



CORSO DI LAUREA IN INFORMATICA

PROGRAMMAZIONE WEB

ACCESSO AL DB

a.a 2016-2017

Gestione della persistenza

- Una parte rilevante degli sforzi nello sviluppo di ogni applicazione Web si concentra sul “layer” di persistenza
 - Accesso, manipolazione e gestione di dati persistenti, tipicamente mantenuti in un DB relazionale
- Il mapping Object/Relational si occupa di risolvere il potenziale mismatch fra dati mantenuti in un DB relazionale (table-driven) e il loro processamento fatto da oggetti in esecuzione
 - DB relazionali sono progettati per operazioni di query efficienti su dati di tipo tabellare
 - Necessità di lavorare invece tramite interazione fra oggetti

Accesso diretto alle basi di dati

- È anche possibile (ed è di gran lunga l'accesso a DB più tipico) inserire istruzioni SQL direttamente nel codice di un'applicazione scritta in un linguaggio di programmazione "ospite" (ad es. C, C++, Java, C#)
- Il problema da risolvere è relativo all'integrazione tra i comandi SQL, i quali sono responsabili di realizzare l'accesso al DB, e le normali istruzioni del linguaggio di programmazione
- Una possibile soluzione:
 - Call Level Interface (CLI): l'integrazione con SQL avviene tramite l'invocazione di una opportuna libreria di funzioni che permettono di interagire con un DBMS
 - Es. di soluzione sono ODBC, OLE DB, ADO e JDBC

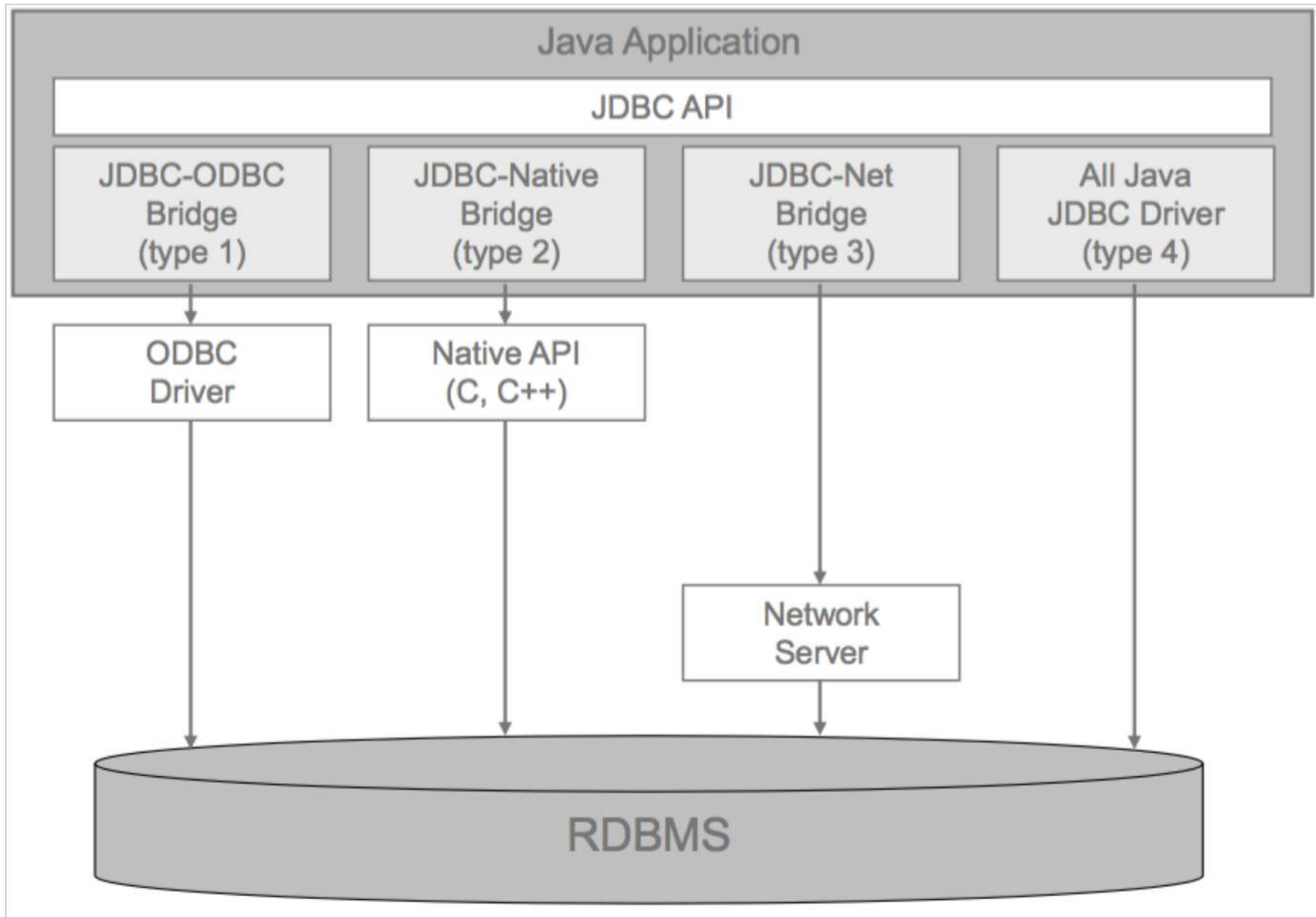
Modo d'uso generale CLI

- Una applicazione che accede (in lettura e/o scrittura) ad una sorgente di dati ha bisogno di fare le seguenti operazioni:
 1. aprire una connessione alla sorgente dati
 2. inviare attraverso la connessione istruzioni (di interrogazione e aggiornamento) alla sorgente dati
 3. processare i risultati ricevuti dalla sorgente dati in risposta alle istruzioni inviate
- Le nostre sorgenti dati sono DB relazionali, gestiti da un DBMS:
 - DB2, MySQL, Hsqldb
- Ogni DBMS espone una API (Application Program Interface)
- Le applicazioni, per noi Java, interagiscono con le API del DBMS attraverso un driver

