# **Introduction to MySQL**

**What & Why it Came**

* **MySQL** is an open‑source Relational Database Management System (RDBMS) first released in 1995 by MySQL AB to address the need for a fast, reliable, and freely available database for web applications.
* It implements the SQL language (Structured Query Language) to define, manipulate and query data in tables.
* **Key motivations**: performance, ease of use, broad platform support, and community‑driven development.

**Core Rules & Properties**

1. **ACID Compliance** (Atomicity, Consistency, Isolation, Durability) when using transactional storage engines (e.g., InnoDB).
2. **Strict SQL Syntax**: identifiers, case sensitivity (table names on Linux), proper quoting (``` backticks).
3. **Storage Engines**: InnoDB (transactions, foreign keys), MyISAM (fast reads, no transactions).
4. **User Privileges & Security**: granular GRANT/REVOKE controls.

**When & Where to Use**

* **Web back‑ends**: powering content management systems (WordPress), e‑commerce (Magento).
* **Analytics**: moderate‑scale reporting when paired with BI tools.
* **Embedded**: light usage in desktop or IoT apps.

**Real‑World Example**  
An online store keeps product, customer, and order tables in MySQL. InnoDB enforces foreign‑key constraints so that an order\_item cannot exist without a matching order row.

**Installation of MySQL with MySQL Workbench**

**What & Why**

* **MySQL Server**: the database engine.
* **MySQL Workbench**: GUI tool for database design, query writing, administration.

**Rules & Best Practices**

1. **Secure Installation**: run mysql\_secure\_installation to set root password, remove anonymous users, disable remote root login.
2. **Port Conflicts**: default is TCP 3306; ensure it’s free or choose another.
3. **User Accounts**: create least‑privilege users for apps, avoid using root.

**Installation Steps (Windows/Mac/Linux)**

1. **Download** the MySQL Community Installer (Windows) or package from dev.mysql.com.  
   https://dev.mysql.com/downloads/installer/
2. **Run Installer**: select Server + Workbench.
3. **Configure Server**: choose Standalone/MySQL as a Service, set authentication method (caching\_sha2), UTF‑8 character set.
4. **Secure**: mysql\_secure\_installation.
5. **Verify**: connect via Workbench → “Local instance,” or CLI mysql -u root -p.

**Or Can use online complier :**  
[MySQL Online Editor](https://onecompiler.com/mysql)  
  
**When & Where to Use**

* **Local Development**: spin up a dev instance with Workbench for schema design, visual querying.
* **DBA Tasks**: monitor performance, set up replication, design ER‑diagrams.

**Real‑World Example**  
A team uses Workbench to reverse‑engineer an existing schema into an ER‑diagram, then iteratively refines table relationships before deployment.

**DBMS vs RDBMS**

| **Aspect** | **DBMS** | **RDBMS** |
| --- | --- | --- |
| Data Model | File‑based or hierarchical | Relational tables with rows/columns |
| Key Concepts | CRUD only | **SQL**, relationships, ACID transactions |
| Schema Enforcement | Optional | Strict schema, data integrity via constraints |
| Example Systems | NoSQL stores, file systems | MySQL, PostgreSQL, Oracle |

**Rules & Why RDBMS Evolved**

* Early DBMS offered basic storage but lacked relational integrity.
* **RDBMS** introduced:
  + **Foreign keys** to maintain referential integrity
  + **Joins** for querying across tables
  + **Transactions** for multi‑step consistency

**When & Where to Use**

* **DBMS** (e.g., MongoDB) for flexible schemas or hierarchical data.
* **RDBMS** when:
  + Data is highly structured
  + Strong consistency and complex queries are required

**Real‑World Example**  
Banking applications use RDBMS to ensure every withdrawal and deposit balances correctly across multiple tables (accounts, transactions, ledgers).

**MySQL Datatypes**

MySQL datatypes dictate how data is stored, validated, and compared. Choosing the right type impacts performance, storage, and accuracy.

| **Category** | **Examples** | **Usage Rules & Notes** | **Real‑World Use Case** |
| --- | --- | --- | --- |
| **Numeric** | INT, BIGINT, DECIMAL(p,s) | Specify signed/unsigned; DECIMAL for exact currency (avoid floats). | DECIMAL(10,2) for invoice totals. |
| **Date & Time** | DATE, DATETIME, TIMESTAMP | TIMESTAMP auto‑updates by default; timezone‑aware. | created\_at TIMESTAMP DEFAULT NOW() |
| **String** | VARCHAR(n), TEXT, CHAR(n) | VARCHAR for variable length; TEXT for large text; CHAR for fixed. | username VARCHAR(50). |
| **Binary/Blob** | BLOB, VARBINARY(n) | Store images/docs, careful with client memory. | image MEDIUMBLOB for photo storage. |
| **Spatial** | POINT, GEOMETRY | Geolocation queries with spatial indexes. | location POINT(lat, lon) for maps. |
| **JSON** | JSON | Schema‑less storage, validated JSON; functions for querying. | Store metadata or flexible configs. |

**Rules & Best Practices**

1. **Minimize Size**: pick the smallest type that fits your data.
2. **Avoid Over‑precision**: don’t oversize VARCHAR(65535) if VARCHAR(255) suffices.
3. **Use Appropriate Types**: e.g., use DATE not VARCHAR to enable date functions.
4. **NULL vs NOT NULL**: declare NOT NULL where possible for performance.

