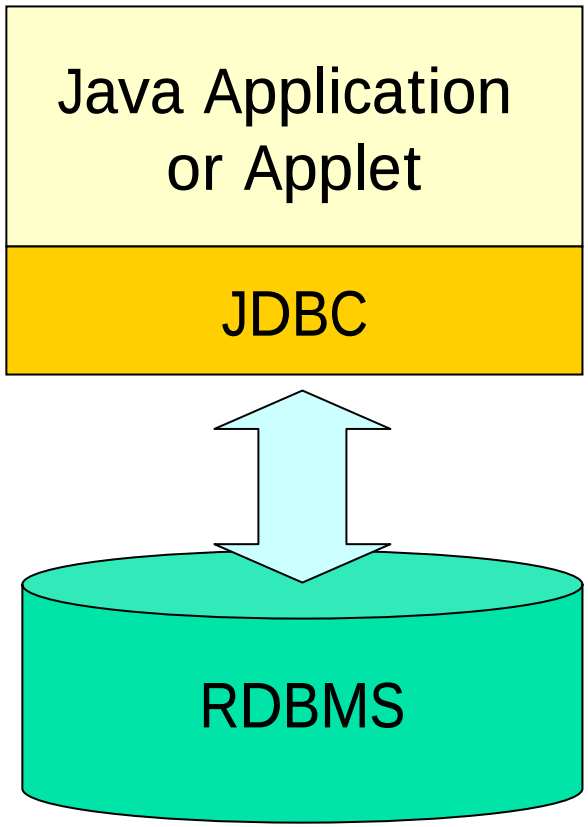
JDBC

What is JDBC



- JDBC, often known as Java Database Connectivity, provides a Java API for updating and querying relational databases using Structured Query Language (SQL)
- JDBC is now at version 2.0, although many databases don't as yet support all of the JDBC 2.0 features!

The 4 step approach to JDBC

- Every JDBC program is made up of the following 4 steps:

Open a connection to the DB

Execute a SQL statement

Process the result

Close the connection to the DB

We'll look at each of the four steps in detail!

Example JDBC program

1. open connection to DB
2. execute SQL statement
3. process result
4. close connection to DB

```
import java.sql.*;
class SelectProducts
{
  public static void main(java.lang.String[ ] args)
  {
    try
      {
          Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
          Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", " db2admin " );
          Statement statement = con.createStatement( );
          ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
          while ( rs.next( ) )
            {
              String name = rs.getString( "NAME" );
              float price = rs.getFloat( "PRICE" );
              System.out.println("Name: "+name+", price: "+price);
            }
          statement.close( );
          con.close( );
      }
    catch( Exception e ) { e.printStackTrace( ); }
  }
}
```

1

2

3

4

Opening a connection to the DB

- There are two parts to this:
  - loading a driver – we need a driver to allow our Java program to talk to the DB
  - opening the connection itself

Loading a driver

- The first step in using JDBC is to load a driver. Here are some examples:

The IBM DB2 driver:

```
Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
```

