Database access and JDBC

Outline

1. Introduction to JDBC
2. Accessing a database: practical steps
3. Prepared statements
4. Design patterns (DAO)
5. Object-Relational Mapping
6. Connection pooling

Introduction to JDBC

Database access and JDBC
Goals

- Enable Java applications to access data stored in Relational Data Bases
  - Query existing data
  - Modify existing data
  - Insert new data

- Data can be used by
  - The algorithms running in the application
  - The user, through the user interface
Goals (for Web Applications)

- Access SQL DBMS’s from JSP pages
  - JDBC technology
- Integrate SQL query results into the resulting HTML content
- Generate SQL queries according to FORM values
Goals (for GUI Applications)

- Access SQL DBMS’s from the JavaFX application
  - JDBC technology
- Load ‘massive’ data directly from database
- Query ‘on-demand’ information from database
- Store computation results
JDBC

- Standard library for accessing relational databases
- Compatible with most/all different databases
- JDBC : Java Database Connectivity
- Defined in package java.sql and javax.sql
- Documentation:
  - JDBC Overview: [http://www.oracle.com/technetwork/java/overview-141217.html](http://www.oracle.com/technetwork/java/overview-141217.html)
  - Tutorial [http://docs.oracle.com/javase/tutorial/jdbc/TOC.html](http://docs.oracle.com/javase/tutorial/jdbc/TOC.html)
JDBC scope

- **Standardizes**
  - Mechanism for connecting to DBMSs
  - Syntax for sending queries
  - Structure representing the results

- **Does not standardize**
  - SQL syntax: dialects, variants, extensions, ...
Architecture
Main elements

- Java application (in our case, JavaFX)
- JDBC Driver Manager
  - For loading the JDBC Driver
- JDBC Driver
  - From DBMS vendor
- DBMS
  - In our case, MySQL
Accessing a database: practical steps

Database access and JDBC
Basic steps

1. Define the connection URL
2. Establish the connection
3. Create a statement object
4. Execute a query or update
5. Process the results
6. Close the connection
JDBC Driver

- A Driver is a DMBS-vendor provided class, that must be available to the Java application
  - In general: Should reside in Project’s libraries
- The application usually doesn’t know the driver class name until run-time (to ease the migration to other DMBSs)
- Needs to find and load the class at run-time
  - Class.forName method in the Java Class Loader (not needed in recent versions)
MySQL JDBC driver

- MySQL Connector/J
  - Provides mysql-connector-java-[version]-bin.jar
  - Copy into CLASSPATH
    - E.g.: c:\Program files\...\jre...\lib\ext
  - Copy into project libraries
  - Copy into Tomcat’s libraries

- The driver is in class
  - com.mysql.jdbc.Driver
1. Define the connection URL

- The Driver Manager needs some information to connect to the DBMS
  - The database type (to call the proper Driver, that we already loaded in the first step)
  - The server address
  - Authentication information (user/pass)
  - Database / schema to connect to

- All these parameters are encoded into a string
  - The exact format depends on the Driver vendor
MySQL Connection URL format

- jdbc:mysql://[[host:port],[host:port].../
  [database][?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...
- jdbc:mysql://
- host:port (localhost)
- /database
- ?user=username
- &password=pppppppp
2. Establish the connection

- Use `DriverManager.getConnection`
  - Uses the appropriate driver according to the connection URL
  - Returns a `Connection` object

- `Connection connection = DriverManager.getConnection(URLString)`
- Contacts DBMS, validates user and selects the database
- On the `Connection` object subsequent commands will execute queries
Example

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

try {
    Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/test?user=monty&password=secret");

    // Do something with the Connection
    ....

} catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
```
Class diagram

Diagram illustrating the relationships between various classes and objects in a system. The diagram includes entities such as Driver Manager, Driver, Application, Connection, ResultSet, Statement, PreparedStatement, CallableStatement, DatabaseMetaData, and their associated methods and relationships.
3. Create a Statement object

- Statement statement = connection.createStatement();
- Creates a Statement object for sending SQL statements to the database.
- SQL statements without parameters are normally executed using Statement objects.

- For efficiency and security reasons, we will always use a PreparedStatement object (see later...).
4. Execute a query

- Use the `executeQuery` method of the `Statement` class
  - `ResultSet executeQuery(String sql)`
  - `sql` contains a `SELECT` statement
- Returns a `ResultSet` object, that will be used to retrieve the query results
Class diagram
Other execute methods

- `int executeUpdate(String sql)`
  - For INSERT, UPDATE, or DELETE statements
  - For other SQL statements that don’t return a resultset (e.g., `CREATE TABLE`)
  - returns either the row count for INSERT, UPDATE or DELETE statements, or 0 for SQL statements that return nothing

- `boolean execute(String sql)`
  - For general SQL statements
Example

String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
5. Process the result

- The ResultSet object *implements a “cursor” over the query results*
  - Data are available a row at a time
    - Method ResultSet.next() goes to the next row
  - The column values (for the selected row) are available through getXXX methods
  - getInt, getString, ...
  - Data types are converted from SQL types to Java types
Cursor

Row Numbers

cursor is on 0th row

<table>
<thead>
<tr>
<th></th>
<th>Emp_Id</th>
<th>First_Name</th>
<th>Last_Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Sushant</td>
<td>Sharma</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Saurabh</td>
<td>Singh</td>
<td>2500</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Ravi</td>
<td>Kumar</td>
<td>1500</td>
</tr>
</tbody>
</table>

ResultSet
ResultSet.getXXX methods

- XXX is the desired datatype
  - Must be compatible with the column type
  - String is almost always acceptable

- Two versions
  - getXXX(int columnIndex)
    - number of column to retrieve (starting from 1 – beware!)
  - getXXX(String columnName)
    - name of column to retrieve
    - Always preferred
ResultSet navigation methods

- boolean next()
  - Moves the cursor down one row from its current position.
  - A ResultSet cursor is initially positioned **before the first row**:
    - the first call to the method next makes the first row the current row
    - the second call makes the second row the current row, …
Other navigation methods (1/2)

- **Query cursor position**
  - boolean isFirst()
  - boolean isLast()
  - boolean isBeforeFirst()
  - boolean isAfterLast()
Other navigation methods (2/2)

- Move cursor
  - `void beforeFirst()`
  - `void afterLast()`
  - `boolean first()`
  - `boolean last()`
  - `boolean absolute(int row)`
  - `boolean relative(int rows) // positive or negative offset`
  - `boolean previous()`
Example