**Operators in MySQL**

Operators let you build expressions in SELECT, WHERE, UPDATE, etc.

| **Category** | **Operators** | **Usage Rules** | **Example** |
| --- | --- | --- | --- |
| **Arithmetic** | +, -, \*, /, % | Division by zero returns NULL (unless SQL mode). | SELECT price \* quantity AS total; |
| **Comparison** | =, <>/!=, <, >, <= | NULLs cause comparisons to yield UNKNOWN. | WHERE age >= 18 AND verified = 1 |
| **Logical** | AND, OR, NOT, XOR | Short‑circuit evaluation. | WHERE status='active' OR last\_login > '2025-01-01' |
| **Bitwise** | &, ` | , ^, ~, <<, >>` | Operate on integer bit patterns. |
| **String** | CONCAT(), LIKE, REGEXP | % and \_ wildcards with LIKE; full regex with REGEXP. | WHERE name LIKE 'A%'; |
| **Assignment** | :=, = in SET clause | Use := in expressions; = to assign column values in UPDATE. | SET counter = counter + 1 |

**Operator Precedence**  
From highest to lowest:

1. Arithmetic (\*, /, %)
2. Arithmetic (+, −)
3. Bitwise shifts (<<, >>)
4. Comparison
5. Bitwise
6. Logical NOT
7. Logical AND
8. Logical OR

**1. DDL (Data Definition Language)**

**What & Why it Came**

DDL statements define or modify the database structure itself. They appeared early in SQL to let developers create, evolve, or tear down schemas in a declarative way.

**Core Rules & Properties**

* **Autocommit**: DDL in MySQL (InnoDB) issues an implicit commit before and after the statement.
* **Syntax**:
  + CREATE [OR REPLACE] OBJECT\_TYPE name (…)
  + DROP OBJECT\_TYPE name
  + ALTER OBJECT\_TYPE name …
  + RENAME TABLE old TO new
* **Limitations**: Certain alterations require table copy or rebuild.

**When & Where to Use**

* **Initial schema design**: define tables, indexes, views.
* **Schema migrations**: evolving column definitions, partitioning strategies.
* **Cleanup**: remove obsolete tables or indexes.

**Real‑World Examples**

1. **Creating a new reporting table**

CREATE TABLE monthly\_sales\_summary (

report\_month DATE NOT NULL,

total\_revenue DECIMAL(12,2),

PRIMARY KEY (report\_month)

) ENGINE=InnoDB;

*Use case*: After deploying an OLTP system, you create a summary table to aggregate daily transactions into monthly reports.

1. **Altering a live table to support soft deletes**

ALTER TABLE users

ADD COLUMN is\_deleted TINYINT(1) NOT NULL DEFAULT 0,

ADD INDEX idx\_is\_deleted (is\_deleted);

*Use case*: Rather than dropping user rows permanently, flag them and index the flag for faster filtering in application queries.

1. **Dropping an unused staging table**

DROP TABLE IF EXISTS temp\_data\_load;

*Use case*: After ETL completes and data is moved into production tables, clean up staging to free disk and avoid clutter.

1. **Renaming for clarity**

RENAME TABLE orders\_archive TO orders\_2024\_archive;

*Use case*: At year‑end, rename the archive table to reflect its contents, making maintenance and backups clearer.

**2. DML (Data Manipulation Language)**

**What & Why it Came**

DML statements handle the actual data within tables—adding, changing, or removing rows. They power day‑to‑day CRUD (Create, Read, Update, Delete) operations.

**Core Rules & Properties**

* **Transactional**: when using InnoDB, DML can be rolled back unless autocommit is on.
* **Syntax**:
  + INSERT INTO … VALUES (…) or INSERT … SELECT …
  + UPDATE … SET … [WHERE …]
  + DELETE FROM … [WHERE …]
* **Batch Operations**: support multi‑row inserts and updates.

**When & Where to Use**

* **User actions**: registering accounts, posting comments.
* **Automated processes**: nightly batch updates, data archiving.
* **Data corrections**: fixing bad data, removing duplicates.

**Real‑World Examples**

1. **Inserting new customer sign‑ups**

INSERT INTO customers (first\_name, last\_name, email, signup\_date)

VALUES ('Aisha', 'Khan', 'aisha@example.com', NOW());

*Use case*: Each time a visitor registers, the application issues this insert.

1. **Bulk‑loading product catalog**

INSERT INTO products (sku, name, price)

SELECT sku, name, price

FROM staging\_products

WHERE import\_flag = 1;

*Use case*: After validating a CSV upload, you bulk‑insert approved rows from a staging table.

1. **Updating shipping status in orders**

UPDATE orders

SET status = 'shipped', shipped\_date = NOW()

WHERE order\_id IN (1234, 1235, 1236);

*Use case*: When the warehouse picks and ships items, it marks those orders as shipped.

1. **Deleting expired sessions**

DELETE FROM user\_sessions

WHERE last\_activity < NOW() - INTERVAL 30 DAY;

*Use case*: A cleanup cron job removes old session rows to keep the sessions table small.

