

Ejercicios resueltos de Acceso a Datos

Los 24 ejercicios de Acceso a Datos de la web, tema a tema: cada uno con su enunciado, los datos que necesitas (código de partida, ejemplos de entrada y salida o la base de datos) y, al final, la solución explicada.

Índice

Manejo de Ficheros

[1. Media de notas desde un fichero CSV](#)

[3. Leer un XML con DOM](#)

[5. Un CSV de verdad: comillas, separadores y saltos de línea](#)

[2. Serializar una lista de productos](#)

[4. Acceso aleatorio a registros](#)

[6. Leer un formato binario propio con versión y CRC](#)

Conectores JDBC

[7. Sentencias preparadas frente a concatenación](#)

[9. Transacciones con puntos de guardado y bloqueos](#)

[8. Un pool de conexiones por dentro](#)

Mapeo Objeto-Relacional (Hibernate/JPA)

[10. El problema N+1 y las estrategias de carga](#)

[11. Los estados de una entidad JPA](#)

BD Objeto-Relacionales y Orientadas a Objetos

[12. Arrays y tipos compuestos de PostgreSQL](#)

[14. Herencia de tablas en PostgreSQL](#)

[13. Profundidad de activación en una base de objetos](#)

Componentes de Acceso a Datos

[15. Propiedad ligada \(bound property\).](#)

[17. Persistencia de un JavaBean en XML](#)

[19. Un contenedor de inyección de dependencias](#)

[16. Propiedad restringida \(constrained\).](#)

[18. Componente DAO con eventos](#)

Ejercicios de Acceso a Datos

[20. Filtrar un fichero de log.](#)

[22. Media de precios por categoría \(DOM\).](#)

[21. Conversor de CSV a JSON](#)

Ejercicios largos

[23. Conversor de CSV a JSON con comillas y tipos](#)

[24. Fichero de productos con registros de tamaño fijo y acceso directo](#)

Manejo de Ficheros

1. Media de notas desde un fichero CSV

Fácil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

El programa guarda en notas.csv las líneas recibidas por teclado (formato nombre;nota). Completa leerMedia() para que lea el fichero línea a línea con BufferedReader y devuelva la media de las notas. Se imprime con dos decimales.

Código de partida (java)

```
import java.io.*;
import java.util.*;

public class Main {

    static double leerMedia(File fichero) throws IOException {
        double suma = 0;
        int contador = 0;
        // TODO: lee el fichero línea a línea y acumula las notas

        return contador == 0 ? 0 : suma / contador;
    }

    public static void main(String[] args) throws IOException {
        // Guarda en notas.csv las líneas recibidas por teclado
        File fichero = new File("notas.csv");
        Scanner sc = new Scanner(System.in);
        try (PrintWriter pw = new PrintWriter(new FileWriter(fichero))) {
            while (sc.hasNextLine()) pw.println(sc.nextLine());
        }

        System.out.printf(Locale.US, "Media: %.2f\n", leerMedia(fichero));
    }
}
```

Ejemplo: Tres alumnos

ENTRADA

Ana;8
Luis;6
Marta;9.5

SALIDA ESPERADA

Media: 7.83

Ejemplo: Un alumno

ENTRADA

Pedro;5

SALIDA ESPERADA

Media: 5.00

2. Serializar una lista de productos

Medio · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Completa guardar() y cargar() para escribir la lista de productos en productos.obj con ObjectOutputStream y volver a leerla con ObjectInputStream. El programa lee los productos por teclado (nombre;precio), los guarda, los carga de nuevo y muestra cuántos hay y el importe total.

Código de partida (java)

```
import java.io.*;
import java.util.*;

public class Main {

    static class Producto implements Serializable {
        private static final long serialVersionUID = 1L;
        String nombre;
        double precio;
        Producto(String nombre, double precio) { this.nombre = nombre; this.precio = precio; }
    }

    static void guardar(List<Producto> productos, File fichero) throws IOException {
        // TODO: escribe la lista en el fichero
    }

    @SuppressWarnings("unchecked")
    static List<Producto> cargar(File fichero) throws IOException, ClassNotFoundException {
        // TODO: lee la lista desde el fichero
        return new ArrayList<>();
    }

    public static void main(String[] args) throws Exception {
        List<Producto> productos = new ArrayList<>();
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().split(";");
            if (p.length == 2) productos.add(new Producto(p[0], Double.parseDouble(p[1])));
        }

        File fichero = new File("productos.obj");
        guardar(productos, fichero);
        List<Producto> leidos = cargar(fichero);

        double total = 0;
        for (Producto p : leidos) total += p.precio;
        System.out.printf(Locale.US, "%d productos, total %.2f%n", leidos.size(), total);
    }
}
```

Ejemplo: Tres productos

ENTRADA

Teclado;49.99
Ratón;19.5
Monitor;189

SALIDA ESPERADA

3 productos, total 258.49

3. Leer un XML con DOM

Medio · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

El programa guarda en biblioteca.xml el documento recibido por teclado. Completa procesar() para cargarlo con DOM, recorrer los elementos <libro> y mostrar cuántos libros hay y cuántos están disponibles (atributo disponible="si").

Código de partida (java)

```
import java.io.*;
import java.nio.file.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Main {

    static void procesar(File fichero) throws Exception {
        int total = 0, disponibles = 0;
        // TODO: carga el XML con DOM y cuenta los libros

        System.out.println("Libros: " + total + ", disponibles: " + disponibles);
    }

    public static void main(String[] args) throws Exception {
        File fichero = new File("biblioteca.xml");
        Files.write(fichero.toPath(), System.in.readAllBytes());
        procesar(fichero);
    }
}
```

Ejemplo: Tres libros, dos disponibles

ENTRADA

```
<biblioteca><libro disponible="si">
<titulo>Java</titulo></libro><libro
disponible="no"><titulo>SQL</titulo></libro>
<libro disponible="si"><titulo>XML</titulo>
</libro></biblioteca>
```

SALIDA ESPERADA

```
Libros: 3, disponibles: 2
```

4. Acceso aleatorio a registros

Difícil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

El programa crea empleados.dat con registros de tamaño fijo: id (int, 4 bytes), nombre (10 caracteres = 20 bytes) y salario (double, 8 bytes). Completa leerRegistro() para que use seek() y lea directamente el registro número n (empezando en 1) sin recorrer los anteriores. Se lee n por teclado.

Código de partida (java)

```
import java.io.*;
import java.util.*;

public class Main {
    static final int TAM = 4 + 20 + 8; // id + nombre(10 chars) + salario

    static void escribir(RandomAccessFile raf, int id, String nombre, double salario) throws IOException
    {
        raf.writeInt(id);
        StringBuilder sb = new StringBuilder(nombre);
        sb.setLength(10); // rellena o recorta a 10 caracteres
        raf.writeChars(sb.toString());
        raf.writeDouble(salario);
    }

    static String leerRegistro(RandomAccessFile raf, int n) throws IOException {
        // TODO: sitúate en el registro n y lee sus tres campos
        return "";
    }

    public static void main(String[] args) throws IOException {
        try (RandomAccessFile raf = new RandomAccessFile("empleados.dat", "rw")) {
            raf.setLength(0);
            escribir(raf, 1, "Ana", 1850.5);
            escribir(raf, 2, "Luis", 2100);
            escribir(raf, 3, "Marta", 2475.25);
            escribir(raf, 4, "Jorge", 1990);

            int n = new Scanner(System.in).nextInt();
            System.out.println(leerRegistro(raf, n));
        }
    }
}
```

Ejemplo: Registro 1

ENTRADA

1

SALIDA ESPERADA

1 Ana 1850.5

Ejemplo: Registro 3

ENTRADA

3

SALIDA ESPERADA

3 Marta 2475.25

5. Un CSV de verdad: comillas, separadores y saltos de línea

Difícil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Partir cada línea por el separador falla con CSV reales: un campo entre comillas dobles puede contener el separador, saltos de línea y comillas (escritas dos veces: ""). Lee todo el texto (separador ; como el Excel en español) y analízalo carácter a carácter según el RFC 4180: fuera de comillas, ; separa campos y el salto de línea termina el registro (las líneas vacías se ignoran); una comilla al principio abre un campo entrecomillado, dentro del cual "" es una comilla y " cierra. La primera fila es la cabecera. Para cada registro de datos con tantos campos como la cabecera, muestra «Registro i:» y una línea « cabecera = valor» por campo (los saltos de línea dentro del valor se muestran como \n y un valor vacío como «(vacío)»); una comilla en mitad de un campo sin comillas es un carácter normal; si el número de campos no coincide, «Error en el registro i: tiene X campos y se esperaban M». Termina con «Registros válidos: V de T · Campos: M». Si el texto acaba con una comilla sin cerrar: «Error: comilla sin cerrar en el registro i» (i = registros completos antes de ese) y nada más.

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in).useDelimiter("\\A");
        String texto = sc.hasNext() ? sc.next().replace("\r", "") : "";
        List<List<String>> registros = new ArrayList<>();
        // TODO: analizar el CSV carácter a carácter (separador ; y comillas dobles)
    }
}
```

Ejemplo: Alumnos con campos difíciles

ENTRADA

```
nif;nombre;direccion;nota
12345678Z;"García; Ana";"Calle Mayor 3
2º B";8,5
87654321X;Luis "el rápido";Avenida Sol 1;7
X1234567L;"Pérez, ""Lolo""";
00000000T;Falta;un campo
```

SALIDA ESPERADA

```
Registro 1:
nif = 12345678Z
nombre = García; Ana
direccion = Calle Mayor 3
2º B
nota = 8,5
Registro 2:
nif = 87654321X
nombre = Luis "el rápido"
direccion = Avenida Sol 1
nota = 7
Registro 3:
nif = X1234567L
nombre = Pérez, "Lolo"
direccion = (vacío)
nota = (vacío)
Error en el registro 4: tiene 3 campos y se
esperaban 4
Registros válidos: 3 de 4 · Campos: 4
```

6. Leer un formato binario propio con versión y CRC

Muy difícil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Muchas aplicaciones guardan sus datos en un formato binario propio. Este guarda alumnos así (todo en big-endian, como `DataOutputStream`): 4 bytes ASCII «ALUM»; un short con la versión (1 o 2); un int con el número de registros; por cada registro, un int id, un short con la longitud en bytes del nombre seguido del nombre en UTF-8, en la versión 2 un byte con el curso, y un float con la nota; y al final un int con el CRC32 de todos los bytes anteriores. La entrada es el fichero en hexadecimal. Comprueba en este orden: la firma (si no es ALUM, «Formato desconocido»), la versión («Versión X no soportada»), que no se acaben los bytes antes de tiempo («Fichero truncado») y el CRC («CRC incorrecto: esperado XXXXXXXX, calculado YYYYYYYY», en hexadecimal con 8 cifras). Si todo es correcto, muestra cada registro como «id · nombre · nota» (en la versión 2, «id · nombre · curso N · nota»), con la nota con dos decimales, y «Registros: n · media M · versión v · CRC OK».

Código de partida (java)

```
import java.nio.*;
import java.util.*;
import java.util.zip.CRC32;

public class Main {
    static byte[] desdeHex(String h) {
        byte[] b = new byte[h.length() / 2];
        for (int i = 0; i < b.length; i++) b[i] = (byte) Integer.parseInt(h.substring(2 * i, 2 * i + 2),
            16);
        return b;
    }

    public static void main(String[] args) {
        byte[] datos = desdeHex(new Scanner(System.in).nextLine().trim());
        ByteBuffer buf = ByteBuffer.wrap(datos); // big-endian, como DataInputStream
        // TODO: leer cabecera, registros y CRC
    }
}
```

Ejemplo: Versión 1 correcta

ENTRADA

```
414C554D000100000000300000001000B416E612047617263C
3AD614108000000000002000B49C3B1616B69205275697A40
C800000000003000C5A6FC3AB204D617274C3AD6E411C000
054FEC7DF
```

SALIDA ESPERADA

```
1 · Ana García · 8.50
2 · Iñaki Ruiz · 6.25
3 · Zoë Martín · 9.75
Registros: 3 · media 8.17 · versión 1 · CRC OK
```

Ejemplo: Versión 2 con curso

ENTRADA

```
414C554D00020000000020000000A00064C7563C3AD610140F
000000000000B000C4A6F73CA9204D6172C3AD610240A000
0027FC88E5
```

SALIDA ESPERADA

```
10 · Lucía · curso 1 · 7.50
11 · José María · curso 2 · 5.00
Registros: 2 · media 6.25 · versión 2 · CRC OK
```

Ejemplo: CRC alterado

ENTRADA

```
414C554D000100000000300000001000B416E612047617263C
3AD614108000000000002000B49C3B1616B69205275697A40
C800000000003000C5A6FC3AB204D617274C3AD6E411C000
054FEC7DE
```

SALIDA ESPERADA

```
CRC incorrecto: esperado 54FEC7DE, calculado
54FEC7DF
```

Conectores JDBC

7. Sentencias preparadas frente a concatenación

Medio · Conectores JDBC · apuntesdam.com/subject/acceso-datos/topic/jdbc

Un PreparedStatement envía la sentencia y los datos por separado, así que un dato nunca puede convertirse en código SQL. Para verlo, simula lo que pasaría al concatenar. La primera línea es la plantilla con ? como marcadores; cada línea siguiente es un parámetro «tipo:valor» con tipo s (texto), i (entero), d (decimal con punto), b (booleano), f (fecha) o n (nulo). Los ? dentro de un literal entre comillas simples de la plantilla no son marcadores. Muestra «Parámetros: N» y, si no coincide con los dados, «Error: la sentencia tiene N parámetros y se han dado M». Valida que los enteros y decimales lo sean («Error: el parámetro k no es un entero: x» / «no es un decimal»). Después muestra «Concatenando: ...» (cada texto entre comillas tal cual) y «Preparada: ...» (con las comillas simples del texto duplicadas, que es como

queda el dato escapado); los booleanos son TRUE/FALSE, las fechas DATE '...' y los nulos NULL. Termina con «¡Inyección! Con la concatenación el texto del usuario cambia la estructura de la consulta» si algún texto contiene una comilla simple, o «Las dos versiones son equivalentes con estos datos».

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String plantilla = sc.nextLine();
        List<String[]> params = new ArrayList<>();           // {tipo, valor}
        while (sc.hasNextLine()) {
            String l = sc.nextLine();
            if (l.isBlank()) continue;
            int dos = l.indexOf(':');
            params.add(new String[] { l.substring(0, dos), l.substring(dos + 1) });
        }
        // TODO: localizar los ? fuera de los literales y montar las dos versiones
    }
}
```

Ejemplo: Inicio de sesión atacado

ENTRADA

```
SELECT * FROM usuario WHERE login = ? AND clave = ?
s:admin
s:' OR '1'='1'
```

SALIDA ESPERADA

Parámetros: 2
Concatenando: SELECT * FROM usuario WHERE login = 'admin' AND clave = '' OR '1'='1'
Preparada: SELECT * FROM usuario WHERE login = 'admin' AND clave = '' OR ''='1'
¡Inyección! Con la concatenación el texto del usuario cambia la estructura de la consulta

Ejemplo: Tipos y ? en un literal

ENTRADA

```
UPDATE alumno SET nota = ?, aprobado = ?,
nacimiento = ?, tutor = ? WHERE nombre = ? AND
obs <> '¿Qué?'
d:7.5
b:true
f:2005-03-14
n:
s:O'Brien
```

SALIDA ESPERADA

Parámetros: 5
Concatenando: UPDATE alumno SET nota = 7.5, aprobado = TRUE, nacimiento = DATE '2005-03-14', tutor = NULL WHERE nombre = 'O'Brien' AND obs <> '¿Qué?'
Preparada: UPDATE alumno SET nota = 7.5, aprobado = TRUE, nacimiento = DATE '2005-03-14', tutor = NULL WHERE nombre = 'O'Brien' AND obs <> '¿Qué?'
¡Inyección! Con la concatenación el texto del usuario cambia la estructura de la consulta

8. Un pool de conexiones por dentro

Difícil · Conectores JDBC · apuntesdam.com/subject/acceso-datos/topic/jdbc

Abrir una conexión a la base de datos cuesta decenas de milisegundos, así que las aplicaciones usan un pool (HikariCP, el de Tomcat...) que las reutiliza. La primera línea es «max M espera W» y las siguientes, ordenadas por tiempo, «t pedir cliente» o «t devolver cliente». Al pedir: si hay una conexión libre se entrega la que lleva más tiempo libre («t: X recibe la conexión #k (reutilizada)»); si no, y hay menos de M creadas en total, se crea una nueva numerada desde 1 («(nueva)»); si no, el cliente espera en una cola FIFO («t: X espera (pool lleno)»). Al devolver: si alguien espera, la conexión pasa al primero de la cola («t: X devuelve #k → pasa a Y (esperó d ms)»); si no, queda libre («t: X devuelve #k (queda libre)»); si el cliente no tenía conexión, «t: Error: X no tiene conexión». Quien lleva esperando más de W ms (su inicio + W es menor que el instante del siguiente evento) se va antes de procesar ese evento: «inicio+W: X

agota la espera (timeout)»; al terminar los eventos se agotan los que quedan. Termina con «Conexiones creadas: c · Reutilizadas: r · Esperas: e · Timeouts: o · Espera media: m ms» (media entera de las esperas que sí recibieron conexión; 0 si ninguna) y «Al final: c abiertas, l libres, u en uso».

