



# Advanced Programming

## JDBC

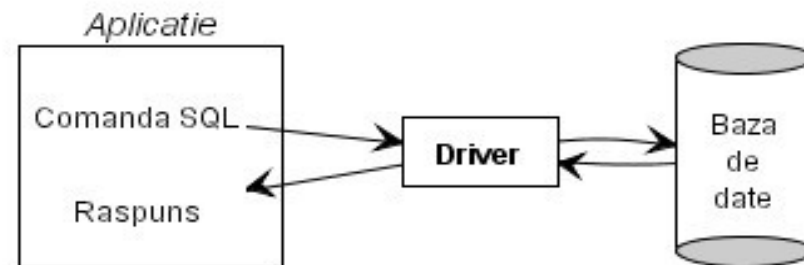
# Databases

- **DB** – Collection of *structured* data
- **DBMS** – A Database Management System offers all the “tools” for: *creating, accessing, updating a db*
- *Efficiency* (indexes, etc.)
- *Consistency* (FK, PK, triggers, etc.)
- *Security* (users, permissions, etc.)
- Models: **relational**, object-oriented, graph, XML, NoSQL, NewSQL, etc.
- Producers: Oracle, Microsoft, Sybase, etc.



# Applications That Use a DB

- Create the database: **SQL script**
- Connect to the database: **driver**
- Communicate with the database:
  - Execution of SQL commands
    - **DDL, DML, DCL**
  - Processing results

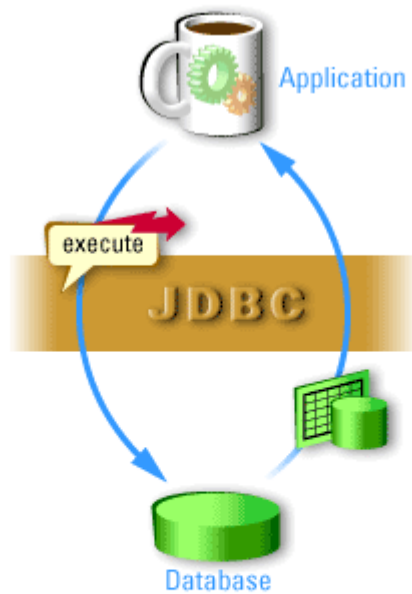


# JDBC

- JDBC (Java Database Connectivity) is a Java API that can access any kind of tabular data, especially data stored in a **relational database**.
- Allows the **integration of SQL statements** into a general programming environment by providing library routines which interface with the database.
- **Independent** of the database type
- Based on **adapters (drivers)** between the client and the DBMS
- **java.sql** – the core JDBC API
- **javax.sql** – Java EE specific

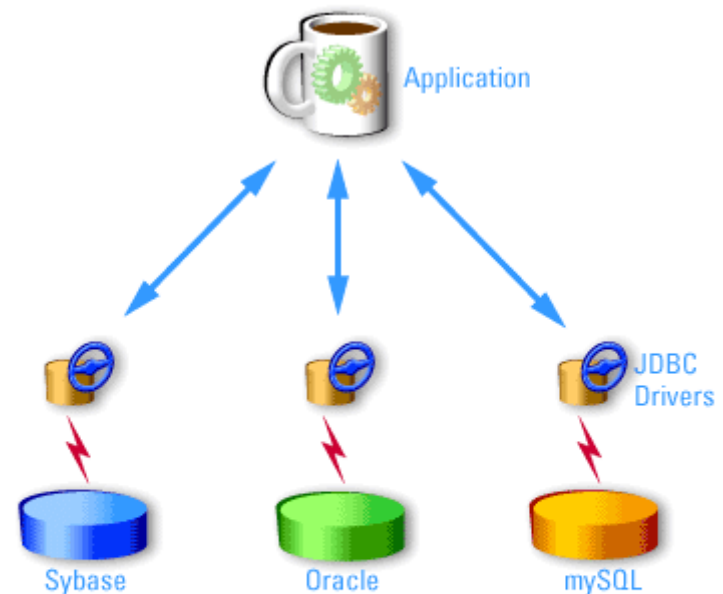
# Driver

The interface between the application and the database



JDBC allows an application to send SQL statements to a database and receive the results.

