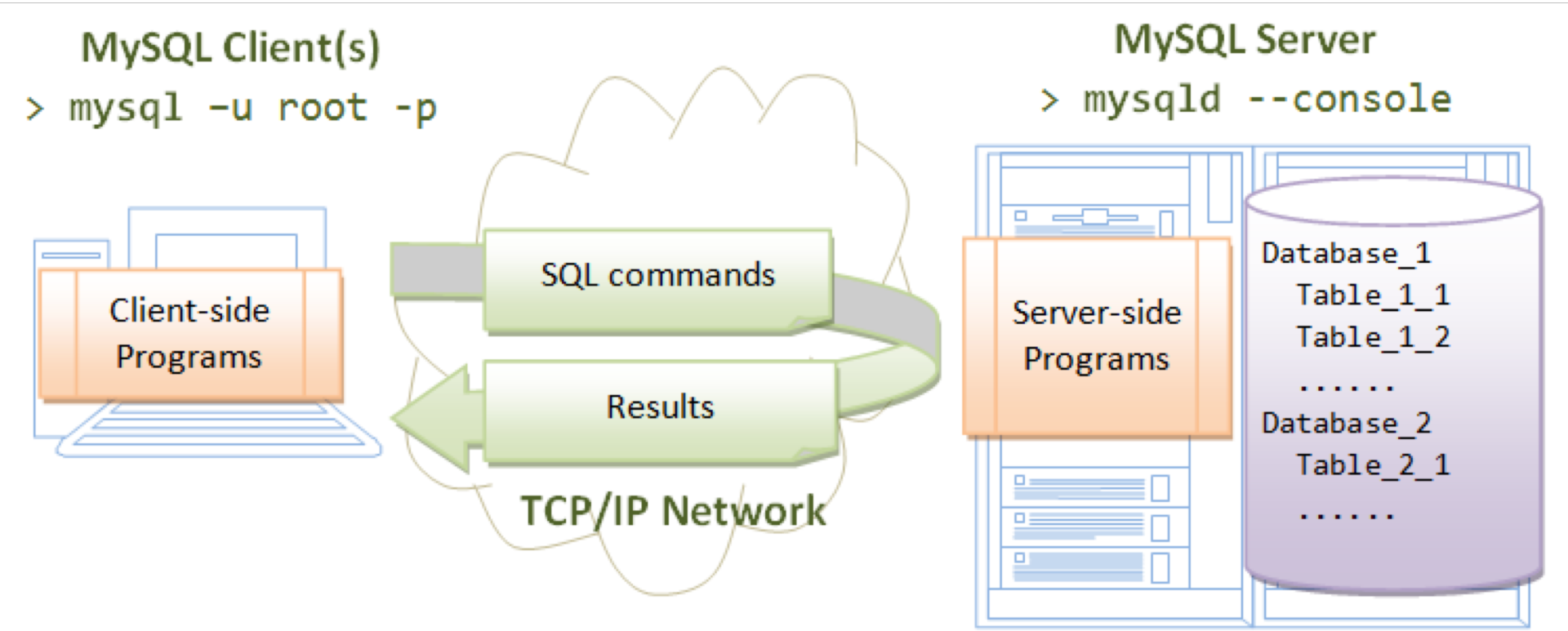
LECTURE 06: JAVA DATABASE CONNECTIVITY (JDBC)

COMP2004J: Databases and Information Systems

Dr. Ruihai Dong (ruihai.dong@ucd.ie)
UCD School of Computer Science
Beijing-Dublin International College

