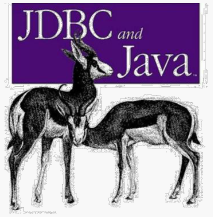
Introduction to SQL Programming Techniques

CSC 375, Fall 2016

The Six Phases of a Project:

Enthusiasm
Disillusionment
Panic
Search for the Guilty
Punishment of the Innocent
Praise for non-participants



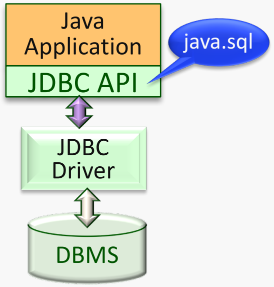
SQL in Application Code

- SQL commands can be called from within a host language (e.g., C++ or Java) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to *connect* to the right database.
- Two main integration approaches:
  - Embed SQL in the host language (Embedded SQL, SQLJ)
  - Create special API to call SQL commands (JDBC)

Database API Approaches

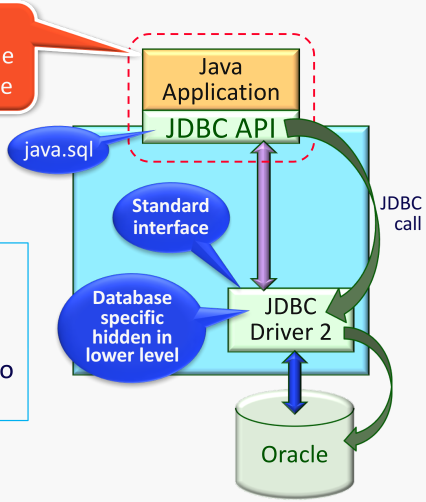
ODBC = Open DataBase Connectivity
JDBC = Java DataBase Connectivity

- JDBC is a collection of Java classes and interface that enables database access
- JDBC contains methods for
  - connecting to a remote data source,
  - executing SQL statements,
  - receiving SQL results
  - transaction management, and
  - exception handling
- The classes and interfaces are part of the java.sql package



Advantage of API Approach

Applications using ODBC or JDBC are DBMS-independent at the source code level and at the level of the executable



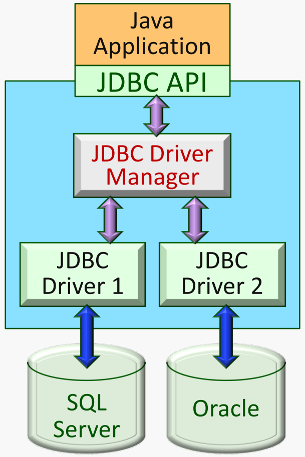
This is achieved by introducing an extra level of indirection

- A DBMS-specific “driver” traps the calls and translates them into DBMS-specific code

Driver Manager

Drivers are registered with a **driver manager**

- Drivers are loaded dynamically on demand
- The application can access several different DBMS's simultaneously



Java Application

JDBC API

JDBC Driver Manager

JDBC Driver 1

JDBC Driver 2

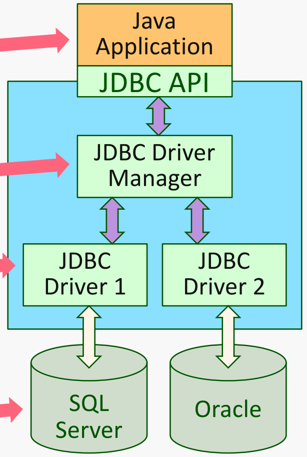
SQL Server

Oracle

JDBC: Architecture

Four architectural components:

- Application (initiates and terminates connections, submits SQL statements)
- Driver manager (loads JDBC driver and passes function calls)
- Driver (connects to data source, transmits requests and returns/translates results and error codes)
- Data source (processes SQL statements)



Java Application

JDBC API

JDBC Driver Manager

JDBC Driver 1

JDBC Driver 2

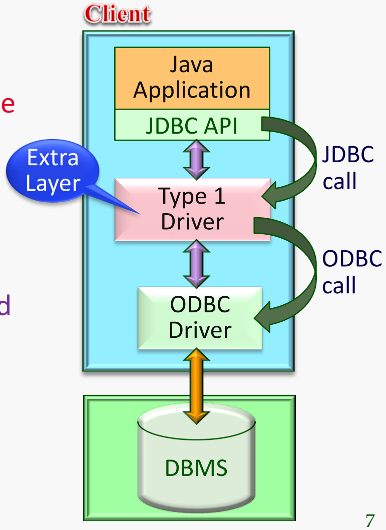
SQL Server

Oracle

JDBC: Four Types of Drivers (1)

Bridge:

