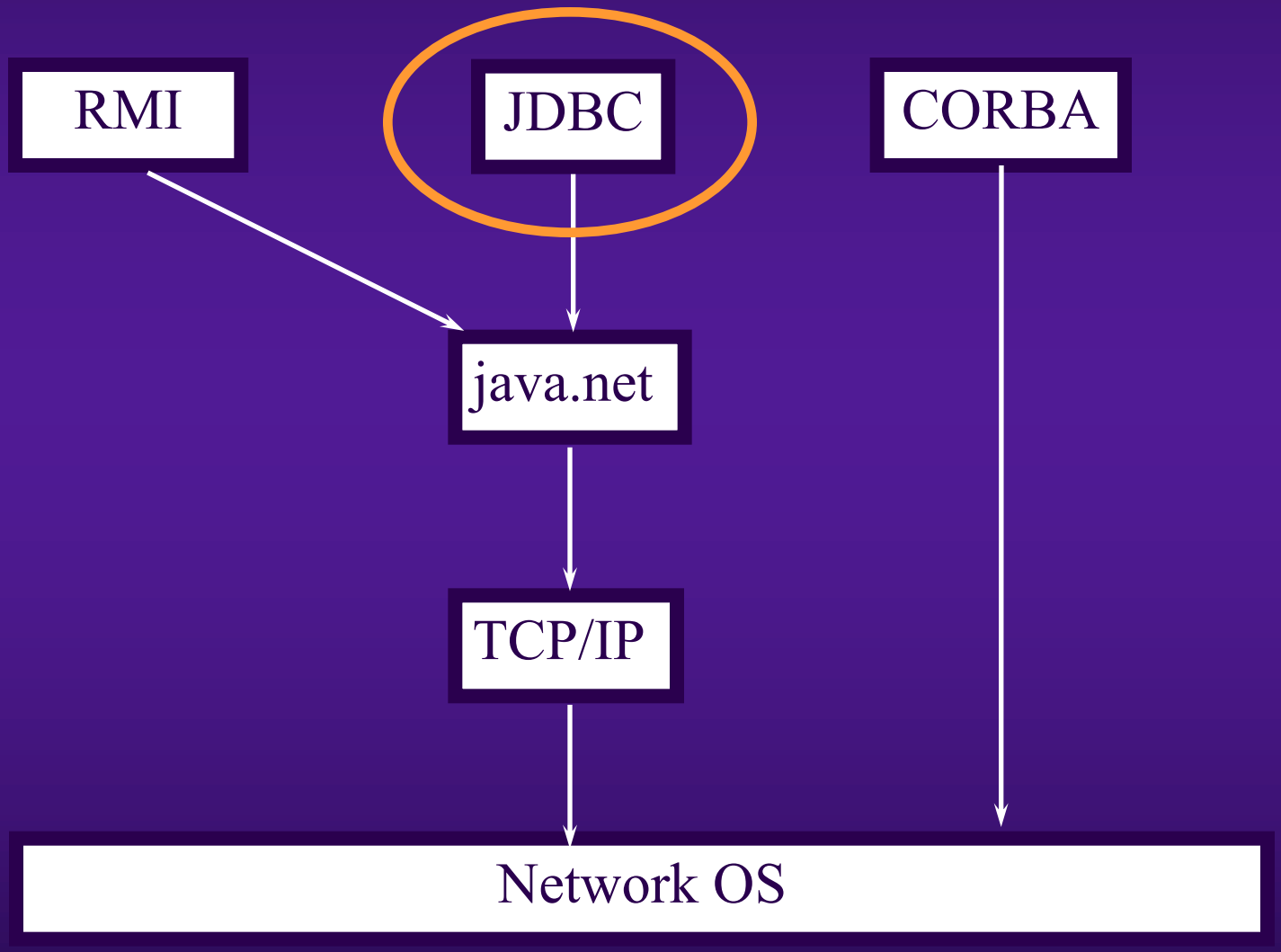
JDBC and Database Programming in Java

Agenda

- Overview of Databases and Java
- Overview of JDBC
- JDBC APIs
- Other Database Techniques

Overview



Why Java?