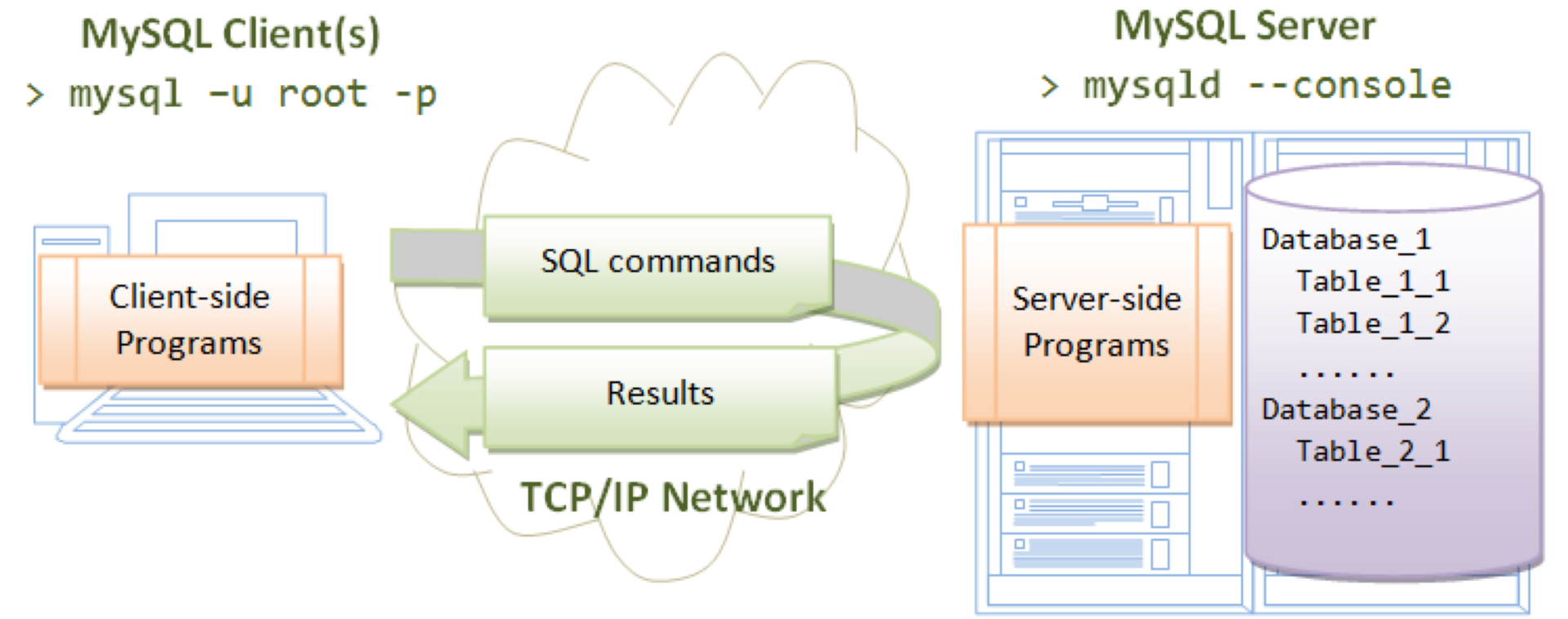
MySQL Architecture

- Like most Relational Database Management Systems (RDBMS), MySQL uses a **client/server** architecture.