JDBC

- API Java standard definita da Sun Microsystems nel 1996
- Permette di accedere ai database (locali e remoti) in modo uniforme
- Garantisce accesso ai database in modo indipendente dalla piattaforma
- I driver JDBC sono collezioni di classi Java che implementano metodi definiti dalle specifiche JDBC
- I driver possono essere suddivisi in due categorie:
 - 2-tier: client colloquiano direttamente con DB
 - 3-tier: client comunicano con un middle-tier che accede al DB
- Le classi Java che realizzano funzionalità JDBC sono contenute nei package:
 - `java.sql`: classi fondamentali
 - `javax.sql`: estensioni

Architettura JDBC



JDBC Driver Manager

- Rappresenta il livello di gestione di JDBC e opera tra l'utente e i driver
- Tiene traccia dei driver disponibili e gestisce la creazione di una connessione tra un DB e il driver appropriato
- L'insieme delle classi Java che implementano le interfacce JDBC rappresentano un modulo software chiamato driver JDBC
 - Ogni DBMS ha il proprio driver rilasciato dal produttore o sviluppato da terze parti
 - Sono i driver che realizzano la vera comunicazione con il DB

Schema di uso di JDBC

- Accesso a DB con JDBC consiste nel:
 - Caricare la classe del driver JDBC
 - Ottenere una connessione dal driver
 - Eseguire statement SQL
 - Utilizzare risultati delle query
 - Chiudere la connessione e rilasciare strutture dati utilizzate per la gestione del dialogo con il DB

```
Class.forName("com.ibm.db2.jcc.DB2Driver").newInstance();
Connection conn =
    DriverManager.getConnection("jdbc:db2:MYDB");
Statement stm = conn.createStatement();
ResultSet res = stm.executeQuery("SELECT * FROM MYTABLE");
while (res.next())
{
    String col1 = res.getString("MYCOL1");
    int col2 = res.getInt("MYCOL2");
}
```


Interfacce e classi JDBC

- L'API JDBC 4.0 core mette a disposizione più di 20 tra interfacce e classi
- Alcune fra le più importanti sono
 - **Driver**
 - **DriverManager**
 - **Connection**
 - **Statement**
 - **ResultSet**

Interfaccia Driver

- Rappresenta il punto di partenza per ottenere una connessione a un DBMS
- La classe che implementa Driver può essere considerata la “factory” per altri oggetti JDBC
 - ad esempio, oggetti di tipo Connection
- È possibile ottenere un’istanza effettiva della classe Driver ricercando la classe con il metodo `forName`:

`Driver d = Class.forName(“com.mysql.jdbc.Driver“).newInstance();`

- Avendo a disposizione un oggetto Driver è possibile ottenere la connessione al database
- Ogni driver JDBC ha una stringa di connessione che riconosce nella forma:
 - `jdbc:mysql://ip:port/db`
 - `jdbc:mysql://localhost:3306/mioDB`

Classe DriverManager

- Facilita la gestione di oggetti di tipo Driver
- Quando un oggetto Driver viene istanziato, esso viene automaticamente registrato nella classe **DriverManager**
- Ogni applicazione può registrare uno o più driver JDBC diversi tra loro
- Consente la connessione con il DBMS sottostante
 - Mediante il metodo statico **getConnection**
 - Usa il driver opportuno tra quelli registrati

Interfaccia Connection

- Un oggetto di tipo Connection rappresenta una connessione attiva con il DB
- Il metodo **getConnection** di **DriverManager**, se non fallisce, restituisce un oggetto di tipo Connection
- L'interfaccia mette a disposizione una serie di metodi per le operazioni, tra le quali:
 - Preparazione di query SQL da inviare tramite oggetti
 - Statement
 - PreparedStatement

Interfaccia Statement

- Gli oggetti di tipo Statement possono essere usati per inviare query SQL semplici verso il DBMS sottostante
 - query che non fanno uso di parametri
- La query può essere di modifica
 - **UPDATE, INSERT, CREATE**
- oppure di selezione
 - **SELECT**
- Per query di tipo SELECT il risultato è inserito in un oggetto **ResultSet**
- Un oggetto Statement può essere creato con il metodo **createStatement** di Connection

Interfaccia PreparedStatement

- Gli oggetti di tipo PreparedStatement possono essere usati per creare query SQL parametriche e precompilate (“prepared”)
- Il valore di ciascun parametro non è specificato nel momento in cui lo statement SQL è definito, ma rimpiazzato dal carattere ‘?’
- È più efficiente perchè una volta che la query è preparata, sarà pre-compilata per usi futuri
- Un oggetto PreparedStatement può essere creato con il metodo **prepareStatement** di Connection

Interfaccia ResultSet

- L'oggetto ResultSet è il risultato di una query di selezione (SELECT)
- Rappresenta una tabella composta da righe (gli elementi selezionati) e colonne (gli attributi richiesti)
 - Si può accedere in modo casuale alle righe, inserirle, cancellarle e modificarle

Programmare un'applicazione JDBC

1. Importazione package
2. Registrazione driver JDBC
3. Apertura connessione al DB (**Connection**)
4. Creazione oggetto **Statement**
5. Esecuzione query e eventuale restituzione oggetto **ResultSet**
6. Utilizzo risultati
7. Chiusura oggetto/i **ResultSet** e oggetto/i **Statement**
8. Chiusura connessione

Oggetto Statement

- Un oggetto Statement fornisce tre metodi per eseguire una query SQL:

`executeQuery(stmt SQL)`

- per statement che generano un unico result set (SELECT)

`executeUpdate(stmt SQL)`

- per statement di modifica (UPDATE, INSERT, ecc.)

