JDBC

- JDBC provides a standard library for accessing relational databases.
- By using the JDBC API, you can access a wide variety of SQL databases with exactly the same Java syntax.
- The JDBC API standardizes the approach for connecting to databases, the syntax for sending queries and committing transactions, and the data structure representing the result.
- JDBC does not attempt to standardize the SQL syntax.
- So, you can use any SQL extensions your database vendor supports.
- However, since most queries follow standard SQL syntax, using JDBC lets you change database hosts, ports, and even database vendors with minimal changes to your code.

JDBC is a broad topic

- JDBC is a broad topic, and a complete discussion of its object model requires a class of its own.

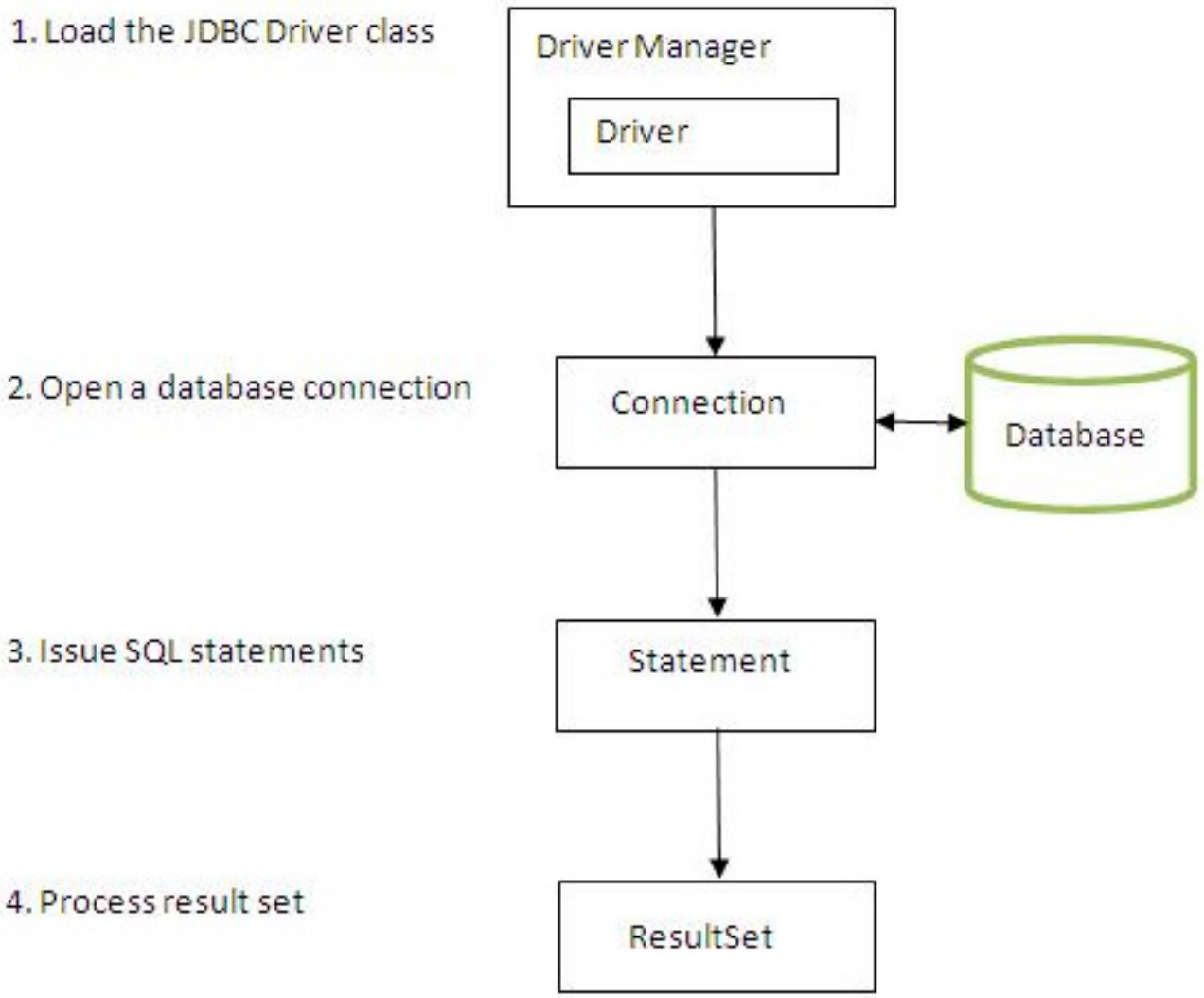
- This lecture starts with a look at the object model in the java.sql package, but does not attempt to explain JDBC thoroughly.

