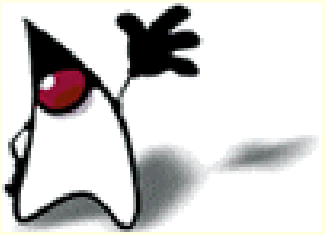
JDBC – Java Database Connectivity



JDBC

- JDBC (Java Database Connectivity) API allows Java programs to connect to databases
- Database access is the same for all database vendors
- The JVM uses a JDBC driver to translate generalized JDBC calls into vendor specific database calls
- There are four general types of JDBC drivers

Introduction to JDBC

- **JDBC** is used for accessing databases from Java applications
- Information is transferred from relations to objects and vice-versa
  - *databases* optimized for *searching/indexing*
  - *objects* optimized for *engineering/flexibility*

JDBC

- JDBC is a Sun trademark
  - It is often taken to stand for Java Database Connectivity
- Java is very standardized, but there are many versions of SQL
- JDBC is a means of accessing SQL databases from Java
  - JDBC is a standardized API for use by Java programs
  - JDBC is also a specification for how third-party vendors should write database drivers to access specific SQL versions

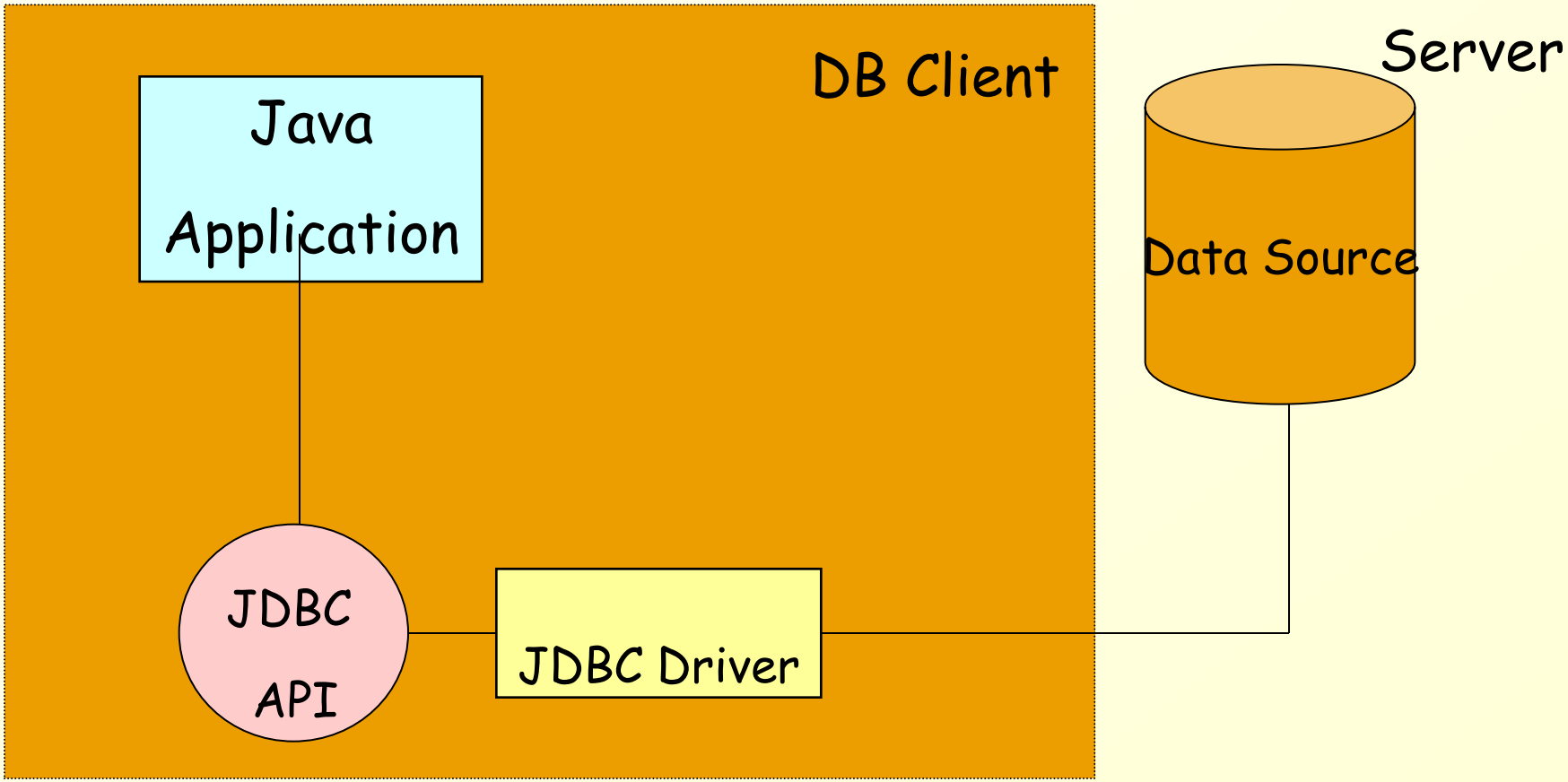
Driver types

- There are four types of drivers:
  - **JDBC Type 1 Driver** -- JDBC/ODBC Bridge drivers
    - ODBC (Open DataBase Connectivity) is a standard software API designed to be independent of specific programming languages
    - Sun provides a JDBC/ODBC implementation
  - **JDBC Type 2 Driver** -- use platform-specific APIs for data access
  - **JDBC Type 3 Driver** -- 100% Java, use a net protocol to access a remote listener and map calls into vendor-specific calls
  - **JDBC Type 4 Driver** -- 100% Java
    - Most efficient of all driver types

Pure Java Driver (Type 4)

- These drivers convert the JDBC API calls to direct network calls using vendor-specific networking protocols by making direct socket connections with the database
- It is the most efficient method to access database, both in performance and development time
- It is the simplest to deploy
- All major database vendors provide pure Java JDBC drivers for their databases and they are also available from third party vendors
- For a list of JDBC drivers, refer to
  - http://industry.java.sun.com/products/jdbc/drivers

Pure Java Driver (2)



Connector/J

- Connector/J is a JDBC Type 4 Driver for connecting Java to MySQL
- Installation is very simple:
  - Download the "Production Release" ZIP file from http://dev.mysql.com/downloads/connector/j/3.1.html
  - Unzip it
  - Put the JAR file where Java can find it
    - Add the JAR file to your CLASSPATH, or
    - In Eclipse: Project --> Properties --> Java Build Path --> Libraries --> Add External Jars...