`execute(stmt SQL)`

- per statement che generano più di un risultato o più di un contatore di aggiornamento

executeQuery

- Usato tipicamente per query di tipo SELECT
- Restituisce un oggetto **ResultSet**

Name	Age	Gender
John	27	Male
Jane	21	Female
Jeanie	31	Female

executeUpdate

- Usato per statement di tipo DML quali INSERT, UPDATE o DELETE
- e per statement di tipo DDL quali CREATE TABLE e DROP TABLE
- Restituisce un intero rappresentante il numero di righe che sono state inserite/aggiornate/cancellate
 - contatore di aggiornamento
- In caso di statement di tipo DDL, restituisce sempre il valore 0

execute

- Usato quando la query restituisce più di un risultato o più di un contatore di aggiornamento
- Utilizza i seguenti metodi:
 - `getResultSet()`
 - per ottenere il resultset successivo
 - `getUpdateCount()`
 - per ottenere il contatore di aggiornamento successivo
 - `getMoreResults()`
 - per sapere se ci sono altri resultset o contatori di aggiornamento
- Restituisce **true** se il primo risultato è di tipo ResultSet; **false** se il risultato è di tipo Count o non ci sono più risultati

Oggetto ResultSet

- Un oggetto ResultSet contiene il risultato di una query SQL (cioè una tabella)
- Un oggetto ResultSet mantiene un cursore alla riga corrente
- Per ottenere un valore relativo alla riga corrente:
 - `getXXX(column-name)`
 - `getXXX(column-number)`
- Per spostare il cursore dalla riga corrente a quella successiva:
 - `next()` (restituisce `true` in caso di successo; `false` se non ci sono più righe nell'insieme risultato)

I metodi getxxx

- Permettono la lettura degli attributi di una tabella
- `getBytes`
- `getShort`
- `getInt`
- `getLong`
- `getFloat`
- `getDouble`
- `getBigDecimal`
- `getBoolean`
- `getString`
- `getBytes`
- `getDate`
- `getTime`
- `getTimestamp`
- `getAsciiStream`
- `getUnicodeStream`
- `getBinaryStream`
- `getObject`

Controllo sui valori NULL

- I valori NULL SQL sono convertiti in null, 0, o false, dipendentemente dal tipo di metodo getXXX
- Per determinare se un particolare valore di un risultato corrisponde a NULL in JDBC:
 - Si legge la colonna
 - Si usa il metodo **wasNull()**

Oggetto PreparedStatement

- Usato quando la query SQL prende uno o più parametri come input, o quando una query semplice deve essere eseguita più volte
- L'interfaccia PreparedStatement estende l'interfaccia Statement ereditandone tutte le funzionalità. In più sono presenti metodi per la gestione dei parametri
- L'oggetto viene creato con l'istruzione

Connection.prepareStatement(stmt SQL)

- I parametri vengono poi settati mediante il metodo **setXXX(n,value)**
- La query pre-compilata viene eseguita mediante i metodi **executeQuery()**, **executeUpdate()** o **execute()** senza bisogno di passare alcun parametro!!

I metodi setxxx

- Permettono l'assegnamento dei parametri di uno statement SQL
- setByte
- setShort
- setInt
- setLong
- setFloat
- setDouble
- setBigDecimal
- setBoolean
- setNull
- setString
- setBytes
- setDate
- setTime
- setTimestamp
- setAsciiStream
- setUnicodeStream
- setBinaryStream
- setObject

Esempio (query di lettura)

```
String sql = "SELECT * FROM aerei";  
ResultSet rs = st.executeQuery(sql); // rs contiene le righe della tabella  
while (rs.next()) {  
    // modello è il nome della terza colonna della tabella  
    String modello = rs.getString("modello");  
    // posti è il nome della quinta colonna della tabella  
    int posti= rs.getInt("numposti");  
    String produttore= rs.getString(2);  
    // elaborazione dei campi  
    System.out.printf("%s %s %d\n", modello, produttore, posti);  
}
```

Esempio (query parametriche)

// il ? rappresenta il parametro

```
String sql = "SELECT * FROM aerei WHERE produttore = ?";
```

```
String produttore = "boeing";
```

```
PreparedStatement ps = con.prepareStatement(sql);
```

```
ps.setString(1, produttore); // associamo al (primo e unico) parametro la stringa produttore
```

```
ResultSet rs = ps.executeQuery(); // eseguiamo la query
```

// nessun argomento per il metodo executeQuery, attenzione!

