Chapter 11: Database Programming

Object Oriented Programming with Java

Persistent storage | The relational model | SQLite | SQL commands | JOIN and GROUP BY | JDBC | Insert, read, update, delete | PreparedStatement | Transactions

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_11_Database_Programming.html

INTRODUCTION

Learning Outcomes

Concepts

Explain why databases are needed, how they differ from files, and the relational model (tables, rows, columns, primary keys).

SQL

Write `CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`; narrow queries with `WHERE`, `ORDER BY`, `LIMIT`; combine tables with `JOIN` and group with `GROUP BY`.

JDBC

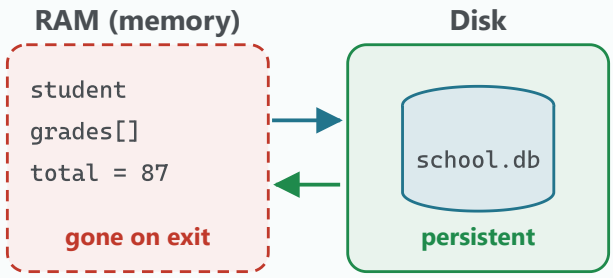
Connect from Java to SQLite with `Connection`, `Statement` and `ResultSet` to insert, read, update and delete records; close resources with try-with-resources.

Security and integrity

Prevent SQL injection with `PreparedStatement`; run several statements as one transaction with `commit` / `rollback`.

Why Do Databases Matter?

- A program's variables and objects live in **RAM**: they are **lost** when the program exits.
- **Persistent storage** keeps data for the long term, after the program has exited. Student grades, bank accounts and game scores must all persist.
- A **database** is an organized collection of data that is easy to query. The software that manages it is a **database management system** (DBMS).
- Analogy: a database is like a library's **cataloged archive**; you find a book without walking every shelf.

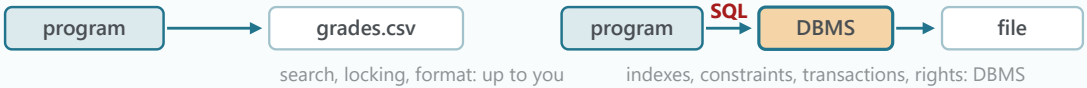


When the program restarts, the data is read from disk.

Files or a Database?

Plain files can be **enough** for some applications (a settings file, a log, a short list). As data grows and gets complex, **you need a database**: it offers key features for processing data efficiently.

Feature	Flat file (txt, csv)	Database
Search	The file is read start to end	Fast with an index; a one-line SQL query
Update	Often the whole file is rewritten	Only the affected row changes
Concurrent access	Two programs writing at once can corrupt data	Safe with locking and transactions
Consistency	Format checking is entirely up to the programmer	Data types and constraints (`NOT NULL` , `UNIQUE` , `CHECK`)
Security	File permissions: all or nothing	Rights per user, table and field (server DBMSs; file permissions in SQLite)
After a crash	A half-done write can corrupt the file	A half-done transaction is rolled back



Key Features

Fast data access

Indexes find a record among millions of rows in no time.

Field-level security

Rights can be granted per table or even per column: only accounting sees the salary column (in server DBMSs, e.g. PostgreSQL/MySQL `GRANT`; SQLite has no users or rights, access depends on file permissions).

Portable, standard APIs

SQL is a standard; in Java, **JDBC** reaches different databases through one interface.

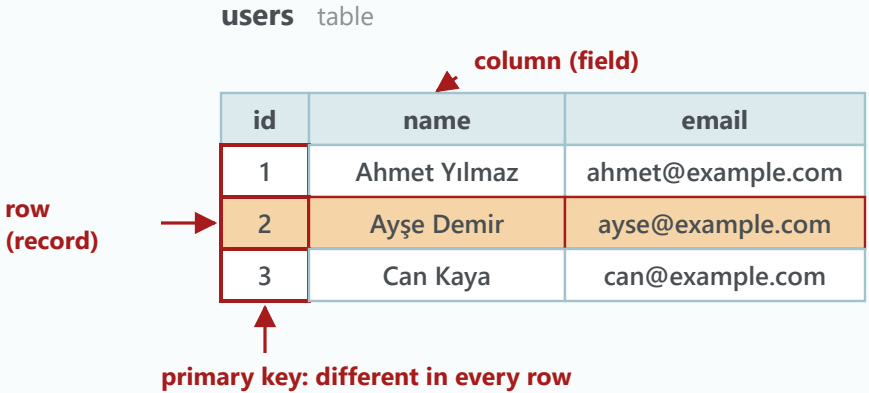
Independence from file layout

The program doesn't know how data is stored. It says **what** it wants in SQL; the DBMS decides **how**.



On top of these, databases offer integrity constraints and transaction support.

The Relational Model: Tables, Rows, Columns



- A **relational** database stores data in **tables**.
- **Column**: one attribute of the data, with a name and a data type (`name TEXT`).
- **Row**: all the information about one entity (one user).
- **Primary key**: identifies each row uniquely; no two rows share an `id` .
- Java analogy: table ≈ **class**, column ≈ **field**, row ≈ **object**.

Kinds of Databases

Relational

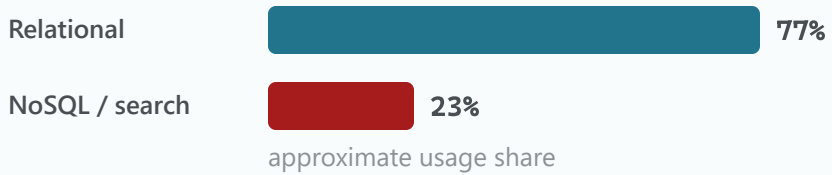
Tables and SQL. **Oracle**, **SQL Server**, **MySQL**, **PostgreSQL**. About 77% of usage.

NoSQL / search

Table-less models such as documents, key–value stores and search indexes, used in specific scenarios: **MongoDB**, **Elasticsearch** (about 23%).

Embedded relational

SQLite: a relational database that runs inside the program; common where little data is shared.



approximate usage share

The examples use SQLite, the easiest to set up. SQL and JDBC code is almost the same for other relational databases.

What Is SQLite?

- An **embedded** relational database: it runs as a **library** inside the program.
- **Serverless**: no separate program, installation or administrator.
- **Self-contained**: all tables are stored in **a single disk file** (e.g. `sample.db`).
- **Cross-platform**: the same file opens on Windows, Linux, macOS and Android.
- Easy to install and use; **open source and free**.
- The most widely used embedded relational database in the world.

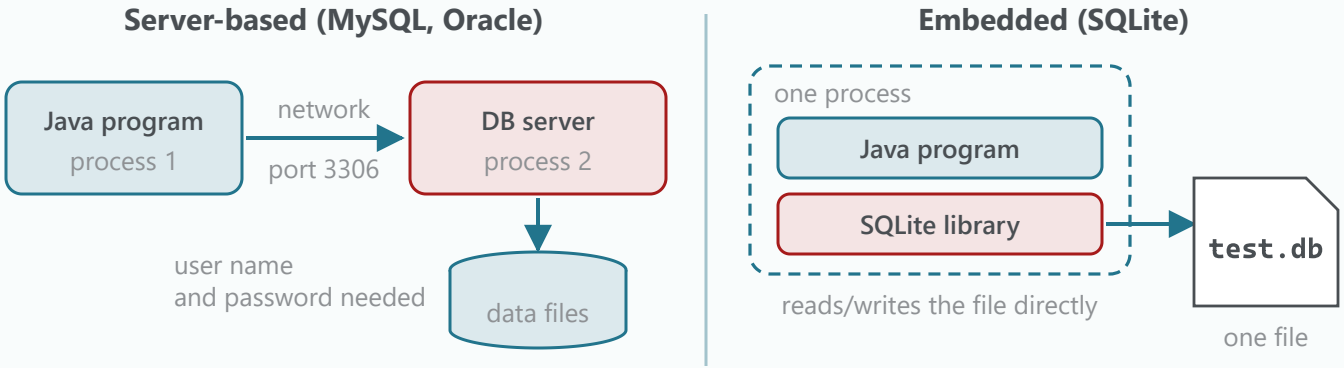
Common uses

+ Mobile apps
+ Embedded systems
+ Small-scale applications
+ Desktop programs such as browsers and e-mail clients

Not a good fit for

– Large systems where many users write at once over a network

Server-Based vs Embedded Databases



	Server-based	SQLite
Setup	Install a server, define users and passwords	Add one jar to the project
Data	Many files managed by the server	A single `.db` file
Concurrent writes	Many users	One writer at a time

The SQLite Driver and DB Browser

SQLite JDBC driver

The jar that lets Java talk to SQLite (`sqlite-jdbc-3.x.jar`): github.com/xerial/sqlite-jdbc

DB Browser for SQLite

A popular SQL client for SQLite: opens a `.db` file, shows the tables visually and runs SQL: sqlitebrowser.org/dl

Adding the driver to a project in Eclipse

1. Download the jar and put it in a `lib` folder in the project.
2. Right-click the project → **Build Path** → **Configure Build Path...**
3. **Libraries** tab → **Classpath** → **Add JARs...** → pick the jar → **Apply and Close**.

From the command line

```
javac Connect.java
java -cp ".;sqlite-jdbc-3.40.0.0.jar" Connect
```

On Linux and macOS the separator is `:` instead of `;`. Compiling needs no driver; the `java.sql` package ships with the JDK.

SQL BASICS

Common SQL Commands

Command	Purpose	Example	CRUD
`CREATE TABLE`	Creates a new table	`CREATE TABLE users (id INTEGER, name TEXT)`	—
`INSERT INTO`	Adds a new record	`INSERT INTO users(name) VALUES('Ali')`	**C**reate
`SELECT`	Queries the table	`SELECT * FROM users`	**R**ead
`UPDATE`	Updates existing records	`UPDATE users SET name = 'Efe' WHERE id = 1`	**U**pdate
`DELETE`	Deletes records	`DELETE FROM users WHERE id = 1`	**D**elete



- **SQL** (Structured Query Language) is the common language of relational databases.
- Keywords are case-insensitive; they are written in CAPITALS for readability. Text values go in **single quotes**: `'Ali'`.
- `*` means “all columns”. `WHERE` picks the rows a command applies to.

CREATE TABLE and Data Types

```
CREATE TABLE IF NOT EXISTS members (
  id    INTEGER PRIMARY KEY,    -- unique
  name  TEXT NOT NULL,          -- required
  email TEXT UNIQUE,            -- no repeats
  age   INTEGER CHECK (age >= 0)
)
```

- `IF NOT EXISTS`: no error if the table already exists.
- `INTEGER PRIMARY KEY`: if no value is given, SQLite assigns **the largest id + 1**.
- A command that breaks a constraint is rejected: `NOT NULL constraint failed: members.name`, `CHECK constraint failed: age >= 0`.

SQLite type	Java counterpart
`INTEGER`	`int`, `long` · `getInt`
`REAL`	`double` · `getDouble`
`TEXT`	`String` · `getString`
`BLOB`	`byte[]` · `getBytes`
`NULL`	no value

SQL Commands, Step by Step

```
CREATE TABLE users (id INTEGER PRIMARY KEY,
                    name TEXT, email TEXT)
```

users

id	name	email
(table empty)		

The table is created: three columns, **no rows** yet.

SQL Commands, Step by Step

```
INSERT INTO users(name, email)
  VALUES('Ahmet Yılmaz', 'ahmet@example.com')
```

users

id	name	email
1	Ahmet Yılmaz	ahmet@example.com

No `id` was given; SQLite assigned **1**. 1 row affected.

SQL Commands, Step by Step

```
INSERT INTO users(name, email)
  VALUES('Ayşe Demir', 'ayse@example.com')
```

users

id	name	email
1	Ahmet Yılmaz	ahmet@example.com
2	Ayşe Demir	ayse@example.com

The second record got `id = 2` (the largest id + 1).

SQL Commands, Step by Step

```
UPDATE users SET name = 'Mehmet Akdeniz'
  WHERE id = 1
```

users

id	name	email
1	Mehmet Akdeniz	ahmet@example.com
2	Ayşe Demir	ayse@example.com

`WHERE id = 1` picked only the first row; only `name` changed, `email` stayed the same.

SQL Commands, Step by Step

```
DELETE FROM users WHERE id = 1
```

users

id	name	email
~~1~~	~~Mehmet Akdeniz~~	~~ahmet@example.com~~
2	Ayşe Demir	ayse@example.com

The row with `id = 1` is deleted. 1 row affected.

SQL Commands, Step by Step

```
SELECT * FROM users
```

users

id	name	email
2	Ayşe Demir	ayse@example.com

A query **doesn't change** the table, it only reads: the result is one row, `(2, Ayşe Demir, ayse@example.com)`.