- Translates JDBC function calls into function calls of another non-native API such as ODBC.
- The application can use JDBC calls to access an ODBC compliant data source.
- Advantage: no new drivers needed
- Disadvantage:
  - The additional layer affects performance
  - Client requires the ODBC installation
  - Not good for the Web



Client

Java Application

JDBC API

Extra Layer

Type 1 Driver

ODBC Driver

JDBC call

ODBC call

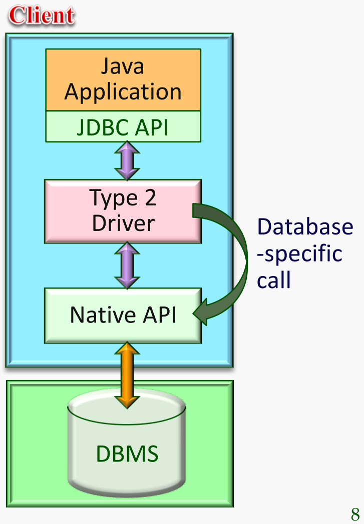
DBMS

JDBC: Four Types of Drivers (2)

Direct translation to native API via non-Java driver:

Convert JDBC calls into database-specific calls (e.g., Oracle native API)

- **Advantage**: Better performance
- **Disadvantage**:
  - Native API must be installed in client
  - Not good for the Web



Client

Java Application

JDBC API

Type 2 Driver

Native API

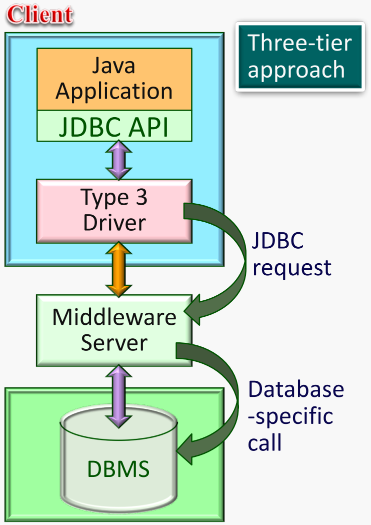
Database -specific call

DBMS

JDBC: Four Type of Drivers (3)

Network bridge:

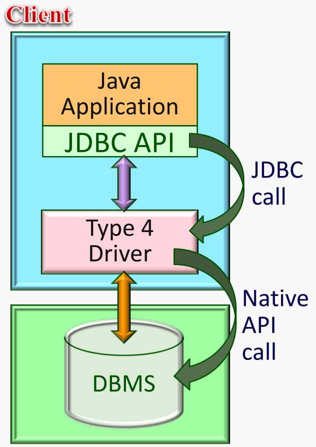
- The driver sends commands over the network to a middleware server
- The middleware server translates the JDBC requests into database-specific calls
- Advantage: Needs only small JDBC driver at each client
- Disadvantage: Need to maintain another server



JDBC: Four Type of Drivers (4)

Direct translation to the Native API via Java Driver:

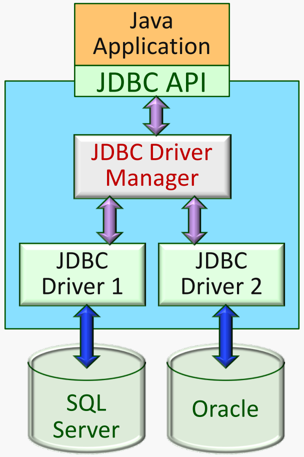
- The driver translates JDBC calls into the native API of the database system
- The driver uses java networking libraries to communicate directly with the database server (i.e., java sockets)
- Advantage:
  - Implementation is all Java
  - Performance is usually quite good
  - Most suitable for Internet access
- Disadvantage: Need a different driver for each database



JDBC Classes and Interfaces

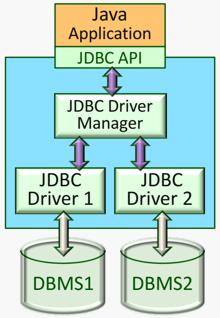
Steps to submit a database query:

1. Load the JDBC driver
2. Connect to the data source
3. Execute SQL statements



JDBC Driver Management

- DriverManager class:
  - Maintains a list of currently loaded drivers
  - Has methods to enable dynamic addition and deletion of drivers
- Two ways of loading a JDBC driver:
  1. In the Java code:
     Class.forName(“oracle/jdbc.driver.Oracledriver”);
     /* This method loads an instance of the driver class
  2. Enter at command line when starting the Java application:
     -Djdbc.drivers=oracle/jdbc.driver



Connections in JDBC

- We interact with a data source through sessions.
- A session is started through creation of a Connection object
- Each connection identifies a logical session with a data source
- Connections are specified through a URL that uses the jdbc protocol: jdbc:<subprotocol>:<otherParameters>