// si possono scorrere i record

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

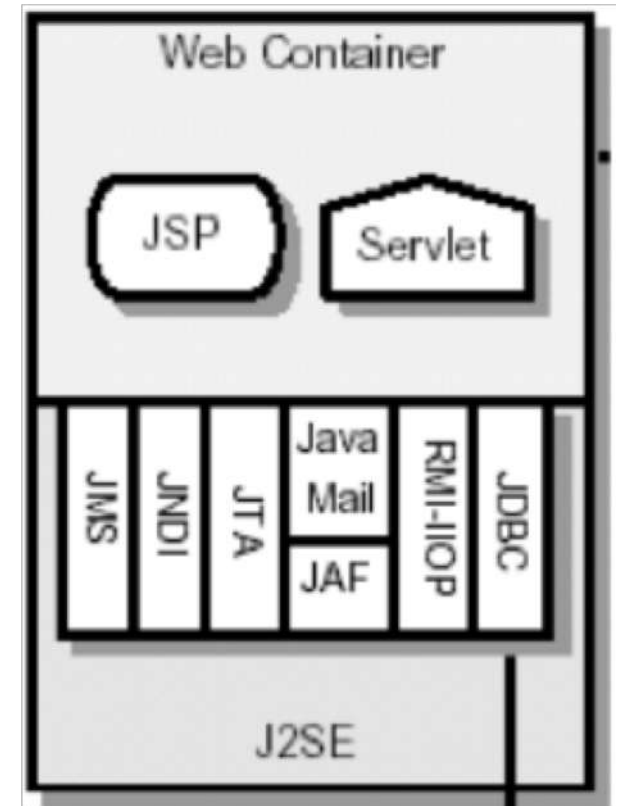
// lettura ed elaborazione dei record

// ...

```
}
```

Servizi del Container

- Il container (J2EE) mette a disposizione delle Servlet una serie di servizi, proprio per evitare limitazioni di riuso e monoliticità:
 - JMS per gestire code di messaggi
 - JNDI per accedere a servizi di naming
 - JDBC per accedere ai database
 - JTA per gestire transazioni
 - Java Mail per inviare e ricevere messaggi di posta elettronica
 - RMI per l'accesso ad oggetti remoti
 - ...
- Esaminiamo brevemente due di questi servizi:
 - **JNDI** e **JDBC**

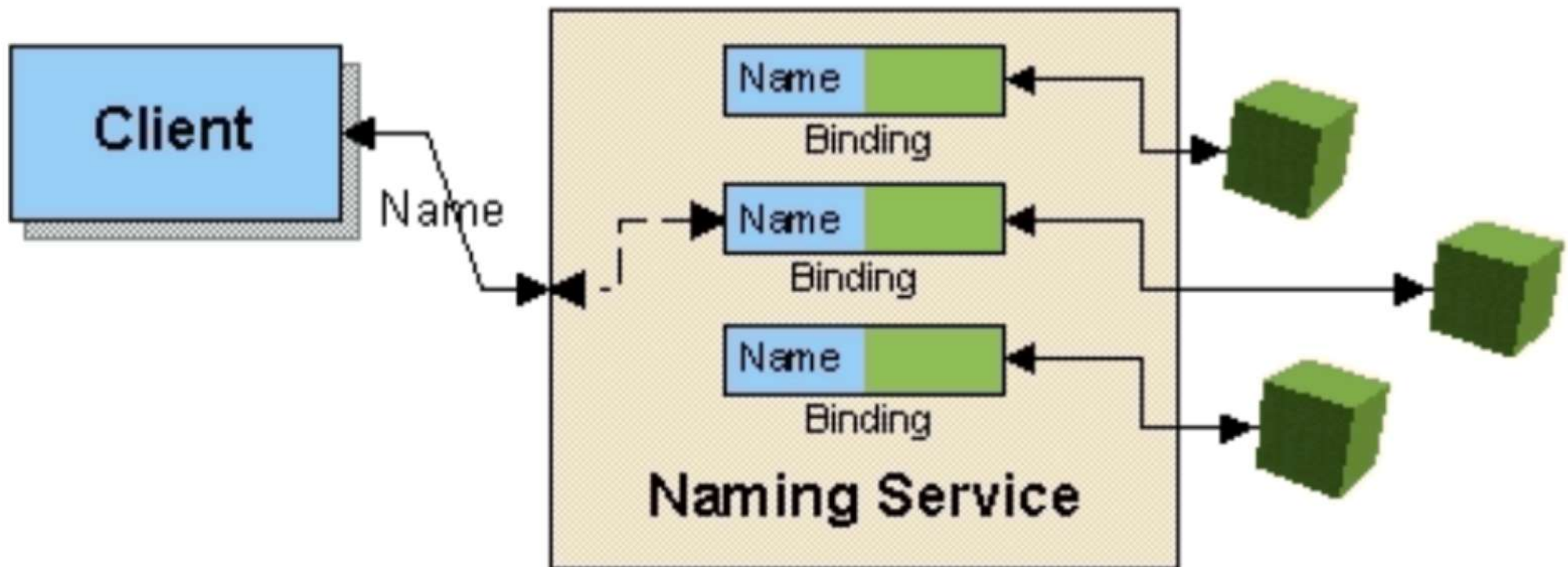


Servizi del container: JNDI

- JNDI è una API Java standard per l'accesso uniforme a **servizi di naming**
- Permette di accedere a qualsiasi servizio di naming:
 - DNS server
 - File System
 - RDBMS
- Ha una struttura ad albero (JNDI-tree) ed è basata su coppie **chiave-valore**
- Permette di accedere ad oggetti identificati da nomi logici e consente di rendere facilmente configurabile un'applicazione
- Le classi JNDI sono contenute in **javax.naming**
- Il Container mette a disposizione delle servlet un servizio JNDI

Naming

- Naming service mantiene un insieme di binding fra nomi e oggetti (o riferimenti a oggetti)
- Java Naming & Directory Interface (JNDI) come interfaccia che supporta funzionalità comuni ai vari differenti sistemi di nomi



JNDI Provider

- In JNDI i servizi di naming vengono acceduti attraverso plugin chiamati **(name service) provider**
- Provider JNDI
 - NON è servizio di naming ma un'interfaccia di connessione verso uno specifico servizio di naming esterno (in terminologia JNDI, è cliente JNDI verso vero servizio di nomi esterno)
- Provider JNDI si occupa di supporto alla persistenza (binding salvati e nel caso ripristinati dopo un fault) e distribuzione su rete
- Essenzialmente, provider JNDI è scelto configurando le proprietà
 - **java.naming.factory.initial**
 - **java.naming.provider.url**
- Es:
 - proprietà initial `com.sun.jndi.rmi.registry.RegistryContextFactory`
 - proprietà url `rmi://www.sesa.unisa.it:8081`

JNDI: Context e InitialContext

- **Context** è interfaccia che specifica Naming System, con metodi per **aggiungere, cancellare, cercare, ridenominare, ...** oggetti
 - La classe Context svolge un ruolo centrale in JNDI
 - Context rappresenta insieme di binding all'interno di un servizio di nomi e che condividono stessa convenzione di naming
 - Oggetto Context è usato per fare binding/unbinding di nomi a oggetti, per fare renaming e per elencare binding correnti
- **InitialContext** è un'implementazione di Context e rappresenta il contesto di partenza per operazioni di naming
- Tutte le operazioni di naming in JNDI sono svolte in relazione a un Context
 - Si parte da una classe InitialContext, istanziata con proprietà che definiscono tipo di servizio di nomi in uso (eventualmente ID e password)

