The Java Series

Introduction to JDBC

What is JDBC

Stands for Java Database Connectivity

It s an API for programmers to access databases in an homogeneous way.

With JDBC all databases are accessed in the same way.

This isolates programs from details of databases.

To use JDBC with any database you have to know:

- Standard SQL
- What is your database name, login, pwd, etc.

But you don t have to know:

- The specifics of how to communicate with your database

Databases

A database is a set of tables.

Each table contains a set of rows according to a certain structure.

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Each column contains typed information (string, float, integer, etc.)

Typically databases hold lots of data.

SQL

Is a standard language to manipulate databases.

To create tables

```
CREATE TABLE COFFEES (COF_NAME varchar(32),
SUP_ID int, PRICE float, SALES int, TOTAL int)
```

To retrieve information

```
SELECT * FROM COFFEES WHERE PRICE>8
```

To insert information

```
INSERT INTO COFFEES VALUES (‘Italian’, 101, 5.66, 0, 0)
```

To update information

```
UPDATE COFFEES SET PRICE=10.99 WHERE COF_NAME='Espresso'
```

JDBC & SQL

Although SQL is quite standard, each database has its own protocols, connection procedures, etc. before starting to use SQL.

Programs wanting to access different databases must know SQL AND how to access EACH database.

JDBC aims at hiding vendor specific logistics.

For this, JDBC differentiates:

- Access, protocol, connection to the db: provided by the vendor following JDBC standards.
- Manipulation of the db: done by the programmer using SQL.

How does JDBC work?

Each database vendor provides a **JDBC Driver** which encapsulates all the specifics of their database.

A JDBC Driver is a set of Java classes which:

- Follow the JDBC standards (Java interfaces).
- Implement connections, access, etc. to the database.

To access a database the programmer must:

- Load the correspondent JDBC Driver.
- Use the JDBC API to execute SQL statements and standard database operations.

To access a different database:

- Load a different JDBC Driver.
- **No need to modify database manipulation code.**

Setting things up

To access a db through JDBC, you have to:

- Obtain the JDBC classes. This are part of the java.sql package and are distributed as part of jdk from version 1.1
- Obtain the JDBC driver for the database you want to access. Make it accessible to your java tools (classpath).
- Obtain the login data to your db (login name, password, etc.)

We are going to see examples accessing a sample database stored in the CERN central Oracle service.

- Oracle JDBC Drivers are available at **`www.oracle.com`** or at **`/afs/cern.ch/project/oracle/@sys/prod/jdbc/lib/classes111.zip`**
- We are going to connect the **devdb** test database. The Oracle service provides login info. Machine: **`oradev`**, Port: **`10521`**, SID: **`D`**, login name: **`rramos`**, password: **`raul`**.

Summarizing, at CERN just type:

```
[]> setenv CLASSPATH .:$ORACLE_HOME/jdbc/lib/classes111.zip
```

In your program

Load the JDBC Driver.

Use the **`Connection`** and **`DriverManager`** classes to establish a connection to the DB through your driver.

Use the **`Statement`** class to send SQL statements to the database.

Catch the **`SQLException`** exception whenever appropriate (the compiler will tell you if you forget)

Use the **`ResultSet`** class to access data returned by the database.

Scenario 1: Accessing a table

```
import java.sql.*; . . .
public static void main (String args [])
 { try {
    // Load the Oracle JDBC driver
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
      System.out.println ("Oracle device driver does not exist"); System.exit(1);
    }

    Connection conn =
    DriverManager.getConnection ("jdbc:oracle:thin:@oradev:10521:D", "rramos", "raul");
    // Create a Statement
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("select ENAME, JOB, SAL, HIREDATE from EMP");

    // Iterate through the result and print data
    while (rset.next ())  {
      System.out.print   ("Name "+rset.getString (1));
      System.out.print   (" Job "+rset.getString(2));
      System.out.print   (" Salary "+ rset.getString(3));
      System.out.println (" HDate "+ rset.getString(4));
    }
    // Close Everything
    rset.close();    stmt.close();    conn.close();
   } catch (SQLException e) {
      System.out.println("Error accessing DB "+ e.getErrorCode()+e.getMessage());
  }
}
```

Load driver

Connect to database

Create and execute statement
Using **executeQuery** method

Use ResultSet to iterate
and access the data.