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] conf = sc.nextLine().trim().split(" ");    // «max M espera W»
        int max = Integer.parseInt(conf[1]), espera = Integer.parseInt(conf[3]);
        List<String[]> eventos = new ArrayList<>();          // {tiempo, accion, cliente}
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (!l.isEmpty()) eventos.add(l.split(" "));
        }
        // TODO: simular el pool (conexiones libres, en uso, cola de espera y tiempos límite)
    }
}
```

Ejemplo: Aplicación web

ENTRADA

```
max 2 espera 100
0 pedir web1
5 pedir web2
10 pedir web3
40 devolver web1
45 pedir web4
50 devolver web3
60 pedir web5
70 devolver web2
80 devolver web5
90 pedir web7
200 pedir web6
210 devolver web9
320 devolver web4
330 pedir web8
340 pedir web10
```

SALIDA ESPERADA

```
0: web1 recibe la conexión #1 (nueva)
5: web2 recibe la conexión #2 (nueva)
10: web3 espera (pool lleno)
40: web1 devuelve #1 → pasa a web3 (esperó 30 ms)
45: web4 espera (pool lleno)
50: web3 devuelve #1 → pasa a web4 (esperó 5 ms)
60: web5 espera (pool lleno)
70: web2 devuelve #2 → pasa a web5 (esperó 10 ms)
80: web5 devuelve #2 (queda libre)
90: web7 recibe la conexión #2 (reutilizada)
200: web6 espera (pool lleno)
210: Error: web9 no tiene conexión
300: web6 agota la espera (timeout)
320: web4 devuelve #1 (queda libre)
330: web8 recibe la conexión #1 (reutilizada)
340: web10 espera (pool lleno)
440: web10 agota la espera (timeout)
Conexiones creadas: 2 · Reutilizadas: 2 ·
Esperas: 5 · Timeouts: 2 · Espera media: 15 ms
Al final: 2 abiertas, 0 libres, 2 en uso
```

9. Transacciones con puntos de guardado y bloqueos

Muy difícil · Conectores JDBC · apuntesdam.com/subject/acceso-datos/topic/jdbc

Simula dos conexiones JDBC (sesiones A y B) sobre unas cuentas, como haría una base de datos con nivel de aislamiento READ COMMITTED. La primera línea es «cuentas C1=v1 C2=v2 ...» y cada línea siguiente «S orden»: BEGIN (abre una transacción: setAutoCommit(false); si ya hay una, «S: ya hay una transacción abierta»); SELECT c («S ve c = v»: el valor propio pendiente si la sesión lo ha cambiado en su transacción y, si no, el confirmado: nunca ve lo no confirmado de la otra); UPDATE c ±n (si la otra sesión tiene c cambiada en una transacción abierta, «S: UPDATE c bloqueado por O (se cancela la sentencia)»; si el saldo quedaría negativo, «S: UPDATE c rechazado: el saldo quedaría en v»; si no, sin transacción se confirma al momento «S: c = v (confirmado)» y dentro de una, «S: c = v»); SAVEPOINT nombre; ROLLBACK TO nombre (vuelve al estado del punto y borra los puntos posteriores; «S: vuelta a nombre» o «S: no existe el punto de guardado nombre»); COMMIT («S: COMMIT (n cambios)», n = cuentas distintas cambiadas) y ROLLBACK («S: ROLLBACK (n cambios descartados)»). COMMIT, ROLLBACK y SAVEPOINT sin transacción: «S: no hay transacción». Un UPDATE rechazado no deshace la transacción. Al final, por cada sesión con una transacción abierta (A y luego B), «S tenía una transacción abierta: se deshace al cerrar la conexión», y «Confirmado: C1=v1 ...» en el orden inicial.

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, Integer> confirmado = new LinkedHashMap<>(); // lo que ve cualquiera
        for (String par : sc.nextLine().trim().split(" ")) {
            if (!par.contains("=")) continue; // «cuentas A1=100 A2=50 ...»
            String[] kv = par.split("=");
            confirmado.put(kv[0], Integer.parseInt(kv[1]));
        }
        // Por sesión: cambios pendientes (null si no hay transacción) y puntos de guardado
        Map<String, Map<String, Integer>> pendientes = new HashMap<>();
        Map<String, LinkedHashMap<String, Map<String, Integer>>> puntos = new HashMap<>();
        // TODO: procesar las órdenes de las sesiones A y B
    }
}
```

Ejemplo: Transferencia con comisión

ENTRADA

```
cuentas A1=100 A2=50 A3=0
A BEGIN
A UPDATE A1 -30
A UPDATE A2 +30
B SELECT A1
A SELECT A1
B UPDATE A1 +5
B UPDATE A3 +10
A SAVEPOINT antes_comision
A UPDATE A2 -100
A UPDATE A3 -2
A ROLLBACK TO antes_comision
A UPDATE A2 -1
A COMMIT
B SELECT A1
B UPDATE A1 +5
B BEGIN
B UPDATE A2 -79
B SELECT A2
A SELECT A2
```

SALIDA ESPERADA

```
A: BEGIN
A: A1 = 70
A: A2 = 80
B ve A1 = 100
A ve A1 = 70
B: UPDATE A1 bloqueado por A (se cancela la
sentencia)
B: A3 = 10 (confirmado)
A: SAVEPOINT antes_comision
A: UPDATE A2 rechazado: el saldo quedaría en -20
A: A3 = 8
A: vuelta a antes_comision
A: A2 = 79
A: COMMIT (2 cambios)
B ve A1 = 70
B: A1 = 75 (confirmado)
B: BEGIN
B: A2 = 0
B ve A2 = 0
A ve A2 = 79
B tenía una transacción abierta: se deshace al
cerrar la conexión
Confirmado: A1=75 A2=79 A3=10
```

Mapecto Objeto-Relacional (Hibernate/JPA)

10. El problema N+1 y las estrategias de carga

Medio · Mapecto Objeto-Relacional (Hibernate/JPA) · apuntesdam.com/subject/acceso-datos/topic/hibernate

Con JPA, una consulta inocente como «SELECT a FROM Autor a» seguida de a.getLibros() en un bucle puede lanzar una consulta por autor: el problema N+1. Simula lo que hace Hibernate. La entrada tiene líneas «autor id nombre», «libro id idAutor título» y «estrategia E». Se ejecuta la consulta de autores (ordenados por id) y después, para cada autor, se accede a sus libros (ordenados por id) y se muestra «Nombre: n libros (Título, Título)» («1 libro», o «0 libros» sin paréntesis). Muestra cada SQL en el momento en que se lanza: «SQL: select * from autor order by id» al principio y, según la estrategia: LAZY, «SQL: select * from libro where autor_id = X» justo antes de usar los libros de cada autor; EAGER, esas mismas consultas pero todas seguidas tras la primera (con JPQL, EAGER no evita el N+1); JOIN FETCH, una única consulta «SQL: select * from autor a left join libro l on l.autor_id = a.id order by a.id» que lo trae todo;

BATCH k, al necesitar los libros de un autor no cargado se cargan los de hasta k autores no cargados a partir de él: «SQL: select * from libro where autor_id in (1, 2)»; SUBSELECT, la primera vez se cargan los de todos: «SQL: select * from libro where autor_id in (select id from autor)». Termina con «Consultas: n (estrategia E)».

Código de partida (java)

```
import java.util.*;

public class Main {
    record Autor(int id, String nombre) { }
    record Libro(int id, int autorId, String titulo) { }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<Autor> autores = new ArrayList<>();
        List<Libro> libros = new ArrayList<>();
        String estrategia = "LAZY";
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.startsWith("autor ")) {
                String[] p = l.split(" ", 3);
                autores.add(new Autor(Integer.parseInt(p[1]), p[2]));
            } else if (l.startsWith("libro ")) {
                String[] p = l.split(" ", 4);
                libros.add(new Libro(Integer.parseInt(p[1]), Integer.parseInt(p[2]), p[3]));
            } else if (l.startsWith("estrategia ")) {
                estrategia = l.substring(11).trim();
            }
        }
        autores.sort(Comparator.comparingInt(Autor::id));
        libros.sort(Comparator.comparingInt(Libro::id));
        // TODO: simular «SELECT a FROM Autor a» y el acceso a a.getLibros() de cada autor
    }
}
```

Ejemplo: Estrategia LAZY

ENTRADA

```
autor 1 Cervantes
autor 2 Galdós
autor 3 Emilia Pardo Bazán
autor 4 Machado
autor 5 Lorca
libro 10 1 Don Quijote
libro 11 2 Fortunata y Jacinta
libro 12 1 Novelas ejemplares
libro 13 3 Los pazos de Ulloa
libro 14 5 Bodas de sangre
libro 15 5 Yerma
libro 16 2 Misericordia
estrategia LAZY
```

SALIDA ESPERADA

```
SQL: select * from autor order by id
SQL: select * from libro where autor_id = 1
Cervantes: 2 libros (Don Quijote, Novelas ejemplares)
SQL: select * from libro where autor_id = 2
Galdós: 2 libros (Fortunata y Jacinta, Misericordia)
SQL: select * from libro where autor_id = 3
Emilia Pardo Bazán: 1 libro (Los pazos de Ulloa)
SQL: select * from libro where autor_id = 4
Machado: 0 libros
SQL: select * from libro where autor_id = 5
Lorca: 2 libros (Bodas de sangre, Yerma)
Consultas: 6 (estrategia LAZY)
```

Ejemplo: Estrategia JOIN FETCH

ENTRADA

```
autor 1 Cervantes
autor 2 Galdós
autor 3 Emilia Pardo Bazán
autor 4 Machado
autor 5 Lorca
libro 10 1 Don Quijote
libro 11 2 Fortunata y Jacinta
libro 12 1 Novelas ejemplares
libro 13 3 Los pazos de Ulloa
libro 14 5 Bodas de sangre
libro 15 5 Yerma
libro 16 2 Misericordia
estrategia JOIN FETCH
```

SALIDA ESPERADA

```
SQL: select * from autor a left join libro l on
l.autor_id = a.id order by a.id
Cervantes: 2 libros (Don Quijote, Novelas
ejemplares)
Galdós: 2 libros (Fortunata y Jacinta,
Misericordia)
Emilia Pardo Bazán: 1 libro (Los pazos de Ulloa)
Machado: 0 libros
Lorca: 2 libros (Bodas de sangre, Yerma)
Consultas: 1 (estrategia JOIN FETCH)
```

Ejemplo: Estrategia BATCH 2

ENTRADA

```
autor 1 Cervantes
autor 2 Galdós
autor 3 Emilia Pardo Bazán
autor 4 Machado
autor 5 Lorca
libro 10 1 Don Quijote
libro 11 2 Fortunata y Jacinta
libro 12 1 Novelas ejemplares
libro 13 3 Los pazos de Ulloa
libro 14 5 Bodas de sangre
libro 15 5 Yerma
libro 16 2 Misericordia
estrategia BATCH 2
```

SALIDA ESPERADA

```
SQL: select * from autor order by id
SQL: select * from libro where autor_id in (1, 2)
Cervantes: 2 libros (Don Quijote, Novelas
ejemplares)
Galdós: 2 libros (Fortunata y Jacinta,
Misericordia)
SQL: select * from libro where autor_id in (3, 4)
Emilia Pardo Bazán: 1 libro (Los pazos de Ulloa)
Machado: 0 libros
SQL: select * from libro where autor_id in (5)
Lorca: 2 libros (Bodas de sangre, Yerma)
Consultas: 4 (estrategia BATCH 2)
```

11. Los estados de una entidad JPA

Difícil · Mapeo Objeto-Relacional (Hibernate/JPA) · apuntesdam.com/subject/acceso-datos/topic/hibernate

En JPA una entidad puede estar NUEVA (transient), GESTIONADA (managed: el EntityManager vigila sus cambios), SEPARADA (detached) o ELIMINADA (removed), y el SQL no se lanza cuando llamas a los métodos sino en el flush. Simula un EntityManager con una única tabla por entidad e ids IDENTITY (1, 2, 3...). Órdenes: «new v Clase c1=v1,c2=v2» («v: NUEVA (transient)»); «persist v» (NUEVA → inserta al momento: «SQL: insert into clase (c1, c2) values ('v1', 'v2') → id N» y pasa a GESTIONADA; si ya lo está, «v ya está gestionada: no se hace nada»; si está separada, «PersistentObjectException: v está separada (usa merge)»); «set v campo=valor» (si no está gestionada, «v (estado): el cambio solo está en memoria»); «find v Clase id» (si esa fila ya está gestionada en el contexto, se devuelve la misma instancia sin SQL: «v es la misma instancia que w (caché de primer nivel, sin SQL)», con w la primera variable por orden alfabético que la apunta; si no, «SQL: select * from clase where id = N» y «v: GESTIONADA {campos}» o «v = null»); «detach v»; «merge v w» (copia el estado de w en la instancia gestionada con su id, cargándola con un select si no está en el contexto; w no cambia: «v: copia gestionada de w (w sigue separada)»); «remove v» (solo gestionadas: «v: ELIMINADA (se borrará en el flush)»; si no, «IllegalArgumentException: v no está gestionada»); «flush» y «commit» (recorren el contexto en orden: las eliminadas lanzan «SQL: delete ...» y dejan de estar en el contexto; las gestionadas con campos distintos de su copia del último flush lanzan «SQL: update clase set c = 'v', ... where id = N» solo con los campos cambiados; después «flush: n sentencias» o «COMMIT (n sentencias en el flush)», con «1 sentencia» en singular); «close» («Contexto cerrado: n entidades pasan a SEPARADA») y «estado v».

Código de partida (java)

```
import java.util.*;

public class Main {
    static class Entidad {
        Long id;
        String tabla;
        Map<String, String> campos = new LinkedHashMap<>();
        String estado = "NUEVA"; // NUEVA, GESTIONADA, SEPARADA, ELIMINADA
        Map<String, String> instantanea; // copia al pasar a gestionada o tras el último flush
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, Entidad> variables = new HashMap<>();
        Map<Long, Map<String, String>> baseDatos = new TreeMap<>();
        List<Entidad> contexto = new ArrayList<>(); // contexto de persistencia (en orden de entrada)
        long[] siguienteId = { 1 };
        while (sc.hasNextLine()) {
            String linea = sc.nextLine().trim();
            if (linea.isEmpty()) continue;
            String[] p = linea.split(" ", 3);
            // TODO: new, persist, set, find, detach, merge, remove, flush, commit, close y estado
        }
    }
}
```

Ejemplo: Ciclo de vida completo

ENTRADA

```
new e1 Alumno nombre=Ana,nota=6
persist e1
set e1 nota=8
set e1 nota=6
flush
set e1 nota=9
find e2 Alumno 1
commit
detach e1
set e1 nombre=Ana María
commit
persist e1
merge e3 e1
estado e3
commit
remove e1
remove e3
commit
find e4 Alumno 1
```

SALIDA ESPERADA

```
e1: NUEVA (transient)
SQL: insert into alumno (nombre, nota) values ('Ana', '6') → id 1
flush: 0 sentencias
e2 es la misma instancia que e1 (caché de primer nivel, sin SQL)
SQL: update alumno set nota = '9' where id = 1
COMMIT (1 sentencia en el flush)
e1: SEPARADA
e1 (separada): el cambio solo está en memoria
COMMIT (0 sentencias en el flush)
PersistentObjectException: e1 está separada (usa merge)
SQL: select * from alumno where id = 1
e3: copia gestionada de e1 (e1 sigue separada)
e3: GESTIONADA id=1 {nombre=Ana María, nota=9}
SQL: update alumno set nombre = 'Ana María' where id = 1
COMMIT (1 sentencia en el flush)
IllegalArgumentException: e1 no está gestionada
e3: ELIMINADA (se borrará en el flush)
SQL: delete from alumno where id = 1
COMMIT (1 sentencia en el flush)
SQL: select * from alumno where id = 1
e4 = null
```

12. Arrays y tipos compuestos de PostgreSQL

Medio · BD Objeto-Relacionales y Orientadas a Objetos · apuntesdam.com/subject/acceso-datos/topic/bd-objeto-relacionales

PostgreSQL permite columnas de tipo array (text[]) y tipos compuestos (CREATE TYPE direccion AS (calle text, cp text, ciudad text)). Si se leen por JDBC con getString, llegan en su formato de texto y hay que interpretarlo. Cada línea de la entrada es «array {...}» o «compuesto (...)»; implementa leerArray y leerCompuesto, que devuelven la lista de valores (null para los nulos). Array: elementos separados por comas; un elemento entre comillas dobles puede contener comas y espacios (los espacios de fuera de las comillas se ignoran) y dentro de él \\ escapa el carácter siguiente (\ es una comilla); sin comillas se quitan los espacios de alrededor y NULL (en cualquier combinación de mayúsculas) es nulo, pero "NULL" entre comillas es el texto NULL; {} es un array vacío. Compuesto: campos separados por comas; un campo vacío sin comillas es nulo y "" es una cadena vacía; entre comillas, "" o \ es una comilla; sin comillas el texto se toma tal cual. El main ya muestra cada resultado como «tipo ["a", null, ...] · n valores · nulos: k».

Código de partida (java)

```
import java.util.*;

public class Main {
    static String json(List<String> valores) {
        List<String> partes = new ArrayList<>();
        for (String v : valores)
            partes.add(v == null ? "null" : "\"" + v.replace("\\", "\\\\").replace("\"", "\\\"") + "\"");
        return "[" + String.join(", ", partes) + "]";
    }

    static List<String> leerArray(String texto) {
        // TODO: elementos separados por comas, comillas con escapes y NULL
        return new ArrayList<>();
    }

    static List<String> leerCompuesto(String texto) {
        // TODO: campos separados por comas, vacío = nulo y comillas dobladas
        return new ArrayList<>();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String l = sc.nextLine();
            if (l.isBlank()) continue;
            String tipo = l.substring(0, l.indexOf(' ')), texto = l.substring(l.indexOf(' ') + 1).trim();
            List<String> v = tipo.equals("array") ? leerArray(texto) : leerCompuesto(texto);
            System.out.println(tipo + " " + json(v) + " · " + v.size() + (v.size() == 1 ? " valor" : " valores")
                + " · nulos: " + v.stream().filter(Objects::isNull).count());
        }
    }
}
```

Ejemplo: Arrays y compuestos

ENTRADA

```
array {Lunes,Martes,"Fin de semana",NULL}  
array {}  
array {"NULL",null, 3 , "con \"comillas\" y  
\\barra"}  
compuesto ("Calle Mayor, 3",28013,Madrid,)  
compuesto (,"",abc)  
compuesto ("dijo \"hola\"",x)
```

SALIDA ESPERADA

```
array ["Lunes", "Martes", "Fin de semana", null]  
· 4 valores · nulos: 1  
array [] · 0 valores · nulos: 0  
array ["NULL", null, "3", "con \"comillas\" y  
\\barra"] · 4 valores · nulos: 1  
compuesto ["Calle Mayor, 3", "28013", "Madrid",  
null] · 4 valores · nulos: 1  
compuesto [null, "", "abc"] · 3 valores · nulos:  
1  
compuesto ["dijo \"hola\"", "x"] · 2 valores ·  
nulos: 0
```

13. Profundidad de activación en una base de objetos

Difícil · BD Objeto-Relacionales y Orientadas a Objetos · apuntesdam.com/subject/acceso-datos/topic/bd-objeto-relacionales

Las bases de datos orientadas a objetos (db4o, ObjectDB) guardan grafos de objetos con referencias entre ellos. Al recuperar un objeto no cargan todo el grafo: lo «activan» solo hasta cierta profundidad, y los objetos más lejanos quedan como referencias sin rellenar. Las líneas «objeto ID Clase campo=valor ref=@ID ...» definen el almacén (los valores con @ son referencias a otros objetos, que pueden formar ciclos). Implementa activar para que «activar ID k» muestre el árbol: con profundidad ≥ 1 el objeto se activa y se muestra «Clase ID {campo=valor, ...}» y después cada referencia, en orden, con dos espacios más de sangría y el prefijo «campo → », activada con profundidad $k-1$; con profundidad 0 se muestra «Clase ID (sin activar)»; si el id no existe, «null (ID no existe)». El main cuenta los objetos distintos activados.

