JDBC

Voronezh, 2018

T · · Systems

Agenda

- What is JDBC?
- JDBC Architecture
- Common JDBC Components
- JDBC Driver Types
- Database Connections
- Statements
- Result Sets
- Data Types
- Transactions
- Spring JDBCTemplate
- JDBC Quickstart

T · · Systems

What is JDBC?

- **JDBC** stands for **J**ava **D**ata**B**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

- The **JDBC library** includes APIs for each of the tasks mentioned below that are commonly associated with database usage:
  - Making a connection to a database.
  - Creating SQL statements.
  - Executing SQL queries in the database.
  - Viewing and Modifying the resulting records.

- **Package**:
  - java.sql
  - javax.sql

T · · Systems

JDBC Architecture (1)

- The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers:

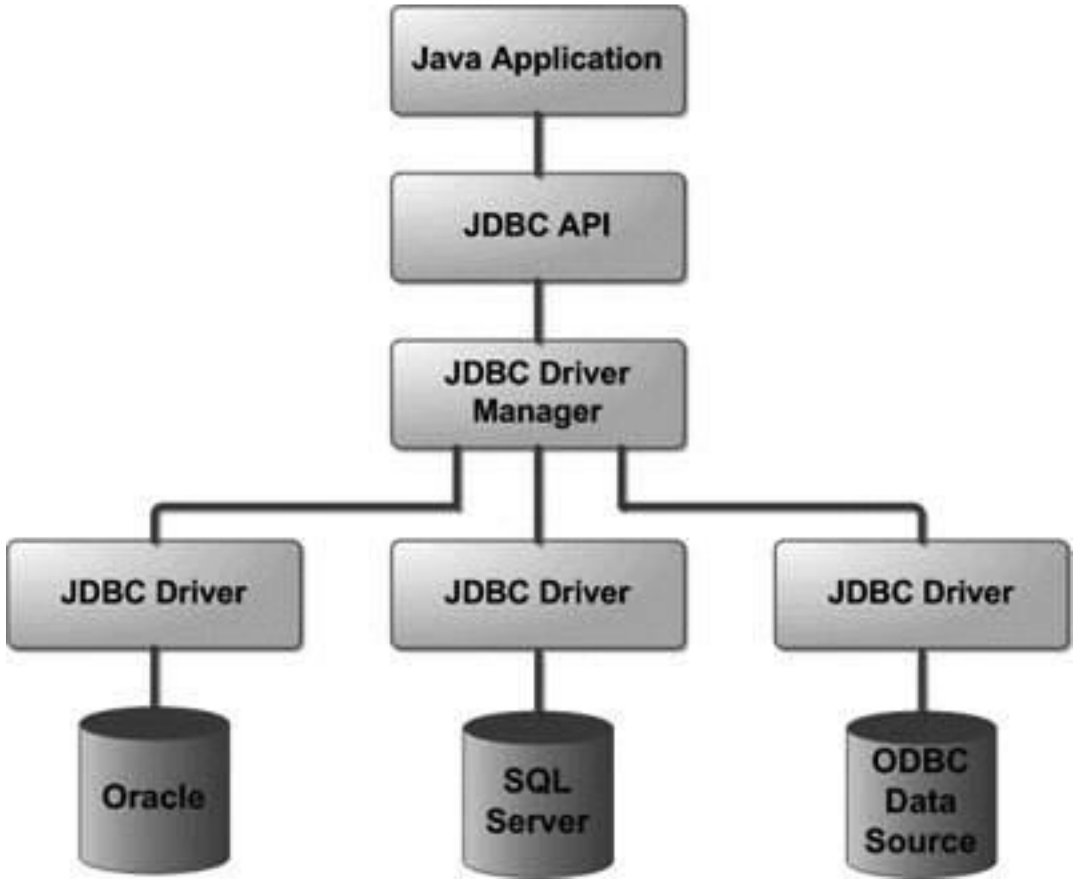
  - **JDBC API:** This provides the application-to-JDBC Manager connection.
  - **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

- The **JDBC API** uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

- The **JDBC driver manager** ensures that the correct driver is used to access each data source.

- The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

JDBC Architecture (2)



Common JDBC Components (1)

- **DriverManager:** This class **manages a list of database drivers**. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface **handles the communications with the database server**. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

- **Connection:** This interface with all methods **for contacting a database**. The connection object represents communication context, i.e. all communication with database is through connection object only.