**3. DQL (Data Query Language)**

**What & Why it Came**

The SELECT statement is the heart of SQL, introduced to query and project data from one or more tables. It lets you filter, aggregate, and shape result sets.

**Core Rules & Properties**

* **Structure**:

SELECT [DISTINCT] columns

FROM tables

[JOIN …]

[WHERE …]

[GROUP BY …]

[HAVING …]

[ORDER BY …]

[LIMIT …];

* **Optimization**: relies on indexes, statistics; EXPLAIN reveals execution plans.
* **NULL semantics**: three‑valued logic in predicates.

**When & Where to Use**

* **Reporting & analytics**: dashboards, BI tools.
* **Application reads**: fetching user profiles, product listings.
* **Data exploration**: ad‑hoc analysis by DBAs.

**Real‑World Examples**

1. **Fetching active products in a category**

SELECT p.product\_id, p.name, p.price

FROM products p

JOIN categories c ON p.category\_id = c.category\_id

WHERE c.name = 'Electronics' AND p.stock > 0

ORDER BY p.name ASC;

*Use case*: Show customers all in‑stock electronics sorted by name.

1. **Monthly revenue report**

SELECT DATE\_FORMAT(o.order\_date, '%Y-%m') AS month,

COUNT(\*) AS orders\_count,

SUM(o.total\_amount) AS total\_revenue

FROM orders o

WHERE o.status = 'completed'

AND o.order\_date BETWEEN '2025-01-01' AND '2025-03-31'

GROUP BY month

HAVING total\_revenue > 10000

ORDER BY month;

*Use case*: Finance team reviews high‑performing months in Q1 of 2025.

1. **Paginated user list with search**

SELECT user\_id, username, email

FROM users

WHERE username LIKE 'a%'

ORDER BY signup\_date DESC

LIMIT 20 OFFSET 40;

*Use case*: Admin UI shows page 3 of users whose names start with “a.”

**4. DCL (Data Control Language)**

**What & Why it Came**

DCL statements manage security and permissions in the database—who can do what on which objects.

**Core Rules & Properties**

* **Syntax**:
  + GRANT privileges ON object TO user[@host] [WITH GRANT OPTION];
  + REVOKE privileges ON object FROM user[@host];
* **Privilege Scope**: global, database‑level, table‑level, column‑level.
* **Privilege Types**: SELECT, INSERT, UPDATE, DELETE, EXECUTE, etc.

**When & Where to Use**

* **Hardening**: lock down production servers by granting least privilege.
* **Onboarding**: create new application‑user accounts with only required rights.
* **Auditing & Compliance**: revoke privileges when roles change.

**Real‑World Examples**

1. **Grant read‑only access to analytics team**

CREATE USER 'reporting'@'%' IDENTIFIED BY 's3cr3t!';

GRANT SELECT ON ecommerce.\* TO 'reporting'@'%';

*Use case*: Analysts connect with this user to run reports but cannot modify data.

1. **Revoke delete rights from junior staff**

REVOKE DELETE ON support.tickets FROM 'junior\_agent'@'internal-host';

*Use case*: Prevent accidental ticket deletions by less experienced support agents.

1. **Grant stored‑procedure execution**

GRANT EXECUTE ON PROCEDURE ecommerce.calculate\_discount TO 'app\_user'@'app-server';

*Use case*: Microservice only needs to call a billing procedure, not view underlying tables.

**5. TCL (Transaction Control Language)**

**What & Why it Came**

TCL statements control transactions—grouping multiple DML operations into atomic units so that either all succeed or none do.

**Core Rules & Properties**

* **Syntax**:
  + START TRANSACTION or BEGIN
  + SAVEPOINT name
  + ROLLBACK [TO SAVEPOINT name | ALL]
  + COMMIT
  + SET TRANSACTION [transaction\_characteristics];
* **Isolation Levels** (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE) set via SET TRANSACTION ISOLATION LEVEL ….
* **Autocommit** default can be toggled: SET autocommit = 0;.

**When & Where to Use**

* **Multi‑step updates**: e.g., transferring funds between accounts.
* **Batch processes**: ensure partial failures don’t corrupt data.
* **Complex workflows**: with intermediate savepoints for partial rollbacks.

**Real‑World Examples**

1. **Money transfer between bank accounts**

START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE account\_id = 1001;

UPDATE accounts SET balance = balance + 500 WHERE account\_id = 2002;

COMMIT;

*Use case*: Both debit and credit must succeed, or neither change should persist.

1. **E‑commerce order placement with savepoints**

START TRANSACTION;

SAVEPOINT sp\_order;

INSERT INTO orders (…) VALUES (…);

SAVEPOINT sp\_items;

INSERT INTO order\_items (…) VALUES (…);

-- Suppose stock check fails

ROLLBACK TO sp\_items;

-- Correct item quantities, then

INSERT INTO order\_items (…) VALUES (…);

COMMIT;

*Use case*: Allows partial rollback of only the items insert if validation fails, without losing the order header.

1. **Changing isolation level for analytics**

SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

START TRANSACTION;

-- run large reporting queries without blocking OLTP writes

COMMIT;

*Use case*: Data‑warehouse queries see only committed data, avoiding dirty reads but allowing concurrent writes.