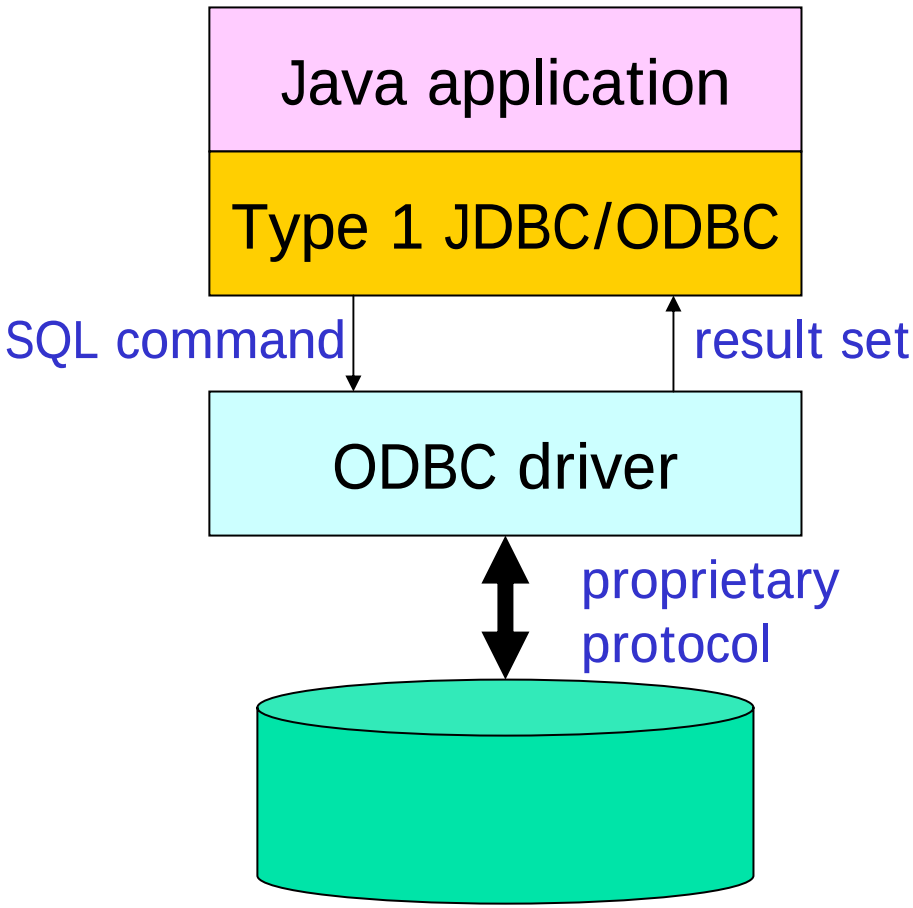
The SUN JDBC/ODBC Bridge driver:

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

There are 4 categories of driver

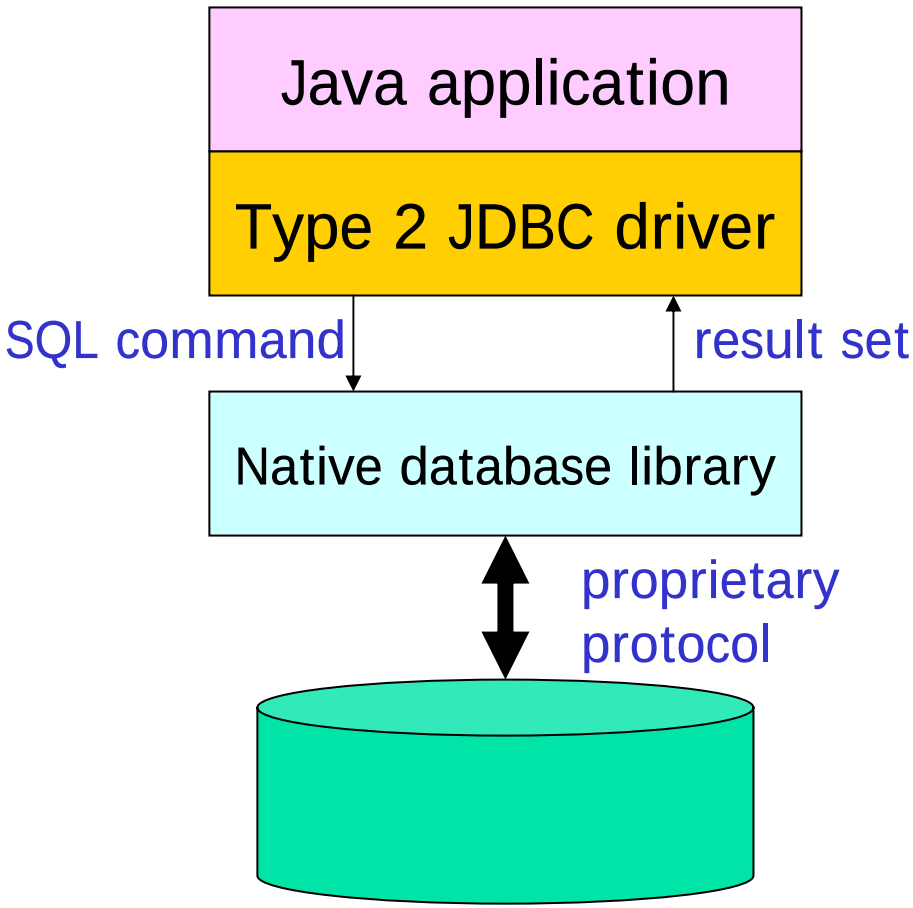
- Type 1 JDBC-ODBC Bridge (Native Code)
  - provides a Java bridge to ODBC
  - implemented in native code and requires some non-Java software on the client
- Type 2 Native-API (Partly Java)
  - uses native code to access the DB with a thin Java wrapper
  - can crash the JVM
- Type 3 Net-protocol (All Java)
  - defines a generic network protocol that interfaces with some middleware that accesses the DB
- Type 4 Native-protocol (All Java)
  - written entirely in Java