SQL BASICS

SELECT: Filtering, Sorting, Limiting

`scores`

player	score
ada	72
can	95
ece	58
efe	95
naz	81

Query	Result
`SELECT player, score FROM scores WHERE score < 90 ORDER BY score DESC`	naz 81, ada 72, ece 58
`SELECT player, score FROM scores ORDER BY score DESC, player LIMIT 3`	can 95, efe 95, naz 81
`SELECT player, score FROM scores WHERE player LIKE 'e%'`	ece 58, efe 95
`SELECT COUNT(*), AVG(score), MAX(score) FROM scores WHERE score >= 60`	4 · 85.75 · 95

- `WHERE` filters rows (`=` , `<>` , `<` , `>=` , `AND` , `OR` , `LIKE 'e%'` : “starts with e”).
- `ORDER BY … DESC` sorts from largest to smallest; ties use the second key (`player`). `LIMIT 3` takes the first 3 rows.

How Does a `SELECT` Query Run?

```
String sql = "SELECT player, score FROM scores "
           + "WHERE score > 70 "
           + "ORDER BY player DESC LIMIT 2";
try (Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("player") + " "
                       + rs.getInt("score"));
  }
}
```

FROM | WHERE | ORDER BY | LIMIT

player	score
ada	72
can	95
ece	58
efe	95
naz	81

The query is a single string. The `scores` table has 5 rows.

Step 1 / 10

VARIABLES

`sql` = **SELECT … LIMIT 2**

OUTPUT

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
5       ResultSet rs = st.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("player") + " "
8                         + rs.getInt("score"));
9    }
10 }
```

FROM | WHERE | ORDER BY | LIMIT

player	score
ada	72
can	95
ece	58
efe	95
naz	81

5 rows

`executeQuery` sends the query. The database starts with `FROM` : every row of the table.

Step 2 / 10

VARIABLES

`stage = FROM`

OUTPUT

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
▶       ResultSet rs = st.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("player") + " "
8                         + rs.getInt("score"));
9    }
10 }
```

FROM | **WHERE** | ORDER BY | LIMIT

player	score
ada	72
can	95
ece	58
efe	95
naz	81

ece 58 dropped

`WHERE score > 70` : `ece` fails the condition and is dropped; 4 rows remain.

Step 3 / 10

VARIABLES

`stage = WHERE`

OUTPUT

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
▶       ResultSet rs = st.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("player") + " "
8                         + rs.getInt("score"));
9    }
10 }
```

FROM | WHERE | **ORDER BY** | LIMIT

player	score
naz	81
efe	95
can	95
ada	72

name, Z → A

`ORDER BY player DESC` : names are sorted in reverse alphabetical order (not by score).

Step 4 / 10

VARIABLES

`stage = ORDER BY`

OUTPUT

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
▶       ResultSet rs = st.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("player") + " "
8                         + rs.getInt("score"));
9    }
10 }
```

FROM | WHERE | ORDER BY | **LIMIT**

player	score
naz	81
efe	95
can	95
ada	72

first 2 rows

`LIMIT 2` : the first two rows of the sorted list form the result set.

Step 5 / 10

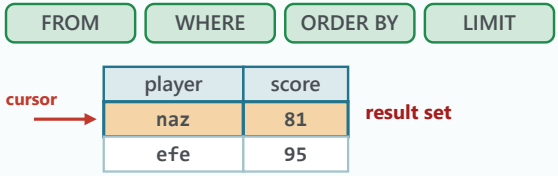
VARIABLES

`stage = LIMIT`

OUTPUT

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
5       ResultSet rs = st.executeQuery(sql)) {
▶    while (rs.next()) {
7      System.out.println(rs.getString("player") + " "
8                         + rs.getInt("score"));
9    }
10 }
```



The cursor starts **before** the first row; `next()` moves it to row 1.

Step 6 / 10

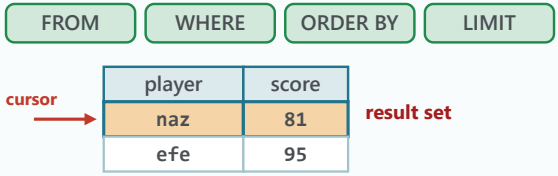
VARIABLES

`rs.next() = true`

OUTPUT

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
5       ResultSet rs = st.executeQuery(sql)) {
6    while (rs.next()) {
▶      System.out.println(rs.getString("player") + " "
▶                         + rs.getInt("score"));
9    }
10 }
```



The columns are read by name and printed.

Step 7 / 10

VARIABLES

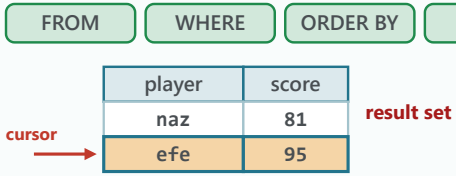
`player = naz` `score = 81`

OUTPUT

```
naz 81
```

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
5       ResultSet rs = st.executeQuery(sql)) {
▶    while (rs.next()) {
7      System.out.println(rs.getString("player") + " "
8                         + rs.getInt("score"));
9    }
10 }
```



The cursor moves to row 2.

Step 8 / 10

VARIABLES

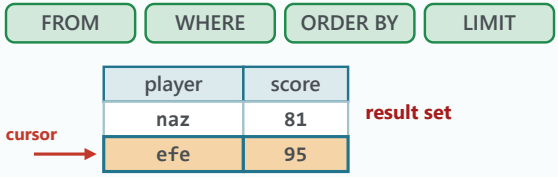
`rs.next() = true`

OUTPUT

```
naz 81
```

How Does a `SELECT` Query Run?

```
1  String sql = "SELECT player, score FROM scores "
2             + "WHERE score > 70 "
3             + "ORDER BY player DESC LIMIT 2";
4  try (Statement st = conn.createStatement();
5       ResultSet rs = st.executeQuery(sql)) {
6    while (rs.next()) {
▶      System.out.println(rs.getString("player") + " "
▶                         + rs.getInt("score"));
9    }
10 }
```



`efe 95` is printed.

Step 9 / 10

VARIABLES

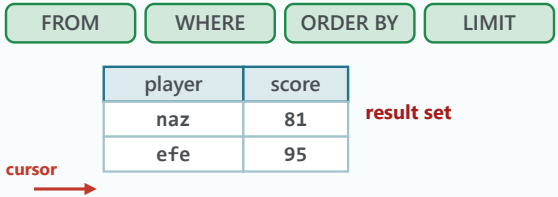
`player = efe` `score = 95`

OUTPUT

```
naz 81
efe 95
```

How Does a `SELECT` Query Run?

```
1 String sql = "SELECT player, score FROM scores "
2              + "WHERE score > 70 "
3              + "ORDER BY player DESC LIMIT 2";
4 try (Statement st = conn.createStatement();
5      ResultSet rs = st.executeQuery(sql)) {
▶   while (rs.next()) {
7     System.out.println(rs.getString("player") + " "
8                        + rs.getInt("score"));
9   }
▶ }
```



No more rows: `next()` returns `false` and the loop ends. When the `try` ends, `rs` and `st` are closed.

Step 10 / 10

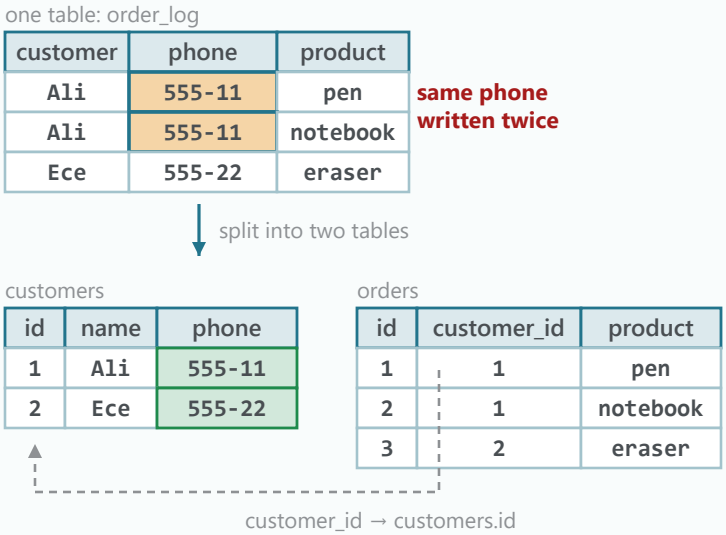
VARIABLES

`rs.next() = false`

OUTPUT

```
naz 81
efe 95
```

Avoiding Repetition: Normalization



- In one table, the same fact (Ali's phone) is **repeated** in every order: wasted space and a risk of inconsistency.
- **Update anomaly:** when the phone changes, every row must be fixed; miss one and Ali has two different phones.
- **Insert anomaly:** a customer with no orders yet can't be recorded.
- **Delete anomaly:** deleting Ece's only order loses Ece's details too.
- **Normalization**: split the tables so each fact is kept **in one place**. The tables are linked by a key (`customer_id`).
- First normal form: **one value** per cell. A list like `'555-11, 555-33'` moves to separate rows or a separate table.

Foreign Keys and Relationships

```
CREATE TABLE depts (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL);
CREATE TABLE students (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  dept_id INTEGER REFERENCES depts(id));
```

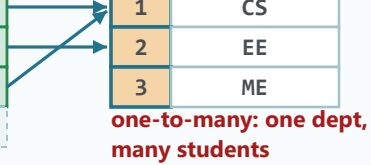
students

id	name	dept_id
1	Ada	1
2	Bora	2
3	Cem	1
4	Deniz	NULL

Deniz has no dept: dept_id NULL

depts

id	name
1	CS
2	EE
3	ME



one-to-many: one dept, many students

- A **foreign key** is a column that points to another table's primary key. A dept can have many students: a **one-to-many** relationship.
- Adding a student to a missing dept (`dept_id = 9`) or deleting a dept that has students is rejected: `FOREIGN KEY constraint failed` .
- SQLite **doesn't check this by default**: every connection needs `stmt.execute("PRAGMA foreign_keys = ON")` . With it off, `dept_id = 9` is silently accepted.
- A **many-to-many** relationship (student–course) needs a third, linking table: `enrollments(student_id, course_id)` .
- Java analogy: a `Dept` reference inside a `Student` object.

Combining Tables with `JOIN`

```sql
SELECT s.name AS student, d.name AS dept
FROM students s JOIN depts d
  ON s.dept_id = d.id
ORDER BY s.id
```

student	`JOIN` (INNER)	`LEFT JOIN`
Ada	CS	CS
Bora	EE	EE
Cem	CS	CS
Deniz	*no row*	`NULL`

- A **JOIN** matches the rows of two tables by the `ON` condition and produces one result table.
- `s` and `d` are table **aliases**. Both tables have a `name` column, so we write `s.name` and `d.name`; a bare `name` gives `ambiguous column name: name`.
- `AS` renames a result column: Java reads it with `rs.getString("dept")`.
- **INNER JOIN** (plain `JOIN`): only rows that have a match. Deniz, whose dept is `NULL`, is missing from the result.
- **LEFT JOIN**: every row of the left table; with no match the right side is `NULL`, `null` in Java.
- Forget `ON` (`FROM students, depts`) and every row pairs with every row: a **12-row** Cartesian product, 4 × 3.

A `JOIN` Query, Step by Step

```
String sql = "SELECT s.name AS student, d.name AS dept "
    + "FROM students s JOIN depts d "
    + "ON s.dept_id = d.id ORDER BY s.id";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("student") + " - "
        + rs.getString("dept"));
  }
}
```

students s

name	dept_id
Ada	1
Bora	2
Cem	1
Deniz	NULL

result: no rows yet

depts d

id	name
1	CS
2	EE
3	ME

The query refers to the two tables by aliases: `s` = students, `d` = depts.

Step 1 / 9

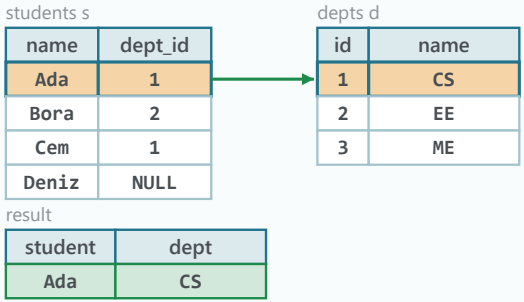
VARIABLES

—

OUTPUT

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
5   while (rs.next()) {
6     System.out.println(rs.getString("student") + " - "
7         + rs.getString("dept"));
8   }
9 }
```



1 = 1 → match

`executeQuery` : Ada's `dept_id` is 1; `id = 1` is found in depts → (Ada, CS).