1 of 5



JDBC interfaces for specific database engines are implemented by a set of classes called JDBC drivers. Since the JDBC driver handles the low-level connection and translation issues, you can focus on the database application development without worrying about the specifics of each database.

2 of 5

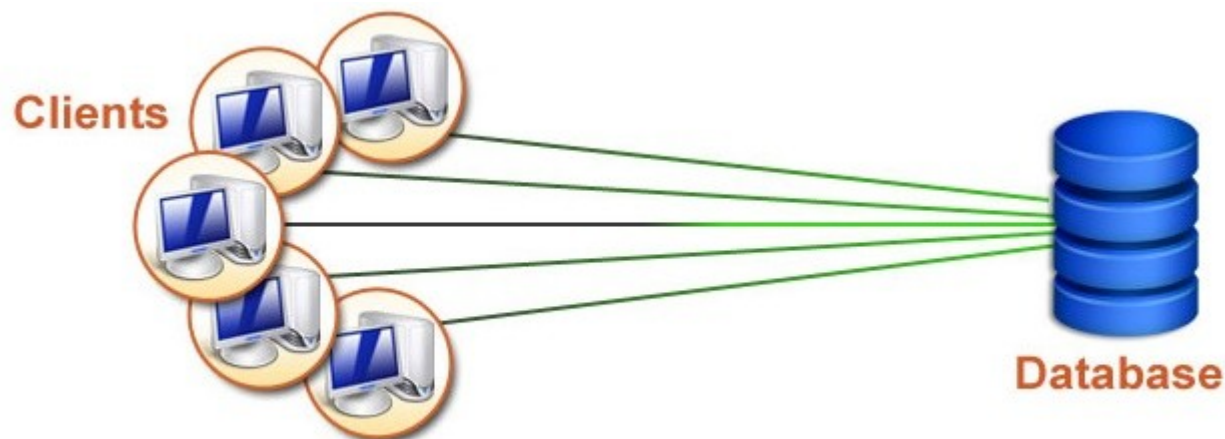
# Using a Specific Driver

- Identifying the specific database driver
  - ✓ for example: *mysql-connector-java.jar*
  - ✓ adding the jar to the CLASSPATH
  - ✓ identifying the driver class: *com.mysql.jdbc.Driver*
- Loading the driver class (Not required anymore)
  - ✓ `DriverManager.registerDriver(new com.mysql.jdbc.Driver());`
  - ✓ `Class.forName("com.mysql.jdbc.Driver").newInstance();`
  - ✓ `System.setProperty("jdbc.drivers", "com.mysql.jdbc.Driver");`
  - ✓ `java -Djdbc.drivers=com.mysql.jdbc.Driver MyApplication`

NOTE: The DataSource interface, new in the JDBC 2.0 API, provides another way to connect to a data source. The use of a DataSource object is the preferred means of connecting to a data source.

# Connections

- **Connection (session)** - A context through which the communication with a database takes place.
- SQL statements are executed and results are returned within the context of a connection.
- An application may create multiple connections (to the same database or to different databases).



# Locating a Database

## JDBC URL

**`jdbc:sub-protocol:identifier`**

The sub-protocol identifies the driver type, for instance:  
*`odbc`, `mysql`, `oracle`, `sybase`, `postgres`, etc.*

The database identifier is usually specific to a protocol:

```
jdbc:postgresql://192.168.0.1:5432/test
```

```
jdbc:mysql://localhost/test
```

```
jdbc:oracle:thin@persistentjava.com:1521:test
```

```
jdbc:sybase:test
```



# Connectiong to a Database

A connection is represented by an object of type *java.sql.Connection*

```
Connection conn = DriverManager.getConnection(url);
```

```
Connection conn = DriverManager.getConnection(  
    url, username, password);
```

```
Connection conn = DriverManager.getConnection(  
    url, dbproperties);
```