Type 1 JDBC-ODBC Bridge



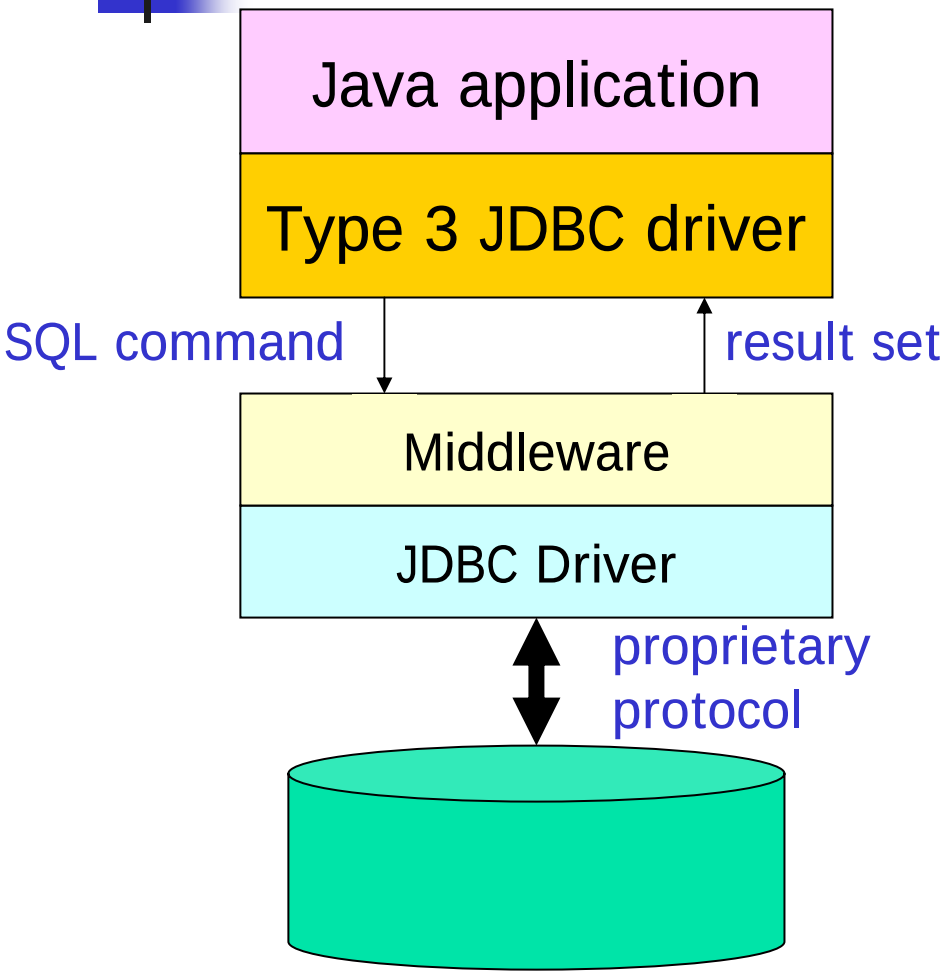
- Provides a Java bridge to ODBC
- Implemented in native code and requires some non-Java software on the client
- Not a mandatory component of the JDK, and is not automatically supported by Java run-time environments
- Only recommended for light use