```java
while( resultSet.next() )
{
    out.println( "<p>" +
                 resultSet.getString(1) + " - " +
                 resultSet.getString(2) + " - " +
                 resultSet.getString(3) + "</p>" ) ;
}
```
# Datatype conversions (MySQL)

<table>
<thead>
<tr>
<th>These MySQL Data Types</th>
<th>Can always be converted to these Java types</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE, TIME, DATETIME, TIMESTAMP</td>
<td>java.lang.String, java.sql.Date, java.sql.Timestamp</td>
</tr>
</tbody>
</table>
# Datatype conversions

<table>
<thead>
<tr>
<th></th>
<th>TINYINT</th>
<th>SMALLINT</th>
<th>INTEGER</th>
<th>BIT</th>
<th>VARBINARY</th>
<th>BINARY</th>
<th>CHAR</th>
<th>LONGVARBINARY</th>
<th>DATE</th>
<th>TIME</th>
<th>TIMESTAMP</th>
<th>ARRAY</th>
<th>REF</th>
<th>STRUCT</th>
<th>JAVA OBJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>getByte</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getShort</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getLong</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getFloat</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getDouble</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getBigDecimal</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getBoolean</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getString</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getBytes</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getDate</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getTime</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getTimestamp</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getAsciiStream</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getUnicodeStream</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 5.1: Use of ResultSet.getXXX Methods to Retrieve JDBC Types
6. Close the connection

- Additional queries may be done on the same connection.
  - Each returns a different ResultSet object, unless you re-use it
  - When no longer needed, ResultSet resources can be freed by ‘closing’ it: resultSet.close()

- When no additional queries are needed, close the connection to the database:
  - connection.close();
Prepared statements
Callable statements

Database access and JDBC
What’s wrong with statements?

- String user = txtUserName.getText(); // JavaFX
- String user = request.getParameter("username"); // JSP
- String sql = "select * from users where username='" + user + "'";

Problems:
- Security
- Performance
Security risk

- SQL injection – syntax errors or privilege escalation
  - Example
    - Username: '; delete * from users ; --'
    - Must detect or escape all dangerous characters!
      - Will never be perfect…
SQL injection attempt 😊
SQL injection attempt 😊

http://xkcd.com/327/
Performance limitations

- **Performance limit**
  - Query must be re-parsed and re-optimized every time
  - Complex queries require significant set-up overhead

- When the same query is repeated (even with different data), parsing and optimization wastes CPU time in the DBMS server
  - Increased response-time latency
  - Decreased scalability of the system
Prepared statements

- **Separate statement creation from statement execution**
  - At creation time: define SQL syntax *(template)*, with placeholders for variable quantities *(parameters)*
  - At execution time: define actual quantities for placeholders *(parameter values)*, and run the statement

- Prepared statements can be re-run many times

- Parameter values are automatically
  - Converted according to their Java type
  - Escaped, if they contain dangerous characters
  - Handle non-character data *(serialization)*
Example

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

String template = "UPDATE music SET price = ? WHERE id = ?";

PreparedStatement statement = connection.prepareStatement(template);

float[] newPrices = getNewPrices();
int[] recordingIDs = getIDs();

for(int i=0; i<recordingIDs.length; i++) {
    statement.setFloat(1, newPrices[i]); // Price
    statement.setInt(2, recordingIDs[i]); // ID
    statement.execute();
}
Prepared statements

- Easier to write
  - Data type conversion done by JDBC library
- Secure (no SQL injection possible)
  - Quoting is done by JDBC library
- More efficient
  - Query re-use
  - Parameter values sent in binary form

- The bottom line: **Always use prepared statements.**
Callable statements

- Many DBMSs allow defining “stored procedures”, directly defined at the DB level
- Stored procedures are SQL queries (with parameters), or sequences of queries
  - Language for defining stored procedures is DBMS-dependent: not portable!
- Calling stored procedures: use CallableStatement in JDBC
Design patterns (DAO)

Database access and JDBC
Problems

- Database code involves a lot of «specific» knowledge
  - Connection parameters
  - SQL commands
  - The structure of the database
- Bad practice to «mix» this low-level information with main application code
  - Reduces portability and maintainability
  - Creates more complex code
  - Breaks the «one-class one-task» assumption
- What is a better code organization?
Goals

- Encapsulate DataBase access into separate classes, distinct from application ones
  - All other classes should be shielded from DB details
- DataBase access should be independent from application needs
  - Potentially reusable in different parts of the application
- Develop a reusable development patterns that can be easily applied to different situations
Data Access Object (DAO) – 1/2

- «Client» classes:
  - Application code that needs to access the database
  - Ignorant of database details (connection, queries, schema, ...)

- «DAO» classes:
  - Encapsulate all database access code (JDBC)
