Chapter Eight

Introduction to JDBC Programming

Introduction

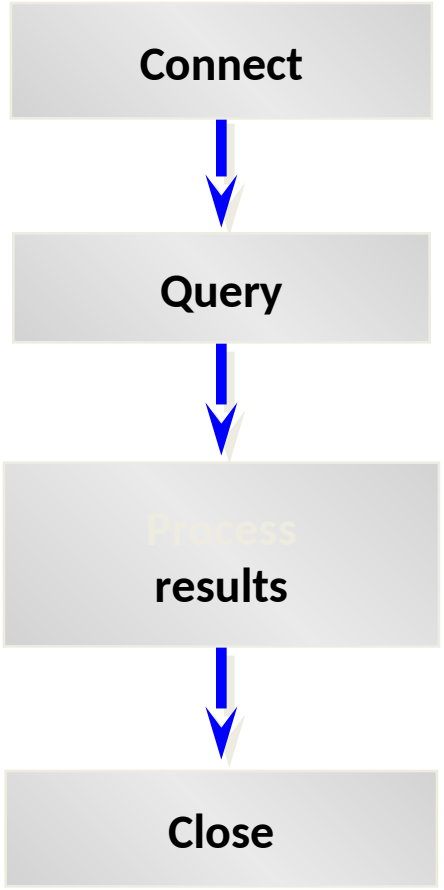
- A great promise of Java has been the ability to build platform-independent client/server database applications.
- This has come to fruition with Java DataBase Connectivity (JDBC).
- One of the major problems with databases has been the feature wars between the database companies.
- There is a “standard” database language, Structured Query Language (SQL-92), but you must usually know which database vendor you’re working with despite the standard.

- JDBC is designed to be platform-independent, so you don’t need to worry about the database you’re using while you’re programming.
- JDBC, like many of the APIs in Java, is designed for simplicity.
- The JDBC classes and interfaces are in the ***java.sql*** package.