Step 2 / 9

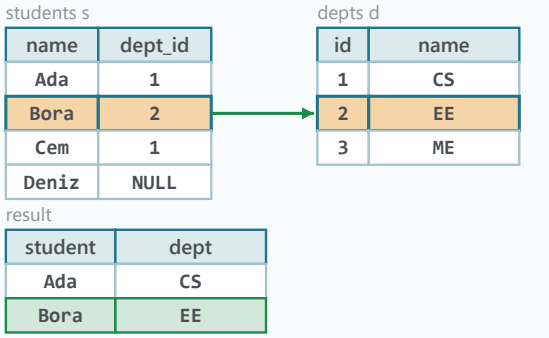
VARIABLES

—

OUTPUT

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
5   while (rs.next()) {
6     System.out.println(rs.getString("student") + " - "
7         + rs.getString("dept"));
8   }
9 }
```



Bora → 2 → EE.

Step 3 / 9

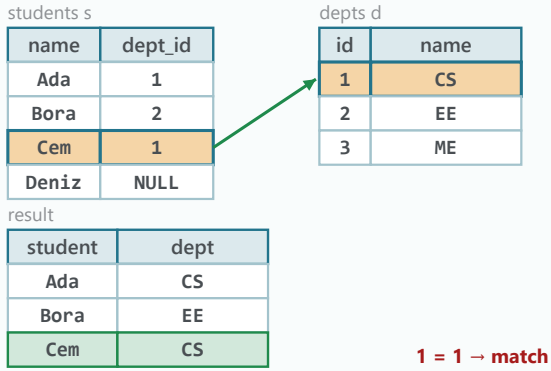
VARIABLES

—

OUTPUT

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
5   while (rs.next()) {
6     System.out.println(rs.getString("student") + " - "
7         + rs.getString("dept"));
8   }
9 }
```



Cem → 1: the same dept row is used a second time.

Step 4 / 9

VARIABLES

—

OUTPUT

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
5   while (rs.next()) {
6     System.out.println(rs.getString("student") + " - "
7         + rs.getString("dept"));
8   }
9 }
```

students s

name	dept_id
Ada	1
Bora	2
Cem	1
Deniz	NULL

depts d

id	name
1	CS
2	EE
3	ME

result

student	dept
Ada	CS
Bora	EE
Cem	CS

NULL = ? → no match

Deniz's `dept_id` is `NULL`: it equals no `id`, so the row **doesn't** make it into the result.

Step 5 / 9

VARIABLES

—

OUTPUT

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
4 try (ResultSet rs = stmt.executeQuery(sql)) {
▶   while (rs.next()) {
▶     System.out.println(rs.getString("student") + " - "
▶         + rs.getString("dept"));
8   }
9 }
```

students s

name	dept_id
Ada	1
Bora	2
Cem	1
Deniz	NULL

depts d

id	name
1	CS
2	EE
3	ME

result

student	dept
Ada	CS
Bora	EE
Cem	CS

cursor at row 1

`next()` moves to the first row; the columns are read by the names given with `AS`.

Step 6 / 9

VARIABLES

`row = 1`

OUTPUT

```
Ada - CS
```

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
4 try (ResultSet rs = stmt.executeQuery(sql)) {
▶   while (rs.next()) {
▶     System.out.println(rs.getString("student") + " - "
▶         + rs.getString("dept"));
8   }
9 }
```

students s

name	dept_id
Ada	1
Bora	2
Cem	1
Deniz	NULL

depts d

id	name
1	CS
2	EE
3	ME

result

student	dept
Ada	CS
Bora	EE
Cem	CS

cursor at row 2

The second row is printed.

Step 7 / 9

VARIABLES

`row = 2`

OUTPUT

```
Ada - CS
Bora - EE
```

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
4 try (ResultSet rs = stmt.executeQuery(sql)) {
▶   while (rs.next()) {
▶     System.out.println(rs.getString("student") + " - "
▶         + rs.getString("dept"));
8   }
9 }
```

students s

name	dept_id
Ada	1
Bora	2
Cem	1
Deniz	NULL

depts d

id	name
1	CS
2	EE
3	ME

result

student	dept
Ada	CS
Bora	EE
Cem	CS

cursor at row 3

The third row is printed.

Step 8 / 9

VARIABLES

`row = 3`

OUTPUT

```
Ada - CS
Bora - EE
Cem - CS
```

A `JOIN` Query, Step by Step

```
1 String sql = "SELECT s.name AS student, d.name AS dept "
2     + "FROM students s JOIN depts d "
3     + "ON s.dept_id = d.id ORDER BY s.id";
4 try (ResultSet rs = stmt.executeQuery(sql)) {
▶   while (rs.next()) {
6     System.out.println(rs.getString("student") + " - "
7         + rs.getString("dept"));
8   }
▶ }
```

students s

name	dept_id
Ada	1
Bora	2
Cem	1
Deniz	NULL

depts d

id	name
1	CS
2	EE
3	ME

result

student	dept
Ada	CS
Bora	EE
Cem	CS

next() → false

No more rows; the loop ends. With `LEFT JOIN`, a fourth row `Deniz – null` would appear.

Step 9 / 9

VARIABLES

`row = none`

OUTPUT

```
Ada - CS
Bora - EE
Cem - CS
```

Grouping: `GROUP BY` and `HAVING`

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

GROUP BY customer

customer	num	total
Ali	3	200
Can	1	200
Ece	2	120

one row per group

- `GROUP BY` collects rows with the same value into a group; **aggregate functions** (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) are computed per group.
- `WHERE` filters **rows before** grouping, `HAVING` filters **groups after** it. `WHERE SUM(amount) >= 150` is an error: `misuse of aggregate: SUM()`.
- Logical order of evaluation: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT`.

Query (`SELECT customer, COUNT(*), SUM(amount) FROM orders …`)	**Result**
`GROUP BY customer`	Ali (3, 200) · Can (1, 200) · Ece (2, 120)
`WHERE amount >= 50 GROUP BY customer`	Ali (2, 170) · Can (1, 200) · Ece (1, 80)
`GROUP BY customer HAVING SUM(amount) >= 150`	Ali (3, 200) · Can (1, 200)

With a JOIN: `SELECT d.name, COUNT(s.id) FROM depts d LEFT JOIN students s ON s.dept_id = d.id GROUP BY d.id` → CS 2 · EE 1 · ME 0. `COUNT(s.id)` skips `NULL`s; with `COUNT(*)`, ME, which has no students, would get 1.

A `GROUP BY` Query, Step by Step

```
String sql = "SELECT customer, COUNT(*) AS num, "
    + "SUM(amount) AS total FROM orders "
    + "GROUP BY customer HAVING SUM(amount) >= 150 "
    + "ORDER BY total DESC, customer";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("customer") + ": "
        + rs.getInt("num") + " orders, "
        + rs.getInt("total") + " TL");
  }
}
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

The `orders` table has 6 rows; the query is a single string.

Step 1 / 8

VARIABLES

—

OUTPUT

A `GROUP BY` Query, Step by Step

```
1  String sql = "SELECT customer, COUNT(*) AS num, "
2      + "SUM(amount) AS total FROM orders "
3      + "GROUP BY customer HAVING SUM(amount) >= 150 "
4      + "ORDER BY total DESC, customer";
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("customer") + ": "
8          + rs.getInt("num") + " orders, "
9          + rs.getInt("total") + " TL");
10   }
11 }
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

3 groups: Ali, Can, Ece

`FROM` and `GROUP BY customer`: the rows are split into **three groups** by customer name.

Step 2 / 8

VARIABLES

—

OUTPUT

A `GROUP BY` Query, Step by Step

```
1  String sql = "SELECT customer, COUNT(*) AS num, "
2      + "SUM(amount) AS total FROM orders "
3      + "GROUP BY customer HAVING SUM(amount) >= 150 "
4      + "ORDER BY total DESC, customer";
▶  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("customer") + ": "
8          + rs.getInt("num") + " orders, "
9          + rs.getInt("total") + " TL");
10   }
11 }
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

groups

customer	num	total
Ali	3	200
Can	1	200
Ece	2	120

For each group, `COUNT(*)` and `SUM(amount)`: Ali 120 + 50 + 30 = 200, Can 200, Ece 80 + 40 = 120.

Step 3 / 8

VARIABLES

—

OUTPUT

A `GROUP BY` Query, Step by Step

```
1  String sql = "SELECT customer, COUNT(*) AS num, "
2      + "SUM(amount) AS total FROM orders "
3      + "GROUP BY customer HAVING SUM(amount) >= 150 "
4      + "ORDER BY total DESC, customer";
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("customer") + ": "
8          + rs.getInt("num") + " orders, "
9          + rs.getInt("total") + " TL");
10   }
11 }
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

groups

customer	num	total
Ali	3	200
Can	1	200
~~Ece~~	~~2~~	~~120~~

HAVING: 120 < 150

`HAVING SUM(amount) >= 150` : Ece's **group** is dropped (120).

Step 4 / 8

VARIABLES

—

OUTPUT

[redacted]

A `GROUP BY` Query, Step by Step

```
1  String sql = "SELECT customer, COUNT(*) AS num, "
2      + "SUM(amount) AS total FROM orders "
3      + "GROUP BY customer HAVING SUM(amount) >= 150 "
4      + "ORDER BY total DESC, customer";
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      System.out.println(rs.getString("customer") + ": "
8          + rs.getInt("num") + " orders, "
9          + rs.getInt("total") + " TL");
10   }
11 }
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

tie → name: Ali, Can

groups

customer	num	total
Ali	3	200
Can	1	200
~~Ece~~	~~2~~	~~120~~

HAVING: 120 < 150

result (sorted)

customer	num	total
Ali	3	200
Can	1	200

`ORDER BY total DESC, customer`: Ali and Can tie at 200; the second key is the name, so Ali comes first.

Step 5 / 8

VARIABLES

—

OUTPUT

A `GROUP BY` Query, Step by Step

```
1 String sql = "SELECT customer, COUNT(*) AS num, "
2     + "SUM(amount) AS total FROM orders "
3     + "GROUP BY customer HAVING SUM(amount) >= 150 "
4     + "ORDER BY total DESC, customer";
5 try (ResultSet rs = stmt.executeQuery(sql)) {
▶   while (rs.next()) {
▶     System.out.println(rs.getString("customer") + ": "
▶         + rs.getInt("num") + " orders, "
▶         + rs.getInt("total") + " TL");
10  }
11 }
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

groups

customer	num	total
Ali	3	200
Can	1	200
~~Ece~~	~~2~~	~~120~~

HAVING: 120 < 150

result (sorted)

customer	num	total
Ali	3	200
Can	1	200

`next()` moves to the first row; the aliases (`num` , `total`) are read as column names.

Step 6 / 8

VARIABLES

`row = 1`

OUTPUT

```
Ali: 3 orders, 200 TL
```

A `GROUP BY` Query, Step by Step

```
1  String sql = "SELECT customer, COUNT(*) AS num, "
2      + "SUM(amount) AS total FROM orders "
3      + "GROUP BY customer HAVING SUM(amount) >= 150 "
4      + "ORDER BY total DESC, customer";
5  try (ResultSet rs = stmt.executeQuery(sql)) {
▶    while (rs.next()) {
▶      System.out.println(rs.getString("customer") + ": "
▶          + rs.getInt("num") + " orders, "
▶          + rs.getInt("total") + " TL");
10   }
11 }
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

groups

customer	num	total
Ali	3	200
Can	1	200
~~Ece~~	~~2~~	~~120~~

HAVING: 120 < 150

result (sorted)

customer	num	total
Ali	3	200
Can	1	200

The second row is printed.

Step 7 / 8

VARIABLES

`row = 2`

OUTPUT

```
Ali: 3 orders, 200 TL
Can: 1 orders, 200 TL
```

A `GROUP BY` Query, Step by Step

```
1  String sql = "SELECT customer, COUNT(*) AS num, "
2      + "SUM(amount) AS total FROM orders "
3      + "GROUP BY customer HAVING SUM(amount) >= 150 "
4      + "ORDER BY total DESC, customer";
5  try (ResultSet rs = stmt.executeQuery(sql)) {
▶    while (rs.next()) {
7      System.out.println(rs.getString("customer") + ": "
8          + rs.getInt("num") + " orders, "
9          + rs.getInt("total") + " TL");
10   }
▶  }
```

orders

id	customer	amount
1	Ali	120
2	Ece	80
3	Ali	50
4	Can	200
5	Ece	40
6	Ali	30

next() → false

groups

customer	num	total
Ali	3	200
Can	1	200
~~Ece~~	~~2~~	~~120~~

HAVING: 120 < 150

result (sorted)

customer	num	total
Ali	3	200
Can	1	200

The loop ends. The result has **one row per group**; individual orders don't appear.

Step 8 / 8

VARIABLES

`row = none`

OUTPUT