Scenario 1

- Just three objects to perform the query
  - A Connection object to establish the connection.
  - An Statement object to send an statement.
  - A ResultSet object returned by executing an statement.
- Note that:
  - The SQL query is given as a String. SQL syntax errors will trigger an SQLException at run time.
  - The ResultSet is an iterator on the rows returned.
    - next() method to go to next row.
    - getString(int) method to retrieve data on a certain column on the current row.
    - all data is retrieved as String.
    - we can also retrieve data as native types.

Scenario 2. Casting data.

```
import java.sql.*; . . .
public static void main (String args [])
 { try {
    // Load the Oracle JDBC driver
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
      System.out.println ("Oracle device driver does not exist"); System.exit(1);
    }
    Connection conn =
    DriverManager.getConnection ("jdbc:oracle:thin:@oradev:10521:D", "rramos", "raul");
    // Create a Statement
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("select ENAME, JOB, SAL, HIREDATE from EMP");

   // Iterate through the result and print the employee names
    while (rset.next ())  {
      System.out.print   ("Name "+rset.getString (1));
      System.out.print   (" Job "+rset.getString(2));
      System.out.print   (" Salary "+ rset.getBigDecimal(3,2));
      Date d = rset.getDate(4);
      String formattedDate = DateFormat.getDateInstance().format(d);
      System.out.println (" HDate "+ formattedDate);
    }

    // Close Everything
    rset.close();    stmt.close();    conn.close();
   } catch (SQLException e) {
      System.out.println("Error accessing DB "+ e.getErrorCode()+e.getMessage());
  }
}
```

Use different methods with different kinds of SQL data

And now we can manipulate data in their genuine types

Mapping Data Types

For every SQL Data Type there is a correspondent ResultSet.get method. (See Table).

Table 23 Use of ResultSet.getXXX Methods to Retrieve JDBC Types

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x	x	X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

LONGxxxx SQL types
are converted into streams.

Other database operations

We use the Statement.executeQuery method for retrieving information from the database.

- SELECT

We use the Statement.executeUpdate method for manipulating information in the database.

- CREATE TABLE
- INSERT
- UPDATE
- DELETE

Scenario 3. Create and Insert

```
public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
      System.out.println ("Oracle device driver does not exist");
      System.exit(0);
    }

    // Connect to the database
    Connection conn =
    DriverManager.getConnection ("jdbc:oracle:thin:@oradev:10521:D", "rramos", "raul");

    // Create a Statement
    Statement stmt = conn.createStatement ();
    stmt.executeUpdate ("create table COFFEES " +
                  "(COF_NAME varchar(32), SUP_ID int, PRICE float, SALES int, TOTAL int)");

    int r=0;
    r+=stmt.executeUpdate ("insert into COFFEES values ('Colombian', 101, 7.99, 0, 0)");
    r+=stmt.executeUpdate ("insert into COFFEES values ('Espresso',  150, 9.99, 0, 0)");
    System.out.println("A total of "+r+" were rows inserted");

    // Close the Statement
    stmt.close(); conn.close();
  }
}
```

We use **executeUpdate** to create a table

We use **executeUpdate** to insert rows in the table. It returns how many rows were inserted

Scenario 3. Update & Delete

```
public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
      System.out.println ("Oracle device driver does not exist");
      System.exit(0);
    }

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
    DriverManager.getConnection ("jdbc:oracle:thin:@oradev:10521:D", "rramos", "raul");

    // Create a Statement
    Statement stmt = conn.createStatement ();

    int r = stmt.executeUpdate ("update COFFEES set PRICE=15.99 where COF_NAME='Espresso'");
    r+=stmt.executeUpdate ("delete COFFEES where price=7.99");

    System.out.println(r+" rows where updated/modified");

    // Close everything
    stmt.close();  conn.close();
  }
}
```

Again, we use **executeUpdate**, now to update and delete rows.

Prepared statements

In the previous example:

```
r+=stmt.executeUpdate ("insert into COFFEES values ('Colombian', 101, 7.99, 0, 0)");
r+=stmt.executeUpdate ("insert into COFFEES values ('Espresso',  150, 9.99, 0, 0)");
```

we are issuing the same statement but with different data.

At run time, each time, JDBC must parse and compile the SQL statement before sending it to the database. This is not desirable when having large amounts of data to process.

Prepared Statements are like templates of SQL statements which are parsed/compiled only once and then the data is provided as many times as required.

