Università degli Studi di Bologna

Facoltà di Ingegneria

Tecnologie Web T
A.A. 2019 – 2020

Esercitazione 07
Accesso Diretto alle Basi di Dati:
JDBC con approccio “forza bruta”

Agenda

- **JDBC**
  - **creazione del proprio SCHEMA sul DB TW_STUD** seguendo le istruzioni presenti nelle **note sull'utilizzo di DB2 in LAB4** (rif. pagina 5), sezione **Laboratorio** sito Web del corso
  - esemplificazione uso API JDBC mediante un **esercizio guidato**
  - proposta **esercizio da svolgere in autonomia**

    "gestione persistenza di un ordine rappresentato dal carrello"
    - prima senza utilizzo di componenti Web
    - poi integrando anche la componente Web
  - test pratico su **SQL injection**
  - **esercizio guidato tratto dall'appello d'esame** del 17 gennaio 2018: gestione persistenza basata su metodologia «forza bruta»

Metodologia forza bruta

- Ci ricordiamo i passi implementativi della metodologia «forza bruta» vero? ☺
  - Per ogni *classe* `MyC` che rappresenta una entità del dominio, si definiscono:
    - un metodo **`doRetrieveByKey(X key)`** che
      - *restituisce un oggetto istanza di* **`MyC`** *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* **`MyC`**)
      - *recupera i dati per chiave*
    - un metodo **`saveOrUpdate(…)`** 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*
    - uno o più metodi **`doRetrieveByCond(…)`** che *restituiscono una collezione di oggetti istanza della classe* ***`MyC`*** *che soddisfano una qualche condizione* (basata sui *parametri* del metodo)
    - un metodo **`doDelete(…)`** che *cancella dal database i dati dell'oggetto corrente*

Esercizio guidato

Obiettivo: esemplificare l'API JDBC

- scriviamo codice che ha unicamente lo scopo di evidenziare l'uso della API JDBC
- non ci preoccupiamo della qualità del codice Java

Operazioni

- inserimento di una tupla nel DB
- cancellazione di una tupla
- ricerca di un tupla per chiave primaria
- ricerca di un insieme di tuple (per qualche proprietà)

Lato Java

Consideriamo la classe **`Student`**:

```
package model;
import java.util.Date;

public class Student {
  private int code;
  private String firstName;
  private String lastName;
  private Date birthDate;

   public Student(){}
   public String getFirstName() {
         return firstName;
   }
   public void setFirstName(String firstname) {
         this.firstName = firstName;
   }
   // seguono tutti gli altri metodi getter e setter
}
```

Lato database

…e il DB *tw_stud*:

```
CREATE TABLE students
(
  code INT NOT NULL PRIMARY KEY,
  firstname CHAR(40),
  lastname CHAR (40),
  birthdate DATE
)
```

Ambiente

DBMS

- a scelta tra DB2 (consigliato), HSQLDB e MySQL

Driver JDBC per il DBMS scelto

- per **DB2**: driver **`com.ibm.db2.jcc.DB2Driver`**, contenuto nel file **`db2jcc.jar`** (direttorio *C:\Programmi\IBM\SQLLIB\java* della macchina client del LAB4)
- per **HSQLDB**: driver **`org.hsqldb.jdbcDriver`**, contenuto nel file **`hsqldb.jar`** (scaricabile da http://hsqldb.org/ o dal sito del corso)
- per **MySQL**: driver **`com.mysql.jdbc.Driver`**, contenuto nel file **`mysql-connector-java-5.1.x-bin.jar`** (scaricabile da http://dev.mysql.com/downloads/ o dal sito del corso)

Ambiente Java standard: **`.jar/.zip`** del driver deve essere nel **`CLASSPATH`**, ad esempio:

Eclipse Project → Properties → Java Build Path → Libraries → Add JARs

HSQLDB e MySQL

L'ambiente DB2 è già configurato in LAB4; per i DBMS aggiuntivi, occorre operare un breve setup

