

Java DataBase Connectivity (JDBC)

- Presentation of JDBC
- JDBC Drivers
- Work with JDBC
- Interfaces and JDBC classes
- Exceptions
- SQL requests
- Transactions and exceptions

- JDBC (*Java Data Base Connectivity*) is the basic API for accessing **relational databases** with the SQL language, from Java programs
- Provided by package `java.sql`
- The JDBC API is nearly completely independant from SGBDs
- Supplies homogenous access to RDBMSs, by providing an abstraction of the targeted databases

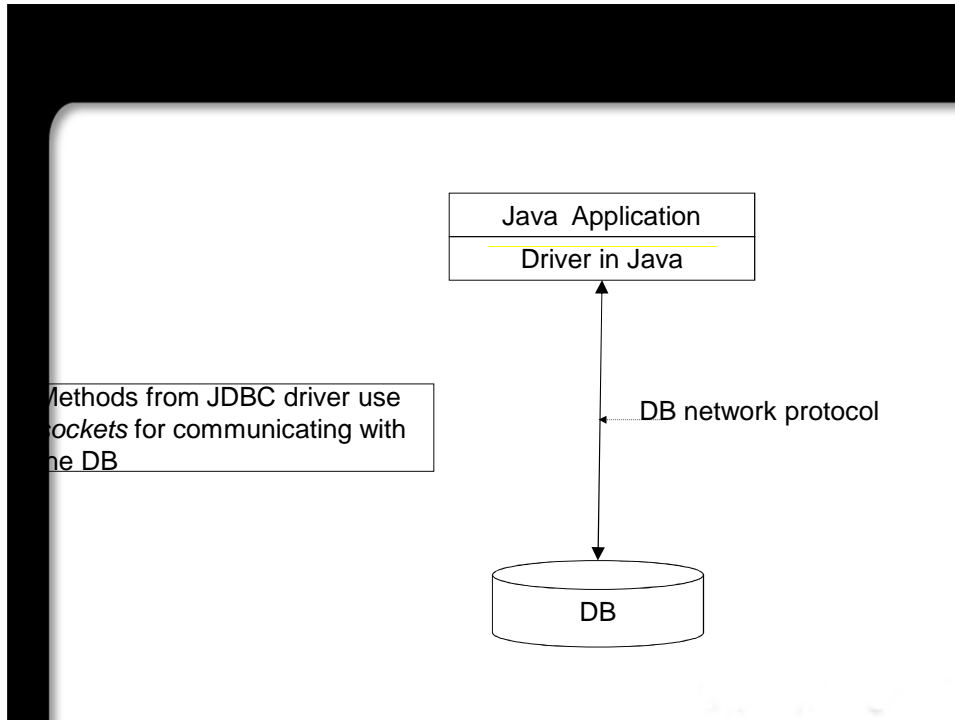
- First JDBC versions supported only *SQL-2 Entry Level*
- JDBC 2, 3 and 4 support SQL3
- JDBC 4 comes with Java 6 and add several goodies for easier use like « automatic driver loading » and support for LOBs, etc.

- Many interfaces, few classes
- Interfaces = API for the developer
- JDBC do not provide the classes that implements these interfaces, it's the driver !

JDBC drivers

- Classes that implements JDBC interfaces
- Database dependants
- All Database vendors propose a JDBC driver

- **Type 1** : JDBC-ODBC bridge
- **Type 2** : uses native function from the Database API (often written in C)
- **Type 3** : allow the use of a middleware driver
- **Type 4** : 100% Java driver that uses the network protocol of the Database
 - Modern drivers are all of type 4 !



Work with JDBC

- Driver classes must be in the classpath
- .java files that use JDBC must import `java.sql` :
`import java.sql.*;`
- Until JDBC 4, it was necessary to load in memory the driver classes for example:
`Class.forName("oracle.jdbc.OracleDriver");`
- No need to do this anymore with JDBC 4! Examples and explanations in a few slides...

