Devuelve el DNI del último cliente.

Actividades propuestas



- 6.7.** Modifica el programa anterior para que muestre todos los contenidos de los nodos devueltos por cada consulta, utilizando un método como `muestraNodo`, utilizado en los programas de ejemplo anteriores para parsing DOM. Se trata, simplemente, de copiar el método y hacerlo funcionar en este programa. Prueba distintas opciones para devolver los resultados de la búsqueda con XPath. En particular, otros valores en lugar de `XPathConstants.NODESET`, como `XPathConstants.NODE` o `XPathConstants.STRING`. Pueden ser necesarios cambios adicionales en el programa, al cambiar el tipo de objeto devuelto (ya no sería un `NodeList`). Piensa en qué situaciones podrían ser útiles estas opciones.
- Prueba con los XPath anteriores y experimenta con otros nuevos, y con otros ficheros de XML.
- 6.8.** Investiga la forma de definir una clave primaria para un elemento en un esquema de XML con `<xs:key>` y XPath. Apícalo sobre el esquema de ejemplo para datos de clientes para verificar que no hay DNI duplicados.

6.11. El lenguaje XSL

XSL (del inglés *extended stylesheet language*) es un lenguaje de transformación que permite generar nuevos documentos a partir de un documento de XML. A los documentos en lenguaje XSL se les suele llamar plantillas XSL, y son documentos de XML.



Figura 6.8
El lenguaje de transformación XSL

El nombre *lenguaje de hojas de estilo* viene del uso para el que fue concebido originalmente: la generación, a partir de unos mismos datos (en un documento de XML), de distintos documentos en lenguaje HTML o similares, según la plantilla utilizada, para adaptar la presentación

a distintos dispositivos, desacoplando así los contenidos (en XML) de la presentación (en XSL). Pero XSL permite generar documentos en cualquier formato de texto, además de en HTML o en XML, como por ejemplo en formato CSV.

Sería muy largo explicar todas las posibilidades del lenguaje XSL. Baste indicar que es un lenguaje muy potente con sentencias de asignación, condicionales, bucles, posibilidad de definir funciones, etc. Se muestran dos plantillas de ejemplo para el documento de XML con datos de clientes, una al lado de la otra para facilitar su comparación. Una genera HTML, que se puede mostrar en un navegador web. La otra genera un documento de tipo CSV, pero con campos, y por ende registros, de longitud fija (y rellena con espacios por la derecha con `concat` y `substring`), y con el carácter “|” como separador de campos y con un salto de línea como separador de registros. O, visto de otra manera, un fichero con campos y registros de longitud fija como los vistos en el capítulo anterior dedicado a ficheros, pero con separador de campos y de registros.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Plantilla para generar HTML
-->
<xsl:stylesheet xmlns:xsl="http://
  www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"
    encoding="UTF-8"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>Lista de clientes
        </title>
      </head>
      <body>
        <xsl:apply-templates
          select="clientes"/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="clientes">
    <h1>Clientes</h1>
    <table border="1">
      <thead>
        <tr>
          <th>DNI</th>
          <th>Apellidos</th>
          <th>CP</th>
        </tr>
      </thead>
      <tbody>
        <xsl:apply-templates
          select="cliente"/>
      </tbody>
    </table>
  </xsl:template>
```



```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Plantilla para generar CSV
-->
<xsl:stylesheet xmlns:xsl="http://
  www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="text"
    encoding="UTF-8"/>
  <xsl:strip-space elements="*"/>
  <xsl:variable
    name='salto_linea'>
    <xsl:text>\n</xsl:text>
  </xsl:variable>
  <xsl:variable
    name='separador_campos'>
    <xsl:text>|</xsl:text>
  </xsl:variable>
  <xsl:template match="/">
    <xsl:apply-templates
      select="clientes"/>
  </xsl:template>
  <xsl:template match="clientes">
    <xsl:apply-templates
      select="cliente"/>
  </xsl:template>
  <xsl:template match="cliente">
    <xsl:if test="not(validez/#
      estado='borrado')">
      <xsl:value-of select="#
        DNI"/>
      <xsl:apply-templates/>
      <xsl:value-of
        select="$salto_linea"/>
    </xsl:if>
  </xsl:template>
```

```

<xsl:template
  match="cliente"><xsl:if
    test="not(validez/@
      estado='borrado')">
      <tr><td><xsl:value-of
        select="@DNI"/>
      </td>
      <xsl:apply-templates/>
      </tr>
    </xsl:if>
  </xsl:template>
  <xsl:template
    match="apellidos|CP">
    <td><xsl:apply-templates/>
    </td>
  </xsl:template>
</xsl:stylesheet>

```



```

<xsl:template match="apellidos">
  <xsl:value-of
    select="$separador_campos"/>
  <xsl:value-of
    select="substring(concat(., ,
      ' ', .), 1, 1)" />
</xsl:template>
<xsl:template match="CP">
  <xsl:value-of
    select="$separador_campos"/>
  <xsl:value-of
    select="substring(concat(., ,
      ' ', .), 1, 5)" />
</xsl:template>
</xsl:stylesheet>

```

Figura 6.9

Plantillas XSL para generar HTML y CSV a partir de documento de XML con datos de clientes

Algunas opciones de XSL controlan aspectos del documento generado. Por ejemplo, en `<xsl:output>` el atributo `method`, con valores `html` y `text` utilizados aquí (otro posible valor es `xml`). Para generar texto sin que se añadan separadores de relleno, se utiliza `<xsl:strip-space elements="*"/>`. Para generar texto, se utilizan dos variables: `separador_campo` y `salto_linea`. Esta última se define como `#xa;`, es decir, un `byte` con valor 10 (a en hexadecimal). Esto muestra que con XSL se pueden escribir directamente `bytes` y, por tanto, generar ficheros binarios. El resto está basado en el uso conjunto de `<xsl:template match="..."/>`, al que se le indica un XPath (se puede ver que se empieza con el nodo raíz `"/"`) y `<xsl:apply-templates select="..."/>`, que selecciona mediante XPath nodos sobre los que aplicar otras plantillas. Hay muchas más posibilidades que no se ilustran aquí, como por ejemplo la posibilidad de definir parámetros a los que se les pasa un valor en el momento de utilizar la plantilla. Esto se podría usar, por ejemplo, para especificar el separador de campos y de líneas al generar un documento en formato CSV.

Por último, un programa en Java que aplica las dos plantillas anteriores al documento de XML de ejemplo con datos de clientes para generar un fichero HTML y otro CSV. Utiliza la misma clase `Transformer` que se utilizó en un programa anterior, pero de una manera muy diferente. Esta es una clase muy versátil, pero en general relacionada con la transformación de documentos de XML. En el programa anterior se construía un `Transformer` sobre un documento DOM para serializarlo en un fichero. En este se construye un `Transformer` sobre una plantilla XSL para aplicarla a un documento de XML y generar un fichero.

```

// Plantillas XSL para generar HTML y CSV a partir de un documento de XML

package plantillasxsl;

import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import javax.xml.transform.Source;
import javax.xml.transform.Result;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

```

```

import javax.xml.transform.TransformerException;
public class PlantillasXSL {
    public static void main(String[] args) {
        String fFXML = "clientes.xml";
        String fXSLHTML = "clientes_HTML.xsl";
        String fXSLCSV = "clientes_CSV.xsl";
        String fHTML = "clientes.html";
        String fCSV = "clientes.csv";
        try {
            Source sXML = new StreamSource(fFXML); // Documento fuente XML
            Source sXSLHTML = new StreamSource(fXSLHTML); // Plantilla para
            // HTML
            Result osHTML = new StreamResult(new FileOutputStream(fHTML));
            Transformer tHTML = TransformerFactory.newInstance(),
            newTransformer(sXSLHTML);
            tHTML.transform(sXML, osHTML);

            Source sXSLCSV = new StreamSource(fXSLCSV); // Plantilla para CSV
            Result osCSV = new StreamResult(new FileOutputStream(fCSV));

            Transformer tCSV = TransformerFactory.newInstance(),
            newTransformer(sXSLCSV);
            tCSV.transform(sXML, osCSV);
        } catch (FileNotFoundException | TransformerException e) {
            System.err.println(e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Resumen

- XML es un lenguaje formal. XML es un estándar del W3C. Los documentos en lenguaje XML son documentos de texto con estructura jerárquica que se utilizan para guardar información de todo tipo.
- DOM es un modelo que define interfaces de programación para acceder a los contenidos de un documento de XML y para modificarlos. En el modelo DOM un documento de XML se representa como un árbol con distintos tipos de nodos.
- Un parser o analizador sintáctico para un lenguaje formal como XML permite verificar que un documento es conforme a la sintaxis del lenguaje y, si es el caso, proporciona acceso a sus contenidos.
- Un parser DOM permite obtener un árbol DOM a partir de un documento de XML.
- Un parser SAX es un tipo de parser para XML basado en eventos. No devuelve una estructura de datos como un parser DOM. En lugar de ello, genera diversos tipos de eventos conforme va analizando el documento, y estos eventos pueden gestionarse en métodos manejadores de eventos.

- La validación de un documento de XML consiste en verificar que es conforme a un conjunto de restricciones adicionales relativas a los elementos que contiene, sus atributos y los valores para todos ellos. Estas restricciones se pueden especificar en DTD y en esquemas de XML. Se puede validar un documento de XML durante el *parsing*, y en ese caso el parser lanzará excepciones ante cualquier error de validación.
- El *binding* es un mecanismo sencillo para traducir de documentos XML a objetos y viceversa, proporcionado por la API JAXB (Java Arquitecture for XML Binding). Para utilizarla es necesario un esquema de XML. A partir de este, un compilador de *binding* genera clases que se corresponden con los elementos del esquema de XML. En la terminología de JAXB, *marshalling* es serialización de objetos de Java en documentos de XML, y *unmarshalling* es deserialización de documentos de XML en objetos de Java.
- XPath es un lenguaje de consulta para documentos de XML. Se creó para XSL, y es fundamental para el lenguaje de consulta XQuery.
- XSL es un lenguaje que permite convertir documentos de XML a otros tipos, que pueden ser documentos XML de otros tipos, otros tipos de documentos de texto e incluso documentos binarios.



Ejercicios propuestos

Como en el capítulo anterior, para la realización de estos ejercicios puede ser necesario consultar la documentación de Java SE 8 (<http://docs.oracle.com/javase/8/docs/api/>).

1. Crea un programa en Java que, a partir de los contenidos del fichero XML como el del ejemplo, con los datos de varios clientes, muestre un nuevo documento de XML que sea el resultado de añadir los datos de un nuevo cliente al principio, antes de todos los clientes que puedan existir. Debe utilizarse un parser DOM y las interfaces de DOM para añadir los datos del nuevo cliente, incluyendo DNI, apellidos y CP. La salida debe ser a salida estándar (`System.out`) y en formato XML, y debe generarse utilizando la clase `Transformer`. El fichero se debe especificar mediante un argumento de línea de comandos. Si el fichero no es un fichero XML correcto, o si lo es pero no tiene un único elemento con nombres de clientes en el nivel más alto, se debe terminar la ejecución inmediatamente, y mostrar mensajes de error apropiados.
2. Cambia el programa anterior para que la salida se realice al mismo fichero, y reemplace sus contenidos. La salida se debe generar también con la clase `Transformer`. Antes de nada hay que hacer una copia del fichero en un fichero con el mismo nombre pero incluyendo una marca de tiempo y al final ".bak". Se aconseja generar la salida en un fichero temporal, borrar el fichero original y renombrar el fichero temporal con el nombre del fichero original. En los programas de ejemplo del capítulo anterior dedicado a ficheros se utiliza esta técnica. Se recomienda consultarlos y copiar y adaptar su código de programa. Si, como es lógico, no funciona a la primera, se tiene una copia del fichero original que se puede renombrar para intentarlo de nuevo.

3. Crea un programa como el anterior, pero que no permita añadir el nuevo cliente si en el fichero ya existe un cliente con su DNI. En ese caso, debe mostrarse un mensaje de error y terminarse inmediatamente la ejecución del programa. Además, el cliente debe añadirse al final, en lugar de al principio.
4. Crea un programa en Java que haga todo lo anterior, pero basándose en una clase `ClientesXML` para encapsular convenientemente todas las operaciones. El método `main()` debe crear una instancia de esa clase y utilizarla para todas las operaciones. Su constructor debe tomar como argumento un nombre de fichero y obtener el documento DOM en una variable de clase de tipo `Document`. El constructor no debe gestionar ninguna excepción, de manera que debe añadirse en su declaración una cláusula `throws` con todos los tipos de excepción que sea necesario incluir. La inserción de un nuevo cliente debe realizarse mediante un método público de clase `insertaCliente(String DNI, String apellidos, String CP)`, y debe insertarse al final, después de todos los clientes que puedan existir. Este método debe devolver un `Node` con el nuevo nodo creado, o `null` si por la razón que fuera no se crea el nodo. No debe reflejar los cambios en el fichero, solo en el `Document`. No debe gestionar las excepciones que se puedan producir al insertar los datos del nuevo cliente en el `Document`, de manera que hay que añadir la cláusula `throws` apropiada. Debe tener un método `grabar()` que permita reemplazar los contenidos del fichero con los contenidos del documento DOM. La generación del fichero (serialización del documento DOM) se debe realizar con la clase `Transformer`. Este método debe, antes que nada, hacer una copia del fichero en un fichero cuyo nombre esté basado en el nombre original del fichero y que incluya una marca de tiempo con la fecha y hora. El programa de prueba debe insertar un cliente con un DNI ya existente en el momento en que se intenta la inserción y dos clientes cuyos DNI no existan. Si el DNI ya existe, debe mostrarse un mensaje de error y seguir adelante. El método `main()` debe quedar más o menos así:

```
ClientesXML cXML = new ClientesXML(nombreFichero);
cXML.insertaCliente(dni1_nuevo, nombre1, CP1);
cXML.insertaCliente(dni2_repetido, nombre2);
cXML.insertaCliente(dni3_nuevo, nombre3, CP3);
cXML.grabar();
```

5. El formato tmx es un tipo de fichero XML que se utiliza para almacenar memorias de traducción. Una memoria de traducción es una colección de traducciones de un idioma a otro. Hay muchos ficheros de este tipo en el sitio <http://opus.nlpl.eu>. Utilizar un parser SAX para escribir las primeras quinientas entradas de la memoria de traducción disponible en <http://opus.nlpl.eu/download.php?f=EMEA/en-es.tmx.gz>. Este es un documento tmx, comprimido en formato zip, que hay que descargar y descomprimir. El principio de este fichero tiene el siguiente aspecto:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tmx version="1.4">
<header creationdate="Sun Oct 23 00:51:26 2011"
```

```

srcLang="en"
(...)
/>
<body>
  <tu>
    <tuv xml:lang="en"><seg>(texto en inglés)</seg></tuv>
    <tuv xml:lang="es"><seg>(texto en español)</seg></tuv>
  </tu>
  <tu>
    <tuv xml:lang="en"><seg>(texto en inglés)</seg></tuv>
    <tuv xml:lang="es"><seg>(texto en español)</seg></tuv>
  </tu>
...

```

El programa debe mostrar primero la lengua original y después los textos de cada segmento, y resaltar el que esté en la lengua original (en este caso, en inglés). La salida del programa debe ser similar a la siguiente:

```

Lengua original: en
---
[1]
*en=>(Texto segmento 1 en inglés)
es=>(Texto segmento 1 en español)
---
[2]
*en=>(Texto segmento 2 en inglés)
es=>(Texto segmento 2 en español)
---
[500]
*en=>(Texto segmento 500 en inglés)
es=>(Texto segmento 500 en español)

```

- Escribe una clase similar a la desarrollada en la actividad 4 anterior, pero que utilice JAXB para traducir los contenidos del fichero en objetos. No hay que utilizar DOM para nada, las modificaciones deben realizarse sobre objetos. Las operaciones de serialización y deserialización deben realizarse con los métodos que JAXB proporciona para *marshalling* y *unmarshalling*.
- Haz una copia del programa desarrollado en la actividad 3 anterior y modifícalo para que la comprobación de si existe algún cliente con el DNI proporcionado se haga mediante XPath, en lugar de recorriendo el árbol DOM. También se podría hacer esta mejora sobre una copia del programa desarrollado para la actividad 4.
- Crea un fichero tmx a partir del utilizado para el ejercicio 5 que contenga varias decenas de traducciones. Crea una plantilla XSL que genere un documento de HTML que las muestre en una tabla con dos columnas. En la cabecera de las columnas aparecerán los idiomas, y en cada fila aparecerá un texto en el idioma original y su traducción al otro idioma.

ACTIVIDADES DE AUTOEVALUACIÓN

1. Los datos en un documento de XML están estructurados de manera:
 - a) Secuencial, al tratarse de ficheros de texto.
 - b) Jerárquica, dado que pueden existir elementos dentro de otros elementos, y la información que contienen está asociada a ellos, como un texto o como valores para sus atributos.
 - c) Estricta, y de acuerdo a su esquema de XML asociado.
 - d) Ninguna de las respuestas anteriores es correcta.
2. Un documento de XML es válido si:
 - a) Está bien formado, es decir, es conforme a la sintaxis del lenguaje XML.
 - b) Está bien indentado, de manera que se pueda leer con claridad.
 - c) Es correcto sintácticamente y además cumple una serie de restricciones adicionales, con respecto a su estructura y contenidos, que se pueden especificar en un esquema de XML.
 - d) Todas las etiquetas de apertura tienen su correspondiente etiqueta de cierre.
3. El modelo DOM:
 - a) Es una especificación detallada de las estructuras de datos que hay que utilizar para representar en memoria un documento de XML utilizando un lenguaje orientado a objetos como Java.
 - b) Es la especificación formal de un tipo de parser para documentos de XML que permite un rápido acceso a sus contenidos y, una vez terminado el proceso de *parsing*, permite su modificación.
 - c) Especifica un conjunto de interfaces de programación para el acceso a los contenidos de documentos de XML y para su modificación.
 - d) Ninguna de las respuestas anteriores es correcta.
4. Un parser DOM:
 - a) En general es el más apropiado para documentos muy grandes porque permite un acceso muy rápido a sus contenidos.
 - b) Está basado en eventos.
 - c) No es apropiado para documentos muy grandes porque almacena en memoria todos los contenidos del documento.
 - d) Permite representar aspectos de un documento de XML que no contempla un parser SAX.
5. Un parser SAX:
 - a) Almacena por defecto los contenidos del documento en memoria. Una vez terminado el *parsing*, permite modificar los contenidos del documento y volverlo a grabar en un fichero con los cambios realizados.
 - b) No se puede utilizar para implementar un parser DOM porque el modelo de eventos de SAX es incompatible con el modelo de datos de DOM.
 - c) Está basado en eventos. Un programa que utilice un parser SAX debe proporcionar métodos manejadores para diversos tipos de eventos que pueden suceder durante el *parsing*.
 - d) Es un tipo de parser disponible solo para el lenguaje Java.

6. Un esquema de XML:
- a) No permite especificar restricciones que sí se pueden especificar con un DTD.
 - b) Es un documento de XML.
 - c) Es necesario para un documento para poder aplicarle una plantilla de XSL.
 - d) Utiliza expresiones de XPath para indicar qué elementos pueden ir dentro de otros.
7. En un documento de XML:
- a) Es obligatorio especificar un valor para todos los atributos, entre comillas, y no es admisible el valor “ ”.
 - b) En un mismo elemento no puede aparecer más de una vez el mismo nombre de atributo.
 - c) En un mismo elemento puede aparecer más de una vez el mismo nombre de atributo, siempre que no se le asigne más de una vez el mismo valor. Esto es útil para asignar una lista de valores a un atributo.
 - d) Se puede especificar un valor nulo para un atributo de la forma `atrib=null`.
8. Un JAXB es:
- a) Un mecanismo de *binding* que se puede aplicar con cualquier documento de XML, pero solo si es válido según el esquema utilizado para el *binding*. Además, impide cualquier modificación sobre los objetos que contravengan las restricciones impuestas por el esquema.
 - b) Un mecanismo que permite traducir de objetos a XML, pero no a la inversa.
 - c) Un mecanismo de *binding* que requiere un esquema de XML. Permite *marshalling* (serialización) a documentos de XML y *unmarshalling* (deserialización) desde documentos de XML.
 - d) Un mecanismo de *binding* que puede utilizarse con Java y otros lenguajes, siempre que sean orientados a objetos.
9. Una expresión XPath aplicada a un documento devuelve:
- a) Un conjunto de textos.
 - b) Un conjunto de nodos de un árbol DOM.
 - c) Un conjunto de nodos que pueden ser atributos también.
 - d) Ninguna de las opciones anteriores es correcta.
10. El lenguaje de transformación XSL:
- a) Solo permite generar documentos de XML.
 - b) Solo permite transformar documentos de XML.
 - c) Solo permite transformar documentos de XML válidos según algún esquema.
 - d) Ninguna de las respuestas anteriores es correcta.

SOLUCIONES:

1. a b c d
 2. a b c d
 3. a b c d
 4. a b c d

5. a b c d
 6. a b c d
 7. a b c d
 8. a b c d

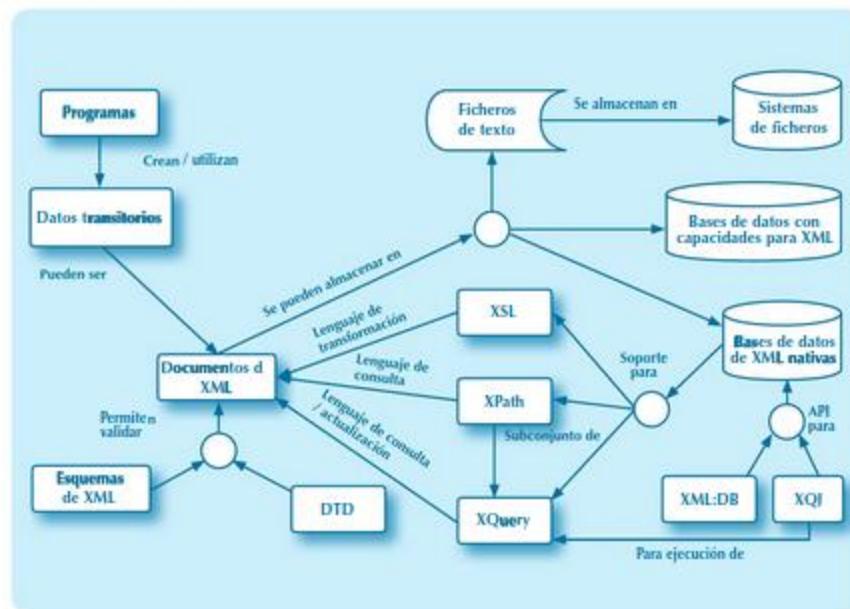
9. a b c d
 10. a b c d

Bases de datos de XML

Objetivos

- ✓ Conocer los distintos tipos de aplicaciones basadas en XML: *data-centric* y *document-centric*.
- ✓ Comprender el propósito y los fundamentos de las bases de datos de XML nativas.
- ✓ Entender las diversas formas en que distintas bases de datos de XML nativas permiten organizar los documentos en colecciones.
- ✓ Aprender los lenguajes estándares XPath y XQuery para consulta y actualización de documentos.
- ✓ Utilizar la base de datos de XML nativa eXist para almacenar, consultar y modificar documentos de XML.
- ✓ Estudiar las API estándares para interacción con bases de datos de XML: XML:DB y XQJ.
- ✓ Trabajar con XML:DB para gestionar colecciones y documentos.
- ✓ Realizar operaciones de consulta y modificación de documentos con XQJ.
- ✓ Utilizar transacciones con XQJ.

Mapa conceptual



Glosario

Base de datos nativa de XML. Base de datos para documentos de XML que, para su almacenamiento, utilizan estructuras diseñadas y optimizadas para este tipo de documentos.

XML:DB. API para bases de datos de XML que permite realizar todo tipo de operaciones.

XML. Lenguaje que permite representar cualquier tipo de información en una estructura jerárquica dentro de un documento de texto.

XPath. Lenguaje de consulta para documentos de XML que permite recuperar un conjunto de nodos de su estructura jerárquica.

XQJ (XQuery for Java). API para bases de datos de XML que permite realizar operaciones de consulta utilizando XQuery, y operaciones de modificación utilizando extensiones de XQuery.

XQuery. Lenguaje de consulta para documentos de XML, más potente que XPath, y del que XPath es un subconjunto. Se han propuesto varias extensiones de XPath que permiten modificar documentos de XML. Entre ellas está XQUF, que es un estándar de W3C.

7.1. XML como soporte para almacenamiento e intercambio de datos

XML es un formato muy sencillo y flexible que permite representar cualquier tipo de información en documentos de texto con una estructura jerárquica. Su uso se ha extendido cada vez más desde su creación como estándar de W3C en 1998, y en torno a él ha ido surgiendo un abundante cuerpo de estándares. Desde su introducción, el desarrollo de nuevos servicios y aplicaciones sobre internet se ha basado cada vez más en XML. XML se utiliza a menudo como soporte para almacenamiento de datos. Sus aplicaciones son cada vez más diversas, pero se pueden distinguir a grandes rasgos dos áreas de aplicación:

1. *XML centrado en documentos (document-centric XML)*. Corresponde al uso inicial de los lenguajes de marcado antecesores de XML, como HTML. Se utilizan documentos de XML que no tienen una estructura fija y regular, producidos mediante procesos manuales por personas para su utilización en procesos manuales realizados por personas. Suelen contener textos en lenguaje natural.
2. *XML centrado en datos (data-centric XML)*. Se utilizan documentos XML con una estructura bien definida y regular. Contienen datos. Son normalmente producidos por máquinas y consumidos por máquinas mediante procesos automáticos que utilizan lenguajes estándares como XML Schema, XPath, XQuery, XSL, etc.

Estos son dos casos extremos, y puede haber planteamientos intermedios. Por ejemplo: dentro de un documento de XML centrado en documento puede existir una sección con una estructura bien definida y regular, o bien dentro de un documento de XML centrado en datos puede haber una sección donde se admite un formato más flexible.

7.2. Alternativas para el almacenamiento de documentos de XML

Existen varias posibilidades para la persistencia de documentos de XML. Cada una puede ser más o menos apropiada dependiendo del tipo de documentos que almacenar y su uso previsto.

1. *Sistemas de ficheros*. Cada documento de XML se almacena como un fichero en una jerarquía de directorios. Para el manejo de los documentos se dispone solo de las funcionalidades que para el manejo de ficheros proporciona el sistema operativo, entre las que no suele estar el control de accesos concurrentes, ni tampoco el soporte para transacciones. Cualquier tipo de consulta o de modificación que se realice sobre el contenido de los documentos debe hacerse manualmente o programarse a medida.
2. *Bases de datos con capacidades para XML (XML-enabled)*. Pueden ser de distintos tipos, como por ejemplo relacionales u orientadas a objetos. Se necesitan mecanismos de traducción entre XML y el esquema de almacenamiento de la base de datos, que no estará en general optimizado para documentos de XML, lo que puede conllevar un mayor consumo de espacio de almacenamiento y un menor rendimiento en las operaciones realizadas sobre los documentos.
3. *Bases de datos nativas de XML*. El almacenamiento se realiza en estructuras diseñadas y optimizadas para documentos de XML. Proporcionan, además, soporte para lenguajes estándares que permiten realizar diversos tipos de operaciones sobre documentos de XML: consulta (XPath, XQuery), actualización (diversas extensiones de XQuery), y

transformación (XSL). Proporcionan implementaciones de API estándares para bases de datos de XML, tales como XML:DB y XQJ.

7.3. Almacenamiento de XML en SGBD relacionales

Es de especial interés plantear las distintas posibilidades para persistencia de documentos XML en bases de datos relacionales, por varios motivos:

1. Son, con diferencia, las más empleadas en la actualidad para todo tipo de aplicaciones. En principio es recomendable utilizarlas a no ser que el uso de una base de datos de XML nativa suponga una clara ventaja.
2. Pueden perfectamente almacenar documentos de XML, por grandes que sean, en columnas de tipo CLOB (*character large object*) o BLOB (*binary large object*). Si se trata de documentos muy pequeños (pocos kilobytes), podrían incluso almacenarse en columnas de tipo VARCHAR. Por tanto, son perfectamente válidas si lo único que se necesita es almacenar y recuperar documentos de XML enteros.
3. Es relativamente sencillo almacenar los datos que contienen los documentos de XML centrados en datos (*data-centric*) en esquemas relacionales, y generar los documentos a partir de ellos mediante procesos automáticos. Además, como se explica a continuación, las últimas versiones del estándar SQL incluyen características para XML que pueden facilitar mucho esta tarea.
4. El estándar SQL incluye, desde SQL:2003, SQL/XML en su parte 14. SQL/XML incluye, entre otras cosas:
 - a) El tipo de datos XML para almacenar documentos de XML.
 - b) Correspondencias entre tipos de datos de SQL y de XML para facilitar el almacenamiento y manipulación de XML en bases de datos con SQL.
 - c) Posibilidad de utilizar XQuery dentro de consultas en SQL. Como resultado de una consulta en XQuery, se puede obtener una relación, como con una consulta en SQL. Esto permite realizar consultas de SQL que combinen y relacionen información existente en esquemas relacionales y documentos de XML.
 - d) A la inversa, se pueden generar documentos de XML con el resultado de consultas en SQL. Esto permite realizar consultas en XQuery que combinen y relacionen información existente en documentos de XML y en esquemas relacionales.
5. Algunas bases de datos relacionales se pueden considerar *XML-enabled* porque implementan SQL/XML, lo que incluye soporte para XQuery, e incluso soporte para otros estándares de XML, como por ejemplo XSL.
6. Las bases de datos relacionales proporcionan, en general, un magnífico y completo soporte para transacciones, incluyendo en muchos casos transacciones distribuidas. No se puede decir tanto de las bases de datos de XML nativas, si bien es cierto que algunas bases de datos de XML nativas tienen soporte para transacciones.

La separación entre bases de datos relacionales y de XML nativas no es nítida. El estándar SQL incluye desde SQL:2003 muchas características para XML, y algunas bases de datos relacionales añaden soporte para otras tecnologías de XML como XSL, y la implementación de API estándar para bases de datos de XML. Si la funcionalidad y el rendimiento son satisfactorios, las bases de datos relacionales ofrecen una alternativa viable para el almacenamiento de XML.

rios, lo de menos es que el almacenamiento se realice, en última instancia, sobre un esquema relacional. Todas las versiones de la base de datos de Oracle desde la 12c incluyen XML DB.

XML DB incluye un buen soporte para SQL/XML y diversas tecnologías de XML.

En conclusión, antes de optar por el uso de una base de datos de XML nativa conviene considerar el uso de una base de datos relacional.

Recurso web



El siguiente enlace proporciona información acerca de Oracle XML DB, incluyendo código de ejemplo y documentación:

<https://www.oracle.com/database/technologies/appdev/xmldb.html>

7.4. Características de las bases de datos de XML nativas

Antes que nada, hay que aclarar que, en lo sucesivo, su utilizará el término *base de datos* (se sobreentiende de XML) con dos significados: el de almacenamiento de documentos de XML y el de SGBD (sistema gestor de bases de datos). El significado exacto será claro por el contexto. *Base de datos de XML nativa* significará siempre SGBD de XML nativo.

Las bases de datos de XML nativas deben permitir el almacenamiento directo de documentos de XML en estructuras diseñadas específicamente para XML, sean del tipo que sean: basados en contenido o en documentos, o cualquier otra posibilidad. Esto no excluye que puedan almacenar documentos que no sean de XML, si bien el conjunto de operaciones que se pueda realizar sobre ellos será más limitado. Por lo demás, suelen tener las siguientes características:

1. *Colecciones*. Dentro de una base de datos de XML existente en un SGBD de XML, se pueden organizar los documentos en colecciones. La manera de organizar los documentos en colecciones varía mucho de un SGBD de XML nativo a otro.
2. *Validación*. Se pueden usar esquemas o DTD para validar los documentos almacenados.
3. *Mecanismos estándares para consulta y transformación de documentos de XML*. A saber: XPath, XQuery y XSL (todos estándares de W3C). Pueden incluir la extensión XQuery Full Text, también estándar de W3C, para búsqueda en textos de lenguaje natural, tan frecuentes en los documentos de XML.
4. *Mecanismos estándares para modificación de documentos*. Debería ser posible modificar los contenidos de un documento sin reemplazarlo globalmente por otro. Han surgido diversos lenguajes para permitir esto, en general extensiones de XQuery.
5. *Soporte para API estándares*. Las dos API más importantes para la interacción con bases de datos de XML son XML:DB y XQJ. Pero cada base de datos de XML tendrá, en general, sus propias API, y las implementaciones de las API estándares se basarán en ellas.
6. *Indexación*. El uso de índices es fundamental para agilizar las consultas sobre los contenidos de los documentos de XML. Pero su indexación es mucho más problemática que la de las tablas de una base de datos relacional. Esto es debido a la mayor complejidad de los documentos de XML, con una estructura jerárquica en vez de tabular, ya que, en

una misma colección, puede haber documentos de distintos tipos. Se necesitan mecanismos de indexación muy flexibles y adaptativos. Las bases de datos de XML nativas suelen utilizar índices estructurales de todos los documentos, en los que se incluyen los valores de todos los elementos y todos los atributos. En ellos cada documento viene identificado por un identificador único asignado automáticamente por el sistema. Debido al mayor coste computacional de actualizar los índices, algunas bases de datos no lo hacen automáticamente tras cada cambio en los datos, lo que hay que tener muy en cuenta para el desarrollo de aplicaciones y para su administración. Las distintas bases de datos permiten crear distintos tipos de índices. La indexación de bases de datos de XML nativas es un aspecto poco o nada estandarizado.

7. *Transacciones*. Las diversas bases de datos de XML nativas proporcionan un cierto grado de soporte para transacciones, pero con frecuencia no ofrecen un completo soporte para transacciones ACID, como es la norma para las bases de datos relacionales. La API XQJ incluye métodos para gestión de transacciones, pero el soporte para transacciones es opcional para una implementación de XQJ. En cualquier caso, para que se puedan utilizar desde la API, el SGBD debe soportarlas.

7.5. Gestores comerciales y libres

Existen multitud de bases de datos de XML, tanto nativas como *XML-enabled*, y tanto comerciales como libres. Hay que tener en cuenta que, aunque XML se ha venido utilizando cada vez más, las bases de datos de XML nativas siempre han tenido, y siguen teniendo, un uso muy reducido. Se han desarrollado muchas desde finales de los años noventa, comerciales y libres, pero en pocas se invierte actualmente esfuerzo de soporte y desarrollo. En cambio, las bases de datos relacionales han ido cada vez más adoptando características para XML, incluso proporcionando implementaciones bastante completas de SQL/XML, parte del estándar SQL.



WWW

Recurso web

En la siguiente dirección se puede encontrar una descripción muy completa de las características más importantes de muchos productos de bases de datos de XML. Son de especial interés los enlaces titulados "Native XML Databases" y "XML-Enabled Databases":

<http://www.rpbourret.com/xml/XMLDatabaseProds.htm>

A continuación se reseñan brevemente varios SGBD de XML nativos comerciales y libres.

1. *Tamino*. Es un SGBD comercial de la empresa Software AG. Surgió durante el *boom* de XML a finales de los noventa. Es un producto muy caro y con muy buen rendimiento debido a su almacenamiento de datos diseñado específicamente para XML y a la indexación que realiza de los documentos. En el momento de la redacción de este libro, en la página web de Software AG no existe documentación ni información de producto para Tamino.

2. *Marklogic*. Este SGBD comercial comenzó como un SGBD de XML nativo a principios de los 2000 para evolucionar más recientemente hacia una base de datos multimodelo, es decir, que integra varios mecanismos de almacenamiento (en este caso, XML, JSON y tripletas semánticas de RDF), y ofrece una vista lógica unificada de todos los datos. Organiza los documentos en directorios y colecciones. Los directorios forman una jerarquía. Las colecciones agrupan documentos que tienen algo en común, como por ejemplo tema o autor. Un documento está en un directorio y puede estar en muchas colecciones. Existe una versión gratuita, descargable previo registro, para uso personal y no comercial. Se puede encontrar más información de Marklogic en su página web: <https://www.marklogic.com>
3. *eXist*. Es un SGBD de software libre. En una instalación de eXist solo existe una base de datos que contiene una jerarquía de colecciones, y dentro de cada una puede haber documentos de XML y de otros tipos. Asigna a cada documento un identificador único y crea y mantiene automáticamente índices estructurales para todos ellos. Se puede encontrar más información de eXist en su página web: <https://exist-db.org>.
4. *BaseX*. Es un SGBD de software libre. En una base de datos de BaseX no se crean las colecciones explícitamente. En lugar de ello, se crean y borran implícitamente, dependiendo de la existencia de documentos en *paths* específicos. Los documentos pueden ser de tipo XML o de tipo *raw* (que en este caso significa todo lo que no sea XML). Se puede encontrar más información de BaseX en su página web: <http://basex.org>
5. *Sedna*. Es un SGBD de software libre. Está escrito en C/C++, mientras que prácticamente todos los SGBD de XML nativos están escritos en Java. Es muy eficiente y está muy optimizado en todos los aspectos, como por ejemplo su esquema de almacenamiento para XML y su API propia basada en un protocolo propio binario, y no, como es habitual, en protocolos tales como SOAP, XML-RPC o similares. Fue un magnífico SGBD y un desarrollo muy prometedor en su momento, pero, de acuerdo con la información de su página web y de sourceforge.net, no parece haber actividad ni nuevos desarrollos desde 2012. Se puede encontrar más información de Sedna en su página web: <http://www.sedna.org>.

7.6. Instalación y configuración del SGBD de XML nativo eXist

Se ha elegido eXist como SGBD de XML nativo para trabajar por su facilidad de uso y funcionalidad. Como casi todas las bases de datos de XML nativas, está desarrollada en Java, por lo que funciona para Windows, Linux y cualquier sistema operativo para el que esté disponible una máquina virtual de Java.

Para instalarlo basta ejecutar el programa de instalación y aceptar todas las opciones por defecto. Se debería indicar la contraseña para el usuario administrador **admin**. Si no, se puede hacer más adelante. Con ello se crean opciones en el menú de inicio para lanzar y parar eXist, y para instalarlo o desinstalarlo como servicio del sistema.

Es recomendable instalarlo como servicio del sistema. Si no se hace, arrancará automáticamente, pero, como recuerda un mensaje que puede aparecer en ese momento, se pueden perder cambios realizados en la base de datos si se apaga el ordenador sin cerrar antes eXist.



Figura 7.1
Instalación de eXist-db como servicio de Windows

El *dashboard* (panel de control) de eXist está disponible en un servidor web que escucha por defecto por el puerto 8080.



TOMA NOTA

Hay diversos programas que suelen utilizar el puerto 8080, por ejemplo Tomcat y el servidor de base de datos Oracle. Si ya está en uso, se mostrará un error y se dará la opción de visualizar los logs, en los que aparecerá un mensaje similar al siguiente: **ERROR (JettyStart.java [run]:369) - ERROR: Could not bind to port because Address already in use: bind.** Se puede cambiar el puerto en el fichero `tools\jetty\etc\jetty-http.xml` en el directorio de instalación de eXist, en la línea donde dice `<SystemProperty name="jetty.port" default="8080"/>`. En esta instalación se ha cambiado a 8090, como se puede ver en algunos ejemplos siguientes.

Una vez iniciado eXist, se puede acceder al *dashboard* o panel de control en la dirección `http://localhost:puerto`. En él se pueden encontrar, entre otras cosas:

1. *Usermanager*. Para crear nuevos usuarios y cambiar sus propiedades. Si no se asignó contraseña para el usuario `admin` durante la instalación, se puede hacer aquí.
2. *Collections*. Para gestionar, crear y borrar colecciones. Una instalación de eXist gestiona una sola base de datos en la que pueden definirse colecciones dentro de otras colecciones, formando una jerarquía. En cada una puede haber documentos de XML y de otros tipos.
3. *eXide*. Magnífico IDE (entorno de desarrollo integrado) para XQuery. Permite navegar por los contenidos de la base de datos y visualizar los resultados de consultas con XQuery, y también ejecutar sentencias de XQuery que modifiquen los documentos. Permite modificar los contenidos de los documentos de XML mediante un editor de texto.
4. *Java Admin Client*. También se puede lanzar desde la bandeja del sistema. Además de consultas, permite realizar diversas tareas de gestión de los contenidos de la base de datos y administrativas, como por ejemplo:

- a) Importar ficheros y directorios seleccionados desde un sistema de ficheros.
- b) Gestionar colecciones.

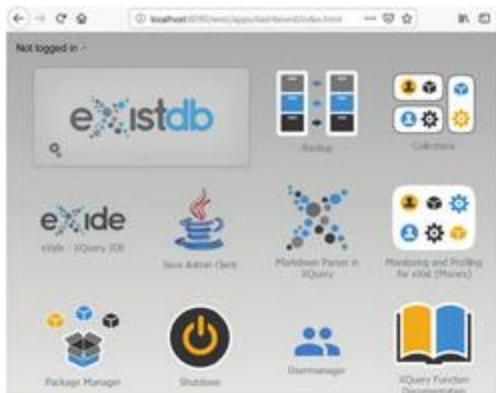


Figura 7.2
Dashboard (panel de control) de eXist

- c) Crear, borrar, mover y renombrar documentos.
 - d) Reindexar colecciones.
 - e) Gestión de usuarios, incluyendo sus permisos sobre colecciones y documentos.
 - f) Realizar y restaurar copias de seguridad, en modo de mantenimiento.
5. *Package Manager*. Permite instalar muchos paquetes y utilidades adicionales. Se recomienda instalar “eXist-db Demo Apps”. Con ello se instalará alguna base de datos de ejemplo que se utilizará después.
 6. *Shutdown*. Para parar el SGBD eXist. Si no se ha arrancado mediante un servicio, hay que pararlo antes de apagar el ordenador. En caso contrario podrían no guardarse todos los cambios realizados, como se ve en la figura 7.1.

7.7. API para gestión de bases de datos nativas de XML

Cada base de datos nativa de XML suele incluir sus propias API. Aparte de ello, para Java se han desarrollado las API estándares XML:DB y XQJ, cuyo planteamiento es similar al de JDBC para bases de datos relacionales.

- **XML:DB**. Fue la primera API estándar para bases de datos de XML. Su última revisión es de 2001.
- **XQJ**. Son las siglas de XQuery for Java. Es una API más reciente: su última revisión es de 2009. Ha sido diseñada como un proyecto JCP (*Java Community Process*), pero no está incluida en la biblioteca estándar de clases de Java.

Como su propio nombre indica, XQJ es una API para XQuery, es decir, para operaciones de consulta en documentos de XML existentes, y de modificación con diversas extensiones de XQuery. XML:DB permite utilizar XQuery y además hacer operaciones no posibles con XQuery, en particular las realizadas sobre colecciones y sobre documentos como un todo (creación y borrado), y además diversas tareas administrativas y de gestión. En el resto de este capítulo se aprenderá a realizar operaciones de creación y borrado de colecciones y documentos con XML:DB, y de consulta y modificación de documentos con XQJ.

TOMA NOTA



Es posible que las implementaciones de XML:DB o XQJ para algunas bases de datos no proporcionen determinadas funcionalidades que sí están disponibles en las API propias, o que estas últimas proporcionen mejor rendimiento para algunas operaciones. En ese caso, el uso de las API propias puede ser una alternativa que considerar, dependiendo de la aplicación.

7.8. La API XML:DB

No existe ninguna nueva versión oficial de esta API desde 2001. A pesar de ello, sigue siendo ampliamente utilizada internamente por muchos SGBD de XML nativos, como es el caso de eXist. Esto, unido al hecho de que permite gestionar diversos aspectos como las colecciones, la

creación y borrado de documentos, los usuarios y sus permisos, y otras cosas que varían mucho de una base de datos a otra, hace que pueda haber diferencias significativas entre implementaciones para distintas bases de datos.



Recurso web

Se puede acceder a los Javadoc de XML:DB en la siguiente dirección: <http://xmldb-org.sourceforge.net/xapi/api/index.html>.

Para ilustrar el funcionamiento de esta API, se utilizará para realizar algunos tipos de operaciones sobre la base de datos que no son posibles con XQuery. En particular:

- Operaciones sobre colecciones.
- Operaciones de creación y borrado de documentos de XML dentro de colecciones.

Para utilizar el *driver* de XML:DB para eXist hay que añadir al proyecto algunos ficheros jar disponibles en la instalación de eXist. Siempre hay que añadir **exist.jar** (presente en el directorio base de la instalación) y un fichero jar cuyo nombre comienza por **xmldb-db-api**, y otros presentes en el directorio **lib/core**. Según las funcionalidades que utilice el programa, podrá ser necesario añadir más ficheros jar para que se puedan cargar en tiempo de ejecución todas las clases necesarias. Si falta alguna, se producirá una excepción **ClassNotFoundException**. Entonces habrá que localizar el fichero jar que proporciona dicha clase y añadirlo al proyecto. En caso de dificultades, puede ser de ayuda algún buscador de ficheros jar que permita obtener el nombre del fichero jar a partir del nombre de la clase, como www.findjar.com

7.8.1. Establecimiento de conexiones y acceso a servicios con XML:DB

Para establecer una conexión con un SGBD con XML:DB, es necesario crear una instancia de una clase que implemente la interfaz **Database**, y proporcionar una cadena de conexión para la base de datos. El formato de la cadena de conexión varía según el SGBD. En la siguiente tabla se indica su formato para algunos SGBD de XML nativos. En ella se puede indicar el nombre de la base de datos y la colección, según el SGBD.

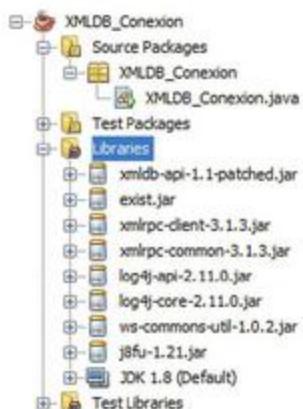


Figura 7.3
Proyecto que usa la API XML:DB

Cuadro

Propiedades para conexión para drivers de XML:DB para distintas bases de datos nativas de XML

SGBD	Clase que implementa Database	Cadena de conexión
eXist	org.exist.xmldb.DatabaseImpl	xmlDb:exist://host:puerto/exist/xmlrpc/db/colección
BaseX	org.basex.api.xmldb.BXDatabase	xmlDb:basex://localhost:puerto/basedatos
Sedna	net.cfoster.sedna.DatabaseImpl	xmlDb:sedna://host:puerto/basedatos/colección

Con eXist no se intenta abrir la conexión en el mismo momento en que se solicita, sino en el momento en que se intenta alguna operación que requiere acceso efectivo a la base de datos. Cualquier error en los datos de conexión pasará inadvertido hasta entonces. Además, si la conexión es a `localhost`, se ignora el puerto indicado y se toma de la configuración de eXist.

Una colección proporciona una serie de servicios, y cada uno de ellos puede proporcionar una serie de operaciones. El siguiente programa de ejemplo abre una conexión con una base de datos de eXist, y accede a una colección determinada dentro de ella, que siempre existe en una instalación típica, y muestra las colecciones y los documentos que hay en ella. También muestra una lista con los servicios proporcionados por la colección, y la versión de cada uno. Otros SGBD pueden proporcionar un conjunto distinto de servicios. Por ejemplo, la versión 3.5 de Sedna proporciona, además de los que ofrece eXist, los servicios `TransactionService`, `SednaUpdateService`, `XQueryService`, `IndexManagementService` y `ModuleManagementService`, cuyos nombres dan idea de su utilidad. Algunos de ellos están documentados en la especificación de XML:DB, y algunos de ellos no, al ser propios del SGBD Sedna.