HSQLDB

- il file *hsqldb.jar* presente nella directory *lib* del progetto contiene già tutto il necessario
- per avviare il server, si consiglia di utilizzare l'apposito target ANT *97.database.start*, oppure *java -cp ../lib/hsqldb.jar org.hsqldb.Server -database.0 file:tw_stud.txt -dbname.0 tw_stud*
- per arrestare il server, avviare il target ANT *98.database.frontend*, inserire in *URL* la stringa *jdbc:hsqldb:hsql://localhost/tw_stud*, eseguire *SHUTDOWN*
- se si vuole si possono scaricare server e connector tramite *http://hsqldb.org → download → hsqldb → hsqldb_2_2 → hsqldb-2.2.0.zip* (il connector si trova nella directory *lib*)

MySQL

- in LAB4 copiare la directory di MySQL da \\pdc4\public\TecnologieWeb a C:\temp
- il connector si trova nella directory lib del progetto Eclipse
- per avviare il server *bin/mysqld.exe, root* user di default senza password
- **per creare il database**, *bin/mysql.exe -u root → CREATE DATABASE tw_stud;*
- per arrestare il server *bin/mysqladmin.exe -u root shutdown*
- visualizzare database esistenti, *bin/mysql.exe -u root → SHOW DATABASES;*
- da casa scaricare il server tramite *www.mysql.com/donwloads → MySQL Community Server → mysql-5.5.x-win32.zip* e il scaricare il connector tramite *www.mysql.com/downloads → Connector/J → ZIP Archive* e all'interno del file zip si trova il file jar *mysql-connector-java-5.1.x-bin.jar*

Le classi fondamentali di JDBC

Package **`java.sql`** (va importato)

Classe **`DriverManager`**

Interfaccia **`Driver`**

Interfaccia **`Connection`**

Interfaccia **`PreparedStatement`**

Interfaccia **`ResultSet`**

Eccezione **`SQLException`**

Primo passo

- Importare il progetto Eclipse ANT-based presente nel file **07_TecWeb.zip** come visto nelle precedenti esercitazioni, senza esploderne l'archivio su file system (lo farà Eclipse)

File → Import → General → Existing Projects into Workspace → Next → Select archive file

- Confiniamo nella classe **`DataSource`** le operazioni necessarie per ottenere la connessione
  - il suo compito è fornire connessioni alle altre classi che ne hanno bisogno
  - metodo **`Connection getConnection()`** che restituisce una nuova connessione ad ogni richiesta

È una soluzione artigianale usata solo a fini didattici; verrà raffinata successivamente! ☺

La classe DataSource

```
import java.sql.*;
public class DataSource {
   private String dbURI = "jdbc:db2://diva.disi.unibo.it:50000/tw_stud";
   private String userName = "*******";
   private String password = "*******";

   public Connection getConnection() throws PersistenceException {
    Connection connection;
    try {
        Class.forName("com.ibm.db2.jcc.DB2Driver");
        connection = DriverManager.getConnection(dbURI,userName,
                                                 password);
    } catch (ClassNotFoundException e) {
        throw new PersistenceException(e.getMessage());
    } catch(SQLException e) {
        throw new PersistenceException(e.getMessage());
    }
    return connection;
   }
}
```

! Nel progetto Eclipse classe *DataSource* gestisce DB2, HSQLDB e MySQL

Definiamo istruzioni SQL

Vediamo ora il codice JDBC che esegue istruzioni SQL per:

- salvare (rendere persistenti) oggetti nel DB
- cancellare oggetti dal DB
- trovare oggetti dal DB

Si faccia riferimento alla classe `StudentRepository` (concentriamoci in particolare sul codice dei singoli metodi, piuttosto che del progetto di tale classe)

Istruzione SQL: uso di PreparedStatement

Per eseguire una istruzione SQL è necessario creare un oggetto della classe che implementa `PreparedStatement` creato dall'oggetto `Connection` invocando il metodo:

```
PreparedStatement prepareStatement(String s);
```

la stringa `s` è una istruzione SQL parametrica: i parametri sono indicati con il simbolo `?`

