Database access and JDBC

- Integrating database access into JSP applications

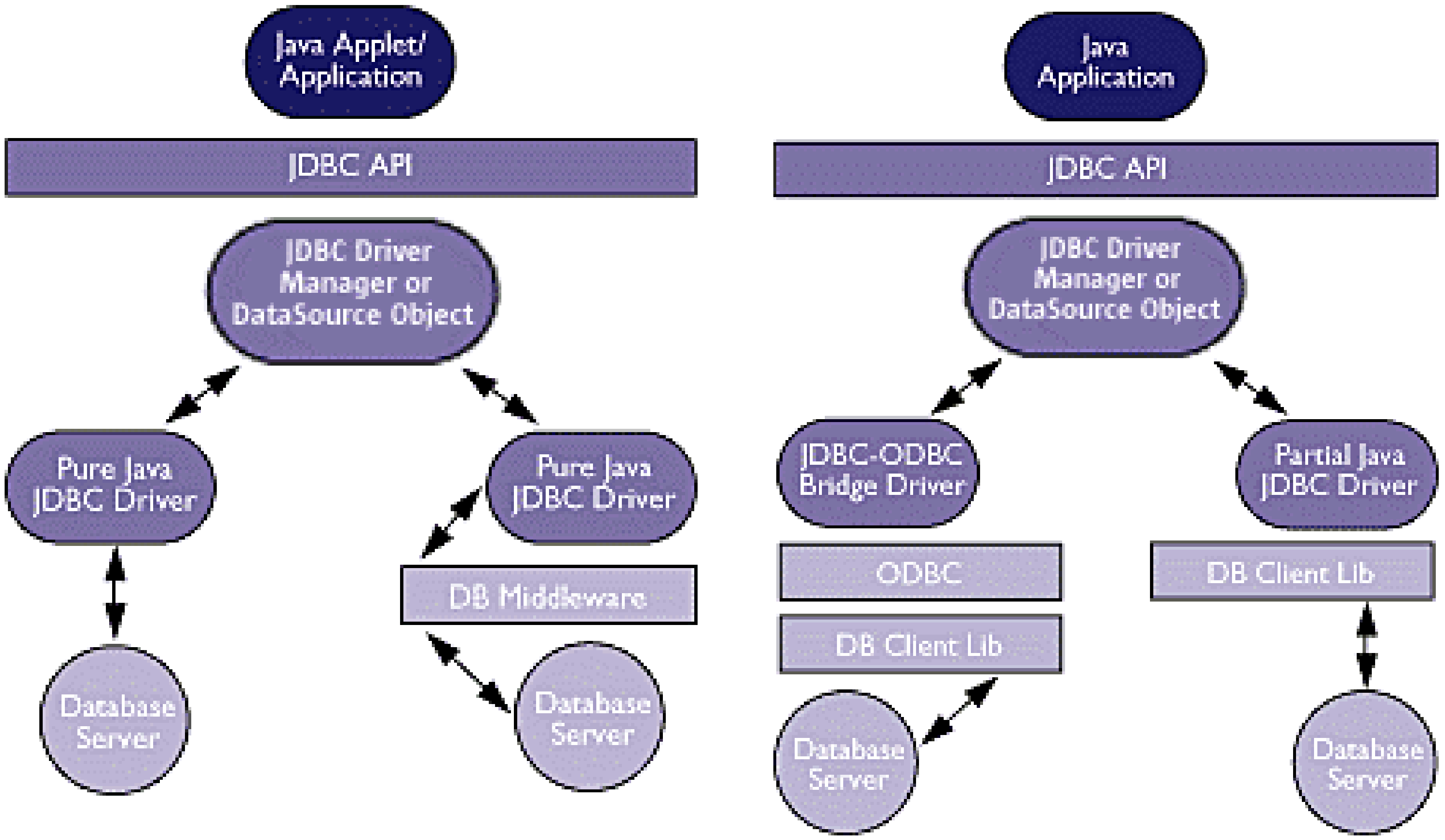
Goals

- Access SQL DBMS’s from JSP pages
  - JDBC technology
- Integrate SQL query results into the resulting HTML content
- Generate SQL queries according to FORM values

JDBC

- Standard library for accessing relational databases
- Compatible with most/all different databases
- JDBC does *not* mean Java Database Connectivity :)
- Defined in package java.sql and javax.sql
- Documentation:
  - http://java.sun.com/javase/technologies/database/index.jsp

Architecture



JDBC scope

- Standardizes
  - Mechanism for connecting to DBMSs
  - Syntax for sending queries
  - Structure representing the results
- Does not standardize
  - SQL syntax: dialects, variants, extensions, ...

Main elements

- Java application (in our case, JSP)
- JDBC Driver Manager
  - For loading the JDBC Driver
- JDBC Driver
  - From DBMS vendor
- DBMS
  - In our case, MySQL

Basic steps

- Load the JDBC driver
- Define the connection URL
- Establish the connection
- Create a statement object
- Execute a query or update
- Process the results
- Close the connection

1. Loading the driver

- A Driver is a DMBS-vendor provided class, that must be available to the Java application
  - Must reside in Tomcat’s CLASSPATH
- The application usually doesn’t know the driver class name until run-time (to ease the migration to other DMBSs)
- Needs to find and load the class at run-time
  - Class.forName method in the Java Class Loader

Types of drivers (1/3)

- A JDBC-ODBC bridge
  - provides JDBC API access via one or more ODBC drivers. ODBC native code must be loaded on each client machine that uses this type of driver.
- A native-API partly Java technology-enabled driver
  - converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Requires that some binary code be loaded on each client machine.