Código de partida (java)

```
import java.util.*;

public class Main {
    static class Objeto {
        String id, clase;
        Map<String, String> datos = new LinkedHashMap<>(); // campos simples
        Map<String, String> referencias = new LinkedHashMap<>(); // campo → id del objeto apuntado
    }

    static Map<String, Objeto> almacen = new LinkedHashMap<>();
    static Set<String> activados = new HashSet<>();

    static void activar(String id, int profundidad, String sangria, String prefijo) {
        // TODO: mostrar el objeto y activar sus referencias con profundidad - 1
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            if (p[0].equals("objeto")) { // objeto ID Clase campo=valor ref=@ID ...
                Objeto o = new Objeto();
                o.id = p[1]; o.clase = p[2];
                for (int i = 3; i < p.length; i++) {
                    String[] kv = p[i].split("=");
                    if (kv[1].startsWith("@")) o.referencias.put(kv[0], kv[1].substring(1));
                    else o.datos.put(kv[0], kv[1]);
                }
                almacen.put(o.id, o);
            } else if (p[0].equals("activar")) { // activar ID profundidad
                activados.clear();
                System.out.println("== activar " + p[1] + " con profundidad " + p[2]);
                activar(p[1], Integer.parseInt(p[2]), "", "");
                System.out.println("Objetos leídos del disco: " + activados.size());
            }
        }
    }
}
```

Ejemplo: Profundidad 1

ENTRADA

```
objeto C1 Curso nombre=1DAM tutor=@P1
delegado=@A1
objeto P1 Profesor nombre=Ana departamento=@D1
objeto D1 Departamento nombre=Informática
jefe=@P1
objeto A1 Alumno nombre=Luis grupo=@C1 mentor=@A9
activar C1 1
```

SALIDA ESPERADA

```
== activar C1 con profundidad 1
Curso C1 {nombre=1DAM}
  tutor → Profesor P1 (sin activar)
  delegado → Alumno A1 (sin activar)
Objetos leídos del disco: 1
```

Ejemplo: Profundidad 3 con ciclo

ENTRADA

```
objeto C1 Curso nombre=1DAM tutor=@P1
delegado=@A1
objeto P1 Profesor nombre=Ana departamento=@D1
objeto D1 Departamento nombre=Informática
jefe=@P1
objeto A1 Alumno nombre=Luis grupo=@C1 mentor=@A9
activar C1 3
```

SALIDA ESPERADA

```
== activar C1 con profundidad 3
Curso C1 {nombre=1DAM}
  tutor → Profesor P1 {nombre=Ana}
  departamento → Departamento D1
  {nombre=Informática}
    jefe → Profesor P1 (sin activar)
  delegado → Alumno A1 {nombre=Luis}
    grupo → Curso C1 {nombre=1DAM}
      tutor → Profesor P1 (sin activar)
      delegado → Alumno A1 (sin activar)
    mentor → null (A9 no existe)
Objetos leídos del disco: 4
```

14. Herencia de tablas en PostgreSQL

Muy difícil · BD Objeto-Relacionales y Orientadas a Objetos · apuntesdam.com/subject/acceso-datos/topic/bd-objeto-relacionales

PostgreSQL permite que una tabla herede de otra (CREATE TABLE alumno (curso text) INHERITS (persona)): la hija tiene las columnas del padre más las suyas y, al consultar el padre, también salen las filas de las hijas. Pero las restricciones UNIQUE (y las claves primarias) no se heredan entre tablas: cada tabla solo comprueba sus propias filas. Simúlalo. Órdenes: «tabla t columnas=a,b [hereda=padre] [unico=col]» (muestra «CREATE TABLE t (todas las columnas) [INHERITS (padre)] [UNIQUE (col)]»; las columnas heredadas van primero); «insertar t a=1,b=2» (columnas no indicadas a NULL; si una no existe, «ERROR: la columna x no existe en t»; si viola el único de esa tabla, comparando solo con las filas de esa misma tabla, «ERROR: valor duplicado col=v en t»; si no, «INSERT en t»); «consultar t» y «consultar solo t» (como SELECT * FROM t y SELECT * FROM ONLY t). La consulta muestra «SELECT * FROM [ONLY] t (col | col ...)», cada fila con dos espacios delante y sus valores para las columnas de t separados por « | » (NULL si no tiene), añadiendo « [hija]» a las filas que vienen de otra tabla, y al final « n filas» («1 fila»). Primero van las filas propias y después, sin ONLY, las de cada hija en orden de creación (y las de sus hijas, recursivamente).

Código de partida (java)

```
import java.util.*;

public class Main {
    static class Tabla {
        String nombre, padre, unico;
        List<String> columnas = new ArrayList<>();
        List<Map<String, String>> filas = new ArrayList<>();
        List<String> hijas = new ArrayList<>();
    }

    static Map<String, Tabla> tablas = new LinkedHashMap<>();

    // TODO: static void recoger(Tabla consultada, Tabla t, boolean solo, List<String> salida)

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            String[] p = l.split(" ");
            // TODO: tabla, insertar, consultar y consultar solo
        }
    }
}
```

Ejemplo: Personas, alumnos y profesores

ENTRADA

```
tabla persona columnas=dni,nombre unico=dni
tabla alumno columnas=curso hereda=persona
unico=dni
tabla profesor columnas=departamento
hereda=persona
tabla delegado columnas=desde hereda=alumno
insertar persona dni=111,nombre=Conserje
insertar alumno dni=222,nombre=Ana,curso=1DAM
insertar alumno dni=222,nombre=Otra,curso=2DAM
insertar alumno dni=111,nombre=Luis,curso=1DAM
insertar profesor
dni=333,nombre=Marta,departamento=Informática
insertar delegado
dni=444,nombre=Eva,curso=2DAM,desde=2025
insertar profesor dni=555,nombre=Pepe,curso=1DAM
insertar persona dni=111,nombre=Duplicado
consultar persona
consultar solo persona
consultar alumno
```

SALIDA ESPERADA

```
CREATE TABLE persona (dni, nombre) UNIQUE (dni)
CREATE TABLE alumno (dni, nombre, curso) INHERITS
(persona) UNIQUE (dni)
CREATE TABLE profesor (dni, nombre, departamento)
INHERITS (persona)
CREATE TABLE delegado (dni, nombre, curso, desde)
INHERITS (alumno)
INSERT en persona
INSERT en alumno
ERROR: valor duplicado dni=222 en alumno
INSERT en alumno
INSERT en profesor
INSERT en delegado
ERROR: la columna curso no existe en profesor
ERROR: valor duplicado dni=111 en persona
SELECT * FROM persona (dni | nombre)
  111 | Conserje
  222 | Ana   [alumno]
  111 | Luis  [alumno]
  444 | Eva   [delegado]
  333 | Marta [profesor]
  5 filas
SELECT * FROM ONLY persona (dni | nombre)
  111 | Conserje
  1 fila
SELECT * FROM alumno (dni | nombre | curso)
  222 | Ana | 1DAM
  111 | Luis | 1DAM
  444 | Eva | 2DAM [delegado]
  3 filas
```

Componentes de Acceso a Datos

15. Propiedad ligada (bound property)

Fácil · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Completa el JavaBean Termostato para que su propiedad temperatura sea ligada: cada vez que cambie de valor debe avisar a los oyentes con `firePropertyChange`. El programa lee temperaturas por teclado y los oyentes imprimen cada cambio (si el valor no cambia, no se debe notificar).

Código de partida (java)

```
import java.beans.*;
import java.io.Serializable;
import java.util.Scanner;

public class Main {

    public static class Termostato implements Serializable {
        private final PropertyChangeSupport soporte = new PropertyChangeSupport(this);
        private int temperatura = 20;

        public Termostato() {}

        public int getTemperatura() { return temperatura; }

        public void setTemperatura(int nueva) {
            // TODO: asigna el nuevo valor y avisa a los oyentes del cambio
            temperatura = nueva;
        }

        public void addPropertyChangeListener(PropertyChangeListener l) {
            soporte.addPropertyChangeListener(l); }
        public void removePropertyChangeListener(PropertyChangeListener l) {
            soporte.removePropertyChangeListener(l); }
    }

    public static void main(String[] args) {
        Termostato t = new Termostato();
        t.addPropertyChangeListener(e ->
            System.out.println(e.getPropertyName() + ": " + e.getOldValue() + " -> " + e.getNewValue()));

        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) t.setTemperatura(sc.nextInt());
        System.out.println("Final: " + t.getTemperatura());
    }
}
```

Ejemplo: Tres cambios

ENTRADA

22 25 18

SALIDA ESPERADA

temperatura: 20 -> 22
temperatura: 22 -> 25
temperatura: 25 -> 18
Final: 18

Ejemplo: Valor repetido: no se notifica

ENTRADA

20 21 21

SALIDA ESPERADA

temperatura: 20 -> 21
Final: 21

16. Propiedad restringida (constrained)

Medio · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

El componente CuentaBancaria tiene una propiedad saldo restringida: antes de cambiarla pregunta a los oyentes con `fireVetoableChange`, y cualquiera puede vetar el cambio lanzando `PropertyVetoException`. Completa `setSaldo`. El programa registra un oyente que veta saldos negativos y aplica los movimientos leídos por teclado.

Código de partida (java)

```
import java.beans.*;
import java.util.Scanner;

public class Main {

    public static class CuentaBancaria {
        private final VetoableChangeSupport vetos = new VetoableChangeSupport(this);
        private double saldo = 100;

        public double getSaldo() { return saldo; }

        public void setSaldo(double nuevo) throws PropertyVetoException {
            // TODO: pregunta a los oyentes y, si nadie veta, cambia el saldo
            saldo = nuevo;
        }

        public void addVetoableChangeListener(VetoableChangeListener l) {
            vetos.addVetoableChangeListener(l);
        }
    }

    public static void main(String[] args) {
        CuentaBancaria cuenta = new CuentaBancaria();
        cuenta.addVetoableChangeListener(e -> {
            if ((double) e.getNewValue() < 0) throw new PropertyVetoException("saldo negativo", e);
        });

        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            int movimiento = sc.nextInt();
            try {
                cuenta.setSaldo(cuenta.getSaldo() + movimiento);
                System.out.println("OK " + movimiento + " -> " + cuenta.getSaldo());
            } catch (PropertyVetoException ex) {
                System.out.println("Vetado " + movimiento + ": " + ex.getMessage());
            }
        }
    }
}
```

Ejemplo: Ingreso y retirada válidos

ENTRADA

50 -120

SALIDA ESPERADA

OK 50 -> 150.0
OK -120 -> 30.0

Ejemplo: Retirada que deja saldo negativo

ENTRADA

-150 20

SALIDA ESPERADA

Vetado -150: saldo negativo
OK 20 -> 120.0

17. Persistencia de un JavaBean en XML

Medio · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Los JavaBeans pueden guardarse en XML con XMLEncoder y recuperarse con XMLDecoder, siempre que sigan las convenciones (constructor vacío, getters y setters). Completa guardar() y cargar() para el bean Configuración. El programa lee el tema y el tamaño de letra por teclado, guarda el bean en config.xml, lo vuelve a cargar y muestra sus valores.

Código de partida (java)

```
import java.beans.*;
import java.io.*;
import java.util.Scanner;

public class Main {

    public static class Configuracion {
        private String tema = "claro";
        private int tamanoLetra = 14;

        public Configuracion() {}
        public String getTema() { return tema; }
        public void setTema(String tema) { this.tema = tema; }
        public int getTamanoLetra() { return tamanoLetra; }
        public void setTamanoLetra(int t) { this.tamanoLetra = t; }
    }

    static void guardar(Configuracion c, File fichero) throws IOException {
        // TODO: guarda el bean en XML con XMLEncoder
    }

    static Configuracion cargar(File fichero) throws IOException {
        // TODO: recupera el bean con XMLDecoder
        return new Configuracion();
    }

    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        Configuracion c = new Configuracion();
        c.setTema(sc.next());
        c.setTamanoLetra(sc.nextInt());

        File f = new File("config.xml");
        guardar(c, f);
        Configuracion leida = cargar(f);
        System.out.println(leida.getTema() + " " + leida.getTamanoLetra());
        System.out.println("XML generado: " + (f.length() > 0));
    }
}
```

Ejemplo: Tema oscuro, letra 16

ENTRADA

oscuro 16

SALIDA ESPERADA

oscuro 16
XML generado: true

18. Componente DAO con eventos

Difícil · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Implementa `AlumnoDaoMemoria`, un componente de acceso a datos que cumple la interfaz `AlumnoDao` y además avisa a sus oyentes (`DaoListener`) cada vez que se guarda o se borra un alumno. El programa ejecuta las órdenes leídas por teclado: 'G nombre nota' guarda, 'B nombre' borra y 'L' lista los alumnos ordenados por nombre.

Código de partida (java)

```
import java.util.*;

public class Main {

    public interface DaoListener { void datosCambiados(String accion, String nombre); }

    public interface AlumnoDao {
        void guardar(String nombre, double nota);
        boolean borrar(String nombre);
        Map<String, Double> listar();
        void addDaoListener(DaoListener l);
    }

    public static class AlumnoDaoMemoria implements AlumnoDao {
        private final Map<String, Double> alumnos = new TreeMap<>();
        private final List<DaoListener> oyentes = new ArrayList<>();

        public void guardar(String nombre, double nota) {
            // TODO: guarda y avisa a los oyentes
        }

        public boolean borrar(String nombre) {
            // TODO: borra y avisa a los oyentes si existía
            return false;
        }

        public Map<String, Double> listar() { return Collections.unmodifiableMap(alumnos); }

        public void addDaoListener(DaoListener l) { oyentes.add(l); }
    }

    public static void main(String[] args) {
        AlumnoDao dao = new AlumnoDaoMemoria();
        dao.addDaoListener((accion, nombre) -> System.out.println("[evento] " + accion + " " + nombre));

        Scanner sc = new Scanner(System.in);
        while (sc.hasNext()) {
            String orden = sc.next();
            switch (orden) {
                case "G" -> dao.guardar(sc.next(), Double.parseDouble(sc.next()));
                case "B" -> { String n = sc.next(); if (!dao.borrar(n)) System.out.println("No existe " + n); }
                case "L" -> dao.listar().forEach((n, nota) -> System.out.println(n + " " + nota));
                default -> System.out.println("Orden desconocida");
            }
        }
    }
}
```

Ejemplo: Guardar y listar

ENTRADA

G Luis 6 G Ana 8.5 L

SALIDA ESPERADA

[evento] guardado Luis
[evento] guardado Ana
Ana 8.5
Luis 6.0

Ejemplo: Borrar existente e inexistente

ENTRADA

G Ana 7 B Ana B Pepe L

SALIDA ESPERADA

[evento] guardado Ana
[evento] borrado Ana
No existe Pepe

19. Un contenedor de inyección de dependencias

Muy difícil · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Spring, Jakarta CDI o Micronaut son contenedores de componentes: el programa declara qué componentes hay y de qué dependen, y el contenedor los crea en el orden correcto y los inyecta. Simula uno. «componente N [depende=A,B] [ambito=prototype]» declara un componente (singleton por defecto: una sola instancia; prototype: una nueva cada vez). «obtener N» muestra «obtener N» y antes de crear nada valida el grafo recorriendo las dependencias en orden: si falta un componente, « Error: no existe el componente N» (el pedido) o « Error: falta el componente X (lo necesita Y)»; si hay un ciclo, « Error: dependencia circular A → B → C → A» (desde la primera aparición). Si es válido, crea primero las dependencias (en orden, recursivamente) y después el componente, numerando las instancias globalmente desde 1: « crear X#k» (con « (prototype)» si lo es); los singletons ya creados se reutilizan sin mostrar nada. Termina con «→ N#k» y « (ya existía)» si era un singleton ya creado. «cerrar» destruye los singletons en orden inverso a su creación (« destruir X#k»; los prototype no los gestiona el contenedor), los olvida y muestra «Contenedor cerrado».