  - The only ones that will ever contact the database
  - Ignorant of the goal of the Client
Data Access Object (DAO) – 2/2

- Low-level database classes: DriverManager, DataSource, ResultSet, etc
  - Used by DAO (only!) but invisible to Client

- «Transfer Object» (TO) or «Data Transfer Object» (DTO) classes
  - Contain data sent from Client to Dao and/or returned by DAO to Client
  - Represent the data model, as seen by the application
  - Usually POJO or JavaBean
  - Ignorant of DAO, ignorant of database, ignorant of Client
DAO class diagram
DAO Sequence diagram
DAO design criteria

- DAO has no state
  - No instance variables (except Connection - maybe)
- DAO manages one ‘kind’ of data
  - Uses a small number of DTO classes and interacts with a small number of DB tables
  - If you need more, create many DAO classes
- DAO offers CRUD methods
  - Create, Read, Update, Delete
- DAO may offer search methods
  - Returning collections of DTO
public interface/class UserDAO

- public User find(Long id)
- public boolean find(Long id, User u)
- public boolean find(User u) // uses u.id
- public User find(String email, String password)
- public List<User> list()
- List<User> searchUserByByName(String name)
  - List<User> searchByName(User u); // only u.name matters
public interface/class UserDAO

- public void create(User user)
- public Long create(User user) // returns new ID
- public void update(User user) // modify all except ID
- public void delete(User user)
- public boolean existEmail(String email)
- public void changePassword(User user)
Object-Relational Mapping

Database access and JDBC
Mapping Tables to Objects

- Goal: guidelines for creating a set of Java Beans (DTO) to represent information stored in a relational database
- Goal: guidelines for designing the set of methods for DAO objects
Tables → Beans

- Create one Java Bean per each main database entity
  - except tables used to store n:n relationships!
- Bean names should match table names (in the singular form)
- The bean should have one private property for each column in the table, with matching names
  - according to Java naming conventions (NUMERO_DATI -> numeroDati)
  - Except columns uses as foreign keys
- The main constructor must accept all the fields in the bean
  - Fields corresponding to foreign keys may not be present in the constructor (lazy object creation)
- Add get()/set() methods for all properties
- Define equals and hashCode, using the exact set of fields that compose the primary key of the table
Relationships, Foreign keys

- Define additional attributes in the Java Bean classes, for every relationship that we want to easily navigate in our application
  - Not necessarily *all* relationships!

- A relationship with cardinality 1 maps to an attribute referring to the corresponding Java object (not the PK value). Use singular nouns.
- A relationship with cardinality n maps to an attribute containing a collection, whose elements are corresponding Java objects (not PK values). Use plural nouns.
- The collection may be Set or List.
- The bean should have methods for reading (get, ...) and modifying (add, ...) the collection
1:1 relationship

<table>
<thead>
<tr>
<th>STUDENTE</th>
<th>PERSONA</th>
</tr>
</thead>
<tbody>
<tr>
<td>matricola (PK)</td>
<td>codice_fiscale (PK)</td>
</tr>
<tr>
<td>fk_persona</td>
<td>fk_studente</td>
</tr>
</tbody>
</table>

class Studente { private Persona persona ; } 
class Persona { private Studente studente ; }
1:N relationship

<table>
<thead>
<tr>
<th>STUDENTE</th>
<th>CITTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>matricola (PK)</td>
<td>cod_citta (PK)</td>
</tr>
<tr>
<td>fk_citta_residenza</td>
<td>nome_citta</td>
</tr>
</tbody>
</table>

class Studente {
    private Citta cittaResidenza ;
}

class Citta {
    private Collection<Studente> studentiResidenti ;
}
N:M relationship

ARTICLE AUTHORSHIP CREATOR

class Article
{
private Collection<Creator> creators ;}
class Creator
{
private Collection<Article> articles ;}
Connection pooling

Database access and JDBC
Connection pooling

- Opening and closing DB connection is expensive
  - Requires setting up TCP/IP connection, checking authorization, …
  - After just 1-2 queries, the connection is dropped and all partial results are lost in the DBMS

- Connection pool
  - A set of “already open” database connections
  - DAO methods “lend” a connection for a short period, running queries
  - The connection is then returned to the pool (not closed!) and is ready for the next DAO needing it
JDBC 3.0 Connection pooling architecture
## Benchmarks

The first time, the connections must be created

Second time, reuse connections

Negligible overhead

<table>
<thead>
<tr>
<th></th>
<th>100 Iterations</th>
<th>100 Iterations</th>
<th>1000 Iterations</th>
<th>3000 Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pooling</td>
<td>547 ms</td>
<td>&lt;10 ms</td>
<td>47 ms</td>
<td>31 ms</td>
</tr>
<tr>
<td>Non-Pooling</td>
<td>4859 ms</td>
<td>4453 ms</td>
<td>43625 ms</td>
<td>134375 ms</td>
</tr>
</tbody>
</table>

10x slower

No improvement

Linear increase
Support in J2EE and Tomcat

- The Java EE Platform Specification requires:
  - Java EE Application Servers must provide a `DataSource` implementation
  - `DataSource` is a connection pool for JDBC connections
  - Tomcat implements this specification

- `DataSource` – interface `javax.sql.DataSource`
  - Alternative to `DriverManager`
  - `DataSource` implementations can be located through JNDI (Java Naming and Directory)
  - Tomcat implements a simplified JNDI service
Configure JNDI

- Tomcat’s JNDI is stored in WEB-INF/web.xml
- Define a resource to access a DataSource object, with a symbolic reference name

