JDBC

Pradeep LN

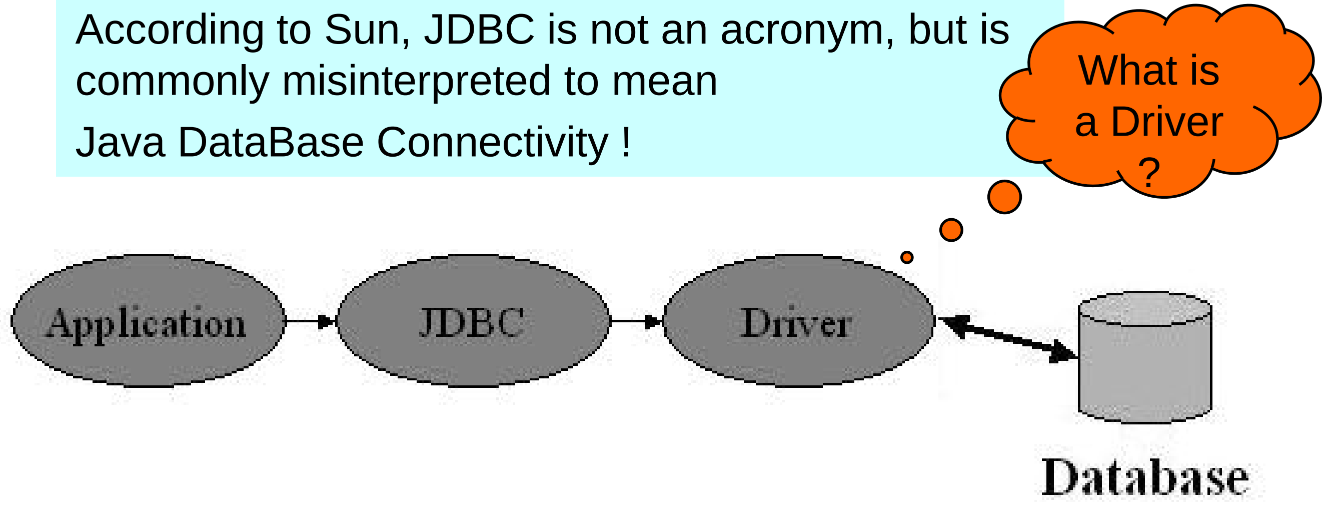
Topics

- What is JDBC?
- JDBC Driver
- Why JDBC?
- Various driver types.
- Usage of interfaces in JDBC
- Various ways of loading the driver
- Connecting to database.
- The connection URL
- Connection
- Statement
- Relationship between connection and statement
- ResultSet and its methods.
- Metadata.
- Prepared statements.
- Parameterizing Prepared statements
- Callable Statements.
- Transactions.

What is JDBC?

According to Sun, JDBC is not an acronym, but is commonly misinterpreted to mean

Java DataBase Connectivity !



- JDBC is a Java API for executing SQL statements.
- It consists of a set of classes and interfaces written in the Java programming language.
- This allows Java programs to interact with any SQL-compliant database.
- JDBC was developed by JavaSoft, a subsidiary of Sun Microsystems.

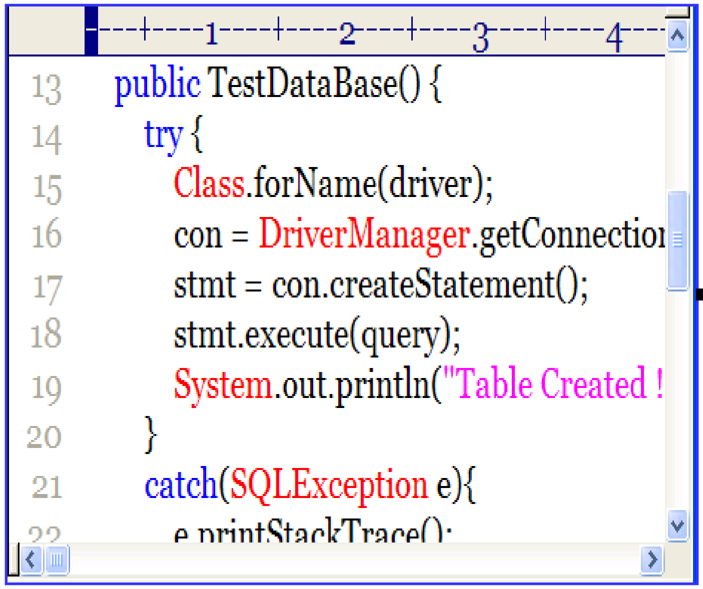
What is a Driver ?

- A driver acts like a translator between a device and programs that use the device.

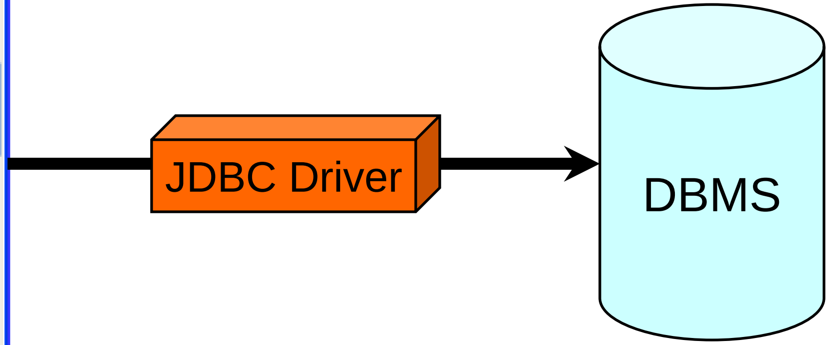
- Example
  - Driver for Sony Handycam
  - Driver for HP Printer

- Each device has its own set of specialized commands that only its driver knows.
  - In contrast, most programs access devices by using generic commands.
  - The driver, therefore, accepts generic commands from a program and then translates them into specialized commands for the device.