Scenario 4. Prepared Statements

```
public static void main (String args [])
     throws SQLException
{
  // Load the Oracle JDBC driver
  . . . . .
  // Connect to the database
  Connection conn =
  DriverManager.getConnection ("jdbc:oracle:thin:@oradev:10521:D", "rramos", "raul");

  // Create a Statement
  PreparedStatement stmt =
         conn.prepareStatement ("insert into COFFEES values (?, ?, ?, 0, 0)" );

  stmt.setString(1, "French");
  stmt.setInt(2, 300);
  stmt.setDouble(3, 6.87);
  int r=stmt.executeUpdate ();

  stmt.setString(1, "Decaf");
  stmt.setInt(2, 250);  // Same price as before
  r+=stmt.executeUpdate ();

  System.out.println("A total of "+r+" were rows inserted");

  // Close the Statement
  stmt.close(); conn.close();
}
}
```

Create prepared statement.
? indicates places where data will be inserted later on.

Insert data into the statement and execute it

Insert and reuse data and execute it again

Transactions

Transactions group multiple database operations to allow to implement data consistency.

Operations in a Transaction are ALL executed or NONE is executed.

Typical example:

- in an account transfer we have to perform two operations in the database: one debit and one credit.
- either BOTH operations are performed or NONE is.
- we define them as part of the same transaction.
- if there is an error after debiting but before crediting we can tell the db to restore the information that existed BEFORE the transaction begun. --> ROLLBACK.
- whenever the transaction is finished with no errors we COMMIT the changes.
- after committing the changes we cannot rollback anymore.

Transactions in JDBC

By default, every database operation is committed automatically. This is autocommit mode.

To start a transaction:

- Disable autocommit mode on Connection class.

Then perform database operations.

To finish a transaction:

- Invoke Connection.commit() method.

To rollback in case of errors (exception):

- Invoke Connection.rollback() method in exception handler.

Scenario 5. Transactions

```
. . . load driver . . .
Connection conn=null;
try {
   conn =
     DriverManager.getConnection ("jdbc:oracle:thin:@oradev:10521:D", "rramos", "raul");
    // Create a Statement
   Statement stmt = conn.createStatement ();

   conn.setAutoCommit(false);
   String s = "update COFFEES set PRICE=10.99 where COF_NAME='Espresso'";
   int r = stmt.executeUpdate (s);

   r+=stmt.executeUpdate ("update COFFEES set PRICE=9.99 where COF_NAME='French'");
   conn.commit();
   System.out.println(r+" rows where updated/modified");
   stmt.close(); conn.close();

} catch (SQLException e) {
   if (conn!=null) {
      try {
        conn.rollback();
        System.out.println("Error ocurred. Transaction aborted");
      } catch (SQLException rb) {
        System.out.println("Could roll back "+rb.getMessage());
      }
   }
   e.printStackTrace();
}
}
```

Disable autocommit -> Transaction starts

Execute database operations

Finish transaction

If something happens during transaction we rollback

If something happens during rollback print error

ODBC

Open Database Connectivity: to isolate Windows applications from database specifics.

Database vendors provide ODBC Drivers so that Windows applications access homogeneously to dbs.

When using Java under Windows we can also access databases using the existing ODBC Drivers.

We just load the JDBC ODBC Driver which acts as a bridge between them.

Scenario 6. Accessing Access97

Create an Access97 database and store it in a file.

See in Windows: ControlPanel->ODBC->ODBC Drivers the list of ODBC drivers available in your system.

In ODBC programmers are also isolated from login specifics by Data Source Names (DSN).