Don't forget to close the connection: `conn.close()`

# Example

```
String url = "jdbc:mysql://localhost/test" ;
```

```
Connection con = null;
```

```
try {
```

```
    Connection con = DriverManager.getConnection(  
        url, "myUserName", "mySecretPassword");
```

```
} catch(SQLException e) {
```

```
    System.err.println("Cannot connect to DB: " + e);
```

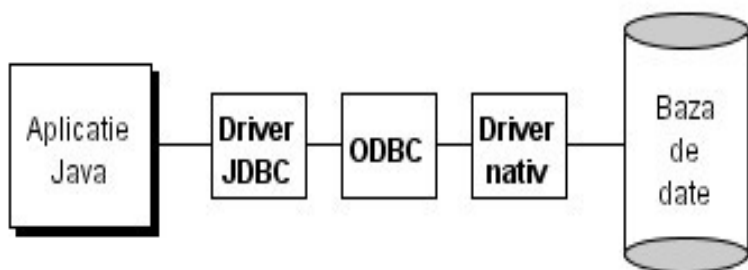
```
} finally {
```

```
    if (con != null) con.close() ;
```

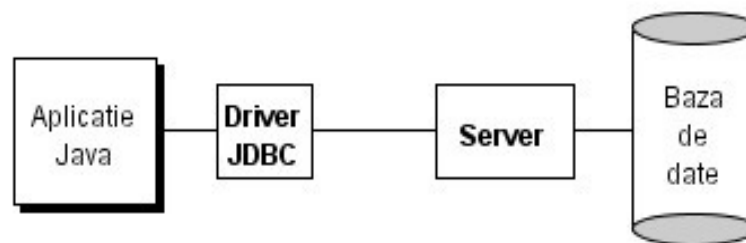
```
}
```

# Driver Types

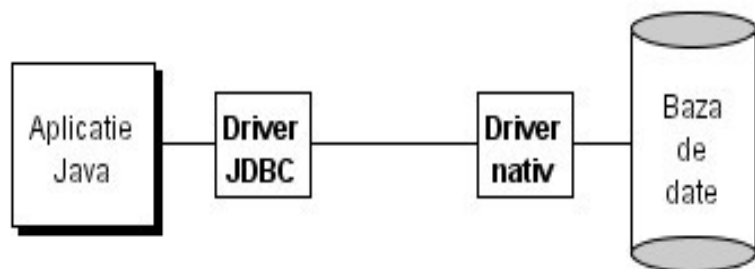
## Type 1



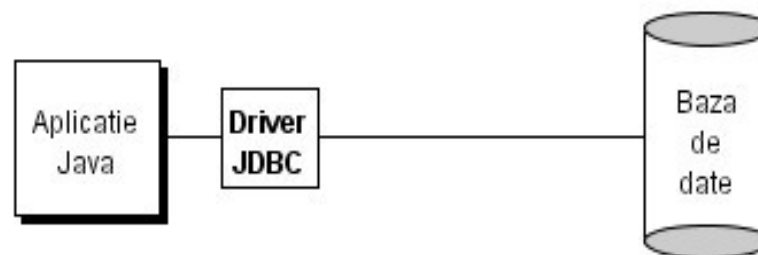
## Type 3



## Type 2



## Type 4



# JDBC-ODBC Bridge (obsolete)

- **ODBC**: Open Database Connectivity
- Driver: *sun.jdbc.odbc.JdbcOdbcDriver*
- URL: *jdbc:odbc:identifier*
  - DSN Identifier (Data Source Name)
- Easy to use, "universal" solution to connect to a database
- Not portable, poor execution speed

*“The JDBC-ODBC Bridge should be considered a transitional solution. It is not supported by Oracle. Consider using this only if your DBMS does not offer a Java-only JDBC driver.”*

# Using Connections

- Creating statements for executing SQL commands and returning the results.
  - *Statement, PreparedStatement,*
  - *CallableStatement*
- Getting the metadata: information regarding the database or the results of queries
  - *DatabaseMetaData, ResultSetMetaData*
- Transaction control
  - *commit, rollback*
  - *setAutoCommit*

# Statement

The object used for executing a **static** SQL statement and returning the results it produces.