- Write once, run anywhere
  - Multiple client and server platforms
- Object-relational mapping
  - databases optimized for searching/indexing
  - objects optimized for engineering/flexibility
- Network independence
  - Works across Internet Protocol
- Database independence
  - Java can access any database vendor
- Ease of administration
  - zero-install client

Database Architectures

- Two-tier
- Three-tier
- N-tier

Database Technologies

- Hierarchical
  - obsolete (in a manner of speaking)
  - any specialized file format can be called a hierarchical DB
- Relational (aka SQL) (RDBMS)
  - row, column
  - most popular
- Object-relational DB (ORDBMS)
  - add inheritance, blobs to RDB
  - NOT object-oriented -- “object” is mostly a marketing term
- Object-oriented DB (OODB)
  - data stored as objects
  - high-performance for OO data models

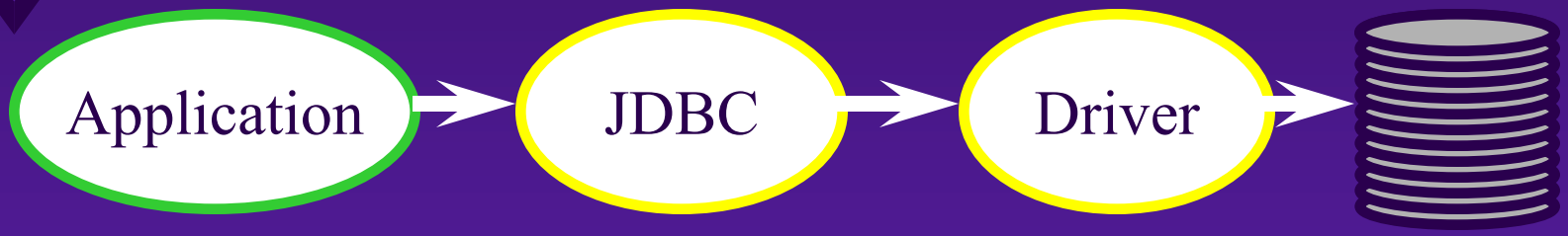
Transactions

- Transaction = more than one statement which must all succeed (or all fail) together
- If one fails, the system must reverse all previous actions
- Also can't leave DB in inconsistent state halfway through a transaction
- COMMIT = complete transaction
- ROLLBACK = abort

JDBC Goals

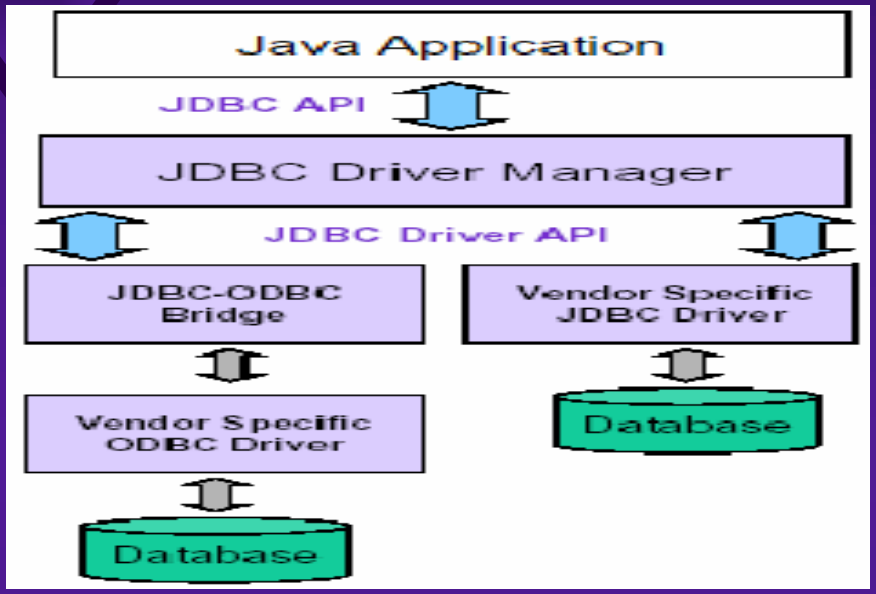
- SQL-Level
- 100% Pure Java
- Keep it simple
- High-performance
- Leverage existing database technology
  - why reinvent the wheel?
- Use strong, static typing wherever possible
- Use multiple methods to express multiple functionality