Código de partida (java)

```
import java.util.*;

public class Main {
    static Map<String, List<String>> dependencias = new LinkedHashMap<>();
    static Map<String, String> ambitos = new HashMap<>();
    static Map<String, Integer> singletons = new HashMap<>(); // componente → número de instancia
    static List<String> creados = new ArrayList<>(); // singletons en orden de creación (para cerrar)
    static int contador = 0;

    // TODO: static String validar(String nombre, List<String> camino, String quien)
    // static int resolver(String nombre)

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            switch (p[0]) {
                case "componente" -> { // componente N [depende=A,B]
                    [ambito=prototype]
                    List<String> deps = new ArrayList<>();
                    String ambito = "singleton";
                    for (int i = 2; i < p.length; i++) {
                        if (p[i].startsWith("depende="))
                            deps.addAll(List.of(p[i].substring(8).split(",")));
                        if (p[i].startsWith("ambito=")) ambito = p[i].substring(7);
                    }
                    dependencias.put(p[1], deps);
                    ambitos.put(p[1], ambito);
                }
                case "obtener" -> {
                    System.out.println("obtener " + p[1]);
                    // TODO: validar y, si no hay error, resolver
                }
                case "cerrar" -> {
                    // TODO: destruir los singletons en orden inverso al de creación
                    System.out.println("Contenedor cerrado");
                }
            }
        }
    }
}
```

Ejemplo: Tienda

ENTRADA

```
componente DataSource
componente PedidoDao depende=DataSource
componente ClienteDao depende=DataSource
componente Correo ambito=prototype
componente Notificador depende=Correo
componente PedidoService
depende=PedidoDao,ClienteDao,Notificador
componente Informe depende=PedidoDao,Correo
ambito=prototype
obtener PedidoService
obtener PedidoService
obtener Informe
obtener Informe
obtener Factura
cerrar
obtener ClienteDao
```

SALIDA ESPERADA

```
obtener PedidoService
  crear DataSource#1
  crear PedidoDao#2
  crear ClienteDao#3
  crear Correo#4 (prototype)
  crear Notificador#5
  crear PedidoService#6
→ PedidoService#6
obtener PedidoService
→ PedidoService#6 (ya existía)
obtener Informe
  crear Correo#7 (prototype)
  crear Informe#8 (prototype)
→ Informe#8
obtener Informe
  crear Correo#9 (prototype)
  crear Informe#10 (prototype)
→ Informe#10
obtener Factura
  Error: no existe el componente Factura
destruir PedidoService#6
destruir Notificador#5
destruir ClienteDao#3
destruir PedidoDao#2
destruir DataSource#1
Contenedor cerrado
obtener ClienteDao
  crear DataSource#11
  crear ClienteDao#12
→ ClienteDao#12
```

Ejercicios de Acceso a Datos

20. Filtrar un fichero de log

Fácil · Ejercicios de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/ejercicios-acceso-datos

El programa guarda en app.log las líneas recibidas por teclado. Completa filtrar() para que lea app.log y escriba en errores.log solo las líneas que contienen "ERROR", devolviendo cuántas ha copiado. Después se muestra el número y el contenido de errores.log.

Código de partida (java)

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class Main {

    static int filtrar(Path origen, Path destino) throws IOException {
        // TODO: copia a destino las líneas de origen que contienen "ERROR"
        return 0;
    }

    public static void main(String[] args) throws IOException {
        Path log = Path.of("app.log");
        Files.write(log, new String(System.in.readAllBytes()).lines().toList());

        Path errores = Path.of("errores.log");
        int n = filtrar(log, errores);
        System.out.println("Errores: " + n);
        if (Files.exists(errores)) Files.readAllLines(errores).forEach(System.out::println);
    }
}
```

Ejemplo: Dos errores

ENTRADA

```
10:00 INFO arranque
10:01 ERROR conexión perdida
10:02 WARN reintento
10:03 ERROR tiempo agotado
```

SALIDA ESPERADA

```
Errores: 2
10:01 ERROR conexión perdida
10:03 ERROR tiempo agotado
```

Ejemplo: Sin errores

ENTRADA

```
INFO todo bien
INFO fin
```

SALIDA ESPERADA

```
Errores: 0
```

21. Conversor de CSV a JSON

Medio · Ejercicios de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/ejercicios-acceso-datos

Convierte un fichero CSV con cabecera (nombre;ciudad;edad) en un array JSON, sin librerías externas. El programa guarda la entrada en personas.csv; completa convertir() para que devuelva el JSON con el formato exacto del ejemplo: [{"nombre":"Ana","ciudad":"Logroño","edad":20}, ...] (la edad sin comillas, porque es un número).

Código de partida (java)

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class Main {

    static String convertir(Path csv) throws IOException {
        List<String> lineas = Files.readAllLines(csv);
        String[] cabecera = lineas.get(0).split(";");
        List<String> objetos = new ArrayList<>();
        // TODO: recorre las líneas de datos y construye un objeto JSON por línea

        return "[" + String.join(",", objetos) + "]";
    }

    public static void main(String[] args) throws IOException {
        Path csv = Path.of("personas.csv");
        Files.write(csv, new String(System.in.readAllBytes()).lines().toList());
        System.out.println(convertir(csv));
    }
}
```

Ejemplo: Dos personas

ENTRADA

```
nombre;ciudad;edad
Ana;Logroño;20
Luis;Arnedo;22
```

SALIDA ESPERADA

```
[{"nombre":"Ana","ciudad":"Logroño","edad":20},
{"nombre":"Luis","ciudad":"Arnedo","edad":22}]
```

Ejemplo: Solo cabecera

ENTRADA

```
nombre;ciudad;edad
```

SALIDA ESPERADA

```
[]
```

22. Media de precios por categoría (DOM)

Difícil · Ejercicios de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/ejercicios-acceso-datos

El programa guarda el XML recibido en catalogo.xml. Cada <producto> tiene un atributo categoria y un elemento <precio>. Completa medias() para devolver un mapa ordenado categoría → precio medio. Se imprime cada categoría con su media con dos decimales.

Código de partida (java)

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Main {

    static Map<String, Double> medias(File fichero) throws Exception {
        Map<String, Double> resultado = new TreeMap<>();
        // TODO: calcula el precio medio de cada categoría

        return resultado;
    }

    public static void main(String[] args) throws Exception {
        File f = new File("catalogo.xml");
        Files.write(f.toPath(), System.in.readAllBytes());
        medias(f).forEach((cat, media) -> System.out.printf(Locale.US, "%s %.2f%n", cat, media));
    }
}
```

Ejemplo: Dos categorías

ENTRADA

```
<catalogo><producto categoria="audio">
<precio>40</precio></producto><producto
categoria="video"><precio>100</precio></producto>
<producto categoria="audio"><precio>50</precio>
</producto></catalogo>
```

SALIDA ESPERADA

```
audio 45.00
video 100.00
```

Ejercicios largos

23. Conversor de CSV a JSON con comillas y tipos

Difícil · Java · 75 minutos · apuntesdam.com/ejercicios/java/conversor-csv-a-json

Una academia exporta sus alumnos de una hoja de cálculo en CSV y quiere cargarlos en una aplicación web que solo entiende JSON. Parece trivial (separar por comas), pero los datos reales lo complican: hay apellidos como «López, Juan» que llevan una coma dentro y por eso van entre comillas, y textos que contienen las propias comillas, que en CSV se escriben dobles ("").

Además, en JSON los tipos importan: 8.5 es un número, true un booleano, un campo vacío es null y el resto son textos entre comillas. Hay una trampa clásica: un código postal como 08001 parece un número, pero si se convierte se pierde el cero. En JSON un número no puede empezar por cero, así que esos valores se quedan como texto.

Algunas líneas vienen mal (les faltan campos o tienen comillas sin cerrar). No se convierten, pero el programa dice cuáles eran para poder corregirlas.

Requisitos

- La primera línea no vacía es la cabecera, con los nombres de los campos. Cada línea siguiente es un registro. Las líneas en blanco se ignoran (pero cuentan para numerar las líneas).
- Los campos se separan por comas. Un campo puede ir entre comillas dobles, y entonces puede contener comas; dentro de las comillas, "" representa una comilla. A cada campo (y a cada nombre de la cabecera) se le quitan los espacios del principio y del final.
- Un registro con un número de campos distinto del de la cabecera, o con comillas sin cerrar, no se convierte: se anota su número de línea.
- Tipos: un campo vacío es null; un campo que cumple el formato de número JSON (signo menos opcional, 0 o una cifra del 1 al 9 seguida de más cifras, y opcionalmente un punto y decimales) se escribe tal cual; true y false son booleanos; todo lo demás es texto entre comillas, con " escrito \", \ escrito \\ y el tabulador escrito \t.
- Salida: un array JSON con un objeto por registro, en el orden del fichero, con dos espacios de sangría para cada objeto y cuatro para cada propiedad ("campo": valor), comas entre elementos y nada al final de la última línea. Sin registros, [].
- Después del JSON, Registros: N · Líneas ignoradas: M (11, 12...), sin el paréntesis si M es 0. Si la entrada no tiene ni la cabecera, solo El fichero está vacío.

Formato de la entrada

- Un fichero CSV: cabecera y una línea por registro, campos separados por comas y entre comillas cuando hace falta.

Ejemplo: Alumnos de una academia

ENTRADA

```
id,nombre,ciudad,cp,nota,beca
1,Ana Gil,Madrid,28001,8.5,true
2,"López, Juan",Barcelona,08001,6,false
3,"Eva \"la rápida\" Ruiz",Sevilla,41001,,true
4,Luis,Valencia,46001,7.25
5,Marta Díaz,Bilbao,48001,-0.5,false
```

SALIDA ESPERADA

```
[
  {
    "id": 1,
    "nombre": "Ana Gil",
    "ciudad": "Madrid",
    "cp": 28001,
    "nota": 8.5,
    "beca": true
  },
  {
    "id": 2,
    "nombre": "López, Juan",
    "ciudad": "Barcelona",
    "cp": "08001",
    "nota": 6,
    "beca": false
  },
  {
    "id": 3,
    "nombre": "Eva \"la rápida\" Ruiz",
    "ciudad": "Sevilla",
    "cp": 41001,
    "nota": null,
    "beca": true
  },
  {
    "id": 5,
    "nombre": "Marta Díaz",
    "ciudad": "Bilbao",
    "cp": 48001,
    "nota": -0.5,
    "beca": false
  }
]
Registros: 4 · Líneas ignoradas: 1 (5)
```

Ejemplo: Espacios, barras y comillas sin cerrar

ENTRADA

```
codigo , descripcion , precio
A1 , "Cable HDMI 2m" , 7.95
A2,"Ruta C:\temp",0
A3,"Sin cerrar,1
A4,,
```

SALIDA ESPERADA

```
[
  {
    "codigo": "A1",
    "descripcion": "Cable HDMI 2m",
    "precio": 7.95
  },
  {
    "codigo": "A2",
    "descripcion": "Ruta C:\\temp",
    "precio": 0
  },
  {
    "codigo": "A4",
    "descripcion": null,
    "precio": null
  }
]
Registros: 3 · Líneas ignoradas: 1 (4)
```

Código de partida (java)

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {
    /** Campos de una línea CSV separados por comas, con comillas dobles opcionales ("" es una comilla).
     * null si hay comillas sin cerrar. */
    static List<String> campos(String linea) {
        // TODO
        return new ArrayList<>();
    }

    /** Texto JSON entre comillas, con las comillas, barras y tabuladores escapados. */
    static String cadena(String s) {
        // TODO
        return "\"" + s + "\"";
    }

    /** El valor JSON de un campo: null, número, booleano o texto. */
    static String valor(String campo) {
        // TODO
        return cadena(campo);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // TODO: lee la cabecera y cada línea, y escribe el JSON y el resumen
    }
}
```

24. Fichero de productos con registros de tamaño fijo y acceso directo

Muy difícil · Java · 90 minutos · apuntesdam.com/ejercicios/java/fichero-de-registros-con-acceso-directo

Antes de las bases de datos, los programas de gestión guardaban sus datos en ficheros binarios con registros de tamaño fijo. Todavía hoy es la forma de entender cómo trabaja un gestor de bases de datos por dentro: si cada registro ocupa siempre lo mismo, la posición del registro número N se calcula con una multiplicación y se puede ir directamente a él, sin leer los anteriores.

En este fichero cada producto ocupa 77 bytes: el código (`int` , 4 bytes), el nombre (30 caracteres `char` de 2 bytes, 60 en total, rellenando con el carácter `\0`), el precio (`double` , 8), el stock (`int` , 4) y una marca de borrado (`boolean` , 1). El producto con código N vive en el byte $(N - 1) \times 77$: es acceso directo, y por eso puede haber huecos entre productos.

Borrar un registro desplazando todos los siguientes sería lentísimo en un fichero grande, así que las bajas son lógicas: se pone la marca de borrado a `true` y el hueco queda libre para un alta futura con ese código.

Requisitos

- El programa crea un fichero temporal vacío y ejecuta una orden por línea (la orden puede ir en minúsculas; las líneas vacías se ignoran; una orden desconocida escribe `Orden desconocida`).
- `ALTA código;nombre;precio;stock` : el código es un entero de 1 a 9999, el nombre no puede estar vacío, el precio es un número no negativo con hasta dos decimales y el stock un entero no negativo (si no, `Datos no válidos`). Si ya hay un producto activo con ese código, `Ya existe el producto N` . Si no, se escribe el registro en su posición y se escribe `Alta del producto N en el byte B` , añadiendo `(nombre recortado a 30 caracteres)` si el nombre era más largo.
- `PRECIO código precio` cambia solo el precio, sin tocar el resto del registro: `Precio del producto N: antes → después` (o `Datos no válidos` / `No existe el producto N`).
- `BAJA código` pone la marca de borrado: `Baja lógica del producto N` , o `No existe el producto N` si no está activo. `VER código` escribe `N · nombre · precio · S uds` o `No existe el producto N` .

- **LISTAR** escribe **Productos activos: A** y, con dos espacios delante, cada producto activo por orden de código con el formato de **VER**.
- **INFORME** escribe **Fichero: T bytes, H huecos de registro, A activos, L libres o borrados · valor del stock X**, donde H es el tamaño del fichero entre 77 y el valor es la suma de $\text{precio} \times \text{stock}$ de los activos. Los precios se escriben con dos decimales, coma decimal y **€**.
- Un registro está activo si el fichero llega hasta él, el código guardado en él es el suyo (los huecos que se crean al escribir más allá del final están llenos de ceros) y no tiene la marca de borrado.

Formato de la entrada

- Una orden por línea: **ALTA**, **PRECIO**, **BAJA**, **VER**, **LISTAR** o **INFORME**.

Estructura de un registro (77 bytes)

Campo	Tipo	Bytes	Desplazamiento
código	int	4	0
nombre	30 × char	60	4
precio	double	8	64
stock	int	4	72
borrado	boolean	1	76

Ejemplo: Altas, cambios y bajas

ENTRADA

```
ALTA 1;Ratón inalámbrico;9.95;20
ALTA 3;Teclado mecánico;49.90;5
ALTA 1;Repetido;1;1
LISTAR
PRECIO 3 44.50
BAJA 1
VER 1
VER 3
VER 2
INFORME
```

SALIDA ESPERADA

```
Alta del producto 1 en el byte 0
Alta del producto 3 en el byte 154
Ya existe el producto 1
Productos activos: 2
  1 · Ratón inalámbrico · 9,95 € · 20 uds
  3 · Teclado mecánico · 49,90 € · 5 uds
Precio del producto 3: 49,90 € → 44,50 €
Baja lógica del producto 1
No existe el producto 1
3 · Teclado mecánico · 44,50 € · 5 uds
No existe el producto 2
Fichero: 231 bytes, 3 huecos de registro, 1
activos, 2 libres o borrados · valor del stock
222,50 €
```

Ejemplo: Datos que no valen

ENTRADA

```
ALTA 0;Cero;1;1
ALTA 10;;1;1
ALTA 10;Monitor;-5;1
ALTA 10;Monitor;99.999;1
ALTA abc
PRECIO 7 10
BAJA 7
listar
BORRAR 1
```

SALIDA ESPERADA

```
Datos no válidos
Datos no válidos
Datos no válidos
Datos no válidos
Datos no válidos
No existe el producto 7
No existe el producto 7
Productos activos: 0
Orden desconocida
```

Código de partida (java)

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Locale;
import java.util.Scanner;

public class Main {
    static final int NOMBRE = 30; // caracteres del nombre
    static final int TAM = 4 + NOMBRE * 2 + 8 + 4 + 1; // código + nombre + precio + stock + borrado
    = 77 bytes

    static RandomAccessFile f;

    /** Acceso directo: el registro del código N empieza en el byte (N - 1) x 77. */
    static long posicion(int codigo) {
        return (long) (codigo - 1) * TAM;
    }

    static String euros(double x) {
        return String.format(Locale.ROOT, "%.2f", x).replace('.', ',') + " €";
    }

    static void orden(String linea) throws IOException {
        // TODO: ALTA, PRECIO, BAJA, VER, LISTAR e INFORME con seek, read... y write...
    }

    public static void main(String[] args) throws IOException {
        File fichero = File.createTempFile("productos", ".dat");
        fichero.deleteOnExit();
        try (RandomAccessFile raf = new RandomAccessFile(fichero, "rw")) {
            f = raf;
            Scanner sc = new Scanner(System.in);
            while (sc.hasNextLine()) {
                String linea = sc.nextLine().trim();
                if (!linea.isEmpty()) orden(linea);
            }
        }
    }
}
```

Soluciones

1. Media de notas desde un fichero CSV

Fácil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Solución (java)

```
import java.io.*;
import java.util.*;

public class Main {

    static double leerMedia(File fichero) throws IOException {
        double suma = 0;
        int contador = 0;
        try (BufferedReader br = new BufferedReader(new FileReader(fichero))) {
            String linea;
            while ((linea = br.readLine()) != null) {
                String[] partes = linea.split(";");
                if (partes.length < 2) continue; // líneas vacías o mal formadas
                suma += Double.parseDouble(partes[1].trim());
                contador++;
            }
        }
        return contador == 0 ? 0 : suma / contador;
    }

    public static void main(String[] args) throws IOException {
        // Guarda en notas.csv las líneas recibidas por teclado
        File fichero = new File("notas.csv");
        Scanner sc = new Scanner(System.in);
        try (PrintWriter pw = new PrintWriter(new FileWriter(fichero))) {
            while (sc.hasNextLine()) pw.println(sc.nextLine());
        }

        System.out.printf(Locale.US, "Media: %.2f%n", leerMedia(fichero));
    }
}
```

`BufferedReader.readLine()` devuelve una línea cada vez y `null` al llegar al final, por eso el bucle típico es `while ((linea = br.readLine()) != null)`. Cada línea se parte por el `;` y la segunda parte es la nota.

El `try (...)` con recursos cierra el fichero automáticamente al salir, incluso si salta una excepción: es la forma recomendada de abrir cualquier fichero en Java.

2. Serializar una lista de productos

Medio · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Solución (java)

```
import java.io.*;
import java.util.*;

public class Main {

    static class Producto implements Serializable {
        private static final long serialVersionUID = 1L;
        String nombre;
        double precio;
        Producto(String nombre, double precio) { this.nombre = nombre; this.precio = precio; }
    }

    static void guardar(List<Producto> productos, File fichero) throws IOException {
        try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(fichero))) {
            out.writeObject(new ArrayList<>(productos));
        }
    }

    @SuppressWarnings("unchecked")
    static List<Producto> cargar(File fichero) throws IOException, ClassNotFoundException {
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(fichero))) {
            return (List<Producto>) in.readObject();
        }
    }

    public static void main(String[] args) throws Exception {
        List<Producto> productos = new ArrayList<>();
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().split(";");
            if (p.length == 2) productos.add(new Producto(p[0], Double.parseDouble(p[1])));
        }