```
// Muestra colecciones, documentos servicios disponibles en una colección
package XMLDB_Conexion;
import org.xmlDb.api.base.Collection
import org.xmlDb.api.base.Database
import org.xmlDb.api.base.Service
import org.xmlDb.api.base.XMLDBException
import org.xmlDb.api.DatabaseManager
public class XMLDB_Conexion
    private static Collection obtenColeccion(String nomCol) throws Exception
        Database dbDriver
        Collection col
        dbDriver = (Database)
        Class.forName("org.exist.xmldb.DatabaseImpl").newInstance()
        DatabaseManager.registerDatabase(dbDriver)
        col = DatabaseManager.getCollection(
            "xmlDb:exist://localhost:8090/exist/xmlrpc/db"   nomCol,
            "(usuario)", "(contraseña)")
        return col
    public static void main(String[] args)
        Collection col = null
```

```

try {
    col = obtenerColección("/apps/shared-resources");
    System.out.println("Colección actual: " + col.getName());
    int numHijas = col.getChildCollectionCount();
    System.out.println(numHijas + " colecciones hijas.");
    if (numHijas > 0) {
        String nomHijas[] = col.listChildCollections();
        for (int i = 0; i < numHijas; i++) {
            System.out.println("\t" + nomHijas[i]);
        }
    }
    int numDocs = col.getResourceCount();
    System.out.println(numDocs + " documentos.");
    if (numDocs > 0) {
        String nomDocs[] = col.listResources();
        for (int i = 0; i < numDocs; i++) {
            System.out.println("\t" + nomDocs[i]);
        }
    }
    Service servicios[] = col.getServices();
    System.out.println("Servicios proporcionados por colección " + col.
        getName() + ":");
    for (int i = 0; i < servicios.length; i++) {
        System.out.println("\t" + servicios[i].getName() + " - Versión:
            " + servicios[i].getVersion());
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (col != null) {
            col.close();
        }
    } catch (XMldbException e) {
        e.printStackTrace();
    }
}
}
}

```

7.8.2. Creación y borrado de colecciones con XML:DB

El siguiente programa crea dos colecciones en el nivel más alto de la jerarquía, y acto seguido borra una de ellas. Primero, obtiene un `CollectionManagementService` a partir de la colección raíz, y después, utiliza los métodos `createCollection` y `removeCollection` de este servicio para crear y eliminar colecciones dentro de ella. El método `obtenColección` en este programa y en los siguientes es igual que en el anterior y, por tanto, su código se omite.

```
// Creación y borrado de colecciones  
package XMLDB_CreacionColecciones;  
  
import org.xmldb.api.base.Collection;  
import org.xmldb.api.base.Database;
```

```

import org.xmldb.api.base.XMLDBException;
import org.xmldb.api.DatabaseManager;
import org.xmldb.api.modules.CollectionManagementService;
public class XMLDB_CreacionColecciones {
    private static Collection obtenColeccion(String nomCol) throws Exception {
        {...}
    }
    public static void main(String[] args) {
        Collection col = null;
        try {
            col = obtenColeccion("");
            System.out.println("Colección actual: " + col.getName());
            System.out.println(col.getChildCollectionCount()+" colecciones
                hijas antes.");
            CollectionManagementService cmServ =
                (CollectionManagementService) col.getService(
                    "CollectionManagementService", "1.0"
                );
            cmServ.createCollection("pruebal");
            cmServ.createCollection("pruebas");
            cmServ.removeCollection("pruebal");
            System.out.println(col.getChildCollectionCount()+" colecciones
                hijas después.");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (col != null) {
                    col.close();
                }
            } catch (XMLDBException xe) {
                xe.printStackTrace();
            }
        }
    }
}

```

7.8.3. Creación y borrado de documentos con XML:DB

El siguiente programa crea tres documentos en una colección creada por un programa anterior, y después borra uno de ellos. Los documentos se crean con `createResource` y se almacenan con `storeResource`. Para asignar y recuperar el contenido de un documento, la interfaz `XMLResource` tiene métodos `setContent` y `getContent`, y además `setContentAsDOM`, `getContentAsDOM`, `setContentAsSAX` y `getContentAsSAX`. eXist permite almacenar documentos que no son de XML (a los que denomina binarios). Para ello se utilizaría `BinaryResource` en lugar de `XMLResource`.

```

// Creación y borrado de documentos
package XMLDB_CreacionBorradoDocumentos;
import org.xmldb.api.base.*;
import org.xmldb.api.DatabaseManager;

```

```

import org.xmldb.api.modules.*;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.InputSource;
import org.xml.sax.ContentHandler;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;
import java.io.StringReader;
import java.io.StringWriter;
public class XMLDB_CreacionBorradoDocumentos {
    private static Collection obtenColeccion(String nomCol) throws Exception {
        (...)}
    public static void main(String[] args) {
        Collection col = null;
        try {
            col = obtenColeccion("/pruebas");
            XMLResource recurso;
            // Crear documento a partir de String
            recurso = (XMLResource) col.createResource("Clientes.xml",
                XMLResource.RESOURCE_TYPE);
            recurso.setContent("<clientes>\n"
                + "<cliente DNI="78901234X">\n"
                + "<apellidos>NADALES</apellidos><CP>44126</CP></cliente>\n"
                + "<cliente DNI="89012345E">\n"
                + "<apellidos>ROJAS</apellidos>\n"
                + "<validez estado="borrado" timestamp="1528286082"/>\n"
                + "</cliente>\n"
                + "<cliente DNI="56789012B">\n"
                + "<apellidos>SAMPER</apellidos><CP>29730</CP></cliente>\n"
                + "</clientes>");

            col.storeResource(recurso);
            recurso = (XMLResource) col.createResource("Empresa_.xml",
                XMLResource.RESOURCE_TYPE);
            recurso.setContent("<empresa CIF="A34246801">MegaExport"
                + "<sedes><sede>LEÓN</sede><sede>CÁCERES</sede></sedes>\n"
                + "</empresa>");

            col.storeResource(recurso);
            // Crear documento a partir de objeto SAX
            StringWriter out = new StringWriter();
            XMLOutputFactory xof = XMLOutputFactory.newInstance();
            XMLStreamWriter xsw;
            xsw = xof.createXMLStreamWriter(out);
            xsw.writeStartDocument();
            xsw.writeStartElement("empresa");
            xsw.writeAttribute("CIF", "A34246801");
            xsw.writeCharacters("MegaExport");
            xsw.writeStartElement("sedes");
            xsw.writeStartElement("sede");
            xsw.writeCharacters("LEÓN");
            xsw.writeEndElement();
            xsw.writeStartElement("sede");
            xsw.writeCharacters("CÁCERES");
            xsw.writeEndElement();
            xsw.writeEndElement();
            xsw.writeEndElement();
        }
    }
}

```

```

        xsw.writeEndDocument();
        xsw.flush();
        xsw.close();
        recurso = (XMLResource) col.createResource("DatosEmpresa",
            XMLResource.RESOURCE_TYPE);
        ContentHandler ch = recurso.setContentAsSAX();
        XMLReader reader = XMLReaderFactory.createXMLReader();
        reader.setContentHandler(ch);
        reader.parse(
            new InputSource(new StringReader(out.toString())));
        col.storeResource(recurso);
        col.removeResource(col.getResource("Empress_.xml"));
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (col != null) {
                col.close();
            }
        } catch (XMLDBException xe) {
            xe.printStackTrace();
        }
    }
}

```



Actividad propuesta 7.1

Crea un nuevo documento, en la misma colección que el anterior programa, utilizando `setContentAsDOM`. Primero hay que crear un documento DOM, de la manera en que se vio en el capítulo anterior dedicado a XML. Puedes copiar el código de un programa de ejemplo de este capítulo para crear el documento. Si es necesario, consulta los Javadocs de la API XML:DB.

7.9. El lenguaje XQuery

XQuery es un lenguaje estándar de W3C que se creó para realizar consultas sobre documentos de XML, lo mismo que su antecesor XPath. XPath se explicó en el capítulo dedicado a XML, que incluye numerosas consultas de ejemplo en XPath. XQuery 1.0 se definió de manera que incluye XPath 2.0. Es decir, cualquier sentencia de XPath 2.0 lo es de XQuery 1.0 y devuelve los mismos resultados. Lo mismo para XQuery 3.1 respecto de XPath 3.1.

Recurso web

WYU

Se puede encontrar información de XQuery en el sitio web de W3C:

<http://www.w3.org/TR/xquery>

Pronto se planteó la necesidad de disponer de un lenguaje para realizar operaciones de modificación sobre documentos de XML. Para ello surgieron diversas propuestas, como extensiones de XQuery, que se citan a continuación en orden cronológico:

- XUpdate. De 2000. Promovido por el grupo que desarrolló la API XML:DB.
- XQuery Update Extension. De 2001. Promovido por Patrick Lethi.
- XQuery Update Facility (XQUF). De 2011. Es un estándar de W3C.

Estas propuestas no solo tienen nombres muy parecidos, sino que también son similares entre sí. Si la base de datos proporciona soporte para ella, es preferible utilizar la última, porque es un estándar de W3C.



Debido a la frecuencia de las operaciones de búsqueda por texto en documentos de XML, y a que los documentos de XML contienen muy a menudo lenguaje natural (es decir, texto redactado por humanos para ser leído por humanos), se desarrolló, como estándar de W3C, una extensión de XQuery para facilitar consultas sobre este tipo de textos: XQuery Full Text. Se puede encontrar una concisa descripción de sus posibilidades en la siguiente dirección:

<http://docs.basex.org/wiki/Full-Text>.

7.9.1. Consultas con XQuery

En este apartado se explica XQuery con unas cuantas consultas de ejemplo. Se recomienda ejecutarlas con eXide (eXist IDE o entorno de desarrollo integrado de eXist). eXide permite navegar por los contenidos de la base de datos, ejecutar consultas de XQuery y ver los resultados. Tiene características muy útiles, como verificación y autocompletado.

Una consulta de XQuery se puede ejecutar sobre todos los documentos de una colección, como se ve en la siguiente consulta, realizada sobre una colección de ejemplo, disponible si se ha instalado “eXist-db Demo Apps”. Cada uno de los resultados se obtiene de un documento diferente.

Figura 7.4

Consulta en XQuery sobre todos los documentos de una colección, ejecutada en eXide

Las siguientes consultas de ejemplo se harán en documentos de la colección `pruebas`, creada por un programa de ejemplo anterior. Se harán en general en `Clients.xml`, pero al-

guna de ellas también en `productos.xml`, que debe crearse previamente con eXide. Se puede enlazar la información de ambos documentos utilizando el DNI.

Clientes.xml	<pre><clientes> <cliente DNI="78901234X"> <apellidos>NADALES</apellidos> <CP>44126</CP> </cliente> <cliente DNI="89012345E"> <apellidos>ROJAS</apellidos> <validez estado="borrado" timestamp="1528286082"/> </cliente> <cliente DNI="56789012B"> <apellidos>SAMPER</apellidos> <CP>29730</CP> </cliente> </clientes></pre>
productos.xml	<pre><productos> <producto nombre="tuerca"> <precio>0.25</precio> <prov id="78901234X"/> </producto> <producto nombre="tornillo"> <precio>0.10</precio> <prov id="89012345E"/> <prov id="78901234X"/> </producto> </productos></pre>

Las sentencias de XQuery son de tipo FLWOR (*for, let, where, order by, return*). La analogía con las cláusulas de una consulta de SQL (*select, from, where, group by, having, order by*) es evidente.

CUADRO 7.2
Cláusulas de una sentencia FLWOR de XQuery

FOR	Vincula variables a expresiones en XPath. Estas expresiones pueden devolver múltiples resultados, cada una de las cuales se tiene en cuenta para el resto de las cláusulas.
LET	Permite asignar a una variable el resultado de evaluar una expresión en XPath, y se suele utilizar para evitar repetirla más de una vez. Si la expresión devuelve más de un resultado, se concatenan todos para asignar el resultado a la variable.
WHERE	Especifica condiciones que permiten filtrar los resultados de las cláusulas FOR y LET.
ORDER BY	Permite ordenar los resultados de las cláusulas FOR y LET.
RETURN	Especifica una expresión que se evalúa para cada uno de los resultados proporcionados por el conjunto de cláusulas anteriores. En esta cláusula se pueden generar directamente XML y, en ese caso, se pueden incluir sentencias de XQuery entre llaves {}, lo que resulta un mecanismo muy potente.

Actividad propuesta 7.2



Utilizando eXide, crea en la colección `pruebas` el documento `productos.xml` con los contenidos indicados.

A continuación se explican varias sentencias de XQuery de ejemplo. Se sugiere ejecutarlas en eXide y experimentar con variaciones sobre ellas.

<pre>for \$n in doc('/db/pruebas/Clientes.xml') return \$n/clientes/cliente</pre>	Devuelve los datos de todos los clientes. Cada uno de los resultados es un elemento de nombre <code>cliente</code> del documento.
<pre>for \$n in doc('/db/pruebas/Clientes.xml') return \$n/clientes/cliente/apellidos/text()</pre>	Se utiliza el paso <code>text()</code> para obtener no los elementos, sino el texto dentro.
<pre>for \$n in doc('/db/pruebas/Clientes.xml')/ clientes/cliente where substring(\$n/CP,1,2) = "29" return concat(\$n/apellidos, "-", \$n/string(@DNI))</pre>	Se utiliza la cláusula WHERE para seleccionar clientes de Málaga. Se concatena un XPath a la especificación del documento <code>doc(...)</code> . Se utiliza la función <code>string()</code> para obtener el valor de un atributo. Se utiliza la función <code>concat()</code> para concatenar cadenas.
<pre>for \$n in doc('/db/pruebas/Clientes.xml')/ clientes/cliente order by number(\$n/CP) return <cli dni="{\$n/string(@DNI)}" ape="{\$n/apellidos}"> {\$n/CP} </cli></pre>	Se utiliza la cláusula ORDER BY con la función <code>number()</code> para ordenar numéricamente y no alfabéticamente. Se devuelve XML, del que algunas partes se obtienen con XQuery entre llaves <code>{}</code> . Los resultados son XML, no texto, y su representación textual puede diferir de lo que aparece literalmente en la sentencia de XQuery, como es el caso de un cliente sin el elemento <code>CP</code> , cuyos datos se muestran como <code><cli dni="89012345E" ape = "ROJAS"/></code> .
<pre><clientes> { for \$n in doc('/db/pruebas/Clientes.xml')/clientes/cliente return <cliente> <ident tipo="dni">{\$n/string(@DNI)}</ident> <apell>{\$n/apellidos/text()}</apell> </cliente> } </clientes></pre>	Esta consulta muestra cómo con XQuery se puede incluir una sentencia FLWOR dentro de un documento de XML, y cómo XQuery permite generar documentos de XML con total flexibilidad, utilizando XPath para obtener todos los datos que sean necesarios en cualquier lugar en que sean necesarios.

<pre>let \$cl:=doc('/db/pruebas/clientes.xml')/clientes/ cliente return <nuestrosclientes>{\$cl}</nuestrosclientes></pre>	<p>Uso de cláusula LET, no es el más habitual. El XPath devuelve varios resultados, pero se concatenan y esta consulta devuelve un único resultado.</p>
<pre>for \$cli in doc('/db/pruebas/clientes.xml')/ clientes/cliente let \$id_cli:=\$cli/string(@DNI) return <prod_cli dni="{\$id_cli}" ape="{\$cli/apellidos}"> { for \$prod in doc('/db/pruebas/productos.xml')/ productos/producto/prov[@id=\$id_cli] return <prod> { \$prod/string(@nombre) } </prod> } </prod_cli></pre>	<p>Uso de cláusula LET en una consulta sobre dos documentos. Se utiliza LET para guardar el DNI y en la cláusula RETURN se hace una nueva consulta para obtener todos los productos suministrados al cliente.</p>
<pre>for \$n in doc('/db/pruebas/Clientes.xml')/ clientes/cliente order by number(\$n/CP) return element cli { attribute dni { \$n/string(@DNI) }, data (\$n/apellidos), element cp {\$n/CP/text()} }</pre>	<p>El uso de constructores para elementos, atributos y textos de XML permite obtener un código de XQuery más escueto, aunque más alejado de la sintaxis de XML.</p>

Por lo demás, estos ejemplos solo dan una idea de las posibilidades de XQuery. XQuery tiene muchas funciones predefinidas y se pueden definir nuevas. XQuery tiene una sentencia condicional `if ... then ... else`. Con XQuery se dispone de mucha flexibilidad para añadir más de una vez las anteriores cláusulas y no necesariamente en ese orden. Existe una cláusula GROUP BY similar a la del mismo nombre en SQL, y funciones de grupo `count`, `sum`, `min`, `max`, etc.

Recurso web



El siguiente tutorial de la empresa Altova explora todas estas posibilidades:

<https://www.altova.com/training/xquery3>

7.9.2. Sentencias de modificación de datos con XQuery

En este apartado se explican, con algunos ejemplos, las sentencias de modificación de datos disponibles para eXist, pertenecientes a lo que en su documentación se denomina *XQuery Update*.

Extension. Estos ejemplos modifican el documento `Cientes.xml` de la colección `pruebas`, y se recomienda ejecutarlos con eXide. Otras bases de datos pueden utilizar distintas extensiones de XQuery, pero el planteamiento y la sintaxis son similares.

<pre>update insert <cliente DNI="67890123C"> <apellidos> GAMBOA</apellidos> <CP>52351</CP> </cliente> into doc('/db/pruebas/Cientes.xml')/clientes</pre>	<p>Inserta datos de un nuevo cliente. Los datos se insertan como último elemento bajo el elemento <code>clientes</code> indicado.</p>
<pre>update insert <cliente DNI="23456789D"> <apellidos>DOLCE</apellidos> <CP>11895</CP> </cliente> following doc('/db/pruebas/Cientes.xml')/clientes/ cliente[@DNI="78901234X"]</pre>	<p>Inserta datos de un nuevo cliente después de los de otro cliente especificado. Si no se especifica la posición de inserción, esta se realiza por defecto. Existe una cláusula <code>preceding</code> para realizar la inserción después de una posición determinada.</p>
<pre>update value doc('/db/pruebas/Cientes. xml')/clientes/cliente[@DNI="23456789D"]/ apellidos with "DORCE"</pre>	<p>Cambia el valor en elemento <code>apellidos</code> para el cliente antes insertado.</p>
<pre>update replace doc('/db/pruebas/Cientes. xml')/clientes/cliente[@DNI="23456789D"] with <cliente DNI="12345678Z"> <apellidos>ARCOS</apellidos> </cliente></pre>	<p>Cambia el elemento indicado por un nuevo elemento. Nótese la diferencia de esta sentencia con la anterior. En la anterior solo se reemplazaba el texto dentro de un elemento, en esta se reemplaza el elemento entero.</p>
<pre>update rename doc('/db/pruebas/Cientes. xml')//validez as "valid"</pre>	<p>Cambia el nombre de elementos con nombre <code>validez</code> y les asigna el nombre <code>valid</code>.</p>
<pre>for \$cli in doc('/db/pruebas/Cientes.xml')/ clientes/cliente/valid[@estado="borrado"] return update delete \$cli</pre>	<p>Borra clientes bajo los que existe un elemento <code>valid</code> con un atributo <code>estado</code> con valor <code>"borrado"</code>. Las sentencias que hacen cualquier cambio deben ejecutarse con mucha precaución. Igual que antes de ejecutar una sentencia que modifica los datos con SQL es conveniente ejecutar un <code>SELECT</code> con la misma cláusula <code>WHERE</code> para asegurarse de a qué datos va a afectar, con XQuery es conveniente ejecutar antes una consulta. En este caso, la consulta consistiría en eliminar <code>update delete</code> en la anterior sentencia.</p>

7.10. La API XQJ

XQJ es a XQuery y las bases de datos de XML lo que JDBC a SQL y las relacionales. Si JDBC permite ejecutar sentencias de SQL desde programas en Java, XQJ hace lo propio con XQuery.

y sus extensiones. La API XQJ proporciona una serie de interfaces que los *drivers* para diferentes bases de datos deben implementar. XQJ no forma parte de la biblioteca estándar de clases de Java.

Para operaciones de consulta, el planteamiento de XQJ (y de XML:DB) es similar al de JDBC para bases de datos relacionales. Para operaciones de consulta con JDBC, se ejecuta una sentencia SELECT de SQL, y se obtiene un **ResultSet**, del que se consiguen los resultados mediante un iterador. Con XQJ se hace algo similar, pero las consultas se hacen con XQuery.

Las operaciones de actualización son más problemáticas porque, como ya se ha visto anteriormente, se han propuesto varios lenguajes para este tipo de operaciones, si bien todos son similares y se han planteado como extensiones de XQuery.

Recursos web



En la siguiente dirección están disponibles los Javadoc de XQJ:

<http://xqj.net/javadoc>

En las siguientes direcciones se pueden descargar *drivers* de XQJ para eXist, BaseX, Sedna y Marklogic. El fichero de la descarga es un fichero comprimido que contiene varios ficheros jar que hay que incluir en el proyecto. Entre ellos hay al menos uno con la API XQJ y otro con el *driver* para la base de datos que contiene clases que implementan las interfaces de XQJ:

<http://xqj.net/exist>

<http://xqj.net/basex>

<http://xqj.net/sedna>

<http://xqj.net/marklogic>

En cada una de las páginas anteriores hay a la derecha un enlace para descargar el *driver*. Hay también un enlace: Compliance Definition Statement. Como explica allí al principio, para la conformidad con el estándar XQJ, se requiere una declaración de cómo se han implementado todos los aspectos de la especificación XQJ que en ella se describen como dependientes de la implementación. Entre ellos está el soporte para transacciones.



Actividad propuesta 7.3

¿Cuáles de los drivers XQJ anteriores proporcionan soporte para transacciones?

7.10.1. Establecimiento de conexiones con XQJ

Para crear un programa en Java que utilice XQJ para trabajar con una base de datos, deben añadirse al proyecto los ficheros jar incluidos en la descarga del *driver* para la base de datos.

En la figura se muestra un ejemplo con la base de datos eXist. El fichero `xqjapi.jar` contiene la API XQJ. Para establecer una conexión debe, primero, crearse una instancia de la clase que implementa la interfaz `XQDataSource`. En este caso es `ExistXQDataSource`, en `exist-xqj-1.0.1.jar`, que contiene la implementación de XQJ para eXist. Después, se debe proporcionar un valor para los parámetros de conexión necesarios para la base de datos en particular, con `setProperty`. El nombre de la clase y los parámetros de conexión son aspectos dependientes de la implementación y, por tanto, están documentados en la Compliance Definition Statement del driver. Se indican en el cuadro 7.3 para varias bases de datos. No es siempre necesario proporcionar valor para todas las propiedades. La propiedad `description`, por ejemplo, es siempre opcional. En algunos casos, si no se indica un puerto, se puede asumir uno por defecto. Para eXist, por ejemplo, si se indica `localhost` para `serverName`, no es necesario indicar el puerto. Por último, al igual que con XML:DB, con eXist nunca se producirá ningún error ni excepción al abrir la conexión a la base de datos desde el programa, porque esta no se establece hasta que no se realiza alguna operación que requiere el acceso efectivo a la base de datos.

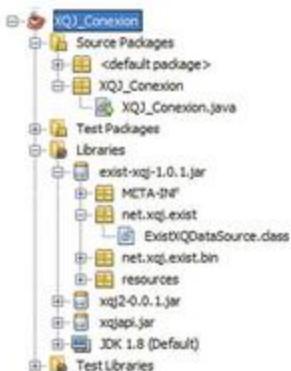


Figura 7.5
Proyecto que usa la API XQJ

CUADRO 7.3

Cadenas de conexión para drivers de XQJ para distintas bases de datos nativas de XML

SGBD	Clase que implementa XQDataSource	Propiedades para conexión
MarkLogic	<code>net.xqj.marklogic.</code> <code>MarkLogicXQDataSource</code>	<code>serverName</code> , <code>databaseName</code> , <code>port</code> , <code>user</code> , <code>password</code> , <code>description</code> , <code>mode</code>
eXist	<code>net.xqj.exist.</code> <code>ExistXQDataSource</code>	<code>serverName</code> , <code>port</code> , <code>user</code> , <code>password</code> , <code>description</code>
BaseX	<code>net.xqj.basex.</code> <code>BaseXXQDataSource</code>	<code>serverName</code> , <code>databaseName</code> , <code>port</code> , <code>user</code> , <code>password</code> , <code>description</code>
Sedna	<code>net.xqj.sedna.</code> <code>SednaXQDataSource</code>	<code>serverName</code> , <code>databaseName</code> , <code>port</code> , <code>user</code> , <code>password</code> , <code>description</code>

En el cuadro 7.3 se indica la clase que implementa la interfaz `XQDataSource` y las propiedades disponibles para la conexión. No es obligatorio proporcionar un valor para todas.

El siguiente programa establece una conexión con una base de datos eXist, indica si la conexión proporciona soporte para transacciones, y cierra la conexión. Para establecer la conexión, se ha creado el método `obtenConexion`, y para mostrar información específica de una excepción de tipo `XQException`, el método `muestraErrorXQuery`. Para averiguar si hay soporte para transacciones, se consultan los metadatos de la conexión, obtenidos con `getMeta-`

Data. Para que este programa funcione para otra base de datos, hay que asignar a nomClaseDS el nombre de la clase que implementa la interfaz XQDataSource para ella.