**1. The WHERE Clause**

**What & Why it Came**

* **Purpose**: Introduced in early SQL to filter rows based on boolean predicates.
* **Why**: Without WHERE, every DML or DQL operation would affect or return all rows—inefficient and often incorrect.

**Rules & Syntax**

1. Always follows the FROM (and any JOIN) clause, before GROUP BY, HAVING, ORDER BY, or LIMIT.
2. Expressions evaluate using three‑valued logic (TRUE, FALSE, UNKNOWN), so rows with NULL in a predicate evaluate as UNKNOWN and are excluded unless explicitly tested with IS NULL.
3. Combine conditions with AND, OR, and negate with NOT.
4. Supports comparison (=, <>, <, >, <=, >=), pattern matching (LIKE, REGEXP), range tests (BETWEEN … AND …), set membership (IN (…)), and NULL tests (IS NULL, IS NOT NULL).

**When & Where to Use**

* **Data retrieval**: fetch only the rows you need (e.g. active users, this month’s orders).
* **Data modification**: limit UPDATE or DELETE to specific records.
* **Report filtering**: slice aggregations to a subset (e.g. high‑value customers).

**Real‑World Examples**

1. **Range filter on orders**

-- All orders between March 1 and March 31, 2025

SELECT order\_id, order\_date, total\_amount

FROM orders

WHERE order\_date BETWEEN '2025-03-01' AND '2025-03-31';

1. **Pattern match on usernames**

-- Usernames starting with 'dev\_' but not ending in a digit

SELECT user\_id, username

FROM users

WHERE username LIKE 'dev\\_%'

AND username NOT REGEXP '[0-9]$';

1. **Combining NULL and Boolean logic**

-- Products that are either discontinued OR have no stock recorded

SELECT product\_id, name

FROM products

WHERE is\_discontinued = 1

OR stock IS NULL;

1. **Filtering with a subquery**

-- Customers who’ve never placed an order

SELECT c.customer\_id, c.name

FROM customers c

WHERE c.customer\_id NOT IN (

SELECT DISTINCT customer\_id FROM orders

);

**2. The DISTINCT Keyword**

**What & Why it Came**

* **Purpose**: Remove duplicate rows from a query’s result set.
* **Why**: Early SQL returned all matching rows, leading to redundant data when joining or selecting non‑key columns.

**Rules & Syntax**

1. Placed immediately after SELECT:

SELECT DISTINCT column1, column2, … FROM …

1. Considers the **entire** list of selected columns when deduplicating: two rows must match on *all* selected columns to collapse into one.
2. Can be combined with COUNT() to count unique values:

SELECT COUNT(DISTINCT column) FROM …

1. Slight performance overhead—MySQL may use a temporary table to eliminate duplicates.

**When & Where to Use**

* **Reporting**: list all distinct regions, categories, or tags.
* **Data cleansing**: identify unique values in a noisy dataset.
* **Summary queries**: combine with aggregates to count unique items.

**Real‑World Examples**

1. **List all countries with customers**

SELECT DISTINCT country

FROM customers;

1. **Unique combinations of product category and manufacturer**

SELECT DISTINCT p.category\_id, m.name AS manufacturer

FROM products p

JOIN manufacturers m ON p.manufacturer\_id = m.manufacturer\_id;

1. **Count of distinct email domains**

SELECT COUNT(DISTINCT SUBSTRING\_INDEX(email, '@', -1)) AS unique\_domains

FROM users;

1. **Distinct tags on active blog posts**

SELECT DISTINCT t.tag\_name

FROM post\_tags pt

JOIN tags t ON pt.tag\_id = t.tag\_id

WHERE pt.post\_id IN (

SELECT post\_id

FROM posts

WHERE status = 'published'

);

**3. Practical Examples of Common MySQL Commands**

Below is a curated set of real‑world snippets covering administration, DDL, DML, DQL, DCL, and TCL commands.

| **Category** | **Command & Example** |
| --- | --- |
| **Admin** | **Show all databases** SHOW DATABASES;  **Switch to a database** USE ecommerce; |
| **DDL** | **Create a new database** CREATE DATABASE analytics CHARSET=utf8mb4;  **Create a table** sql<br>CREATE TABLE products ( <br> product\_id INT AUTO\_INCREMENT PRIMARY KEY,<br> name VARCHAR(100) NOT NULL,<br> price DECIMAL(8,2) NOT NULL<br>) ENGINE=InnoDB;<br> |
|  | **Alter a table** sql<br>ALTER TABLE products<br> ADD COLUMN sku VARCHAR(50) UNIQUE;<br>  **Drop a table** DROP TABLE IF EXISTS temp\_logs; |
| **DML** | **Insert a single row** sql<br>INSERT INTO customers (name, email, signup\_date)<br>VALUES ('Ravi Patel','ravi@example.com', NOW());<br>  **Bulk insert from staging** sql<br>INSERT INTO products (sku, name, price)<br>SELECT sku, name, price FROM staging\_products WHERE valid = 1;<br> |
|  | **Update rows** UPDATE orders SET status='shipped', shipped\_date=NOW() WHERE order\_id=5012;  **Delete rows** DELETE FROM sessions WHERE last\_activity < NOW() - INTERVAL 90 DAY; |
| **DQL** | **Simple select with WHERE & ORDER** sql<br>SELECT user\_id, username, email<br>FROM users<br>WHERE status='active'<br>ORDER BY signup\_date DESC<br>LIMIT 10;<br>  **Join across two tables** sql<br>SELECT o.order\_id, c.name, o.total\_amount<br>FROM orders o<br>JOIN customers c ON o.customer\_id = c.customer\_id<br>WHERE o.total\_amount > 100;<br> |
|  | **Group‑by aggregation** sql<br>SELECT category\_id, COUNT(\*) AS num\_products, AVG(price) AS avg\_price<br>FROM products<br>GROUP BY category\_id<br>HAVING AVG(price) > 20;<br> |
| **DCL** | **Create an application user** CREATE USER 'app\_user'@'%' IDENTIFIED BY 'P@ssw0rd!';  **Grant privileges** GRANT SELECT, INSERT, UPDATE ON ecommerce.\* TO 'app\_user'@'%'; |
|  | **Revoke privileges** REVOKE DELETE ON ecommerce.orders FROM 'app\_user'@'%'; |
| **TCL** | **Start a transaction** START TRANSACTION; … (multiple DML statements) … COMMIT;  **Rollback on error** sql<br>SAVEPOINT before\_items;<br>-- insert order\_items…;<br>ROLLBACK TO before\_items;<br>COMMIT;<br> |