Esempio 1

```
String insert = "insert into students (code,
firstname, lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);
```

Esempio 2

```
String delete = "delete from students where code=?";
statement = connection.prepareStatement(delete);
```

Istruzione SQL: uso di PreparedStatement

I parametri sono assegnati mediante opportuni metodi della classe che implementa **`PreparedStatement`**

- metodi **`setXXX (<numPar>, <valore>)`**

  un metodo per ogni tipo, il primo argomento corrisponde all'indice del parametro nella query, il secondo al valore da assegnare al parametro

Esempio 1 (cont.)

```
PreparedStatement statement;
String insert = "insert into students (code, firstname,
lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);
statement.setInt(1, student.getCode());
statement.setString(2, student.getFirstName());
statement.setString(3, student.getLastName());

long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));
```

Osservazioni

JDBC usa **`java.sql.Date`**, mentre la classe Student usa **`java.util.Date`**

Le istruzioni

```
long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));
```

servono a "convertire" una data da una rappresentazione all'altra (per tutti i dettagli si consulti la documentazione)

Esecuzione istruzioni SQL

Una volta assegnati i valori ai parametri, l'istruzione può essere eseguita

Distinguiamo due tipi di operazioni:

- **aggiornamenti** (insert, update, delete): modificano lo stato del database
  - vengono eseguiti invocando il metodo **`executeUpdate()`** sull'oggetto **`PrepareStatement`**
- **interrogazioni** (select): non modificano lo stato del database e restituiscono una sequenza di tuple
  - vengono eseguite invocando il metodo **`executeQuery()`** che restituisce il risultato in un oggetto **`ResultSet`**

Aggiornamenti (insert, delete, update)

Esempio 1 (cont.)

```
PreparedStatement statement;
String insert = "insert into students(code,
firstname, lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);

statement.setString(1, student.getCode());
statement.setString(2, student.getFirstName());
statement.setString(3, student.getLastName());
long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));

statement.executeUpdate();
```

Interrogazioni (select)

Esempio 2 (cont.)

```
PreparedStatement statement;
String query = "select * from students where code=?";
statement = connection.prepareStatement(query);

statement.setInt(1,code);
ResultSet result = statement.executeQuery();
```

Gestire il risultato di una query

Esempio 2 (cont.)

```
String retrieve = "select * from students where code=?";
statement = connection.prepareStatement(retrieve);
statement.setInt(1, code);
ResultSet result = statement.executeQuery();
if (result.next()) {
   student = new Student();
   student.setCode(result.getInt("code"));
   student.setFirtsName(result.getString("firstname"));
   student.setLastName(result.getString("lastname"));
   long secs = result.getDate("birthdate").getTime();
   birthDate = new java.util.Date(secs);
   student.setBirthDate(birthDate);
}
```

Gestire il risultato di una query

Un altro esempio

```
List<Student> students = null;
Student student = null;
Connection connection = this.dataSource.getConnection();
PreparedStatement statement;
String query = "select * from students";
statement = connection.prepareStatement(query);
ResultSet result = statement.executeQuery();
if(result.next()) {
   students = new LinkedList<Student>();
   student = new Student();
   student.setCode(result.getInt("code"));
   student.setFirstName(result.getString("firstname"));
   …
```

Gestire il risultato di una query

```
    …
    student.setLastName(result.getString("lastname"));
    student.setBirthDate(
      new java.util.Date(result.getDate("birthdate")
                                          .getTime()));
    students.add(student);
}
while(result.next()) {
    student = new Student();
    student.setCode(result.getInt("code"));
    student.setFirstName(result.getString("firstname"));
    student.setLastName(result.getString("lastname"));
    student.setBirthDate(
      new java.util.Date(result.getDate("birthdate")
                                          .getTime()));
    students.add(student);
}
```