Common JDBC Components (2)

- **Statement:** You use objects created from this interface **to submit the SQL statements to the database**. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects **hold data retrieved from a database** after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

- **SQLException:** This class **handles any errors** that occur in a database application.

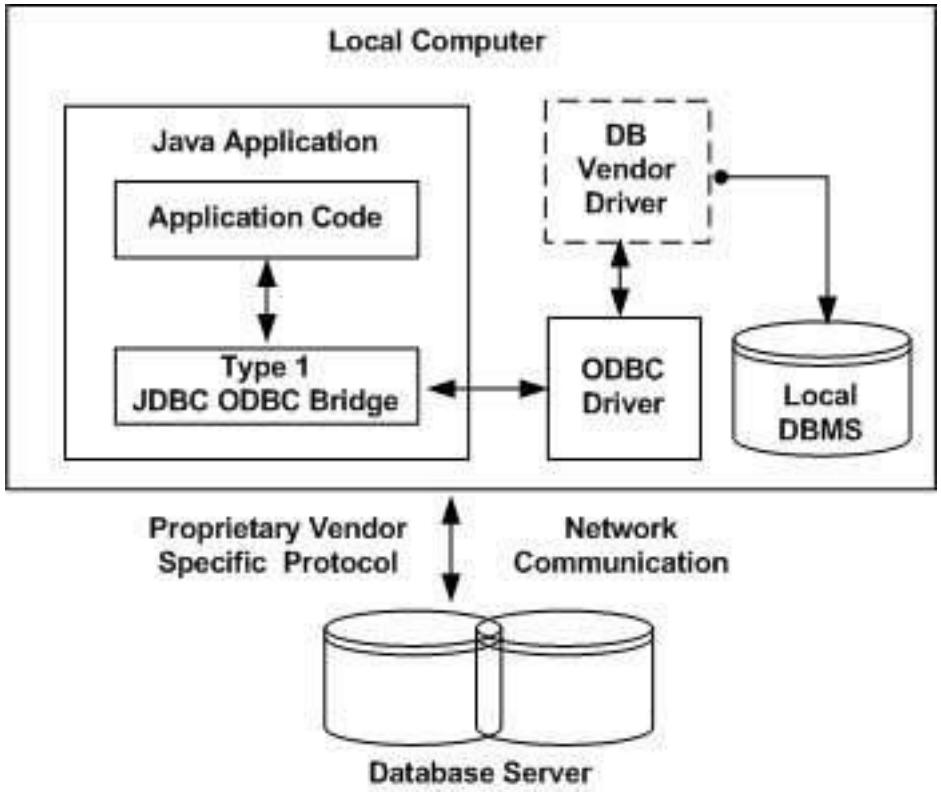
T-Systems

What is JDBC Driver?

- **JDBC drivers** implement the defined interfaces in the JDBC API, **for interacting with your database server**.
- **For example**, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- The **java.sql** package that ships with JDK, contains various classes with their behaviors defined and their actual implementations are done in third-party drivers.
- Third party vendors implements the **java.sql.Driver** interface in their database driver.