**1. What is MySQL?**

A. A programming language  
B. A web server  
C. An open-source relational database management system  
D. A file transfer protocol  
  
**Answer: C**  
**Explanation:** MySQL is a free and open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) to manage and manipulate data.

**2. MySQL is developed by:**

A. Microsoft  
B. Oracle Corporation  
C. IBM  
D. Red Hat  
  
**Answer: B**  
**Explanation:** MySQL was originally developed by MySQL AB, later acquired by Sun Microsystems, which was acquired by Oracle Corporation.

**3. Which of the following is used to interact with MySQL?**

A. FTP  
B. HTML  
C. SQL  
D. XML  
  
**Answer: C**  
**Explanation:** SQL (Structured Query Language) is the standard language used to communicate with relational databases like MySQL.

**4. What does SQL stand for?**

A. Simple Query Language  
B. Structured Query Language  
C. Styled Query Language  
D. Synchronized Query Language  
  
**Answer: B**  
**Explanation:** SQL stands for Structured Query Language, which is used for querying and managing relational databases.

**5. Which command is used to create a database in MySQL?**

A. MAKE DATABASE  
B. CREATE DATABASE  
C. NEW DATABASE  
D. INIT DATABASE  
  
**Answer: B**  
**Explanation:** CREATE DATABASE is the correct SQL syntax to create a new database in MySQL.

**6. Which statement is used to remove a database in MySQL?**

A. REMOVE DATABASE  
B. DROP DATABASE  
C. DELETE DATABASE  
D. CLEAR DATABASE  
  
**Answer: B**  
**Explanation:** The DROP DATABASE command permanently deletes a database and all its contents.

**7. Which command is used to view all databases?**

A. SHOW ALL  
B. SHOW DATABASES  
C. DISPLAY DATABASES  
D. LIST DATABASES  
  
**Answer: B**  
**Explanation:** SHOW DATABASES displays a list of all databases available on the MySQL server.

**8. What is the default port number for MySQL?**

A. 1433  
B. 3306  
C. 1521  
D. 5432  
  
**Answer: B**  
**Explanation:** MySQL uses port 3306 by default to listen for incoming client connections.

**9. Which of the following is a valid data type in MySQL?**

A. VARCHAR  
B. TEXTUAL  
C. CHARSTR  
D. STR  
  
**Answer: A**  
**Explanation:** VARCHAR is a valid data type used to store variable-length character strings.

**10. What is the purpose of the PRIMARY KEY?**

A. To store large values  
B. To uniquely identify each row in a table  
C. To create a backup  
D. To encrypt the data  
  
**Answer: B**  
**Explanation:** A PRIMARY KEY uniquely identifies each row in a table and ensures no duplicate or null values in the key column.

**11. Which clause is used to filter records in a SELECT statement?**

A. HAVING  
B. FROM  
C. WHERE  
D. ORDER  
  
**Answer: C**  
**Explanation:** The WHERE clause is used to filter rows based on a condition.

**12. What does the SELECT \* command do?**

A. Selects all columns from a table  
B. Deletes all columns  
C. Selects specific rows only  
D. Selects only primary keys  
  
**Answer: A**  
**Explanation:** SELECT \* retrieves all columns for each row from a specified table.

**13. What is used to sort the result set?**

A. ORDER BY  
B. SORT  
C. GROUP BY  
D. SELECT BY  
  
**Answer: A**  
**Explanation:** ORDER BY is used to sort result sets in ascending (default) or descending order based on one or more columns.

**14. Which command is used to insert data into a table?**

A. ADD INTO  
B. INSERT INTO  
C. APPEND INTO  
D. UPDATE INTO  
  
**Answer: B**  
**Explanation:** INSERT INTO is the standard SQL command used to add new rows into a table.

**15. Which SQL clause is used to group rows that have the same values?**

A. SORT BY  
B. GROUP BY  
C. COMBINE BY  
D. MERGE BY  
  
**Answer: B**  
**Explanation:** GROUP BY is used with aggregate functions to group rows that have the same values in specified columns.