Gestire le eccezioni

- Tutti i metodi delle classi dell'API JDBC visti fino ad ora "lanciano" una eccezione **`SQLException`**
- *Connection*, *statement*, *ResultSet* devono essere **sempre "chiusi"** dopo essere stati usati
  - l'operazione di chiusura corrisponde al rilascio di risorse
- Effettuiamo queste operazioni nella clausola **`finally`**
  - abbiamo così la garanzia che vengano comunque effettuate
- Se vengono sollevate eccezioni, le rilanciamo con una eccezione di livello logico opportuno
- Il codice che segue è un po' verboso, ma didatticamente ci aiuta a ragionare ancora sulla gestione delle eccezioni

Clausola `finally`

```
finally {
   try {
       if (statement != null)
          statement.close();
       if (connection!= null)
          connection.close();
   }
   catch (SQLException e) {
       throw new PersistenceException(e.getMessage());
   }
}
```

Ora a voi (1)

Prendete spunto **dall'esercizio guidato appena mostrato** e dall'**Esercitazione 04** su JSP (nel dettaglio, esercizio «Negozio online» - slides 11-14)

1) Creare un classe **ShopRepository** che

a) renda persistente un ordine, costituito non solo dall'elenco degli item selezionati nel carrello (Java Bean Cart), ma anche **dall'email dell'acquirente**

b) offra la possibilità di ottenere l'ordine relativo ad una email

c) offra la possibilità di recuperare tutti gli ordini suddivisi per email

- quali problematiche insorgono su questo ultimo punto ragionando sullo schema DB corrispondente? Come possiamo superarle?

Nota 1: testare **senza componenti Web** e utilizzando DB2 e almeno uno tra HSQLDB e MySQL

2) Una volta testato il punto precedente, integrare le classi realizzate all'interno del progetto JSP (rif. file zip Esercitazione 04), creando

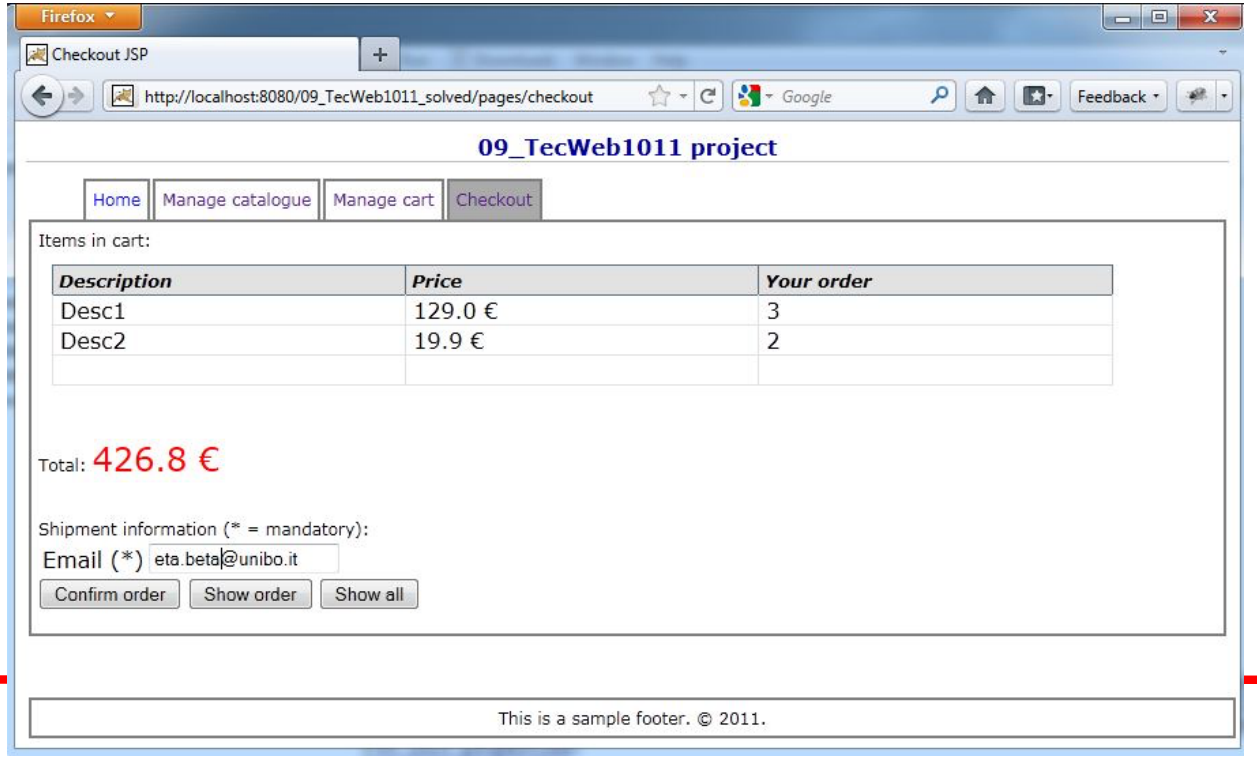
a) **checkout.jsp**: offre la possibilità di a) rendere persistente un ordine, b) data un'email recuperare l'ordine relativo, c) recuperare tutti gli ordini, suddivisi per email