1. Get a connection object (of type `connection`)
2. Create SQL instructions (`statement`, `PreparedStatement` Or `CallableStatement`)
3. Execute these instructions :
 - Query the DB (`executeQuery`)
 - Modify the DB content (`executeUpdate`)
 - Or any other kind of order (`execute`)
- (4. Validate or invalidate the transaction with `commit` Or `rollback`)
5. Close the connection (`close()`)

Classes and JDBC interfaces

- **Driver** : returns an instance of **Connection**
- **Connection** : connection to the DB
- **Statement** : SQL order
- **PreparedStatement** : compiled once by the driver and may be executed many times with different parameters
- **CallableStatement** : stored procedures in the DB
- **ResultSet** : lines that came back after a SELECT order
- **ResultSetMetaData** : description of lines returned

- **DriverManager** : manages drivers and connections
- **Date** : SQL date
- **Time** : SQL hours, minutes, seconds
- **TimeStamp** : date and hour, microsecond accuracy,
- **Types** : constants for SQL types (for conversion to Java types)

- The **connect** method from **Driver** takes as parameter a URL and other informations such as login/password and returns an instance of **Connection**
- This instance of **Connection** will be use to execute requests to the DB
- **connect** returns **null** if the driver is not correct or if parameters are not valid
- **Used by DriverManager ; not visible by the developer.**

- A URL for a database looks like this:

`jdbc:subprotocol:databaseName`

- Example:

`jdbc:oracle:thin:@sirocco.unice.fr:1521:INFO`

- `oracle:thin` is a subprotocol (driver « *thin* » ; Oracle provides also another heavier driver)
- `@sirocco.unice.fr:1521:INFO` designs the INFO database located on a server named sirocco that runs an oracle DB on port 1521

- The **DriverManager** class is the one to use !
- Note: with JDBC 1-3 it was necessary to load by hand the driver class before using DriverManager. With JDBC 4 this is useless

```
Class.forName("oracle.jdbc.OracleDriver");
```

- Ask the DriverManager class through its static method getConnection():

```
String url =  
"jdbc:oracle:thin:@sirocco.unice.fr:1521:INFO";  
Connection conn =  
    DriverManager.getConnection(url,  
                                "toto", "pass");
```

- The DriverManager class tries all drivers registered in the classpath until one gives a response that is not null.

- Connections are costly objects, it takes time to obtain one,
- Furthermore, an instance of Connection cannot be shared by several *threads*, it is not thread-safe!
- **Best practice for web applications: use a connexion pool provided by a DataSource object.**
- **Example next slide**, note that we removed all necessary try catch finally etc.

```
import javax.sql.DataSource;
import java.sql.Date;

@ApplicationScoped @JDBC
public class BookRepositoryJDBCImpl implements BookRepository {
    @Resource(name = "jdbc/bookstore") // defined in the application server
    private DataSource dataSource;

    addBook(String title, String description) {
        Connection connection = dataSource.getConnection();
        PreparedStatement preparedStatement = connection.prepareStatement("insert into
book(title, description, price, pubdate) values (?, ?, ?, ?)");
        preparedStatement.setString(1, title);
        preparedStatement.setString(2, description);

        int rowCount = preparedStatement.executeUpdate();
        connection.close();
    }
}
```

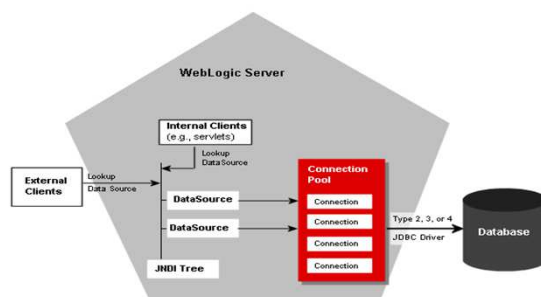
DataSource

- Provides access to connections in a shared resources environment
- Generally provides advanced features:
 - Connection pooling
 - PreparedStatement pooling
 - Distributed transactions (XA protocol)
- Retrieved using the @Resource annotation in Servlets, Beans, EJBs
- Or retrieved from a JNDI naming service:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("DataSourceName");
```

ConnectionPoolDataSource

- Connection pool architecture.



A connection pool contains a group of JDBC connections that are created when the connection pool is registered—when starting up WebLogic Server or when deploying the connection pool to a target server or cluster. Connection pools use a JDBC driver to create physical database connections. Your application borrows a connection from the pool, uses it, then returns it to the pool by closing it.

- By default, a connection is in « *auto-commit* » mode: a commit is automatically performed after any SQL order that modifies the DB content.
- **Best practice: set this setting off**
`conn.setAutoCommit(false)`
- The transaction will be manually validated or invalidated using:
 - `conn.commit()`
 - `conn.rollback()`

Connection best practice

- Create a method that does:
 - Get the connexion,
 - Set the proper settings for transactions
 - Begin a transaction
 - Run a SQL statement, get results,
 - Commit or rollback the transaction,
 - Close the connection (note that in case of a connection pool, this will not « really » close the connection to the DB)

- We will practice that with the « BookRepository using JDBC » exercice.

Transactions

- A transaction is a sequence of actions that must be considered as a whole: it can be committed or rolled back but cannot be partially executed
- The default mode is autocommit: each query is committed as soon as it is send
- The Connection offers methods to manage the transaction mode:

```
c.setAutoCommit(false);  
c.commit();  
c.rollback();
```

Interactions

- Possible interactions between transactions:
 - Dirty reads (DR):
 - Transactions can see uncommitted changes
 - Nonrepeatable reads (NR):
 - Transaction A reads a row
 - Transaction B changes the row
 - Transaction A reads the same row (changed)
 - Phantom reads (PR):
 - Transaction A reads all rows satisfying a where
 - Transaction B inserts a row satisfying the where
 - Transaction A rereads and see the new row

Isolation levels

- Isolation level can be set on the Connection. The higher the level, the worse the performance, the lower the risk
- Isolation levels:
 1. TRANSACTION_READ_UNCOMMITTED: DR, NR, PR
 2. TRANSACTION_READ_COMMITTED: NR, PR
 3. TRANSACTION_REPEATABLE_READ: PR
 4. TRANSACTION_SERIALIZABLE: no interaction, each transaction totally locks all the data it needs

- The transaction isolation level can be modified using a method from the Connection object:

```
conn.setTransactionIsolation(  
    Connection.TRANSACTION_SERIALIZABLE);
```

Savepoints

- Savepoints provide fine-grained control of transaction
- Each savepoint is represented by a named object:

```
Savepoint sp1 = c.setSavepoint("SP1");
```

- A transaction can be then rolled back to a savepoint:

```
c.rollback(sp1);
```

- A savepoint can be released:

```
c.releaseSavepoint(sp1);
```

Execute a SQL order

- Use an instance of **Statement** created using the `createStatement()` method from the `Connection`:

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

- When a `Statement` has been created, we can use it for different kind of orders (`executeQuery` or `executeUpdate`)
- Several statements can be retrieved from the same connection to be used concurrently

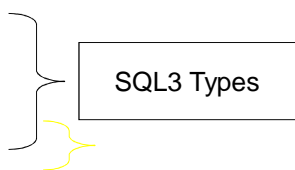
- The method to call depends on the nature of the SQL request:
 - Search (select) : **executeQuery** returns a ResultSet for processing lines one by one
 - Modify data (update, insert, delete) or DDL orders (create table,...) : **executeUpdate** returns the number of lines affected
 - If we don't know at execution time the nature of the SQL order: **execute**

```
Statement stmt = conn.createStatement();  
// rset contains returned lines  
ResultSet rset =  
    stmt.executeQuery("SELECT nameE FROM emp");  
// Get each line one by one  
while (rset.next())  
    System.out.println (rset.getString(1));  
    // or . . . (rset.getString("nameE"));  
stmt.close();
```

First column has number 1

- `executeQuery()` returns an instance of `ResultSet`
- `ResultSet` will give access of lines returned by `select`
- At the beginning, `ResultSet` is positioned before the first line, we must call `next()` first
- `next()` returns `true` if the next line exists, and `false` otherwise

- When `ResultSet` is positioned on one line, the `getXXX` methods are used to retrieve data:
 - `getXXX(int columnNumber)`
 - `getXXX(String columnName)`
 - `XXX` is the Java type of the value we are going to get. For example `String`, `Int` or `Double`
- For example, `getInt` returns an `int`

- CHAR, VARCHAR, LONGVARCHAR
 - BINARY, VARBINARY, LONGVARBINARY
 - BIT, TINYINT, SMALLINT, INTEGER, BIGINT
 - REAL, DOUBLE, FLOAT
 - DECIMAL, NUMERIC
 - DATE, TIME, TIMESTAMP
 - BLOB, CLOB
 - ARRAY, DISTINCT, STRUCT, REF
 - JAVA_OBJECT
- 
- SQL3 Types

- getString() can retrieve nearly any SQL type
- However, edicated methods are recommended:
 - **CHAR** and **VARCHAR** : getString, **LONGVARCHAR** : getAsciiStream and getCharacterStream
 - **BINARY** and **VARBINARY** : getBytes, **LONGVARBINARY** : getBinaryStream
 - **REAL** : getFloat, **DOUBLE** and **FLOAT** : getDouble
 - **DECIMAL** and **NUMERIC** : getBigDecimal
 - **DATE** : getDate, **TIME** : getTime, **TIMESTAMP** : getTimestamp

- From `java.util.Date` to `java.sql.Date`, use the `getTime()` method:

```
■ java.util.Date date = new
  java.util.Date();
  java.sql.Date dateSQL =
    new java.sql.Date(date.getTime());
  java.sql.Time time =
    new Time(date.getTime());
  java.sql.Timestamp time =
    new Timestamp(date.getTime());
```

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(
    "SELECT nameE, reward FROM emp");
while (rset.next()) {
    name= rset.getString("nameE");
    commission = rset.getDouble("reward");
    if (rset.wasNull())
        System.out.println(name + ": no
        reward");
    else
        System.out.println(name + " has reward:
        " + commission + "%");
```

```
Statement stmt = conn.createStatement();  
String city= "NICE";  
int nbLinesModified = stmt.executeUpdate(  
    "INSERT INTO dept (dept, name, place"  
    + "VALUES (70, 'DIRECTION'," +  
    + "'" + city+ "')");
```

Do not forget
the space here!

- Most DB can analyse only once a request executed many times during a connection
- JDBC takes advantage of this with parameterized requests associated to instances of the **PreparedStatement** interface (that inherits from the **Statement** interface)

```
PreparedStatement pstmt =  
    conn.prepareStatement("UPDATE emp SET sal =  
    ?" + " WHERE name = ?");
```

- "?" indicates parameter locations
- This request will be executable with several parameter sets: (2500, 'DUPOND'), (3000, 'DURAND'), etc.

```
PreparedStatement pstmt =  
    conn.prepareStatement(  
        "UPDATE emp SET sal = ? "  
        + "WHERE nomE = ?");  
for (int i=0; i<10; i++) {  
    pstmt.setDouble(1, employe[i].getSalaire());  
    pstmt.setString(2, employe[i].getNom());  
    pstmt.executeUpdate();  
}
```

Starts at 1, not 0!

- Use `setNull(n, type)` (type is one of the `Types`)
- Or pass `null` as a parameter if `setXXX()` has an `Object` as parameter (i.e. `setString`, `setDate`,...)

- Faster if used several times, cacheable in case of using a connection pool, even after the connection has been closed on client side,
- Better portability as `setXXX` methods handle differences between DBRMs
- They protect against SQL injection

- Like `PreparedStatement`, `CallableStatement` are compiled but also grouped in the DBRM.
- They are used to call stored procedures,
- Less network access means better performance but as callable statements very often uses particularity of DBRMs, the portability is affected.

```
create or replace procedure augment
  (oneDept in integer, percentage in
  number,
  cost out number) is
begin
  select sum(sal) * percentage / 100
    into cost
  from emp
  where dept = oneDept;
update emp
  set sal = sal * (1 + percentage /
100)
```


- **CallableStatement** inherits from **PreparedStatement**
- An instance of **CallableStatement** created by calling the **prepareCall** method from **Connection** interface
- This method accepts as parameter a String that describes how to call the stored procedure and if it returns a value or not.