```
Ali: 3 orders, 200 TL
Can: 1 orders, 200 TL
```

`NULL`: The Absence of a Value

- `NULL` means “unknown / missing”; it is not `0` or empty text (`''`).
- Any comparison with `NULL` gives an unknown result: `WHERE dept_id = NULL` selects no rows. Use `IS NULL` or `IS NOT NULL` .
- `WHERE dept_id <> 1` doesn't select Deniz either: we don't even know that `NULL` differs from 1.
- `NULL + 1` is `NULL` . `COALESCE(dept_id, 0)` puts 0 in place of `NULL` .
- `COUNT(*)` counts rows, `COUNT(dept_id)` non- `NULL` values: 4 and 3. With no matching rows, `AVG` and `SUM` return `NULL` .

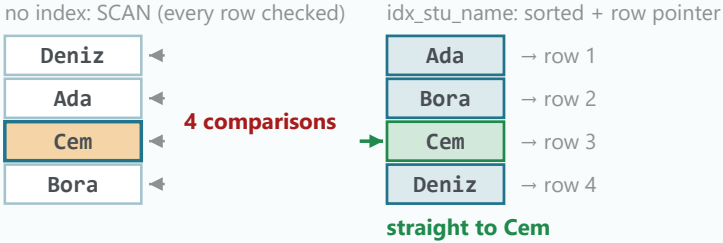
```
String sql = "SELECT name, dept_id FROM students "
    + "WHERE id >= 3";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    int d = rs.getInt("dept_id");
    boolean none = rs.wasNull();
    System.out.println(rs.getString("name") + " " + d
        + (none ? " (NULL)" : ""));
  }
}
```

OUTPUT

```
Cem 1
Deniz 0 (NULL)
```

- `getInt` returns **0** for `NULL` ; to tell 0 from `NULL` , call `wasNull()` right after.
- `wasNull()` looks at the **last column read**: called after `getString("name")` , it would ask about the name and `(NULL)` would never print.
- `getString` and `getObject` return `null` for `NULL` .

Indexes: Faster Searches



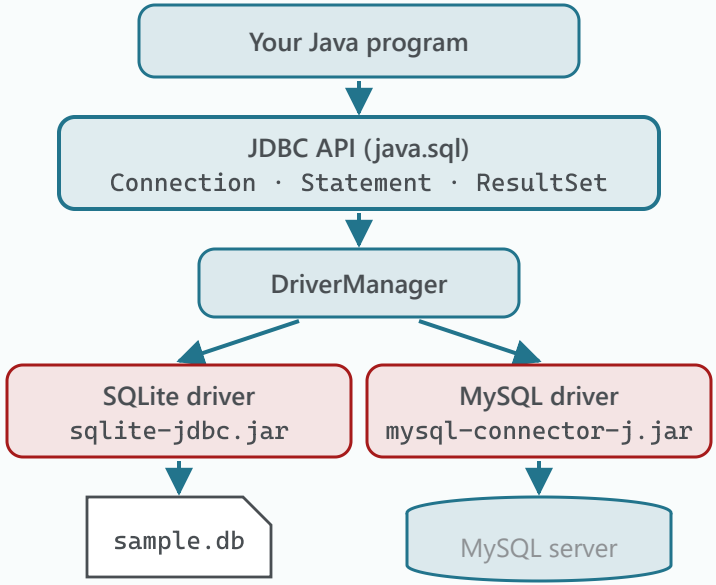

```
CREATE INDEX idx_stu_name ON students(name);
EXPLAIN QUERY PLAN
  SELECT * FROM students WHERE name = 'Cem';
```

- An **index** is an extra structure (a B-tree) that keeps a column's values **sorted**, like the index at the back of a book. A search goes to the right row instead of scanning the whole table.
- Primary keys and `UNIQUE` columns get an index automatically.
- The cost: every `INSERT`, `UPDATE`, `DELETE` updates the index too, and the index takes disk space. Put indexes on columns searched often and used in `WHERE`, `JOIN` and `ORDER BY`.
- `EXPLAIN QUERY PLAN` (SQLite) shows how a query will be executed.

Case	`EXPLAIN QUERY PLAN` output
No index	`SCAN students`
With the index	`SEARCH students USING INDEX idx_stu_name (name=?)`
`WHERE id = 3`	`SEARCH … USING INTEGER PRIMARY KEY (rowid=?)`
`WHERE name LIKE '%em'`	`SCAN students` (a leading wildcard can't use the index)

JDBC (Java Database Connectivity)

- **JDBC** (Java Database Connectivity) is Java's **standard API** for connecting to databases; it lives in `java.sql` and ships with the JDK.
- The program uses only the JDBC **interfaces**; each database vendor supplies a **driver** that implements them.
- Switching databases usually changes only the **driver jar** and the **connection URL**.
- Polymorphism at work: `Connection` is an interface; in SQLite the real object's class is `org.sqlite.jdbc4.JDBC4Connection`.



The Core JDBC Types

Type	Purpose	Key methods
`DriverManager`	Finds the right driver and opens a connection	`getConnection(url)`, `getConnection(url, user, password)`
`Connection`	An open session with the database	`createStatement()`, `prepareStatement(sql)`, `commit()`, `rollback()`, `close()`
`Statement`	Runs a fixed SQL string	`execute`, `executeUpdate`, `executeQuery`
`PreparedStatement`	Precompiled SQL with `?` placeholders	`setInt`, `setString`, `executeUpdate()`, `executeQuery()`
`ResultSet`	The rows of a query result, plus a cursor	`next()`, `getInt("id")`, `getString("name")`
`SQLException`	A database error; a **checked** exception	`getMessage()`, `getErrorCode()`

All of them are in `java.sql`: put `import java.sql.*;` at the top of the file. `Connection`, `Statement` and `ResultSet` are resources that must be closed (`AutoCloseable`).

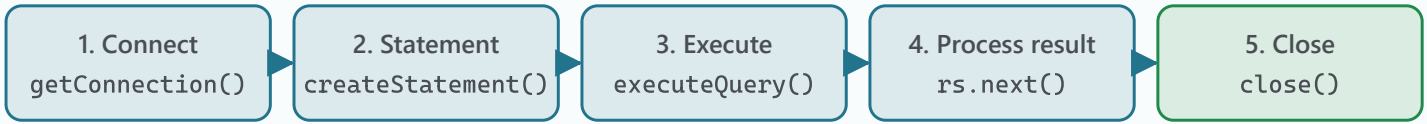
The Connection URL (JDBC URL)

Connection URL	Meaning
`jdbc:sqlite:sample.db`	`sample.db` in the working directory; **created empty** if missing
`jdbc:sqlite:C:/data/school.db`	A file by its full path
`jdbc:sqlite::memory:`	A temporary in-memory database, gone when the connection closes (for experiments)
`jdbc:mysql://localhost:3306/testdb`	A MySQL server: host, port, database name; plus a user name and password



- Format: `jdbc:<subprotocol>:<database file or address>`. `DriverManager` picks the driver from the subprotocol.
- Since JDBC 4.0 (Java 6), the driver is loaded **automatically** from the jar on the classpath. The `Class.forName("org.sqlite.JDBC")` line in old code is no longer needed.
- If the driver isn't on the classpath: `No suitable driver found for jdbc:sqlite:sample.db`.

The Steps of a JDBC Program




```
try (Connection conn = DriverManager.getConnection(url); // 1
     Statement stmt = conn.createStatement();             // 2
     ResultSet rs = stmt.executeQuery(sql)) {             // 3
  while (rs.next()) {                                     // 4
    System.out.println(rs.getInt(1));
  }
} // 5: rs, stmt, conn auto-closed in reverse
```

- Resources are opened in the **try-with-resources** parentheses; when the block ends (even on an error) they close **in reverse order**: first `rs`, then `stmt`, finally `conn`.
- Unclosed connections can leave the file locked and leak memory.
- `getInt(1)`: column numbers start at **1**.

BASIC OPERATIONS

Connecting to SQLite

```
// Path to the database file (connection URL)
String url = "jdbc:sqlite:sample.db";
try (Connection conn = DriverManager.getConnection(url)) {
  // Print a message if connected.
  if (conn != null) {
    System.out.println("Connected!");
  }
} catch (SQLException e) {
  // Print the error message on failure.
  System.out.println(e.getMessage());
}
```

OUTPUT

```
Connected!
```

- `getConnection` **creates** the file if it doesn't exist (an empty database).
- If it can't connect, it doesn't return `null`; it throws `SQLException`. `if (conn != null)` is just an extra safeguard.
- `SQLException` is a checked exception: catch it or declare it with `throws`.

Creating a Table

```
String url = "jdbc:sqlite:sample.db";
String sql = "CREATE TABLE IF NOT EXISTS users "
           + "(id INTEGER PRIMARY KEY, name TEXT, email TEXT)";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Create the table.
  stmt.execute(sql);
  System.out.println("Table created!");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

OUTPUT

```
Table created!
```

- `execute(sql)` runs any SQL; it returns `true` if it produced a result set, `false` for `CREATE`.
- Thanks to `IF NOT EXISTS`, running the program again is no error. Without it: `table users already exists`.
- Several resources go in one `try (…; …)`, separated by `;`.

Inserting Data

```
String url = "jdbc:sqlite:sample.db";
String sql = "INSERT INTO users(name, email) "
           + "VALUES('Ahmet Yılmaz', 'ahmet@example.com')";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Insert data into the database.
  int n = stmt.executeUpdate(sql);
  System.out.println("Data inserted! (" + n + " row)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

OUTPUT

```
Data inserted! (1 row)
```

- `executeUpdate` returns the **number of rows affected** (an `int`).
- No `id` was given; SQLite assigned `1`.
- Each run of the program adds **a new row** (id 2, 3, …).
- Text values in SQL go in single quotes.

Reading Data

```
String url = "jdbc:sqlite:sample.db";
String sql = "SELECT * FROM users";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  // Read and print the rows one by one.
  while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id")
        + ", Name: " + rs.getString("name")
        + ", Email: " + rs.getString("email"));
  }
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

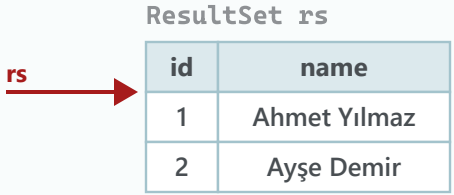
OUTPUT

```
ID: 1, Name: Ahmet Yılmaz, Email: ahmet@example.com
```

- `executeQuery` returns a `ResultSet`.
- `rs.next()` moves the cursor to the next row; it returns `true` if there is one, `false` otherwise.
- `getInt("id")`, `getString("name")`: read the current row's columns by name.

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
▶  try (Connection conn = DriverManager.getConnection(url);
▶       Statement stmt = conn.createStatement();
▶       ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```



cursor: before the first row

The connection opens and the query runs. `rs` holds the two-row result; the cursor waits **before the first row**.

Step 1 / 10

OUTPUT

VARIABLES

—

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
▶    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

rs →

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

cursor: row 1

`rs.next()` moves the cursor to row 1 and returns `true` : we enter the loop.

Step 2 / 10

OUTPUT

VARIABLES

`rs.next()` = **true**

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
▶      int id = rs.getInt("id");
▶      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

rs →

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

cursor: row 1

The columns are read **from the current row**.

Step 3 / 10

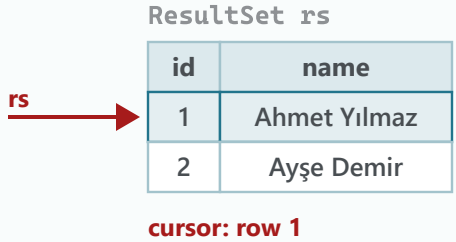
OUTPUT

VARIABLES

`rs.next() = true` `id = 1` `name = "Ahmet Yılmaz"`

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
▶      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```



The row is printed.

Step 4 / 10

OUTPUT

```
1 - Ahmet Yılmaz
```

VARIABLES

`rs.next()` = **`true`** `id` = **`1`** `name` = **`"Ahmet Yılmaz"`**

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
▶    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

rs →

cursor: row 2

The cursor moves to row 2; `true` again.

Step 5 / 10

OUTPUT

```
1 - Ahmet Yılmaz
```

VARIABLES

`rs.next() = true` `id = 1` `name = "Ahmet Yılmaz"`

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
▶      int id = rs.getInt("id");
▶      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

rs →

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

cursor: row 2

This time row 2's values are read.

Step 6 / 10

OUTPUT

```
1 - Ahmet Yılmaz
```

VARIABLES

`rs.next() = true` `id = 2` `name = "Ayşe Demir"`

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
▶      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

rs →

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

cursor: row 2

The second row is printed.