- **Creating a *Statement***

```
Connection con = DriverManager.getConnection(url);  
Statement stmt = con.createStatement();
```

- **Executing a query**

```
String sql = "SELECT * FROM persons";  
ResultSet rs = stmt.executeQuery(sql);
```

- **Executing an update or a delete**

```
String sql = "DELETE FROM persons WHERE age < 0";  
int nbRowsAffected = stmt.executeUpdate(sql);  
sql = "DROP TABLE temp";  
stmt.executeUpdate(sql); // Returns 0
```

- **Generic SQL statements**

```
stmt.execute("any kind of SQL command");
```

# PreparedStatement

An object that represents a precompiled SQL statement.

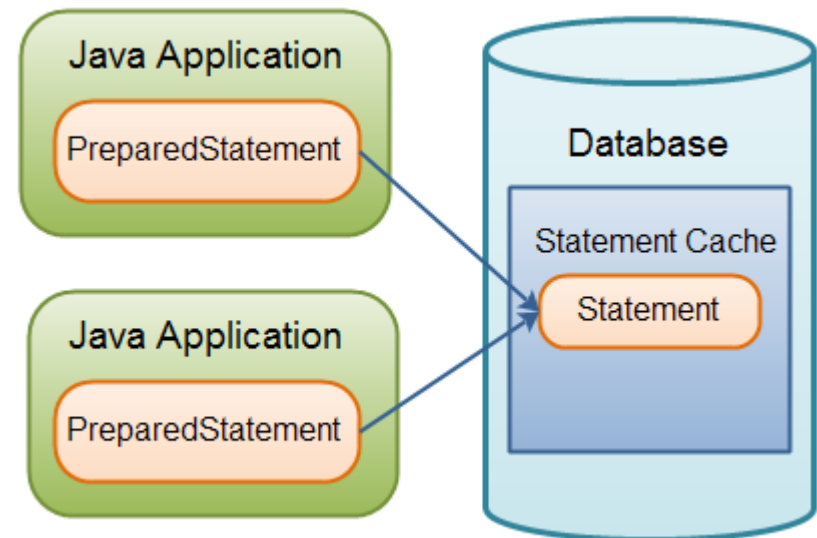
An SQL statement is precompiled and stored in a *PreparedStatement* object. This object can then be used to efficiently execute this statement multiple times.

→ Batch Commands

```
String sql = "UPDATE persons SET name = ? WHERE id = ?";  
Statement pstmt = con.prepareStatement(sql);
```

```
pstmt.setString(1, "Ionescu");  
pstmt.setInt(2, 100);  
pstmt.executeUpdate();
```

```
pstmt.setString(1, "Popescu");  
pstmt.setInt(2, 200);  
pstmt.executeUpdate();
```



# JDBC Data Types

*java.sql.Types* → defines the constants that are used to identify generic SQL types, called JDBC types.

## *Java Data Types – SQL Data Types*

setObject - If arbitrary parameter type conversions are required, the method *setObject* should be used with a target SQL type.

```
pstmt.setObject(1, "Ionescu", Types.CHAR);  
pstmt.setObject(2, 100, Types.INTEGER); // or simply  
pstmt.setObject(2, 100);
```

## setNull

```
pstmt.setNull(1, Types.CHAR);  
pstmt.setInt(2, null);
```



# Handling Large Values

**setBinaryStream, setAsciiStream, setCharacterStream**

When a very large binary or char value is input to a *LONG Type* parameter, it may be more practical to send it via a stream object. The data will be read from the stream as needed until end-of-file is reached.

```
File file = new File("someFile");  
  
InputStream fin = new FileInputStream(file);  
  
java.sql.PreparedStatement pstmt =  
    con.prepareStatement(  
        "UPDATE files SET contents = ? " +  
        "WHERE name = 'someFile'");  
  
pstmt.setBinaryStream (1, fin);  
  
pstmt.executeUpdate();
```