JDBC Driver



```
public TestDataBase() {
   try {
      Class.forName(driver);
      con = DriverManager.getConnectio
      stmt = con.createStatement();
      stmt.execute(query);
      System.out.println("Table Created !
   }
   catch(SQLException e){
      e.printStackTrace();
```

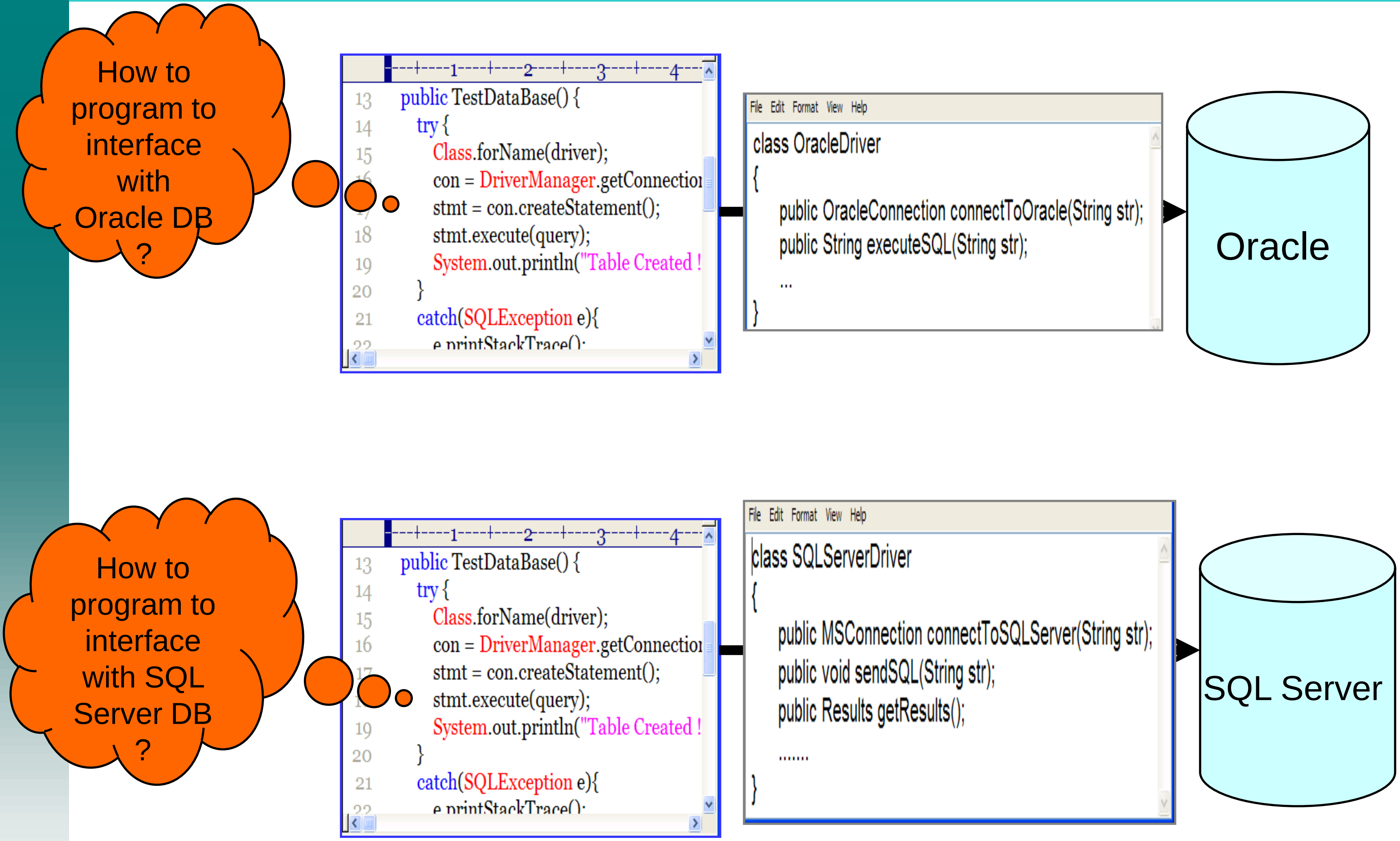


- JDBC Driver acts as a bridge between the Java application and the Database.
- It translates JDBC calls to vendor specific calls.

JDBC and ODBC

- ODBC, which is a Microsoft technology, pre-dates the JDBC technology.
- Introduced the concept of drivers.
- However, ODBC is a C API
  - Not Object Oriented
  - Uses Pointers
  - Other dangerous programming constructs
- ODBC needs to be installed on clients machine.
- Since the JDBC driver is written completely in Java, JDBC driver is automatically installable, portable, and secure on all Java platforms.

Why JDBC ?