Example:

Host

Port

```
String url="jdbc:oracle:www.bookstore.com:3083";
Connection con;
try{
      con = DriverManager.getConnection(url,userId,password);
} catch(SQLException excpt)  { …}
```

Different drivers have slightly different URL formats – check the documentation

Discuss later

Connection Class Interface (1)

- void setTransactionIsolation(int *level*)
  Sets isolation level for the current connection
- public int getTransactionIsolation()
  Get isolation level of the current connection
- void setReadOnly(boolean b)
  Specifies whether transactions are read-only
- public boolean getReadOnly()
  Tests if transaction mode is read-only
- void setAutoCommit(boolean b)
  - *If autocommit is set, then each SQL statement is considered its own transaction.*
  - *Otherwise, a transaction is committed using commit(), or aborted using rollback().*
- public boolean getAutoCommit()
  Test if autocommit is set

Connection Class Interface (2)

- public boolean isClosed()
  Checks whether connection is still open.
- connectionname.close()
  Close the connection connectionname

Executing SQL Statements

- Three different ways of executing SQL statements:
  1. Statement (both static and dynamic SQL statements)
  2. PreparedStatement (semi-static SQL statements)
  3. CallableStatment (stored procedures)
- PreparedStatement class:

  Used to create precompiled, **parameterized SQL statements**
  - SQL structure is fixed
  - Values of parameters are determined at run-time

PreparedStatement

Place holder

```
String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatment pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4,age);

int numRows = pstmt.executeUpdate();
```

Connection name

Good style to always clear

Setting parameter values
sid, sname, rating, age are java variables

Number of rows modified

Use executeUpdate() when no rows are returned

ResultSet Example

- PreparedStatement.executeUpdate only returns the number of affected records
- PreparedStatement.executeQuery returns data, encapsulated in a ResultSet object
  - ResultSet is similar to a cursor
  - Allows us to read one row at a time
  - Intially, the ResultSet is positioned before the first row
  - Use next() to read the next row
  - next() returns false if there are no more rows

ResultSet Example

```
ResultSet  rs=pstmt.executeQuery(sql);
                        // rs is now a cursor
While (rs.next()) {
  // process the data
}
```

Common ResultSet Methods (1)

POSITIONING THE CURSOR	
next()	Move to next row
previous()	Moves back one row
absolute(int num)	Moves to the row with the specified number
relative(int num)	Moves forward or backward (if negative)
first()	Moves to the first row
Last()	Moves to the last row

Common ResultSet Methods (2)

RETRIEVE VALUES FROM COLUMNS	
getString(string columnName):	Retrieves the value of designated column in current row
getString(int columnIndex)	Retrieves the value of designated column in current row
getFloat (string columnName)	Retrieves the value of designated column in current row

Matching Java and SQL Data Types

SQL Type	Java class	ResultSet get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

SQL Data Types

BIT	A boolean value
CHAR(*n*)	A character string of fixed length *n*
VARCHAR(*n*)	A variable-length character string with a maximum length *n*
DOUBLE	A double-precision floating point value
FLOAT(*p*)	A floating point value with a precision value *p*
INTEGER	A 32-bit signed integer value
REAL	A high precision numeric value
DATE	A day/month/year value
TIME	A time of day (hour, minutes, second) value
TIMESTAMP	A day/month/year/hour/minute/second value

Statement – Another Way to Execute an SQL Statement

SQL coming

```
Statement   stmt = con.createStatement();
             // create an empty statement object
String      query = "SELECT name, rating
                      FROM Sailors";
ResultSet   rs = stmt.executeQuery(query);
```

Here is the SQL

Note: The query can be dynamically created

Review: Throwable Class

- Throwable class: is the **superclass** of all errors and exceptions in the Java language
- Throwable object: can have an associated message that provides more detail about the particular **error** or **exception** that is being thrown
- getMessage(): returns the **error message** string of the throwable object

JDBC: Exceptions

- Most of the methods in java.sql can throw an exception of type SQLException if an error occurs.
- SQLException has the following methods:
  - public String getMessage()