JDBC Architecture



- Java code calls JDBC library
- JDBC loads a *driver*
- Driver talks to a particular database
- Can have more than one driver -> more than one database
- Ideal: can change database engines without changing any application code

General Architecture



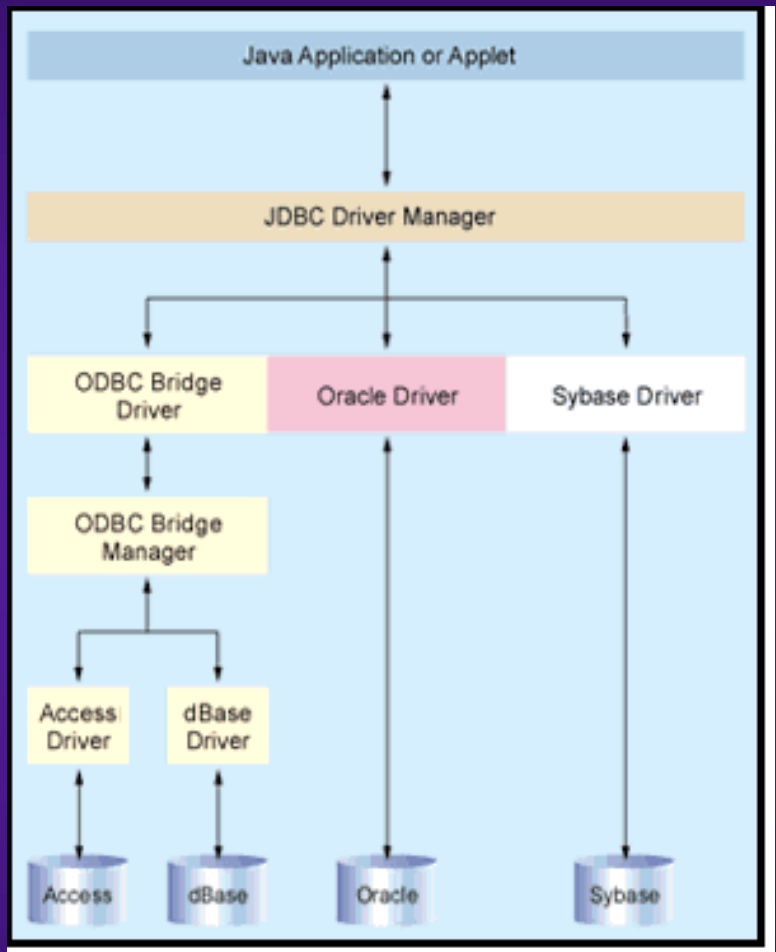
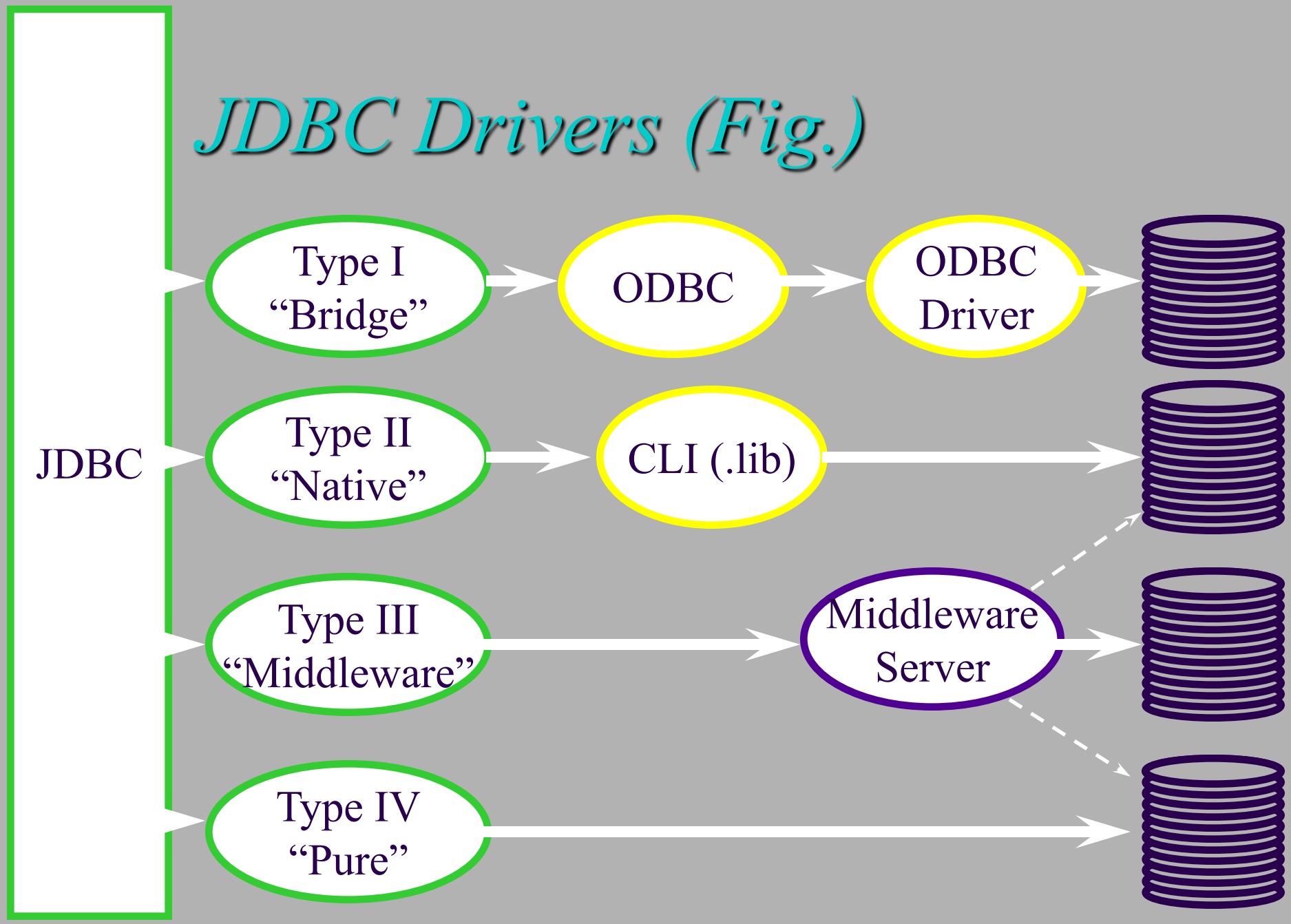


Figure 1. Anatomy of Data Access. The Driver Manager provides a consistent layer between your Java app and back-end database. JDBC works natively (such as with the Oracle driver in this example) or with any ODBC datasource.

JDBC Drivers

- Type I: “Bridge”
- Type II: “Native”
- Type III: “Middleware”
- Type IV: “Pure”

JDBC Drivers (Fig.)