Interfaccia Context

- **void bind(String stringName, Object object)**
 - Il nome non deve essere associato già ad alcun oggetto
- **void rebind(String stringName, Object object)**
- **Object lookup(String stringName)**
- **void unbind(String stringName)**
- **void rename(String stringOldName, String stringNewName)**
- **NamingEnumeration listBindings(String stringName)**
 - Restituisce enumeration con nomi del context specificato, insieme a oggetti associati e loro classi

(Metodo manuale) Schema di uso di JDBC con il DriverManager

- Accesso a DB con JDBC consiste nel:
 - Caricare la classe del driver JDBC
 - Ottenere una connessione dal driver
 - Eseguire statement SQL
 - Utilizzare risultati delle query
 - ProductnameDM.java nell'esempio

```
Class.forName("org.hsqldb.jdbcDriver");  
Connection conn = DriverManager.getConnection(  
    "jdbc:hsqldb:hsqldb://localhost:1701");  
Statement stm = conn.createStatement();  
ResultSet res = stm.executeQuery("SELECT * FROM MYTABLE");  
while (res.next())  
{  
    String col1 = res.getString("MYCOL1");  
    int col2 = res.getInt("MYCOL2");  
}
```

Connection pool

- **Connection Pool** sono oggetti, amministrati dall'application server, preposti a gestire connessioni verso DB
- Sono configurabili attraverso opportuni file
- Il vantaggio principale nell'utilizzo di Connection Pool risiede nel fatto che le connessioni sono esistenti quando l'applicazione necessita di connettersi a DB, eliminando quindi overhead dovuto alla creazione delle connessioni ad ogni richiesta (**Reduced connection creation time**)
- L'Application server può applicare un bilanciamento di carico alle applicazioni che usano DB, assegnando o rilasciando connessioni alle applicazioni in dipendenza dalle loro necessità (**Connection pooling**)
- Il bilanciamento può anche includere un incremento o riduzione del numero di connessioni nel pool al fine di adattarlo al cambiamento delle condizioni di carico (**Controlled resource usage**)

DriverManagerConnectionPool

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.LinkedList;
import java.util.List;
```

```
public class DriverMaagerConnectionPool {
    private static List<Connection> freeDbConnections;

    static {
        freeDbConnections = new LinkedList<Connection>();

        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("DB driver not found!", e);
        } catch (IOException e) {
            System.out.println("DB connection pool error!", e);
        }
    }
    ...
}
```



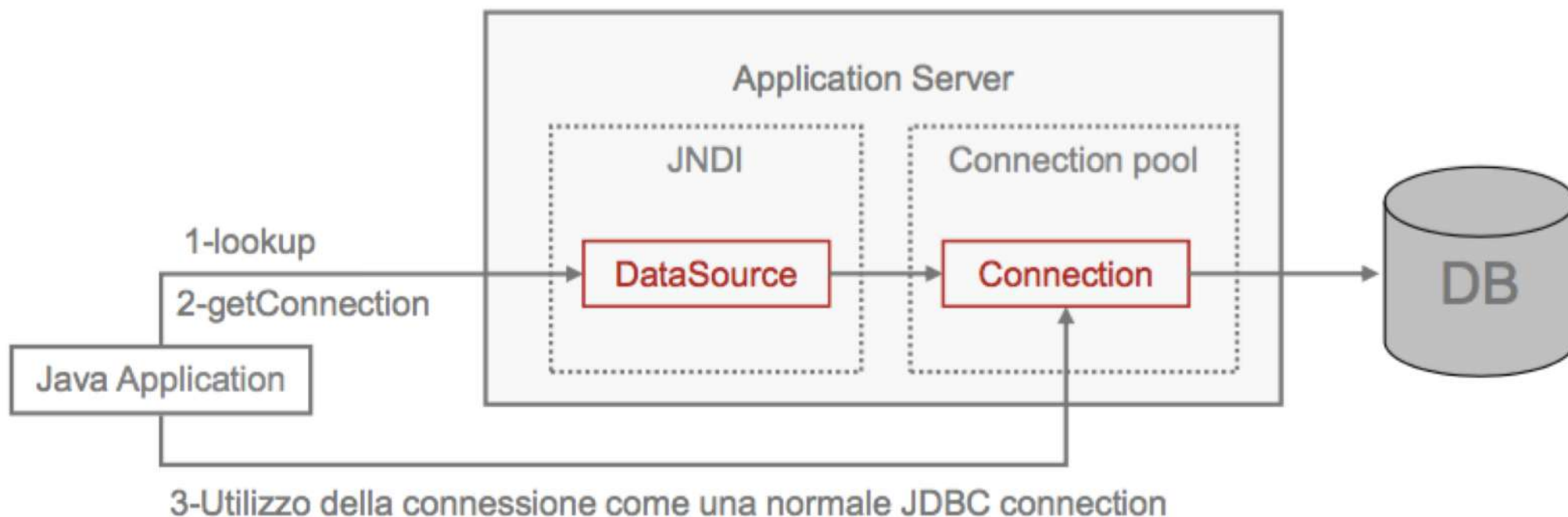
```
...  
private static Connection createDBConnection() throws SQLException {  
    Connection newConnection = null;  
    String db = "mioDb";  
    String username = "login";  
    String password = "password";  
  
    newConnection = DriverManager.getConnection(  
        "jdbc:mysql://localhost:3306/"+db, username, password);  
  
    newConnection.setAutoCommit(false);  
  
    return newConnection;  
}  
...
```

...