) **CheckoutServlet**: riceve le richieste di checkout.jsp e accede al database tramite ShopRepository

Nota 2: focalizzarsi unicamente sulla **gestione dell'ordine** di un utente (non modificare la gestione del carrello)

Ora a voi (2)

- L'obiettivo dell'esercitazione è quello di rendere persistente l'ordine rappresentato dal carrello
  - per semplicità la soluzione proposta non decrementa il numero di oggetti presenti nel catalogo
- La JSP checkout.jsp accede al carrello degli articoli scelti dall'utente
  - dovrà avere accesso al Java Bean Cart (con scope di sessione)
  - al contrario non avrà bisogno di accedere al Java Bean Catalogue
- Si ricorda che il Java Bean **Cart** è realizzato mediante un oggetto di tipo **Map<Item, Integer>**, che associa ad un Item la quantità di tale Item inserita dall'utente nel suo carrello



APPENDICE

(esercizio guidato tratto dall'appello d'esame del 17 gennaio 2018: gestione persistenza basata su metodologia «forza bruta»)

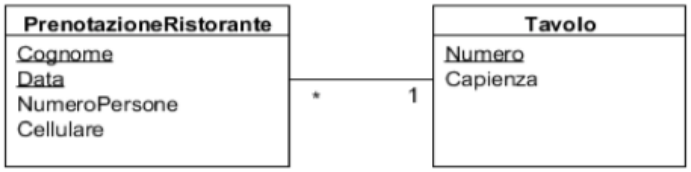
Agenda

- Esercizio tratto **dall'appello d'esame del 17 gennaio 2018**: presentazione esercizio
- Passi principali:
  - Dall'UML ai JavaBean
  - Mapping delle relazioni e dei vincoli
  - Implementazione delle query
  - Tricky elements

Esame 17/01/2018: Esercizio 3

ESERCIZIO 3 (11 punti)

Partendo dalla realtà illustrata nel **diagramma UML** di seguito riportato, si fornisca una soluzione alla gestione della persistenza basata su metodologia **Forza Bruta** in grado di "mappare" il modello di dominio rappresentato dai **JavaBean** del diagramma UML con le corrispondenti **tabelle relazionali derivata dalla progettazione logica** del diagramma stesso.



Si consideri inoltre la presenza del vincolo: "*Uno stesso tavolo può essere prenotato più volte solo se in date diverse*".