How to program to interface with Oracle DB ?
Oracle
How to program to interface with SQL Server DB ?
SQL Server

```
public TestDataBase() {
  try {
    Class.forName(driver);
    con = DriverManager.getConnectio
    stmt = con.createStatement();
    stmt.execute(query);
    System.out.println("Table Created !
  }
  catch(SQLException e){
    e.printStackTrace();
```

```
class OracleDriver
{
    public OracleConnection connectToOracle(String str);
    public String executeSQL(String str);
    ...
}
```

```
public TestDataBase() {
  try {
    Class.forName(driver);
    con = DriverManager.getConnectio
    stmt = con.createStatement();
    stmt.execute(query);
    System.out.println("Table Created !
  }
  catch(SQLException e){
    e.printStackTrace();
```

```
class SQLServerDriver
{
    public MSConnection connectToSQLServer(String str);
    public void sendSQL(String str);
    public Results getResults();
    .......
}
```

JDBC is a Standard

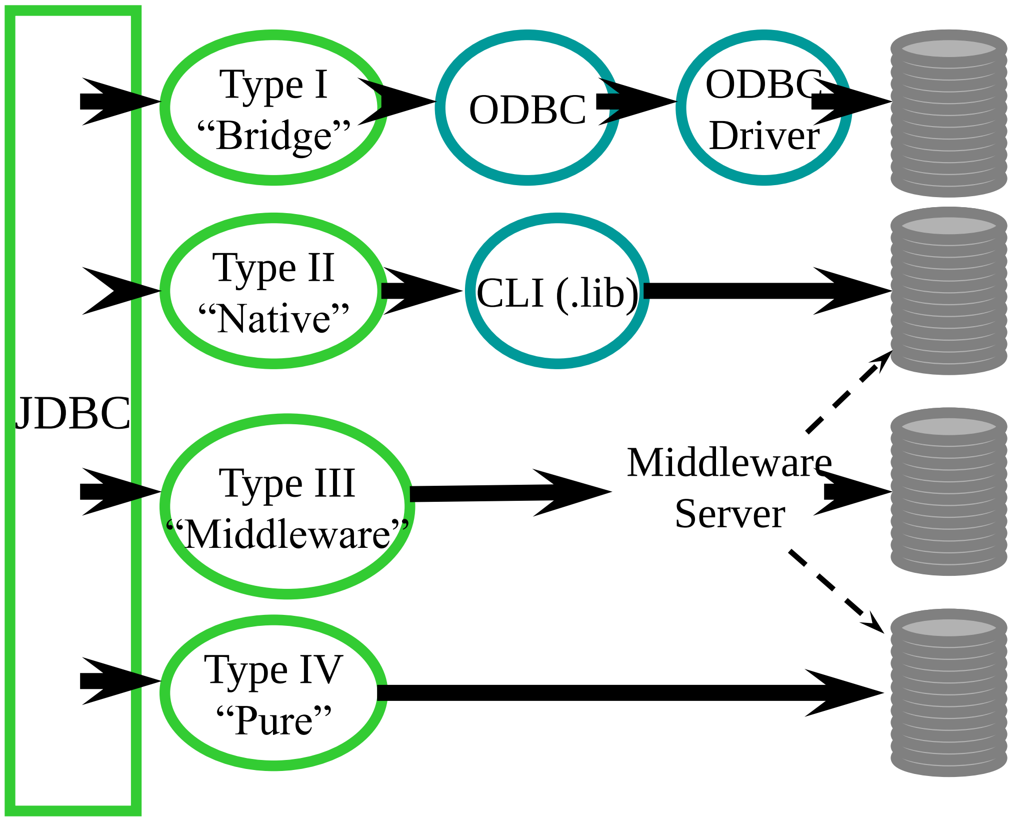
- JDBC defines a standard way of interfacing with different relational databases.
- JDBC is a set of specifications that different database driver vendors abide by.
- This makes the developer’s job a lot easier.
  - The developers need to learn a single unified API.
- JDBC enables portability of applications across different databases.

Usage of Interfaces in JDBC

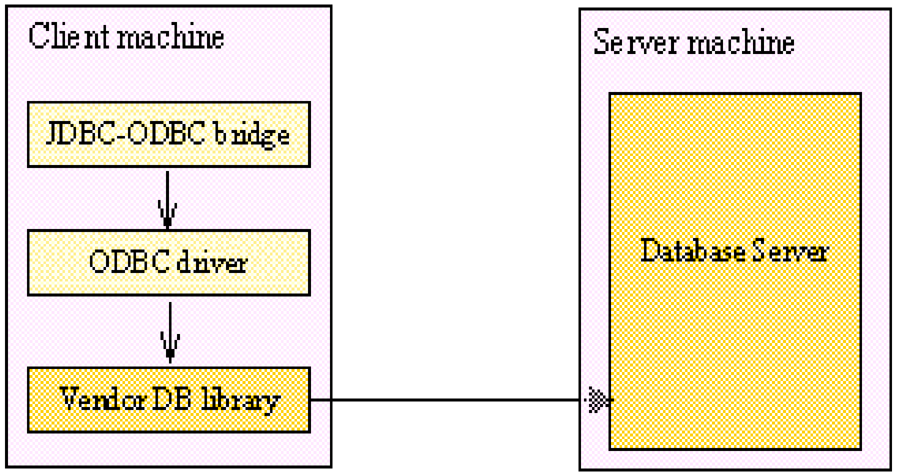
- The JDBC API defines a set of Interfaces.
  - Each interface is implemented differently by individual vendors
- Every vendor provides a set of classes that implement the JDBC interfaces for a particular database engine
  - This is made available as the JDBC driver for that particular database.
  - So, a JDBC driver is nothing but one particular implementation of the JDBC interfaces.
- Implementation of these underlying driver classes is completely abstracted from the developers.
  - The objective of JDBC is to hide the specifics of the underlying database

JDBC Driver Types

- JDBC drivers have evolved over a period of time, to enable easier interfacing with the database.