Type 1: JDBC-ODBC Bridge Driver

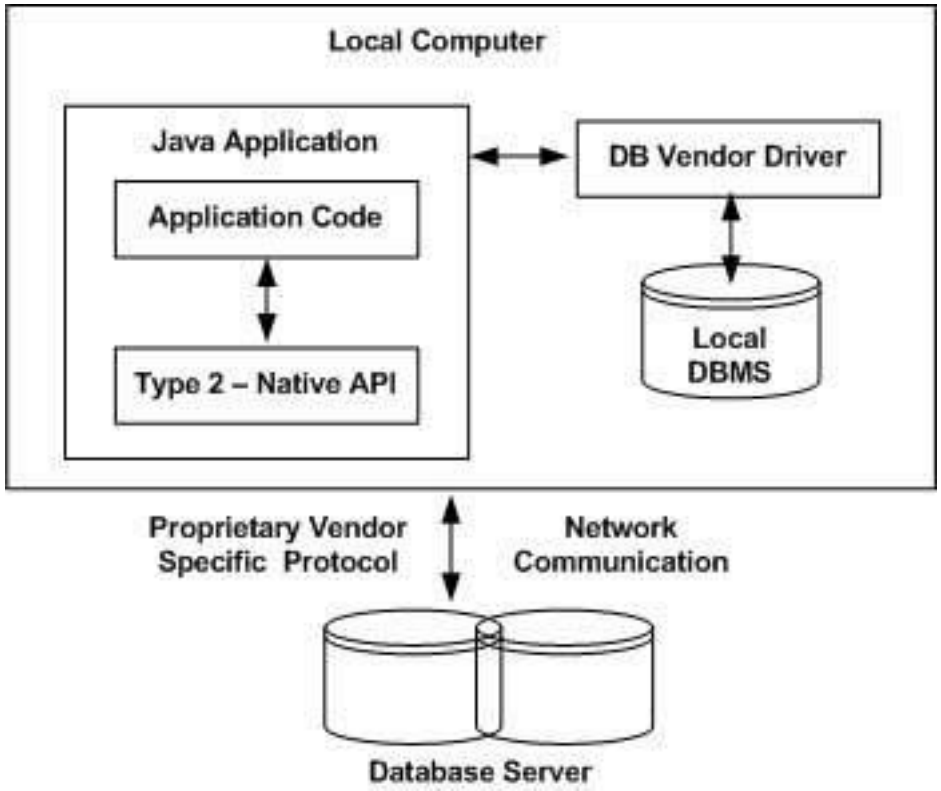
- In a **Type 1** driver, a JDBC bridge is used to access ODBC drivers installed on each client machine.
- Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.



T · · Systems

Type 2: JDBC-Native API

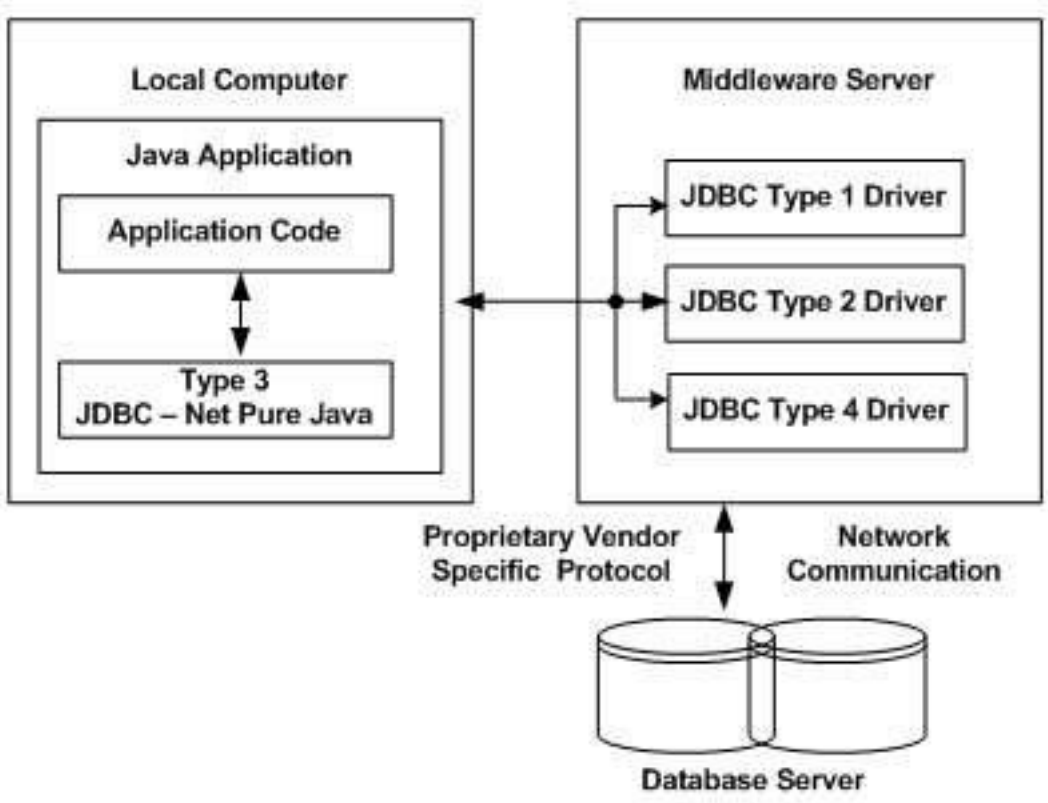
- In a **Type 2** driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database.
- These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.