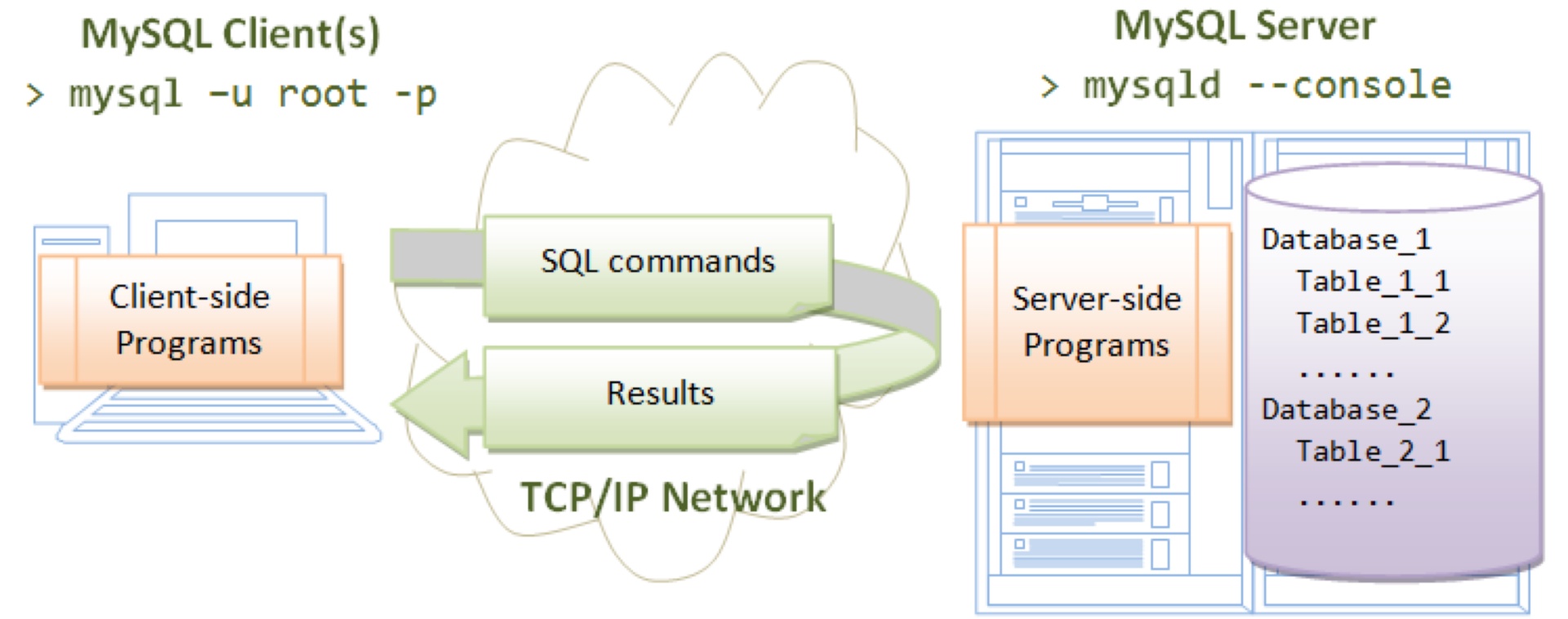
MySQL Architecture

- The database itself is a collection of files that is controlled by the MySQL **daemon**: a non-interactive program that runs in the background.