Type 2 Native API

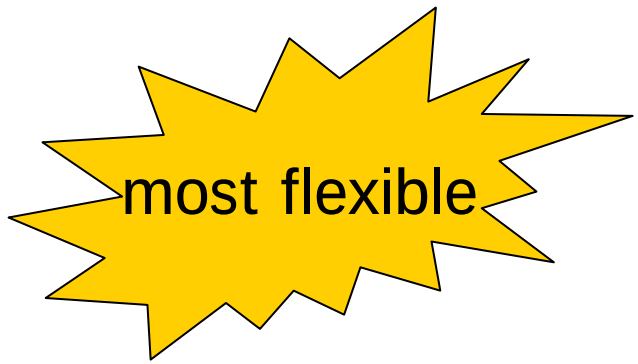


- Converts JDBC commands into DBMS-specific native calls
- Implemented in native code and requires some non-Java software on the client
- Interfaces directly with the DB, so has performance advantages over Type 1

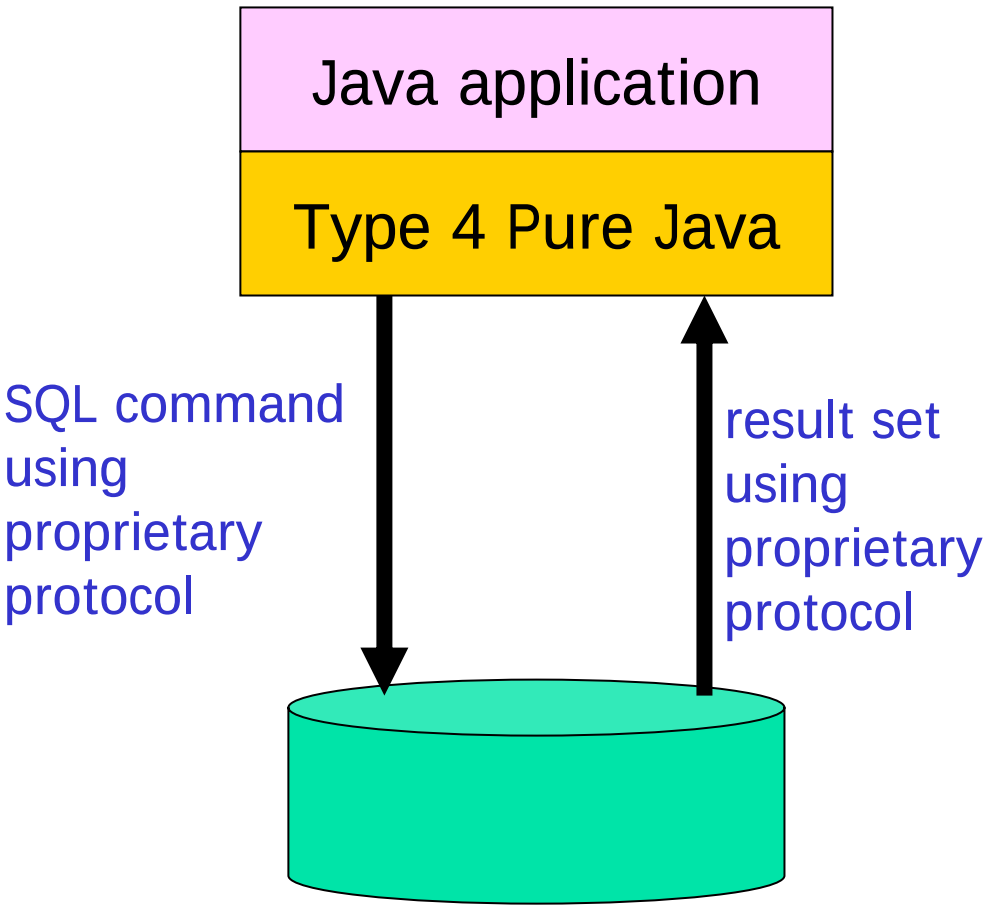
Type 3 JDBC-Net drivers



- A three tier solution
- Allows pure Java clients
- Can change DBMS without affecting the client