T · · Systems

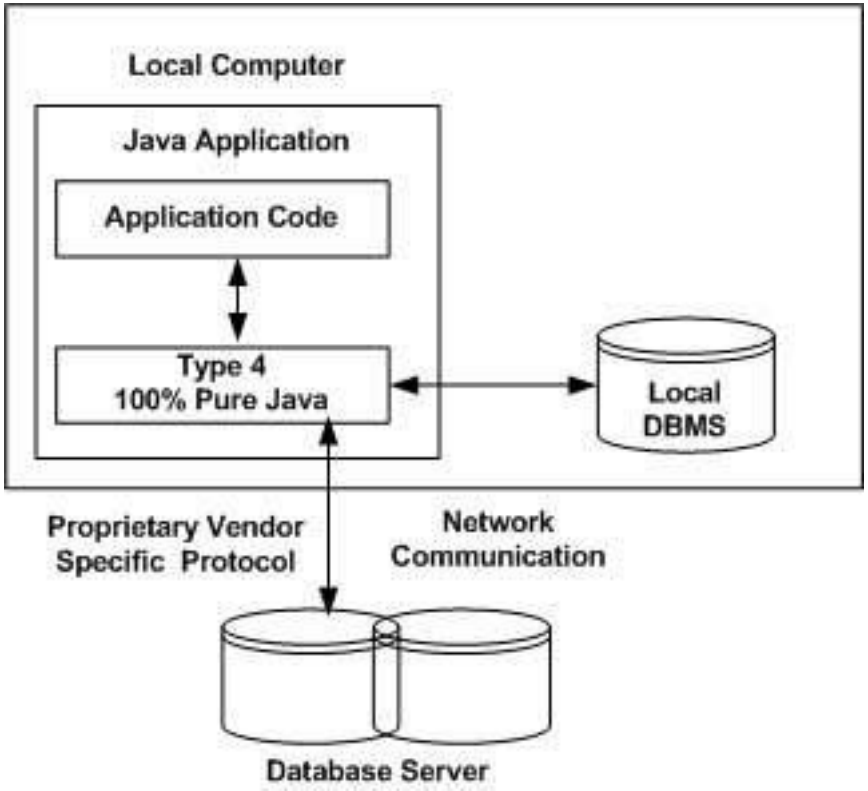
Type 3: JDBC-Net pure Java

- In a **Type 3** driver, a three-tier approach is used to access databases.
- The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.



Type 4: 100% Pure Java

- In a **Type 4 driver**, a pure Java-based driver communicates directly with the vendor's database through socket connection.
- This is the **highest performance driver** available for the database and is usually provided by the vendor itself.



T · · Systems

Database Connections (1)

- To establish a connection use the **DriverManager.getConnection(…)** method:
  - getConnection(String **url**)
  - getConnection(String **url**, Properties **properties**)
  - getConnection(String **url**, String **user**, String **password**)

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**hostname/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:portNumber:databaseName

Database Connections (2)

- **getConnection(String url, String user, String password)**

```
String URL = "jdbc:mysql://localhost/tuni";
String USER = "tuni";
String PASS = "Qazxsw23"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

- **getConnection(String url)**

```
String URL = "jdbc:mysql://localhost/tuni?user=tuni&password=Qazxsw23";
Connection conn = DriverManager.getConnection(URL);
```

Database Connections (3)

- **getConnection(String url, Properties properties)**

```
String URL = "jdbc:mysql://localhost/tuni";

Properties properties = new Properties();
properties.setProperty("user", "tuni");
properties.setProperty("password", "Qazxsw23");

Connection conn = DriverManager.getConnection(URL, properties);
```

Database Connections (4)

- At the end of your JDBC program, it is **required explicitly to close all the connections** to the database to end each database session.

- Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy ☺

```
String URL = "jdbc:mysql://localhost/tuni?user=tuni&password=Qazxsw23";
Connection conn = DriverManager.getConnection(URL);
. . .
conn.close();
```

- To ensure that a connection is closed, you could provide a **'finally'** block in your code. A finally block always executes, regardless of an exception occurs or not.

JDBC Statements

- Once a connection is obtained we can interact with the database.
- The JDBC **Statement**, **CallableStatement**, and **PreparedStatement** interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are **using static SQL statements** at runtime. The Statement interface **cannot accept parameters**.
PreparedStatement	Use the when you plan to **use the SQL statements many times**. The PreparedStatement interface **accepts input parameters** at runtime.
CallableStatement	Use the when you want to **access the database stored procedures**. The CallableStatement interface can also **accept runtime input parameters**.

Statement (1)

- To create **Statement** use the Connection object's **createStatement( )** method:

```
String SQL= "SELECT id, first, last, age FROM Registration";
Statement stmt = null;

try {
      stmt = conn.createStatement();
      stmt.executeQuery(SQL);
} catch (SQLException e) {
      // handle exception
} finally {
      stmt.close();
}
```

- Just as you **close** a Connection object to save database resources, for the same reason you should also close the Statement object.

Statement (2)

- Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods:
  - **boolean execute(String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute **SQL DDL** statements (create, alter and drop schema objects, etc.) or when you need to use truly dynamic SQL.
  - **int executeUpdate(String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an **INSERT**, **UPDATE**, or **DELETE** statement.
  - **ResultSet executeQuery(String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a **SELECT** statement.

PreparedStatement (1)

- The **PreparedStatement** interface extends the Statement interface and gives you the flexibility of supplying arguments dynamically.

```
String SQL = "UPADTE Employees SET age = ? WHERE id = ?";
PreparedStatement pstmt = null;

try {
      pstmt = conn.prepareStatement(SQL);
} catch (SQLException e) {
      // handle exception
} finally {
        pstmt.close();
}
```

- Just as you close a Statement object, for the same reason you should also **close** the PreparedStatement object.

PreparedStatement (2)

- String SQL = “UPDATE Employees SET age = **?** WHERE id = **?**";
- All parameters in JDBC are represented by the **?** symbol, which is known as the **parameter marker**. You **must supply values** for every parameter **before executing** the SQL statement.
- The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an **SQLException**.
- Each parameter marker is referred by its ordinal position. **The first marker represents position 1**, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.
- Prepared statements are **resilient against SQL injection**, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.

CallableStatement (1)

- Just as a Connection object also creates the **CallableStatement**, which would be used to execute a call to a database stored procedure.

```
String SQL = "{call someProcedure(?, ?)}";
CallableStatement cstmt = null;

try {
      cstmt = conn.prepareCall(SQL);
} catch (SQLException e) {
      // handle exception
} finally {
        cstmt.close();
}
```

- Just as you close other Statement object, for the same reason you should also **close** the CallableStatement object.

T · · Systems

CallableStatement (2)

- The **PreparedStatement** object only uses the **IN** parameter.
- The **CallableStatement** object can use three types of parameters: **IN**, **OUT**, and **INOUT**.

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to **IN** parameters with the **setXXX()** methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the **OUT** parameters with the **getXXX()** methods.
INOUT	A parameter that provides both input and output values. You bind variables with the **setXXX()** methods and retrieve values with the **getXXX()** methods.

CallableStatement (3)

- The String variable SQL = "{call **someProcedure**(**?**, **?**)}"; represents the stored procedure, with parameter placeholders.

- Using the CallableStatement objects is much like using the PreparedStatement objects. You **must bind values** to all the parameters **before executing the statement**, or you will receive an **SQLException**.

- If you have **IN** parameters, use the **setXXX()** method that corresponds to the Java data type you are binding.

- When you use **OUT** and **INOUT** parameters you must employ an additional CallableStatement method, **registerOutParameter()** – It binds the JDBC data type, to the data type that the stored procedure is expected to return.

- Once you call your stored procedure, you retrieve the value from the **OUT** parameter with the appropriate **getXXX()** method. This method **casts the retrieved value** of SQL type to a Java data type.

T-Systems

Result Sets

- The SQL statements that read data from a database query, **return the data in a result set**.
- A **ResultSet** object **maintains a cursor** that points to the current row in the result set. The term "result set" **refers to** the **row** and **column** data contained in a ResultSet object.
- The **cursor is movable** based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created:
  - **createStatement(int RSType, int RSConcurrency);**
  - **prepareStatement(**String SQL, **int RSType, int RSConcurrency);**
  - **prepareCall(**String sql, **int RSType, int RSConcurrency);**
- The **first argument indicates the type of a ResultSet** object and the **second argument** is one of two ResultSet constants for **specifying** whether a **result set is read-only or updatable**.

Type of ResultSet

- If you do not specify any ResultSet type, you will automatically get one that is **TYPE_FORWARD_ONLY**.
- The possible RSType are given below:

Type	Description
TYPE_FORWARD_ONLY	The cursor can **only move forward** in the result set.
TYPE_SCROLL_INSENSITIVE	The cursor can **scroll forward and backward**, and the result set is **not sensitive to changes** made by others to the database that occur after the result set was created.
TYPE_SCROLL_SENSITIVE	The cursor can **scroll forward and backward**, and the result set **is sensitive to changes** made by others to the database that occur after the result set was created.

Concurrency of ResultSet

- If you do not specify any Concurrency type, you will automatically get one that is **CONCUR_READ_ONLY**.
- The possible RSConcurrency are given below:

Type	Description
CONCUR_READ_ONLY	Creates a **read-only** result set. This is the default
CONCUR_UPDATABLE	Creates an **updateable** result set.

ResultSet Type and Concurrency Example