Type I Drivers - JDBC-ODBC

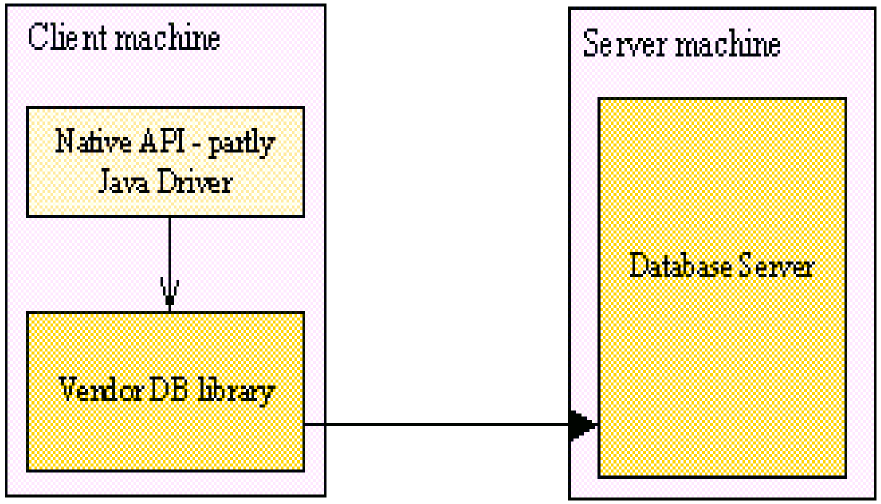


- The JDBC ODBC bridge driver maps every JDBC call to its equivalent ODBC call.
- Introduced as a stop-gap arrangement, till the time JDBC as a technology became widespread.
- Is made available along with JDK installation.

Type I Drivers - JDBC-ODBC

- **Pros**
  - The JDBC-ODBC Bridge allowed access to almost any database, since the database's ODBC drivers were already available.
- **Cons**
  - Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native db connectivity interface.
  - Constrained by the capabilities of the ODBC driver.
  - The ODBC driver needs to be installed on the client machine.
    - Considering the client-side software needed, this might not be suitable for applets

Type II Drivers - native drivers

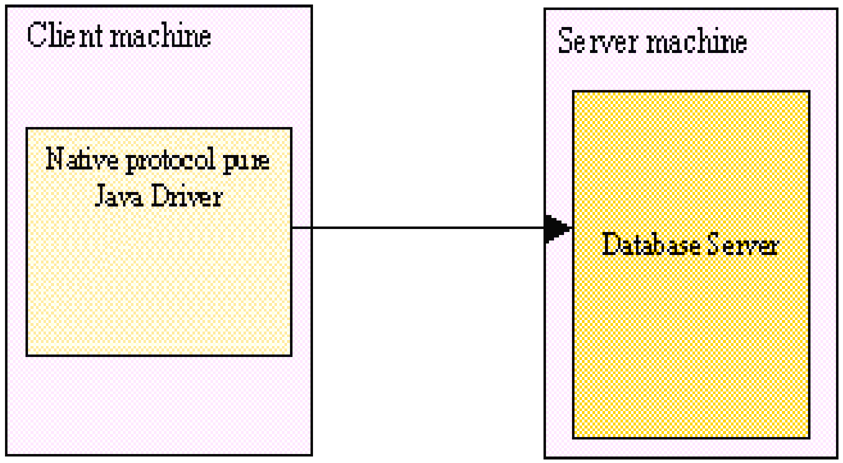


- Native API drivers
  - Uses Java Native Interface.
- Does not rely on ODBC driver
- Converts JDBC calls into database-specific calls
  - Used to leverage existing CLI (Call Level Interface) libraries.

Type II Drivers - native drivers

- **Pros**
  - Type 2 drivers typically offer significantly better performance than the JDBC-ODBC Bridge.
- **Cons**
  - The vendor client library needs to be installed on the client machine.
  - Cannot be used in internet due the client side software needed.
  - Usually not thread safe.

Type IV Drivers - direct to db

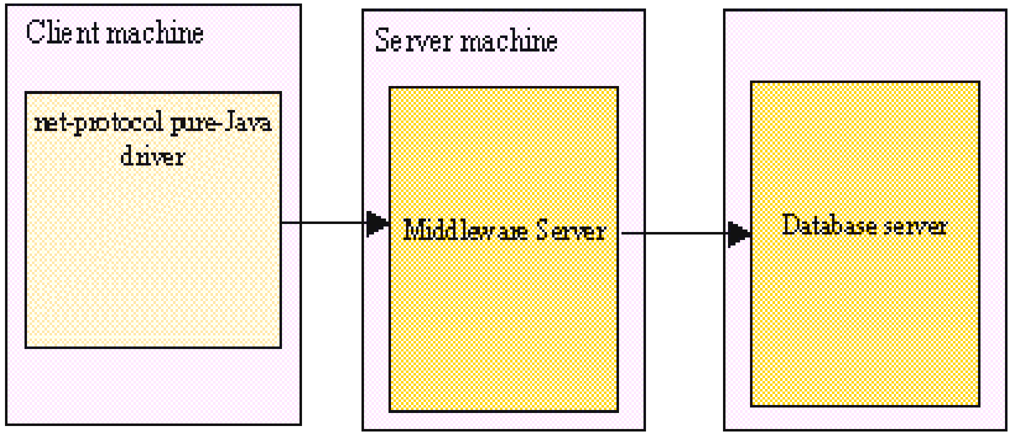


- JDBC calls are directly translated to vendor specific database calls and sent across the network.
- Uses Java networking libraries to talk directly to database engines.
- Almost all database vendors now provide Type IV drivers.

Type IV Drivers - direct to db

- **Pros**
  - Performance is quite good as compared to type I, and type II drivers.
  - There is no need to install special software on the client or server.
    - These drivers can be downloaded dynamically.
- **Cons**
  - With type 4 drivers, the user needs a different driver for each database.

Type III Drivers - DB middleware



- Follows a three tier communication approach.
- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol
    - The net server translates this request into database commands for that database.
- Thus the client driver to middleware communication is database independent.