Step 7 / 10

OUTPUT

```
1 - Ahmet Yılmaz
2 - Ayşe Demir
```

VARIABLES

`rs.next()` = **true** | `id` = **2** | `name` = **"Ayşe Demir"**

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
▶    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

rs →

cursor: after the last row

No more rows: `rs.next()` returns `false` and the loop ends.

Step 8 / 10

OUTPUT

```
1 - Ahmet Yılmaz
2 - Ayşe Demir
```

VARIABLES

`rs.next() = false`

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
▶    System.out.println("Done reading.");
12 } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

rs →

cursor: after the last row

The line after the loop runs.

Step 9 / 10

OUTPUT

```
1 - Ahmet Yılmaz
2 - Ayşe Demir
Done reading.
```

VARIABLES

`rs.next()` = **false**

Reading with a ResultSet, Step by Step

```
1  String url = "jdbc:sqlite:reading.db";
2  String sql = "SELECT * FROM users";
3  try (Connection conn = DriverManager.getConnection(url);
4       Statement stmt = conn.createStatement();
5       ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) {
7      int id = rs.getInt("id");
8      String name = rs.getString("name");
9      System.out.println(id + " - " + name);
10   }
11   System.out.println("Done reading.");
▶ } catch (SQLException e) {
13   System.out.println(e.getMessage());
14 }
```

ResultSet rs

id	name
1	Ahmet Yılmaz
2	Ayşe Demir

rs →

cursor: after the last row

The try block ends: `rs`, `stmt`, `conn` are **closed automatically**, in that order.

Step 10 / 10

OUTPUT

```
1 - Ahmet Yılmaz
2 - Ayşe Demir
Done reading.
```

VARIABLES

—

Updating Data

```
String url = "jdbc:sqlite:sample.db";
String sql = "UPDATE users SET name = 'Mehmet Akdeniz' WHERE id = 1";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Update data in the database.
  int n = stmt.executeUpdate(sql);
  System.out.println("Data updated! (" + n + " row)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

OUTPUT

```
Data updated! (1 row)
```

- Read before: `ID: 1, Name: Ahmet Yılmaz, …` ; after: `ID: 1, Name: Mehmet Akdeniz, …`
- `SET` changes only the listed column; `email` stays the same.
- `WHERE id = 1` picks the row to update. **Forget WHERE and every row** is updated.

Deleting Data

```java
String url = "jdbc:sqlite:sample.db";
String sql = "DELETE FROM users WHERE id = 1";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Delete data from the database.
  int n = stmt.executeUpdate(sql);
  System.out.println("Data deleted! (" + n + " row)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

OUTPUT (RUNNING THE PROGRAM TWICE)

```
Data deleted! (1 row)
Data deleted! (0 row)
```

- On the second run there is no row left to delete: no error, `executeUpdate` just returns `0`.
- So it is better to word the message by the returned count (Exercise 1).
- `DELETE FROM users` without WHERE deletes **every** row; the table itself remains.

BASIC OPERATIONS

execute, executeUpdate, executeQuery

Method	Returns	Use
`execute`	`boolean`	`CREATE TABLE`, `DROP`
`executeUpdate`	`int` row count	`INSERT`, `UPDATE`, `DELETE`
`executeQuery`	`ResultSet`	`SELECT`

The wrong method fails: running an `INSERT` with `executeQuery` → `query does not return ResultSet`.

RUNNING THE PROGRAMS IN ORDER

```
> java Connect
Connected!
> java CreateTable
Table created!
> java InsertData
Data inserted! (1 row)
> java ReadData
ID: 1, Name: Ahmet Yılmaz, Email:
ahmet@example.com
> java UpdateData
Data updated! (1 row)
> java ReadData
ID: 1, Name: Mehmet Akdeniz, Email:
ahmet@example.com
> java DeleteData
Data deleted! (1 row)
> java ReadData
> java DeleteData
Data deleted! (0 row)
```

What Does `executeUpdate` Return?

```
String[] names = {"stapler", "scissors", "tape"};
int[] stock = {40, 0, 15};
PreparedStatement ins = conn.prepareStatement(
    "INSERT INTO supplies(name, stock) VALUES(?, ?)");
for (int i = 0; i < names.length; i++) {
  ins.setString(1, names[i]);
  ins.setInt(2, stock[i]);
  ins.executeUpdate();
}
int u = stmt.executeUpdate(
    "UPDATE supplies SET stock = stock + 10 WHERE stock < 20");
int d = stmt.executeUpdate("DELETE FROM supplies WHERE stock > 100");
System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock

table empty

The insert statement is prepared **once**; in the loop only the values change.

Step 1 / 8

VARIABLES

`names = [stapler, scissors, tape]`

`stock = [40, 0, 15]`

OUTPUT

What Does `executeUpdate` Return?

```
1  String[] names = {"stapler", "scissors", "tape"};
2  int[] stock = {40, 0, 15};
3  PreparedStatement ins = conn.prepareStatement(
4      "INSERT INTO supplies(name, stock) VALUES(?, ?)");
5  for (int i = 0; i < names.length; i++) {
▶    ins.setString(1, names[i]);
▶    ins.setInt(2, stock[i]);
8    ins.executeUpdate();
9  }
10 int u = stmt.executeUpdate(
11     "UPDATE supplies SET stock = stock + 10 WHERE stock <
   20");
12 int d = stmt.executeUpdate("DELETE FROM supplies WHERE
   stock > 100");
13 System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock

placeholders

1st ? = "stapler"

2nd ? = 40

On the first pass the placeholders are filled.

Step 2 / 8

VARIABLES

`i = 0`

OUTPUT

What Does `executeUpdate` Return?

```
1  String[] names = {"stapler", "scissors", "tape"};
2  int[] stock = {40, 0, 15};
3  PreparedStatement ins = conn.prepareStatement(
4      "INSERT INTO supplies(name, stock) VALUES(?, ?)");
5  for (int i = 0; i < names.length; i++) {
6    ins.setString(1, names[i]);
7    ins.setInt(2, stock[i]);
▶    ins.executeUpdate();
9  }
10 int u = stmt.executeUpdate(
11     "UPDATE supplies SET stock = stock + 10 WHERE stock <
   20");
12 int d = stmt.executeUpdate("DELETE FROM supplies WHERE
   stock > 100");
13 System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock
1	stapler	40

1 row inserted

`executeUpdate` returns 1; SQLite assigns `id = 1`.

Step 3 / 8

VARIABLES

`i = 0` `ret = 1`

OUTPUT

What Does `executeUpdate` Return?

```
1  String[] names = {"stapler", "scissors", "tape"};
2  int[] stock = {40, 0, 15};
3  PreparedStatement ins = conn.prepareStatement(
4      "INSERT INTO supplies(name, stock) VALUES(?, ?)");
▶  for (int i = 0; i < names.length; i++) {
▶    ins.setString(1, names[i]);
▶    ins.setInt(2, stock[i]);
▶    ins.executeUpdate();
9  }
10 int u = stmt.executeUpdate(
11     "UPDATE supplies SET stock = stock + 10 WHERE stock <
   20");
12 int d = stmt.executeUpdate("DELETE FROM supplies WHERE
   stock > 100");
13 System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock
1	stapler	40
2	scissors	0

placeholders

`1st ? = "scissors"`

`2nd ? = 0`

1 row inserted

The same prepared statement runs again with new values.

Step 4 / 8

VARIABLES

`i = 1` `ret = 1`

OUTPUT

What Does `executeUpdate` Return?

```
1  String[] names = {"stapler", "scissors", "tape"};
2  int[] stock = {40, 0, 15};
3  PreparedStatement ins = conn.prepareStatement(
4      "INSERT INTO supplies(name, stock) VALUES(?, ?)");
▶  for (int i = 0; i < names.length; i++) {
▶    ins.setString(1, names[i]);
▶    ins.setInt(2, stock[i]);
▶    ins.executeUpdate();
9  }
10 int u = stmt.executeUpdate(
11     "UPDATE supplies SET stock = stock + 10 WHERE stock <
   20");
12 int d = stmt.executeUpdate("DELETE FROM supplies WHERE
   stock > 100");
13 System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock
1	stapler	40
2	scissors	0
3	tape	15

placeholders

`1st ? = "tape"`

`2nd ? = 15`

1 row inserted

The third item is inserted; when `i` = 3 the loop ends.

Step 5 / 8

VARIABLES

`i = 2` `ret = 1`

OUTPUT

What Does `executeUpdate` Return?

```
1  String[] names = {"stapler", "scissors", "tape"};
2  int[] stock = {40, 0, 15};
3  PreparedStatement ins = conn.prepareStatement(
4      "INSERT INTO supplies(name, stock) VALUES(?, ?)");
5  for (int i = 0; i < names.length; i++) {
6    ins.setString(1, names[i]);
7    ins.setInt(2, stock[i]);
8    ins.executeUpdate();
9  }
▶ int u = stmt.executeUpdate(
▶     "UPDATE supplies SET stock = stock + 10 WHERE stock <
   20");
12 int d = stmt.executeUpdate("DELETE FROM supplies WHERE
   stock > 100");
13 System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock
1	stapler	40
2	scissors	10
3	tape	25

WHERE stock < 20: 2 rows

`WHERE stock < 20` picks scissors (0) and tape (15); 10 is added to both. The return value is the **number of rows affected**: 2.

Step 6 / 8

VARIABLES

`u = 2`

OUTPUT

What Does `executeUpdate` Return?

```
1  String[] names = {"stapler", "scissors", "tape"};
2  int[] stock = {40, 0, 15};
3  PreparedStatement ins = conn.prepareStatement(
4      "INSERT INTO supplies(name, stock) VALUES(?, ?)");
5  for (int i = 0; i < names.length; i++) {
6    ins.setString(1, names[i]);
7    ins.setInt(2, stock[i]);
8    ins.executeUpdate();
9  }
10 int u = stmt.executeUpdate(
11     "UPDATE supplies SET stock = stock + 10 WHERE stock <
   20");
▶  int d = stmt.executeUpdate("DELETE FROM supplies WHERE
   stock > 100");
13 System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock
1	stapler	40
2	scissors	10
3	tape	25

no match: 0

No item has more than 100 in stock: no error, just 0.

Step 7 / 8

VARIABLES

`u = 2` `d = 0`

OUTPUT

[redacted]

What Does `executeUpdate` Return?

```
1  String[] names = {"stapler", "scissors", "tape"};
2  int[] stock = {40, 0, 15};
3  PreparedStatement ins = conn.prepareStatement(
4      "INSERT INTO supplies(name, stock) VALUES(?, ?)");
5  for (int i = 0; i < names.length; i++) {
6    ins.setString(1, names[i]);
7    ins.setInt(2, stock[i]);
8    ins.executeUpdate();
9  }
10 int u = stmt.executeUpdate(
11     "UPDATE supplies SET stock = stock + 10 WHERE stock <
   20");
12 int d = stmt.executeUpdate("DELETE FROM supplies WHERE
   stock > 100");
▶  System.out.println(u + " updated, " + d + " deleted");
```

supplies

id	name	stock
1	stapler	40
2	scissors	10
3	tape	25

From these counts the program can give accurate messages such as "record not found".

Step 8 / 8

VARIABLES

`u = 2` `d = 0`

OUTPUT

```
2 updated, 0 deleted
```

SQL Injection

```
String input = "x' OR '1'='1";   // the "name" the user typed

// DANGEROUS: input pasted into the SQL text
String sql = "SELECT name FROM people WHERE name = '"
           + input + "'";
System.out.println(sql);
ResultSet rs1 = stmt.executeQuery(sql);

// SAFE: input sent only as a value
PreparedStatement ps = conn.prepareStatement(
    "SELECT name FROM people WHERE name = ?");
ps.setString(1, input);
ResultSet rs2 = ps.executeQuery();
```

OUTPUT (3 PEOPLE IN THE TABLE; ROWS COUNTED)

```
SELECT name FROM people WHERE
name = 'x' OR '1'='1'
Concatenation: 3 rows
PreparedStatement: 0 rows
```

- The user's quote **changed** the SQL text: `'1'='1'` is always true, so every row came back.
- On a login screen, this means **logging in without a password**.
- Input sent with `?` is only a **value**; it can never become part of the SQL.

PREPAREDSTATEMENT · STEP-BY-STEP TRACE

SQL Injection, Step by Step

```
▶ String input = "x' OR '1'='1";
2 String sql = "SELECT name FROM people WHERE name = '"
3            + input + "'";
4 int n = 0;
5 try (ResultSet rs = stmt.executeQuery(sql)) {
6   while (rs.next()) n++;
7 }
8 System.out.println("Concatenation: " + n + " rows");
9 PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
(none yet)
```

id	name
1	Ali
2	Ayşe
3	Can

Instead of a name, the user typed text with a quote and an `OR`. The table has 3 people.

Step 1 / 10

VARIABLES

`input = x' OR '1'='1`

OUTPUT

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
▶  String sql = "SELECT name FROM people WHERE name = '"
▶              + input + "'";
4  int n = 0;
5  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
8  System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = 'x' OR '1'='1'
```

id	name
1	Ali
2	Ayşe
3	Can

After concatenation the user's quote **closed** the name value, and the `OR '1'='1'` after it is now part of the SQL.