```
String SQL= "SELECT id, first, last, age FROM Registration";
Statement stmt = null;

try {
      stmt = conn.createStatement(
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
} catch (SQLException e) {
      // handle exception
} finally {
      stmt.close();
}
```

ResultSet Methods

- The methods of the ResultSet interface can be broken down into three categories:
  - **Navigational methods:** Used to move the cursor around.
  - **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
  - **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

Navigating a Result Set

- **public void beforeFirst() throws SQLException**

  Moves the cursor just before the first row.
- **public boolean first() throws SQLException**

  Moves the cursor to the first row.
- **public void last() throws SQLException**

  Moves the cursor to the last row.
- **public boolean absolute(int row) throws SQLException**

  Moves the cursor to the specified row.
- **public boolean relative(int row) throws SQLException**

  Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
- **public boolean previous() throws SQLException**

  Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
- **public boolean next() throws SQLException**

  Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
- **public int getRow() throws SQLException**

  Returns the row number that the cursor is pointing to.

Viewing a Result Set

- The ResultSet interface contains **dozens** of methods **for getting the data of the current row.**

- There is a **get method for each of the possible data types**, and each get method has two versions:
  - One that takes in a **column name**.
  - One that takes in a **column index**.

  - **public int getInt(String columnName) throws SQLException**
    Returns the int in the current row in the column named columnName.

  - **public int getInt(int columnIndex) throws SQLException**
    Returns the int in the current row in the specified column index. **The column index starts at 1**, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Updating a Result Set (1)

- The ResultSet interface contains a **collection** of update **methods for updating the data of a result set.**

- As with the get methods, there are two update methods for each data type:
  - One that takes in a **column name**.
  - One that takes in a **column index**.

  - **public void updateString(int columnIndex, String s) throws SQLException**
    Changes the String in the specified column to the value of s.

  - **public void updateString(String columnName, String s) throws SQLException**
    Similar to the previous method, except that the column is specified by its name instead of its index.

Updating a Result Set (2)

- **Updating a row** in the result **set changes** the columns of the current row **in the ResultSet** object, but **not in the underlying database**.