Type III Drivers - DB middleware

Pros:

- Since the communication between client and the middleware server is database independent, there is no need for the vendor db library on the client machine. Also the client to middleware need not be changed for a new database.
- The Middleware Server (Can be a full fledged J2EE Application server) can provide typical middleware services like caching (connections, query results, and so on), load balancing, logging, auditing etc..
- Can be used in internet since there is no client side software needed.
- At client side a single driver can handle any database. (It works provided the middleware supports that database!!)

Cons:

- Requires database-specific coding to be done in the middle tier.
- An extra layer added may result in a time-bottleneck. But typically this is overcome by providing efficient middleware services described above.

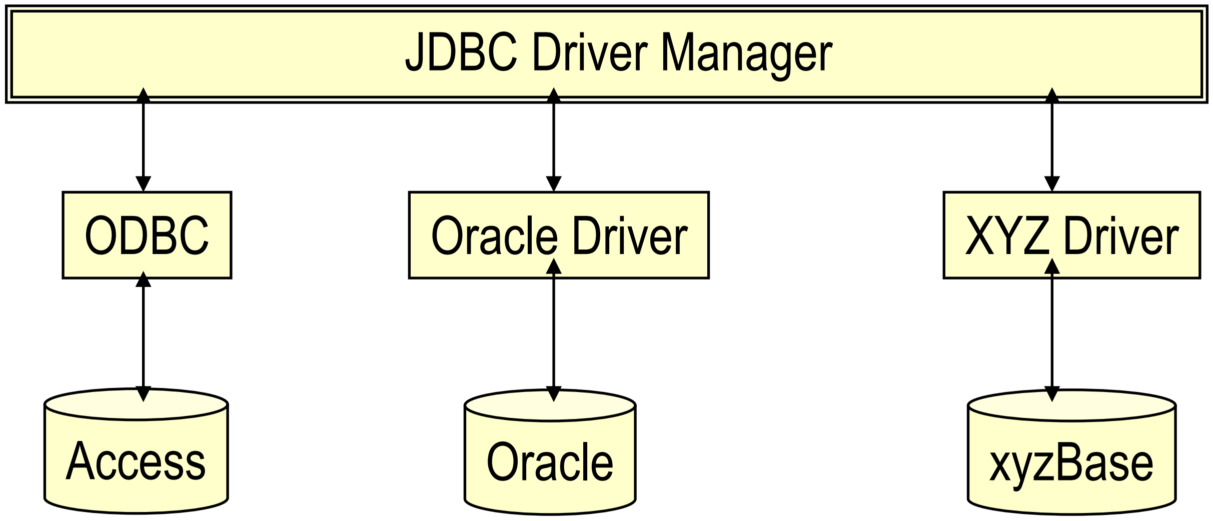
JDBC API

- To use JDBC you need
  - Database server …Access
  - Database driver …JDBC_ODBC Bridge
- Package
  - java.sql.*;
- Classes & Interfaces
  - DriverManager Class
  - Connection Interface
  - Statement Interface
  - ResultSet Interface
  - ResultSetMetaData Interface
  - DatabaseMetaData Interface

Note that with the exception of DriverManager class, all are interfaces in JDBC API

JDBC Driver Manager.

- It is possible that a Java application may need to talk to multiple databases, possibly from different vendors (especially if it is a distributed system)
- The JDBC driver manager provides the ability to communicate with multiple databases and keep track of which driver is needed for which database.
- Manages JDBC drivers loaded in memory



Loading the Driver

- The Driver class needs to be loaded in memory
  - DriverManager can then use the loaded Driver to establish a connection with the database.
  - Driver documentation provides the class name to use

- Driver can be loaded in 2 ways
  - Class.forName(driverClassName)
  - Setting the jdbc.drivers System property

Loading the Driver

- Driver class can be loaded in memory by invoking the below static method of the class called Class
  - forName(String driverClassName)
  - Provide the fully qualified class name (package.class)

```
// To load Type I bride driver
Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);

// To load Oracle database’s driver
Class.forName(“oracle.jdbc.driver.OracleDriver”);
```

Loading the Driver

- The driver class can be loaded by setting a System property – jdbc.drivers
  - System properties are set by using the
    -Dname=value option
  - System properties are available to all classes in the application (at the JVM level)
- Example
  - java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver Demo1
- DriverManager then retrieves the driver classes by calling
  - System.getProperty(“jdbc.drivers”);

Establishing Connection

- Once the appropriate driver is loaded, we can have the driver establish connection to the database.

- DriverManager provides a static method using which we can connect to a database
  - Connection getConnection(String url)
  - Connection getConnection(String url, String userName, String password)

Establishing Connection

- The connection url is of the form
  - **jdbc : sub protocol : connection details**

Primary protoco l (fixed)

Secondary protocol (available in driver documentation)

Includes IP address, port of the server machine, database name etc

- Example
  - DriverManager.getConnection(“jdbc : oracle:thin : @ 192.168.0.5:1521:pratian”)
  - DriverManager.getConnection(“jdbc : oracle:thin : @ 192.168.0.5:1521:pratian”,”scott”,”tiger”)
  - DriverManager.getConnection(“jdbc:odbc:datSrc”)

Establishing Connection

```
public class DBConnect
{
      private Connection conn;
      public Connection connect() {
            try {
                  Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
                  Connection conn = DriverManager.getConnection(“jdbc:odbc:myDSN”);
                   System.out.println(“Connection Established”);
            }
            catch(ClassNotFoundException e) {
                   e.printStackTrace();
             }
            catch(SQLException e) {
                   e.printStackTrace();
             }
      return conn;
      }
}
```

Connection Interface

- Encapsulates a connection to the database

- Important Methods
  - **void close()**
    - Closes the connection
  - **Statement createStatement()**
    - Creates a Statement object
    - Statement represents a simple SQL statement
  - **PreparedStatement prepareStatement()**
    - Creates a PreparedStatement object
    - Prepared statements are used to execute bulk of repetitive tasks efficiently
  - **CallableStatement prepareCall()**
    - Creates a CallableStatement object
    - Callable statements are used to execute stored procedures.