# CallableStatement

The interface used to execute SQL stored procedures.

```
//Creating a CallableStatement
Connection con = DriverManager.getConnection(url);
CallableStatement cstmt = con.prepareCall(
    "{call myStoredProcedure(?, ?)}");

//Setting the IN parameters
cstmt.setString(1, "Ionescu");
cstmt.setInt(2, 100);

//Registering the OUT parameters
cstmt.registerOutParameter(1, java.sql.Types.FLOAT);

//Executing the call and retrieving the results
cstmt.executeQuery();
float result = cstmt.getDouble(1);
```

# ResultSet

A *table of data* representing a *database result set*, which is usually generated by executing a statement that queries the database.

```
Statement stmt = con.createStatement();  
String sql = "SELECT id, name FROM persons";
```

```
ResultSet rs = stmt.executeQuery(sql);
```

id	name
100	Ionescu
200	Popescu

```
while (rs.next()) {  
    int cod = rs.getInt("id"); //rs.getInt(1)  
    String nume = rs.getString("name");  
    System.out.println(id + ", " + name);  
}
```

A *ResultSet* object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The *next* method moves the cursor to the next row.

# Scrollable and Modifiable Cursors

```
Statement stmt = con.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
String sql = "SELECT id, name FROM persons";  
ResultSet rs = stmt.executeQuery(sql);  
    // rs will be scrollable,  
    // will not show changes made by others  
    // and will be updatable
```

## Additional Methods

- absolute
- updateRow
- moveToInsertRow
- insertRow
- moveToCurrentRow
- deleteRow

## supports Positioned Update/Delete

A default *ResultSet* object is not updatable and has a cursor that moves forward only.

# RowSet

Adds support to the JDBC API for the JavaBeans component model

- Extends *ResultSet*
- Conforms to JavaBeans specifications
  - Properties
  - Supports JavaBeans events
- *JdbcRowSet*
- *CachedRowSet* (disconnected)
- *WebRowSet* (XML)
- *JoinRowSet* (offline join)
- *FilteredRowSet* (offline filtering)

# Examples

```
JoinRowSet jrs = new JoinRowSetImpl();
```

```
ResultSet rs1 = stmt.executeQuery("SELECT * FROM EMPLOYEES");  
CachedRowSet empl = new CachedRowSetImpl();  
empl.populate(rs1);  
empl.setMatchColumn(1);  
jrs.addRowSet(empl);
```

```
ResultSet rs2 = stmt.executeQuery("SELECT * FROM BONUS_PLAN");  
CachedRowSet bonus = new CachedRowSetImpl();  
bonus.populate(rs2);  
bonus.setMatchColumn(1); // EMP_ID is the first column  
jrs.addRowSet(bonus);
```

```
FilteredRowSet frs = new FilteredRowSetImpl();  
frs.populate(rs1);  
Range name = new Range("Ionescu", "Popescu", "EMP_NAME");  
frs.setFilter(name); //accepts Predicate objects  
frs.next();
```

# DatabaseMetaData

Comprehensive information about the database as a whole.

Implemented by driver vendors to let users know the capabilities of a DBMS in combination with the JDBC driver that is used with it → *tables, stored procedures, connection capabilities, supported SQL grammar, etc.*

```
Connection con = DriverManager.getConnection (url);
```

```
DatabaseMetaData dbmd = con.getMetaData();
```

```
// Get the tables of the database
```

```
ResultSet rs = dbmd.getTables (null, null, null, null);
```

```
// catalog, schemaPattern, tableNamePattern, types)
```

```
while (rs.next ())
```

```
    System.out.println(rs.getString ("TABLE_NAME"));
```

```
    con . close ();
```

```
}
```

# *ResultSetMetaData*

Information about the types and properties of the columns in a *ResultSet* object: the number of columns, their types, their names, etc.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM someTable");
```