- Not standardized, differs from one DBRM to another
 - JDBC uses its own syntax to address this problem
 - If stored procedure returns a value:
{ ? = **call** procedure-name(?, ?,...) }
 - Returns no value:
{ **call** interface (?, ?,...) }
 - No parameter:
{ **call** interface }
- Driver will translate into DBRM syntax

```
CallableStatement cstmt =  
    conn.prepareStatement("{? = call  
        augment(?,?)}");
```

1. In and out parameters passed with **setXXX**
Paramètres types for « out » and « in/out » set
with registerOutParameter method
2. **Execute** with either **executeQuery**,
executeUpdate or **execute**,
3. « out », « in/out », and eventually the return
value, collected with **getXXX**

```
CallableStatement csmt = conn.prepareCall(
    "{ call augment(?, ?, ?) }");
    / 2 digits after comma for third parameter
    csmt.registerOutParameter(3, Types.DECIMAL, 2);
    / Augment 2,5 % salaries from dept number 10
    csmt.setInt(1, 10);
    csmt.setDouble(2, 2.5);
    csmt.executeQuery(); // or execute()
    double cost = csmt.getDouble(3);
    System.out.println("Total cost of augmentation:
    " + cost);
```

- JDBC allows getting informations about the data we retrieved with a SELECT (interface **ResultSetMetaData**),
- Also about the Database itself (interface **DatabaseMetaData**)
- Results differ from one DBRM to another

```
ResultSet rs =  
    stmt.executeQuery("SELECT * FROM emp");  
ResultSetMetaData rsmd = rs.getMetaData();  
int nbColumns = rsmd.getColumnCount();  
for (int i = 1; i <= nbColumns; i++) {  
    String typeColumn = rsmd.getColumnTypeName(i);  
    String nomColumn = rsmd.getColumnName(i);  
    System.out.println("Column " + i + " of name "  
        + nomColumn + " of type "  
        + typeColumn);  
}
```

```
private DatabaseMetaData metaData;  
private java.awt.List listTables = new  
    List(10);  
...  
metaData = conn.getMetaData();  
String[] types = { "TABLE", "VIEW" };  
ResultSet rs =  
    metaData.getTables(null, null, "", types);  
String tablesNames;  
while (rs.next()) {  
    nomTable = rs.getString(3);  
}
```

Joker for table
and view names

Auto generated keys

- Execute (or prepare and execute) a statement specifying your will or the keys you want to retrieve:

```
■ ps = c.prepareStatement("...",  
    Statement.RETURN_GENERATED_KEYS);  
■ s.executeUpdate("");  
■ s.executeUpdate("", {"Client_ID"});
```

- Retrieve the keys:

```
ResultSet r = s.getGeneratedKeys();
```

Batch updates

- To optimize performances, you may execute a set of updates at once

- You add your updates to the statement:

```
s.addBatch("INSERT INTO t VALUES (1)");
```

- For prepared and callable statements, you add a set of parameter values:

```
ps.setInt(1, 453);  
ps.addBatch();
```

- The execution returns all the results:

```
int[] tab = s.executeBatch();
```

More on ResultSet

- Interface providing methods for manipulating the result of executed queries
- ResultSet can have different functionality, based on their characteristics:
 - Type
 - Concurrency
 - Holdability

Type

- **TYPE_FORWARD_ONLY:**
 - Forward only (only the next() method is available)
 - Does not reflect changes in the database
- **TYPE_SCROLL_INSENSITIVE:**
 - Scrollable
 - Does not reflect changes in the database
- **TYPE_SCROLL_SENSITIVE:**
 - Scrollable
 - Reflects changes in the database

Concurrency

- **CONCUR_READ_ONLY:**
 - Default concurrency
 - The ResultSet cannot be updated
- **CONCUR_UPDATABLE:**
 - The ResultSet can be updated

Holdability

- **HOLD_CURSORS_OVER_COMMIT:**
 - The ResultSets are held open when a commit operation is performed on the connection
- **CLOSE_CURSORS_AT_COMMIT:**
 - The ResultSets are closed when a commit operation is performed on the connection
- The default holdability is implementation defined

Retrieving information

- The `ResultSet` provides methods to move the cursor:
 - `first()`, `next()`, `previous()`, `last()`, `beforeFirst()`, `afterLast()`
 - `relative(int)`, `absolute(int)`
- The methods `getXXX()` retrieve the value in the column given by number or name to the Java type `XXX`
- The method `wasNull` indicates whether the latest retrieved value was null

JDBC advanced: RowSet

- The `RowSet` extends the `ResultSet`
- It is the representation of the result of a `SELECT` query
- It holds all the connection information it needs (URL, user...) to be autonomous
- The `RowSet` is a `Javabean`, it generates events when a modification of its values occurs
- Good tutorial in french:
<http://java.developpeur.com/faq/jdbc/?page=generalite-rowset#defRowSet>