Statement Interface

- Represents an SQL statement that needs to be executed on the database server

- Important Methods
  - **ResultSet executeQuery()**
    - Executes SQL statement such as “SELECT” that queries a database
    - Data retrieved from the database on executing this query is returned as a ResultSet object.
    - Example

      ResultSet r = s.executeQuery("Select * from Employee");

  - **int executeUpdate()**
    - Executes SQL statement such as “INSERT”, “UPDATE”, or “DELETE” that modifies data in the database.
    - The number of rows affected is returned by the method.
    - Example

      stmt.executeUpdate(“insert into Employee values (‘James’, 24, 5545)”);

Statement Interface

- **boolean execute()**
  - Can be used to execute any SQL (DML, DDL, DQL)
    - Typically this method is used when the nature of SQL statement to be executed is not known.
  - The method returns a boolean value indicating whether or not a ResultSet object was create on executing the statement.
  - If the method returns true, then the ResultSet object can be retrieved using the method

    ResultSet getResultSet()

Example

```
public class ExecuteUpdateDemo
{
    public void insertIntoTable()
    {
            DBConnect db = new DBConnect();
            Connection con = db.connect();
            try
            {
                    Statement stmt = con.createStatement();
                    String query = "insert into Employee values (‘James’, 24, 5545)”;
                    stmt.executeUpdate(query);
                    System.out.println(“Values inserted !!!");
            }
            catch(SQLException e) {
                    e.printStackTrace();
            }
    }
}
```

ResultSet Interface

- On executing a ‘select’ query, the Statement object returns a ResultSet object
- ResultSet represents data retrieved on querying the database, in a tabular (row-column) format.
- ResultSet object maintains a pointer to a row within the tabular results called the cursor.
- Important Methods
  - boolean next()
    - Returns ‘true’ if next row exists in the result set, and moves cursor to the next row
  - int getRow()
    - Returns the current row number
  - String getString(int columnIndex)
    - Retrieves the value in the specified row as a String
  - int getInt(String columnName)
    - Retrieves the value in the specified row as an integer

ResultSetMetaData Interface

- ResultSetMetaData contains data about the ResultSet.
  - It consists of information like number of columns, column names, column types etc.

- ResultSet provides an object view of the records in the table, while information about the table itself can be retrieved through the ResultSetMetaData object.

- Important Methods
  - **int getColumnCount()**
    - Returns the number of columns in the ResultSet object
  - **String getColumnName(int index)**
    - Get the designated column's name.
  - **int getColumnType(int index)**
    - Retrieves the designated column's SQL type.
  - **int isNullable(int index)**
    - Indicates the nullability of values in the designated column.

Example

```
public class ExecuteQueryDemo
{
     public void readFromTable()
     {
          DBConnect db = new DBConnect();
          Connection con = db.connect();
          try
          {
               Statement stmt = con.createStatement();
               String query = “select * from Employee ”;
               ResultSet rs = stmt.executeQuery(query);
               displayResults(rs);
          }
          catch(SQLException e) {
               e.printStackTrace();
          }
     }
```

Example contd…

```
    static void displayResults(ResultSet rs) throws SQLException
    {
        ResultSetMetaData rMeta = rs.getMetaData();
        int noOfCols = rMeta.getColumnCount();

        for(int i=1; i<=noOfCols; ++i)
                System.out.print(rMeta.getColumnName(i) + "\t\t");
        System.out.println("");

        while(rs.next())
        {
                for(int i=1; i<=noOfCols; ++i)
                        System.out.print(rs.getString(i) + "\t\t");
                System.out.println("");
        }
    }
}
```

Exercise

- Write a program to accept an SQL statement on command line and have the query executed on the database.

More about ResultSet

- The ResultSet maintains a connection to the database and fetches the rows on demand.
- If a query returns about 1000 records, it is not feasible to have them all represented as a ResultSet object in memory
- ResultSets have a default ‘fetch size’ that can be altered
  - This can be done by calling the method setFetchSize(int rows) on ResultSet

Connection, Statement, ResultSet

- Statement and ResultSet have a one-one relationship.
  - Only one ResultSet object can be active at a time, for a Statement object.
  - Use separate Statement objects to return multiple result sets.
- ResultSet object is automatically closed when the Statement object that generated it is closed
- When finished processing, a program should close both the Statement and the Connection.

Exercise

- Write a class AccountDAO with methods
  - withdraw
  - deposit
  - getBalance

ResultSet Enhancements

- In JDBC 1.0(prior to Java1.2), resultsets were
  - forward only
  - non-updatable
  - static
- From JDBC 2.0(Java 1.2 onwards) result sets
  - Support Multidirectional traversing
    - Can randomly position the cursor to point to any row in the resultset
  - are Updatable
    - Changes made to values in the resultset object will be reflected in the underlying database.
  - are Dynamic

Creating Dynamic ResultSets

- The createStatement method of Connection has been overloaded to take two additional parameters
  - Statement createStatement(int resultSetType, int resultSetConcurrency)
- The first parameter indicates direction and dynamism.
  - Can take one of the three constants defined

    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.TYPE_SCROLL_INSENSITIVE
- The second parameter indicates concurrency mode
  - Can take one of the constants defined

    ResultSet.CONCUR_READ_ONLY

    ResultSet.CONCUR_UPDATABLE

Creating Dynamic ResultSets

- Example

```
Statement stmt = con.createStatement (
                        ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);


ResultSet rs = stmt.executeQuery("SELECT a, b FROM
                                    TABLE2");
```

Traversing the ResultSet

- More methods in ResultSet to facilitate multi directional traversing
  - **boolean previous()**
    - Moves the cursor backwards by one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.
  - **first()**
    - Moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.
  - **last()**
    - Moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.
  - **relative(int rows)**
    - Moves the cursor relative to its current position.
  - **absolute(int row)**
    - positions the cursor on the row-th row of the ResultSet object.