```
public static synchronized Connection getConnection() throws SQLException {  
    Connection connection;  
  
    if (! freeDbConnections.isEmpty()) {  
        connection = (Connection) freeDbConnections.get(0);  
        DriverManagerConnectionPool.freeDbConnections.remove(0);  
        try {  
            if (connection.isClosed())  
                connection =  
DriverMaagerConnectionPool.getConnection();  
        } catch (SQLException e) {  
            connection = DriverManagerConnectionPool.getConnection();  
        }  
    } else connection = DriverManagerConnectionPool.createDBConnection();  
    return connection;  
}  
  
public static synchronized void releaseConnection(Connection connection) {  
    DriverManagerConnectionPool.freeDbConnections.add(connection);  
}
```

(Metodo alternativo) Schema di uso di JDBC con il DataSource

- **DataSource** sono factory di connessioni verso sorgenti dati fisiche rappresentate da oggetti di tipo **javax.sql.DataSource**
- Oggetti di tipo DataSource vengono pubblicati su JNDI e vengono creati sulla base di una configurazione contenuta in un descrittore (web.xml)
- DataSource è un wrapper di connection pool



Accesso a sorgente e connessione

- Per accedere a DB via data source è necessario fare lookup da JNDI e ottenere Connection dall'istanza di tipo DataSource
- Il container fa in modo, automaticamente, che il contesto iniziale punti al servizio JNDI gestito dal container stesso

```
// Contesto iniziale JNDI
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");

// Look up del data source
DataSource ds =
    (DataSource) envCtx.lookup("jdbc/EmployeeDB");

//Si ottiene una connessione da utilizzare come una normale
//connessione JDBC
Connection conn = ds.getConnection();
... uso della connessione come visto nell'esempio JDBC ...
```


Definizione della risorsa

- Definizione della risorsa in /WEB-INF/web.xml

```
<resource-ref>  
  <description>  
    Riferimento JNDI ad un data source  
  </description>  
  <res-ref-name>jdbc/EmployeeDB</res-ref-name>  
  <res-type>javax.sql.DataSource</res-type>  
  <res-auth>Container</res-auth>  
</resource-ref>
```

Definizione del contesto

- Definizione della risorsa in /META-INF/context.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<Context>
  <Resource name="jdbc/EmployeeDB"
    auth="Container"
    driverClassName="com.mysql.jdbc.Driver"
    type="javax.sql.DataSource"
    username="username"
    password="password"
    url="jdbc:mysql://localhost:3306/company" />
</Context>
```

Metodologie per la gestione della persistenza

- In generale possiamo pensare che le operazioni (metodi) degli oggetti facade
 - caricano tuple dalla base di dati e le usano per creare gli oggetti del modello
 - effettuano le operazioni della logica applicativa (es. evadi ordine)
 - salvano in maniera persistente gli oggetti del modello nella base di dati
- **L'interazione con il DB è una operazione critica**
- Esistono diverse metodologie (di complessità crescente) per realizzare questa interazione:
 - **forza bruta**
 - pattern DAO (Data Access Object) e DTO (Data Transfer Object)
 - framework ORM (Object-Relational Mapping)
 - Es. Hibernate

simile al Java Bean

Forza bruta

- È la tecnica più semplice per gestire la persistenza forte accoppiamento con la sorgente dati
- Consiste nello scrivere dentro le classi del modello un insieme di metodi che implementano le operazioni CRUD
- Operazioni CRUD
 - **C**reate: inserimento di una tupla (che rappresenta un oggetto) nel database (INSERT)
 - **R**etrieve: ricerca di una tupla secondo un qualche criterio di ricerca (SELECT)
 - **U**ppdate: aggiornamento di una tupla nel database (UPDATE)
 - **D**eleate: eliminazione di una tupla nel database (DELETE)
- Ci possono essere diverse operazioni di Retrieve
 - diversi criteri: “per chiave” vs. “per attributo” (es. ricerca cliente per codice, per nome, etc.)
 - diversi risultati (un oggetto, una collezione di oggetti) a seconda del criterio
 - ordinamento

Forza bruta in sintesi

- Per ogni **classe MyData** che rappresenta una entità del dominio, si definiscono:
 - un metodo **doSave(MyData o)** che salva i dati dell'oggetto corrente nel database
 - il metodo esegue una istruzione SQL insert
 - un metodo **doSaveOrUpdate(MyData o)** che salva i dati dell'oggetto corrente nel database
 - il metodo esegue una istruzione SQL update o insert a seconda che l'oggetto corrente esista già o meno nel database
- un metodo **doDelete(X key)** che cancella dal database i dati dell'oggetto corrente
 - Si può usare anche **doDelete(MyData a)**