        File fichero = new File("productos.obj");
        guardar(productos, fichero);
        List<Producto> leidos = cargar(fichero);

        double total = 0;
        for (Producto p : leidos) total += p.precio;
        System.out.printf(Locale.US, "%d productos, total %.2f%n", leidos.size(), total);
    }
}
```

`writeObject` serializa la lista entera de una vez: el `ArrayList` y todos los productos que contiene, que deben implementar `Serializable`. Al leer, `readObject` devuelve un `Object` y hay que convertirlo al tipo que se guardó; de ahí el `@SuppressWarnings("unchecked")`.

Se guarda una copia como `ArrayList` porque la lista recibida podría ser de un tipo no serializable (por ejemplo, la de `List.of`). Los dos flujos se abren con `try` con recursos para que el fichero se cierre siempre.

3. Leer un XML con DOM

Medio · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Solución (java)

```
import java.io.*;
import java.nio.file.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Main {

    static void procesar(File fichero) throws Exception {
        int total = 0, disponibles = 0;
        Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(fichero);
        NodeList libros = doc.getElementsByTagName("libro");
        total = libros.getLength();
        for (int i = 0; i < libros.getLength(); i++) {
            Element libro = (Element) libros.item(i);
            if (libro.getAttribute("disponible").equals("si")) disponibles++;
        }

        System.out.println("Libros: " + total + ", disponibles: " + disponibles);
    }

    public static void main(String[] args) throws Exception {
        File fichero = new File("biblioteca.xml");
        Files.write(fichero.toPath(), System.in.readAllBytes());
        procesar(fichero);
    }
}
```

DOM carga el XML entero en memoria como un árbol. Se obtiene con la cadena `DocumentBuilderFactory` → `DocumentBuilder` → `parse`, y `getElementsByTagName("libro")` devuelve todos los elementos con ese nombre, estén donde estén.

Cada nodo de la lista se convierte a `Element` para poder leer sus atributos con `getAttribute`, que devuelve una cadena vacía si el atributo no existe (nunca `null`).

4. Acceso aleatorio a registros

Difícil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Solución (java)

```
import java.io.*;
import java.util.*;

public class Main {
    static final int TAM = 4 + 20 + 8; // id + nombre(10 chars) + salario

    static void escribir(RandomAccessFile raf, int id, String nombre, double salario) throws IOException
    {
        raf.writeInt(id);
        StringBuilder sb = new StringBuilder(nombre);
        sb.setLength(10); // rellena o recorta a 10 caracteres
        raf.writeChars(sb.toString());
        raf.writeDouble(salario);
    }

    static String leerRegistro(RandomAccessFile raf, int n) throws IOException {
        raf.seek((long) (n - 1) * TAM); // salta directamente al registro n
        int id = raf.readInt();
        StringBuilder nombre = new StringBuilder();
        for (int i = 0; i < 10; i++) nombre.append(raf.readChar());
        double salario = raf.readDouble();
        return id + " " + nombre.toString().trim() + " " + salario;
    }

    public static void main(String[] args) throws IOException {
        try (RandomAccessFile raf = new RandomAccessFile("empleados.dat", "rw")) {
            raf.setLength(0);
            escribir(raf, 1, "Ana", 1850.5);
            escribir(raf, 2, "Luis", 2100);
            escribir(raf, 3, "Marta", 2475.25);
            escribir(raf, 4, "Jorge", 1990);

            int n = new Scanner(System.in).nextInt();
            System.out.println(leerRegistro(raf, n));
        }
    }
}
```

Como todos los registros miden lo mismo (32 bytes), el registro n empieza en el byte $(n - 1) * TAM$. `seek` coloca ahí el puntero del fichero sin leer nada de lo anterior: eso es el acceso aleatorio.

Los campos se leen en el mismo orden y con los mismos tipos con que se escribieron: `readInt`, diez `readChar` y `readDouble`. El nombre se rellenó con caracteres nulos hasta 10, y `trim()` los quita porque son menores que el espacio.

5. Un CSV de verdad: comillas, separadores y saltos de línea

Difícil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Solución (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in).useDelimiter("\\A");
        String texto = sc.hasNext() ? sc.next().replace("\r", "") : "";
        List<List<String>> registros = new ArrayList<>();

        List<String> registro = new ArrayList<>();
        StringBuilder campo = new StringBuilder();
        boolean enComillas = false, alInicio = true, entrecomillado = false;
        for (int i = 0; i < texto.length(); i++) {
            char c = texto.charAt(i);
            if (enComillas) {
                if (c != '"') campo.append(c);
                else if (i + 1 < texto.length() && texto.charAt(i + 1) == '"') { campo.append('"'); i++; }
                // " es una comilla
                else enComillas = false;
                // " cierra
            } else if (c == '"' && alInicio) {
                enComillas = true;
                entrecomillado = true;
                alInicio = false;
            } else if (c == ';' || c == '\n') {
                registro.add(campo.toString());
                campo.setLength(0);
                alInicio = true;
                if (c == '\n') {
                    boolean vacio = registro.size() == 1 && registro.get(0).isEmpty() && !entrecomillado;
                    if (!vacio) registros.add(registro); // las líneas vacías se ignoran
                    registro = new ArrayList<>();
                }
                entrecomillado = false;
            } else {
                campo.append(c);
                alInicio = false;
            }
        }
        if (enComillas) {
            System.out.println("Error: comilla sin cerrar en el registro " + registros.size());
            return;
        }
        if (!alInicio || !registro.isEmpty()) { // último registro sin salto de línea
            registro.add(campo.toString());
            registros.add(registro);
        }
        if (registros.isEmpty()) {
            System.out.println("Registros válidos: 0 de 0 · Campos: 0");
            return;
        }

        List<String> cabecera = registros.get(0);
        int validos = 0;
        for (int r = 1; r < registros.size(); r++) {
            List<String> reg = registros.get(r);
            if (reg.size() != cabecera.size()) {
                System.out.println("Error en el registro " + r + ": tiene " + reg.size() + " campos y se esperaban " + cabecera.size());
                continue;
            }
            validos++;
            System.out.println("Registro " + r + ":");
            for (int k = 0; k < reg.size(); k++) {
                String v = reg.get(k);
                System.out.println("  " + cabecera.get(k) + " = " + (v.isEmpty() ? "(vacío)" : v.replace("\n", "\\n")));
            }
        }
    }
}
```

```

    }
}
System.out.println("Registros válidos: " + validos + " de " + (registros.size() - 1) + " .
    Campos: " + cabecera.size());
}
}

```

Un CSV real no se puede partir con `split`, porque el separador y el salto de línea pueden estar dentro de un campo. La solución es una pequeña máquina de estados que lee carácter a carácter y recuerda si está dentro de comillas: dentro, todo se copia salvo la comilla (que cierra) o la comilla doble `""` (que es una comilla); fuera, `;` cierra el campo y el salto de línea cierra el registro.

La variable `alInicio` distingue la comilla que abre un campo (la primera) de una comilla en mitad de un texto, que es un carácter normal. `entrecomillado` sirve para no confundir una línea vacía con un registro que tiene un único campo `""`.

Si al terminar el texto seguimos dentro de comillas, el fichero está mal formado; si no, el último registro puede no tener salto de línea final y hay que añadirlo a mano.

6. Leer un formato binario propio con versión y CRC

Muy difícil · Manejo de Ficheros · apuntesdam.com/subject/acceso-datos/topic/gestion-ficheros

Solución (java)

```
import java.nio.*;
import java.nio.charset.StandardCharsets;
import java.util.*;
import java.util.zip.CRC32;

public class Main {
    static byte[] desdeHex(String h) {
        byte[] b = new byte[h.length() / 2];
        for (int i = 0; i < b.length; i++) b[i] = (byte) Integer.parseInt(h.substring(2 * i, 2 * i + 2),
            16);
        return b;
    }

    public static void main(String[] args) {
        byte[] datos = desdeHex(new Scanner(System.in).nextLine().trim());
        ByteBuffer buf = ByteBuffer.wrap(datos); // big-endian, como DataInputStream

        if (datos.length < 4 || !new String(datos, 0, 4, StandardCharsets.US_ASCII).equals("ALUM")) {
            System.out.println("Formato desconocido");
            return;
        }
        buf.position(4);
        List<String> lineas = new ArrayList<>();
        int version, n;
        double suma = 0;
        try {
            version = buf.getShort();
            if (version != 1 && version != 2) {
                System.out.println("Versión " + version + " no soportada");
                return;
            }
            n = buf.getInt();
            for (int i = 0; i < n; i++) {
                int id = buf.getInt();
                byte[] nombre = new byte[buf.getShort() & 0xFFFF];
                buf.get(nombre);
                String curso = version == 2 ? " · curso " + buf.get() : "";
                float nota = buf.getFloat();
                suma += nota;
                lineas.add(String.format(Locale.US, "%d · %s%s · %.2f", id, new String(nombre,
                    StandardCharsets.UTF_8), curso, nota));
            }
            int finDatos = buf.position();
            long esperado = buf.getInt() & 0xFFFFFFFFL; // el int del fichero, sin signo
            CRC32 crc = new CRC32();
            crc.update(datos, 0, finDatos);
            if (crc.getValue() != esperado) {
                System.out.printf("CRC incorrecto: esperado %08X, calculado %08X%n", esperado,
                    crc.getValue());
                return;
            }
        } catch (BufferUnderflowException e) { // se intentó leer más allá del final
            System.out.println("Fichero truncado");
            return;
        }
        lineas.forEach(System.out::println);
        System.out.printf(Locale.US, "Registros: %d · media %.2f · versión %d · CRC OK%n", n, n == 0 ? 0
            : suma / n, version);
    }
}
```

ByteBuffer lee tipos binarios en big-endian, igual que **DataInputStream**: **getShort** lee 2 bytes, **getInt** 4 y **getFloat**

4. El nombre tiene longitud variable, así que primero se lee su longitud y luego exactamente esos bytes, que se decodifican como UTF-8 (por eso «İñaki» ocupa más bytes que letras).

Para detectar un fichero truncado no hace falta comprobar el tamaño antes de cada lectura: si se intenta leer más allá del final, `ByteBuffer` lanza `BufferUnderflowException`, y basta con capturarla una vez alrededor de toda la lectura.

El CRC se calcula sobre los bytes desde el principio hasta donde empieza el CRC guardado. Como el `int` de Java tiene signo y el CRC32 no, se convierte con `& 0xFFFFFFFFL` antes de comparar.

7. Sentencias preparadas frente a concatenación

Medio · Conectores JDBC · apuntesdam.com/subject/acceso-datos/topic/jdbc

Solución (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String plantilla = sc.nextLine();
        List<String[]> params = new ArrayList<>();           // {tipo, valor}
        while (sc.hasNextLine()) {
            String l = sc.nextLine();
            if (l.isBlank()) continue;
            int dos = l.indexOf(':');
            params.add(new String[] { l.substring(0, dos), l.substring(dos + 1) });
        }

        // Posiciones de los ? que no están dentro de un literal '...'
        List<Integer> marcadores = new ArrayList<>();
        boolean enLiteral = false;
        for (int i = 0; i < plantilla.length(); i++) {
            char c = plantilla.charAt(i);
            if (c == '\\' ) enLiteral = !enLiteral;
            else if (c == '?' && !enLiteral) marcadores.add(i);
        }
        System.out.println("Parámetros: " + marcadores.size());
        if (marcadores.size() != params.size()) {
            System.out.println("Error: la sentencia tiene " + marcadores.size() + " parámetros y se han
                dado " + params.size());
            return;
        }
        for (int k = 0; k < params.size(); k++) {
            String tipo = params.get(k)[0], valor = params.get(k)[1];
            if (tipo.equals("i") && !valor.matches("-?\\d+")) {
                System.out.println("Error: el parámetro " + (k + 1) + " no es un entero: " + valor);
                return;
            }
            if (tipo.equals("d") && !valor.matches("-?\\d+(\\.\\d+)?")) {
                System.out.println("Error: el parámetro " + (k + 1) + " no es un decimal: " + valor);
                return;
            }
        }

        StringBuilder concatenada = new StringBuilder(), preparada = new StringBuilder();
        boolean inyeccion = false;
        int desde = 0;
        for (int k = 0; k < marcadores.size(); k++) {
            String trozo = plantilla.substring(desde, marcadores.get(k));
            String tipo = params.get(k)[0], valor = params.get(k)[1];
            String literal = switch (tipo) {
                case "b" -> valor.equalsIgnoreCase("true") ? "TRUE" : "FALSE";
                case "f" -> "DATE '" + valor + "'";
                case "n" -> "NULL";
                case "s" -> null;
                default -> valor; // enteros y decimales, tal cual
            };
            if (tipo.equals("s")) {
                if (valor.contains("'")) inyeccion = true;
                concatenada.append(trozo).append('\\').append(valor).append('\\');
                preparada.append(trozo).append('\\').append(valor.replace("'", "''")).append('\\');
            } else {
                concatenada.append(trozo).append(literal);
                preparada.append(trozo).append(literal);
            }
            desde = marcadores.get(k) + 1;
        }
        concatenada.append(plantilla.substring(desde));
        preparada.append(plantilla.substring(desde));
        System.out.println("Concatenando: " + concatenada);
    }
}
```

```

        System.out.println("Preparada: " + preparada);
        System.out.println(inyeccion
            ? "¡Inyección! Con la concatenación el texto del usuario cambia la estructura de la consulta"
            : "Las dos versiones son equivalentes con estos datos");
    }
}

```

Para localizar los marcadores se recorre la plantilla recordando si se está dentro de un literal entre comillas simples: cada `'` cambia el estado, y solo cuentan los `?` de fuera. Así el `?` de `'¿Qué?'` no se toma por un parámetro.

Las dos versiones se construyen a la vez, copiando los trozos de plantilla entre marcadores y poniendo en su lugar cada valor. La única diferencia está en los textos: al concatenar se pegan tal cual, y con la sentencia preparada la comilla del dato se duplica, de modo que nunca puede cerrar el literal.

Con `' OR '1'='1` se ve el problema: al concatenar, la comilla del usuario cierra el literal y el resto se convierte en una condición que siempre es cierta. Por eso los datos del usuario se pasan siempre con `PreparedStatement` y `setString`.

8. Un pool de conexiones por dentro

Difícil · Conectores JDBC · apuntesdam.com/subject/acceso-datos/topic/jdbc

Solución (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] conf = sc.nextLine().trim().split(" ");    // «max M espera W»
        int max = Integer.parseInt(conf[1]), espera = Integer.parseInt(conf[3]);
        List<String[]> eventos = new ArrayList<>();          // {tiempo, accion, cliente}
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (!l.isEmpty()) eventos.add(l.split(" "));
        }

        Deque<Integer> libres = new ArrayDeque<>();          // la primera es la que lleva más tiempo libre
        Map<String, Integer> enUso = new HashMap<>();        // cliente → conexión
        Deque<Object[]> cola = new ArrayDeque<>();          // {cliente, instante en que empezó a esperar}
        int creadas = 0, reutilizadas = 0, esperas = 0, timeouts = 0, servidas = 0;
        long esperaTotal = 0;

        for (int e = 0; e <= eventos.size(); e++) {
            long t = e < eventos.size() ? Long.parseLong(eventos.get(e)[0]) : Long.MAX_VALUE;
            // Antes de cada evento (y al final), se van quienes han superado la espera
            while (!cola.isEmpty() && (long) cola.peekFirst()[1] + espera < t) {
                Object[] w = cola.pollFirst();
                System.out.println(((long) w[1] + espera) + ": " + w[0] + " agota la espera (timeout)");
                timeouts++;
            }
            if (e == eventos.size()) break;
            String accion = eventos.get(e)[1], cliente = eventos.get(e)[2];

            if (accion.equals("pedir")) {
                if (!libres.isEmpty()) {
                    int k = libres.pollFirst();
                    enUso.put(cliente, k);
                    reutilizadas++;
                    System.out.println(t + ": " + cliente + " recibe la conexión #" + k + " (reutilizada)");
                } else if (creadas < max) {
                    int k = ++creadas;
                    enUso.put(cliente, k);
                    System.out.println(t + ": " + cliente + " recibe la conexión #" + k + " (nueva)");
                } else {
                    cola.addLast(new Object[] { cliente, t });
                    esperas++;
                    System.out.println(t + ": " + cliente + " espera (pool lleno)");
                }
            } else {
                Integer k = enUso.remove(cliente);
                if (k == null) {
                    System.out.println(t + ": Error: " + cliente + " no tiene conexión");
                } else if (!cola.isEmpty()) {
                    Object[] w = cola.pollFirst();
                    long esperado = t - (long) w[1];
                    enUso.put((String) w[0], k);
                    servidas++;
                    esperaTotal += esperado;
                    System.out.println(t + ": " + cliente + " devuelve #" + k + " → pasa a " + w[0] + " (esperó " + esperado + " ms)");
                } else {
                    libres.addLast(k);
                    System.out.println(t + ": " + cliente + " devuelve #" + k + " (queda libre)");
                }
            }
        }
    }
}
```

```

        System.out.println("Conexiones creadas: " + creadas + " · Reutilizadas: " + reutilizadas + " ·
        Esperas: " + esperas
            + " · Timeouts: " + timeouts + " · Espera media: " + (servidas == 0 ? 0 : esperaTotal /
            servidas) + " ms");
        System.out.println("Al final: " + creadas + " abiertas, " + libres.size() + " libres, " +
            enUso.size() + " en uso");
    }
}

```

El pool se modela con tres estructuras: una cola de conexiones libres (`ArrayDeque`: se devuelven por el final y se entregan por el principio, así sale la que lleva más tiempo libre), un mapa de quién tiene cada conexión y una cola FIFO de clientes esperando con el instante en que empezaron.

Los tiempos límite no son eventos de la entrada, así que se comprueban antes de procesar cada evento: mientras el primero de la cola lleve esperando más de W , se va. El bucle da una vuelta extra al final con un instante «infinito» para agotar los que queden, sin repetir el código.

Al devolver una conexión, si alguien espera se le pasa directamente, sin que llegue a quedar libre: así funciona un pool real, y por eso no cuenta como reutilizada.