Type 4 Native Protocol drivers

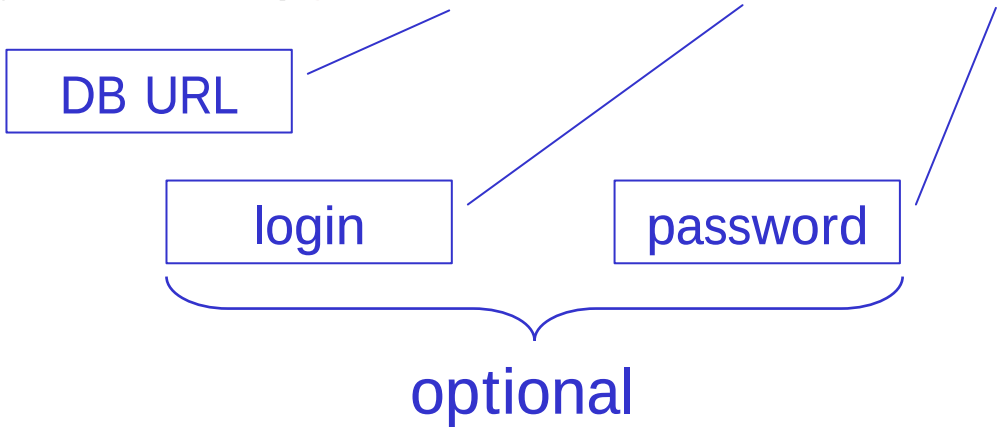


- Native Protocol drivers communicate directly with the DB
- They convert JDBC commands directly into the DB's native protocol
- No additional transformation or middleware layers, therefore has high performance

Making a connection

- Next, the driver must connect to the DBMS:

```
9   Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", " db2admin " );
```



- The object con gives us an open database connection

Creating Statements

- A Statement object is used to send SQL statements to the DB
- First we get a Statement object from our DB connection con

```
10    Statement statement = con.createStatement( );
```

Example – creating a table

```
import java.sql.*;
class CreateProductTable
{
  public static void main(java.lang.String[ ] args)
  {
    try
      {
          Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
          String url = "jdbc:db2:TEST";
          Connection con = DriverManager.getConnection( url, "db2admin", "db2admin" );
          Statement statement = con.createStatement();
          String createProductTable = "CREATE TABLE PRODUCT " +
                                      "(NAME VARCHAR(64), " +
                                      "ID VARCHAR(32) NOT NULL, " +
                                      "PRICE FLOAT, " +
                                      "DESC VARCHAR(256), " +
                                      "PRIMARY KEY(ID))";
          statement.executeUpdate( createProductTable );
      } catch( Exception e ) { e.printStackTrace(); }
  }
}
```

executeUpdate(String sql)

- Use the executeUpdate() method of the Statement object to execute DDL and SQL commands that update a table (INSERT, UPDATE, DELETE):

```
12        String createProductTable = "CREATE TABLE PRODUCT " +
13                                    "(NAME VARCHAR(64), " +
14                                    "ID VARCHAR(32) NOT NULL, " +
15                                    "PRICE FLOAT, " +
16                                    "DESC VARCHAR(256), " +
17                                    "PRIMARY KEY(ID))";
18        statement.executeUpdate( createProductTable );
```

Be careful to always put spaces in the SQL string at the right places!

Example: inserting rows

```
import java.sql.*;
class InsertProducts
{
  public static void main(java.lang.String[ ] args)
  {
    try
      {
          Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
          String url = "jdbc:db2:TEST";
          Connection con = DriverManager.getConnection( url, "db2admin", " db2admin " );
          Statement statement = con.createStatement();
          statement.executeUpdate("INSERT INTO PRODUCT " +
                                  "VALUES ( 'UML User Guide', " +
                                  "'0-201-57168-4', 47.99, 'The UML user guide')" );
          statement.executeUpdate("INSERT INTO PRODUCT " +
                                  "VALUES ( 'Java Enterprise in a Nutshell', " +
                                  "'1-56592-483-5', 29.95, 'A good introduction to J2EE')" );
          con.close();
          statement.close();
      }catch( Exception e ) { e.printStackTrace(); }
  }
}
```

executeQuery(String sql)

- We use the executeQuery(…) method of the Statement object to execute a SQL statement that returns a single ResultSet:

```
11  ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
```

- Typically, the SQL statement is a SQL SELECT
- executeQuery(…) *always* returns a ResultSet, never null. However, the ResultSet may be empty