```
// Abre conexión con XQJ
package ConexionXQJ;

import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQMetaDataSet;
public class ConexionXQJ {

    private static String nomClaseDS = "net.xqj.exist.ExistXQDataSource";
    private static XQConnection obtenConexion() throws ClassNotFoundException,
    InstantiationException, IllegalAccessException, XQException {
        XQDataSource xqs=(XQDataSource) Class.forName(nomClaseDS).
            newInstance();
        xqs.setProperty("serverName", "localhost");
        xqs.setProperty("port", (puerto));
        xqs.setProperty("user", (usuario));
        xqs.setProperty("password", (contraseña));
        return xqs.getConnection();
    }
    private static void muestraErrorXQuery(XQException e) {
        System.err.println("XQuery ERROR mensaje: " + e.getMessage());
        System.err.println("XQuery ERROR causa: " + e.getCause());
        System.err.println("XQuery ERROR código: " + e.getVendorCode());
    }
    public static void main(String[] args) {
        XQConnection c = null;
        try {
            c = obtenConexion();
            XQMetaDataSet xqmd = c.getMetaData();
            System.out.println("Conexión establecida como:" + xqmd.
                getUserName() + ".");
            System.out.println(
                "Transacciones: " + (xqmd.isTransactionSupported() ? "Sí":
                "No") + ".\n");
        } catch (XQException e) {
            muestraErrorXQuery(e);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (c != null) {
                    c.close();
                }
            } catch (XQException xe) {
                xe.printStackTrace();
            }
        }
    }
}
```

7.10.2. Consultas con XQJ

La forma de realizar consultas con XQJ es similar a JDBC. A partir de una conexión `XQConnection` se obtiene una expresión de XQuery `XQExpression` sobre la que se ejecuta una consulta con `executeQuery(String query)`, y se obtiene un conjunto de resultados `XQResultSequence`, análogo a un `ResultSet` de JDBC. Sobre este se itera utilizando el método `next()`. Los resultados se pueden obtener en forma de `XMLStreamReader` con `getItemAsStream()`, o en forma de `Node` del modelo DOM con `getNode()`.

El siguiente programa de ejemplo recupera los apellidos de todos los clientes presentes en el documento `Cièntes.xml` de la colección `/db/pruebas`.

```
// Consulta con XQJ
package XQJ_Consulta;

import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQResultSequence;
import javax.xml.xquery.XQExpression;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.XMLStreamConstants;

public class XQJ_consulta {

    private static String nomClaseDS = "net.xqj.exist.ExistXQDataSource";
    private static XQConnection obtenConexion() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException, XQException {
        (...)

    }
    private static void muestraErrorXQuery(XQException e) {
        (...)

    }
    public static void main(String[] args) {
        XQConnection c = null;
        try {
            c = obtenConexion();
            String cad = "doc('/db/pruebas/Cièntes.xml')/cièntes/ciènte/
                apellidos";
            XQExpression xqe = c.createExpression();
            XQResultSequence xqrs = xqe.executeQuery(cad);
            int i=1;
            while (xqrs.next()) {
                System.out.println("[Resultado "+(i++)+"]");
                XMLStreamReader xsr = xqrs.getItemAsStream();
                while (xsr.hasNext()) {
                    if (xsr.getEventType() == XMLStreamConstants.CHARACTERS) {
                        System.out.println(xsr.getText());
                    }
                }
                xsr.next();
            }
        } catch (XQException e) {
            muestraErrorXQuery(e);
        }
    }
}
```

```
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (c != null) {
                c.close();
            }
        } catch (XQException xe) {
            xe.printStackTrace();
        }
    }
}
```



Actividad propuesta 7.4

Modifica el programa anterior para obtener los resultados de la consulta como nodos del modelo DOM mediante el método `getNode()`. Escribe los resultados en forma de árbol utilizando el método `mostrarNodo(Node nodo, int nivel, PrintStream ps)` de un programa de ejemplo del capítulo anterior dedicado a XML. Si es necesario, consulta los Javadoc de la API XQJ.

7.10.3. Modificaciones de documentos con XQJ

La ejecución de sentencias para modificación se hace también con una **XQExpression**, pero con el método **executeCommand**. El siguiente programa de ejemplo inserta un nuevo elemento con los datos de un nuevo cliente, en la posición anterior (cláusula **preceding**) a una determinada, indicada mediante un XPath.

```

// Operaciones de modificación con XQJ

package XQJ_Modificacion;

import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;

public class XQJ_Modificacion {

    private static String nomClaseDS = "net.xqj.exist.ExistXQDataSource";
    private static XQConnection obtenConexion() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException, XQException {
        ...
    }
    private static void muestraErrorXQuery(XQException e) {
        ...
    }
}

```

```
public static void main(String[] args) {
    XQConnection c = null;
    try {
        c = obtenerConexion();
        String cad = "update insert "
            + "<cliente DNI='09876543K'>"
            + "<apellidos>LAMQUIZ</apellidos>"
            + "<CP>43001</CP>"
            + "</cliente>"
            + " preceding doc('db/pruebas/Clientes.xml')/clientes/cliente/"
            + " apellidos[text()='SAMPER']/..";
        XQEExpression xqe = c.createExpression();
        xqe.executeCommand(cad);
    } catch (XQException e) {
        muestraErrorXQuery(e);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (c != null) {
                c.close();
            }
        } catch (XQException e) {
            e.printStackTrace();
        }
    }
}
```

Actividades propuestas



- 7.5.** Escribe un programa que cambie el nombre del cliente añadido por el programa de ejemplo anterior. La sentencia utilizada debe localizar el cliente que se va a borrar mediante su DNI. Precaución: escribe una consulta en XQuery para recuperar el nombre de este cliente y después transfórmala en una sentencia que lo cambie.

7.6. Escribe un programa que borre el cliente añadido por el programa de ejemplo anterior. La sentencia utilizada debe localizar el cliente que se va a borrar mediante su DNI. Precaución: escribe una consulta en XQuery para recuperar la información de este cliente y después transfórmala en una sentencia que la borre.

7.19.4. Transacciones con XQJ

Una transacción es un conjunto de operaciones que se ejecutan como un todo, y que cumplen los requisitos ACID (atomicidad, consistencia, aislamiento y durabilidad). El concepto de transacción se explicó en detalle en el capítulo anterior dedicado a bases de datos relacionales. Para que en una aplicación se puedan utilizar transacciones, debe proporcionar soporte para ellas el sistema operativo.

SGBD, la API y el *driver*. Ya se ha comentado que, al contrario de lo que sucede con los SGBD relacionales, solo algunos SGBD de XML nativos proporcionan soporte para transacciones, y eXist no es uno de ellos. XQJ sí permite utilizar transacciones siempre que el SGBD y el *driver* lo permitan, y además de manera muy similar a JDBC.

Al igual que en JDBC, en XQJ, por defecto, los cambios realizados por cada operación realizada sobre una conexión se confirman en la base de datos al finalizar la operación. Este comportamiento se puede cambiar con `setAutoCommit(false)` si hay soporte para transacciones, y entonces solo se confirman los cambios cuando se ejecuta `commit()`. Se pueden deshacer todos los cambios realizados en el curso de una transacción ejecutando `rollback()`. De acuerdo con la especificación XQJ, si se hace `setAutoCommit(false)` sobre una conexión para la que no hay soporte de transacciones, la implementación (léase el *driver*) debe lanzar un error. Si se produce alguna excepción durante la ejecución de una transacción, esta debe capturarse y debe ejecutarse el método `rollback` para deshacer los cambios. Todo esto se refleja en el siguiente código de programa para implementar una transacción:

```
// Transacción con XQJ

package XQJ_Transaccion;

import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;

public class XQJ_Transaccion {
    // Sedna soporta transacciones.
    private static String nomClaseDS = "net.cfoster.sedna.DatabaseImpl";

    private static XQConnection obtenConexion() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException, XQException {
        (...)

    }

    private static void muestraErrorXQuery(XQException e) {
        (...)

    }

    public static void main(String[] args) {
        XQConnection c = null;
        try {
            c = obtenConexion();
            c.setAutoCommit(false);
            XQExpression xqe = c.createExpression();
            // Ahora se pueden ejecutar múltiples sentencias de modificación,
            // entre las que se pueden intercalar sentencias de consulta con
            // executeQuery
            xqe.executeCommand("update... ");
            xqe.executeCommand("update... ");
            (...)

            c.commit();
        } catch (XQException e) {
            muestraErrorXQuery(e);
            try {
                c.rollback();
                System.err.println("Se hace ROLLBACK");
            }
        }
    }
}
```

```
        } catch (XQException er) {
            System.err.println("ERROR haciendo ROLLBACK");
            muestraErrorXQuery(er);
        }
    } catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (c != null) {
            c.close();
        }
    } catch (XQException e) {
        e.printStackTrace();
    }
}
}
```

Resumen

- XML es un formato muy flexible para el almacenamiento e intercambio de información. Se pueden distinguir, a grandes rasgos, dos áreas de aplicación de XML: XML centrado en documentos (*document-centric XML*) y XML centrado en datos (*data-centric XML*).
- Las bases de datos de XML nativas almacenan los documentos de XML en estructuras diseñadas y optimizadas para documentos de XML, y además proporcionan soporte para tecnologías de XML tales como XPath, XQuery y XSL.
- Existen también bases de datos no nativas de XML pero con capacidades para XML (*XML-enabled*). Un ejemplo de ellas son las bases de datos relacionales que implementan SQL/XML, añadido al estándar SQL en SQL:2003. Pueden ser la mejor opción para aplicaciones de XML basadas en datos (*data-centric XML*).
- Las bases de datos de XML nativas suelen organizar los documentos en colecciones. Algunas permiten definir una jerarquía de colecciones, y algunas permiten almacenar otros tipos de documentos además de documentos de XML.
- Las bases de datos de XML nativas implementan mecanismos de indexación estructural, en los que se tienen en cuenta los valores almacenados en todos los elementos y atributos.
- El principal lenguaje de consulta para bases de datos de XML es XQuery, un estándar de W3C, que es un superconjunto de XPath, otro estándar anterior de W3C. Se han propuesto varias extensiones de XQuery para permitir operaciones de modificación sobre el contenido de los documentos de XML; la última es XQUF, también un estándar de W3C.
- Las API estándares existentes para bases de datos nativas de XML son XML:DB y XQJ. XQJ está concebida para ejecutar consultas de XQuery y también sentencias de modificación de documentos de XML expresadas en alguna de las extensiones de XQuery propuestas para ello, entre ellas el estándar XQUF de W3C.
- Algunas bases de datos nativas de XML proporcionan un buen soporte para transacciones, pero este no es el caso en general, muy al contrario que con las bases de datos relacionales.

Ejercicios propuestos



Todos los documentos que se creen para estas actividades deben crearse en una colección con nombre `ejercicios_bd_XML`, a no ser que se indique otra cosa.

1. Crea un programa genérico al que se le pasen por línea de comandos el nombre de una colección, el nombre de un documento y una consulta en XQuery. El programa ejecutará la consulta y devolverá una lista de resultados. Si no se indica un nombre de documento, la consulta debe realizarse sobre todos los documentos de la colección. Los resultados se deben mostrar en formato XML. Si cada resultado es un nodo de un árbol DOM, debe mostrarse el fragmento del documento por debajo del nodo obtenido. Prueba este programa con las consultas de ejemplo de este capítulo.
2. Crea un programa genérico al que se le pasen por línea de comandos el nombre de una colección, el nombre de un documento y una sentencia de modificación de XQuery para ejecutar sobre el documento en cuestión. El programa debe crear una copia del documento original en la misma colección, en cuyo nombre debe incluirse el nombre del documento original y una marca de tiempo incluyendo la fecha y la hora. Una vez ejecutada la consulta, deben mostrarse los contenidos tanto del documento original (es decir, de la copia que se ha hecho previamente) como del documento resultante tras ejecutar la sentencia. Debe pedirse confirmación al usuario para confirmar los cambios. Si la da, debe borrarse la copia creada. Si no, debe borrarse el documento y renombrar la copia, para que todo quede como al principio. Si se produce cualquier excepción, todo debe quedar como al principio. Prueba este programa con las sentencias de modificación de datos de este capítulo.
3. Crea una clase con varios métodos para consultar datos de un cliente, crear un cliente, borrar un cliente y modificar datos de un cliente (al menos, nombre, código postal y DNI). Los datos de los clientes se deben almacenar en un documento de XML con un formato como el del documento de ejemplo utilizado durante este capítulo. A los métodos que realizan cualquier cambio sobre un cliente se les debe indicar el cliente mediante el DNI. Debe haber un método para obtener el número de clientes almacenados y otro para mostrar todos los datos de todos los clientes, que se puede limitar a escribir los contenidos del documento. El constructor de la clase debe crear el documento para almacenar los datos de los clientes, en caso de que no exista, conteniendo solo un elemento con nombre `clientes`. Debe crearse un programa de prueba en el método `main()` que realice varias operaciones de creación, borrado y modificación de clientes, y por último muestre todos los datos de todos los clientes.
4. Crea una clase con la misma interfaz que la anterior, pero que internamente funcione con un esquema de almacenamiento diferente. A saber, para cada cliente debe haber un documento de XML con sus datos, y todos ellos deben estar en una misma colección, que inicialmente estará vacía. No debería ser necesario que, tras cambiar el DNI de un cliente, sigan coincidiendo el valor del DNI almacenado y el nombre del fichero, pero, opcionalmente, se puede hacer que se cambie el nombre del documento al modificar el DNI. En su defecto, se puede crear un método estático de la clase para renombrar correctamente todos los documentos para que su nombre coincida con el del DNI almacenado. El método `main()` debe ser el mismo que para la anterior actividad.

5. Crea una aplicación que gestione los datos de los empleados de una empresa almacenados en un fichero XML. El fichero debe tener la estructura siguiente:

```
<empresa cif="..." nombre="...">
    <departamento código="..." nombre="...">
        <empleado dni="..." nombre="...">
            <puesto>...</puesto>
        </empleado>
        <empleado dni="..." nombre="...">
        ...
        <empleado dni="..." nombre="...">
            <puesto>...</puesto>
        </empleado>
    </departamento>
    ...
    <departamento código="..." nombre="...">
        <empleado dni="..." nombre="...">
        ...
        </empleado>
    ...
    <empleado dni="..." nombre="...">
    ...
    </empleado>
</departamento>
</empresa>
```

En resumen, dentro de la empresa hay departamentos y dentro de cada departamento, empleados. No hace falta hacer validaciones sobre el formato de ningún dato. No puede haber dos departamentos distintos con el mismo código, ni dos empleados distintos con el mismo DNI, y ambos atributos son obligatorios. También los atributos **cif** y **nombre** del elemento **empresa**. El elemento **puesto** es opcional. Un empleado no puede estar en más de un departamento.

Debe crearse una clase **GestorEmpresa** que gestione los datos del documento, almacenado en una base de datos de eXist. El documento debe estar en una colección con nombre **empresa_y_empleados**. El constructor debe tener como parámetros el CIF y el nombre de la empresa, y debe crear la colección si no existe, y el documento si no existe, con el CIF y el nombre de la empresa. El método **main()** de la clase contendrá un programa de prueba que utilice esta clase para crear varios departamentos, añadir varios empleados en distintos departamentos y cambiar algunos empleados de departamento, utilizando los métodos que se indican más adelante. Los métodos no deben gestionar ninguna excepción, esto debe hacerlo el programa principal en el método **main()**. Esta clase debe tener métodos para:

- Crear un departamento, dado su código y nombre. Si ya existe un departamento con ese nombre, se debe mostrar un mensaje de error y devolver **false**. En otro caso, se debe devolver **true**.
- Crear un empleado, dado su DNI y nombre y el código del departamento al que pertenece. Si ya existe un empleado con ese DNI, se debe mostrar un mensaje de error y devolver **false**. También si no existe el departamento. En otro caso, se debe devolver **true**.

- Eliminar un empleado dado su DNI. Debe devolver `false` y mostrar un mensaje de error si el empleado no existe. En otro caso, debe eliminar el empleado y devolver `true`.
- Eliminar un departamento dado su código. Debe devolver `false` y mostrar un mensaje de error si el departamento no existe o si tiene algún empleado. En otro caso, debe eliminar el departamento y devolver `true`.
- Cambiar un empleado de departamento, dado el DNI del empleado y el código del nuevo departamento. Debe devolver `true` si se puede hacer la operación. En caso contrario, deben mostrarse mensajes de error y devolver `false`. A saber, si el empleado o el departamento no existen, o si el empleado ya está en el departamento.

El método `main()` podría quedar más o menos así:

```
try {
    GestorEmpresa gEmpresa = new GestorEmpresa(nombreFich);
    if(!gEmpresa.creaDepartamento(codDep1,nomDep1)) return;
    if(!gEmpresa.creaEmpleado(dniEmp1,nomEmp1,codDep1)) return;
    if(!gEmpresa.creaEmpleado(dniEmp2,nomEmp2,codDep1)) return;
    if(!gEmpresa.creaDepartamento(codDep2,nomDep2)) return;
    if(!gEmpresa.creaEmpleado(dniEmp3,nomEmp3,codDep2)) return;
    System.out.println("Cambios realizados correctamente");
}
catch(...) {
...
}
catch(...) {
...
}
```

ACTIVIDADES DE AUTOEVALUACIÓN

1. XML es un formato de documento:
 - a) Con una estructura fija y regular.
 - b) Que tiene dos variantes: una para almacenar datos y otra para textos en lenguaje natural.
 - c) Muy sencillo y flexible, pero con el que se pueden también crear documentos con una estructura fija y regular.
 - d) Con una estructura que puede ser jerárquica o tabular.
2. El almacenamiento de documentos de XML en bases de datos relacionales:
 - a) Es siempre una mala alternativa, al tener los documentos de XML una estructura jerárquica que no se presta al almacenamiento en estructuras tabulares.

- b) Es la única manera de poder disponer de transacciones en aplicaciones que trabajan con datos en formato XML.
- c) Solo es posible si se almacenan los documentos como un todo en campos de tipo BLOB, CLOB o VARCHAR.
- d) Ninguna de las respuestas anteriores es correcta.
3. Las bases de datos de XML nativas:
- a) Son todas de código abierto, al ser XML un estándar de W3C.
- b) Son las únicas que permiten utilizar tecnologías de XML tales como XML Schema, XPath, XQuery y XSL, sobre los documentos de XML almacenados.
- c) Suelen organizar los documentos en colecciones.
- d) Solo permiten almacenar documentos en formato XML.
4. Las colecciones en las bases de datos de XML nativas:
- a) Son jerárquicas, es decir, puede haber colecciones dentro de colecciones.
- b) Deben siempre crearse explícitamente antes de añadir documentos a ellas.
- c) Deben contener siempre documentos de igual tipo para toda la colección.
- d) Ninguna de las respuestas anteriores es correcta.
5. SQL/XML:
- a) Es un lenguaje de consulta para bases de datos de XML inspirado en SQL.
- b) Es una extensión para XML incluida en todas las versiones de Oracle desde la 12c.
- c) Es una parte del estándar SQL que añade soporte para XML.
- d) Es un SGBD comercial multimodelo, a la vez relacional y de XML nativo.
6. Los índices de las bases de datos de XML nativas:
- a) Son más costosos de mantener que los de las bases de datos relacionales.
- b) No son actualizados automáticamente en algunos SGBD de XML nativos.
- c) Son estructurales, basados en los valores de todos los elementos y atributos.
- d) Todas las respuestas anteriores son correctas.
7. Las transacciones en los SGBD de XML nativos:
- a) No siempre están soportadas, pero si no lo están, se pueden conseguir con un driver de XQJ, porque el soporte para transacciones es obligatorio en XQJ.
- b) No están soportadas por muchos de ellos, al contrario de lo que sucede con los SGBD relacionales.
- c) Están en general soportadas, pero, como son muy costosas computacionalmente, suelen estar deshabilitadas por defecto.
- d) Solo están soportadas por la API XQJ, pero no por XML:DB.
8. El lenguaje XQuery:
- a) Es un subconjunto del lenguaje XPath.
- b) Permite realizar consultas sobre varios documentos de XML, siempre que estén en la misma colección.
- c) Obliga a incluir las cláusulas de una sentencia FLWOR en un orden fijo, al igual que sucede con las cláusulas de una sentencia SELECT de SQL.

- d) Permite la anidación de consultas FLWOR, es decir, el uso de una consulta FLWOR dentro de una cláusula de una consulta FLWOR.
9. La API XML:DB:
- a) No permite realizar consultas con XQuery.
 - b) Está obsoleta y se utiliza cada vez menos, en especial para nuevos desarrollos.
 - c) No proporciona servicios para transacciones.
 - d) Permite realizar tareas administrativas y de gestión que XQJ no permite.
10. La API XQJ:
- a) No permite realizar operaciones de modificación sobre documentos de XML.
 - b) Es parte de las bibliotecas estándares de Java (Java standard libraries).
 - c) Permite obtener los resultados de una consulta como nodos de DOM.
 - d) Permite borrar documentos enteros de una colección.

SOLUCIONES:

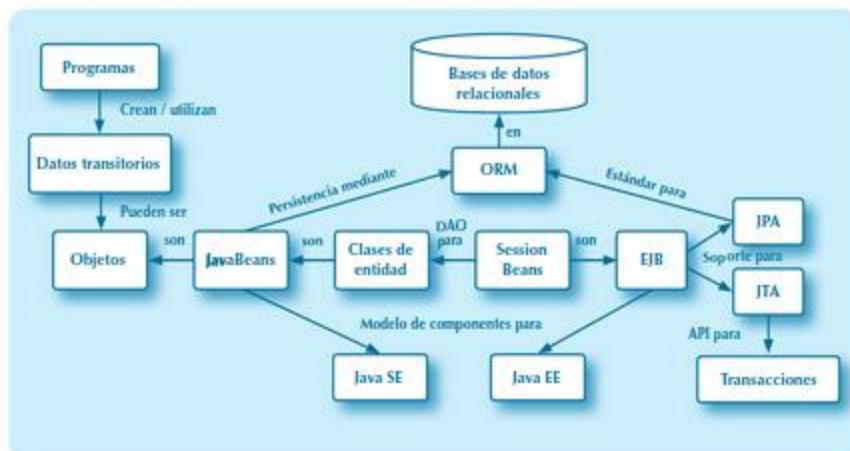
1. a b c d2. a b c d3. a b c d4. a b c d5. a b c d6. a b c d7. a b c d8. a b c d9. a b c d10. a b c d

Componentes para el acceso a datos

Objetivos

- ✓ Comprender qué es un componente de software y su importancia en la ingeniería del software.
- ✓ Entender el modelo de componentes para Java SE, basado en JavaBeans.
- ✓ Conocer los elementos fundamentales para desarrollar aplicaciones web en la plataforma Java: *servlets* y *JSP*.
- ✓ Comprender el modelo MVC y la manera en que se pueden desarrollar con Java aplicaciones web según este modelo.
- ✓ Desarrollar aplicaciones web sencillas en la plataforma Java SE según el modelo MVC y que utilicen JavaBeans para el modelo y ORM para persistencia de objetos.
- ✓ Estudiar el modelo de componentes para Java EE basado en EJB.
- ✓ Trabajar con aplicaciones web sencillas en la plataforma Java EE que hagan uso de EJB para el modelo, de su soporte integrado de JPA para persistencia de objetos y de JTA para transacciones.

Mapa conceptual



Glosario

Aplicación web. Aplicación a la que se accede mediante el protocolo HTTP o HTTPS, por lo que se puede utilizar a través un navegador web.

Componente de software. Componente independiente que proporciona una funcionalidad o unos servicios determinados, accesibles mediante una interfaz pública.

EJB (Enterprise JavaBeans). Componente de software para la plataforma Java EE que se despliega sobre un contenedor de EJB, incluido en un servidor de aplicaciones para Java EE.

Java EE. Plataforma de computación con el mismo planteamiento que Java SE, pero con una biblioteca de clases más amplia que incluye muchas pensadas para aplicaciones empresariales, y la especificación de un servidor de aplicaciones, que a su vez incluye un contenedor de servlets y un contenedor de EJB.

Java SE. Plataforma de computación sobre la que se pueden ejecutar programas en Java, constituida por una máquina virtual de Java más una biblioteca de clases estándares.

JavaBean. Componente de software para la plataforma Java SE.

JSP (JavaServer Pages). Tecnología que permite generar documentos en formato HTML de manera dinámica, utilizando el lenguaje de programación Java.

Modelo de componentes. Modelo que define las características de un tipo de componentes y, en su caso, de los contenedores sobre los que se pueden desplegar, así como los servicios proporcionados por los contenedores a los componentes.

Modelo MVC. Del inglés *model, view, controller*, es decir, modelo, vista, controlador.

Modelo que estructura las aplicaciones en tres partes diferenciadas y que interactúan entre sí: el *modelo*, formado por un conjunto de objetos que constituyen un modelo de los datos relevantes para la aplicación; la *vista*, que muestra el estado del *modelo* de una manera significativa para los usuarios; y el *controlador*, que responde a las acciones realizadas por los usuarios sobre la *vista*, realiza los cambios apropiados sobre el *modelo* y actualiza la *vista* para que refleje los cambios realizados sobre el *modelo*.

Servidor de aplicaciones. Servidor que proporciona determinados servicios a aplicaciones. También puede proporcionar contenedores para determinados modelos de componentes.

Servlet. Objeto que se despliega en un contenedor de *servlets* de la plataforma Java EE y con el que se puede interactuar mediante el protocolo HTTP o HTTPS.

8.1. Componentes de software

Los sistemas informáticos tienen una enorme complejidad, tanto en lo que respecta al *hardware* como al *software*. La modularización facilita el desarrollo de sistemas complejos. Un sistema modular no se desarrolla como un bloque monolítico, sino que integra un conjunto de módulos o componentes. Un componente queda perfectamente especificado por lo que hace (su funcionalidad) y por cómo interactúa con el mundo exterior (sus interfaces). Esto permite su reutilización en diversos sistemas. El diseñador de un sistema que utiliza componentes solo tiene que tener en cuenta sus especificaciones, es decir, su funcionalidad y sus interfaces, para integrarlo en el sistema. La posibilidad de diseñar y desarrollar un sistema basándose en componentes reutilizables es especialmente importante en la ingeniería del *software*. Los componentes se pueden desarrollar y probar de manera independiente, y se distribuyen como paquetes independientes que se integran en el sistema tal y como vienen, aunque requieran, posiblemente, una configuración inicial. Esto reduce la complejidad global del sistema y contribuye a mejorar la calidad y reducir el tiempo de desarrollo.

El planteamiento modular para el desarrollo de sistemas complejos se viene utilizando de manera general, desde hace mucho tiempo, para sistemas electrónicos, y en particular para *hardware* de ordenadores. A una placa base, por ejemplo, se pueden conectar diversos componentes o módulos, como memoria, microprocesador, dispositivos de almacenamiento y muchos otros. Pero es mucho más reciente y está menos generalizado para sistemas de *software*.

8.2. Modelos de componentes

Un modelo de componentes define unos requisitos formales para los componentes, así como mecanismos para la interacción con ellos. Una plataforma de componentes es una infraestructura de *software* sobre la que se pueden desplegar los componentes y que proporciona servicios básicos para ellos.

Los primeros modelos de componentes se crearon para componentes visuales. Microsoft fue pionera con OLE y COM. Pronto surgieron otros modelos de componentes, como los JavaBeans para la plataforma Java. Los IDE (entornos integrados de desarrollo) incluyeron soporte para su uso y desarrollo. Más tarde, se empezaron a utilizar componentes no visuales, como por ejemplo componentes para el acceso a datos (DAO o *data access objects*).

Algunos modelos de componentes son:

- .NET de Microsoft. Es el último paso en la evolución desde OLE, pasando por COM, VBX, OCX, ActiveX, COM+ y DCOM.
- JavaBeans para Java SE (Java Standard Edition).
- EJB para Java EE (Java Enterprise Edition).
- CORBA (Common Object Request Broker Architecture), un estándar definido por OMG (Object Management Group).

8.3. La plataforma Java: Java SE y Java EE

Java es un lenguaje de programación en torno al cual se ha desarrollado una plataforma de computación: la plataforma Java. Existen dos versiones de esta plataforma: Java SE y Java EE, y un modelo de componentes para cada una.

Java es un lenguaje compilado, pero con la peculiaridad de que no se compila para ningún hardware ni sistema operativo en particular, sino a bytecode, que puede ser directamente ejecutado por una máquina virtual de Java (o JVM, siglas en inglés de *Java virtual machine*). Los programas de Java, una vez compilados a bytecode, se pueden ejecutar sobre cualquier sistema operativo, siempre que para él exista una máquina virtual de Java. La máquina virtual de Java forma, junto con una biblioteca de clases estándares, el JRE (*Java runtime environment* o entorno de ejecución de Java). Hay dos versiones del JRE:

- Java SE. Incluye una biblioteca de clases de uso general. El modelo de componentes para Java SE está basado en JavaBeans.
- Java EE. Incluye una biblioteca de clases ampliada para dar soporte a aplicaciones empresariales. Incluye la especificación de un servidor de aplicaciones con diversos tipos de contenedores para desplegar diversos tipos de elementos, a saber: contenedores de *servlets* y contenedores de EJB. El modelo de componentes para Java EE está basado en EJB.

Java SE y Java EE son especificaciones. El lenguaje de programación Java, la máquina virtual de Java, Java SE y Java EE fueron desarrollados originalmente por Sun, pero sus especificaciones son públicas. El desarrollo de la plataforma Java ha estado siempre a cargo de una comunidad liderada, primero, por Sun y, después, por Oracle, que han desarrollado siempre implementaciones de referencia para cada nueva versión, cuyo código fuente se ha hecho público. De esta manera, cualquier empresa o institución puede desarrollar y vender su propia implementación. Esto es lo que han hecho empresas como BEA, IBM y otras, que han desarrollado y vendido servidores de aplicaciones conformes a la especificación Java EE.

Oracle siempre ha mantenido Java EE como un proceso de la comunidad Java (JCP o *Java community process*). En septiembre de 2017, Oracle anunció que cedería Java EE a la fundación Eclipse. Pero Oracle no ha cedido el nombre Java EE, por lo que su nuevo nombre se cambió a EE4J (Eclipse Enterprise for Java). En febrero de 2018 se anunció que el nuevo nombre sería Jakarta EE. Aquí se le seguirá llamando Java EE.

8.4. JavaBeans

El modelo de componentes de Java SE se basa en los JavaBeans. Estos se empezaron a utilizar como componentes visuales para interfaces gráficas de usuario. Más adelante se empezaron a utilizar JavaBeans no visuales con otras finalidades, como por ejemplo el acceso a datos.

RECUERDA

- ✓ No hay que confundir los JavaBeans con los Enterprise JavaBeans o EJB, que se verán posteriormente.

Los JavaBeans son clases de Java que cumplen una serie de requisitos formales. La mayoría de ellos son los que se exigían a las clases de Java (POJO) para poder ser clases persistentes con Hibernate. De hecho, estos POJO son, en general, JavaBeans, aunque los JavaBeans pueden tener algunas características adicionales relacionadas con la gestión de eventos. Los requisitos que tienen que cumplir los JavaBeans son:

1. Un constructor sin argumentos, que no tiene por qué ser el único constructor.
2. Implementar la interfaz `Serializable`, para lo que basta con incluir en su definición la opción `implements Serializable`.
3. Propiedades definidas como privadas (`private`) y accesibles mediante métodos públicos `getter` y `setter`. El prefijo del nombre de un método `getter` debe ser `get`, a menos que sea de tipo `boolean`, en cuyo caso puede ser `is`. Las propiedades pueden ser:
 - a) *Simples*: constan de un único valor. Su método `getter` es de la forma `Tipo getX()`, y el `setter` de la forma `void setX(Tipo valor)`, donde `Tipo` es el tipo de la propiedad y `X` su nombre pero con la inicial en mayúsculas.
 - b) *Indexadas*: constan de un conjunto de valores. Su métodos `getter` y `setter` tienen un parámetro de tipo `int`, a saber: `Tipo getX(int indice)`, y `void setX(int indice, Tipo valor)`.
 - c) *Compartidas*: son propiedades de cuyos cambios de valor se notifica a otros objetos. Para mantener la lista de objetos a los que notificar del cambio en el valor de una propiedad compartida, se deben implementar los métodos `addPropertyChangeListener` para añadir un objeto a la lista, y `removePropertyChangeListener` para eliminar un objeto de la lista. Para ello se utiliza normalmente un objeto auxiliar de la clase `PropertyChangeSupport`. Los JavaBeans receptores deben implementar el método `propertyChange` de la interfaz `PropertyChangeListener`. Las notificaciones de cambios en el valor de la propiedad se hacen con los métodos `firePropertyChange` o `fireIndexedPropertyChange` de dicho objeto auxiliar.
 - d) *Restringidas*: son similares a las compartidas, pero los objetos notificados pueden vetar el cambio de valor. Para mantener la lista de objetos notificados se implementan `addVetoableChangeListener` y `removeVetoableChangeListener`. Para ello se utiliza normalmente un objeto auxiliar de la clase `VetoableChangeSupport`. Los JavaBeans receptores deben implementar el método `vetoableChange` de la

interfaz `vetoableChangeListener`. Las notificaciones de cambios en el valor de la propiedad se hacen llamando al método `fireVetoableChange` de dicho objeto auxiliar. El objeto receptor de la notificación puede lanzar una excepción de la clase `PropertyVetoException`, en cuyo caso el objeto emisor capturaría la excepción y restauraría el antiguo valor.

Para gestionar los JavaBeans se utilizan los siguientes mecanismos:

1. *Reflexión e introspección.* Java SE incluye una API de reflexión que permite a un programa en Java obtener información detallada acerca de cualquier clase, incluyendo sus atributos y métodos. También una API de introspección que permite obtener información acerca de los JavaBeans utilizando la reflexión e interpretando la información obtenida teniendo en cuenta los convenios de nomenclatura utilizados en los JavaBeans. Se puede proporcionar directamente información acerca de un JavaBean con una clase cuyo nombre sea el del JavaBean más `BeanInfo` y que implemente la interfaz `BeanInfo`. Una posibilidad, más sencilla, es que esta clase extienda la clase `SimpleBeanInfo`, que implementa esta interfaz.
2. *Persistencia.* La posibilidad de almacenar los contenidos actuales de un JavaBean para restaurarlo posteriormente. La interfaz `Serializable` implementada por las clases de JavaBeans proporciona un mecanismo básico de persistencia en ficheros. Pero se suele implementar la persistencia de JavaBeans mediante ORM (correspondencia objeto-relacional).
3. *Personalización.* La posibilidad de modificar la apariencia y la conducta del JavaBean durante el diseño. Esto se puede hacer, por ejemplo, utilizando editores de propiedades. Esto es especialmente importante para los componentes visuales.

Los atributos, métodos y eventos que se declaran como públicos constituyen la interfaz del JavaBean. La funcionalidad viene determinada por las operaciones que realizan sus métodos (lo que hacen), y no por su implementación (cómo lo hacen). La especificación de la funcionalidad de un JavaBean debe formar parte de la documentación del JavaBean y distribuirse junto con él. Los JavaBeans se pueden empaquetar para su distribución como parte de un fichero de tipo jar (Java Archive).

8.5. El modelo MVC para desarrollo de aplicaciones web con Java

En este capítulo se muestra el desarrollo en Java de una aplicación web de ejemplo basada en la base de datos para sedes, departamentos y empleados, utilizada en un capítulo anterior dedicado a ORM. Una aplicación web es una aplicación a la que se accede mediante el protocolo HTTP o HTTPS, normalmente utilizando un navegador web.

Se desarrollará la misma aplicación para Java SE y para Java EE. La aplicación se desarrollará conforme al modelo MVC. El modelo MVC es un modelo para el desarrollo de aplicaciones visuales, es decir, con una interfaz gráfica de usuario. Las aplicaciones MVC constan de tres partes que interactúan entre sí:

- *Modelo.* Constituido por objetos persistentes. En el caso de Java, el modelo se implementa mediante JavaBeans. Para aplicaciones basadas en Java EE se utilizan también EJB, que se verán en breve.

- **Vista.** Se encarga de la presentación del modelo en un formato apropiado y significativo para las personas que utilizan la aplicación. La vista es una interfaz de usuario interactiva, en la que se muestra información, y sobre la que el usuario puede actuar para realizar determinadas operaciones sobre el modelo. En el caso de Java, la vista se implementa mediante ficheros de tipo JSP, que se verán en breve.
- **Controlador.** Tiene como misión responder a las acciones del usuario sobre la vista y, si es necesario, realizar las modificaciones pertinentes sobre el modelo y actualizar la vista para reflejar estas modificaciones. En el caso de Java, el controlador se implementa mediante *servlets*, que se verán en breve.



Figura 8.1
Modelo MVC para aplicaciones basadas en Java SE

Es muy recomendable utilizar el modelo MVC para el desarrollo de aplicaciones web. Las aplicaciones que siguen este modelo tienen una relativa complejidad intrínseca. Pero ello se compensa por su mayor fiabilidad, robustez, mantenibilidad y escalabilidad, especialmente importantes para aplicaciones empresariales de gran envergadura.

Existen distintos *frameworks* para el desarrollo de aplicaciones web según el modelo MVC con Java. Entre ellos se pueden citar JSF, Struts y Spring. Los IDE más importantes facilitan la creación de aplicaciones para estos *frameworks* mediante *plugins*, que proporcionan asistentes que automatizan la creación de la estructura fundamental de una aplicación. Los *frameworks* ayudan a evitar tareas repetitivas y complejas para la creación y puesta a punto de la estructura y los componentes fundamentales de las aplicaciones, así como de servicios básicos para ellas. Pero son relativamente complejos y requieren un tiempo significativo de aprendizaje, y por ello no se utilizarán para las aplicaciones de ejemplo.

Para desarrollar una aplicación web en Java conforme al modelo MVC, es necesario un conocimiento básico de las tecnologías que lo hacen posible. Este es un campo muy amplio, por lo que solo se proporcionarán breves explicaciones que permitan entender sus fundamentos y su aplicación en las aplicaciones desarrolladas. El objetivo es proporcionar una buena base a partir de la cual se puedan ampliar conocimientos, investigar y profundizar.

8.6. JSP (JavaServer Pages)

JSP se utiliza para la vista, lo que en el ámbito de las aplicaciones web significa la generación dinámica de páginas HTML. Una página JSP permite generar código HTML utilizando el lenguaje de programación Java. En este sentido, es similar a tecnologías como PHP y ASP.

Una página JSP tiene el aspecto de una página web, con etiquetas de HTML, pero además puede incluir directivas propias de JSP, fragmentos de código en lenguaje Java y referencias a constantes y variables de Java.

8.6.1. Directivas de JSP

Tienen la sintaxis `<%@ directiva atributo="valor" %>`. Se puede especificar valor para más de un atributo.

CUADRO 8.1

Directivas de JSP

<code><%@page import="clase"%></code>	Importa clases o paquetes de Java. Es el equivalente a <code>import clase;</code> Se puede especificar más de una clase separándolas por punto y coma.
<code><%@page contentType="tipo_mime" pageEncoding="codificación"%></code>	Especifica el tipo MIME para la respuesta del protocolo HTTP, si es un tipo de texto, la codificación. Por defecto es <code>contentType="text/html; charset=ISO-8859-1"</code> . Se puede especificar UTF-8 con <code>contentType="text/html" pageEncoding="UTF-8"</code> .
<code><%@page errorPage="página_de_error"%></code>	Especifica la página de error que será invocada si ocurre alguna excepción.
<code><%@page isErrorPage="true"%></code>	Especifica que se trata de una página de error. Solo este tipo de páginas tienen acceso a una variable <code>exception</code> , que contiene la excepción.
<code><%@include file="fichero"%></code>	Incluye en el JSP el contenido del fichero especificado.

8.6.2. Scriptlets

Son fragmentos de código Java con la sintaxis:

`<% código de Java %>`

8.6.3. Variables implícitas

Son variables predefinidas cuyo valor proporciona información acerca del entorno de ejecución de JSP. Entre ellas se tienen las que se recogen en el cuadro 8.2.

CUADRO 8.2

Variables implícitas de JSP

<code>request</code>	Contiene la petición HTTP que se le ha hecho llegar al JSP al invocarlo.
<code>response</code>	Contiene la respuesta HTTP. El JSP puede acceder a este objeto y modificarlo, para proporcionar la respuesta HTTP apropiada.
[.../...]	

CUADRO 8.2 (CONT.)

out	Permite escribir directamente en la salida del JSP, es decir, en el HTML generado. No es práctica recomendable utilizar directamente esta variable para escribir en la salida. En breve se explicará cómo se puede escribir de una manera alternativa, y preferible, en la salida del JSP.
exception	Ya se ha hablado de ella, está disponible solo para páginas definidas como páginas de error mediante una directiva apropiada, y contiene la excepción que se produjo en la página que invocó la página de error.

8.6.4. Referencias a variables de Java

Se puede escribir el valor de una variable en la salida del JSP de la siguiente manera:

```
<%=nombre_variable%>
```

En lugar de una variable, se podría incluir una expresión. Un bloque como el anterior se incluiría en medio de un fragmento de HTML, para insertar el valor de la variable o expresión. Esto sería equivalente, y preferible, a por ejemplo:

```
<out.print(variable)%>
```

8.7. Servlets

Un *servlet* es un proceso servidor accesible mediante el protocolo HTTP (o HTTPS). Un *servlet* está todo el tiempo a la escucha de peticiones HTTP. Cuando recibe una petición, responde en el mismo protocolo. Los aspectos fundamentales acerca de los *servlets* son:

1. *Programación*. En lenguaje Java, un *servlet* es un objeto de una clase que implementa la interfaz **Servlet**.
2. *Despliegue*. En un contenedor de *servlets*, por ejemplo, Apache Tomcat. Un contenedor de *servlets* forma parte de un servidor de aplicaciones en la plataforma Java EE. El fichero de despliegue de una aplicación web para Java EE contiene toda la información acerca de los *servlets* que necesita el contenedor de *servlets* para trabajar con ellos.
3. *Interacción*. Los *servlets* reciben peticiones HTTP o HTTPS y responden en el mismo protocolo. El contenedor de *servlets* aloja los *servlets*, les proporciona servicios básicos, y hace de intermediario (o *proxy*) entre ellos y los clientes. Cuando recibe una petición de un cliente, la redirige al *servlet* apropiado y, cuando obtiene la respuesta del *servlet*, la envía de vuelta al cliente.

Las páginas JSP se implementan mediante *servlets*. JSP no es más que un mecanismo para facilitar el desarrollo de *servlets* que generan código HTML. Cuando se invoca una página JSP

mediante una petición HTTP por primera vez, un compilador de JSP genera un *servlet* a partir de ella, lo compila y le redirige la petición. A partir de ese momento, cada vez que se invoca la página JSP, la petición se redirige directamente al *servlet*.

Un contenedor de *servlets* como Tomcat funciona como servidor web para páginas HTML además de como proxy para los *servlets*. Cuando recibe una petición para una página HTML, la sirve directamente como haría cualquier servidor web.

NetBeans integra un servidor Tomcat y soporte para la creación y el despliegue de *servlets*.

8.8. Desarrollo de una aplicación web MVC basada en JavaBeans

Este apartado proporciona los conocimientos mínimos para el desarrollo de una aplicación web sencilla en Java basada en el modelo MVC. La atención se centra en la funcionalidad de acceso a datos. Para mayor simplicidad, no se utilizará ningún *framework*.

La aplicación se desarrollará para el modelo de datos utilizado para el capítulo anterior dedicado a persistencia de objetos mediante ORM. Permitirá crear sedes y listar las existentes, así como mostrar los detalles de una sede, incluyendo sus departamentos. Será sencillo añadir a esta aplicación el borrado y la modificación de sedes, con lo que se tendrá una aplicación CRUD para sedes. CRUD son las siglas de las operaciones básicas *create, read, update, delete* (crear, leer, actualizar, borrar). Será también muy sencillo añadir las mismas operaciones para los departamentos de una sede y los empleados de un departamento.

La aplicación representará los datos de una sede mediante JavaBeans. Se usará Hibernate para su persistencia en una base de datos relacional.

8.8.1. Creación de la aplicación web

Lo primero es crear una aplicación web en NetBeans. Esto se hace con la opción “File”, “New Project...”, y seleccionando después la opción “Java Web”, “Web Application”. Después se indica un nombre para el proyecto.

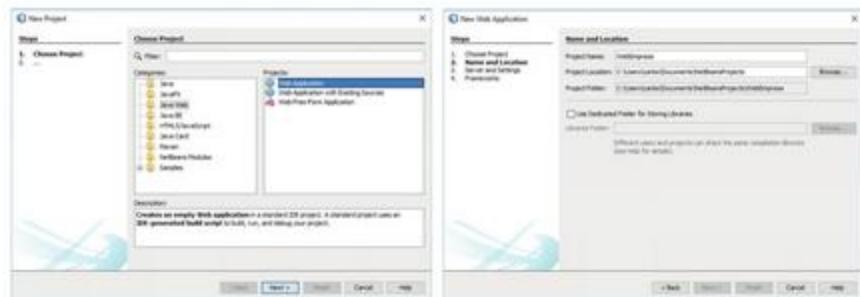


Figura 8.2

Creación de aplicación web con NetBeans

A continuación, es necesario especificar algunas opciones específicas de las aplicaciones web, y los *frameworks* que utilizar. Se selecciona Hibernate, que se utilizará para persistencia de los JavaBeans. El resto no hacen falta.

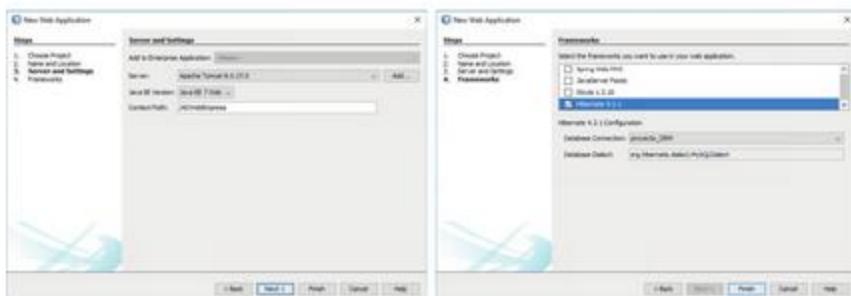


Figura 8.3
Opciones para la aplicación web y frameworks que se van a utilizar

Cuando se instala NetBeans se pueden instalar como servidores de aplicaciones Apache Tomcat, que incluye un contenedor de *servlets*, y GlassFish, que incluye soporte para EJB. Para esta aplicación no se necesitan EJB y basta con Apache Tomcat. Se puede elegir la versión de Java EE más reciente entre todas las disponibles.

El *context path* es la parte de la URL que viene después del servidor (nombre de *host* o IP y puerto, en su caso). Con él selecciona el servidor una aplicación de las que tiene desplegadas y le redirige la petición HTTP. Una vez obtenida la respuesta HTTP de la aplicación, la redirige al cliente. Como *context path* de esta aplicación se especifica `ADWebEmpresa`.

8.8.2. Persistencia de objetos con Hibernate

Es necesario hacer lo mismo que para los proyectos desarrollados en el capítulo anterior dedicado a ORM, a saber:

1. Crear una conexión con la base de datos `proyecto_orm`, con las mismas opciones que en el capítulo anterior dedicado a ORM. El nombre de la conexión será `proyecto_ORM`. Si ya está creada, se puede utilizar directamente.
2. Crear los ficheros `hibernate.cfg.xml` e `hibernate.reveng.xml` utilizando los asistentes de NetBeans para ello. Deben crearse en el paquete por defecto (`<default package>`).
3. Crear el fichero `HibernateUtil.java` con el asistente de NetBeans, en el paquete por defecto. En el capítulo anterior dedicado a ORM se vio que el fichero que se crea es para una versión antigua de Hibernate, y qué cambios hay que hacer en él.
4. Crear POJO para las tablas de la base de datos, con el asistente de NetBeans, en el paquete `ORM`.
5. En su caso, eliminar los `JAR` para Hibernate incluidos por NetBeans, para sustituirlos por los de una versión reciente.

8.8.3. Creación del servlet controlador

Se crea el *servlet* controlador pulsando con el botón derecho sobre la aplicación y seleccionando "New ...", "Servlet ...". Se especifica su nombre (en este caso, `controlador`), y se configura el

servlet. Hay que marcar la opción para añadir esta información al descriptor de despliegue `web.xml`. Como patrón de url para el *servlet* se indica `/controlador`. El *servlet* se podrá invocar con `http://(host)/ADWebEmpresa/controlador`. `/ADWebEmpresa` indica al contenedor de *servlets* Tomcat la aplicación web, y `/controlador`, un *servlet* de la aplicación. Se indican los parámetros del *servlet* y sus valores por defecto. En este caso será solo uno con nombre `op` y valor por defecto nulo. El valor del parámetro `op` indica la acción que se va a realizar por el controlador, y un valor nulo indica que se debe ir a la página principal.

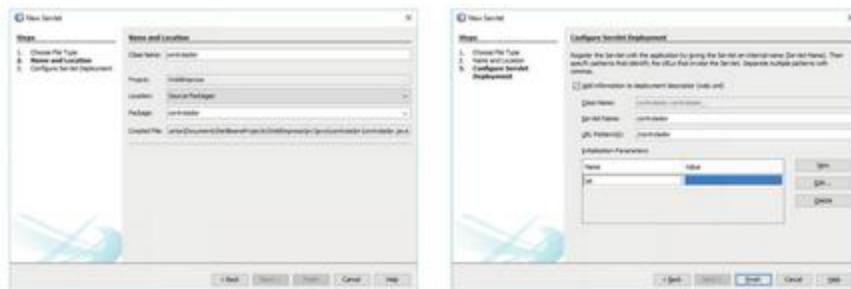


Figura 8.4
Creación y configuración del *servlet* controlador

A continuación, se muestra el contenido del descriptor de despliegue `web.xml`. En un elemento `<servlet>` se describe un *servlet*, y en `<init-param>`, los valores por defecto para cada parámetro (para el caso en que no se especifique ninguno en la petición HTTP con que se le invoca). En `<servlet-mapping>` se especifica la correspondencia entre patrones de URL y *servlets*. En los contenidos mostrados aquí se incluye el cambio, que habría que hacer ahora, de `index.html` por `controlador` en el elemento `<welcome-file>`, para que se invoque directamente el *servlet* controlador. El fichero `index.html` se puede borrar sin más.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-instance
    http://www.w3.org/2001/XMLSchema-instance">
  <servlet>
    <servlet-name>controlador</servlet-name>
    <servlet-class>controlador.controlador</servlet-class>
    <init-param>
      <param-name>op</param-name>
      <param-value></param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>controlador</servlet-name>
    <url-pattern>/controlador</url-pattern>
  </servlet-mapping>
</web-app>
```

```

<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>controlador</welcome-file>
</welcome-file-list>
</web-app>

```

Después, se edita el código del *servlet controlador*, que debe quedar así:

```

// Servlet controlador (controlador.java)
package controlador;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.RequestDispatcher;

public class controlador extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        String oper = request.getParameter("op");
        if(oper==null) oper = "";
        RequestDispatcher rd;
        switch(oper) {
            case "altaSede":
                response.sendRedirect("frmNuevaSede.jsp");
                break;
            case "insertSede":
                rd = request.getRequestDispatcher("procNuevaSede.jsp");
                rd.forward(request, response);
                break;
            case "muestraSede":
                rd = request.getRequestDispatcher("muestraSede.jsp");
                rd.forward(request, response);
                break;
            default:
                response.sendRedirect("home.jsp");
        }
    }
    (...)
```

El *servlet controlador* obtiene el valor del parámetro *op* de la petición HTTP y realiza la acción que se indica en él, que en general consistirá en pasar el control a un *servlet* o JSP, y redirigirle la petición HTTP. Si no se indica ninguna, se invoca la página inicial de la aplicación *home.jsp*, que se desarrollará en breve, al igual que los JSP a los que se hace referencia en este *servlet*.

8.8.4. Uso de JavaBeans asociado a formularios HTML con JSP

Esta aplicación hará uso extensivo del mecanismo disponible en JSP para utilizar JavaBeans para recopilar los datos enviados en los parámetros de una petición HTTP.

El siguiente código define un JavaBean (`jsp:useBean`) de nombre `sede` (`id="sede"`) y clase `ORM.Sede` (`class="ORM.Sede"`). Su ámbito es la petición HTTP, que invoca al JSP (`scope="request"`). Esto significa que solo está disponible para el propio JSP.

El valor de cualquier parámetro de la petición HTTP cuyo nombre coincide con el de cualquier atributo (`property="*"`) de este JavaBean (`name="sede"`) se asigna al atributo.

```
<jsp:useBean id="sede" scope="request" class="ORM.Sede"/>
<jsp:setProperty name="sede" property="*"/>
```

El JavaBean se obtiene mediante el siguiente código en Java:

```
ORM.Sede sedeBean = (ORM.Sede) request.getAttribute("sede");
```

Si no existe ningún parámetro de la petición HTTP cuyo nombre coincide con el de ningún atributo del JavaBean, entonces `sedeBean` toma valor `null`.

8.8.5. Creación de los JSP

Para crear un JSP se pulsa con el botón derecho del ratón sobre la aplicación y se selecciona la opción “New”, “JSP ...”. A continuación, se muestran los JSP de esta aplicación.

`home.jsp` muestra un listado de todas las sedes y un enlace para crear una nueva sede. Al lado de cada una se muestra un enlace para mostrar sus detalles. Todos estos enlaces dirigen al `servlet controlador` mediante peticiones HTTP de tipo POST, con los valores para los parámetros enviados desde el formulario `frm_muestra_sede`. Sería más sencillo emplear elementos `<a>` con la URL en el atributo `href` y la asignación de valores de los parámetros codificados en ella, pero esto generaría peticiones HTTP de tipo GET en las que serían visibles todos estos detalles en la URL mostrada por el navegador, lo que hay que evitar por seguridad. Para las clases de los JavaBeans, se ha implementado la persistencia con Hibernate, lo que hace muy sencillo recuperarlos de la base de datos y reflejar en ella los cambios sobre ellos.

```
<!-- home.jsp -->
<%@page import="java.util.List"%>
<%@page import="org.hibernate.Session"%>
<%@page import="org.hibernate.query.Query"%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Empresa</title>
    </head>
```

```

<body>
    <form name="frm_muestra_sede" method="post" action="controlador">
        <input type="hidden" name="op" value="muestraSede">
        <input type="hidden" name="idSede">
        <table border="1">
            <tr><td colspan="2" align="center">Sedes [<a
                href="javascript:void(0)" onclick="javascript:document.
                frm_muestra_sede.op.value = 'altaSede';document.frm_muestra_
                sede.submit();">Nueva sede</a>]</td></tr>
        <br>
        Session s = ORM.HibernateUtil.getSessionFactory().openSession();
        try {
            Query q = s.createQuery("FROM Sede ORDER BY
                nomSede,idSede").setReadOnly(true);
            List<ORM.Sede> listaSedes = (List<ORM.Sede>)
                q.getResultList();
            if (listaSedes.isEmpty()) {<br>
                <tr><td colspan="2">No existen sedes</td></tr>
                <br>
            } else {
                for (ORM.Sede unaSede: listaSedes) {<br>
                    <tr>
                        <td><%=unaSede.getIdSede()%></td>
                        <td><a href="javascript:void(0)"
                            onclick="javascript:document.frm_muestra_sede.
                            idSede.value = '<%=unaSede.getIdSede()%>';
                            document.frm_muestra_sede.submit();"><%=unaSede.
                            getNomSede()%></a></td>
                    </tr>
                    <br>
                }
            }
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
        <br>
    </table>
</form>
</body>
</html>

```

`frmNuevaSede.jsp` permite la creación de una nueva sede, y muestra un formulario para introducir sus datos, que envía los datos a `procNuevaSede.jsp` para crearla.

```

<!-- frmNuevaSede.jsp -->
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<jsp:useBean id="sede" scope="request" class="ORM.Sede"/>
<jsp:setProperty name="sede" property="*"/>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Alta de Sede</title>
    </head>

```

```

<body>
    <form method="post" action="controlador">
        <table>
            <tr>
                <td>
                    <input type="hidden" name="op" value="insertSede"/>
                    <input name="nomSede" required type="text" size="20"
                           maxlength="20"/>
                </td>
            </tr>
            <tr>
                <td>
                    <input type="submit" value="Crear"/>
                    <input type="reset" name="cancelar" value="Cancelar"/>
                </td>
            </tr>
        </table>
    </form>
    <a href="controlador">Inicio</a>
</body>
</html>

```

`procNuevaSede.jsp` crea una nueva sede con los datos recogidos en el JavaBean llamado `sede`, que se hace persistente con el método `save()` de la interfaz `Session` de Hibernate.

```

<!-- // procNuevaSede.jsp -->
<%@page import="org.hibernate.Session"%>
<%@page import="org.hibernate.Transaction"%>
<%@page import="org.hibernate.query.Query"%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<jsp:useBean id="sede" scope="request" class="ORM.Sede"%>
<jsp:setProperty name="sede" property="*"/>

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Creación de sede</title>
    </head>
    <body>
<%
    ORM.Sede sedeBean = (ORM.Sede) request.getAttribute("sede");
    if(sedeBean == null) {
%>
        ERROR: no se proporcionaron datos de sede para crear.
<%
    }
    else {
        Transaction t = null;
        try(Session s = ORM.HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();
            s.save(sedeBean);
            t.commit();
        }
%>

```

```

Creada nueva sede: [<@=sedeBean.getIdSede()@>] <@=sedeBean.
    getNomSede()@><br/>
<a href="controlador">Volver</a>
<br>
    } catch (Exception e) {
        e.printStackTrace(System.err);
        if (t != null) {
            t.rollback();
        }
    }
</body>
</html>

```

Por último, `muestraSede.jsp` muestra los datos de una sede, incluyendo la lista de sus departamentos. Se usa un JavaBean de nombre `sede` en el que está disponible el identificador de la sede `idSede`. El resto de los datos se obtienen mediante el método `refresh()` de la interfaz `Session` de Hibernate.

```

<!-- // muestraSede.jsp -->
<%@page import="java.util.Iterator"%>
<%@page import="org.hibernate.Session"%>
<%@page import="org.hibernate.query.Query"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<jsp:useBean id="sede" scope="request" class="ORM.Sede"/>
<jsp:setProperty name="sede" property="*"/>
<br>
    ORM.Sede sedeBean = (ORM.Sede) request.getAttribute("sede");
    if (sedeBean == null) {
<br>
ERROR: no se especificó sede a mostrar.
<br>
    } else {
        try (Session s = ORM.HibernateUtil.getSessionFactory().openSession()) {
            s.refresh(sedeBean);
        }
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Sede <@=sedeBean.getIdSede()@> - <@=sedeBean.getNomSede()@></title>
    </head>
    <body>
        <p>Sede <@=sedeBean.getIdSede()@> - <@=sedeBean.getNomSede()@></p>
        <table border="1"><tr><td colspan="2" align="center">Departamentos</td></tr>
        <br>
        if (sedeBean.getDepartamentos().isEmpty()) {%
            <tr><td colspan="2">No existen departamentos en esta sede</td></tr>
            <br>
        } else {

```

```

Iterator itDeptos = sedeBean.getDepartamentos().iterator();
while (itDeptos.hasNext()) {
    ORM.Departamento unDepto = (ORM.Departamento) itDeptos.next();
    %>
<tr>
    <td><%=unDepto.getIdDepto()%></td>
    <td><%=unDepto.getNomDepto()%></td>
</tr>
<%
}
%>
</table>
</body>
</html>
<%
    } catch (Exception e) {
        e.printStackTrace(System.err);
    }
%>

```

El proyecto quedaría como se muestra en la figura 8.5. La vista está constituida por varias páginas JSP situadas bajo el apartado “Web Pages”. El controlador consta de un único fichero de clases de Java dentro de un paquete **controlador**, y el modelo está constituido por varios POJO junto con sus ficheros de correspondencia de Hibernate, en el paquete **ORM**. En el proyecto están, además, los ficheros adicionales necesarios para Hibernate.