9. Transacciones con puntos de guardado y bloqueos

Muy difícil · Conectores JDBC · apuntesdam.com/subject/acceso-datos/topic/jdbc

Solución (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, Integer> confirmado = new LinkedHashMap<>(); // lo que ve cualquiera
        for (String par : sc.nextLine().trim().split(" ")) {
            if (!par.contains("=")) continue; // «cuentas A1=100 A2=50 ...»
            String[] kv = par.split("=");
            confirmado.put(kv[0], Integer.parseInt(kv[1]));
        }
        // Por sesión: cambios pendientes (null si no hay transacción) y puntos de guardado
        Map<String, Map<String, Integer>> pendientes = new HashMap<>();
        Map<String, LinkedHashMap<String, Map<String, Integer>>> puntos = new HashMap<>();

        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            if (p.length < 2) continue;
            String s = p[0], otra = s.equals("A") ? "B" : "A";
            Map<String, Integer> pend = pendientes.get(s);
            String orden = p[1].equals("ROLLBACK") && p.length > 3 ? "ROLLBACK TO" : p[1];
            switch (orden) {
                case "BEGIN" -> {
                    if (pend != null) System.out.println(s + ": ya hay una transacción abierta");
                    else {
                        pendientes.put(s, new LinkedHashMap<>());
                        puntos.put(s, new LinkedHashMap<>());
                        System.out.println(s + ": BEGIN");
                    }
                }
                case "SELECT" -> {
                    String c = p[2];
                    int v = pend != null && pend.containsKey(c) ? pend.get(c) : confirmado.get(c);
                    System.out.println(s + " ve " + c + " = " + v);
                }
                case "UPDATE" -> {
                    String c = p[2];
                    Map<String, Integer> deLaOtra = pendientes.get(otra);
                    if (deLaOtra != null && deLaOtra.containsKey(c)) {
                        System.out.println(s + ": UPDATE " + c + " bloqueado por " + otra + " (se cancela la sentencia)");
                        break;
                    }
                    int actual = pend != null && pend.containsKey(c) ? pend.get(c) : confirmado.get(c);
                    int nuevo = actual + Integer.parseInt(p[3]);
                    if (nuevo < 0) System.out.println(s + ": UPDATE " + c + " rechazado: el saldo quedaría en " + nuevo);
                    else if (pend == null) { // autocommit
                        confirmado.put(c, nuevo);
                        System.out.println(s + ": " + c + " = " + nuevo + " (confirmado)");
                    } else {
                        pend.put(c, nuevo);
                        System.out.println(s + ": " + c + " = " + nuevo);
                    }
                }
                case "SAVEPOINT" -> {
                    if (pend == null) { System.out.println(s + ": no hay transacción"); break; }
                    puntos.get(s).remove(p[2]);
                    puntos.get(s).put(p[2], new LinkedHashMap<>(pend)); // foto de los cambios pendientes
                    System.out.println(s + ": SAVEPOINT " + p[2]);
                }
                case "ROLLBACK TO" -> {
                    if (pend == null) { System.out.println(s + ": no hay transacción"); break; }
                    LinkedHashMap<String, Map<String, Integer>> pts = puntos.get(s);
                    String nombre = p[3];

```

```

        if (!pts.containsKey(nombre)) {
            System.out.println(s + ": no existe el punto de guardado " + nombre);
            break;
        }
        pendientes.put(s, new LinkedHashMap<>(pts.get(nombre)));
        // se borran los puntos posteriores
        boolean despues = false;
        for (Iterator<String> it = pts.keySet().iterator(); it.hasNext(); ) {
            String k = it.next();
            if (despues) it.remove();
            if (k.equals(nombre)) despues = true;
        }
        System.out.println(s + ": vuelta a " + nombre);
    }
    case "COMMIT" -> {
        if (pend == null) { System.out.println(s + ": no hay transacción"); break; }
        confirmado.putAll(pend);
        System.out.println(s + ": COMMIT (" + pend.size() + " cambios)");
        pendientes.put(s, null);
    }
    case "ROLLBACK" -> {
        if (pend == null) { System.out.println(s + ": no hay transacción"); break; }
        System.out.println(s + ": ROLLBACK (" + pend.size() + " cambios descartados)");
        pendientes.put(s, null);
    }
    default -> { }
}
}

for (String s : List.of("A", "B")) {
    if (pendientes.get(s) != null) System.out.println(s + " tenía una transacción abierta: se deshace al cerrar la conexión");
}
StringBuilder fin = new StringBuilder("Confirmado:");
confirmado.forEach((c, v) -> fin.append(' ').append(c).append('=').append(v));
System.out.println(fin);
}
}

```

La idea clave del aislamiento READ COMMITTED es que cada sesión trabaja sobre su propio mapa de cambios pendientes y nadie más los ve hasta el COMMIT. Por eso leer es «el valor pendiente propio si existe y, si no, el confirmado», y confirmar es copiar los pendientes sobre el mapa común con `putAll`.

Un punto de guardado es una foto (una copia) de los cambios pendientes en ese momento. Volver a él es restaurar la copia y borrar los puntos creados después; usar un `LinkedHashMap` mantiene su orden de creación, y se borran con un `Iterator` para poder eliminar mientras se recorre.

El bloqueo se simula comprobando si la otra sesión tiene esa cuenta entre sus pendientes. Sin transacción (`pend == null`), cada UPDATE se confirma en el acto: es el modo autocommit de JDBC.

10. El problema N+1 y las estrategias de carga

Medio · Mapeo Objeto-Relacional (Hibernate/JPA) · apuntesdam.com/subject/acceso-datos/topic/hibernate

Solución (java)

```
import java.util.*;

public class Main {
    record Autor(int id, String nombre) { }
    record Libro(int id, int autorId, String titulo) { }

    static int consultas = 0;

    static void sql(String s) {
        System.out.println("SQL: " + s);
        consultas++;
    }

    static void mostrar(Autor a, List<Libro> libros) {
        List<String> titulos = new ArrayList<>();
        for (Libro l : libros) if (l.autorId() == a.id()) titulos.add(l.titulo());
        int n = titulos.size();
        System.out.println(a.nombre() + ": " + n + (n == 1 ? " libro" : " libros") + (n == 0 ? "" : " ("
            + String.join(", ", titulos) + ")"));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<Autor> autores = new ArrayList<>();
        List<Libro> libros = new ArrayList<>();
        String estrategia = "LAZY";
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.startsWith("autor ")) {
                String[] p = l.split(" ", 3);
                autores.add(new Autor(Integer.parseInt(p[1]), p[2]));
            } else if (l.startsWith("libro ")) {
                String[] p = l.split(" ", 4);
                libros.add(new Libro(Integer.parseInt(p[1]), Integer.parseInt(p[2]), p[3]));
            } else if (l.startsWith("estrategia ")) {
                estrategia = l.substring(11).trim();
            }
        }
        autores.sort(Comparator.comparingInt(Autor::id));
        libros.sort(Comparator.comparingInt(Libro::id));

        if (estrategia.equals("JOIN FETCH")) {
            sql("select * from autor a left join libro l on l.autor_id = a.id order by a.id");
            for (Autor a : autores) mostrar(a, libros);
        } else {
            sql("select * from autor order by id");
            Set<Integer> cargados = new HashSet<>(); // autores con sus libros ya cargados
            if (estrategia.equals("EAGER")) {
                for (Autor a : autores) {
                    sql("select * from libro where autor_id = " + a.id());
                    cargados.add(a.id());
                }
            }
            for (int i = 0; i < autores.size(); i++) {
                Autor a = autores.get(i);
                if (!cargados.contains(a.id())) { // a.getLibros() en una colección sin cargar
                    if (estrategia.equals("SUBSELECT")) {
                        sql("select * from libro where autor_id in (select id from autor)");
                        for (Autor x : autores) cargados.add(x.id());
                    } else if (estrategia.startsWith("BATCH")) {
                        int k = Integer.parseInt(estrategia.substring(6).trim());
                        List<String> ids = new ArrayList<>();
                        for (int j = i; j < autores.size() && ids.size() < k; j++) {
                            if (cargados.add(autores.get(j).id()))
                                ids.add(String.valueOf(autores.get(j).id()));
                        }
                    }
                }
            }
        }
    }
}
```

```

        sql("select * from libro where autor_id in (" + String.join(", ", ids) + ")");
    } else {
        sql("select * from libro where autor_id = " + a.id());
        cargados.add(a.id());
    }
}
mostrar(a, libros);
}
}
System.out.println("Consultas: " + consultas + " (estrategia " + estrategia + ")");
}
}

```

Todas las estrategias comparten la misma estructura: se recorren los autores y, antes de usar los libros de uno, se comprueba si su colección ya está cargada (un `Set` de ids). Lo único que cambia es qué consulta se lanza cuando no lo está y cuántos autores deja cargados: uno (LAZY), un lote de k (BATCH) o todos (SUBSELECT).

EAGER carga todo justo después de la primera consulta, pero con una consulta por autor: sigue siendo N+1, solo que antes. JOIN FETCH es la única que lo trae todo en una sola consulta.

El método `sql` imprime y cuenta a la vez, así el total de consultas no puede desajustarse. `cargados.add(id)` devuelve `false` si el autor ya estaba, lo que permite construir el lote saltando los ya cargados en la misma línea.

11. Los estados de una entidad JPA

Difícil · Mapeo Objeto-Relacional (Hibernate/JPA) · apuntesdam.com/subject/acceso-datos/topic/hibernate

Solución (java)

```
import java.util.*;

public class Main {
    static class Entidad {
        Long id;
        String tabla;
        Map<String, String> campos = new LinkedHashMap<>();
        String estado = "NUEVA"; // NUEVA, GESTIONADA, SEPARADA, ELIMINADA
        Map<String, String> instantanea; // copia al pasar a gestionada o tras el último flush
    }

    static Map<String, String> leerCampos(String texto) {
        Map<String, String> m = new LinkedHashMap<>();
        for (String par : texto.split(",")) {
            int i = par.indexOf('=');
            m.put(par.substring(0, i), par.substring(i + 1));
        }
        return m;
    }

    static Entidad gestionada(List<Entidad> contexto, String tabla, long id) {
        for (Entidad e : contexto) if (e.tabla.equals(tabla) && e.id == id &&
            e.estado.equals("GESTIONADA")) return e;
        return null;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, Entidad> variables = new HashMap<>();
        Map<Long, Map<String, String>> baseDatos = new TreeMap<>();
        List<Entidad> contexto = new ArrayList<>(); // contexto de persistencia (en orden de entrada)
        long[] siguienteId = { 1 };
        while (sc.hasNextLine()) {
            String linea = sc.nextLine().trim();
            if (linea.isEmpty()) continue;
            String[] p = linea.split(" ", 3);
            String v = p.length > 1 ? p[1] : "";
            Entidad e = variables.get(v);
            switch (p[0]) {
                case "new" -> {
                    String[] t = p[2].split(" ", 2);
                    Entidad n = new Entidad();
                    n.tabla = t[0].toLowerCase();
                    n.campos = leerCampos(t[1]);
                    variables.put(v, n);
                    System.out.println(v + ": NUEVA (transient)");
                }
                case "persist" -> {
                    switch (e.estado) {
                        case "GESTIONADA" -> System.out.println(v + " ya está gestionada: no se hace nada");
                        case "SEPARADA" -> System.out.println("PersistentObjectException: " + v + " está separada (usa merge)");
                        default -> {
                            e.id = siguienteId[0]++;
                            List<String> valores = new ArrayList<>();
                            for (String x : e.campos.values()) valores.add("'" + x + "'");
                            System.out.println("SQL: insert into " + e.tabla + " (" + String.join(", ",
                                e.campos.keySet() + ") values ("
                                    + String.join(", ", valores) + ") → id " + e.id);
                            baseDatos.put(e.id, new LinkedHashMap<>(e.campos));
                            e.estado = "GESTIONADA";
                            e.instantanea = new LinkedHashMap<>(e.campos);
                            if (!contexto.contains(e)) contexto.add(e);
                        }
                    }
                }
            }
        }
    }
}
```

```

    }
    case "set" -> {
        e.campos.putAll(LeerCampos(p[2]));
        if (!e.estado.equals("GESTIONADA")) {
            System.out.println(v + " (" + e.estado.toLowerCase() + "): el cambio solo está en memoria");
        }
    }
}
case "find" -> {
    String[] t = p[2].split(" ");
    String tabla = t[0].toLowerCase();
    long id = Long.parseLong(t[1]);
    Entidad g = gestionada(contexto, tabla, id);
    if (g != null) {
        // caché de primer nivel
        String otra = new TreeMap<>(variables).entrySet().stream()
            .filter(x -> x.getValue() == g && !x.getKey().equals(v))
            .map(Map.Entry::getKey).findFirst().orElse(v);
        variables.put(v, g);
        System.out.println(v + " es la misma instancia que " + otra + " (caché de primer nivel, sin SQL)");
        break;
    }
    System.out.println("SQL: select * from " + tabla + " where id = " + id);
    Map<String, String> fila = baseDatos.get(id);
    if (fila == null) {
        variables.remove(v);
        System.out.println(v + " = null");
        break;
    }
    Entidad n = new Entidad();
    n.id = id;
    n.tabla = tabla;
    n.campos = new LinkedHashMap<>(fila);
    n.estado = "GESTIONADA";
    n.instantanea = new LinkedHashMap<>(fila);
    contexto.add(n);
    variables.put(v, n);
    System.out.println(v + ": GESTIONADA " + n.campos);
}
case "detach" -> {
    contexto.remove(e);
    e.estado = "SEPARADA";
    System.out.println(v + ": SEPARADA");
}
}
case "merge" -> {
    String w = p[2].trim();
    Entidad origen = variables.get(w);
    Entidad g = gestionada(contexto, origen.tabla, origen.id);
    if (g == null) {
        // se carga la fila en el contexto
        System.out.println("SQL: select * from " + origen.tabla + " where id = " + origen.id);
        g = new Entidad();
        g.id = origen.id;
        g.tabla = origen.tabla;
        g.campos = new LinkedHashMap<>(baseDatos.get(origen.id));
        g.estado = "GESTIONADA";
        g.instantanea = new LinkedHashMap<>(g.campos);
        contexto.add(g);
    }
    g.campos.putAll(origen.campos);
    // se copia el estado; origen no cambia
    variables.put(v, g);
    System.out.println(v + ": copia gestionada de " + w + " (" + w + " sigue separada)");
}
case "remove" -> {
    if (!e.estado.equals("GESTIONADA")) {
        System.out.println("IllegalArgumentException: " + v + " no está gestionada");
    } else {
        e.estado = "ELIMINADA";
        System.out.println(v + ": ELIMINADA (se borrará en el flush)");
    }
}
}
case "flush", "commit" -> {

```

```

        int n = 0;
        for (Entidad x : new ArrayList<>(contexto)) {
            if (x.estado.equals("ELIMINADA")) {
                System.out.println("SQL: delete from " + x.tabla + " where id = " + x.id);
                baseDatos.remove(x.id);
                contexto.remove(x);
                n++;
            } else {
                List<String> cambios = new ArrayList<>();
                x.campos.forEach((k, val) -> {
                    if (!val.equals(x.instantanea.get(k))) cambios.add(k + " = '" + val +
                        "'");
                });
                if (cambios.isEmpty()) continue; // dirty checking: sin cambios, sin SQL
                System.out.println("SQL: update " + x.tabla + " set " + String.join(", ",
                    cambios) + " where id = " + x.id);
                baseDatos.put(x.id, new LinkedHashMap<>(x.campos));
                x.instantanea = new LinkedHashMap<>(x.campos);
                n++;
            }
        }
        String sentencias = n + (n == 1 ? " sentencia" : " sentencias");
        System.out.println(p[0].equals("flush") ? "flush: " + sentencias : "COMMIT (" +
            sentencias + " en el flush)");
    }
    case "close" -> {
        for (Entidad x : contexto) x.estado = "SEPARADA";
        System.out.println("Contexto cerrado: " + contexto.size() + " entidades pasan a
            SEPARADA");
        contexto.clear();
    }
    case "estado" -> {
        if (e == null) System.out.println(v + " = null");
        else System.out.println(v + ": " + e.estado + (e.id != null ? " id=" + e.id : "") + "
            " + e.campos);
    }
    default -> { }
}
}
}
}
}

```

El contexto de persistencia es una lista de las entidades gestionadas, y cada una guarda una instantánea de sus campos tal como están en la base de datos. En el flush se compara cada entidad con su instantánea (dirty checking) y solo se lanza un UPDATE con los campos que han cambiado; si nada cambió, no hay SQL. Por eso cambiar una nota a 8 y otra vez a 6 no genera ninguna sentencia.

find primero busca en el contexto: si la fila ya está gestionada devuelve la misma instancia (la caché de primer nivel), y solo si no está lanza el SELECT. Para elegir el nombre de la otra variable se recorren las variables ordenadas con un **TreeMap** y se comparan por identidad (**==**), porque lo que importa es si es el mismo objeto.

merge nunca convierte el objeto separado en gestionado: busca (o carga) la instancia gestionada con ese id y le copia los campos. Por eso, después, **remove** del objeto separado falla y el de la copia funciona.

12. Arrays y tipos compuestos de PostgreSQL

Medio · BD Objeto-Relacionales y Orientadas a Objetos · apuntesdam.com/subject/acceso-datos/topic/bd-objeto-