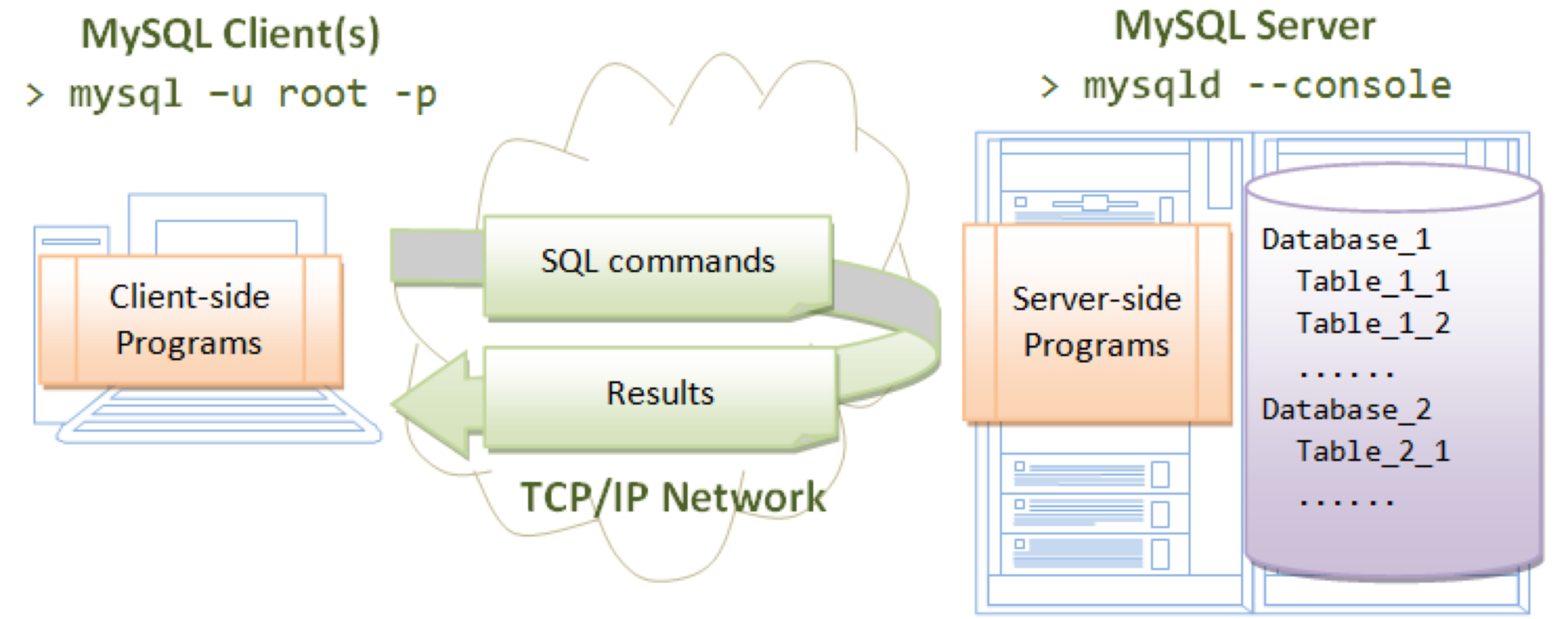
MySQL Architecture

- The MySQL daemon (called `mysqld`) accepts connections from **clients**, processes SQL queries and returns the results.



MySQL Architecture

- The MySQL client we have been using is the default client that MySQL provides (it's called `mysql`). By default it connects to a MySQL daemon on the same computer.



MySQL Architecture

- By using the -h option, the MySQL client can also be used to connect to MySQL servers on other computers.
  - The -h stands for "**host**".

```
mysql -u david -h host.example.com -p
```

- Using the default client is very useful when you need manual, direct access to the database (and when you're learning SQL!).
- In a realistic situation, databases are accessed by **programs**.

What is JDBC?

- A Java-based Application Programming Interface (**API**) to allow connections to relational database management systems, such as MySQL, Oracle, Microsoft SQL Server etc.

- Originally known as the **Java DataBase Connectivity** API.

- Released as part of JDK 1.1 in 1997.

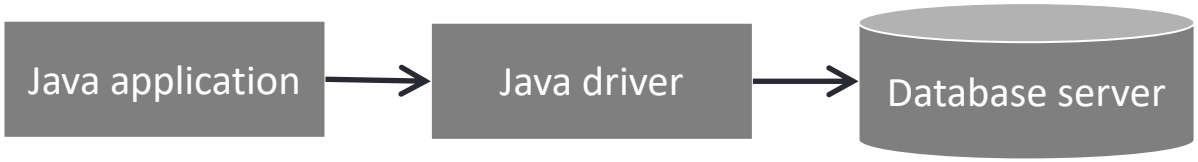
- JDBC is used by Java applications that need to connect with local and remote databases.

JDBC drivers

- Applications that use the JDBC API require drivers.
- JDBC drivers are software libraries that sit between a Java application and a database management system.
- There are four types of drivers.
  - Type 1, Type 2, Type 3, and Type 4.

JDBC drivers – Type 4

- Type 4 driver connects directly to the database server.
- Today, most JDBC drivers are type 4 drivers.



- Written 100% in Java. So portable.
- This type of driver is installed with the application so there are no separate maintenance issues.
- Implemented for a specific database management system. So we need a different driver for each database management system.

MySQL Connector/J driver

- Connector/J is the official JDBC driver for MySQL. Latest version is available to download at the following link: https://dev.mysql.com/downloads/connector/j/
- You will need to download and install the Connector/J driver and add it to the CLASSPATH.
- Detail regarding Connector/J Installation and configuring the CLASSPATH is available at: http://dev.mysql.com/doc/connector-j/en/connector-j-installing.html

SQL statements with JDBC

- In order to process SQL statements with JDBC, the following steps are considered:
  1. Establish a Connection.
  2. Create a Statement.
  3. Execute the Statement.
  4. Process the **ResultSet** Object (if it was a SELECT query)
  5. Close the Connection.

Establish a Connection

- First, we need to establish a connection with MySQL database management system.

- There are two ways to connect: **DriverManager** class and **DataSource** interface.

- In order to connect with MySQL we will use the DriverManager class as it is easier to use.

Establish a Connection

- Following code establishes a connection with MySQL.

```
Connection conn = DriverManager.getConnection(DB_URL,
USERNAME, PASSWORD);
```

- There are three arguments passed to the getConnection method:
  - DB_URL: is the JDBC URL (see next slide).
  - USERNAME: database username.
  - PASSWORD: password for the database username.

Establish a Connection

- JDBC URL contains three main parts, For example:

```
"jdbc:mysql://hostname:port/database_name"
```

- **jdbc:mysql** is the driver name: this doesn't change.
- Replace **hostname** with the IP address or name of the machine that has MySQL server running.
- Replace **port** with the port number on which MySQL server is running (default: 3306)
- Replace **database_name** with the name of the database to which the connection should be made.