Nel dettaglio, dopo aver creato da applicazione Java lo schema della tabella nel proprio schema nel database **TW_STUD** di **DB2** (esplicitando tutti i vincoli derivati dal diagramma UML) e implementato **JavaBean** e metodi necessari per la realizzazione delle **operazioni CRUD**, si richiede la definizione del metodo principale di richiesta prenotazione `boolean RichiestaPrenotazione(String Cognome, Date data, Int numeroPersone, String Cellulare)`. Tale metodo, mediante l'uso del metodo di supporto `String NumeroTavolo DisponibilitaTavolo(Date data, Int numeroPersone)`, verifica la disponibilità di almeno un tavolo di `capienza >= numeroPersone` per la data richiesta e, in caso di esito positivo, restituisce il codice numerico (`NumeroTavolo`) di uno di questi. Tale codice è usato dal metodo `RichiestaPrenotazione` per procedere all'inserimento persistente della prenotazione nel DB e alla restituzione del valore di verità `true` attestante l'accettazione della prenotazione. In caso di esito negativo di disponibilità tavolo invece, il metodo `DisponibilitaTavolo` restituisce `null`, mentre il metodo principale `RichiestaPrenotazione` restituisce direttamente il valore di verità `false` attestante il rifiuto della prenotazione.

Si richiede quindi di realizzare una classe di prova in grado di:

- inserire diversi tavoli e diverse prenotazioni nelle tabelle corrispondenti;
- utilizzare correttamente i metodi `RichiestaPrenotazione` e `DisponibilitaTavolo` per verificare la disponibilità o meno di un tavolo rispetto a una determinata richiesta (si contempli sia il caso di risposta positiva che negativa);
- produrre una stampa completa, opportunamente formattata, delle prenotazioni (complete di numero tavolo) presenti nel DB **prima e dopo** l'inserimento al punto precedente sul file **Prenotazione.txt**.

Dall'Uml ai JavaBean

- La prima cosa da fare negli esercizi relativi alla gestione della persistenza è trasformare il diagramma UML nelle corrispondenti classi Java

- Ogni classe del diagramma corrisponde ad una classe java
- Ad ogni proprietà della classe UML corrisponde una proprietà della classe Java

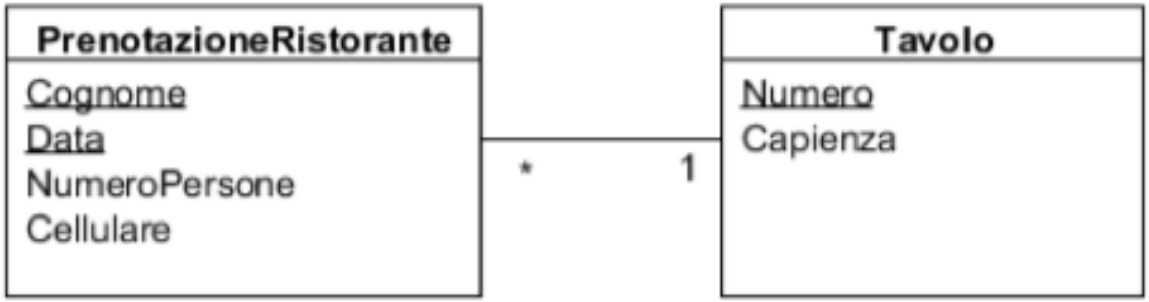
  Quindi, basta contare le classi e le proprietà UML per sapere quante classi e proprietà Java modellare?
  No, come vedremo non proprio è così…

Mapping delle relazioni

- La seconda cosa che può influire sul numero delle classi e sul numero di attributi in Java sono le relazioni tra le classi UML:
  - **Relazione 1-n/n-1**: non comportano classi java aggiuntive, ma solo l'aggiunta di attributi; la cardinalità **n** è mappata con una collezione di oggetti lato java, mentre la cardinalità **1** con un semplice id intero
  - **Relazione n-m**: comporta la creazione di una tabella aggiuntiva nel DB, detta tabella di mapping
    - La tabella di mapping potrebbe avere una sua corrispondente classe java (si parla in questo casi di «materializzazione» degli ID), oppure no (si gestisce esplicitamente la relazione n-m rappresentata nel diagramma UML).
    - La scelta dipende anche da eventuali vincoli esplicitati nel testo dell'esercizio!
- Il numero di attributi nelle classi java dipende anche dalla eventuale richiesta nel testo d'esame di utilizzo di «ID surrogati»
  - Sono attributi solitamente di tipo intero che fungono da **identificatori univoci** degli **oggetti JavaBean** e da **primary key (PK)** nelle tabelle corrispondenti
- **! Anche se la traccia d'esame non lo richiede, per completezza presentiamo una soluzione basata su id surrogati (ipotesi extra traccia)**

Il caso concreto del nostro esercizio

- Come sarà il mapping di questo diagramma?