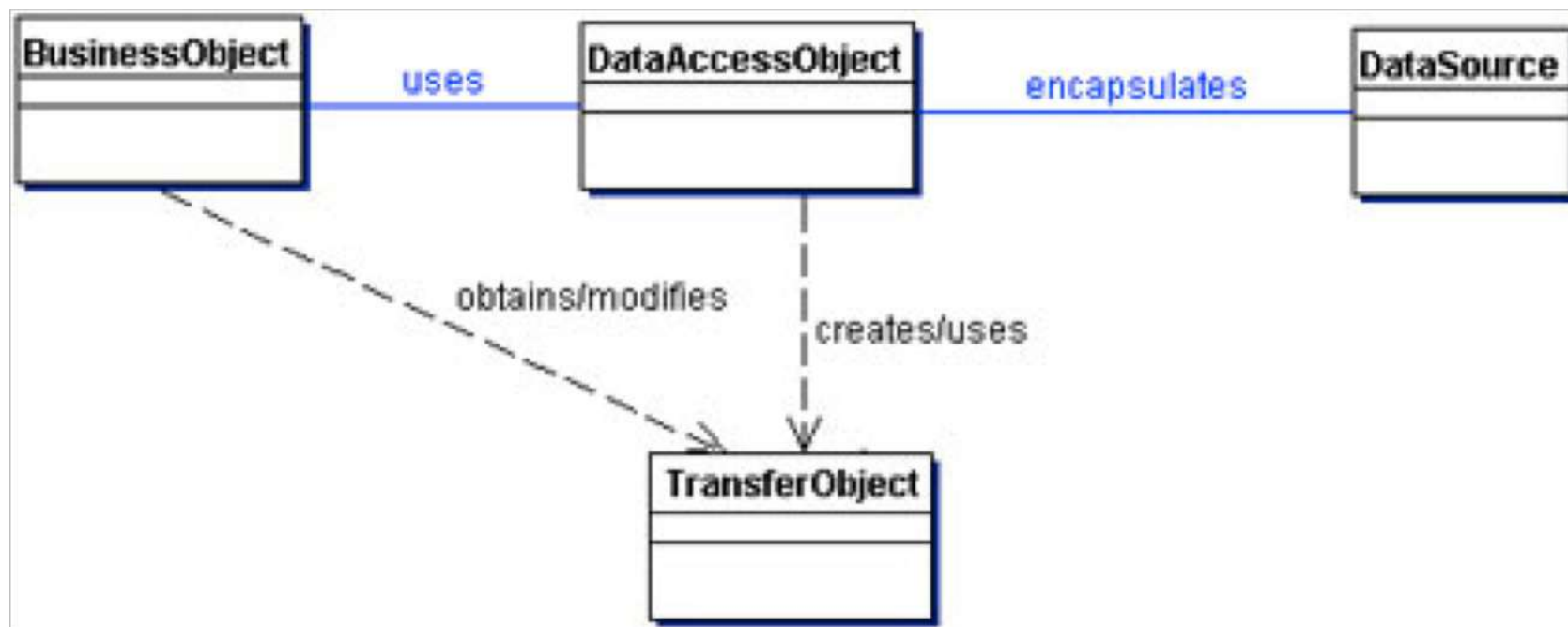
Forza bruta in sintesi (2)

- un metodo **doRetrieveByKey(X key)** che
 - restituisce un oggetto istanza di **MyData** i cui dati sono letti dal database
 - tipicamente da una tabella che è stata derivata dalla stessa classe del modello di dominio che ha dato origine a **MyData**
 - recupera i dati per chiave
- uno o più metodi **doRetrieveByCond(...)** che restituiscono una collezione di oggetti istanza della classe **MyData** che soddisfano una qualche condizione (basata sui parametri del metodo)
- uno metodo **doRetrieveAll(...)** che restituisce tutta la collezione di oggetti istanza della classe **MyData**

DAO (Data Access Object)

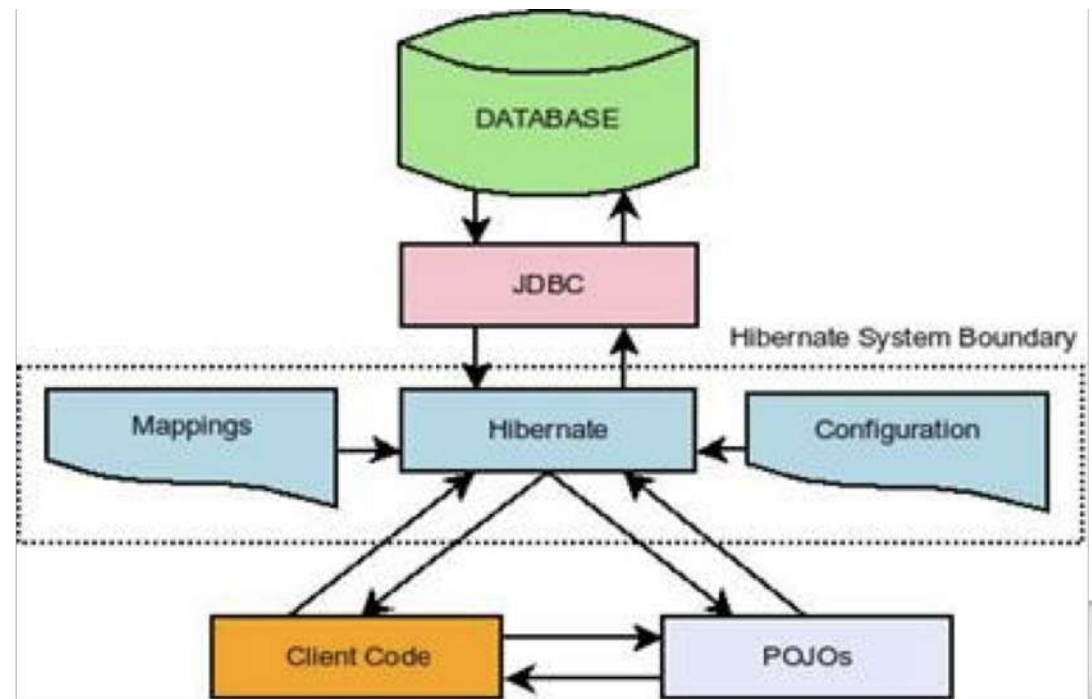
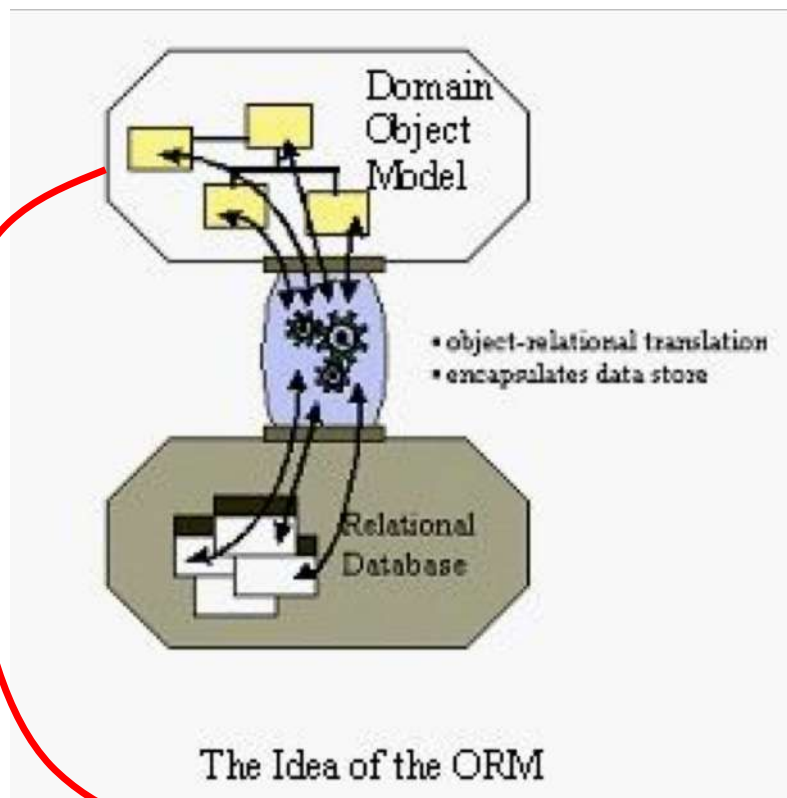
- Il pattern DAO rappresenta un possibile modo di separare:
 - logica di business (es: Servlet, JSP, ...)
 - logica di persistenza (es: r/w su DB, ...)
- Solo gli oggetti previsti dal pattern DAO
 - hanno il permesso di “vedere” il DB
 - espongono metodi di accesso per tutti gli altri componenti
- I valori scambiati tra DB e il resto dell'applicazione sono racchiusi in oggetti detti Data Transfer Object (DTO):
 - campi privati per contenere i dati da leggere/scrivere sul DB
 - metodi *getter* e *setter* per accedere dall'esterno a tali campi
 - metodi di utilità (confronto, stampa, ...)