JDBC Limitations

- No scrolling cursors
- No bookmarks

java.sql Package

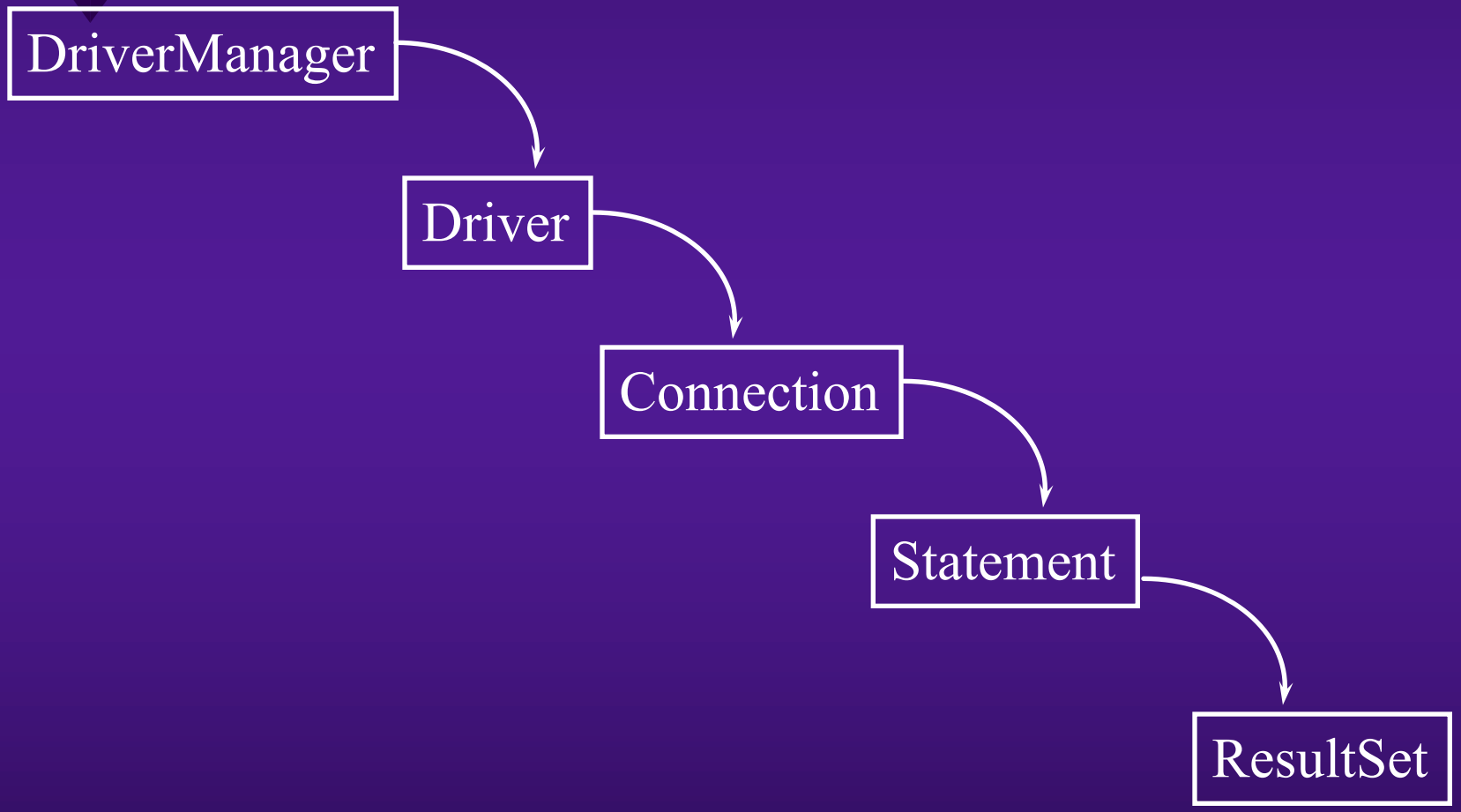
Jdbc Classes

- DriverManager
  - Manages JDBC Drivers
  - Used to Obtain a connection to a Database
- Types
  - Defines constants which identify SQL types
- Date
  - Used to Map between java.util.Date and the SQL DATE type
- Time
  - Used to Map between java.util.Date and the SQL TIME type
- TimeStamp
  - Used to Map between java.util.Date and the SQL TIMESTAMP type