**16. Which command is used to change data in an existing row?**

A. CHANGE  
B. UPDATE  
C. MODIFY  
D. SET  
  
**Answer: B**  
**Explanation:** UPDATE modifies existing records in a table. It's used with the SET clause to specify new values.

**17. Which MySQL function returns the current date and time?**

A. NOWDATE()  
B. NOW()  
C. CURRENT()  
D. GETDATE()  
  
**Answer: B**  
**Explanation:** NOW() is a built-in MySQL function that returns the current date and time.

**18. What does the AUTO\_INCREMENT keyword do?**

A. Automatically increases a numeric value for new rows  
B. Increments all rows  
C. Adds multiple rows  
D. Auto-saves data  
  
**Answer: A**  
**Explanation:** AUTO\_INCREMENT is used on a column (usually primary key) to generate unique values automatically as new records are added.

**19. Which command deletes all rows from a table but not the table itself?**

A. DELETE FROM table\_name  
B. DROP TABLE  
C. REMOVE TABLE  
D. DELETE DATABASE  
  
**Answer: A**  
**Explanation:** DELETE FROM table\_name removes all data but retains the table structure and schema.

**20. Which of these is true about MySQL?**

A. MySQL does not support transactions  
B. MySQL is case-insensitive for table names on Windows  
C. MySQL supports NoSQL features only  
D. MySQL is not open-source  
  
**Answer: B**  
**Explanation:** On Windows, MySQL table names are case-insensitive due to the case-insensitive nature of the Windows file system. On Unix-based systems, table names are case-sensitive.

## **1. Introduction to JDBC Connectivity**

**What & Why It Came**

* **JDBC (Java Database Connectivity)** is a standard Java API introduced by Sun Microsystems in 1997 as part of JDK 1.1, to give Java applications a uniform, vendor‑agnostic way to talk to relational databases (including MySQL).
* **Motivation**: before JDBC, each database vendor provided its own proprietary Java interfaces. JDBC brought a common set of interfaces (java.sql) so that switching databases (or supporting multiple back‑ends) required minimal code changes.

**Key Components**

| **Component** | **Role** |
| --- | --- |
| **JDBC API** | Interfaces in java.sql (e.g. Connection, Statement, ResultSet) |
| **JDBC Driver** | Vendor‑supplied implementation of JDBC API; MySQL provides the “Connector/J” driver JAR. |
| **DriverManager** | Legacy registry for loading/locating drivers and obtaining Connection objects. |
| **DataSource** | Preferred from JDBC 2.0 on for obtaining connections—supports connection pooling & JNDI. |

**2. Core Rules & Best Practices**

1. **Loading the Driver**

// Legacy: explicitly load the driver

Class.forName("com.mysql.cj.jdbc.Driver");

// Modern: Connector/J auto‑registers via Java SPI—no explicit load needed.

1. **Connection URL Syntax**

jdbc:mysql://<host>:<port>/<database>

?serverTimezone=UTC

&useSSL=false

&allowPublicKeyRetrieval=true

1. **Resource Management**
   * Always close ResultSet, Statement/PreparedStatement, and Connection in **reverse** order of creation.
   * Use **try‑with‑resources** (Java 7+):

try (Connection conn = ds.getConnection();

PreparedStatement ps = conn.prepareStatement(sql);

ResultSet rs = ps.executeQuery()) {

// … work with rs …

} // auto‑closes rs, ps, conn

1. **SQL Injection Prevention**
   * **Never** concatenate user input into SQL strings.
   * Use PreparedStatement:

String sql = "SELECT \* FROM users WHERE email = ?";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setString(1, userInputEmail);

ResultSet rs = ps.executeQuery();

…

}

1. **Transaction Management**
   * By default, JDBC connections are in **autocommit** mode. For multi‑statement atomicity:

conn.setAutoCommit(false);

try {

// multiple updates/inserts

conn.commit();

} catch(SQLException e) {

conn.rollback();

throw e;

} finally {

conn.setAutoCommit(true);

}

1. **Connection Pooling**
   * For high‑throughput apps, avoid the overhead of opening/closing raw connections.
   * Use a DataSource implementation (HikariCP, Apache DBCP) via JNDI in your container or directly in code.

**3. When & Where to Use JDBC**

* **Web Applications** (Servlets, Spring MVC/Boot, Jakarta EE)  
  → powering dynamic content, RESTful APIs, session persistence.
* **Desktop/Standalone Java Apps**  
  → inventory management, reporting tools, ETL utilities.
* **Batch Jobs & Data Migrations**  
  → nightly data imports/exports, cross‑database replication scripts.
* **Microservices**  
  → lightweight services using plain JDBC or via higher‑level libraries (e.g. MyBatis, jOOQ).

**4. JDBC Architecture & Workflow**

1. **Load Driver**
   * Either explicitly (Class.forName) or implicitly via SPI.
2. **Obtain Connection**
   * DriverManager.getConnection(url, user, pass) **or**
   * DataSource ds = …; ds.getConnection();
3. **Create Statement**
   * Statement for static SQL
   * PreparedStatement for parameterized SQL
   * CallableStatement for stored procedures
4. **Execute SQL**
   * executeQuery() → returns ResultSet
   * executeUpdate() → returns affected row count
   * execute() → handles unknown result formats
5. **Process Results**
   * Iterate ResultSet via rs.next(), retrieve via getXXX()
6. **Close Resources**
   * In reverse: ResultSet, Statement, Connection

**5. Real‑World Examples**

**Example 1: User Authentication in a Web App**

A servlet validates credentials against a MySQL users table.