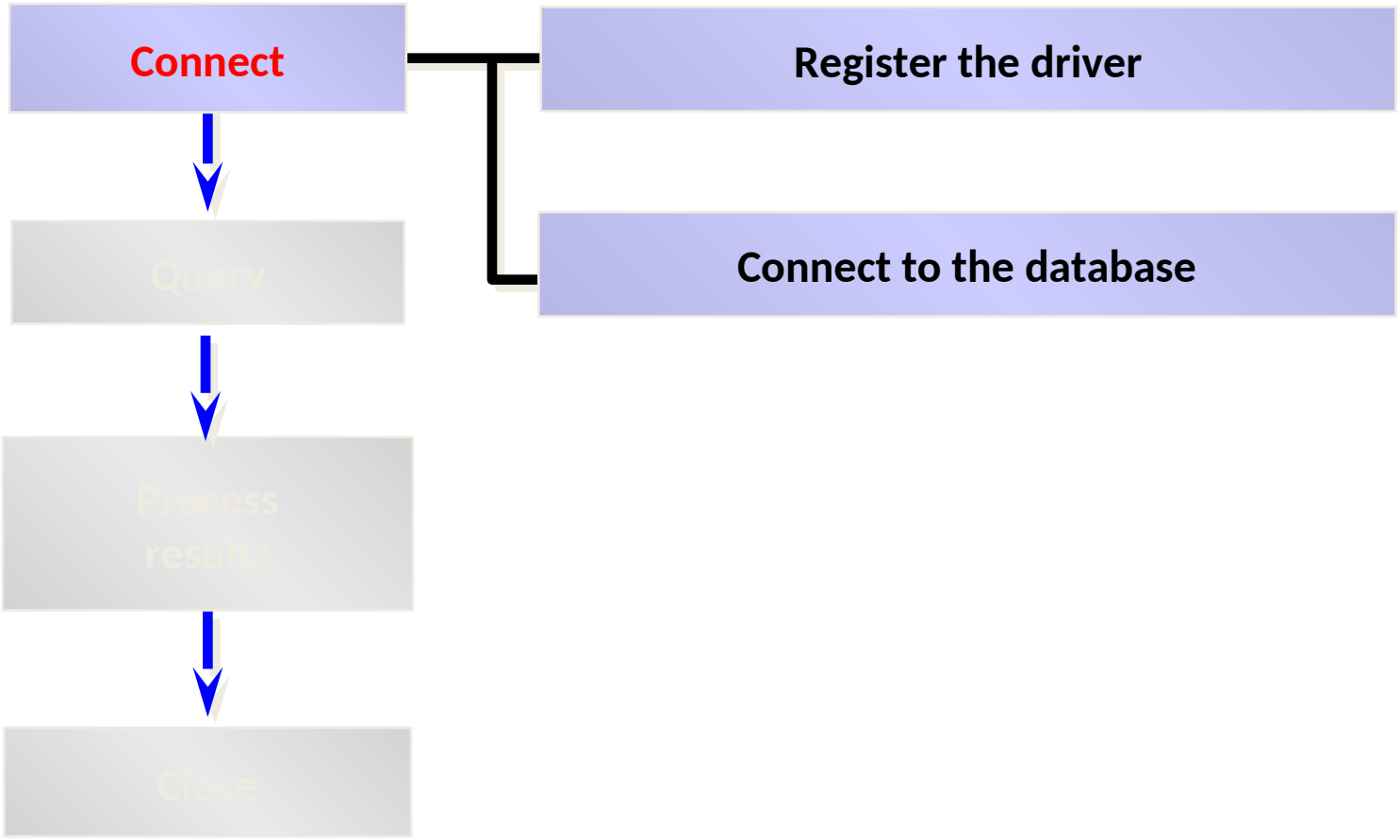
Accessing Database

1. Load the driver
2. Creating a **connection** object
3. Create a **statement** object
4. Execute an SQL query and get results using the **ResultSet** object
5. Close connections

Overview of Querying a Database With JDBC

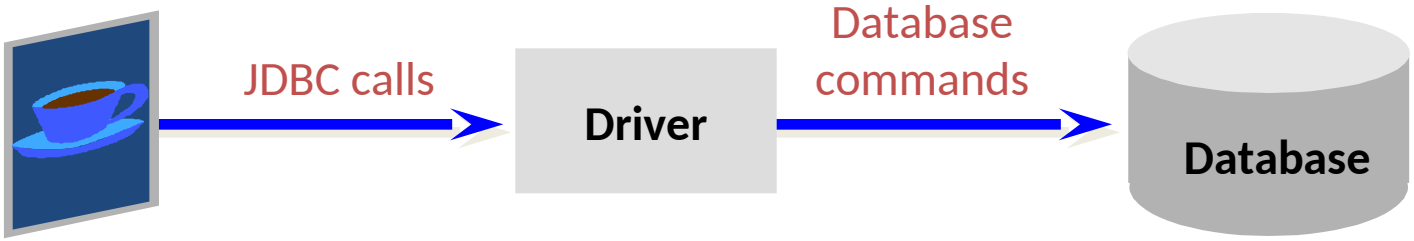


Stage 1: Connect



A JDBC Driver

- Is an interpreter that translates JDBC method calls to vendor-specific database commands