JDBC Interfaces

- Driver
  - All JDBC Drivers must implement the Driver interface. Used to obtain a connection to a specific database type
- Connection
  - Represents a connection to a specific database
  - Used for creating statements
  - Used for managing database transactions
  - Used for accessing stored procedures
  - Used for creating callable statements
- Statement
  - Used for executing SQL statements against the database

JDBC Class Usage



Using JDBC

- To execute a statement against a database, the following flow is observed
  - Load the driver (Only performed once)
  - Obtain a Connection to the database (Save for later use)
  - Obtain a Statement object from the Connection
  - Use the Statement object to execute SQL. Updates, inserts and deletes return Boolean. Selects return a ResultSet
  - Navigate ResultSet, using data as required
  - Close ResultSet
  - Close Statement
- Do NOT close the connection
  - The same connection object can be used to create further statements
  - A Connection may only have one active Statement at a time. Do not forget to close the statement when it is no longer needed.
  - Close the connection when you no longer need to access the

Loading a Driver

- How does one load a "class" into the Virtual machine?
  - Use the static method Class.forName()

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

- A connection is obtained in the following manner:

```
Connection aConnection =
   DriverManager.getConnection("jdbc:odbc:myDatabase");
```

- Overloaded versions of the getConnection method allow the specification of a username and

MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP"; String USER = "username"; String PASS
= "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 Methods

`Statement createStatement()`

- returns a new Statement object

`PreparedStatement prepareStatement(String sql)`

- returns a new PreparedStatement object

`CallableStatement prepareCall(String sql)`

- returns a new CallableStatement object

- Why all these different kinds of statements?
  - Optimization.

Statement

- 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.

- Why all these different kinds of queries? Optimization.

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

SELECT Example

```
import java.sql.*;
public class classname {
public static void main(String args[]) {
String url = "jdbc:odbc:dsn";
Connection con;
Statement st;
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver" );
con = DriverManager.getConnection(url,
  "alex", "8675309");
st = con.createStatement();
ResultSet results = st.executeQuery("SELECT
  EmployeeID, LastName, FirstName FROM
  Employees");
```

SELECT Example (Cont.)

```
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

Executing Dynamic SQL Queries

```
import java.sql.*;
    public class JdbcPreparedstatementExample {
      static private final String driver =
    "sun.jdbc.odbc.JdbcOdbcDriver";
      static private final String connection = "jdbc:odbc:emp";
      public static void main(String args[]) {
      Connection con = null;
      PreparedStatement pst = null;
      ResultSet rs = null;
      try {
      Class.forName(driver);
      con = DriverManager.getConnection(connection);
```

```
String sql =  "select * from Employees where FirstName " + "in(?,?,?)";
        pst = con.prepareStatement(sql);
        pst.setString(1, "komal");
        pst.setString(2, "ajay");
        pst.setString(3, "santosh");
        rs = pst.executeQuery();
        while (rs.next()) {
        System.out.printrs.getString(1));
        System.out.printrs.getString(2));
        System.out.printlnrs.getString(3));
        }
          } catch (Exception e) {
        System.out.println(e);
        }  }}
```

Executing Stored Procedures