```

import java.util.*;

public class Main {
    static String json(List<String> valores) {
        List<String> partes = new ArrayList<>();
        for (String v : valores)
            partes.add(v == null ? "null" : "\"" + v.replace("\\", "\\\\").replace("\"", "\\\"") + "\"");
        return "[" + String.join(", ", partes) + "]";
    }

    static List<String> leerArray(String texto) {
        List<String> res = new ArrayList<>();
        String s = texto.substring(1, texto.length() - 1); // sin las llaves
        if (s.isBlank()) return res;
        int i = 0;
        while (true) {
            while (i < s.length() && s.charAt(i) == ' ') i++;
            if (i < s.length() && s.charAt(i) == '"') { // elemento entre comillas
                StringBuilder sb = new StringBuilder();
                i++;
                while (i < s.length() && s.charAt(i) != '"') {
                    if (s.charAt(i) == '\\') i++; // \ escapa el siguiente carácter
                    sb.append(s.charAt(i++));
                }
                res.add(sb.toString());
                while (i < s.length() && s.charAt(i) != ',') i++;
            } else { // sin comillas
                int fin = s.indexOf(',', i);
                if (fin < 0) fin = s.length();
                String v = s.substring(i, fin).trim();
                res.add(v.equalsIgnoreCase("NULL") ? null : v);
                i = fin;
            }
            if (i >= s.length()) break;
            i++; // la coma
        }
        return res;
    }

    static List<String> leerCompuesto(String texto) {
        List<String> res = new ArrayList<>();
        String s = texto.substring(1, texto.length() - 1); // sin los paréntesis
        int i = 0;
        while (true) {
            if (i < s.length() && s.charAt(i) == '"') {
                StringBuilder sb = new StringBuilder();
                i++;
                while (i < s.length()) {
                    char c = s.charAt(i);
                    if (c == '"' && i + 1 < s.length() && s.charAt(i + 1) == '"') { sb.append('"'); i += 2; }
                    else if (c == '"') { i++; break; }
                    else if (c == '\\') && i + 1 < s.length() { sb.append(s.charAt(i + 1)); i += 2; }
                    else { sb.append(c); i++; }
                }
                res.add(sb.toString()); // "" es una cadena vacía, no nulo
                while (i < s.length() && s.charAt(i) != ',') i++;
            } else {
                int fin = s.indexOf(',', i);
                if (fin < 0) fin = s.length();
                String v = s.substring(i, fin);
                res.add(v.isEmpty() ? null : v); // vacío sin comillas = nulo
                i = fin;
            }
            if (i >= s.length()) break;
            i++;
        }
        return res;
    }
}

```

```

    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String l = sc.nextLine();
            if (l.isBlank()) continue;
            String tipo = l.substring(0, l.indexOf(' ')), texto = l.substring(l.indexOf(' ') + 1).trim();
            List<String> v = tipo.equals("array") ? leerArray(texto) : leerCompuesto(texto);
            System.out.println(tipo + " " + json(v) + " · " + v.size() + (v.size() == 1 ? " valor" : " valores")
                               + " · nulos: " + v.stream().filter(Objects::isNull).count());
        }
    }
}

```

Los dos analizadores siguen el mismo esquema: leer un valor, saltar hasta la coma y repetir hasta el final. Cada valor puede ir entre comillas (y entonces se lee carácter a carácter atendiendo a los escapes) o sin ellas (y entonces llega hasta la siguiente coma).

La diferencia está en cómo se representa el nulo: en un array es la palabra `NULL` sin comillas; en un tipo compuesto es un campo vacío. Por eso `"NULL"` entre comillas es un texto y `"` es una cadena vacía: las comillas indican siempre que hay un valor.

Fíjate en la coma final de un compuesto: tras saltarla, la siguiente vuelta encuentra un campo vacío y lo añade como nulo, que es justo lo que significa.

13. Profundidad de activación en una base de objetos

Difícil · BD Objeto-Relacionales y Orientadas a Objetos · apuntesdam.com/subject/acceso-datos/topic/bd-objeto-

Solución (java)

```

import java.util.*;

public class Main {
    static class Objeto {
        String id, clase;
        Map<String, String> datos = new LinkedHashMap<>(); // campos simples
        Map<String, String> referencias = new LinkedHashMap<>(); // campo → id del objeto apuntado
    }

    static Map<String, Objeto> almacen = new LinkedHashMap<>();
    static Set<String> activados = new HashSet<>();

    static void activar(String id, int profundidad, String sangria, String prefijo) {
        Objeto o = almacen.get(id);
        if (o == null) {
            System.out.println(sangria + prefijo + "null (" + id + " no existe)");
            return;
        }
        if (profundidad <= 0) { // se queda como referencia sin rellenar
            System.out.println(sangria + prefijo + o.clase + " " + o.id + " (sin activar)");
            return;
        }
        activados.add(id);
        System.out.println(sangria + prefijo + o.clase + " " + o.id + " " + o.datos);
        for (Map.Entry<String, String> r : o.referencias.entrySet()) {
            activar(r.getValue(), profundidad - 1, sangria + " ", r.getKey() + " → ");
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            if (p[0].equals("objeto")) { // objeto ID Clase campo=valor ref=@ID ...
                Objeto o = new Objeto();
                o.id = p[1]; o.clase = p[2];
                for (int i = 3; i < p.length; i++) {
                    String[] kv = p[i].split("=");
                    if (kv[1].startsWith("@")) o.referencias.put(kv[0], kv[1].substring(1));
                    else o.datos.put(kv[0], kv[1]);
                }
                almacen.put(o.id, o);
            } else if (p[0].equals("activar")) { // activar ID profundidad
                activados.clear();
                System.out.println("== activar " + p[1] + " con profundidad " + p[2]);
                activar(p[1], Integer.parseInt(p[2]), "", "");
                System.out.println("Objetos leídos del disco: " + activados.size());
            }
        }
    }
}

```

Es un recorrido recursivo del grafo con un límite: cada llamada activa el objeto y llama a **activar** con cada referencia y una profundidad menos. Cuando la profundidad llega a 0 se para, y por eso los ciclos ($P1 \rightarrow D1 \rightarrow P1 \dots$) no producen una recursión infinita: el límite de profundidad hace de caso base.

La sangría y el prefijo se pasan como parámetros y crecen en cada nivel, así cada línea sabe dónde dibujarse sin variables globales. El conjunto **activados** cuenta los objetos distintos: si un objeto aparece dos veces en el árbol, solo se lee una vez del disco.

o.datos se imprime directamente porque el **toString** de un mapa ya tiene el formato **{nombre=Ana}**.

14. Herencia de tablas en PostgreSQL

Muy difícil · BD Objeto-Relacionales y Orientadas a Objetos · apuntesdam.com/subject/acceso-datos/topic/bd-objeto-

```

import java.util.*;

public class Main {
    static class Tabla {
        String nombre, padre, unico;
        List<String> columnas = new ArrayList<>();
        List<Map<String, String>> filas = new ArrayList<>();
        List<String> hijas = new ArrayList<>();
    }

    static Map<String, Tabla> tablas = new LinkedHashMap<>();

    // Añade a salida las filas de t (y, si no es ONLY, las de sus hijas) con las columnas de la tabla
    // consultada
    static void recoger(Tabla consultada, Tabla t, boolean solo, List<String> salida) {
        for (Map<String, String> fila : t.filas) {
            List<String> valores = new ArrayList<>();
            for (String c : consultada.columnas) valores.add(fila.getOrDefault(c, "NULL"));
            salida.add(" " + String.join(" | ", valores) + (t == consultada ? "" : " [" + t.nombre +
                "]"));
        }
        if (solo) return;
        for (String h : t.hijas) recoger(consultada, tablas.get(h), false, salida);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            String[] p = l.split(" ");
            switch (p[0]) {
                case "tabla" -> {
                    Tabla t = new Tabla();
                    t.nombre = p[1];
                    List<String> propias = new ArrayList<>();
                    for (int i = 2; i < p.length; i++) {
                        String[] kv = p[i].split("=", 2);
                        switch (kv[0]) {
                            case "columnas" -> propias.addAll(List.of(kv[1].split(",")));
                            case "hereda" -> t.padre = kv[1];
                            case "unico" -> t.unico = kv[1];
                        }
                    }
                    if (t.padre != null) { // las heredadas van primero
                        t.columnas.addAll(tablas.get(t.padre).columnas);
                        tablas.get(t.padre).hijas.add(t.nombre);
                    }
                    t.columnas.addAll(propias);
                    tablas.put(t.nombre, t);
                    System.out.println("CREATE TABLE " + t.nombre + " (" + String.join(", ", t.columnas)
                        + ")
                        + (t.padre != null ? " INHERITS (" + t.padre + ")" : "")
                        + (t.unico != null ? " UNIQUE (" + t.unico + ")" : ""));
                }
                case "insertar" -> {
                    Tabla t = tablas.get(p[1]);
                    Map<String, String> fila = new LinkedHashMap<>();
                    String error = null;
                    for (String par : l.split(" ", 3)[2].split(",")) {
                        String[] kv = par.split("=", 2);
                        if (!t.columnas.contains(kv[0]) && error == null) error = "ERROR: la columna " +
                            kv[0] + " no existe en " + t.nombre;
                        fila.put(kv[0], kv[1]);
                    }
                    if (error == null && t.unico != null && fila.get(t.unico) != null) {
                        String v = fila.get(t.unico);
                        // UNIQUE solo mira las filas de esta tabla, no las de sus hijas ni las del padre
                    }
                }
            }
        }
    }
}

```

```

        for (Map<String, String> f : t.filas) {
            if (v.equals(f.get(t.unico))) { error = "ERROR: valor duplicado " + t.unico +
                "=" + v + " en " + t.nombre; break; }
        }
    }
    if (error != null) System.out.println(error);
    else {
        t.filas.add(fila);
        System.out.println("INSERT en " + t.nombre);
    }
}
case "consultar" -> {
    boolean solo = p[1].equals("solo");
    Tabla t = tablas.get(solo ? p[2] : p[1]);
    System.out.println("SELECT * FROM " + (solo ? "ONLY " : "") + t.nombre + " (" +
        String.join(" | ", t.columnas) + ")");
    List<String> salida = new ArrayList<>();
    recoger(t, t, solo, salida);
    salida.forEach(System.out::println);
    System.out.println("   " + salida.size() + (salida.size() == 1 ? " fila" : " filas"));
}
default -> { }
}
}
}
}

```

Cada tabla guarda sus columnas completas (las del padre primero) y la lista de sus hijas. Así, consultar una tabla es mostrar sus filas y, si no es ONLY, llamar recursivamente a `recoger` con cada hija, que a su vez hará lo mismo con las suyas: un recorrido en profundidad del árbol de herencia.

Al recoger se usan siempre las columnas de la tabla consultada, no las de la tabla de la fila: por eso una fila de `delegado` vista desde `persona` solo muestra dni y nombre, y una columna que la fila no tiene sale como NULL con `getOrDefault`.

La comprobación de UNIQUE recorre solo `t.filas`, que es justo el comportamiento real (y sorprendente) de PostgreSQL: el mismo dni puede estar en `persona` y en `alumno`, y la consulta al padre los mostrará duplicados.

15. Propiedad ligada (bound property)

Fácil · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Solución (java)

```
import java.beans.*;
import java.io.Serializable;
import java.util.Scanner;

public class Main {

    public static class Termostato implements Serializable {
        private final PropertyChangeSupport soporte = new PropertyChangeSupport(this);
        private int temperatura = 20;

        public Termostato() {}

        public int getTemperatura() { return temperatura; }

        public void setTemperatura(int nueva) {
            int anterior = temperatura;
            temperatura = nueva;
            soporte.firePropertyChange("temperatura", anterior, nueva);
        }

        public void addPropertyChangeListener(PropertyChangeListener l) {
            soporte.addPropertyChangeListener(l); }
        public void removePropertyChangeListener(PropertyChangeListener l) {
            soporte.removePropertyChangeListener(l); }
    }

    public static void main(String[] args) {
        Termostato t = new Termostato();
        t.addPropertyChangeListener(e ->
            System.out.println(e.getPropertyName() + ": " + e.getOldValue() + " -> " + e.getNewValue()));

        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) t.setTemperatura(sc.nextInt());
        System.out.println("Final: " + t.getTemperatura());
    }
}
```

Antes de cambiar el valor se guarda el anterior, porque el evento necesita los dos. `firePropertyChange` se encarga del resto: avisa a todos los oyentes registrados y, si el valor viejo y el nuevo son iguales, no avisa a nadie. Por eso no hace falta comprobarlo a mano.

16. Propiedad restringida (constrained)

Medio · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Solución (java)

```
import java.beans.*;
import java.util.Scanner;

public class Main {

    public static class CuentaBancaria {
        private final VetoableChangeSupport vetos = new VetoableChangeSupport(this);
        private double saldo = 100;

        public double getSaldo() { return saldo; }

        public void setSaldo(double nuevo) throws PropertyVetoException {
            vetos.fireVetoableChange("saldo", saldo, nuevo);    // si alguien veta, lanza la excepción y
            // no seguimos
            saldo = nuevo;
        }

        public void addVetoableChangeListener(VetoableChangeListener l) {
            vetos.addVetoableChangeListener(l); }
    }

    public static void main(String[] args) {
        CuentaBancaria cuenta = new CuentaBancaria();
        cuenta.addVetoableChangeListener(e -> {
            if ((double) e.getNewValue() < 0) throw new PropertyVetoException("saldo negativo", e);
        });

        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            int movimiento = sc.nextInt();
            try {
                cuenta.setSaldo(cuenta.getSaldo() + movimiento);
                System.out.println("OK " + movimiento + " -> " + cuenta.getSaldo());
            } catch (PropertyVetoException ex) {
                System.out.println("Vetado " + movimiento + ": " + ex.getMessage());
            }
        }
    }
}
```

El orden es lo importante: primero se pregunta a los oyentes con `fireVetoableChange` y solo después se cambia el saldo. Si un oyente veta, la excepción sale del método antes de llegar a la asignación, así que el valor nunca llega a cambiar.

Es la diferencia con una propiedad ligada: allí se avisa después del cambio; aquí se pide permiso antes.

17. Persistencia de un JavaBean en XML

Medio · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Solución (java)

```
import java.beans.*;
import java.io.*;
import java.util.Scanner;

public class Main {

    public static class Configuracion {
        private String tema = "claro";
        private int tamanoLetra = 14;

        public Configuracion() {}
        public String getTema() { return tema; }
        public void setTema(String tema) { this.tema = tema; }
        public int getTamanoLetra() { return tamanoLetra; }
        public void setTamanoLetra(int t) { this.tamanoLetra = t; }
    }

    static void guardar(Configuracion c, File fichero) throws IOException {
        try (XMLEncoder enc = new XMLEncoder(new BufferedOutputStream(new FileOutputStream(fichero)))) {
            enc.writeObject(c);
        }
    }

    static Configuracion cargar(File fichero) throws IOException {
        try (XMLDecoder dec = new XMLDecoder(new BufferedInputStream(new FileInputStream(fichero)))) {
            return (Configuracion) dec.readObject();
        }
    }

    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        Configuracion c = new Configuracion();
        c.setTema(sc.next());
        c.setTamanoLetra(sc.nextInt());

        File f = new File("config.xml");
        guardar(c, f);
        Configuracion leida = cargar(f);
        System.out.println(leida.getTema() + " " + leida.getTamanoLetra());
        System.out.println("XML generado: " + (f.length() > 0));
    }
}
```

XMLEncoder recorre los getters del bean y escribe en XML las propiedades que tienen un valor distinto del que daría el constructor vacío. **XMLDecoder** hace lo contrario: crea el objeto con ese constructor y llama a los setters. Por eso el bean tiene que seguir las convenciones de JavaBeans.

Los dos implementan **AutoCloseable**, así que se usan con **try** con recursos. Es importante cerrar el encoder: hasta que se cierra no se termina de escribir el documento.

18. Componente DAO con eventos

Difícil · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-datos

Solución (java)

```
import java.util.*;

public class Main {

    public interface DaoListener { void datosCambiados(String accion, String nombre); }

    public interface AlumnoDao {
        void guardar(String nombre, double nota);
        boolean borrar(String nombre);
        Map<String, Double> listar();
        void addDaoListener(DaoListener l);
    }

    public static class AlumnoDaoMemoria implements AlumnoDao {
        private final Map<String, Double> alumnos = new TreeMap<>();
        private final List<DaoListener> oyentes = new ArrayList<>();

        private void avisar(String accion, String nombre) {
            for (DaoListener l : oyentes) l.datosCambiados(accion, nombre);
        }

        public void guardar(String nombre, double nota) {
            alumnos.put(nombre, nota);
            avisar("guardado", nombre);
        }

        public boolean borrar(String nombre) {
            if (alumnos.remove(nombre) == null) return false; // no existía: no hay nada que avisar
            avisar("borrado", nombre);
            return true;
        }

        public Map<String, Double> listar() { return Collections.unmodifiableMap(alumnos); }

        public void addDaoListener(DaoListener l) { oyentes.add(l); }
    }

    public static void main(String[] args) {
        AlumnoDao dao = new AlumnoDaoMemoria();
        dao.addDaoListener((accion, nombre) -> System.out.println("[evento] " + accion + " " + nombre));

        Scanner sc = new Scanner(System.in);
        while (sc.hasNext()) {
            String orden = sc.next();
            switch (orden) {
                case "G" -> dao.guardar(sc.next(), Double.parseDouble(sc.next()));
                case "B" -> { String n = sc.next(); if (!dao.borrar(n)) System.out.println("No existe " + n); }
                case "L" -> dao.listar().forEach((n, nota) -> System.out.println(n + " " + nota));
                default -> System.out.println("Orden desconocida");
            }
        }
    }
}
```

Un método privado `avisar` recorre los oyentes, y tanto `guardar` como `borrar` lo llaman después de modificar los datos. `Map.remove` devuelve el valor que había (o `null` si la clave no existía), lo que sirve para saber si de verdad se borró algo sin buscarlo antes.

El `main` solo conoce la interfaz `AlumnoDao`: podría cambiarse la implementación en memoria por una con JDBC sin tocar el resto del programa. Esa es la ventaja del patrón DAO.

19. Un contenedor de inyección de dependencias

Muy difícil · Componentes de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/componentes-acceso-

datos

Solución (java)

```
import java.util.*;

public class Main {
    static Map<String, List<String>> dependencias = new LinkedHashMap<>();
    static Map<String, String> ambitos = new HashMap<>();
    static Map<String, Integer> singletons = new HashMap<>(); // componente → número de instancia
    static List<String> creados = new ArrayList<>(); // singletons en orden de creación (para cerrar)
    static int contador = 0;

    // Devuelve el primer error del grafo de dependencias de nombre, o null si es válido
    static String validar(String nombre, List<String> camino, String quien) {
        if (!dependencias.containsKey(nombre)) {
            return quien == null ? "no existe el componente " + nombre
                : "falta el componente " + nombre + " (lo necesita " + quien + ")";
        }
        int i = camino.indexOf(nombre);
        if (i >= 0) { // ya estamos dentro de él: ciclo
            List<String> ciclo = new ArrayList<>(camino.subList(i, camino.size()));
            ciclo.add(nombre);
            return "dependencia circular " + String.join(" → ", ciclo);
        }
        camino.add(nombre);
        for (String d : dependencias.get(nombre)) {
            String error = validar(d, camino, nombre);
            if (error != null) return error;
        }
        camino.remove(camino.size() - 1);
        return null;
    }