Example: selecting rows

```
1   import java.sql.*;
2   class SelectProducts
3   {
4     public static void main(java.lang.String[ ] args)
5     {
6       try
7         {
8             Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
9             Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", " db2admin " );
10            Statement statement = con.createStatement();
11            ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
12            while ( rs.next( ) )
13              {
14                String name = rs.getString( "NAME" );
15                float price = rs.getFloat( "PRICE" );
16                System.out.println("Name: "+name+", price: "+price);
17              }
18            statement.close();
19            con.close();
20        } catch( Exception e ) { e.printStackTrace(); }
21    }
22  }
```

ResultSet

- ResultSet objects provide access to a table
  - usually they provide access to the pseudo table that is the result of a SQL query
- ResultSet objects maintain a cursor pointing to the current row of data
  - this cursor initially points *before* the first row and is moved to the first row by the next() method

```
ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
while ( rs.next( ) )
 {
  String name = rs.getString( "NAME" );
  float price = rs.getFloat( "PRICE" );
  System.out.println("Name: "+name+", price: "+price);
 }
```

JDBC 2.0

Types of ResultSet

Statement statement = con.createStatement(*type*, *concurrency*);

- Depending on the parameters passed into the Connection.createStatement(…) method, we can get a total of 6 different types of ResultSet returned!
- Passing no arguments to createStatement() gives a default forward-only read-only ResultSet
- We'll look at the possible values for *type* and *concurrency* next...

JDBC 2.0

type

type =	**semantics**
ResultSet.TYPE_SCROLL_SENSITIVE	Scrollable. Reflects changes made to the underlying data
ResultSet.TYPE_SCROLL_INSENSITIVE	Scrollable. Does *not* reflect changes made to the underlying data
ResultSet.TYPE_FORWARD_ONLY	Not scrollable. Does *not* reflect changes made to the underlying data

N.B. Scrollable means that we can navigate forwards *and* backwards through the ResultSet

JDBC 2.0

concurrency

concurrency =	**semantics**
ResultSet.CONCUR_READ_ONLY	the **ResultSet** *may not* be updated
ResultSet.CONCUR_UPDATABLE	the **ResultSet** *may* be updated

getXXX(...) methods

see notes!

- The ResultSet has a wide range of methods to return SQL types such as VARCHAR as equivalent Java types
- For example rs.getString("NAME") returns the product name as a String
  - in fact, we can get any of the SQL types with getString(…) and it will automatically be converted to a String
- The getXXX(…) methods can take a column name or the number of the column
  - column numbers start at 1 and go from left to right

JDBC 2.0

ResultSet navigation methods

Method	Semantics
first()	Moves cursor to first row
last()	Moves cursor to last row
next()	Moves cursor to next row
previous()	Moves cursor to previous row
beforeFirst()	Moves cursor to just before the first row
afterLast()	Moves cursor to just after the last row
absolute(int)	Moves cursor to a row index. If positive – counting from the front, if negative – from the back
relative(int)	Moves cursor a relative number of rows, positive or negative from the current position

Working with ResultSets

- We can limit the number of rows that a ResultSet can contain by using:

```
Statement statement = con.createStatement();
statement.setMaxRows(100);
```

- If a Statement returns multiple ResultSets, then we can move to the next ResultSet as follows:

```
while ( statement.getMoreResults() )
        {
        rs = statement. getResultSet();
        ...
```

JDBC 2.0

Updateable ResultSet

- If the statement is created to be of type ResultSet.CONCUR_UPDATABLE, then we may be able to update the database by modifying the ResultSet itself
  - this may not be supported by all DBMSs as it is not a mandatory requirement for JDBC 2.0 compatibility

JDBC 2.0

updateXXX(...) methods

- Like the getXXX(…) methods, the ResultSet has a wide range of updateXXX(…) methods to change the value of SQL types in the ResultSet
- For example rs.updateString(“PRICE”, 40.0F) changes the price of a product
  - we have to be very careful that that all the types in an update expression match
- The updateXXX(…) methods can take a column name or the number of the column
  - column numbers start at 1 and go from left to right

JDBC 2.0

Updating a row

- This is a three step procedure:
  - navigate to the appropriate row using a SELECT and ResultSet navigation methods
  - update the field values in the ResultSet
  - write the change back to the DB

```
rs.first();
rs.updateFloat("PRICE", 40.0F);
rs.updateRow();
```