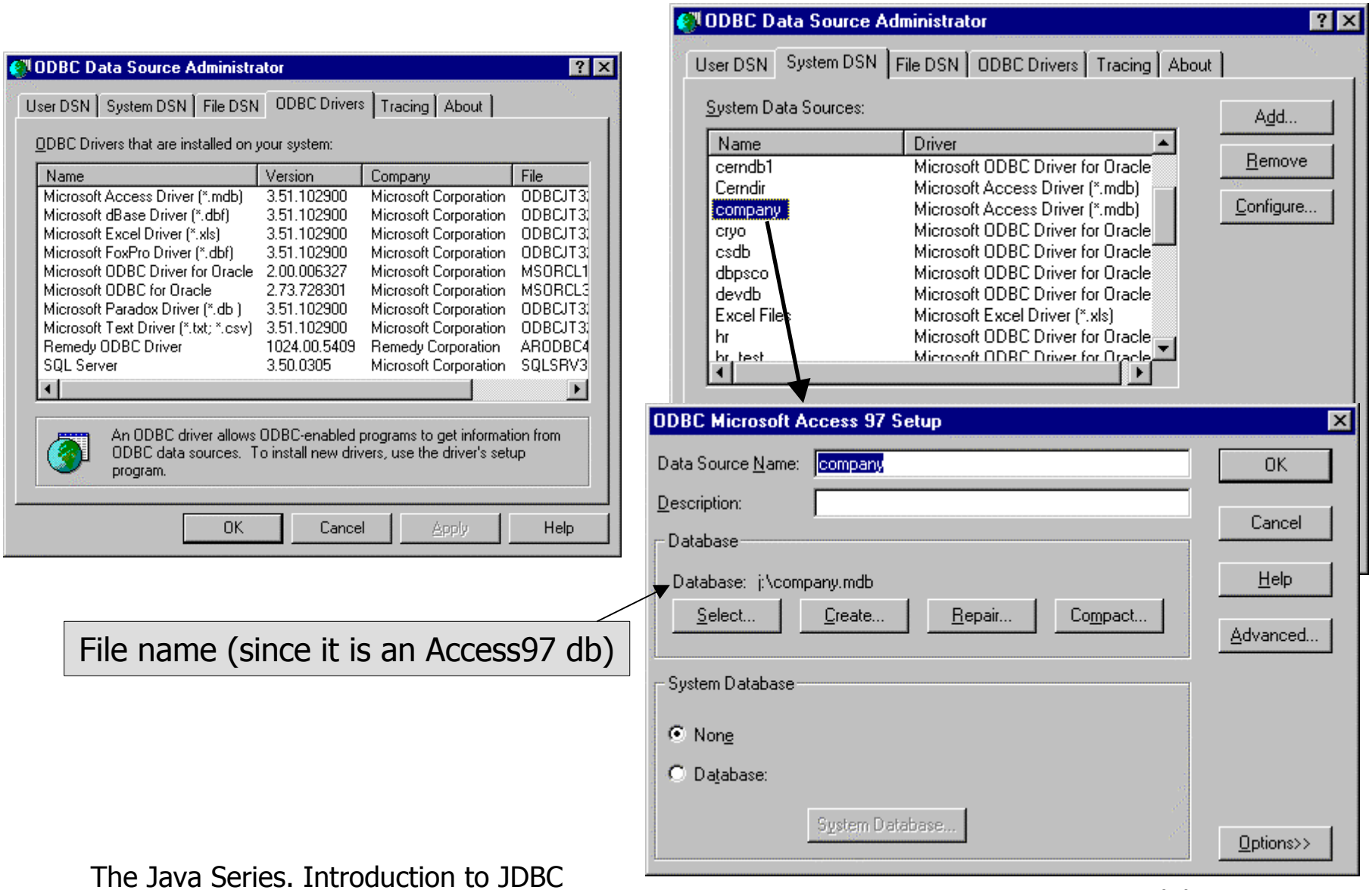
A ODBC Data Source Name is an alias for a database name and a login method.

Create a System DSN for your Access97 file.

When using JDBC ODBC Driver just specify the DSN.

Now access the database as with any other JDBC driver.

Scenario 6. Control Panel Settings



Scenario 6.

```
import java.sql.*;

class ODBCEmployee
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the JDBC ODBC driver
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      System.out.println ("Oracle device driver does not exist");
      System.exit(0);
    }

    Connection conn = DriverManager.getConnection ("jdbc:odbc:company");

    // Create a Statement
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("select ENAME, JOB from EMP");

    // Iterate through the result and print the employee names
    while (rset.next ())
      System.out.println (rset.getString (1));

    // Close the RseultSet
    rset.close(); stmt.close(); conn.close();
  }
}
```

Load JDBC ODBC Driver

When connecting we just refer to the data source

Then we just perform database operations as with any other JDBC database.

Calling Stored Procedures

Databases can have pieces of code called STORED PROCEDURES

Typically stored procedures encapsulate a series of db operations in one single call

This way, db users (including programmers) call the stored procedure instead of performing a series of db operations.

Stored procedures are defined, managed IN the database

From JDBC we can also call stored procedures

Stored Procedures in Oracle

In Oracle stored procedures are programmed in the PL/SQL Language.

PL/SQL is a functional language through which we can also issue SQL statements.

Also, Oracle is aware of PL/SQL: if we change the structure of a table, Oracle can tell us what stored procedures might fail (for instance, because they are using fields we have removed or modified).

Thus, it s CONVENIENT to have the business logic to access the DB as stored procedures, having users and other programmers accessing the db through calls to the procedures.