```xml
<resource-ref>
  <description>
    Resource reference to a factory for java.sql.Connection instances that may be used for talking to a particular database that is configured in the <Context> configuration for the web application.
  </description>

  <res-ref-name>jdbc/TestDB</res-ref-name>

  <res-type>javax.sql.DataSource</res-type>

  <res-auth>Container</res-auth>
</resource-ref>
```
Configure the connection factory

- Implementation instructions are stored in META-INF/context.xml

```xml
<Context ...>
    ...
    <Resource
        name="jdbc/TestDB"
        auth="Container"
        type="javax.sql.DataSource"
        maxActive="100"
        maxIdle="30"
        maxWait="10000"
        username="utente1" password="utente1"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/nazioni?autoReconnect=true"
    />
    ...
</Context>
```
Get a connection from the pool

- Lookup the DataSource, then get a new connection

```java
/* JNDI query to locate the DataSource object */
Context initContext = new InitialContext();

Context envContext = (Context)initContext.lookup("java:/comp/env") ; // JNDI standard naming root

DataSource ds = (DataSource)envContext.lookup("jdbc/TestDB");

/* Ask DataSource for a connection */
Connection conn = ds.getConnection();

... use this connection to access the database ...

conn.close() ; // return connection to the pool
```
c3p0 library for connection pooling

- Open source library for adding connection pooling capabilities to JDBC drivers
  - [http://www.mchange.com/projects/c3p0/](http://www.mchange.com/projects/c3p0/)
- Connection Pooling
- Prepared Statement Pooling
  - Automatically caches, recognizes and re-uses previously used prepared statements
Using c3p0

import com.mchange.v2.c3p0.*;

... ComboPooledDataSource cpds = new ComboPooledDataSource();

cpds.setDriverClass("org.postgresql.Driver");
  //loads the jdbc driver

cpds.setJdbcUrl("jdbc:postgresql://localhost/testdb");

cpds.setUser("dbuser");
cpds.setPassword("dbpassword");

The DataSource object: cpds.getConnection() lends a connection from the pool
Closing up

- To release a connection to the pool:
  - `connection.close();`
  - ...otherwise the pool will run out of available connections!

- To destroy the connection pool and clean up resources:
  - `cpds.close();`
  - Also disconnects from database.
  - May be placed in a `stop()` method in the main JavaFX class
References

- **JDBC Basics: Tutorial**
  - [http://docs.oracle.com/javase/tutorial/jdbc/TOC.html](http://docs.oracle.com/javase/tutorial/jdbc/TOC.html)

- **JDBC reference guide**
  - [http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/gettingstarted/GettingStartedTOC.fm.html](http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/gettingstarted/GettingStartedTOC.fm.html)

- **JDBC JavaDoc**
  - [http://docs.oracle.com/javase/6/docs/api/java/sql/package-summary.html](http://docs.oracle.com/javase/6/docs/api/java/sql/package-summary.html)
  - [http://docs.oracle.com/javase/6/docs/api/javax/sql/package-summary.html](http://docs.oracle.com/javase/6/docs/api/javax/sql/package-summary.html)
References

- **DAO pattern**
  - http://www.corej2eepatterns.com/Patterns2ndEd/DataAccessObject.htm
References

- **Connection pooling**
  - Tomcat tutorial: http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html#JDBC%20Data%20Sources
  - c3p0 - JDBC3 Connection and Statement Pooling: http://www.mchange.com/projects/c3p0/
Licenza d’uso

- Queste diapositive sono distribuite con licenza Creative Commons “Attribuzione - Non commerciale - Condividi allo stesso modo (CC BY-NC-SA)"

- Sei libero:
  - di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
  - di modificare quest'opera

- Alle seguenti condizioni:
  - **Attribuzione** — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera.
  - **Non commerciale** — Non puoi usare quest'opera per fini commerciali.
  - **Condividi allo stesso modo** — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.

  - [http://creativecommons.org/licenses/by-nc-sa/3.0/](http://creativecommons.org/licenses/by-nc-sa/3.0/)