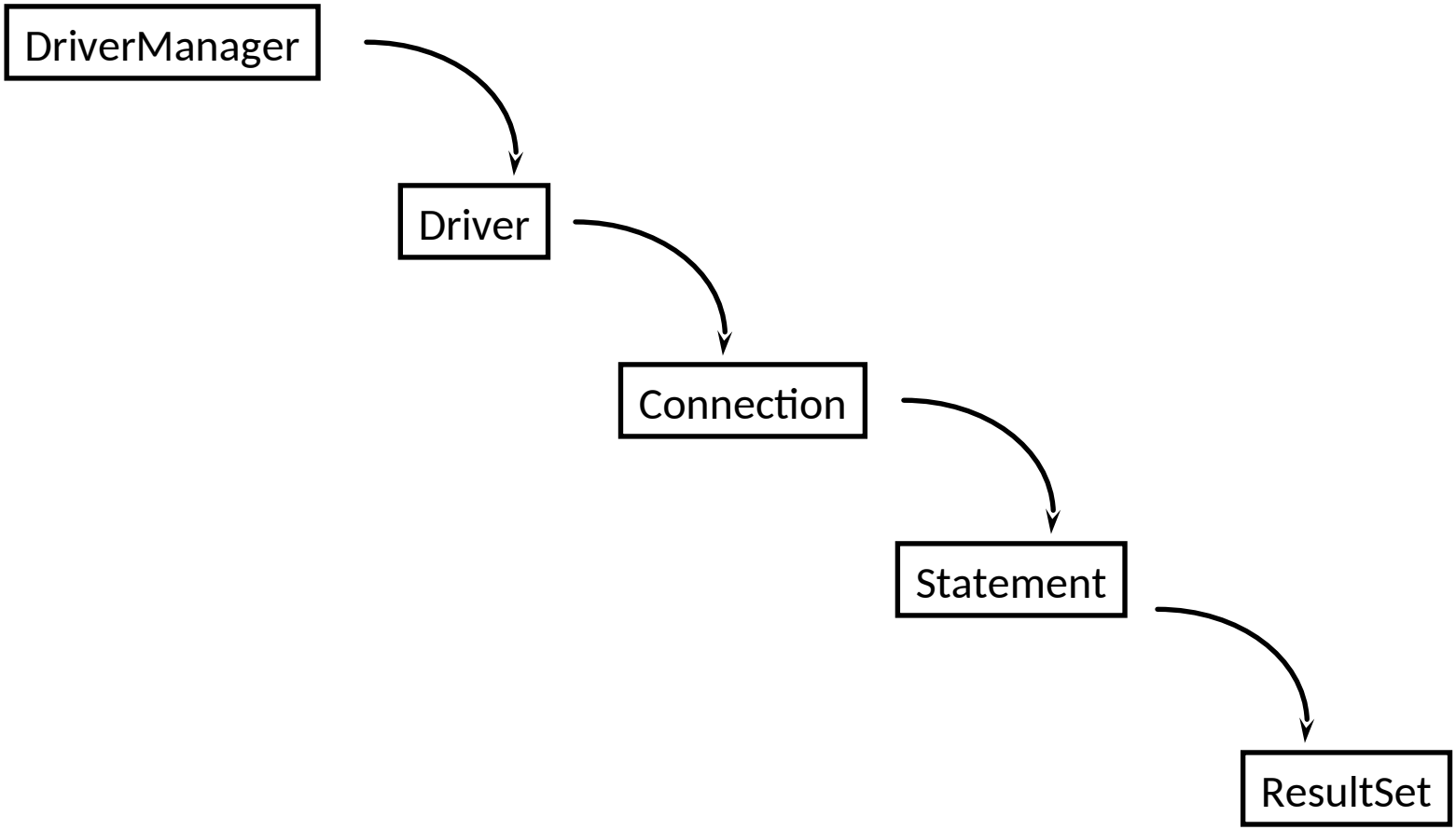
- Implements interfaces **in java.sql**
- Can also provide <u>a vendor's extensions</u> to the JDBC standard
- **Registering a Driver**
  - statically load driver

```
Class.forName("foo.bar.MyDriver");
Connection c = DriverManager.getConnection(...);
```

JDBC Object Classes

- DriverManager
  - Loads, chooses drivers
- Driver
  - connects to actual database
- Connection
  - a series of SQL statements to and from the DB
- Statement
  - a single SQL statement
- ResultSet
  - the records returned from a Statement

JDBC Class Usage



JDBC URLs

- To open a database, you must create a **“database URL”** that specifies:
  - That you’re using JDBC with “**jdbc**.”
  - The “**subprotocol**”: the name of the driver or the name of a database connectivity mechanism.
    - the first subprotocol available is the “**jdbc-odbc** bridge,”
  - The database identifier.
- **jdbc:subprotocol:source**
- each driver has its own subprotocol
- each subprotocol has its own syntax for the source

jdbc:odbc:*DataSource*

- e.g. jdbc:odbc:Northwind

jdbc:mysql://*host[:port]/database*

- e.g. jdbc:mysql://localhost:3306/accounting

DriverManager

- When you're ready to connect to the database,
  - call the **static** method **DriverManager.getConnection( )**
  - and pass it the following to get into the database.
    - the database URL,
    - the user name, and
    - a password

Connection getConnection (String url, String user, String password)

- Connects to given JDBC URL with given user name and password
- Throws java.sql.SQLException
- returns a Connection object

Connection

- A Connection represents a session with a specific database.
- Within the context of a Connection, SQL statements are executed and results are returned.
- Can have multiple connections to a database
- **Obtaining a Connection**

```
String url   = "jdbc:odbc:Northwind";
try {
        Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con = DriverManager.getConnection(url);
}
catch (ClassNotFoundException e) {
        e.printStackTrace();
}
catch (SQLException e){
         e.printStackTrace();
}
```

Statement

- Once the connection is made, you can use the resulting **Connection** object to create a **Statement** object using the **createStatement( )** method**.**
- A **Statement** object is used for executing a static SQL statement and obtaining the results produced by it.
- Statement Methods
  - **ResultSet executeQuery(String) -** Execute a SQL statement that returns a single ResultSet.
  - **int executeUpdate(String)** - Execute a SQL INSERT, UPDATE or DELETE statement. Returns the number of rows changed.
  - **boolean execute(String) -** Execute a SQL statement that may return multiple results.