SQL

- A complete tutorial on database programming is beyond the scope of this class.
- We cover the basics of using JDBC in general, presuming you are already familiar with SQL.

- **The most important members of the java.sql package are as follows:**
  - **The DriverManager class**
    - The DriverManager class provides static methods for managing JDBC drivers.
    - Each JDBC driver you want to use must be registered with the DriverManager.
  - **The Driver interface**
    - The Driver interface is implemented by every JDBC driver class.
    - The driver class itself is loaded and registered with the DriverManager, and the DriverManager can manage multiple drivers for any given connection request.
  - **The Connection interface**
    - The Connection interface represents a connection to the database.
    - An instance of the Connection interface is obtained from the getConnection method of the DriverManager class.
  - **The Statement interface**
    - You use the statement interface method to execute an SQL statement and obtain the results that are produced.
  - **The ResultSet interface**
    - The ResultSet interface represents a table-like database result set.
  - **The PreparedStatement interface**
    - The PreparedStatement interface extends the Statement interface and represents a precompiled SQL statement.
    - You use an instance of this interface to execute efficiently an SQL statement multiple times.
  - **The ResultSetMetaData interface**
    - The ResultSetMetaData interface represents the meta data of a ResultSet object.

Four basic steps required to work with JDBC



Step 1. Load the JDBC Driver

- The driver is the piece of software that knows how to talk to the actual DB server.
- To load the driver,
  - you just load the appropriate class
  - a static block in the driver class itself automatically makes a driver instance and registers it with the JDBC driver manager.

- These requirements bring up two interesting questions.
  - First, how do you load a class without making an instance of it?
  - Second, how can you refer to a class whose name isn't known when the code is compiled?

- The answer to both questions is to use Class.forName.

```
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");
Class.forName("com.mysql.jdbc.Driver");
```

Step 2. Establish a Connection

- After you register a JDBC driver with the DriverManager, you can use it to get a connection to the database.
- In JDBC, a database connection is represented by the java.sql.Connection interface.
- You use the DriverManager class's getConnection method to obtain a Connection object.

```
Connection con = DriverManager.getConnection(“jdbc:mysql//host/db”, “usr", “pwd”);
```

Step 3. Creating a Statement

- After you have a Connection object, your SQL skill takes over.
- Basically, you can pass any SQL statement that your database server understands.
- After you have a Connection object from the DriverManage.getConnection method, you are ready to pass a SQL statement.
- To do this, you need to create another JDBC object called Statement.
- You can do this using the createStatement method of the Connection interface.

```
Statement statement = connection.createStatement();
```

Step 4. Creating a ResultSet

- A ResultSet is the representation of a database table returned from a Statement object.
- A ResultSet object maintains a cursor pointing to its current row of data.
- When the cursor is first returned, it is positioned before the first row.
- To access the first row of the ResultSet, you need to call the next() method of the ResultSet interface.
- The next() method moves the cursor to the next row and can return either a true or false value.
- It returns true if the new current row is valid; it returns false if there are no more rows.
- Normally, you use this method in a while loop to iterate through the ResultSet object.
- To get the data from the ResultSet, you can use one of many the getXXX methods of ResultSet, such as getInt, getLong, and so on.

```
String sql = "SELECT FirstName, LastName FROM Users";
ResultSet resultSet = statement.executeQuery(sql);
while (resultSet.next()) {
System.out.println(resultSet.getString(1) + ":" + resultSet.getString("LastName") );
}
```

Closing JDBC Objects Explicitly Can Save You Headaches!

Most Java programmers close a connection with the database directly—without closing the *ResultSet* or *Statement*.

For example, take a look at the following code snippet:

```
try
{
    Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );

    Connection conn = DriverManager.getConnection("jdbc:odbc:mydb","x","x");
    Statement stmt  = conn.createStatement();
    ResultSet rs    = stmt.executeQuery( "SELECT * FROM TABLE" );
}
catch( SQLException e )
{
    out.println("Some error happened", e);
}
```

This code looks perfectly fine. But not all JDBC drivers clean themselves up.