- **To update your changes to the row in the database**, you need to invoke one of the following methods:
  - **public void updateRow()**

    Updates the current row by updating the corresponding row in the database.
  - **public void deleteRow()**

    Deletes the current row from the database
  - **public void refreshRow()**

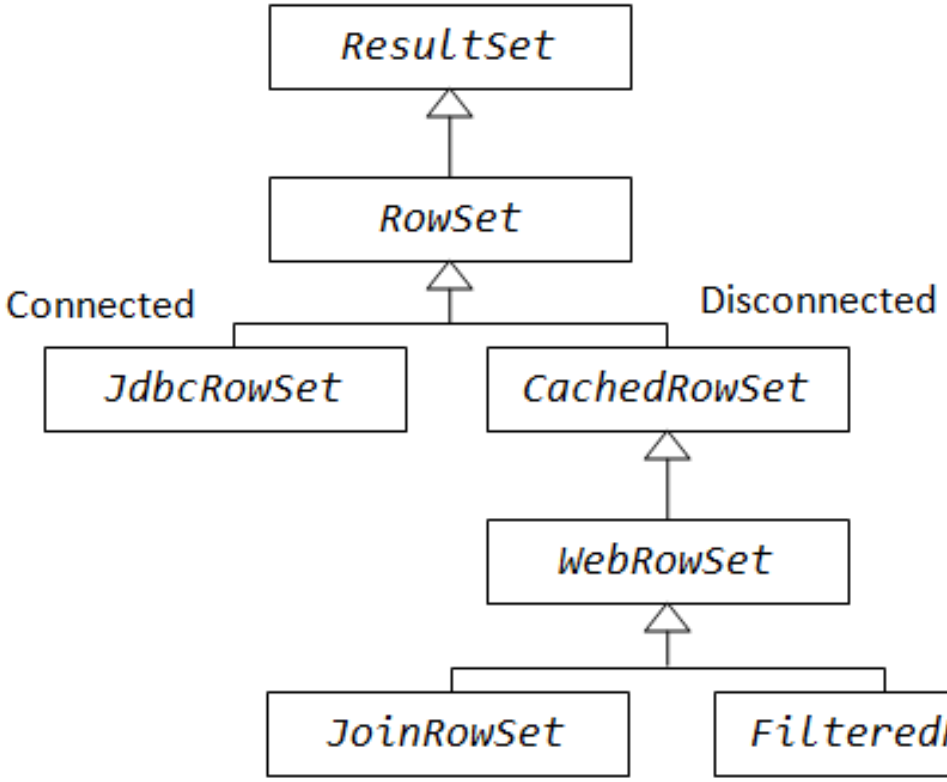
    Refreshes the data in the result set to reflect any recent changes in the database.
  - **public void cancelRowUpdates()**

    Cancels any updates made on the current row.
  - **public void insertRow()**

    Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

RowSet

- **RowSet** interface is another key interface from JDBC API, which **extends the ResultSet** interface to provide support for the **JavaBean** component model.
- Another feature of RowSet is that it **supports event listeners**.



T · · Systems

Data Types (1)

- The **JDBC** driver **converts** the **Java** data **type to** the **appropriate JDBC type**, before sending it to the database.
- It **uses** a **default mapping** for most data types.
- The **setXXX()** and **updateXXX()** methods enable you to convert specific Java types to specific JDBC data types.
- The methods, **setObject()** and **updateObject()**, enable you to map almost any Java type to a JDBC data type.
- ResultSet object provides corresponding **getXXX()** method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position.

Data Types (2)

- SQL's use of **NULL** values and Java's use of **null** are different concepts. So, to handle SQL NULL values in Java, there are three tactics you can use:
  - **Avoid** using **getXXX()** methods that return **primitive** data **types**.
  - Use **wrapper** classes **for primitive** data **types**, and use the ResultSet object's **wasNull()** method to test whether the wrapper class variable that received the value returned by the getXXX() method should be set to null.
  - Use primitive data types and the ResultSet object's wasNull() method to test whether the primitive variable that received the value returned by the getXXX() method should be set to an acceptable value that you've chosen to represent a NULL.

Transactions

- If your JDBC Connection is in **auto-commit** mode, which it is **by default**, then **every SQL statement is committed to the database upon its completion**.

- That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions:
  - To increase **performance**.
  - To maintain the **integrity** of business processes.
  - To use distributed **transactions**.

- To enable manual transaction support instead of the auto-commit mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a **boolean false to setAutoCommit()**, you turn off auto-commit.
- You can pass a **boolean true to turn it back on again**.