// 1. get connection (from DataSource configured in Tomcat)

try (Connection conn = dataSource.getConnection();

PreparedStatement ps = conn.prepareStatement(

"SELECT id, password\_hash FROM users WHERE username = ?")) {

ps.setString(1, usernameInput);

try (ResultSet rs = ps.executeQuery()) {

if (rs.next()) {

String storedHash = rs.getString("password\_hash");

if (PasswordUtil.matches(passwordInput, storedHash)) {

// successful login—establish session

} else {

// invalid credentials

}

} else {

// user not found

}

}

}

* **Why JDBC**: direct, lightweight access to user data; works inside any servlet container.
* **Rules Used**: PreparedStatement for safety; try‑with‑resources for proper cleanup.

**Example 2: Batch Insertion of Sensor Data**

A standalone Java utility loads thousands of IoT sensor readings into MySQL nightly.

String sql = "INSERT INTO sensor\_readings(sensor\_id, reading\_ts, value) VALUES (?, ?, ?)";

try (Connection conn = DriverManager.getConnection(url, user, pass);

PreparedStatement ps = conn.prepareStatement(sql)) {

conn.setAutoCommit(false);

for (SensorReading r : readingsList) {

ps.setInt(1, r.getSensorId());

ps.setTimestamp(2, Timestamp.from(r.getTimestamp()));

ps.setDouble(3, r.getValue());

ps.addBatch();

}

int[] counts = ps.executeBatch();

conn.commit();

}

* **Why JDBC**: full control over batching behavior and transaction boundaries.
* **Rules Used**: batch updates for performance; explicit transaction demarcation.

**Example 3: Calling a Stored Procedure for Reporting**

Use CallableStatement to invoke a MySQL stored routine that aggregates sales.

try (Connection conn = ds.getConnection();

CallableStatement cstmt = conn.prepareCall("{CALL sp\_monthly\_sales(?, ?)}")) {

cstmt.setInt(1, year);

cstmt.setInt(2, month);

boolean hasResultSet = cstmt.execute();

if (hasResultSet) {

try (ResultSet rs = cstmt.getResultSet()) {

while (rs.next()) {

System.out.printf("Category %s: %.2f%n",

rs.getString("category"), rs.getDouble("total"));

}

}

}

}

* **Why JDBC**: seamless integration with vendor‑specific features (stored procedures).
* **Rules Used**: use {CALL …} syntax; process returned ResultSet.

**Key Takeaways**

* **JDBC** gives Java programs a unified API to work with MySQL (and other RDBMS).
* Follow its **rules** around drivers, resource cleanup, SQL injection prevention, and transaction handling to build robust, performant data‑driven applications.
* **Use cases** span from simple command‑line utilities to large‑scale web services and batch processes.

import java.sql.\*;  
  
public class JdbcCRUDExample {  
 *// MySQL connection details* static final String *DB\_URL* = "jdbc:mysql://localhost:3306/world";  
 static final String *USER* = "root";  
 static final String PASS = " your\_password ";  
  
 public static void main(String[] args) {  
 try (Connection conn = DriverManager.*getConnection*(*DB\_URL*, *USER*, *PASS*)) {  
 *// 1. Create Table  
 createTable*(conn);  
  
 *// 2. Insert Data  
 insertStudent*(conn, 1, "Alice", "Computer Science");  
 *insertStudent*(conn, 2, "Bob", "Mechanical");  
  
 *// 3. Read Data  
 readStudents*(conn);  
  
 *// 4. Update Data  
 updateStudent*(conn, 1, "Electronics");  
  
 *// 5. Delete Data  
 deleteStudent*(conn, 2);  
  
 *// 6. Final Read  
 readStudents*(conn);  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
  
 static void createTable(Connection conn) throws SQLException {  
 String sql = "CREATE TABLE IF NOT EXISTS students (" +  
 "id INT PRIMARY KEY," +  
 "name VARCHAR(100)," +  
 "department VARCHAR(100))";  
 try (Statement stmt = conn.createStatement()) {  
 stmt.execute(sql);  
 System.*out*.println("✅ Table created or already exists.");  
 }  
 }  
  
 static void insertStudent(Connection conn, int id, String name, String department) throws SQLException {  
 String sql = "INSERT INTO students (id, name, department) VALUES (?, ?, ?)";  
 try (PreparedStatement pstmt = conn.prepareStatement(sql)) {  
 pstmt.setInt(1, id);  
 pstmt.setString(2, name);  
 pstmt.setString(3, department);  
 int rows = pstmt.executeUpdate();  
 System.*out*.println("🟢 Inserted " + rows + " student(s).");  
 }  
 }  
  
 static void readStudents(Connection conn) throws SQLException {  
 String sql = "SELECT \* FROM students";  
 try (Statement stmt = conn.createStatement();  
 ResultSet rs = stmt.executeQuery(sql)) {  
 System.*out*.println("📋 Students Table:");  
 while (rs.next()) {  
 System.*out*.println(rs.getInt("id") + " - " + rs.getString("name") + " - " + rs.getString("department"));  
 }  
 }  
 }  
  