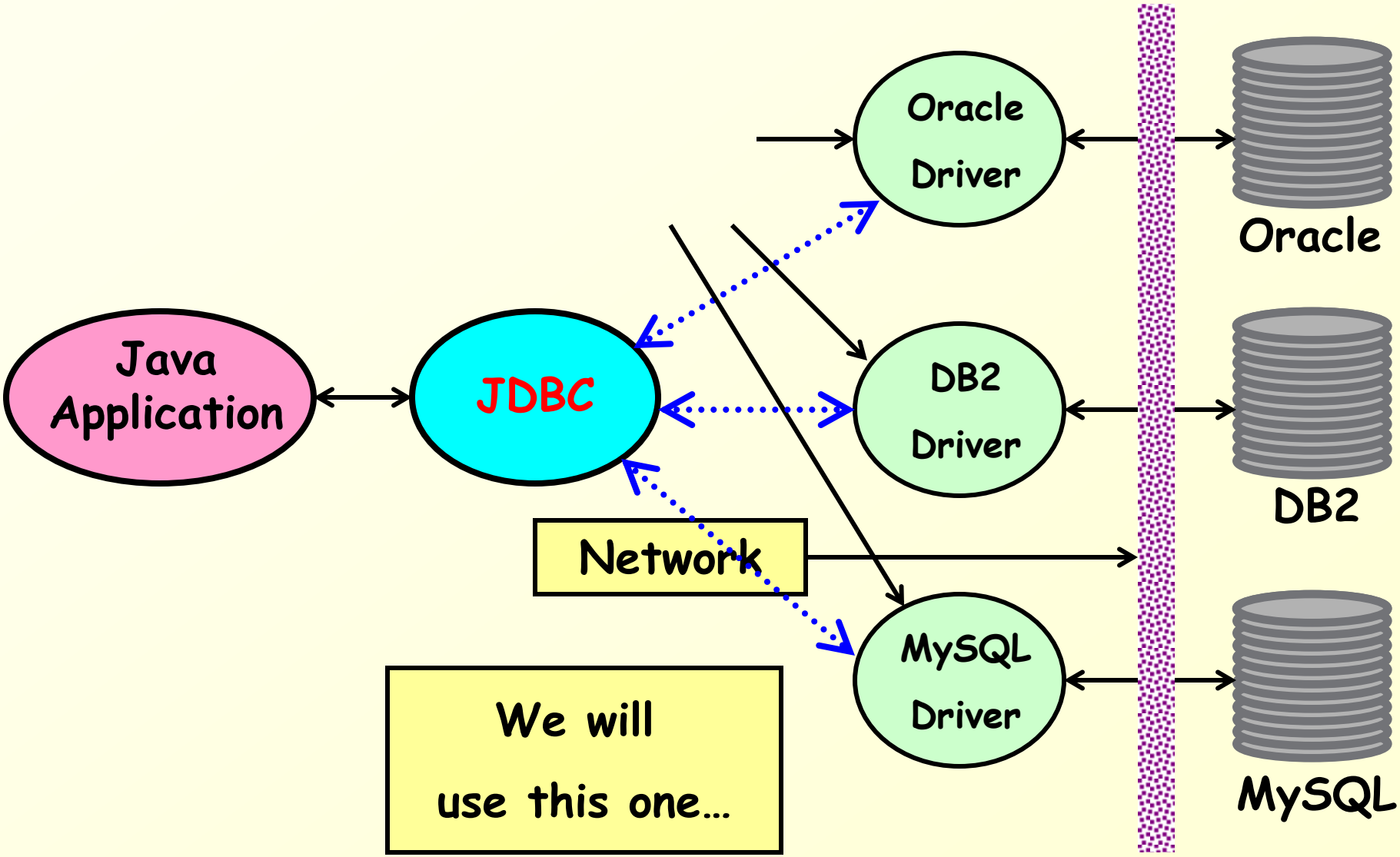
Connecting to the server

- First, make sure the MySQL server is running
- In your program,
  - import java.sql.Connection; // not com.mysql.jdbc.Connection
    import java.sql.DriverManager;
    import java.sql.SQLException;

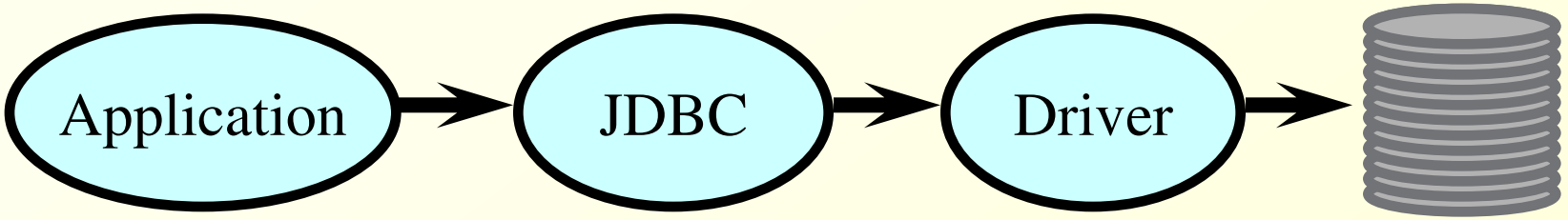
  - Register the JDBC driver,
    Class.forName("com.mysql.jdbc.Driver").newInstance();

  - Invoke the getConnection() method,
    Connection con =
        DriverManager.getConnection("jdbc:mysql:///myDB",
                                    myUserName,
                                    myPassword);
  - or getConnection("jdbc:mysql:///myDB?user=dave&password=xxx")

JDBC Architecture



JDBC Architecture (cont.)



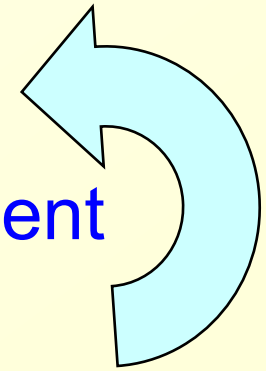
- Java code calls JDBC library
- JDBC loads a *driver*
- Driver talks to a particular database
- An application can work with several databases by using all corresponding drivers
- Ideal: can change database engines *without changing any application code* (not always in practice)

JDBC Driver for MySQL (Connector/J)

- Download Connector/J using binary distribution from : http://dev.mysql.com/downloads/connector/j/5.0.html
- To install simply unzip (or untar) and put mysql-connector-java-*[version]*-bin.jar (I have installed **mysql-connector-java-5.0.4-bin.jar**) in the class path
- For online documentation, see : http://dev.mysql.com/doc/refman/5.0/en/connector-j.html