Establish a Connection

- How do we find out hostname (or IP) and the port number of MySQL server?
- To find out the hostname:

```
mysql> show variables like 'hostname';
+---------------+--------------------+
| Variable_name | Value              |
+---------------+--------------------+
| hostname      | david-laptop.local |
+---------------+--------------------+
1 row in set (0.00 sec)
```

- To find out the port number:

```
mysql> show variables like 'port';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| port          | 3306  |
+---------------+-------+
1 row in set (0.00 sec)
```

Establish a Connection – Example

- Once we know the hostname and the port we can do:

```
private static final String USER_NAME = "user_name";

private static final String PASSWORD = "password";

private static final String DB_URL = "jdbc:mysql://url-
to-remote-machine:3306/database_name";

Connection conn = DriverManager.getConnection(DB_URL,
USERNAME, PASSWORD);
```

- To connect to MySQL server running on local machine use the **localhost** as the hostname. For example:

```
private static final String DB_URL =
"jdbc:mysql://localhost:3306/database_name”;
```

Creating Statements

- Once the connection is established the next step is to create a statement.
- **Statement** interface is used to create statement. We need a connection object to create a statement.

```
Statement st = conn.createStatement();
```

- A Statement is an interface that represents a SQL statement.

Execute the Statements

- In order to execute a query, we call execute methods from Statement. There are two main methods:
- executeQuery:
  - Used when using SELECT SQL statements.
  - Returns one **ResultSet** object.
- executeUpdate:
  - Used when using INSERT, DELETE, or UPDATE SQL statements.
  - Returns an integer representing the number of rows affected.
- Each method take a query as an argument. For example:

```
ResultSet rs = st.executeQuery("SELECT * from
employee");
```

Processing ResultSet Objects

- **ResultSet** object contains the data returned by a query.
- We can access data in a ResultSet object through a **cursor**.
- A cursor is a pointer that points to one row of data in the ResultSet object. Initially, the cursor is positioned **before the first row**.
- We can call various methods defined in the ResultSet object to move the cursor. For example, **.next()** can be used to move the cursor forward by one row or **.last()** to move the cursor to the last row.
  - Both return a boolean value to say whether it was successful or not (.next() will fail if we're already on the last line, for example).

Processing ResultSet Objects

- We can loop through the ResultSet object and retrieve values of specific columns. For example:

```
ResultSet rs = st.executeQuery("SELECT * from
employee");

while (rs.next()){
    int employeeID = rs.getInt("emp_id");
    String firstName = rs.getString("first_name");
    String lastName = rs.getString("last_name");
    String phone = rs.getString("phone");
    int salary = rs.getInt("salary");

    System.out.println(employeeID + "\t" + firstName +
    "\t" + lastName + "\t" + phone + "\t" + salary);
}
```

Closing Connections

- When we are finished using a Statement, we should close resultset, statement, and database connections to immediately release the resources it is using.
- This can be achieved by calling the close() method. These are done in a reverse order. For example:

```
if (rs != null){
    rs.close();
}
if (st != null) {
    st.close();
}
if (conn != null){
    conn.close();
}
```

Putting it all together

- Lets retrieve all the records from an employee table using the Java application. The employee table has the following data:

```
mysql> select * from employee;
+--------+------------+-----------+------------+--------+---------------+
| emp_id | first_name | last_name | phone      | salary | department_id |
+--------+------------+-----------+------------+--------+---------------+
|      1 | Alice      | Smith     | 0863594873 |  31000 |             1 |
|      2 | Bob        | Doe       | 01594000   |  31700 |          NULL |
|      3 | Chris      | Murphy    | 052513013  |  33100 |             2 |
|      4 | Jack       | OHare     | 028273949  |  35700 |             3 |
|      5 | Stuart     | Ullman    | 037494722  |  35000 |             4 |
|      6 | Larry      | Durkin    | 032345343  |  31000 |             5 |
|      7 | Bill       | Watson    | 038472021  |  33000 |             2 |
|      8 | Wendy      | OHare     | 086374492  |  32500 |             2 |
+--------+------------+-----------+------------+--------+---------------+
8 rows in set (0.00 sec)
```

Putting it all together

- Java SQLExceptions will be thrown if something goes wrong.
  - I won't deal with these today, but in a real program, they should be caught using try/catch blocks.