```
ResultSetMetaData rsmd = rs.getMetaData();
```

```
// Find the number of columns in the ResultSety
```

```
int n = rsmd.getColumnCount();
```

```
// Find the names of the columns
```

```
String nume[] = new String[n];
```

```
for(int i=0; i<n; i++) {
```

```
    nume[i] = rsmd.getColumnName(i);
```

```
}
```



# Transaction Control

- Transaction = An ACID unit of work
- ACID = Atomic, Consistent, Isolated, Durable
- COMMIT, ROLLBACK

```
con.commit();  
con.rollback();
```

- Savepoints

```
Savepoint save1 = con.setSavepoint();
```

```
...
```

```
con.rollback(save1);
```

- Disabling the *AutoCommit* Mode

```
con.setAutoCommit(false);
```

# Handling *SQLException*

- *SQLException*

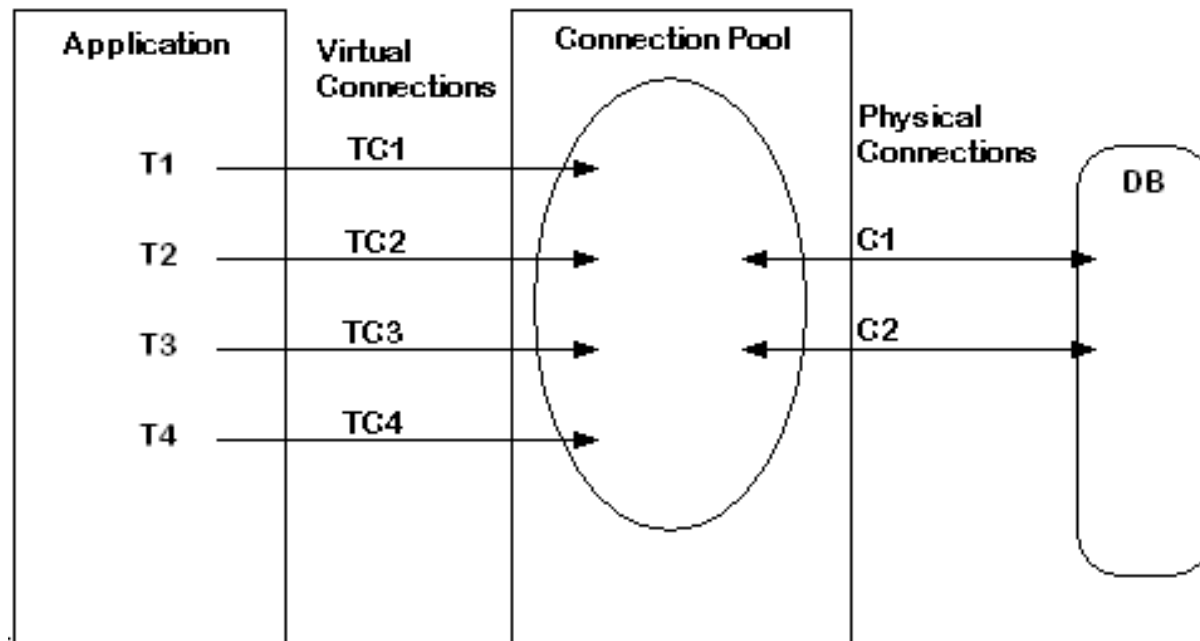