Types of drivers (2/3)

- A net-protocol fully Java technology-enabled driver
  - translates JDBC API calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. Specific protocol depends on the vendor. The most flexible alternative

Types of drivers (3/3)

- A native-protocol fully Java technology-enabled driver
  - converts JDBC technology calls into the network protocol used by DBMSs directly. Direct call from the client machine to the DBMS server. Many of these protocols are proprietary: the database vendors will be the primary source for this style of driver.

MySQL JDBC driver

- MySQL® Connector/J
  - http://www.mysql.com/products/connector/j/
- Provides mysql-connector-java-[version]-bin.jar
  - Copy into CLASSPATH
  - E.g.: c:\Program files\Java\jre1.5.0_09\lib\ext
- The driver is in class
  - com.mysql.jdbc.Driver

Loading the MySQL driver

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

// Notice, do not import com.mysql.jdbc.* or you will have problems!

public class LoadDriver {
    public static void main(String[] args) {
        try {
            // The newInstance() call is a work around for some
            // broken Java implementations

            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (Exception ex) {  // mostly ClassNotFoundException
            // handle the error
        }
}
```

2. Define the connection URL

- The Driver Manager needs some information to connect to the DBMS
  - The database type (to call the proper Driver, that we already loaded in the first step)
  - The server address
  - Authentication information (user/pass)
  - Database / schema to connect to
- All these parameters are encoded into a string
  - The exact format depends on the Driver vendor

MySQL Connection URL format

- jdbc:mysql://[host:port],[host:port].../[database][?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...
  - jdbc:mysql://
  - host:port (localhost)
  - /database
  - ?user=username
  - &password=pppppp

3. Establish the connection

- Use DriverManager.getConnection
  - Uses the appropriate driver according to the connection URL
  - Returns a Connection object
- Connection connection = DriverManager.getConnection(URLString)
- Contacts DBMS, validates user and selects the database
- On the Connection object subsequent commands will execute queries

Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

    try {
        Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost/test?
user=monty&password=greatsqldb");
        // Do something with the Connection
       ....
    } catch (SQLException ex) {
        // handle any errors
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
```

4. Create a Statement object

- Statement statement = connection.createStatement() ;
- Creates a Statement object for sending SQL statements to the database.
- SQL statements without parameters are normally executed using Statement objects.
  - If the same SQL statement is executed many times, it may be more efficient to use a PreparedStatement object.

5. Execute a query

- Use the executeQuery method of the Statement class
  - ResultSet executeQuery(String sql)
  - sql contains a SELECT statement
- Returns a ResultSet object, that will be used to retrieve the query results

Other execute methods

- int executeUpdate(String sql)
  - For INSERT, UPDATE, or DELETE statements
  - For other SQL statements that don’t return a resultset (e.g., CREATE TABLE)
  - returns either the row count for INSERT, UPDATE or DELETE statements, or 0 for SQL statements that return nothing
- boolean execute(String sql)
  - For general SQL statements

Example

```
String query = "SELECT col1, col2, col3
FROM sometable" ;
ResultSet resultSet =
statement.executeQuery(query) ;
```

6. Process the result

- The ResultSet object implements a “cursor” over the query results
  - Data are available a row at a time
    - Method ResultSet.next() goes to the next row
  - The column values (for the selected row) are available trhough getXXX methods
    - getInt, getString, ...
  - Data types are converted from SQL types to Java types

ResultSet.getXXX methods

- XXX is the desired datatype
  - Must be compatible with the column type
  - String is almost always acceptable
- Two versions
  - getXXX(int columnIndex)
    - number of column to retrieve (starting from 1!!!!)
  - getXXX(String columnName)
    - name of column to retrieve

ResultSet navigation methods

- boolean next()
  - Moves the cursor down one row from its current position.
  - NOTE: A ResultSet cursor is initially positioned before the first row; the first call to the method next makes the first row the current row; the second call makes the second row the current row, and so on.

Other navigation methods (1/2)

- Query cursor position
  - boolean isFirst()
  - boolean isLast()
  - boolean isBeforeFirst()
  - boolean isAfterLast()

Other navigation methods (2/2)

- Move cursor
  - void beforeFirst()
  - void afterLast()
  - boolean first()
  - boolean last()
  - boolean absolute(int row)
  - boolean relative(int rows) // positive or negative offset
  - boolean previous()

Example

```
while( resultSet.next() )
{
    out.println( "<p>" +
       resultSet.getString(1) + " - " +
       resultSet.getString(2) + " - " +
       resultSet.getString(3) + "</p>" ) ;
}
```

Datatype conversions (MySQL)

These MySQL Data Types	**Can always be converted to these Java types**
CHAR, VARCHAR, BLOB, TEXT, ENUM, and SET	java.lang.String, java.io.InputStream, java.io.Reader, java.sql.Blob, java.sql.Clob
FLOAT, REAL, DOUBLE PRECISION, NUMERIC, DECIMAL, TINYINT, SMALLINT, MEDIUMINT, INTEGER, BIGINT	java.lang.String, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Double, java.math.BigDecimal
DATE, TIME, DATETIME, TIMESTAMP	java.lang.String, java.sql.Date, java.sql.Timestamp

7. Close the connection

- Additional queries may be done on the same connection.
  - Each returns a different ResultSet object, unless you re-use it
- When no additional queries are needed:
  - connection.close() ;