ResultSet

- A **ResultSet** provides access to a table of data generated by executing a Statement.
- Only one **ResultSet** per Statement can be open at once.
- The table rows are retrieved in sequence.
- A **ResultSet** maintains a cursor pointing to its current row of data.
- The 'next' method moves the cursor to the next row.
  - you can’t rewind

ResultSet Methods

- **boolean next()**
  - activates the next row
  - the first call to next() activates the first row
  - returns false if there are no more rows
- **void close()**
  - disposes of the ResultSet
  - allows you to re-use the Statement that created it
  - automatically called by most Statement methods

ResultSet Methods

- ***Type* get*Type*(int columnIndex)**
  - returns the given field as the given type
  - fields indexed starting at 1 (not 0)
- ***Type* get*Type*(String columnName)**
  - same, but uses name of field
  - less efficient
- **int findColumn(String columnName)**
  - looks up column index given column name

ResultSet Methods

- String getString(String columnName)
- boolean getBoolean(String columnName)
- byte getByte(String columnName)
- short getShort(String columnName)
- int getInt(String columnName)
- long getLong(String columnName)
- float getFloat(String columnName)
- double getDouble(String columnName)
- Date getDate(String columnName)
- Time getTime(String columnName)
- Timestamp getTimestamp(String columnName)

isNull

- In SQL, NULL means the field is empty, not the same as **0** or **“”**
- In JDBC, you must explicitly ask if a field is null by calling **ResultSet.isNull(column)**

- **Sample Database**

Employee ID	Last Name	First Name
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

SELECT Example

```
Connection con = DriverManager.getConnection(url, "alex", "8675");
Statement st = con.createStatement();
ResultSet results = st.executeQuery("SELECT EmployeeID, LastName,
                                     FirstName FROM Employees");
while (results.next()) {
    int id = results.getInt(1);
    String last = results.getString(2);
    String first = results.getString(3);
    System.out.println("" + id + ": " + first + " " + last);
 }
 st.close();
 con.close();
```

Mapping Java Types to SQL Types

SQL type	Java Type
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Example – Connection (MySQL)

```
import java.sql.*;

public class SQLConnect {
  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;

  public SQLConnect(){}
    public void createConnection(){
    try{
      Class.forName("com.mysql.jdbc.Driver“);
    }
    catch (Exception E){
      System.out.println(E);
    }
    try{
      conn = DriverManager.getConnection("jdbc:mysql://localhost/mytest”,”test”
 “1234");
    }
    catch (SQLException E){
      System.out.println(E);
    }
}
```

Importing java.sql.* that contains all the classes we need

Connection, Statement and ResultSet are defined as class variables

Dynamically loading the specific JDBC driver. The runtime environment must know where the library is located!

Connecting to the database using the url

Example – Access and Query

```
public String getPhones(){
  String output = "";
  try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT * FROM phones");
    if (rs != null){
      while (rs.next()){
        output += rs.getString("phone") + "\n";
      }
    }
  }
  catch (Exception E){
    System.out.println(E.getMessage());
  }
```

Creating a statement

Creating a ResultSet, based on a SQL statement

Going through the ResultSet by using rs.next(). Remember – you need to call the next method before you start reading from the ResultSet

Reading a field from the ResultSet

Example – Cleaning off

```
finally {
  if (rs != null) {
    try {
      rs.close();
    }
    catch (SQLException sqlEx) {}
      rs = null;
    }
    if (stmt != null) {
      try {
        stmt.close();
      }
      catch (SQLException sqlEx) {}
      stmt = null;
    }
  }
  return output;
}
```

Cleaning off is best done in the “finally” clause

Cleaning off ResultSet

Cleaning off Statement, after the ResultSet

Cleaning off the connection, in a different method (why?)

```
public void closeConnection(){
  if (conn != null){
    try {
      conn.close();
    }
    catch (SQLException sqlEx){}
    conn = null;
  }
}
```

Example – Test Client

```
public class Test {
        public static void main(String[] args) {
                SQLConnect connect = new SQLConnect();
                connect.createConnection();
                String allPhones = connect.getPhones();
                connect.closeConnection();
                System.out.println("phones:");
                System.out.println(allPhones);
        }
}
```

<u>Output</u>

```
phones:
+972-4-9831894
```