Manipulating updatable ResultSets

- ResultSets comprise of data stored in database tables.
- This data can be modified through the ResultSet object.
  - The ResultSet will then commit the changes in the database, and data will be updated in the underlying database.
- Dynamic ResultSets help to reduce dependency on SQL stmts.
- A record can be modified, deleted or a new record inserted through the ResultSet object.

Modifying Records

- Example:

```
rs.absolute(5);
// moves the cursor to fifth row
rs.updateString("NAME", "AINSWORTH");
rs.updateInt("AGE",24);
// updates the  NAME column of row 5 to be AINSWORTH &
// that of age to 24
rs.updateRow();
// updates the row in the underlying database
```

Inserting records

- New records can be added to the table through the ResultSet object.
  - Not only can we modify existing records, but insert an entirely new row into the table
  - This eliminates the need to execute a SQL to insert the record.
- Position the cursor to a buffer or imaginary row called InsertRow by calling the method

  moveToInsertRow()
- Use the updateXXX() methods to populate various fields
- Finally call the **insertRow()** method
  - A call to this method actually inserts data into the table.

Inserting Record

- Example:

```
rs.moveToInsertRow();
rs.updateString(1,"AINSWORTH");
rs.updateInt(2,35);
rs.updateBoolean(3,true);
rs.insertRow();
rs.moveToCurrentRow();
```

Types of Statement

- JDBC offers three statement types
  - Statement
    - Represents a basic SQL statement
  - Prepared Statement
    - Represents a precompiled SQL statement, which can offer improved performance, especially for large/complex SQL statements
  - Callable Statement
    - Allows JDBC programs to access stored procedures

Prepared Statement

- A PreparedStatement is used for SQL statements that are executed multiple times with different values.
  - For instance, you might want to insert several values into a table, one after another.
- Extends from Statement
- Takes an SQL at creation time

- Example:

```
PreparedStatement pstmt = con.prepareStatement ("UPDATE
EMPLOYEES SET SALARY = ? WHERE ID = ?");
pstmt.setDouble(1, 153833.00)
pstmt.setInt(2, 110592)
pstmt.executeUpdate();
```

Repeat these steps for different set of values

Prepared Statement

- The advantage of PreparedStatement is that it is pre-compiled
  - Reduces the overhead of parsing SQL statements on every execution

- The strength of PreparedStatement is that we can use it over and over again with new parameter values, rather than having to create a new Statement object for each new set of parameters.
  - This approach is obviously more efficient, as only one object is created.

Prepared Statement

- Some Important Methods
  - **boolean execute()**
    - Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
  - **ResultSet executeQuery()**
    - Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.
  - **int executeUpdate()**
    - Executes the SQL statement in this PreparedStatement object, which must be an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that returns nothing, such as a DDL statement.
  - **void setBoolean(int parameterIndex, boolean x)**
  - **void setDouble(int parameterIndex, double x)**
  - **…………**
  - **void clearParameters()**
    - Clears the current parameter values immediately

Exercise

- CustomerDAO with CRUD methods for frequent updates to customer details

Callable Statement

- A CallableStatement object provides a way to call stored procedures that execute on the data base server end.
- A sample stored procedure written in PLSQL for Oracle database would like this

```
CREATE PROCEDURE MYLIBRARY.SQLSPEX2
     + "(IN P1 INTEGER, OUT P2 INTEGER,
     INOUT P3 INTEGER) “
      +   "LANGUAGE SQL SPECIFIC
     MYLIBRARY.SQLSPEX2 "
     + "EX2: BEGIN "
     + "   SET P2 = P1 + 1; "
     + “ SET P3 = P3 + 1; "
     +  "END EX2 ";
```

Stored Procedures follow a programming language syntax and are stored on the DB server end

Callable Statement

- CallableStatement is a sub-interface of PreparedStatement
- The syntax for invoking a stored procedure
  - **{call procedure_name}**
- The syntax for invoking a stored procedure which takes arguments
  - **{call procedure_name(?, ?, ...)}**
    - The values entered as input are referred to as IN parameters
    - The return value for the stored procedure is referred to as the OUT parameter
    - Some parameters can serve as both input and output parameters and are referred to as INOUT parameters

Callable Statement

- Example: With IN parameters

```
CallableStatement stmt = con.prepareCall ("{ call insert_employee(?, ?
                                              )}");
stmt.setString(1, "999999999");
stmt.setString(2, "John");
stmt.execute();
```

- Example: With OUT parameters

```
CallableStatement cstmt = con.prepareCall( "{call getTestData(?, ?)}");
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.registerOutParameter(2, java.sql.Types.DOUBLE);
cstmt.executeQuery();
byte x = cstmt.getByte(1);
double d = cstmt.getDouble(2);
```

Transaction Processing

- JDBC provides a simple way of processing transactions.

- What is a transaction ?

  Transaction consists of a group of related database operations, executing together as a single unit

  - Either all the operations succeed or none of them succeed.

Transactions

- The classic example of a transaction is withdrawing money from one bank account and depositing it in another (fund transfer).

```
SELECT accountBalance INTO aBalance
    FROM Accounts WHERE accountId=aId;
IF (aBalance >= transferAmount) THEN
    UPDATE Accounts
        SET accountBalance = accountBalance - transferAmount
        WHERE accountId = aId;
    UPDATE Accounts
        SET accountBalance = accountBalance + transferAmount
        WHERE accountId = bId;
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (aId, -transferAmount);
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (bId, transferAmount);
ELSE
    FAIL "Insufficient funds in account";
END IF
```

What if the system fails after the first account is updated, but before the second?