Scenario 7. The Stored Procedure

```
-- create the package header
create or replace package r is
type cursorType is ref cursor;
function send_cursor (x number) return cursorType;
end;

-- create the package body
create or replace package body r is
function send_cursor(x number) return cursorType is
        l_cursor    cursorType;
    begin
        update mytable set t1=t1*x;
        open l_cursor for
        select t1,t2 from mytable order by t1;
        return l_cursor;
        end;
end;
```

The procedure is in a package

Functions declaration

Function implementation

This is PL/SQL. We have an update statement, and a select statement. The function returns a cursor (the result of a select)

Thanks Eric Grancher for the example and comments

Sce 7: The Java program

```
// Oracle JDBC driver
String driver_class = "oracle.jdbc.driver.OracleDriver";
String connect_string = "jdbc:oracle:thin:@oradev:10521:D";
Connection conn;
Class.forName(driver_class);
conn = DriverManager.getConnection(connect_string,"grancher", ”xxxx");
conn.setAutoCommit(false);

// prepare the call to the function, register the fact
// type of the out parameter (being a cursor)
String query = "begin :1 := r.send_cursor("+args[0]+"); end;";
CallableStatement cstmt = conn.prepareCall(query);
cstmt.registerOutParameter(1,OracleTypes.CURSOR);

// execute the SQL call and retrieve the output
cstmt.execute();
ResultSet rset = (ResultSet)cstmt.getObject(1);
conn.commit();

while (rset.next ())
        System.out.println(rset.getInt(1)+"--"+rset.getString(2));
cstmt.close();
```

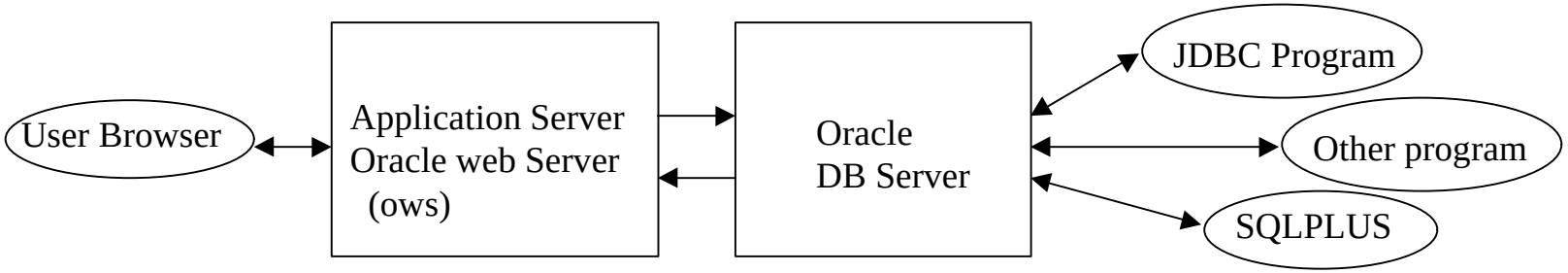
Build the stored procedure invocation

Just use a CallableStatement

Loop through the result

ORACLE and Java

Oracle is integrating Java all over its architecture



For some URL requests the ows can invoke an stored procedure in the db, and return to the browser the HTML generated by the stored procedure.

The Oracle www server can also run servlets which may in turn access the db out-of-the box.

Also, Oracle has just included the possibility to define stored procedures in Java.

Oracle and Java

This results in having Java code being accessed and executed in many different ways:

- Applets served by the ows are executed on the client browser.
- Servlets are stored in the ows and run in the ows JVM
- Java Stored Procedures. Stored in the db and executed by the db whenever invoked. The Oracle db server contains also a JVM.

This, in turn, results in Java stored procedures being invoked by:

- Applets through www browser and JDBC
- Servlets through ows and JDBC
- Other java code (such as scenario 7) directly through JDBC
- Other programs in any other language.

Oracle and Java

Oracle will also allow the db server s JVM to execute Servlets -> less network interaction, tighter integration.

The aim is to separate INTERFACING LOGISTICS from BUSINESS LOGIC from DATABASE ACCESS.

This way: encapsulate business logic in Java BEANS, interfacing logistics with servlets, and database access with stored procs.

This kind of architecture will be highly recommended in the future.

Business logistics used from servlets, from other java apps.

Database access used from Java Beans, servlets, other apps.

Java Considerations

Any Java consideration applies to JDBC:

- Portability, Performance, etc.

There is a lot of work being done in Oracle to improve performance and functionality and also to provide a high and modular level of integration

See **`www.oracle.com/java`** for more information.

Summary

We have done SQL operations:

- SELECT
- UPDATE
- CREATE TABLE
- INSERT
- DELETE

Used Prepared Statements

Used Transactions

Interfaced with ODBC

Called stored procedures

Seen the ORACLE www architecture