Seven Steps

1. Load the driver
2. Define the connection URL
3. Establish the connection
4. Create a Statement object
5. Execute a query using the Statement
6. Process the result
7. Close the connection



Loading the Driver

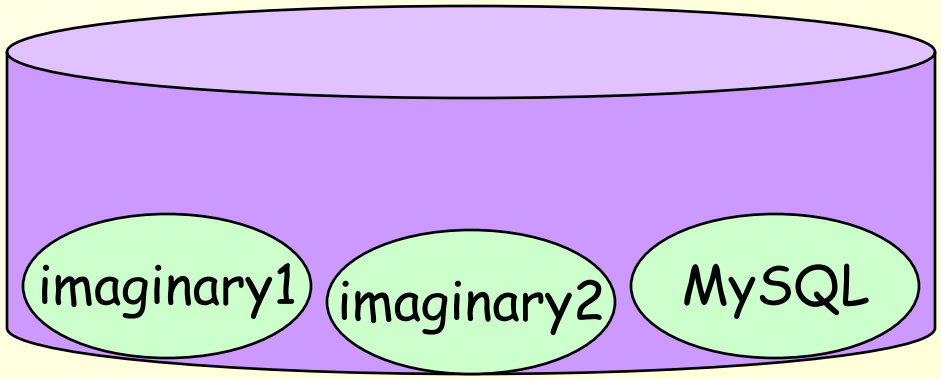
- We can register the driver indirectly using the statement

```
Class.forName("com.mysql.jdbc.Driver");
```

- Class.forName loads the specified class
- When mysqlDriver is loaded, it automatically
  - creates an instance of itself
  - registers this instance with the DriverManager
- Hence, the driver class can be given as an argument of the application

An Example

```
// A driver for imaginary1
Class.forName("ORG.img.imgSQL1.imaginary1Driver");
// A driver for imaginary2
Driver driver = new ORG.img.imgSQL2.imaginary2Driver();
DriverManager.registerDriver(driver);
//A driver for MySQL
Class.forName("com.mysql.jdbc.Driver");
```



Registered Drivers

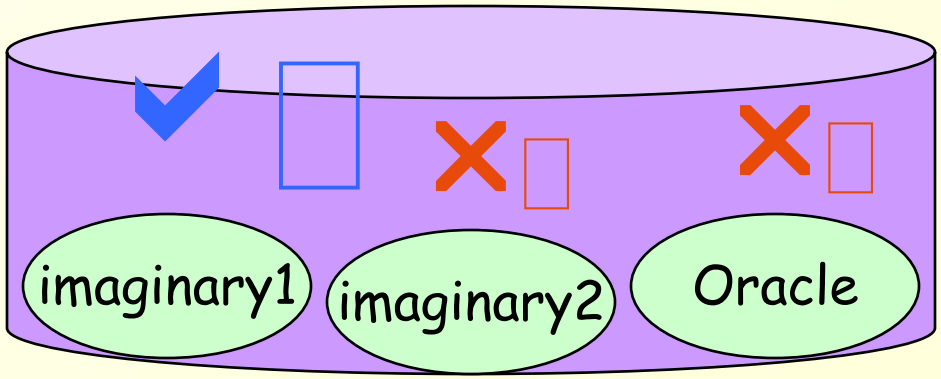
Connecting to the Database

- Every database is identified by a URL
- Given a URL, DriverManager looks for the driver that can talk to the corresponding database
- DriverManager tries all registered drivers, until a suitable one is found

Connecting to the Database

```
Connection con = DriverManager.
    getConnection("jdbc:imaginaryDB1");
```

acceptsURL("jdbc:imaginaryDB1")?