To be safe, remember to close the everything explicitly in reverse order. Most people do it as shown below:

```
Connection conn = null;
ResultSet rs    = null;
Statement stmt  = null;

try
{
    Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );

    conn = DriverManager.getConnection("jdbc:odbc:issue","x","x");
    stmt = conn.createStatement();
    rs   = stmt.executeQuery( "SELECT * FROM TABLE" );
}
catch( SQLException e )
{
    Out.println("Some error happened", e);
}
finally
{
    try
    {
        if (rs  != null)
            rs.close();
        if (stmt  != null)
            stmt.close();
        if (conn  != null)
            conn.close();
    }
    catch (SQLException e)
    {
        // handle exception
    }
}
```

Connection Pool

- Connection Pool is a cache of database connections maintained so that the connections can be reused when future requests to the database are required.
- Connection pools are used to enhance the performance of executing commands on a database.
- Opening and maintaining a database connection for each user, especially requests made to a dynamic database-driven website application, is costly and wastes resources.
- In connection pooling, after a connection is created, it is placed in the pool and it is used over again so that a new connection does not have to be established.
- Connection pooling also cuts down on the amount of time a user must wait to establish a connection to the database.
- Connection pooling being normally used in web-based and enterprise applications is usually handled by an application server. Any dynamic web page can be coded to open a connection and close it normally but behind the scenes when a new connection is requested, one is returned from the connection pool maintained by the application server. Similarly, when a connection is closed it is actually returned to the connection pool.

Transactions

- A transaction is a unit of work that can comprise one or more SQL operations.
- In the examples you have seen so far, every SQL operation is executed and the change is committed (made permanent) on the database right after the SQL statement is executed.
- In some cases, however, a set of SQL operations must succeed or fail as a whole.
- If one of the SQL statements fails, the other SQL statements in the group must also be rolled back.
- Consider the following scenario.
  - *In an online store a customer purchases several books. When the customer checks out and pays using a credit card, the following things must happen:*
    - 1. A record must be added to the Orders table specifying the order, including the delivery address and the credit card details. This operation results in an OrderId used to identify each item in the OrderDetails table.
    - 2. A record must be inserted into the OrderDetails table for each purchase item. Each item is linked to the Orders table using the OrderId value returned by the previous SQL operation.
  - Now, if everything goes smoothly, both SQL statements will be executed successfully in the DB.
  - Things can go wrong, however. Say, for example, the first operation is committed successfully but the second SQL statement fails. In this case, the order details are lost and the customer won't get anything. The customer's credit card will still be debited by the amount of the purchase, however, because a record is added to the Orders table.
  - In this case, you want both SQL statements to succeed or fail as a whole. You can do this using a transaction. Upon a failed transaction, you can notify the customer so that he or she can try to check out again.

Commit and RollBack

- By default, a Connection object's auto-commit state is true, meaning that the database is updated when an SQL statement is executed.
- If you want to group a set of SQL statements in one transaction, you must first tell the Connection object not to update the change until it is explicitly notified to do so.
- You do this by calling the setAutoCommit method and passing false as its argument, as follows:

```
connection.setAutoCommit(false);
```

- Then, you can execute all the SQL statements in the group normally, using the executeQuery and the updateQuery methods.
- After the last call to the executeQuery or the updateQuery method, you call the commit method of the Connection object to make the database changes permanent, such as

```
connection.commit();
```

- If you don't call the commit method within a specified time, all SQL statements will be rolled back after calling the setAutoCommit method.
- Alternatively, you can roll back the transaction explicitly by calling the Connection object's rollback method:

```
connection.rollback();
```

PreparedStatement

- An object that represents a precompiled SQL statement.
- A SQL statement is precompiled and stored in a PreparedStatement object.
- This object can then be used to efficiently execute this statement multiple times.
- In the following example of setting a parameter, con represents an active connection:

```
PreparedStatement pstmt =
                        con.prepareStatement("UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");

pstmt.setBigDecimal(1, 153833.00);
pstmt.setInt(2, 110592);
```

```java
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JavaApplication1 {

    public static void main(String[] args)
    {
        try {
            Class.forName("com.mysql.jdbc.Driver");

            Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/databaseName", "root", "12345678");

            Statement stmnt = con.createStatement();

            ResultSet rs = stmnt.executeQuery("SELECT * FROM student");

            while (rs.next()) {
                System.out.println(rs.getString(1) + "-" + rs.getString(2));
            }
        } catch (SQLException ex) {
            //handle exception
        } catch (ClassNotFoundException ex) {
            //handle exception
        }
    }
}
```