```
conn.setAutoCommit(false);
```

Commit & Rollback

- Once you are done with your changes and you want to commit the changes then call commit() method on connection object as follows:

```
conn.setAutoCommit(false);
. . .
conn.commit();
```

- Otherwise, to roll back updates to the database made using the Connection named conn, use the following code:

```
conn.rollback();
```

Savepoints

- When you set a **savepoint** you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

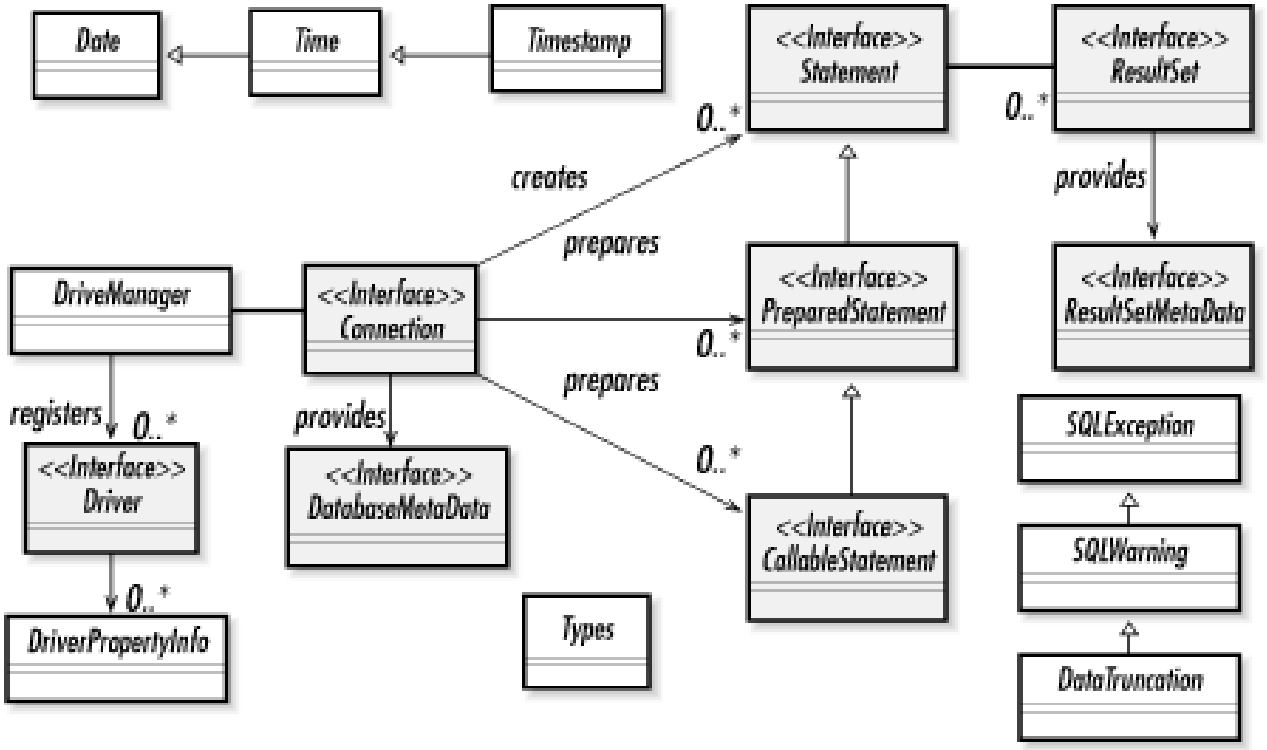
- The Connection object has two new methods that help you manage savepoints:

  - **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.

  - **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

- There is one **rollback(String savepointName)** method, which rolls back work to the specified savepoint.

JDBC API



T · · Systems

Spring JDBCTemplate

- Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC API, but eliminates a lot of problems of JDBC API.

- **Problems of JDBC API**:
  - We need to write a lot of code before and after executing the query, such as creating Connection, Statement, closing ResultSet, Connection etc.
  - We need to perform exception handling code on the database logic.
  - We need to handle transaction.
  - Repetition of all these codes from one to another database logic is a time consuming task.

- **Advantage of Spring JdbcTemplate:**
  - Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

JDBC Quickstart

```
public class JdbcApp {

    public static void main(String[] args) throws SQLException {

        Properties properties = new Properties();
        properties.setProperty("user", "tuni");
        properties.setProperty("password", "Qazxsw23");

        // Class.forName("com.mysql.jdbc.Driver");
        Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/tuni", properties);

        DatabaseMetaData metadata = connection.getMetaData();

        ResultSet rs = metadata.getTables(null, null, null, new String[] { "TABLE" });
        ResultSetMetaData rsmd = rs.getMetaData();

        for (int i = 1; i < rsmd.getColumnCount() + 1; ++i) {
            System.out.println(rsmd.getColumnName(i));
        }

        while (rs.next()) {
            System.out.println(String.format("Table: %s", rs.getString(3)));
        }

        rs.close();
        connection.close();
    }
```

Further Reading

- http://www.tutorialspoint.com/jdbc/
- http://www.mkyong.com/tutorials/jdbc-tutorials/
- http://www.oracle.com/technetwork/java/javase/jdbc/index.html
- http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html
- http://www.tutorialspoint.com/jdbc/jdbc_interview_questions.htm
- http://www.tutorialspoint.com/spring/spring_jdbc_example.htm
- https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#jdbc