Step 2 / 10

VARIABLES

`sql` = `SELECT … WHERE name = 'x' OR '1'='1'`

OUTPUT

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
▶ try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
8  System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = 'x' OR '1'='1'
```

id	name	condition
1	Ali	'Ali'='x' → false OR '1'='1' → TRUE
2	Ayşe	
3	Can	

The database evaluates the condition on each row: for Ali, the right side of `OR` is always true → the row is selected.

Step 3 / 10

VARIABLES

`rs = ResultSet`

OUTPUT

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
▶  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
8  System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = 'x' OR '1'='1'
```

id	name	condition
1	Ali	'Ali'='x' → false OR '1'='1' → TRUE
2	Ayşe	'Ayşe'='x' → false OR true → TRUE
3	Can	'Can'='x' → false OR true → TRUE

Ayşe and Can are selected too: the condition filters out nothing.

Step 4 / 10

VARIABLES

`rs = ResultSet`

OUTPUT

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
5  try (ResultSet rs = stmt.executeQuery(sql)) {
▶    while (rs.next()) n++;
▶  }
8  System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = 'x' OR '1'='1'
```

id	name	condition
1	Ali	'Ali'='x' → false OR '1'='1' → TRUE
2	Ayşe	'Ayşe'='x' → false OR true → TRUE
3	Can	'Can'='x' → false OR true → TRUE

3 rows returned

The loop runs 3 times. On a login screen this means logging in without a password.

Step 5 / 10

VARIABLES

`n = 3`

OUTPUT

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
5  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
▶ System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = 'x' OR '1'='1'
```

id	name	condition
1	Ali	'Ali'='x' → false OR '1'='1' → TRUE
2	Ayşe	'Ayşe'='x' → false OR true → TRUE
3	Can	'Can'='x' → false OR true → TRUE

3 rows returned

The result is printed.

Step 6 / 10

VARIABLES

`n = 3`

OUTPUT

```
Concatenation: 3 rows
```

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
5  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
8  System.out.println("Concatenation: " + n + " rows");
▶  PreparedStatement ps = conn.prepareStatement(
▶      "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = ?
```

id	name
1	Ali
2	Ayşe
3	Can

This time the SQL is compiled **first**; `?` is just a slot for a value. The statement's structure can no longer change.

Step 7 / 10

VARIABLES

`ps = PreparedStatement`

OUTPUT

```
Concatenation: 3 rows
```

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
5  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
8  System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
▶ ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = [x' OR '1'='1]
```

id	name
1	Ali
2	Ayşe
3	Can

The whole input is bound as **one text value**; the quotes in it are ordinary characters.

Step 8 / 10

VARIABLES

`1st ? = x' OR '1'='1`

OUTPUT

```
Concatenation: 3 rows
```

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
5  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
8  System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
▶ try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
14 System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = [x' OR '1'='1]
```

id	name	condition
1	Ali	name = "x' OR '1'='1" → false
2	Ayşe	name = "x' OR '1'='1" → false
3	Can	name = "x' OR '1'='1" → false

0 rows

No one's name equals this 12-character text: no rows.

Step 9 / 10

VARIABLES

`n = 0`

OUTPUT

```
Concatenation: 3 rows
```

SQL Injection, Step by Step

```
1  String input = "x' OR '1'='1";
2  String sql = "SELECT name FROM people WHERE name = '"
3             + input + "'";
4  int n = 0;
5  try (ResultSet rs = stmt.executeQuery(sql)) {
6    while (rs.next()) n++;
7  }
8  System.out.println("Concatenation: " + n + " rows");
9  PreparedStatement ps = conn.prepareStatement(
10     "SELECT name FROM people WHERE name = ?");
11 ps.setString(1, input);
12 n = 0;
13 try (ResultSet rs = ps.executeQuery()) { while
   (rs.next()) n++; }
▶  System.out.println("PreparedStatement: " + n + "
   rows");
```

SQL sent to the database:

```
SELECT name FROM people
WHERE name = [x' OR '1'='1]
```

id	name	condition
1	Ali	name = "x' OR '1'='1" → false
2	Ayşe	name = "x' OR '1'='1" → false
3	Can	name = "x' OR '1'='1" → false

0 rows

Same input, two different results. Always pass user-supplied values to SQL with `?`.

Step 10 / 10

VARIABLES

`n = 0`

OUTPUT

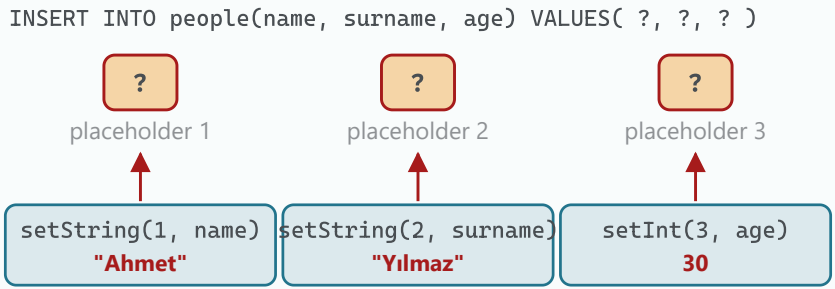
```
Concatenation: 3 rows
PreparedStatement: 0 rows
```

PreparedStatement and Placeholders

```
static void add(Connection conn, String name, String surname,
                int age) throws SQLException {
  String sql = "INSERT INTO people(name, surname, age) "
             + "VALUES(?, ?, ?)";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, name);      // 1st question mark
    ps.setString(2, surname);   // 2nd question mark
    ps.setInt(3, age);          // 3rd question mark
    ps.executeUpdate();
  }
}
```

Why?

+ **Safe**: prevents SQL injection.
+ No quoting trouble: `"O'Brien"` is stored without a hitch.
+ The same statement is **reused** with different values.



`setXxx(index, value)` fills each `?`; the index starts at **1**. Then call `executeUpdate()` with no arguments.

The People Program: Update and Delete Methods

```
static void update(Connection conn, String name, String surname,
                   int age) throws SQLException {
  String sql = "UPDATE people SET surname = ?, age = ? "
             + "WHERE name = ?";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, surname);
    ps.setInt(2, age);
    ps.setString(3, name);
    ps.executeUpdate();
  }
}

static void delete(Connection conn, String name) throws SQLException {
  String sql = "DELETE FROM people WHERE name = ?";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, name);
    ps.executeUpdate();
  }
}
```

- Each method takes the `Connection` as a **parameter**: the connection is opened once and used for many jobs.
- The `?`s are numbered by their **order** in the SQL: in `update`, the name is placeholder 3.
- `throws SQLException`: errors pass to the caller; since `main` also declares `throws`, an error stops the program with a stack trace.

The People Program: The List Method

```
static void list(Connection conn) throws SQLException {
  String sql = "SELECT id, name, surname, age FROM people";
  try (Statement st = conn.createStatement();
       ResultSet rs = st.executeQuery(sql)) {
    while (rs.next()) {
      System.out.println(rs.getInt("id") + " " + rs.getString("name")
          + " " + rs.getString("surname") + " " + rs.getInt("age"));
    }
    System.out.println("---");
  }
}
```

OUTPUT (FIRST CALL TO LIST)

```
1 Ahmet Akdeniz 35
2 Ayşe Demir 40
---
```

- The query has no variable values, so a plain `Statement` is enough.
- Each call to `list` reads the table **as it is now** and ends with `---`.

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
▶    try (Connection conn = DriverManager.getConnection(url);
▶         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
12     delete(conn, "Ahmet");
13     list(conn);
14   }
15 }
```

people

id	name	surname	age
(table empty)			

`people.db` is created if missing; the connection and `stmt` are opened.

Step 1 / 9

CALL STACK

`main()`

OUTPUT

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
▶      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
▶          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
▶          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
12     delete(conn, "Ahmet");
13     list(conn);
14   }
15 }
```

people

id	name	surname	age
(table empty)			

The table is created; it is empty for now.

Step 2 / 9

CALL STACK

`main()`

OUTPUT

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
▶      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
12     delete(conn, "Ahmet");
13     list(conn);
14   }
15 }
```

people

id	name	surname	age
1	Ahmet	Yılmaz	30

`add`: the `?`s are filled and the row is inserted, `id = 1`.

Step 3 / 9

CALL STACK

`add(conn, "Ahmet", "Yılmaz", 30)`

`main()`

OUTPUT

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
▶      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
12     delete(conn, "Ahmet");
13     list(conn);
14   }
15 }
```

people

id	name	surname	age
1	Ahmet	Yılmaz	30
2	Ayşe	Demir	40

The second person is added with `id = 2`.

Step 4 / 9

CALL STACK

```
add(conn, "Ayşe", "Demir", 40)
main()
```

OUTPUT

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
▶      update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
12     delete(conn, "Ahmet");
13     list(conn);
14   }
15 }
```

people

id	name	surname	age
1	Ahmet	Akdeniz	35
2	Ayşe	Demir	40

`WHERE name = ?` : Ahmet's surname and age change.

Step 5 / 9

CALL STACK

```
update(conn, "Ahmet", "Akdeniz", 35)
main()
```

OUTPUT

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
▶      list(conn);
12     delete(conn, "Ahmet");
13     list(conn);
14   }
15 }
```

people

id	name	surname	age
1	Ahmet	Akdeniz	35
2	Ayşe	Demir	40

`list` prints every row and `---`.

Step 6 / 9

CALL STACK

- **list(conn)**
- main()

OUTPUT

```
1 Ahmet Akdeniz 35
2 Ayşe Demir 40
---
```

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
▶      delete(conn, "Ahmet");
13     list(conn);
14   }
15 }
```

people

id	name	surname	age
~~1~~	~~Ahmet~~	~~Akdeniz~~	~~35~~
2	Ayşe	Demir	40

Ahmet's row is deleted.

Step 7 / 9

CALL STACK

```
delete(conn, "Ahmet")
main()
```

OUTPUT

```
1 Ahmet Akdeniz 35
2 Ayşe Demir 40
---
```

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
12     delete(conn, "Ahmet");
▶      list(conn);
14   }
15 }
```

people

id	name	surname	age
2	Ayşe	Demir	40

Only Ayşe is left in the table.

Step 8 / 9

CALL STACK

- list(conn)
- main()

OUTPUT

```
1 Ahmet Akdeniz 35
2 Ayşe Demir 40
---
2 Ayşe Demir 40
---
```

The People Program, Step by Step

```
1  public static void main(String[] args) throws SQLException {
2    String url = "jdbc:sqlite:people.db";
3    try (Connection conn = DriverManager.getConnection(url);
4         Statement stmt = conn.createStatement()) {
5      stmt.execute("CREATE TABLE IF NOT EXISTS people ("
6          + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
7          + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
8      add(conn, "Ahmet", "Yılmaz", 30);
9      add(conn, "Ayşe", "Demir", 40);
10     update(conn, "Ahmet", "Akdeniz", 35);
11     list(conn);
12     delete(conn, "Ahmet");
13     list(conn);
▶    }
15 }
```

people

id	name	surname	age
2	Ayşe	Demir	40

The connection closes; the data **persists** in `people.db`.

Step 9 / 9

CALL STACK

`main()`

OUTPUT

```
1 Ahmet Akdeniz 35
2 Ayşe Demir 40
---
2 Ayşe Demir 40
---
```

Generated Keys: `getGeneratedKeys`

```
String sql = "INSERT INTO students(name, dept_id) "
    + "VALUES(?, ?)";
try (PreparedStatement ps = conn.prepareStatement(sql,
        Statement.RETURN_GENERATED_KEYS)) {
  ps.setString(1, "Ece");
  ps.setInt(2, 3);
  ps.executeUpdate();
  try (ResultSet k = ps.getGeneratedKeys()) {
    if (k.next()) {
      System.out.println("New id: " + k.getInt(1));
    }
  }
}
```

OUTPUT