```
CallableStatement cs2 = con.prepareCall("{call
    ADDITION(?,?,?)}");
cs2.registerOutParameter(3,java.sql.Types.INTEGER);
            cs2.setInt(1,10);
            cs2.setInt(2,25);
            cs2.execute();
            int res = cs2.getInt(3);
            System.out.println(res);
CREATE OR REPLACE PROCEDURE getEmpName
    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR)
    AS BEGIN SELECT first INTO EMP_FIRST FROM
    Employees WHERE ID = EMP_ID; END;
```

Modifying the Database

- use executeUpdate if the SQL contains “INSERT” or “UPDATE”
- Why isn’t it smart enough to parse the SQL? Optimization.
- executeUpdate returns the number of rows modified
- executeUpdate also used for “CREATE TABLE” etc. (DDL)

Transaction Management

- The atomicity of a transaction means that if any part of it fails, the entire transaction is aborted. So the transaction is committed only if each part of it executes successfully.
- The isolation property ensures that transactions don't interfere with each other's processing.
- The durability of transactions ensures that all changes made to the databases by a transaction are permanent once that transaction is committed.
- The consistency of a transaction ensures that when it is committed, all relevant databases are in a consistent state.

Transaction Management

- Transactions are not explicitly opened and closed
- Instead, the connection has a state called *AutoCommit* mode
- if *AutoCommit* is true, then every statement is automatically committed
- default case: true

setAutoCommit

Connection.setAutoCommit(boolean)

- if *AutoCommit* is false, then every statement is added to an ongoing transaction
- you must explicitly commit or rollback the transaction using Connection.commit() and Connection.rollback()

- try{ //Assume a valid connection object conn
  conn.setAutoCommit(false);

Statement stmt = conn.createStatement();

String SQL = "INSERT INTO Employees " + "VALUES (106, 20, 'Rita', 'Tez')";

stmt.executeUpdate(SQL);

//Submit a malformed SQL statement that breaks String SQL = "INSERTED IN Employees " + "VALUES (107, 22, 'Sita', 'Singh')";

stmt.executeUpdate(SQL); // If there is no error.
conn.commit();

}catch(SQLException se)

{ // If there is any error. conn.rollback(); }

- Batch Processing allows to group related SQL statements into a batch and submit them with one call to the database.
- The **addBatch()** method of *Statement, PreparedStatement,* and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method
- Statement stmt = conn.createStatement();
- // Set auto-commit to false conn.setAutoCommit(false);
- // Create SQL statement String SQL = "INSERT INTO Employees (id, first, last, age) " + "VALUES(200,'Zia', 'Ali', 30)";
- // Add above SQL statement in the batch. stmt.addBatch(SQL);
- // Create one more SQL statement String SQL = "INSERT INTO Employees (id, first, last, age) " + "VALUES(201,'Raj', 'Kumar', 35)";
- // Add above SQL statement in the batch. stmt.addBatch(SQL);
- // Create one more SQL statement String SQL = "UPDATE Employees SET age = 35 " + "WHERE id = 100";
- // Add above SQL statement in the batch. stmt.addBatch(SQL);
- // Create an int[] to hold returned values int[] count = stmt.executeBatch();
- //Explicitly commit statements to apply changes conn.commit();

Javax package

- Javax.sql.*Automatic driver loading
- Exception handling improvements
- Enhanced BLOB/CLOB functionality
- Connection and statement interface enhancements
- XML datatype support

- Connection Pooling is a Server-side technology
- It is extremely expensive to open and close database connections every time a user requests a connection to carry out a SQL statement.
- With connection pooling, the application server maintains a pre-defined pool of “open” database connections that are shared between application clients
- When the client creates a connection via a DataSource object, the connection will come out of the pool.
- When a client closes a connection, the connection is returned back to the pool.

```
import java.sql.Connection;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

Connection conn = null;

try {
  Context        ctx          = new InitialContext();
  DataSource     ds           = (DataSource) ctx.lookup("jdbc/MurachBooks");
  Connection conn             = ds.getConnection( "myLogin", "myPassword");
}
catch (Exception e) { e.printStackTrace(); }

try {
  PreparedStatement ps = conn.prepareStatement(  “select * from file”  );
  ResultSet rs = ps.executeQuery();
  while (rs.next()) { ... }
}
```

- javax.sql.RowSet

JDBC Class Diagram