 static void updateStudent(Connection conn, int id, String newDepartment) throws SQLException {  
 String sql = "UPDATE students SET department = ? WHERE id = ?";  
 try (PreparedStatement pstmt = conn.prepareStatement(sql)) {  
 pstmt.setString(1, newDepartment);  
 pstmt.setInt(2, id);  
 int rows = pstmt.executeUpdate();  
 System.*out*.println("🟡 Updated " + rows + " student(s).");  
 }  
 }  
  
 static void deleteStudent(Connection conn, int id) throws SQLException {  
 String sql = "DELETE FROM students WHERE id = ?";  
 try (PreparedStatement pstmt = conn.prepareStatement(sql)) {  
 pstmt.setInt(1, id);  
 int rows = pstmt.executeUpdate();  
 System.*out*.println("🔴 Deleted " + rows + " student(s).");  
 }  
 }  
}

**📝 Setup Notes**

1. **Database Setup**
   * Run this once in MySQL:

CREATE DATABASE testdb;

1. **Add MySQL Connector/J to your project**
   * If you're using an IDE like Eclipse or IntelliJ, add the JAR via project libraries.
   * Or use Maven dependency:

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<version>8.3.0</version>

</dependency>

**🔒 Replace This Before Running**

static final String PASS = "your\_password";

**Add MySQL JDBC Driver to Your Project**

**🔧 Option 1: If you're using Eclipse / IntelliJ**

1. **Download JDBC JAR** (if you don’t already have it):  
   👉 Download MySQL Connector/J  
   [MySQL :: Download Connector/J](https://dev.mysql.com/downloads/connector/j/)
2. **Add JAR to your project**:
   * **Eclipse**:
     + Right-click project → Build Path → Configure Build Path
     + Go to **Libraries** tab → Add External JARs → Select mysql-connector-j-<version>.jar
   * **IntelliJ**:
     + File → Project Structure → Modules → Dependencies → + → JARs or directories

**✅ JDBC Basics Quiz (MySQL + Java)**

**1. What does JDBC stand for?**  
A) Java Database Connector  
B) Java Data Binary Connector  
C) Java Database Connectivity  
D) Java Driver for Binary Communication

**Answer:** ✅ C) Java Database Connectivity  
**Explanation:** JDBC is the Java API for connecting and executing queries on databases.

**2. Which JDBC class is used to execute SQL statements?**  
A) ResultSet  
B) PreparedStatement  
C) Statement  
D) Connection

**Answer:** ✅ C) Statement  
**Explanation:** Statement is used to execute static SQL queries.

**3. What is the correct JDBC URL format for connecting to a MySQL database?**  
A) jdbc:mysql://host/database  
B) mysql:jdbc://host:port/database  
C) jdbc:mysql://host:port/database  
D) jdbc:mysql::host:port:database

**Answer:** ✅ C) jdbc:mysql://host:port/database  
**Explanation:** This is the standard format used by MySQL JDBC driver.

**4. What package do you import to use JDBC?**  
A) java.sql.\*  
B) javax.sql.\*  
C) org.mysql.jdbc.\*  
D) java.db.\*

**Answer:** ✅ A) java.sql.\*  
**Explanation:** All JDBC interfaces like Connection, Statement, ResultSet are in java.sql.

**5. What is the purpose of DriverManager.getConnection()?**  
A) To load JDBC driver  
B) To register the database  
C) To execute SQL queries  
D) To establish a connection to the database

**Answer:** ✅ D) To establish a connection to the database  
**Explanation:** It returns a Connection object for the specified URL, user, and password.

**6. Which interface is used to retrieve data from a SQL SELECT query?**  
A) ResultList  
B) ResultRow  
C) ResultSet  
D) ResultObject

**Answer:** ✅ C) ResultSet  
**Explanation:** ResultSet holds data retrieved from SELECT queries.

**7. Which method is used to load the MySQL JDBC driver class?**  
A) Class.call("com.mysql.jdbc.Driver")  
B) Driver.load("com.mysql.Driver")  
C) Class.forName("com.mysql.cj.jdbc.Driver")  
D) JDBC.load("mysql.driver")

**Answer:** ✅ C) Class.forName("com.mysql.cj.jdbc.Driver")  
**Explanation:** This loads the driver class into memory (older Java required it explicitly).

**8. What does PreparedStatement help prevent?**  
A) Code duplication  
B) Memory leaks  
C) SQL Injection  
D) Syntax errors

**Answer:** ✅ C) SQL Injection  
**Explanation:** PreparedStatement safely handles user inputs using placeholders (?).

**9. Which method is used to execute a SELECT query with Statement or PreparedStatement?**  
A) executeQuery()  
B) executeUpdate()  
C) runQuery()  
D) getData()

**Answer:** ✅ A) executeQuery()  
**Explanation:** It runs SQL SELECT statements and returns a ResultSet.

**10. Which method is used to close a database connection in JDBC?**  
A) disconnect()  
B) shutdown()  
C) end()  
D) close()

**Answer:** ✅ D) close()  
**Explanation:** conn.close() safely terminates the connection to the database.