```
New id: 5
```

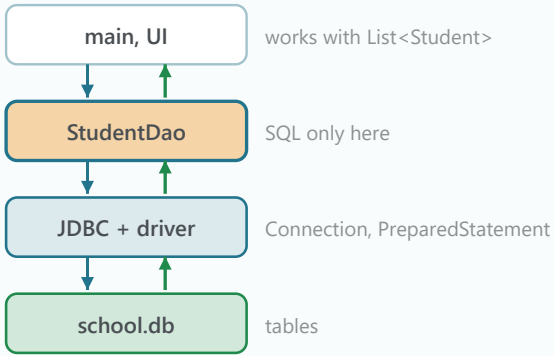
- The database decides a new row's `id`; the program learns it with `getGeneratedKeys()`. Example: to add a new order's line items, you need the order's `id`.
- Pass `Statement.RETURN_GENERATED_KEYS` to `prepareStatement`; the single column of the returned `ResultSet` is the new key.
- The table had 4 students; Ece got `id = 5`.
- If a deleted record's number must never be reused (invoice or order numbers), use `AUTOINCREMENT`. MySQL's equivalent is `AUTO_INCREMENT`.

1, 2, 3 inserted; 3 deleted; new row →	id
`id INTEGER PRIMARY KEY`	**3** (largest id + 1)
`id INTEGER PRIMARY KEY AUTOINCREMENT`	**4** (used numbers aren't reused)

ADVANCED JDBC

From Rows to Objects: The DAO Pattern

```
class Student {
  private final int id;
  private final String name;
  Student(int id, String name) {
    this.id = id;
    this.name = name;
  }
  public String toString() {
    return "#" + id + " " + name;
  }
}
```



StudentDao method	SQL	Returns
`add(name, dept)`	`INSERT`	the new `id`
`find(id)`	`SELECT … WHERE id = ?`	a `Student` or `null`
`inDept(dept)`	`SELECT … WHERE dept_id = ?`	`List<Student>`
`delete(id)`	`DELETE`	deleted? (`boolean`)

- In an object-oriented program, each row becomes an **object**: table ≈ class, row ≈ object.
- A **DAO** (Data Access Object) gathers all of a table's SQL in one class. The rest of the program never sees SQL.
- If the database changes, only the DAO changes. The DAO gets its connection through the constructor.
- **ORM** (object-relational mapping) libraries such as JPA and Hibernate do this mapping automatically.

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
7      ps.setInt(1, dept);
8      try (ResultSet rs = ps.executeQuery()) {
9        while (rs.next()) {
10         list.add(new Student(rs.getInt("id"),
11                              rs.getString("name")));
12       }
13     }
14   }
15   return list;
16 }
17 // inside main:
▶ List<Student> list = inDept(conn, 1);
19 for (Student s : list) System.out.println(s);
```

list (List<Student>)

empty: []

`main` calls the method with `dept` = 1. The connection is open in main.

Step 1 / 8

VARIABLES

`dept = 1`

CALL STACK

`inDept(conn, 1)`

`main()`

OUTPUT

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
7      ps.setInt(1, dept);
8      try (ResultSet rs = ps.executeQuery()) {
9        while (rs.next()) {
10         list.add(new Student(rs.getInt("id"),
11                              rs.getString("name")));
12       }
13     }
14   }
15   return list;
16 }
17 // inside main:
18 List<Student> list = inDept(conn, 1);
19 for (Student s : list) System.out.println(s);
```

list (List<Student>)

empty: []

An empty list is created; the query is prepared with `?`.

Step 2 / 8

VARIABLES

`dept = 1` `list = []`

CALL STACK

`inDept(conn, 1)`

`main()`

OUTPUT

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
▶      ps.setInt(1, dept);
▶      try (ResultSet rs = ps.executeQuery()) {
9        while (rs.next()) {
10         list.add(new Student(rs.getInt("id"),
11                              rs.getString("name")));
12       }
13     }
14   }
15   return list;
16 }
17 // inside main:
18 List<Student> list = inDept(conn, 1);
19 for (Student s : list) System.out.println(s);
```

rs (dept 1)

id	name
1	Ada
3	Cem

list (List<Student>)

empty: []

cursor before the first row

`? = 1` is bound and the query runs: dept 1 has Ada and Cem.

Step 3 / 8

VARIABLES

`dept = 1` `list = []`

CALL STACK

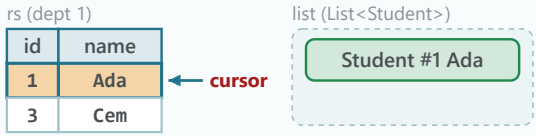
`inDept(conn, 1)`

`main()`

OUTPUT

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
7      ps.setInt(1, dept);
8      try (ResultSet rs = ps.executeQuery()) {
▶        while (rs.next()) {
▶          list.add(new Student(rs.getInt("id"),
▶                               rs.getString("name")));
12       }
13     }
14   }
15   return list;
16 }
17 // inside main:
18 List<Student> list = inDept(conn, 1);
19 for (Student s : list) System.out.println(s);
```



First row: `id` = 1, `name` = Ada are read; `new Student(1, "Ada")` is added to the list.

Step 4 / 8

VARIABLES

`list = [#1 Ada]`

CALL STACK

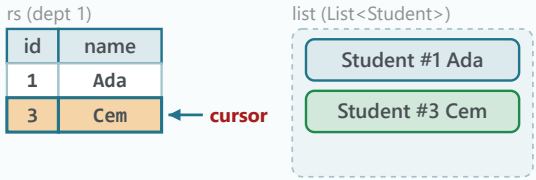
`inDept(conn, 1)`

`main()`

OUTPUT

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
7      ps.setInt(1, dept);
8      try (ResultSet rs = ps.executeQuery()) {
▶        while (rs.next()) {
▶          list.add(new Student(rs.getInt("id"),
▶                               rs.getString("name")));
12       }
13     }
14   }
15   return list;
16 }
17 // inside main:
18 List<Student> list = inDept(conn, 1);
19 for (Student s : list) System.out.println(s);
```



The second row gives a second object: `Student #3 Cem`.

Step 5 / 8

VARIABLES

`list = [#1 Ada, #3 Cem]`

CALL STACK

`inDept(conn, 1)`

`main()`

OUTPUT

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
7      ps.setInt(1, dept);
8      try (ResultSet rs = ps.executeQuery()) {
▶        while (rs.next()) {
10         list.add(new Student(rs.getInt("id"),
11                              rs.getString("name")));
12       }
▶      }
▶    }
15   return list;
16 }
17 // inside main:
18 List<Student> list = inDept(conn, 1);
19 for (Student s : list) System.out.println(s);
```

rs (dept 1)

id	name
1	Ada
3	Cem

list (List<Student>)

- Student #1 Ada
- Student #3 Cem

rs and ps closed

`next()` is false: the try blocks end and `rs` and `ps` close. The connection is closed by `main`, which opened it.

Step 6 / 8

VARIABLES

`list = [#1 Ada, #3 Cem]`

CALL STACK

`inDept(conn, 1)`

`main()`

OUTPUT

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
7      ps.setInt(1, dept);
8      try (ResultSet rs = ps.executeQuery()) {
9        while (rs.next()) {
10         list.add(new Student(rs.getInt("id"),
11                              rs.getString("name")));
12       }
13     }
14   }
▶    return list;
16 }
17 // inside main:
▶  List<Student> list = inDept(conn, 1);
19 for (Student s : list) System.out.println(s);
```

list (List<Student>)

- Student #1 Ada
- Student #3 Cem

returns objects, not rows

The method returns a `List<Student>` (2 items); its frame leaves the stack.

Step 7 / 8

VARIABLES

`list = [#1 Ada, #3 Cem]`

CALL STACK

`main()`

OUTPUT

From Rows to Objects, Step by Step

```
1  static List<Student> inDept(Connection conn,
2      int dept) throws SQLException {
3    List<Student> list = new ArrayList<>();
4    String sql = "SELECT id, name FROM students "
5        + "WHERE dept_id = ? ORDER BY id";
6    try (PreparedStatement ps = conn.prepareStatement(sql)) {
7      ps.setInt(1, dept);
8      try (ResultSet rs = ps.executeQuery()) {
9        while (rs.next()) {
10         list.add(new Student(rs.getInt("id"),
11                              rs.getString("name")));
12       }
13     }
14   }
15   return list;
16 }
17 // inside main:
18 List<Student> list = inDept(conn, 1);
▶ for (Student s : list) System.out.println(s);
```

list (List<Student>)

Student #1 Ada

Student #3 Cem

Each object is printed via `toString()`. `main` knows nothing about SQL.

Step 8 / 8

VARIABLES

`list = [#1 Ada, #3 Cem]`

CALL STACK

`main()`

OUTPUT

```
#1 Ada
#3 Cem
```

Same Code, Another Database: MySQL

```
String url = "jdbc:mysql://localhost:3306/testdb";
String user = "root";
String password = "secret";
String sql = "SELECT id, name, email FROM users";
try (Connection conn =
         DriverManager.getConnection(url, user, password);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getInt("id") + " "
        + rs.getString("name"));
  }
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

- The JDBC code is **the same**; what changes is the **URL**, the **user name**, the **password** and the driver jar (`mysql-connector-j`).
- MySQL is a **server**: it must be installed and running first, and the `testdb` database created.
- The driver class is `com.mysql.cj.jdbc.Driver` ; the old name `com.mysql.jdbc.Driver` is deprecated.
- Real projects never hard-code the password; they read it from a config file or an environment variable.

Transactions and ACID

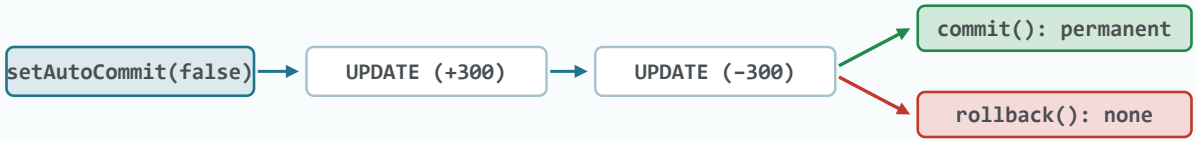
- **A transfer**: add 300 TL to Ece's account **and** take 300 TL from Ali's. If one of the two `UPDATE`s happens and the other doesn't, money is **created from nothing** or **lost**.
- A **transaction** is one or more SQL statements run as a single unit of work: **all or nothing**.
- JDBC's default is **auto-commit**: every statement is permanent at once.
- `conn.setAutoCommit(false)` → statements → `conn.commit()` (make permanent) or `conn.rollback()` (undo).

Atomicity
The transaction is applied **entirely or not at all**.

Consistency
After the transaction, **all constraints hold** (e.g. balance ≥ 0).

Isolation
Concurrent transactions **don't see** each other's half-done state.

Durability
Committed data survives even a power cut.



A Transfer: Setup and Result

```
static final String DEPOSIT =
    "UPDATE accounts SET balance = balance + ? WHERE id = ?";
static final String WITHDRAW =
    "UPDATE accounts SET balance = balance - ? WHERE id = ?";

public static void main(String[] args) throws SQLException {
  String url = "jdbc:sqlite::memory:";  // in memory, temporary
  try (Connection conn = DriverManager.getConnection(url);
       Statement st = conn.createStatement()) {
    st.execute("CREATE TABLE accounts (id INTEGER PRIMARY KEY, "
        + "owner TEXT, balance INTEGER CHECK (balance >= 0))");
    st.execute("INSERT INTO accounts(owner, balance) "
        + "VALUES ('Ali', 1000), ('Ece', 500)");
    transfer(conn, 1, 2, 300);    // Ali → Ece 300 TL
    transfer(conn, 1, 2, 5000);   // Ali can't afford it
    printAll(conn);
  }
}
```

OUTPUT

```
300 TL transferred.
Rolled back: 5000 TL
Ali: 700
Ece: 800
```

- `CHECK (balance >= 0)` : a statement that drives a balance negative throws `SQLException` .
- In the second transfer, 5000 TL **had already been deposited** to Ece; `rollback` undid that too.
- `printAll` is a `SELECT` loop, like `list` .

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
▶    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
6      p1.setInt(1, amount);  p1.setInt(2, to);
7      p1.executeUpdate();              // deposit to receiver
8      p2.setInt(1, amount);  p2.setInt(2, from);
9      p2.executeUpdate();              // withdraw from sender
10     conn.commit();                   // make permanent
11     System.out.println(amount + " TL transferred.");
12   } catch (SQLException e) {
13     conn.rollback();                 // undo
14     System.out.println("Rolled back: " + amount + " TL");
15   } finally {
16     conn.setAutoCommit(true);
17   }
18 }
```

transaction started

id	owner	balance
1	Ali	1000
2	Ece	500

Auto-commit is turned off: from now on, changes are **tentative** until `commit`.