  - Due classi Java, una per classe UML
  - Tutti gli attributi esplicitati nel diagramma
  - Un id surrogato (int) per classe
  - Una Collezione di oggetti PrenotazioneRistorante nella classe Tavolo
  - Un campo contente la chiave materializzata (l'ID tavolo) della tabella Tavolo nella casse PrenotazioneRistorante

```
public class PrenotazioneRistorante {

    private int idPrenotazione;
    private String cognomePrenotazione;
    private Date dataPrenotazione;
    private int numeroPersonePrenotazione;
    private String cellularePrenotazione;
    private int idTavoloPrenotazione;
```

```
public class Tavolo {

    private int idTavolo;
    private int numeroTavolo;
    private int capienzaTavolo;
    private Set<PrenotazioneRistorante> prenotazioniTavolo;
```

Mapping dei vincoli (1)

- Il mapping dei vincoli viene fatto nelle classi java che si occupano dell'interfacciamento con il DB, dette «**`Repository`**»
  - C'è una classe Repository per ogni JavaBean
    - Queste classi implementano tutti i **metodi CRUD**
  - In più ci sono ulteriori due classi strettamente relative al DB fisico
    - Una per la gestione della connessione al DB denominata **`DataSource()`**
    - Una per la gestione personalizzata delle eccezioni denominata **`PersistenceException()`**

- Vincoli esplicitati nell'UML:
  - La coppia Cognome-Data in PrenotazioneRistorante è univoca (UNIQUE)
  - Il Numero tavolo in Tavolo è univoco (UNIQUE)
- Vincoli impliciti o espressi nel testo, tipo:

Si consideri inoltre la presenza del vincolo: "*Uno stesso tavolo può essere prenotato più volte solo se in date diverse*".

- ! Per date diverse si intende «giorni diversi»

Mapping dei vincoli (2)

- Vediamo come si implementano i vincoli individuati…
  - L'implementazione dei vincoli viene demandata al metodo della classe "Repository" che si occupa della creazione della tabella, ovvero alla query "Create Table"

```
// create table
private static String create =
    "CREATE " +
        "TABLE " + TABLE +" ( " +
            ID + " INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1), " +
            COGNOME + " VARCHAR(20) NOT NULL, " +
            DATA + " DATE NOT NULL, " +
            NUMEROPERSONE + " INT, " +
            CELLULARE + " VARCHAR(10), " +
            TAVOLO + " INT NOT NULL REFERENCES tavolo, " +
            "PRIMARY KEY ("+ID+"), "+
            "CONSTRAINT pt_PrenotazioneID UNIQUE ("+COGNOME+","+DATA+"), " +
            "CONSTRAINT pr_PranotazioneTavoloID UNIQUE ("+DATA+","+TAVOLO+") " +
        ") "
    ;
```

! La definizione completa dell'id surrogato **ID** la vediamo tra qualche minuto…

Implementazione query

- Le query richieste dall'esercizio vengono implementate dai seguenti metodi:
  - `boolean RichiestaPrenotazione(String cognome, Date data, int numeroPersone, String cellulare)`
  - `String DisponibilitaTavolo(Date data, int numeroPersone)`
- Il primo metodo, tramite l'utilizzo del secondo, verifica la disponibilità di un tavolo (ovvero, **`capienza >= numero di persone`**); se è disponibile un tavolo, il secondo metodo restituisce il suo codice numerico, altrimenti null;
- Se un tavolo è disponibile, il primo metodo usa il numero di tavolo restituito per inserire persistentemente nel DB la prenotazione