Registered Drivers

We Use:

DriverManager.getConnection(<URL>, <user>, <pwd>);

Where <UR>L is : jdbc:mysql://coe-cognac.engineering.mu.edu:3306/<db_name>

Interaction with the Database

- We use Statement objects in order to
  - Query the database
  - Update the database
- Three different interfaces are used:

  Statement, PreparedStatement, CallableStatement
- All are interfaces, hence cannot be instantiated
- They are created by the Connection

Querying with Statement

```
String queryStr =
        "SELECT * FROM employee " +
        "WHERE lname = 'Wong'";

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(queryStr);
```

- The executeQuery method returns a ResultSet object representing the query result.
  - Will be discussed later…

Changing DB with Statement

```
String deleteStr =
        "DELETE FROM employee " +
        "WHERE lname = 'Wong'";

Statement stmt = con.createStatement();
int delnum = stmt.executeUpdate(deleteStr);
```

- executeUpdate is used for data manipulation: insert, delete, update, create table, etc. (anything other than querying!)
- executeUpdate returns the number of rows modified

About Prepared Statements

- Prepared Statements are used for queries that are executed many times
- They are parsed (compiled) by the DBMS only once
- Column values can be set after compilation
- Instead of values, use '?'
- Hence, Prepared Statements can be though of as statements that contain placeholders to be substituted later with actual values

Querying with PreparedStatement

```
String queryStr =
        "SELECT * FROM employee " +
        "WHERE superssn= ? and salary > ?";

PreparedStatement pstmt =
        con.prepareStatement(queryStr);

pstmt.setString(1, "333445555");
pstmt.setInt(2, 26000);

ResultSet rs = pstmt.executeQuery();
```

Updating with PreparedStatement

```
String deleteStr =
        “DELETE FROM employee " +
        "WHERE superssn = ? and salary > ?";

PreparedStatement pstmt =
        con.prepareStatement(deleteStr);

pstmt.setString(1, "333445555");
pstmt.setDouble(2, 26000);

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

Statements vs. PreparedStatements: Be Careful!

- Are these the same? What do they do?

```
String val = "abc";
PreparedStatement pstmt =
        con.prepareStatement("select * from R where A=?");
pstmt.setString(1, val);
ResultSet rs =  pstmt.executeQuery();
```

```
String val = "abc";
Statement stmt =  con.createStatement( );
ResultSet rs =
        stmt.executeQuery("select * from R where A=" + val);
```

Statements vs. PreparedStatements: Be Careful!

- Will this work?

```
PreparedStatement pstmt =
        con.prepareStatement("select * from ?");


pstmt.setString(1, myFavoriteTableString);
```

- No!!! A '?' can only be used to represent a column value

Timeout

- Use setQueryTimeOut(int seconds) of Statement to set a timeout for the driver to wait for a statement to be completed
- If the operation is not completed in the given time, an SQLException is thrown
- What is it good for?

ResultSet

- ResultSet objects provide access to the tables generated as results of executing a Statement queries
- Only one ResultSet per Statement can be open at the same time!
- The table rows are retrieved in sequence
  - A ResultSet maintains a cursor pointing to its current row
  - The next() method moves the cursor to the next row

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
  - indices start at 1 and not 0!
- *Type* get*Type*(String columnName)
  - same, but uses name of field
  - less efficient
- For example: getString(columnIndex), getInt(columnName), getTime, getBoolean, getType,...
- int findColumn(String columnName)
  - looks up column index given column name

ResultSet Methods

- JDBC 2.0 includes scrollable result sets. Additional methods included are : ‘first’, ‘last’, ‘previous’, and other methods.

ResultSet Example

```
Statement stmt = con.createStatement();

ResultSet rs = stmt.
executeQuery("select lname,salary from Employees");

// Print the result

while(rs.next()) {
 System.out.print(rs.getString(1) + ":");
 System.out.println(rs.getDouble("salary"));
}
```

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

Null Values

- In SQL, NULL means the field is empty
- Not the same as 0 or ""
- In JDBC, you must explicitly ask if the last-read field was null
  - ResultSet.wasNull(column)
- For example, getInt(column) will return 0 if the value is either 0 or NULL!

Null Values

- When inserting null values into placeholders of Prepared Statements:
  - Use the method setNull(*index*, Types.*sqlType*) for primitive types (e.g. INTEGER, REAL);
  - You may also use the set*Type*(*index*, null) for object types (e.g. STRING, DATE).

ResultSet Meta-Data

A ResultSetMetaData is an object that can be used to get information about the properties of the columns in a ResultSet object

An example: write the columns of the result set

```
ResultSetMetaData rsmd = rs.getMetaData();
int numcols = rsmd.getColumnCount();

for (int i = 1 ; i <= numcols; i++) {
        System.out.print(rsmd.getColumnLabel(i)+" ");
}
```

Database Time

- Times in SQL are notoriously non-standard
- Java defines three classes to help
- java.sql.Date
  - year, month, day
- java.sql.Time
  - hours, minutes, seconds
- java.sql.Timestamp
  - year, month, day, hours, minutes, seconds, nanoseconds
  - usually use this one

Cleaning Up After Yourself

- Remember to close the Connections, Statements, Prepared Statements and Result Sets