Step 1 / 8

CALL STACK

`transfer(conn, 1, 2, 300)`

`main()`

OUTPUT

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
3    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
▶      p1.setInt(1, amount);  p1.setInt(2, to);
▶      p1.executeUpdate();              // deposit to receiver
8      p2.setInt(1, amount);  p2.setInt(2, from);
9      p2.executeUpdate();              // withdraw from sender
10     conn.commit();                   // make permanent
11     System.out.println(amount + " TL transferred.");
12   } catch (SQLException e) {
13     conn.rollback();                 // undo
14     System.out.println("Rolled back: " + amount + " TL");
15   } finally {
16     conn.setAutoCommit(true);
17   }
18 }
```

tentative (no commit)

id	owner	balance
1	Ali	1000
2	Ece	800

300 TL is deposited to Ece: 500 → 800.

Step 2 / 8

CALL STACK

`transfer(conn, 1, 2, 300)`

`main()`

OUTPUT

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
3    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
6      p1.setInt(1, amount);  p1.setInt(2, to);
7      p1.executeUpdate();              // deposit to receiver
▶      p2.setInt(1, amount);  p2.setInt(2, from);
▶      p2.executeUpdate();              // withdraw from sender
10     conn.commit();                   // make permanent
11     System.out.println(amount + " TL transferred.");
12   } catch (SQLException e) {
13     conn.rollback();                 // undo
14     System.out.println("Rolled back: " + amount + " TL");
15   } finally {
16     conn.setAutoCommit(true);
17   }
18 }
```

tentative (no commit)

id	owner	balance
1	Ali	700
2	Ece	800

300 TL is withdrawn from Ali: 1000 → 700. The constraint holds.

Step 3 / 8

CALL STACK

`transfer(conn, 1, 2, 300)`

`main()`

OUTPUT

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
3    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
6      p1.setInt(1, amount);  p1.setInt(2, to);
7      p1.executeUpdate();               // deposit to receiver
8      p2.setInt(1, amount);  p2.setInt(2, from);
9      p2.executeUpdate();               // withdraw from sender
▶      conn.commit();                    // make permanent
▶      System.out.println(amount + " TL transferred.");
12   } catch (SQLException e) {
13     conn.rollback();                  // undo
14     System.out.println("Rolled back: " + amount + " TL");
15   } finally {
16     conn.setAutoCommit(true);
17   }
18 }
```

permanent (committed)

id	owner	balance
1	Ali	700
2	Ece	800

`commit()` : both changes become permanent together.

Step 4 / 8

CALL STACK

`transfer(conn, 1, 2, 300)`

`main()`

OUTPUT

```
300 TL transferred.
```

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
3    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
6      p1.setInt(1, amount);  p1.setInt(2, to);
7      p1.executeUpdate();               // deposit to receiver
8      p2.setInt(1, amount);  p2.setInt(2, from);
9      p2.executeUpdate();               // withdraw from sender
10     conn.commit();                    // make permanent
11     System.out.println(amount + " TL transferred.");
12   } catch (SQLException e) {
13     conn.rollback();                  // undo
14     System.out.println("Rolled back: " + amount + " TL");
15   } finally {
▶      conn.setAutoCommit(true);
17   }
18 }
```

permanent state

id	owner	balance
1	Ali	700
2	Ece	800

`finally` : auto-commit is turned back on; the method ends.

Step 5 / 8

CALL STACK

```
transfer(conn, 1, 2, 300)
main()
```

OUTPUT

```
300 TL transferred.
```

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
▶    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
▶      p1.setInt(1, amount);  p1.setInt(2, to);
▶      p1.executeUpdate();               // deposit to receiver
8      p2.setInt(1, amount);  p2.setInt(2, from);
9      p2.executeUpdate();               // withdraw from sender
10     conn.commit();                    // make permanent
11     System.out.println(amount + " TL transferred.");
12   } catch (SQLException e) {
13     conn.rollback();                  // undo
14     System.out.println("Rolled back: " + amount + " TL");
15   } finally {
16     conn.setAutoCommit(true);
17   }
18 }
```

tentative (no commit)

id	owner	balance
1	Ali	700
2	Ece	5800

Second call: 5000 TL is deposited to Ece (tentatively): 800 → 5800.

CALL STACK

```
transfer(conn, 1, 2, 5000)
main()
```

OUTPUT

```
300 TL transferred.
```

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
3    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
6      p1.setInt(1, amount);  p1.setInt(2, to);
7      p1.executeUpdate();              // deposit to receiver
▶      p2.setInt(1, amount);  p2.setInt(2, from);
▶      p2.executeUpdate();              // withdraw from sender
10     conn.commit();                   // make permanent
11     System.out.println(amount + " TL transferred.");
12   } catch (SQLException e) {
13     conn.rollback();                 // undo
14     System.out.println("Rolled back: " + amount + " TL");
15   } finally {
16     conn.setAutoCommit(true);
17   }
18 }
```

tentative (no commit)

id	owner	balance
~~1~~	~~Ali~~	~~700~~
2	Ece	5800

700 − 5000 would be < 0: the `CHECK` fails and `SQLException` is thrown.

Step 7 / 8

CALL STACK

`transfer(conn, 1, 2, 5000)`

`main()`

OUTPUT

```
300 TL transferred.
```

commit and rollback, Step by Step

```
1  static void transfer(Connection conn, int from, int to,
2                       int amount) throws SQLException {
3    conn.setAutoCommit(false);          // transaction starts
4    try (PreparedStatement p1 =
   conn.prepareStatement(DEPOSIT);
5         PreparedStatement p2 =
   conn.prepareStatement(WITHDRAW)) {
6      p1.setInt(1, amount);  p1.setInt(2, to);
7      p1.executeUpdate();               // deposit to receiver
8      p2.setInt(1, amount);  p2.setInt(2, from);
9      p2.executeUpdate();               // withdraw from sender
10     conn.commit();                    // make permanent
11     System.out.println(amount + " TL transferred.");
▶    } catch (SQLException e) {
▶      conn.rollback();                  // undo
▶      System.out.println("Rolled back: " + amount + " TL");
15   } finally {
16     conn.setAutoCommit(true);
17   }
18 }
```

CALL STACK

```
transfer(conn, 1, 2, 5000)
main()
```

rollback: undone

id	owner	balance
1	Ali	700
2	Ece	800

`rollback()` : the tentative deposit is **undone** too. No money was lost or created.

Step 8 / 8

OUTPUT

```
300 TL transferred.
Rolled back: 5000 TL
```

Batching: `addBatch` and `executeBatch`

```
conn.setAutoCommit(false);
try (PreparedStatement ps = conn.prepareStatement(
        "INSERT INTO readings(value) VALUES(?)")) {
  for (int i = 1; i <= 1000; i++) {
    ps.setInt(1, i);
    ps.addBatch();
  }
  int[] result = ps.executeBatch();
  conn.commit();
  System.out.println(result.length + " statements, "
      + "the first affected " + result[0] + " row");
} catch (SQLException e) {
  conn.rollback();
  throw e;
} finally {
  conn.setAutoCommit(true);
}
```

OUTPUT

```
1000 statements, the first affected 1 row
```

- `addBatch()` adds the filled-in statement to a queue; `executeBatch()` sends the whole queue and returns each statement's affected row count as an `int[]`.
- With auto-commit on, every `INSERT` is its own transaction and is written to disk separately. In one test with a `.db` file on disk, 1000 rows took about **2.6 s** one by one and about **0.01 s** batched in one transaction (varies by machine).
- It is all one transaction, so if one fails, `rollback()` undoes the whole queue's effect: all or nothing.
- For very large loads, send the queue in pieces (e.g. an `executeBatch` every 500 `addBatch`es) so memory doesn't balloon.

Summary

Topic	Essence
Database	Persistent, organized, queryable data managed by a DBMS. Tables, rows, columns, primary keys.
SQLite	Serverless, embedded, one file. URL: `jdbc:sqlite:sample.db`; the driver jar must be on the classpath.
SQL	`CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`; `WHERE`, `ORDER BY`, `LIMIT`.
Relations	Foreign keys; `JOIN … ON`; `GROUP BY` / `HAVING`; `IS NULL` for `NULL`.
JDBC	`DriverManager` → `Connection` → `Statement` / `PreparedStatement` → `ResultSet`.
Execution	`execute` (tables), `executeUpdate` → `int`, `executeQuery` → `ResultSet`; `while (rs.next())`.
Security	User input always goes through `?` and `setXxx`, never into the SQL text.
Transactions	`setAutoCommit(false)`, `commit()`, `rollback()`: all or nothing (ACID).
Resources	try-with-resources: `rs`, `stmt`, `conn` close in reverse order; `SQLException` is a checked exception.

SUMMARY

Common Mistakes

Driver not on the classpath
The jar wasn't added:
`No suitable driver found for jdbc:sqlite:sample.db`

Not catching SQLException
No try or throws (compile error):
`unreported exception SQLException; must be caught or declared to be thrown`

Missing or misnamed table
Querying before creating the table:
`no such table: users`

Text without quotes
`VALUES(Ahmet, …)`:
`no such column: Ahmet`

Concatenating text with an apostrophe
`VALUES('O'Brien')`:
`near "Brien": syntax error`

The wrong execute method
`INSERT` with `executeQuery`:
`query does not return ResultSet`

Starting the index at 0
`rs.getString(0)`; indexes start at 1:
`column 0 out of bounds [1,3]`

UPDATE / DELETE without WHERE
No error; it changes or deletes **every row**.

SUMMARY

Common Mistakes: Relations, Grouping, JDBC

Comparing with `= NULL`
`WHERE dept_id = NULL` gives no error but returns **0 rows**. Use `IS NULL`.

Joining without `ON`
`FROM students, depts`: every row pairs with every row, 4 × 3 = **12 rows**.

Ambiguous column name
Both tables have `name`, and no table name was given:
`ambiguous column name: name`

An aggregate in `WHERE`
Group conditions go in `HAVING`:
`misuse of aggregate: SUM()`

Not enabling foreign key checks
In SQLite, without `PRAGMA foreign_keys = ON`, `dept_id = 9` is silently accepted.

Calling `wasNull()` too late
It checks the last column read; call it right after the relevant `getInt`.

Selecting an ungrouped column
`SELECT customer, amount … GROUP BY customer`: SQLite returns the amount of **some** row in the group; many databases reject it.

Thousands of `INSERT`s with auto-commit on
Each row is its own transaction: very slow. Use `setAutoCommit(false)`, `addBatch`, `commit`.

SUMMARY

Practice Problems 1

Problem 1: Which rows does the SELECT return?

```
CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT);
INSERT INTO users(name) VALUES ('Ali');
INSERT INTO users(name) VALUES ('Can');
DELETE FROM users WHERE id = 2;
INSERT INTO users(name) VALUES ('Ece');
UPDATE users SET name = 'Efe' WHERE id = 1;
SELECT * FROM users;
```

Problem 2

The people program (with `people.db`) is run **a second time**. What are the first block of output (up to the first `---`) and the last block?

▶ **Show answers**

SUMMARY

Practice Problems 2

Problem 3: A login check

```
String sql = "SELECT * FROM people WHERE name = '" + name
           + "' AND password = '" + password + "'";
ResultSet rs = stmt.executeQuery(sql);
boolean loggedIn = rs.next();
```

The table has 3 people. If the user types `ahmet` as the name and `' OR '1'='1` as the password, what SQL results, how many rows come back, and does the login succeed?

Problem 4

If the `setAutoCommit(false)`, `commit()` and `rollback()` lines were removed from `transfer` (each statement permanent on its own), what would Ali's and Ece's balances be at the end?

▶ **Show answers**

SUMMARY

Exercises

1. Change the delete program: if `executeUpdate` returns 0, print `Record not found.`
2. Create a `students(id, name, midterm, final)` table; insert Ada (60, 70), Bora (30, 50) and Cem (80, 40) in a loop with `PreparedStatement`. List those whose average (midterm 40%, final 60%) is 50 or more, highest first.
3. Write a menu program: 1-Add, 2-Update, 3-Delete, 4-List. Use `Scanner` and `switch`; if the id to update/delete doesn't exist, print `User not found.`
4. Save a game's scores in a `scores(player, score)` table and list the top 10.

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: JDBC Database Access (Establishing a Connection; Processing SQL Statements with JDBC; Using Prepared Statements; Using Transactions).
- Oracle, *Java SE 21 API Documentation*: the `java.sql` package (`DriverManager` , `Connection` , `PreparedStatement` , `ResultSet`).
- SQLite documentation, sqlite.org: *SQL As Understood By SQLite*; *Datatypes In SQLite*; *Appropriate Uses For SQLite*.
- Xerial, *SQLite JDBC Driver*, github.com/xerial/sqlite-jdbc.
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Accessing Databases with JDBC).
- C. S. Horstmann, *Core Java, Volume II: Advanced Features*, 12th ed., Pearson (Database Programming).
- OWASP, *SQL Injection Prevention Cheat Sheet*.

The End

Chapter 11: Database Programming

Previous: Chapter 10: Threads · End of course