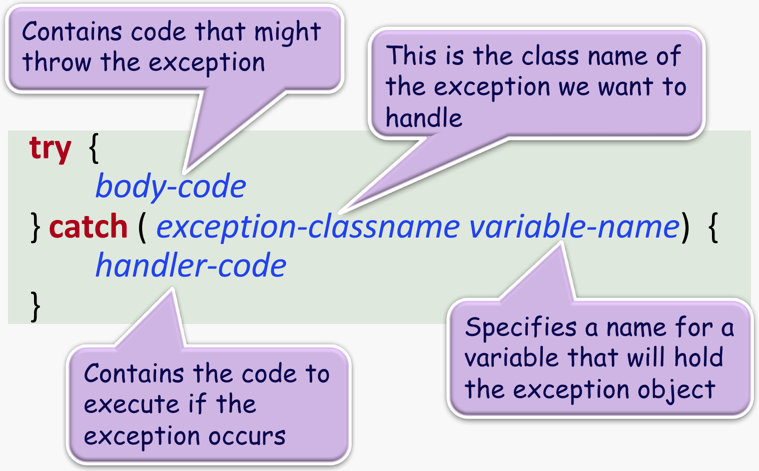
    is inherited from the Throwable class
  - public String getSQLState()

    returns an SQLState identifier according to SQL 99
  - public int getErrorCode()

    retrieves a vendor-specific error code
  - public SQLException getNextException()

    gets the next exception chained to this SQLException object

Catch the Exception



Contains code that might throw the exception

This is the class name of the exception we want to handle

```
try {
      body-code
} catch ( exception-classname variable-name)  {
      handler-code
}
```

Contains the code to execute if the exception occurs

Specifies a name for a variable that will hold the exception object

JDBC: Warnings

- SQLWarning is a subclass of SQLException.
- Warnings are not as severe. They are not thrown and their existence has to be explicitly tested.
  - getWarnings()

    retrieves SQL warning if they exist
  - getNextWarning()

    retrieves the warning chained to this SQLwarning object

Warning & Eception Example

```
try {
   stmt=con.createStatement();  // create an empty statement object
   warning=con.getWarnings();  // retrieve warning if it exists
   while(warning != null) {
      // handle SQLWarnings;
      warning = warning.getNextWarning():
              // get next warning chained to the warning object
   }
   con.clearWarnings();
   stmt.executeUpdate(queryString);
   warning = con.getWarnings();
   …
} //end try
catch( SQLException SQLe) {     // catch the SQLException object
   // handle the exception
}
```

Another Example

```
Connection con =                                                  // connect
   DriverManager.getConnection(url, "login", "pass");

Statement   stmt = con.createStatement();     // create and execute a query
String      query = "SELECT name, rating FROM Sailors";
ResultSet   rs = stmt.executeQuery(query);

try {
     while (rs.next()){                            // loop through result tuples
        String  s = rs.getString("name");          // get the attribute values
        Int      n = rs.getInt("rating");
        System.out.println(s + "   " + n);         // print name and rating
     }
} catch(SQLException  ex) {                        // handle exceptions
    System.out.println(ex.getMessage ()
        + ex.getSQLState () + ex.getErrorCode ());
}
```

rs works like a cursor

Examining Database Metadata

DatabaseMetaData object gives information about the database system and the catalog.

```
DatabaseMetaData  md = con.getMetaData();
// print information about the driver:
System.out.println(
      "Name:" + md.getDriverName() +
      "version: " + md.getDriverVersion());
```

Some DatabaseMetaData Methods

134 methods in JDBC 2.0

- getCatalogs(): retrieves catalog names available in this database
- getIndexInfo(): retrieves a description of the indexes and statistics for the given table
- getTables(): retrieves a description of the tables available in the given catalog
- GetColumns(): retrieves a description of table columns available in the specified catalog
- getPrimaryKeys(): retrieves a description of the given table's primary key columns.

Some DatabaseMetaData Methods

- getTables(): retrieves a description of the tables available in the given catalog. The parameters are:
  - catalog name
  - schema name
  - table name
  - a list of table types

Ex: "getTables(null,null,null,null)" gets information for all tables

- getColumns(): retrieves a description of table columns available in the specified catalog. The parameters are:
  - catalog name
  - Schema name
  - Table name
  - Column name

Ex: "getColumns(null,null,tableName,null)" gets all attributes of tableName

Database Metadata (Contd.)

```
DatabaseMetaData md=con.getMetaData();
ResultSet  trs=md.getTables(null,null,null,null);   // get all tables
String  tableName;
While(trs.next()) {           // for each table, do …
   tableName = trs.getString("TABLE_NAME");  // get TABLE_NAME field
   System.out.println("Table: " + tableName);
   ResultSet crs = md.getColumns(null,null,tableName,null);
                                          // get all attributes of tableName
   while (crs.next()) {
      System.out.println(crs.getString("COLUMN_NAME") + ", ");
   }
}
```

trs

TABLE_NAME	…
Table1	…
Table2	…
► Sailors	…
Table3	…
Table4	…
Table5	…

crs

COLUMN_NAME	…
sid	…
► sname	…
rating	…
age	…