Struttura DAO e DTO



ORM (Object-Relational Mapping)

- ORM è “la persistenza automatica (e trasparente) di oggetti di applicazioni Java in tabelle di un DB relazionale, basata su metadati che descrivono il mapping tra oggetti e DB”



Il problema dell'SQL injection

- Cosa succede se, data una query con valori di input dell'utente (es. tramite interfaccia Web), questi ha la possibilità di agire direttamente sul valore dell'input di tipo stringa (oggetto **String**), aggiungendo, ad esempio, apici e altre istruzioni di controllo?
- ***Può inserire istruzioni arbitrarie che verranno eseguite dal DBMS!!!***
- Esempio:

Statement =

```
"SELECT * FROM users WHERE name = '" + userName + "';"
```

con la variabile **userName** assegnata al valore: **a' ;DROP TABLES users;**

- Si può anche usare:
 - **a' ; DROP TABLES users; --**
 - dove **--** indica l'inizio di un commento SQL
 - **' or 1=1 -** quando viene richiesto un campo di accesso per gli utenti (tipo password)

SQL injection

- È una tecnica che sfrutta la vulnerabilità a livello di sicurezza dello strato DB di una applicazione
- Tale vulnerabilità è presente quando i dati di input dell'utente sono:
 - Filtrati in modo incorretto per la verifica di caratteri literal escape nelle stringhe
 - Non sono fortemente “tipati” o non sono controllati i vincoli di tipo
- SQL injection in quanto l'utente può “iniettare” statement SQL arbitrari con risultati catastrofici:
 - divulgazione di dati sensibili o esecuzione di codice
- *Per proteggere le nostre applicazioni dall'SQL injection, i dati di input dell'utente **NON** devono essere direttamente incastonati all'interno di Statement SQL*

Prevenire SQL injection

- A prevenzione del problema, l'interfaccia **PreparedStatement** permette di gestire in modo corretto anche l'inserimenti di dati "ostili"
- Si tratta di statement "*parametrizzati*" che permettono di lavorare con parametri (o variabili bind) invece che di incastonare i dati di input dell'utente direttamente nello statement
 - SQL statement è fisso
 - i dati di input dell'utente sono assegnati ai parametri (*bounding*)

Per far funzionare il progetto “Storage”...

1. Importare il progetto Storage in Eclipse
2. Create a new connection in MySql Workbench (es. storageconnection)
 - a. Eseguire **storage.sql** nella query view (crea il database)
3. Configurare username e password in:
 - a. webContent/META-INF/**Context.xml** (configurazione del DataSource)
 - b. **DriveManagerConnectionPool.java** (configurazione del DriverManager)
4. Eseguire productControl.java
 - a. **ProductModelDS** usa il DataSource (consigliato)
 - b. **ProductModelDM** usa il DriverManager

storage.sql

```
-- Rimuove il DB
DROP DATABASE IF EXISTS storage;

-- Crea il DB storage e lo usa
CREATE DATABASE storage;
USE storage;

-- Rimuove la tabella product
DROP TABLE IF EXISTS product;

-- Crea la tabella product
CREATE TABLE product (
    code int primary key AUTO_INCREMENT,
    name char(20) not null,
    description char(100),
    price int default 0,
    quantity int default 0
);

-- Popolamento
INSERT INTO product values (1,"Samsung F8000","TV 48 pollici",550,5);
-- ...
```

Riferimenti

- JNDI:
 - <http://www.oracle.com/technetwork/java/jndi/index.html>
- DataSource:
 - <https://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html>
- DAO:
 - <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>
- ORM:
 - <http://www.agiledata.org/essays/mappingObjects.html>
- Hibernate:
 - <http://www.hibernate.org/>
 - <http://www.visualbuilder.com/java/hibernate/tutorial/>