```
con.close();
stmt.close();
pstmt.close();
rs.close()
```

Dealing With Exceptions

- An SQLException is actually a list of exceptions

```
catch (SQLException e) {
   while (e != null) {
         System.out.println(e.getSQLState());
         System.out.println(e.getMessage());
         System.out.println(e.getErrorCode());
         e = e.getNextException();
   }
}
```

Transactions and JDBC

- Transaction: more than one statement that must all succeed (or all fail) together
  - e.g., updating several tables due to customer purchase
- 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 = cancel all actions

Example

- Suppose we want to transfer money from bank account 13 to account 72:

```
PreparedStatement pstmt =
        con.prepareStatement("update BankAccount
                              set amount = amount + ?
                              where accountId = ?");
pstmt.setInt(1,-100);
pstmt.setInt(2, 13);
pstmt.executeUpdate();
pstmt.setInt(1, 100);
pstmt.setInt(2, 72);
pstmt.executeUpdate();
```

What happens if this update fails?

Transaction Management

- Transactions are not explicitly opened and closed
- The connection has a state called AutoCommit mode
- if AutoCommit is true, then every statement is automatically committed
- if AutoCommit is false, then every statement is added to an ongoing transaction
- Default: true

AutoCommit

```
setAutoCommit(boolean val)
```

- If you set AutoCommit to false, you must explicitly commit or rollback the transaction using Connection.commit() and Connection.rollback()
- Note: DDL statements (e.g., creating/deleting tables) in a transaction may be ignored or may cause a commit to occur
  - The behavior is DBMS dependent

Scrollable ResultSet

- Statement createStatement(int resultSetType, int resultSetConcurrency)
- **resultSetType**:
- ResultSet.TYPE_FORWARD_ONLY
- -default; same as in JDBC 1.0
- -allows only forward movement of the cursor
- -when rset.next() returns false, the data is no longer available and the result set is closed.
- ResultSet.TYPE_SCROLL_INSENSITIVE
- -backwards, forwards, random cursor movement.
- -changes made in the database are not seen in the result set object in Java memory.
- ResultSetTYPE_SCROLL_SENSITIVE
- -backwards, forwards, random cursor movement.
- -changes made in the database are seen in the
- result set object in Java memory.

Scrollable ResultSet (cont'd)

- **resultSetConcurrency:**
- ResultSet.CONCUR_READ_ONLY
- This is the default (and same as in JDBC 1.0) and allows only data to be read from the database.
- ResultSet.CONCUR_UPDATABLE
- This option allows for the Java program to make changes to the database based on new methods and positioning ability of the cursor.
- **Example:**

```
Statement stmt = conn.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSetrset= stmt.executeQuery( "SHOW TABLES");
```

Scrollable ResultSet (cont'd)

```
public boolean absolute(int row) throws SQLException
```

- -If the given row number is positive, this method moves the cursor to the given row number (with the first row numbered 1).
- -If the row number is negative, the cursor moves to a relative position from the last row.
- -If the row number is 0, an SQLException will be raised.

```
public boolean relative(int row) throws SQLException
```

- This method call moves the cursor a relative number of rows, either positive or negative.
- An attempt to move beyond the last row (or before the first row) in the result set positions the cursor after the last row (or before the first row).

```
public boolean first() throws SQLException
public boolean last() throws SQLException
public boolean previous() throws SQLException
public boolean next() throws SQLException
```

Scrollable ResultSet (cont'd)

```
public void beforeFirst() throws SQLException
public void afterLast() throws SQLException
public boolean isFirst() throws SQLException
public boolean isLast() throws SQLException
public boolean isAfterLast() throws
  SQLException
public boolean isBeforeFirst() throws
  SQLException
public int getRow() throws SQLException
```

- getRow() method retrieves the current row number: The first row is number 1, the second number 2, and so on.

JDBC Usage in Industry

- Apace DbUtils (http://jakarta.apache.org/commons/dbutils/)
- ORM (Object Relational Mappers):

- Hibernate (http://www.hibernate.org/),
- JDO (http://java.sun.com/products/jdo/),
- TopLink (http://www.oracle.com/technology/products/ias/toplink/index.html)