A continuación, se proponen varias actividades sucesivas para ampliar la aplicación anterior para convertirla en una aplicación CRUD completa, tanto para sedes como para departamentos y empleados.

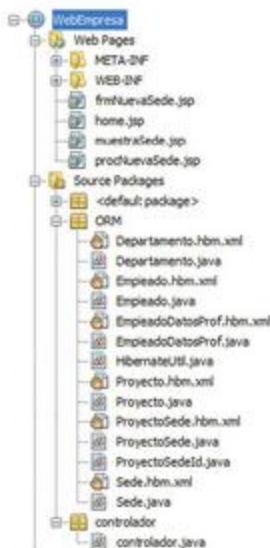


Figura 8.5
Proyecto para aplicación web

Actividad propuesta 8.1



Completa la aplicación anterior para que permita:

- Borrar sedes.
- Modificar los datos de una sede.
- Crear, ver, modificar y borrar departamentos. No olvides que los departamentos siempre pertenecen a una sede. Al mostrar los datos de un departamento se debe mostrar su lista de empleados, de manera análoga a como en el programa de ejemplo, al mostrar los datos de una sede se muestra su lista de departamentos.
- Crear, ver, modificar y borrar empleados. No olvides que los empleados siempre están adscritos a un departamento.
- Buscar empleados por DNI y por nombre.

8.9. Enterprise JavaBeans

El modelo de componentes de la plataforma Java EE está basado en Enterprise JavaBeans o EJB, que se despliegan en un contenedor de EJB, cuya existencia es obligatoria en un servidor para Java EE. Existen distintos tipos de EJB, pero en esta aplicación solo se utilizarán *session beans* de tipo *stateless*, es decir, EJB de sesión sin estado.

NetBeans incluye el servidor GlassFish para Java EE, que proporciona un contenedor de EJB. Tomcat proporciona un contenedor de *servlets*, pero no de EJB.

Se explicará lo mínimo indispensable para poder entender el uso de EJB en una aplicación para Java EE web con idéntica funcionalidad que la anterior para Java SE. Java EE ofrece algunas ventajas para el desarrollo de aplicaciones, a cambio de la necesidad de disponer de un relativamente complejo servidor de aplicaciones instalado y convenientemente configurado. Entre ellas:

- Disponibilidad de implementaciones de las API JPA para persistencia de objetos y JTA para transacciones, como parte de la especificación Java EE.
- Sincronización en el acceso a los datos. Solo un hilo de ejecución o hebra (*thread*) puede acceder a la vez a un *session bean*. Esto evita problemas que podrían darse si más de un *thread* accede simultáneamente a la interfaz `EntityManager` de JPA, que no es *thread-safe*, es decir, que no garantiza que funcionará correctamente en ese caso. Los *session beans* hacen innecesario el uso de métodos de sincronización de los que dispone Java, como la palabra clave `synchronized`.
- Disponibilidad del mecanismo CDI (*context and dependency injection*) para acceso directo y sencillo a servicios básicos de la plataforma Java EE. CDI se introdujo en Java EE 6, y se basa en el uso de anotaciones de Java para acceder a servicios tales como persistencia (con JPA) y transacciones distribuidas (con JTA).

Así pues, los *session beans* utilizan JPA para persistencia de objetos y garantizan la integridad de los datos mediante el uso de transacciones proporcionadas por JTA. Pero, además —y esta es una importante ventaja con respecto a la plataforma Java SE—, proporcionan sincronización en el acceso a dichos objetos persistentes.

En los *session beans* se implementan métodos de muy alto nivel para el acceso a los datos. Se puede decir que los *session beans* sin estado (*stateless*) se utilizan como objetos DAO. El patrón de diseño DAO consiste en el acceso a datos mediante clases que ofrecen una interfaz de alto nivel e independiente del tipo de almacenamiento de datos. En este caso particular, se utiliza JPA para el acceso a los datos que están en una base de datos relacional, pero esto es solo un detalle de la implementación, que no se refleja para nada en la interfaz de las clases DAO.

8.10. Desarrollo de una aplicación web MVC para Java EE con EJB

El planteamiento es muy similar al visto anteriormente para Java SE, pero se utilizan EJB de sesión sin estado (*stateless session EJB*) como DAO. Estos proporcionan métodos de muy alto nivel y próximos a la lógica de negocios. Se puede decir que los *session beans* permiten un acceso sencillo (mediante métodos de alto nivel), sincronizado (mediante el mecanismo de sincronización intrínseco de los *session beans*) y seguro (mediante el uso de transacciones) a las clases de entidad o *entity classes*, que son clases persistentes mediante JPA.

Se puede utilizar cualquier implementación de JPA de las disponibles. Se utiliza la implementación de Hibenate, no la de EclipseLink que se propone por defecto.

8.10.1. Creación de la aplicación web

Se crea también con “File”, “New Project”, “Java Web”, “Web Application”. Como nombre de aplicación se especifica, por ejemplo, **WebEmpresaEJB**.

Después, se especifica servidor. Para utilizar EJB hace falta un servidor con soporte para EJB como GlassFish, que viene integrado con NetBeans y se puede instalar con él. Como *context path* se puede dejar el mismo nombre del proyecto, que se propone por defecto. En el siguiente diálogo no se selecciona ningún *framework*, tampoco Hibenate, aunque, como se ha comentado, se utilizará la implementación de JPA que proporciona Hibenate, pero esto corresponde a un paso posterior.

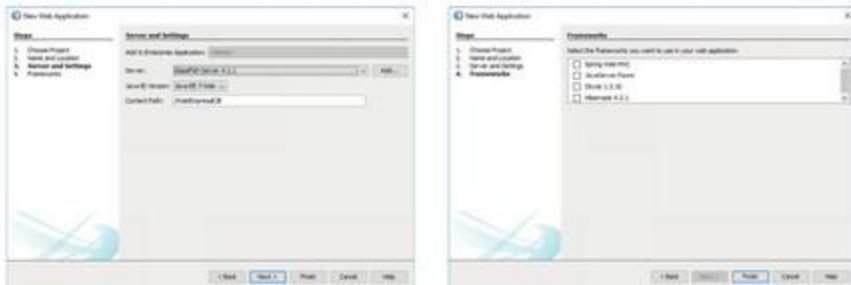


Figura 8.6

Creación de la aplicación web y selección de frameworks

8.10.2. Creación de la unidad de persistencia

La unidad de persistencia se encargará de la persistencia de un conjunto de clases de entidad (*entity classes*), utilizando JPA. Las clases de entidad se asociarán a ella en un paso posterior. Se pulsa con el botón derecho del ratón sobre el proyecto, se selecciona la opción “New...”, “Other...”, “Persistence”, “Persistence Unit”, y se indica **WebEmpresaEJBU** como nombre. Se puede elegir una de las implementaciones disponibles de JPA. La opción por defecto es EclipseLink, pero se ha elegido Hibenate para explorar otra faceta de Hibenate, a saber, su implementación de JPA (en el capítulo dedicado a ORM se utilizó su API nativa y no su implementación de JPA). Por supuesto, se quiere utilizar Java Transaction API (JTA). Como “Table generation strategy” se elige “None” (las tablas ya existen, y las clases de entidad se generarán a partir de ellas). Se crea un nuevo *Data Source* **WebEmpresaeJBDS** basado en la conexión JDBC **proyecto_ORM** para la base de datos **proyecto_orm**, creadas en el capítulo previo dedicado a ORM. Si no existieran, se pueden crear como se explicó en ese capítulo.

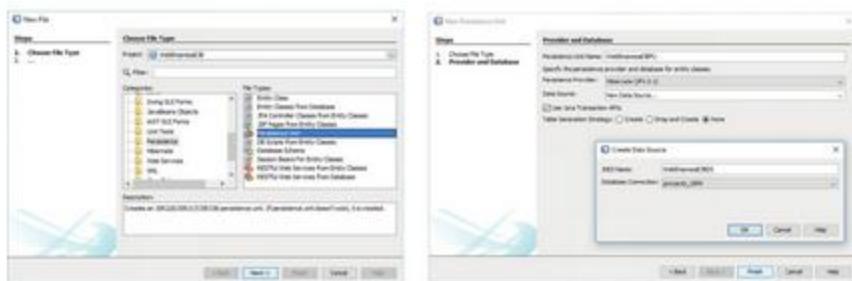


Figura 8.7
Creación de la unidad de persistencia

Con esto, la definición de la unidad de persistencia se graba en `persistence.xml`. En ella está incluido el nombre de la fuente de datos.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/
  persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.
  jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="WebEmpresaEJBPU" transaction-type="JTA">
    <jta-data-source>java:app/WebEmpresaEJBDS</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties/>
  </persistence-unit>
</persistence>
```

Con esto, además, se añaden al proyecto las bibliotecas de Hibernate. Estas se pueden sustituir más adelante por las de una versión más reciente.

8.10.3. Creación de las clases de entidad

Las clases de entidad se crean a partir de las tablas, y quedarán asociadas a la unidad de persistencia. Para ello se pulsa con el botón derecho sobre el proyecto, se elige “New...”, “Other...”, “Persistence”, “Entity Classes from Database”, y se seleccionan todas las tablas.

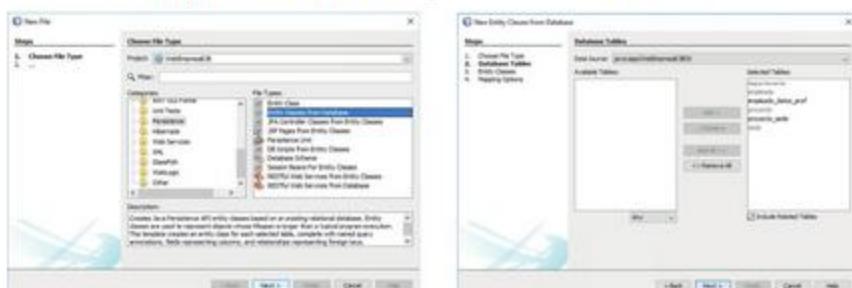


Figura 8.8
Selección de tablas para unidad de persistencia

Hay que especificar un paquete para las clases de entidad, que constituirán el modelo (M) de la aplicación MVC. Se puede especificar `com.miempresa.modelo`. No se necesitan anotaciones para JAXB. Como “Association Fetch” se selecciona “lazy”, cuyo significado se vio en el capítulo dedicado a ORM.

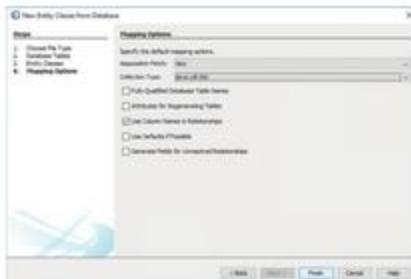


Figura 8.9
Opciones para la creación de las clases de entidad

Con esto están creadas las clases de entidad dentro del paquete `com.miempresa.modelo`. Estas clases son JavaBeans con anotaciones de JPA para implementar su persistencia.

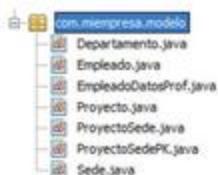


Figura 8.10
Clases de entidad

8.10.4. Creación de los EJB de sesión para las clases de entidad

Se crean pulsando con el botón derecho del ratón sobre el proyecto, y seleccionando “New...”, “Other...”, “Enterprise Java Beans”, “Session Beans for Entity Classes”. Se seleccionan todas las clases de entidad mostradas. Se selecciona la opción de crear interfaz local y, como paquete para los EJB de sesión, se selecciona `com.miempresa.dao`.



Figura 8.11
Creación de Session Beans sin estado para clases de entidad

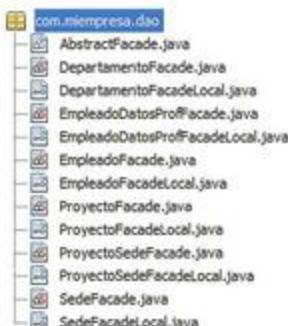


Figura 8.12
EJB de sesión

Con esto se crean los EJB de sesión para las clases de entidad. Por cada una se crea un *session bean* que consta de una interfaz local (etiquetada con `@Local`) y una clase que la implementa (etiquetada con `@Stateless` para indicar que se trata de un EJB de sesión sin estado). Por ejemplo, para la clase de entidad `Sede` se crea la interfaz local `SedeFacadeLocal` y la clase `SedeFacade` que la implementa. La interfaz local incluye, entre otras cosas, todos los métodos necesarios para una aplicación CRUD, que se implementan en la clase mediante métodos de JPA, según se explica en el cuadro 8.3.

CUADRO 8.3
Implementación con JPA de los métodos de la interfaz local

Método	Método de JPA (EntityManager)
C(reate)	<code>create(T entidad)</code>
R(ead)	<code>find(Object id)</code>
U(pdate)	<code>edit(T entidad)</code>
D(elete)	<code>remove(T entidad)</code>

Más en detalle, `SedeFacade` extiende la clase abstracta `AbstractFacade<Sede>`, que es la que proporciona la implementación de los métodos de `SedeFacadeLocal`, y `SedeFacade` solo añade la implementación del método `getEntityManager()`. De hecho, justo este método se debe cambiar, para dejarlo como sigue. Se indican los cambios realizados en negrita, con respecto al código generado por NetBeans. El resto de las clases `Facade` deben cambiarse de igual manera.

```
// SedeFacade.java

package com.miempresa.dao;

import com.miempresa.modelo.Sede;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
// import javax.persistence.PersistenceContext; // Eliminado
import javax.persistence.Persistence; // Añadido

@Stateless
public class SedeFacade extends AbstractFacade<Sede> implements
    SedeFacadeLocal {

    // PersistenceContext(unitName = "WebEmpresaEJBPU") // Eliminado
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        em = Persistence.createEntityManagerFactory("WebEmpresaEJBPU").
            createEntityManager(); // Añadido
```

```

        return em;
    }

    public SedeFacade() {
        super(Sede.class);
    }
}

```

Facade es una transcripción del francés *Façade* y significa que estas clases actúan como una fachada que oculta los detalles de las clases de entidad, y ofrecen una interfaz de más alto nivel para el acceso a los datos. Pero las clases *Facade* son solo un paso en esa dirección, y no se pueden considerar verdaderas clases DAO. Se crea **GestorSedes** como clase DAO para la clase de entidad **Sede**, basada en **SedeFacade**, pero que proporciona métodos de más alto nivel para el acceso a los datos. A estos métodos no se les pasan objetos de clases de entidad, sino tipos de datos sencillos. Por ejemplo, `createSede(String nomSede)` crea una sede con el nombre dado. Las operaciones realizadas por estos métodos se harán, además, dentro de una transacción. Esta clase se crea como un EJB de sesión, también en el paquete `com.miempresa.dao`, pero para ella no se crea una interfaz local. Se puede crear la clase pulsando con el botón derecho del ratón sobre el proyecto y seleccionando la opción “New...”, “Other...”, “Enterprise Java Beans”, “Session Bean”. Las definiciones de sus métodos se pueden crear como se indica en los comentarios incluidos en el código generado automáticamente (Right-click in editor and choose “Insert Code > Add Business Method”), pero esto no aporta realmente mucho valor ni ahorra mucho trabajo.

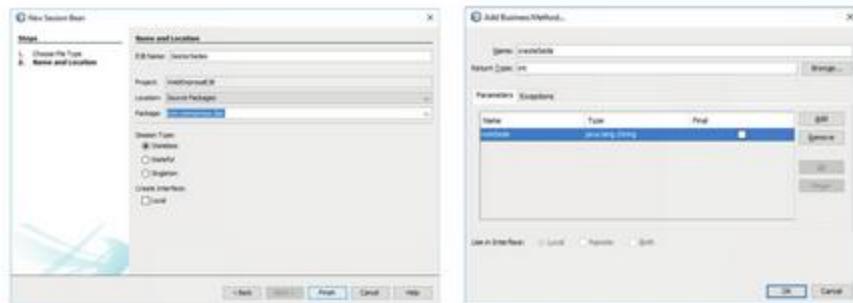


Figura 8.13
Creación de EJB de sesión sin estado que funcionan como clases DAO

El código de la clase **GestorSedes**, incluyendo el método `createSede`, es el siguiente:

```

// GestorSedes.java

package com.miempresa.dao;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;

import com.miempresa.modelo.Sede;

@Stateless
public class GestorSedes {

```

```

private final SedeFacade sedeFacade = new SedeFacade();

// Add business logic below. (Right-click in editor and choose
// "Insert Code > Add Business Method")

public int createSede(String nomSede) {
    Sede entidadSede = new Sede();
    entidadSede.setNomSede(nomSede);
    EntityManager em = sedeFacade.getEntityManager();
    EntityTransaction tx = em.getTransaction();
    try {
        tx.begin();
        sedeFacade.create(entidadSede);
        tx.commit();
        return entidadSede.getIdSede();
    } catch (Exception e) {
        try {
            tx.rollback();
        } catch (Exception el) {
            el.printStackTrace();
        }
    }
    e.printStackTrace();
    return -1;
}
}
}

```

8.10.5. Creación del servlet controlador

Ahora se crea el *servlet* controlador. En lugar de utilizar un único *servlet* controlador, se puede utilizar uno por cada clase de entidad. Para esta aplicación se creará solo el *servlet* controlador para sedes, que hará las mismas funciones que el creado para la aplicación anterior. Se crea pulsando con el botón derecho del ratón sobre el proyecto, con la opción “New...”, “Other...”, “Web”, “Servlet”. Como nombre del *servlet* se indica **ServletSede**, y como nombre del paquete, **com.miempresa.controlador**.

Se selecciona la opción de añadir información al descriptor de despliegue (**web.xml**), y se indica el patrón de URL asociado a él. Se añade un parámetro **op** con valor por defecto sin especificar (cadena vacía). En resumen, como para la aplicación anterior.

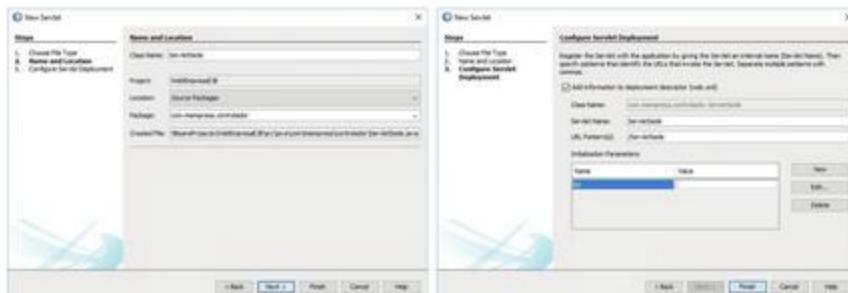


Figura 8.14
Creación del servlet controlador

El código del *Servlet* es prácticamente idéntico al de la aplicación anterior:

```
// ServletSede.java
package com.miempress.controlador;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.annotation.WebServlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet(name="ServletSede")
public class ServletSede extends HttpServlet {
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
String oper = request.getParameter("op");
if(oper==null) oper = "";
RequestDispatcher rd;
switch(oper) {
case "altaSede":
response.sendRedirect("frmNuevaSede.jsp");
break;
case "insertSede":
rd = request.getRequestDispatcher("procNuevaSede.jsp");
rd.forward(request, response);
break;
case "muestraSede":
rd = request.getRequestDispatcher("muestraSede.jsp");
rd.forward(request, response);
break;
default:
response.sendRedirect("home.jsp");
}
}
(...)
```

Actividades propuestas



- 8.2.** Completa la clase `GestorSedes` con métodos `retrieveAll`, `retrieveSede`, `updateSede` y `deleteSede`. `retrieveAll` debe devolver un `List<Sede>` con todas las sedes. `retrieveSede` debe devolver un objeto de la clase `Sede` a partir de su identificador. `updateSede` permitirá cambiar los datos de la sede (ahora mismo, solo el nombre), y `deleteSede` borrará la sede. La sede se indicará a `updateSede` y `deleteSede` con su identificador numérico.

- 8.3. Crea nuevas clases DAO `GestorDepartamentos`, `gestorProyectos` y `gestorEmpleados` tomando como modelo `GestorSedes`. En general, tendrán parámetros adicionales. Por ejemplo, al crear un departamento debe indicarse la sede a la que pertenece, mediante el identificador numérico de la sede.
- 8.4. Amplia la aplicación web de manera similar a como se propuso en actividades anteriores para la anterior aplicación, de manera que se convierta en una aplicación CRUD para sedes, departamentos y empleados. Deben usarse las clases DAO `GestorSedes`, `GestorDepartamentos`, `gestorProyectos` y `gestorEmpleados`.
- 8.5. Añade métodos a las clases DAO anteriores para gestionar las relaciones entre ellas. Por ejemplo, la clase `GestorDepartamentos` puede tener un método para reasignar un empleado a otro departamento. También debe ser posible asignar un proyecto a un departamento si no está asignado a ninguno, o reasignarlo a otro departamento si está asignado a alguno. Esta es una actividad abierta, en la que no se trata tanto de dar con la solución correcta (hay más de una posible), sino de proponer una solución completa y con un planteamiento consistente, y de justificar dicho planteamiento y sus posibles ventajas e inconvenientes frente a otros posibles.
- 8.6. Amplía la aplicación web para utilizar los métodos creados para la actividad propuesta anterior.
- 8.7. ¿Qué cambios hay en este servlet controlador con respecto al de la anterior aplicación?

8.10.6. Configuración básica inicial de la aplicación web

Se trata de que, al invocar la aplicación, se invoque el *servlet* controlador para sedes. Esto se consigue mediante el siguiente cambio en `web.xml`. El fichero `index.html` se puede borrar sin más.

<code>web.xml</code>	<pre><?xml version="1.0" encoding="UTF-8"?> <web-app ...> ... <welcome-file-list> <welcome-file>ServletSede</welcome-file> </welcome-file-list> </web-app></pre>
----------------------	--

8.10.7. Creación de los JSP

Los JSP son muy similares a los de la anterior aplicación, pero algo más sencillos, porque no tienen que ocuparse de los detalles de persistencia y las transacciones. De eso se encargan los métodos de los DAO implementados como *session beans*. Se muestran en negrita los cambios más significativos.

Los JSP que se muestran como ejemplo siguen utilizando la clase `FacadeSede`. Debería utilizarse en su lugar la clase DAO `GestorSedes`. Los cambios necesarios para ello se propondrán como ejercicios. Para llevarlos a cabo será necesario utilizar, a su vez, los nuevos métodos de `GestorSedes` que se propusieron en ejercicios anteriores.