- Remember the steps:
  1. Establish a Connection.
  2. Create Statements.
  3. Execute the Statements.
  4. Process the ResultSet Object.
  5. Close all Connections.

Executing the Statement – Example

```
Connection conn = DriverManager.getConnection(DB_URL, USER_NAME, PASSWORD);
System.out.println("Connected to MySQL server!");

Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * from employee");

while (rs.next()){
   int employeeID = rs.getInt("emp_id");
   String firstName = rs.getString("first_name");
   String lastName = rs.getString("last_name");
   String phone = rs.getString("phone");
   int salary = rs.getInt("salary");
   System.out.println(employeeID + "\t" + firstName +       "\t" +
                                          lastName + "\t" + phone + "\t" + salary);
}

rs.close();
st.close();
conn.close();
```

Running the Java Application

- If we run the application using the following commands.
  - javac MySQLConnect.java
  - java MySQLConnect
- We get the following output:

```
declans-MacBook-Pro:JDBC dec$ javac MySQLConnect.java
declans-MacBook-Pro:JDBC dec$ java MySQLConnect
Connected to MySQL server!
1       Alice   Smith   0863594873      31000
2       Bob     Doe     01594000        31700
3       Chris   Murphy  052513013       33100
4       jack    OHare   028273949       35700
7       Bill    Watson  038472021       33000
MySQL server connection now closed.
```

PreparedStatements

- Often, we want to run the same query again and again, sometimes with different parameters.

- For this, it is more efficient to prepare the statement beforehand, to get a PreparedStatement.
  - This means that the query is pre-compiled into code that MySQL will understand, so it's not necessary to do this every time the query runs.

  - It has another advantage: if there are parameters that can change, we can set **placeholders** so we can fill them in later.

  - PreparedStatement ps = conn.prepareStatement("SELECT first_name, last_name FROM employee WHERE emp_id = ?;");

  - The ? is a placeholder: we will add the employee's ID later.

PreparedStatements

- We can add **bind parameters** to take the place of placeholders before we execute the query.

- ps.setInt(1, 5); // sets the value for the first placeholder (?) to be 5
  - Also, setString, setDouble, setDate, etc.

- The driver **automatically** puts in the appropriate quotes so we don't have to.
  - It also escapes "special" characters that have a particular meaning in SQL.

  - For example:
    - SELECT first_name FROM employees WHERE last_name = ?;
    - … then ...
      - ps.setString(1, "O'Hare");
    - ... becomes ...
    - SELECT first_name FROM employees WHERE last_name='O\'Hare';

  - **Why is this particularly useful?**

USER INPUT IS

DANGEROUS

User Input is DANGEROUS

- Any time we allow users to enter data into our program, we have **no control** over what they enter.
- A malicious user might try to enter data that will change the meaning of the SQL.
- Example (without a PreparedStatement): assume “uname” and “pass” are strings entered by the user. We use the following string for our Statement:
- ```
  String query = "SELECT username FROM users WHERE username='"
     + uname + "' AND password='" + pass + "';";
  ```
- **(Note:** this is just an example: you should NEVER store passwords in plain text in a database.)
  ```

User Input is DANGEROUS

- `String query = "SELECT username FROM users WHERE username='" + uname + "' AND password='" + pass + "';";`

- What if the user enters the following for the password?
  - `fdsa' OR '1'='1`

- Our query now becomes:

```
SELECT username FROM users WHERE username='david'
   AND password='fdsa' OR '1'='1';
```

- Now, a use can gain access **without knowing a password!**
- This type of security problem is called an **SQL Injection Attack**.

PreparedStatements to avoid injection attacks

- Instead, with a PreparedStatement:
- `String query = "SELECT username FROM users WHERE username=? AND password=?;";`

- … after setting the bind parameters, the query becomes:

- `SELECT username FROM users WHERE username='david' AND password='fdsa\' OR \'1\'=\'1';`

Summary

- JDBC can be used to allow a Java program to access a MySQL database.
- Using Connection, Statement (or PreparedStatement) and ResultSet objects, we can easily run SQL queries on a database and see the results.
- We need to be very careful, especially when accepting input from users, that our code is not subject to SQL Injection Attacks.