```
public static void printSQLException(SQLException ex) {  
    for (Throwable e : ex) { //SQLException implements Iterable<Throwable>  
        //chained exceptions  
        if (e instanceof SQLException) {  
            SQLException sqlEx = (SQLException)e;  
            System.err.println("SQLState   : " + sqlEx.getSQLState());  
            System.err.println("Error Code: " + sqlEx.getErrorCode());  
            System.err.println("Message   : " + sqlEx.getMessage());  
            Throwable t = ex.getCause();  
            while(t != null) {  
                System.out.println("Cause: " + t);  
                t = t.getCause();  
            }  
        }  
    }  
}
```

- *SQLWarning* (for example, *Data Truncation*)  
*Connection, Statement, ResultSet - getWarnings()*

# Connection Pools

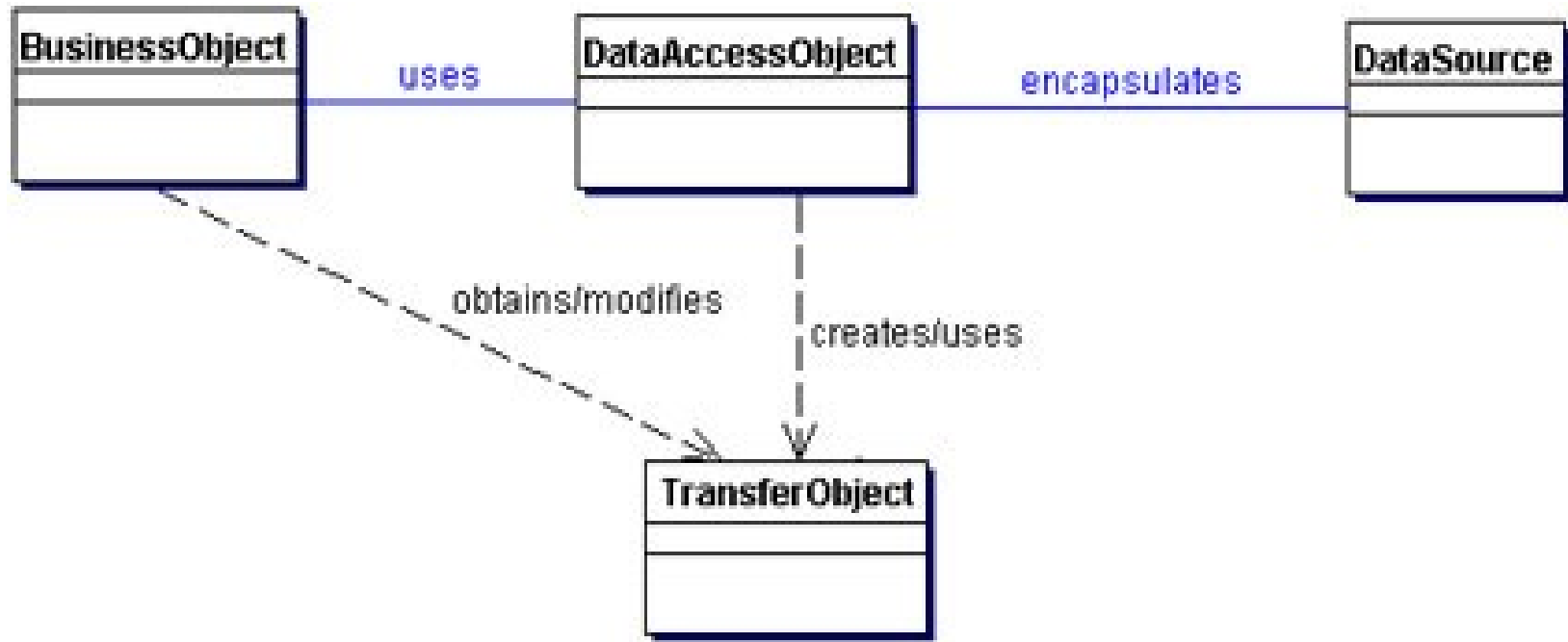
Reusable set (cache) of database connections

- Data access pattern designed at reducing the overhead involved in performing database connections.



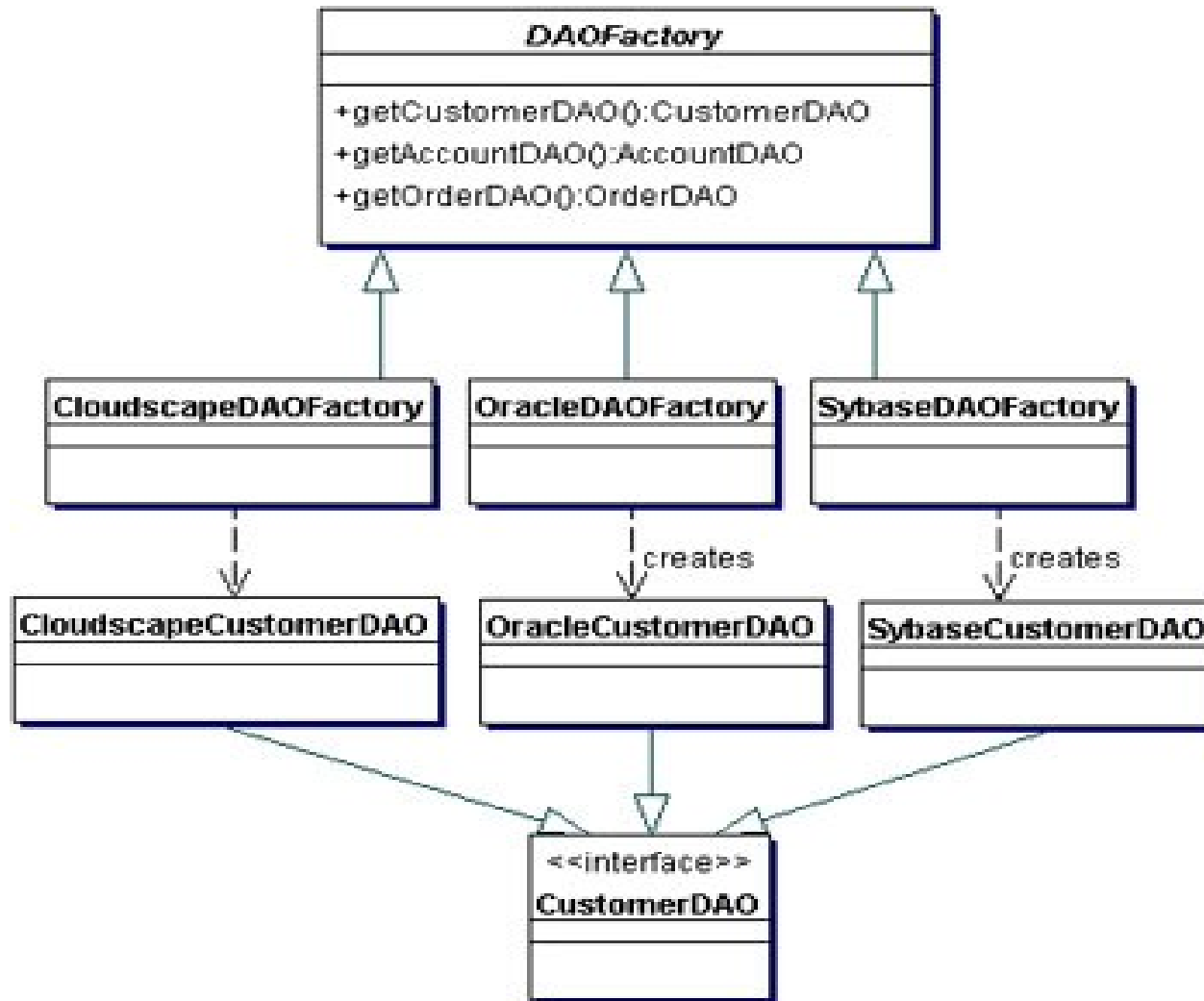
- Apache Commons DBCP, C3PO, HikariCP, etc.

# Data Access Objects (DAO)



- *BusinessObject* – the object that must access the data
- *DataAccessObject* - abstracts and encapsulates all operations related to the data
- *DataSource* - RDBMS, OODBMS, XML, etc.
- *TransferObject* – a representation of the data: entities, beans, etc.

# Abstract Factory



# Java Tutorial

Trail: JDBC(TM) Database Access

<http://docs.oracle.com/javase/tutorial/jdbc/TOC.html>