```
<!-- home.jsp -->
<%@page import="java.util.List"%>
<%@page import="com.miemuestra.dao.SedeFacade"%>
<%@page import="com.miemuestra.modelo.Sede"%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Empresa</title>
</head>
<body>
    <form name="frm_muestra_sede" method="post" action="ServletSede">
        <input type="hidden" name="op" value="muestraSede">
        <input type="hidden" name="idSede">
        <table border="1">
            <tr><td colspan="2" align="center">Sedes [<a href="#" onclick="javascript:document frm_muestra_sede.op.value='altaSede';document frm_muestra_sede.submit();">Nueva sede</a>]</td></tr>
            <tr>
                <td>
                    try {
                        SedeFacade daoSede = new SedeFacade();
                        List<Sede> listaSedes = daoSede.findAll();
                        for (Sede unaSede: listaSedes) {<br>
                            <tr>
                                <td><%=unaSede.getIdSede()%></td>
                                <td><a href="#" onclick="javascript:document frm_muestra_sede.idSede.value='<%=unaSede.getIdSede()%>';document frm_muestra_sede.submit();"><%=unaSede.getNomSede()%></a></td>
                            </tr>
                        }
                    } catch (Exception e) {
                        e.printStackTrace(System.out);
                    }
                </td>
            </tr>
        </table>
    </form>
</body>
</html>
```

Actividad propuesta 8.8



Cambia el JSP anterior para utilizar el método `retrieveAll` de `GestorSedes` en lugar del método `findAll` de `SedeFacade`.

`frmNuevaSede.jsp` es casi idéntico al de la anterior aplicación, solo cambia el nombre del `servlet` controlador.

```
<!-- frmNuevaSede.jsp -->
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<jsp:useBean id="sede" scope="request" class="com.miempresa.modelo.Sede"/>
<jsp:setProperty name="sede" property="" />
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Alta de Sede</title>
    </head>
    <body>
        <form method="post" action="ServletSede">
            <table>
                <tr>
                    <td>
                        <input type="hidden" name="op" value="insertSede"/>
                        <input name="nomSede" required type="text" size="20"
                               maxlength="20"/>
                    </td>
                </tr>
                <tr>
                    <td>
                        <input type="submit" value="Crear"/>
                        <input type="reset" name="cancelar" value="Cancelar"/>
                    </td>
                </tr>
            </table>
        </form>
        <a href="ServletSede">Inicio</a>
    </body>
</html>
```

`procNuevaSede.jsp` es más sencillo que en la anterior aplicación gracias a la clase DAO `GestorSedes`.

```
<!-- procNuevaSede.jsp -->
<%@page import="com.miempresa.modelo.Sede"%>
<%@page import="com.miempresa.dao.GestorSedes"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<jsp:useBean id="sede" scope="request" class="com.miempresa.modelo.Sede"/>
<jsp:setProperty name="sede" property="" />
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Creación de sede</title>
    </head>
    <body>
        <%
            Sede sedeBean = (Sede) request.getAttribute("sede");
            if(sedeBean == null) {
        %>
```

```

ERROR: no se proporcionaron datos de sede para crear.
<%
    }
    else {
        try {
            GestorSedes gestorSedes = new GestorSedes();
            gestorSedes.createSede(sedeBean.getNomSede());
        }
    }
    catch (Exception e) {
        e.printStackTrace(System.err);
    }
}
</body>
</html>

```

Por último, `muestraSede.jsp` es casi idéntico al de la anterior aplicación.

```

<!-- muestraSede.jsp -->
<%@page import="com.miempreza.dao.SedeFacade"%>
<%@page import="com.miempreza.modelo.Sede"%>
<%@page import="com.miempreza.modelo.Departamento"%>
<%@page import="java.util.Iterator"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<jsp:useBean id="sede" scope="request" class="com.miempreza.modelo.Sede"/>
<jsp:setProperty name="sede" property="" />
<%
    Sede sedeBean = (Sede) request.getAttribute("sede");
    if (sedeBean == null) {
%>
ERROR: no se especificó sede a mostrar.
<%
} else {
    try {
        SedeFacade daoSede = new SedeFacade();
        sedeBean = daoSede.find(sedeBean.getIdSede());
    }
<%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Sede <%=sedeBean.getIdSede()%> - <%=sedeBean.getNomSede()%></title>
    </head>
    <body>
        <p>Sede <%=sedeBean.getIdSede()%> - <%=sedeBean.getNomSede()%></p>
        <table border="1">

```

```

<tr>
    <td colspan="2" align="center">Departamentos</td></tr>
    <%
if (sedeBean.getDepartamentoSet().isEmpty()) {%
<tr><td colspan="2">No existen departamentos en esta sede</td></tr>
    <%
} else {
    Iterator itDeptos = sedeBean.getDepartamentoSet().iterator();
    while (itDeptos.hasNext()) {
        Departamento unDepto = (Departamento) itDeptos.next();
        %>
<tr>
    <td><%=unDepto.getIdDepto()%></td>
    <td><%=unDepto.getNomDepto()%><a href="#"></a></td>
</tr>
    <%
    }
}
    %>
</table>
</body>
</html>
<%
} catch (Exception e) {
    e.printStackTrace(System.err);
}
%>

```



Actividad propuesta 8.9

Cambia el JSP anterior para utilizar el método `retrieveSede` de `GestorSedes` en lugar del método `find` de `SedeFacade`.

8.10.8. Incluir los ficheros jar de una versión reciente de Hibernate

En pasos anteriores se han añadido al proyecto las bibliotecas de Hibernate. Antes de compilar y desplegar la aplicación, se pueden sustituir por los de una versión más reciente, como ya se ha hecho en anteriores proyectos.

8.10.9. Despliegue de la aplicación

Antes de desplegar la aplicación, se compila mediante la opción “Clean and Build” mostrada al pulsar con el botón derecho del ratón sobre la aplicación.

La vista “Files” (ficheros) es un buen lugar para entender los contenidos de la aplicación web una vez construida y lista para su despliegue. El fichero de la aplicación está en `build`, y es `WebEmpresaEJB.war`. Se pueden ver sus contenidos. En el paquete por defecto están los

JSP. En otros paquetes están las clases de sus distintos componentes. Se ven los ficheros `persistence.xml`, `glassfish-resources.xml`, y `web.xml`.

En `WEB-INF/lib` se ven los ficheros de biblioteca (jar) de esta aplicación, entre ellos los de Hibernate.

Durante el despliegue de la aplicación se pueden producir errores muy diversos. Cuando se despliega un módulo de EJB, se crean e inicializan todos los recursos asociados a él, entre ellos la unidad de persistencia y los *pools* de conexiones para la base de datos. Hay muchas cosas que podrían fallar, y entonces el módulo no se desplegaría.

La aplicación se despliega con la opción “Deploy”, que aparece al pulsar con el botón derecho del ratón sobre el proyecto. Si hubiera cualquier error que impida el despliegue de la aplicación, hay que consultar los logs del servidor de aplicaciones (GlassFish), lo que se puede hacer desde el propio NetBeans.

GlassFish viene con unos cuantos *drivers* JDBC, y NetBeans establece automáticamente las configuraciones necesarias para utilizarlos. Si se necesita utilizar un *driver* distinto, no hay que incluirlo como biblioteca (*library*) del proyecto, sino que hay que copiarlo manualmente en los directorios del servidor de aplicaciones GlassFish, porque este necesita establecer una conexión con el DataSource antes de desplegar la aplicación. Además, hay que establecer manualmente las opciones de configuración para dicho *driver*. A continuación, se explica cómo hacer esto para el *driver* JDBC de MySQL 8.0. Para otros *drivers*, habrá que consultar la documentación del *driver* en cuestión o de GlassFish.



PARA SABER MÁS

Configuración para un *driver* JDBC, por ejemplo, el de MySQL 8.0

El uso de un *driver* JDBC distinto a los que vienen incluidos con GlassFish requiere añadir a la instalación de GlassFish el fichero jar del *driver* y algunos cambios de configuración. El fichero jar hay que copiarlo en `glassfish/lib` o, según el caso, en `domains/domains1/lib`. El directorio de instalación de GlassFish depende del sistema operativo. En Linux es `~/glassfish-(versión)`, y en Windows es `C:\Archivos de Programa\glassfish-(versión)`.

Para utilizar el *driver* de MySQL 8.0, por ejemplo, hay que copiar en uno de estos directorios el fichero `mysql-connector-java-8.0.11.jar`. Además, hay que hacer los cambios que se indican a continuación en el fichero `glassfish-resources.xml`, que contiene configuraciones de la aplicación específicas para GlassFish. A saber, hay que indicar el nombre de dos clases necesarias para poner en marcha el *pool* de conexiones. Pero, además, se produce un error al intentar establecer una conexión segura, mediante SSL, con MySQL 8.0.

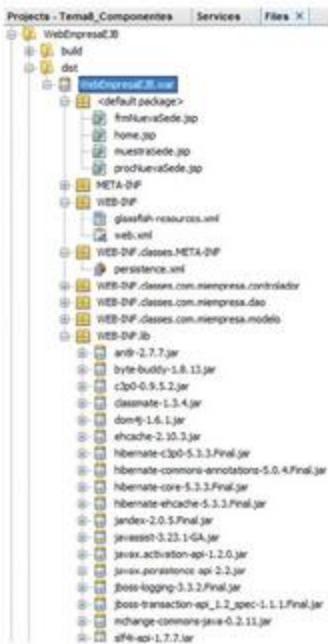


Figura 8.15

Contenidos de paquete war

Ante eso se pueden hacer dos cosas: deshabilitar la conexión segura o hacer los cambios necesarios para establecer correctamente una conexión segura. La cuestión de cómo establecer una conexión segura con MySQL 8.0 queda fuera del alcance de este libro y se opta por la primera opción, la más sencilla.

```
<jdbc-connection-pool (...)  
datasourceclassname="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"  
(...)>  
<property name="URL" value="...&useSSL=false"/>  
<property name="driverClass" value="com.mysql.jdbc.Driver"/>  
</jdbc-connection-pool>
```

Después de estos cambios de configuración, hay que recomilar la aplicación y volver a desplegarla.

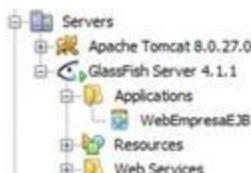


Figura 8.16
Aplicación web desplegada en GlassFish

Una vez desplegada la aplicación, se puede ver bajo GlassFish (figura 8.16).

8.10.10. Solución de errores adicionales en tiempo de ejecución

Todavía se pueden producir algunos errores en el momento de ejecutar la aplicación. Es necesario leer atentamente los logs de GlassFish e intentar comprender la causa de los errores que se pueden producir y buscar solución para ellos.

Recurso digital



En el anexo web 8.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás más información sobre errores.

TOMA NOTA



La correcta instalación y configuración de un servidor de aplicaciones para Java EE, y el despliegue de aplicaciones en él, pueden ser tareas complejas. Parte de estas tareas, y en concreto la puesta a punto de los descriptores de despliegue y demás ficheros de configuración, corresponden más bien a un perfil profesional de administrador de sistemas que a uno de desarrollador, al menos si se trata de desplegar la aplicación en un entorno de producción o de preproducción (*staging*). Un perfil profesional de este tipo dará normalmente soporte a los desarrolladores para la correcta puesta a punto de los servidores de aplicaciones empleados para desarrollo y prueba de aplicaciones.

Resumen

- Los componentes de software facilitan el desarrollo de sistemas complejos mediante un diseño modular que permite reducir la complejidad global del sistema. Los componentes de software se pueden desarrollar y probar de manera independiente, y se distribuyen como paquetes independientes.
- En torno al lenguaje Java se ha desarrollado una plataforma de computación, de la que existen dos versiones o variantes: Java SE y Java EE. La especificación Java EE incluye, además de una biblioteca de clases ampliada para dar soporte a aplicaciones empresariales, un servidor de aplicaciones con contenedores para desplegar distintos tipos de elementos.
- El modelo de componentes de Java SE está basado en JavaBeans. Estas son clases de Java que cumplen determinados requisitos formales.
- El modelo de componentes de Java EE está basado en EJB.
- Para el desarrollo de aplicaciones web se suele emplear el modelo MVC. El modelo se implementa mediante JavaBeans o mediante EJB. En el primer caso, se necesita un contenedor de servlets tal como Apache Tomcat. En el segundo, se necesita un servidor Java EE completo, incluyendo contenedores para EJB, tal como GlassFish. La vista se implementa mediante JSP, y el controlador, mediante servlets.
- En los JSP se pueden utilizar JavaBeans para recopilar los valores para distintos parámetros de una petición HTTP, relativos al mismo objeto.
- Para la persistencia de JavaBeans se puede utilizar ORM mediante la API nativa de Hibernate o mediante JPA.
- Los EJB de sesión sin estado, o stateless session beans, se pueden utilizar como DAO que proporcionan una interfaz de alto nivel para el acceso a datos mediante clases de entidad. Los EJB sincronizan el acceso a los datos y proporcionan persistencia de objetos mediante JPA y transacciones mediante JTA.



Ejercicios propuestos

Se propone una aplicación web según el modelo MVC para gestionar los proyectos que desarrolla una empresa, sus empleados y las asignaciones de empleados a proyectos. Se propone realizar una aplicación para Java SE, utilizando JavaBeans, y una aplicación análoga para Java EE, utilizando session beans sin estado.

Se proporciona el diagrama entidad-relación para describir las entidades y las relaciones entre ellas, y las sentencias SQL necesarias para crear las tablas en MySQL en una nueva base de datos y proporcionar acceso a ellas a un nuevo usuario. Se puede utilizar otra base de datos distinta de MySQL, preferentemente Oracle.

```
CREATE DATABASE proyectos_empresa;
USE proyectos_empresa;
CREATE TABLE empleado{
```

```

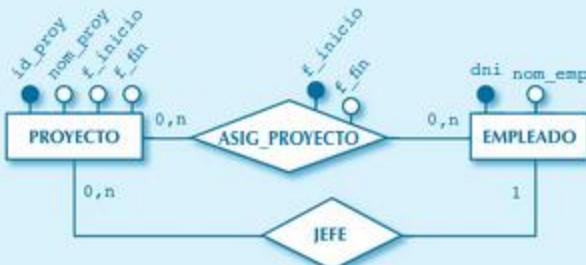
    id_emp INTEGER AUTO_INCREMENT NOT NULL,
    dni CHAR(9) NOT NULL,
    nom_emp VARCHAR(32) NOT NULL,
    PRIMARY KEY(id_emp)
};

CREATE TABLE proyecto(
    id_proy INTEGER AUTO_INCREMENT NOT NULL,
    nom_proy VARCHAR(32) NOT NULL,
    f_inicio DATE NOT NULL,
    f_fin DATE,
    dni_jefe_proy CHAR(9) NOT NULL,
    PRIMARY KEY(id_proy),
    FOREIGN KEY fk_proy_jefe(dni_jefe_proy) REFERENCES empleado(dni)
);

CREATE TABLE asig_proyecto(
    id_emp INTEGER NOT NULL,
    id_proy INTEGER NOT NULL,
    f_inicio DATE NOT NULL,
    f_fin DATE,
    PRIMARY KEY(id_emp, id_proy, f_inicio),
    FOREIGN KEY f_asig_emp(id_emp) REFERENCES empleado(id_emp),
    FOREIGN KEY f_asig_proy(id_proy) REFERENCES PROYECTOS(id_proy)
);

CREATE USER 'proyectos_empresa'@'localhost' IDENTIFIED BY '(password)';
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, EXECUTE
ON proyectos_empresa.* TO 'proyectos_empresa'@'localhost';

```



Para cada una de las dos aplicaciones, se propone desarrollarla siguiendo estos pasos:

1. Listado de empleados. Creación, recuperación, modificación y borrado de empleados.
2. Listado de proyectos. Creación, recuperación, modificación y borrado de proyectos.
3. Creación de asignaciones de empleados a proyectos.
4. Visualización de datos de un empleado, incluyendo lista de asignaciones a proyectos. Enlace disponible en el listado de empleados.
5. Visualización de datos de un proyecto, incluyendo lista de empleados asignados al proyecto. Enlace disponible en el listado de proyectos. Mostrar todas las asignaciones, aunque

hayan concluido. Las asignaciones concluidas se deben mostrar con una apariencia distinta, por ejemplo, sobre fondo gris.

6. Borrado de asignaciones de empleados a proyectos, desde un enlace mostrado en el listado anterior, solo mostrado para asignaciones con fecha de inicio en el futuro.
7. Modificación de asignaciones de empleados a proyectos, desde un enlace mostrado en el listado anterior, solo mostrado para asignaciones vigentes (no concluidas). Debe poderse cambiar la fecha de inicio si está en el futuro, y la fecha de fin siempre, pero no debe poder ponerse la fecha de fin en el pasado. Si así se hiciera, se mostraría un mensaje de error.
8. Por último, como mejora, se podría crear una página JSP específica para gestión de errores. Como se indicó en el apartado correspondiente, habría que utilizar `<%@page errorPage="página_de_error" %>`, `<%@page isErrorPage="true" %>`, y la variable implícita `exception`.

ACTIVIDADES DE AUTOEVALUACIÓN

1. Los componentes de software:

- a) Facilitan el desarrollo de sistemas modulares de software.
- b) Se empaquetan y distribuyen de manera independiente.
- c) Cumplen los requisitos establecidos en un modelo de componentes.
- d) Todo lo anterior es cierto.

2. Los JavaBeans:

- a) Solo sirven para desarrollar componentes visuales para interfaces gráficas de usuario.
- b) No se pueden utilizar en la plataforma Java EE.
- c) Son la base del modelo de componentes de Java SE.
- d) Solo tienen utilidad para el desarrollo de aplicaciones web.

3. Las propiedades de los JavaBeans:

- a) Deben ser públicas para que las puedan modificar las aplicaciones cliente.
- b) Pueden ser simples, compuestas o indexadas.
- c) Deben definirse como privadas y ser accesibles con funciones `getter` y `setter`.
- d) Ninguna de las opciones anteriores es cierta.

4. Los EJB:

- a) Se pueden desplegar en un servidor Apache Tomcat.
- b) Forman parte de la plataforma Java SE.
- c) No necesitan de ningún servidor de aplicaciones, se pueden desplegar en un contenedor autónomo.
- d) Son un modelo de componentes para la plataforma Java EE.

5. El *context path*:

- a) Es un tipo particular de URL utilizada en la plataforma Java EE.
- b) Es una parte de una URL que identifica una aplicación web.
- c) Identifica un *servlet* desplegado en un servidor de *servlets*.
- d) Debe obligatoriamente redirigir a un *servlet* controlador.

6. Un JSP:

- a) Solo puede contener etiquetas de HTML.
- b) Debe realizar su tarea lo más rápidamente posible y ceder el control al *servlet* controlador.
- c) Puede utilizar JavaBeans para recopilar los valores que se le pasan en parámetros de la petición HTTP.
- d) Necesita un contenedor de EJB para funcionar.

7. Los *servlets*:

- a) Se despliegan en un servidor de *servlets* como por ejemplo, Apache Tomcat.
- b) Son accesibles mediante el protocolo HTTP o HTTPS.
- c) Se utilizan para implementar páginas JSP.
- d) Todo lo anterior es cierto.

8. Un EJB de sesión:

- a) No permite el acceso simultáneo por más de un *thread* (hebra de ejecución).
- b) Se puede utilizar para implementar una clase de tipo *Facade* para el acceso a las clases de entidad.
- c) Se puede utilizar como objeto DAO para el acceso a los datos representados por las clases de entidad.
- d) Todo lo anterior es cierto.

9. JPA:

- a) Es un componente opcional de la plataforma Java EE.
- b) Es una alternativa a Hibernate para persistencia en la plataforma Java EE.
- c) Utiliza anotaciones para las clases de Java.
- d) Solo se puede utilizar en la plataforma Java EE, no con Java SE.

10. Una aplicación web MVC para la plataforma Java:

- a) Puede empaquetarse en un fichero de tipo WAR.
- b) Utiliza JavaBeans para el modelo.
- c) Normalmente necesita un servidor de *servlets* para el controlador y para la vista.
- d) Todas las respuestas anteriores son correctas.

SOLUCIONES:

1. a b c d

2. a b c d

3. a b c d

4. a b c d

5. a b c d

6. a b c d

7. a b c d

8. a b c d

9. a b c d

10. a b c d