JDBC 2.0

Inserting a row

- This is a three step procedure:
  - navigate to insert row
  - update the field values in the ResultSet
  - write the row to the DB

```
rs.moveToInsertRow();
rs.updateString("NAME", "UML Distilled");
rs.updateString("ID", "0-201-32563-2");
rs.updateFloat("PRICE", 40.0F);
rs.insertRow();
```

JDBC 2.0

Deleting a row

- This is a simple two step procedure:
  - navigate to row to be deleted
  - delete the row

```
rs.last();
rs.deleteRow();
```

Prepared statements

- If we want to execute the same SQL statement several times, we can create a PreparedStatement object:
  - at the point of creation of a PreparedStatement object the SQL code is sent to the DB and compiled. Subsequent executions may therefore be more efficient than normal statements
  - PreparedStatements can take parameters

Prepared statement example

```
1   import java.sql.*;
2   class PreparedStatementTest
3   {
4     public static void main(java.lang.String[ ] args)
5     {
6       try
7         {
8               Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
9               Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", " db2admin " );
10              PreparedStatement findBooks = con.prepareStatement(
11                                                  "SELECT NAME FROM PRODUCT WHERE NAME LIKE ? " );
12              findBooks.setString( 1, "%Java%" );
13              ResultSet rs = findBooks.executeQuery();
14              while ( rs.next() )
15                { System.out.println("Name: "+ rs.getString( "NAME" ) ); }
16              findBooks.setString( 1, "%UML%" );
17              rs = findBooks.executeQuery();
19              while ( rs.next() )
20                { System.out.println("Name: "+ rs.getString( "NAME" ) ); }
21              findBooks.close();
22              con.close();
23        } catch( Exception e ) { e.printStackTrace(); }
24    }
25  }
```

Transactions

- Normally each SQL statement will be committed automatically when it has completed executing (auto commit is *on*)
- A group of statements can be committed together by turning auto commit *off*, and explicitly committing the statements ourselves
- This ensures that if *any* of the statements fail, they *all* fail. We can then *roll back* the transaction

JDBC transaction modes

con.setTransactionIsolation(*mode*)

- TRANSACTION_NONE
  - transactions are disabled or not supported
- TRANSACTION_READ_UNCOMITTED
  - other transactions may see the results *before* the transaction is committed
  - “dirty read” - uncommitted rows might be rolled back if the transaction fails.
- TRANSACTION_READ_COMMITTED
  - dirty reads are not allowed.
- TRANSACTION_REPEATABLE_READ
  - if a transaction performs multiple reads on a row that is being changed by another transaction, then it does not see the changes
- TRANSACTION_SERIALIZABLE
  - same as TRANSACTION_REPEATABLE_READ but also protects against row insertions
  - if a transaction does a read, another transaction inserts a row, and the first transaction does another read, the first transaction does *not* see the new row.

Transaction example

```
import java.sql.*;
class TransactionTest
{
  public static void main(java.lang.String[ ] args)
  {
    try
      {
            Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
            String url = "jdbc:db2:TEST";
            Connection con = DriverManager.getConnection( url, "db2admin", "db2admin" );
            Statement s = con.createStatement();
            try
              {
                con.setAutoCommit( false );
                s.executeUpdate("UPDATE PRODUCT SET PRICE = 40.00 WHERE ID = '0-201-57168-4' " );
                s.executeUpdate(
                  "UPDATE REVIEW SET COMMENT = 'Now on sale!' WHERE BOOKID = '0-201-57168-4' " );
                con.commit();
              }catch( SQLException e ) { con.rollback(); }
              finally{ con.close(); s.close(); }
      }catch( Exception e ){ e.printStackTrace(); }
  }
}
```

JDBC 2.0

Batch updates

- JDBC 1.0 was very inefficient for loading a lot of data into a DB - a separate SQL command had to be executed for each record changed
- JDBC 2.0 allows batch updates
  - multiple statements can be executed as a single batch
  - we can roll back the whole batch if a single statement fails
- We simply add statements to be batched to a Statement or PreparedStatement object using addBatch()!
- We can remove the statements using clearBatch()