    // Crea (o reutiliza) el componente y sus dependencias; devuelve su número de instancia
    static int resolver(String nombre) {
        boolean prototype = ambitos.get(nombre).equals("prototype");
        if (!prototype && singletons.containsKey(nombre)) return singletons.get(nombre);
        for (String d : dependencias.get(nombre)) resolver(d);
        int n = ++contador;
        System.out.println(" crear " + nombre + "#" + n + (prototype ? " (prototype)" : ""));
        if (!prototype) {
            singletons.put(nombre, n);
            creados.add(nombre);
        }
        return n;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            switch (p[0]) {
                case "componente" -> { // componente N [depende=A,B]
                    [ambito=prototype]
                    List<String> deps = new ArrayList<>();
                    String ambito = "singleton";
                    for (int i = 2; i < p.length; i++) {
                        if (p[i].startsWith("depende="))
                            deps.addAll(List.of(p[i].substring(8).split(",")));
                        if (p[i].startsWith("ambito=")) ambito = p[i].substring(7);
                    }
                    dependencias.put(p[1], deps);
                    ambitos.put(p[1], ambito);
                }
                case "obtener" -> {
                    System.out.println("obtener " + p[1]);
                    String error = validar(p[1], new ArrayList<>(), null);
                    if (error != null) {
                        System.out.println(" Error: " + error);
                        break;
                    }
                }
            }
        }
    }
}
```

```

    }
    boolean existia = !ambitos.get(p[1]).equals("prototype") &&
        singletons.containsKey(p[1]);
    int n = resolver(p[1]);
    System.out.println("→ " + p[1] + "#" + n + (existia ? " (ya existía)" : ""));
}
case "cerrar" -> {
    for (int i = creados.size() - 1; i >= 0; i--) {
        String c = creados.get(i);
        System.out.println("  destruir " + c + "#" + singletons.get(c));
    }
    creados.clear();
    singletons.clear();
    System.out.println("Contenedor cerrado");
}
}
}
}
}

```

La validación es un recorrido en profundidad del grafo que lleva el camino actual: si una dependencia ya está en el camino, hay un ciclo, y el trozo de camino desde su primera aparición es justo lo que hay que mostrar. Al volver de un componente se quita del camino, así dos ramas que comparten una dependencia (DataSource) no se confunden con un ciclo.

`resolver` es el mismo recorrido, pero creando: primero las dependencias y después el componente, que es el orden en que un contenedor puede inyectarlas. Los singletons se guardan en un mapa y se devuelven sin crear nada si ya existen; los prototype se crean siempre.

Destruir en orden inverso a la creación garantiza que ningún componente se destruya antes que los que dependen de él.

20. Filtrar un fichero de log

Fácil · Ejercicios de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/ejercicios-acceso-datos

Solución (java)

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class Main {

    static int filtrar(Path origen, Path destino) throws IOException {
        List<String> errores = new ArrayList<>();
        for (String linea : Files.readAllLines(origen)) {
            if (linea.contains("ERROR")) errores.add(linea);
        }
        Files.write(destino, errores);
        return errores.size();
    }

    public static void main(String[] args) throws IOException {
        Path log = Path.of("app.log");
        Files.write(log, new String(System.in.readAllBytes()).lines().toList());

        Path errores = Path.of("errores.log");
        int n = filtrar(log, errores);
        System.out.println("Errores: " + n);
        if (Files.exists(errores)) Files.readAllLines(errores).forEach(System.out::println);
    }
}
```

`Files.readAllLines` lee el fichero entero como una lista de líneas y `Files.write` escribe una lista de líneas de golpe: con ficheros pequeños es la forma más corta de trabajar con NIO. Se filtran las líneas que contienen «ERROR» y el tamaño de la lista es el número de copias.

Con ficheros de log enormes convendría no cargarlo todo en memoria y usar `Files.lines` (un stream) o un `BufferedReader`.

21. Conversor de CSV a JSON

Medio · Ejercicios de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/ejercicios-acceso-datos

Solución (java)

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class Main {

    static String convertir(Path csv) throws IOException {
        List<String> lineas = Files.readAllLines(csv);
        String[] cabecera = lineas.get(0).split(";");
        List<String> objetos = new ArrayList<>();
        for (int i = 1; i < lineas.size(); i++) {
            if (lineas.get(i).isBlank()) continue;
            String[] valores = lineas.get(i).split(";");
            List<String> pares = new ArrayList<>();
            for (int k = 0; k < cabecera.length; k++) {
                String v = valores[k];
                // los números van sin comillas; el texto, entre comillas
                String json = v.matches("-?\\d+(\\.\\d+)?") ? v : "\"" + v.replace("\"", "\\\"") + "\"";
                pares.add("\"" + cabecera[k] + "\": " + json);
            }
            objetos.add("{ " + String.join(", ", pares) + " }");
        }

        return "[" + String.join(", ", objetos) + "]";
    }

    public static void main(String[] args) throws IOException {
        Path csv = Path.of("personas.csv");
        Files.write(csv, new String(System.in.readAllBytes()).lines().toList());
        System.out.println(convertir(csv));
    }
}
```

Cada línea de datos se convierte en un objeto JSON emparejando cada valor con el nombre de su columna de la cabecera, y los objetos se unen con comas dentro de los corchetes. Construirlo con listas y `String.join` evita tener que controlar si sobra una coma al final.

Los valores que son números se escriben sin comillas y el resto entre comillas, escapando las comillas que pudiera haber dentro del texto para que el JSON siga siendo válido.

22. Media de precios por categoría (DOM)

Difícil · Ejercicios de Acceso a Datos · apuntesdam.com/subject/acceso-datos/topic/ejercicios-acceso-datos

Solución (java)

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Main {

    static Map<String, Double> medias(File fichero) throws Exception {
        Map<String, Double> resultado = new TreeMap<>();
        Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(fichero);
        NodeList productos = doc.getElementsByTagName("producto");
        Map<String, Double> sumas = new HashMap<>();
        Map<String, Integer> cuentas = new HashMap<>();
        for (int i = 0; i < productos.getLength(); i++) {
            Element p = (Element) productos.item(i);
            String categoria = p.getAttribute("categoria");
            double precio =
                Double.parseDouble(p.getElementsByTagName("precio").item(0).getTextContent().trim());
            sumas.merge(categoria, precio, Double::sum);
            cuentas.merge(categoria, 1, Integer::sum);
        }
        for (String c : sumas.keySet()) resultado.put(c, sumas.get(c) / cuentas.get(c));

        return resultado;
    }

    public static void main(String[] args) throws Exception {
        File f = new File("catalogo.xml");
        Files.write(f.toPath(), System.in.readAllBytes());
        medias(f).forEach((cat, media) -> System.out.printf(Locale.US, "%s %.2f%n", cat, media));
    }
}
```

Para cada `<producto>` se lee el atributo con `getAttribute` y el precio buscando el elemento hijo con `getElementsByTagName("precio")` y tomando su texto con `getTextContent()`.

La media necesita la suma y el número de elementos de cada categoría, así que se acumulan en dos mapas con `merge` y al final se divide. El resultado es un `TreeMap`, por eso las categorías salen en orden alfabético.

23. Conversor de CSV a JSON con comillas y tipos

Difícil · Java · 75 minutos · apuntesdam.com/ejercicios/java/conversor-csv-a-json

Solución (java)

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {
    /** Campos de una línea CSV separados por comas, con comillas dobles opcionales ("" es una comilla).
    null si hay comillas sin cerrar. */
    static List<String> campos(String linea) {
        List<String> lista = new ArrayList<>();
        StringBuilder actual = new StringBuilder();
        boolean entreComillas = false;
        for (int i = 0; i < linea.length(); i++) {
            char c = linea.charAt(i);
            if (entreComillas) {
                if (c != '"') {
                    actual.append(c);
                } else if (i + 1 < linea.length() && linea.charAt(i + 1) == '"') {
                    actual.append('');
                    i++;
                } else {
                    entreComillas = false;
                }
            } else if (c == '"') {
                entreComillas = true;
            } else if (c == ',') {
                lista.add(actual.toString().trim());
                actual.setLength(0);
            } else {
                actual.append(c);
            }
        }
        if (entreComillas) return null;
        lista.add(actual.toString().trim());
        return lista;
    }

    /** Texto JSON entre comillas, con las comillas, barras y tabuladores escapados. */
    static String cadena(String s) {
        StringBuilder sb = new StringBuilder("\"");
        for (char c : s.toCharArray()) {
            switch (c) {
                case '"' -> sb.append("\\\"");
                case '\\' -> sb.append("\\\\");
                case '\t' -> sb.append("\\t");
                default -> sb.append(c);
            }
        }
        return sb.append('"').toString();
    }

    /** El valor JSON de un campo: null, número, booleano o texto. */
    static String valor(String campo) {
        if (campo.isEmpty()) return "null";
        // Un número JSON no admite ceros a la izquierda: «08001» es un código postal, no el 8001
        if (campo.matches("-?(0|[1-9]\\d*)(\\.\\d+)?")) return campo;
        if (campo.equals("true") || campo.equals("false")) return campo;
        return cadena(campo);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<String> cabecera = null;
        List<String> objetos = new ArrayList<>();
        List<Integer> ignoradas = new ArrayList<>();
        int numero = 0;
        while (sc.hasNextLine()) {
```

```

        String linea = sc.nextLine();
        numero++;
        if (linea.isBlank()) continue;
        List<String> c = campos(linea);
        if (cabecera == null) {
            cabecera = c;
            continue;
        }
        if (c == null || c.size() != cabecera.size()) {
            ignoradas.add(numero);
            continue;
        }
        StringBuilder obj = new StringBuilder(" {\n");
        for (int i = 0; i < c.size(); i++) {
            obj.append("    ").append(cadena(cabecera.get(i))).append(": ").append(valor(c.get(i)));
            obj.append(i < c.size() - 1 ? ",\n" : "\n");
        }
        objetos.add(obj.append(" }").toString());
    }

    if (cabecera == null) {
        System.out.println("El fichero está vacío");
        return;
    }
    System.out.println(objetos.isEmpty() ? "[" : "[\n" + String.join(",\n", objetos) + "\n]");
    String lista = ignoradas.isEmpty() ? "" : " (" + String.join(", ",
        ignoradas.stream().map(String::valueOf).toList()) + ")";
    System.out.println("Registros: " + objetos.size() + " · Líneas ignoradas: " + ignoradas.size() +
        lista);
}
}

```

El analizador de líneas es una máquina de estados con dos estados (dentro o fuera de comillas). Es la técnica general para leer formatos con delimitadores y escapes, y explica por qué en producción se usan librerías (OpenCSV, Apache Commons CSV) en lugar de `split`.

Separar el análisis (`campos`), la decisión de tipo (`valor`) y el escape (`cadena`) en métodos pequeños permite probar cada uno por separado y reutilizar el escape para las claves.

La regla del número sigue la gramática de JSON: por eso `08001` y `007` siguen siendo texto. Es un error real y frecuente al importar códigos postales, teléfonos o DNI desde hojas de cálculo, que se comen los ceros de la izquierda.

La salida se monta en memoria y se escribe al final: así el JSON es válido aunque haya líneas ignoradas por el medio, y el resumen va después sin mezclarse con él.

24. Fichero de productos con registros de tamaño fijo y acceso directo

Muy difícil · Java · 90 minutos · apuntesdam.com/ejercicios/java/fichero-de-registros-con-acceso-directo

Solución (java)

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Locale;
import java.util.Scanner;

public class Main {
    static final int NOMBRE = 30; // caracteres del nombre
    static final int TAM = 4 + NOMBRE * 2 + 8 + 4 + 1; // código + nombre + precio + stock + borrado
    = 77 bytes

    static RandomAccessFile f;

    /** Acceso directo: el registro del código N empieza en el byte (N - 1) × 77. */
    static long posicion(int codigo) {
        return (long) (codigo - 1) * TAM;
    }

    /** Un registro está activo si existe, guarda ese código y no tiene la marca de borrado. */
    static boolean activo(int codigo) throws IOException {
        if (posicion(codigo) + TAM > f.length()) return false;
        f.seek(posicion(codigo));
        int guardado = f.readInt();
        f.seek(posicion(codigo) + TAM - 1);
        return guardado == codigo && !f.readBoolean();
    }

    static void escribir(int codigo, String nombre, double precio, int stock) throws IOException {
        f.seek(posicion(codigo));
        f.writeInt(codigo);
        StringBuilder sb = new StringBuilder(nombre);
        sb.setLength(NOMBRE); // recorta o rellena con '\0' hasta 30
        caracteres
        f.writeChars(sb.toString());
        f.writeDouble(precio);
        f.writeInt(stock);
        f.writeBoolean(false);
    }

    /** Lee el registro del código (que debe estar activo) como texto. */
    static String leer(int codigo) throws IOException {
        f.seek(posicion(codigo) + 4);
        StringBuilder nombre = new StringBuilder();
        for (int i = 0; i < NOMBRE; i++) {
            char c = f.readChar();
            if (c != '\0') nombre.append(c);
        }
        double precio = f.readDouble();
        int stock = f.readInt();
        return codigo + " · " + nombre + " · " + euros(precio) + " · " + stock + " uds";
    }

    static String euros(double x) {
        return String.format(Locale.ROOT, "%.2f", x).replace('.', ',') + " €";
    }

    static void orden(String linea) throws IOException {
        String[] p = linea.split("\\s+", 2);
        String args = p.length > 1 ? p[1] : "";
        switch (p[0].toUpperCase()) {
            case "ALTA" -> {
                String[] c = args.split(";", -1);
                if (c.length != 4 || !c[0].matches("\\d{1,4}") || Integer.parseInt(c[0]) == 0 ||
                    c[1].isBlank() || !c[2].matches("\\d+(\\.\\d{1,2})?") || !c[3].matches("\\d+")) {
                    System.out.println("Datos no válidos");
                }
            }
        }
    }
}
```

```

        return;
    }
    int codigo = Integer.parseInt(c[0]);
    if (activo(codigo)) {
        System.out.println("Ya existe el producto " + codigo);
        return;
    }
    String nombre = c[1].trim();
    escribir(codigo, nombre, Double.parseDouble(c[2]), Integer.parseInt(c[3]));
    System.out.println("Alta del producto " + codigo + " en el byte " + posicion(codigo)
        + (nombre.length() > NOMBRE ? " (nombre recortado a " + NOMBRE + " caracteres)" :
        ""));
}
case "PRECIO" -> {
    String[] c = args.split("\\s+");
    if (c.length != 2 || !c[0].matches("\\d{1,4}") || !c[1].matches("\\d+(\\.\\d{1,2})?")) {
        System.out.println("Datos no válidos");
    } else if (!activo(Integer.parseInt(c[0]))) {
        System.out.println("No existe el producto " + c[0]);
    } else {
        // Solo se reescriben los 8 bytes del precio: no hace falta leer ni escribir el resto
        // del registro
        long donde = posicion(Integer.parseInt(c[0])) + 4 + NOMBRE * 2;
        f.seek(donde);
        double antes = f.readDouble();
        f.seek(donde);
        f.writeDouble(Double.parseDouble(c[1]));
        System.out.println("Precio del producto " + c[0] + ": " + euros(antes) + " → " +
            euros(Double.parseDouble(c[1])));
    }
}
case "BAJA" -> {
    if (!args.matches("\\d{1,4}") || !activo(Integer.parseInt(args))) {
        System.out.println("No existe el producto " + args);
    } else {
        f.seek(posicion(Integer.parseInt(args)) + TAM - 1);
        f.writeBoolean(true); // borrado lógico: el registro sigue ahí,
        // marcado
        System.out.println("Baja lógica del producto " + args);
    }
}
case "VER" -> System.out.println(args.matches("\\d{1,4}") && activo(Integer.parseInt(args)) ?
    leer(Integer.parseInt(args)) : "No existe el producto " + args);
case "LISTAR" -> {
    int total = (int) (f.length() / TAM);
    int activos = 0;
    StringBuilder sb = new StringBuilder();
    for (int codigo = 1; codigo <= total; codigo++) {
        if (activo(codigo)) {
            activos++;
            sb.append(" ").append(leer(codigo)).append('\n');
        }
    }
    System.out.print("Productos activos: " + activos + "\n" + sb);
}
case "INFORME" -> {
    int total = (int) (f.length() / TAM);
    int activos = 0;
    double valor = 0;
    for (int codigo = 1; codigo <= total; codigo++) {
        if (!activo(codigo)) continue;
        activos++;
        f.seek(posicion(codigo) + 4 + NOMBRE * 2);
        double precio = f.readDouble();
        valor += precio * f.readInt();
    }
    System.out.println("Fichero: " + f.length() + " bytes, " + total + " huecos de registro,
        " + activos
        + " activos, " + (total - activos) + " libres o borrados · valor del stock " +
        euros(valor));
}
default -> System.out.println("Orden desconocida");

```

```

    }
}

public static void main(String[] args) throws IOException {
    File fichero = File.createTempFile("productos", ".dat");
    fichero.deleteOnExit();
    try (RandomAccessFile raf = new RandomAccessFile(fichero, "rw")) {
        f = raf;
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String linea = sc.nextLine().trim();
            if (!linea.isEmpty()) orden(linea);
        }
    }
}
}

```

El tamaño fijo es lo que hace posible el acceso directo: con una multiplicación se sabe dónde está cualquier registro, sin leer los anteriores. Es el mismo principio que usan los gestores de bases de datos al organizar los datos en páginas de tamaño fijo.

`RandomAccessFile` tiene un puntero que avanza con cada lectura o escritura; `seek` lo coloca donde se quiera. Por eso cambiar el precio son dos `seek`: uno para leer el valor anterior y otro para volver a la misma posición antes de escribir.

El borrado lógico cambia un solo byte en lugar de reorganizar el fichero. El precio a pagar es que el fichero no encoge: con el tiempo se llena de huecos, y por eso los sistemas reales tienen operaciones de compactación o reutilizan los huecos (aquí, un alta con el mismo código).

Los huecos que se crean al escribir más allá del final contienen ceros. Guardar el propio código dentro del registro permite distinguir un registro real de un hueco sin una tabla aparte.