Money might have left A's account, but not shown up in B's account

Transactions

- The classic example of a transaction is withdrawing money from one bank account and depositing it in another (fund transfer).

```
SELECT accountBalance INTO aBalance
    FROM Accounts WHERE accountId=aId;
IF (aBalance >= transferAmount) THEN
    UPDATE Accounts
        SET accountBalance = accountBalance - transferAmount
        WHERE accountId = aId;
    UPDATE Accounts
        SET accountBalance = accountBalance + transferAmount
        WHERE accountId = bId;
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (aId, -transferAmount);
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (bId, transferAmount);
ELSE
    FAIL "Insufficient funds in account";
END IF
```

What if the accounts are properly updated, but the account journal is not?

Activities on A and B's monthly bank statements would not be consistent with their account balances.

Transactions

- The classic example of a transaction is withdrawing money from one bank account and depositing it in another (fund transfer).

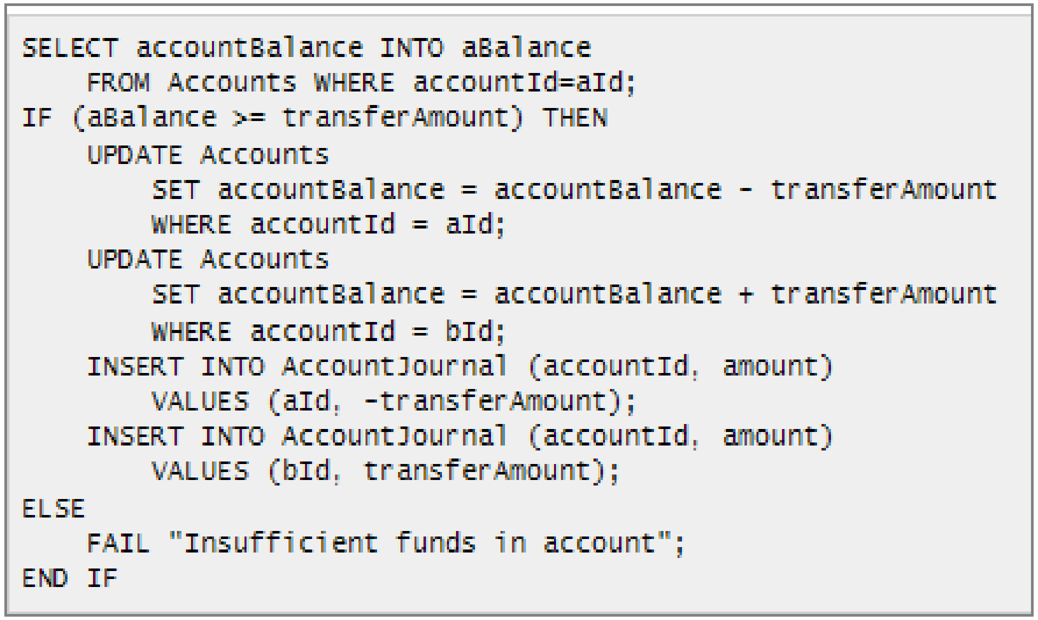
```
SELECT accountBalance INTO aBalance
    FROM Accounts WHERE accountId=aId;
IF (aBalance >= transferAmount) THEN
    UPDATE Accounts
        SET accountBalance = accountBalance - transferAmount
        WHERE accountId = aId;
    UPDATE Accounts
        SET accountBalance = accountBalance + transferAmount
        WHERE accountId = bId;
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (aId, -transferAmount);
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (bId, transferAmount);
ELSE
    FAIL "Insufficient funds in account";
END IF
```

What if transfer of funds is initiated twice, at exactly the same time ?

If account A has a balance of 10000, then two transfers of 10000 might go through, leading to inconsistencies.

Transactions

- The classic example of a transaction is withdrawing money from one bank account and depositing it in another (fund transfer).



```
SELECT accountBalance INTO aBalance
    FROM Accounts WHERE accountId=aId;
IF (aBalance >= transferAmount) THEN
    UPDATE Accounts
        SET accountBalance = accountBalance - transferAmount
        WHERE accountId = aId;
    UPDATE Accounts
        SET accountBalance = accountBalance + transferAmount
        WHERE accountId = bId;
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (aId, -transferAmount);
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (bId, transferAmount);
ELSE
    FAIL "Insufficient funds in account";
END IF
```

Grouping one or more SQL statements that make up a logical unit of work as a transaction, will help deal with the above discussed problems

Transaction Processing

- Transaction Processing using JDBC is managed at a single database connection level.
  - All transaction related methods are in Connection interface

- Transaction related methods in Connection interface
- **commit()**
  - Makes all changes made since the previous commit/rollback permanent
  - releases any database locks currently held by the Connection.
- **rollback()**
  - Drops all changes made since the previous commit/rollback
  - Releases any database locks currently held by this Connection.
- **setAutoCommit(boolean autoCommit)**
  - Sets this connection's auto-commit mode.

Transaction Processing

- Step 1: Turn off auto commit:
  - **conn.setAutoCommit(false);**
- Step 2: create and execute statements that form a logical transaction
- Step 3: Commit or rollback
  - if all succeeded

    conn.commit();
  - else, if one or more failed

    conn.rollback();
- Step 4: Optionally, turn auto commit back on
  - **conn.setAutoCommit(true);**

Exercise

- AccountDAO with another method transfer() that calls withdraw and deposit methods as part of a single transaction

Question time