JDBC 2.0

Batch update example

```
import java.sql.*;
class BatchInsertProducts
{
  public static void main(java.lang.String[ ] args) throws SQLException, ClassNotFoundException
  {
        Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
        String url = "jdbc:db2:TEST";
        Connection con = DriverManager.getConnection( url, "db2admin", "db2admin" );
        Statement s = con.createStatement();
        try
         {
          con.setAutoCommit( false );
          s.addBatch("INSERT INTO PRODUCT " + "VALUES ( 'The Object Constraint Language', " +
                "'0-201-37940-4', 29.95, 'All about constraints')" );
          s.addBatch("INSERT INTO PRODUCT " + "VALUES ( 'The Rational Unified Process', " +
                "'0-201-60459-0',29.95, 'A good introduction to RUP')" );
          int[ ] count = s.executeBatch( );
          con.commit( );

         }catch( SQLException e ) { con.rollback( ); }
         finally{ con.close( ); s.close( ); }
   }
}
```

Stored procedures

- The syntax for *defining* a stored procedure is different for each DBMS
  - use the stored procedure tools that come with the RDBMS
- The syntax for *calling* a stored procedure is different for each DBMS
  - JDBC defines a special *escape sequence syntax* that allows stored procedures to be called in the same way on any RDBMS

Escape sequences

{?= call <procedure-name>(<arg1>,<arg2>, …)}
{call <procedure-name>(<arg1>,<arg2>, …)}

- The ? represents a return value
- <procedure-name> is the name of the stored procedure
- <arg1> etc. are the arguments passed into and out of the stored procedure

Stored procedure example

```
1   import java.sql.*;
2   class StoredProcedureExample
3   {
4     public static void main(java.lang.String[ ] args)
5     {
6       try
7         {
8             Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
9             Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", " db2admin " );
10            CallableStatement cs = con.prepareCall("{call DB2ADMIN.ALLPRODUCTS}");
11            cs.execute();
12            ResultSet rs = cs.getResultSet();
13            while ( rs.next() )
14              {
15                String name = rs.getString( "NAME" );
16                float price = rs.getFloat( "PRICE" );
17                System.out.println("Name: "+name+", price: "+price);
18              }
19            con.close();
20            cs.close();
21        }catch( Exception e ){ e.printStackTrace(); }
22    }
23  }
```

create a callable statement

Using input parameters

```
1   import java.sql.*;
2   class StoredProcedureParameterExample
3   {
4     public static void main(java.lang.String[ ] args)
5     {
6       try
7         {
8             Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
9             Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", "db2admin" );
10            CallableStatement cs = con.prepareCall("{call DB2ADMIN.FINDPROD2(?)}");
11            cs.setString( 1, "%UML%" );
12            cs.execute();
13            ResultSet rs = cs.getResultSet();
14            while ( rs.next() )
15              {
16                String name = rs.getString( "NAME" );
17                float price = rs.getFloat( "PRICE" );
18                System.out.println("Name: "+name+", price: "+price);
19              }
20            con.close();
21            cs.close();
22        }catch( Exception e ){ e.printStackTrace(); }
23    }
24  }
```

set the parameter value

we specify a single parameter

Metadata

- JDBC has facilities to get information about a ResultSet or DB
  - for a ResultSet, this information may include the number and names of the columns, the types of the columns etc.
  - for a DB this information may include the name of the driver, the DB URL etc.
- This information about a ResultSet or DB is known as *metadata*
- See the following classes for details:
  - ResultSet – see ResultSetMetadata
  - Database – see DatabaseMetadata

Getting metadata

- Getting database metadata:

```
Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", "db2admin" );
DatabaseMetaData dmd = con.getMetaData( );
```

- Getting ResultSet metadata:

```
Statement statement = con.createStatement();
ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
ResultSetMetaData rsmd = rs.getMetaData( );
```

Summary

- We have looked at:
  - 4 step approach to JDBC
    - Connection
    - drivers
  - Statement
    - PreparedStatement
    - batch update
    - transactions
    - CallableStatement (stored procedures)
  - ResultSet handling
  - Metadata

Appendix: IBM DB2 (v6)

Installing the JDBC 2.0 driver

- To install JDBC 2.0:
  - go to the directory sqllib\java12
  - run the command usejdbc2
- To switch back to 1.2.2:
  - go to the directory sqllib\java12
  - run the command usejdbc1