Query: DisponibilitàTavolo (1)

- Il metodo deve restituire il numero di un tavolo, qualora ci fosse, con capienza >= al numero di persone richiesto ad una certa data
- Questo risultato può essere ottenuto in due modi differenti:
  1. Effettuando due query distinte, una che seleziona tutti i tavoli e l'altra tutte le prenotazioni per la data richiesta, per poi fare un controllo incrociato delle due collezioni con Java
     - Metodo complicato, laborioso e molto poco efficiente
  2. Oppure con un'unica query innestata
     - Metodo pulito, efficiente ed elegante
- La query viene implementata dal metodo dedicato nella classe «`TavoloRepository()`»

Query: DisponibilitàTavolo (2)

```
private static String DisponibilitàTavolo(Date data, int persone)
{
    return tr.availableTable(data, persone);
}
```

```
public String availableTable(Date data, int persone)
{
    String result=null;
    Connection connection = null;
    PreparedStatement statement = null;
    try{
        connection = this.dataSource.getConnection();
        statement = connection.prepareStatement(read_available_table);
        statement.setInt(1, persone);
        statement.setDate(2, data);
        ResultSet rs = statement.executeQuery();
        if(rs.next())
        {
            result = rs.getString(NUMERO);
        }
        else
            result = null;
        return result;
    }catch (SQLException e) {
        throw new PersistenceException(e.getMessage());
    }
    finally {
        try {
            if (statement != null)
                statement.close();
            if (connection!= null){
                connection.close();
                connection = null;
            }
        }
        catch (SQLException e) {
            throw new PersistenceException(e.getMessage());
        }
    }
}
```

Query: DisponibilitàTavolo (3)

- Di seguito la query innestata "*read_available_table":*

```
static String read_available_table =
        "SELECT "+ NUMERO +
            " FROM " + TABLE + " " +
            "WHERE " + "capienza" + " >= ? AND "+ID+" NOT IN ( SELECT idTavolo FROM prenotazione WHERE data = ?)";
```

- L'inner query seleziona tutti gli ID dei tavoli prenotati nella data richiesta
- L'outer query seleziona tutti I tavoli con capienza >= al numero di persone con ID diverso da tutti quelli restituiti dall'inner query

Query: RichiestaPrenotazione (1)

- Questa query prende il codice del tavolo restituito dal primo metodo e va a fare un'operazione di insert sulla tabella delle prenotazioni del ristorante

```
private static boolean RichiestaPrenotazione(String cognome, Date data, int persone, String cellulare)
{
    String tableavailable = DisponibilitàTavolo(data, persone);
    if(tableavailable == null)
        return false;
    int numTavolo = Integer.parseInt(tableavailable);
    PrenotazioneRistorante prr = new PrenotazioneRistorante();
    prr.setCellularePrenotazione(cellulare);
    prr.setCognomePrenotazione(cognome);
    prr.setDataPrenotazione(data);
    prr.setIdTavoloPrenotazione(numTavolo);
    prr.setNumeroPersonePrenotazione(persone);
    pr.persist(prr);
    return true;
}
```

- Il metodo restituisce TRUE se l'operazione è andata a buon fine, FALSE altrimenti

Query: RichiestaPrenotazione (2)

```
public void persist(PrenotazioneRistorante pr) throws PersistenceException{
    Connection connection = null;
    PreparedStatement statement = null;

        try {
            connection = this.dataSource.getConnection();
            statement = connection.prepareStatement(insert);
            statement.setString(1, pr.getCognomePrenotazione()+"");
            statement.setDate(2, pr.getDataPrenotazione());
            statement.setInt(3, pr.getNumeroPersonePrenotazione());
            statement.setString(4, pr.getCellularePrenotazione());
            statement.setInt(5, pr.getIdTavoloPrenotazione());
            statement.executeUpdate();

            statement = connection.prepareStatement(check_query);
            statement.setString(1, pr.getCognomePrenotazione());
            statement.setDate(2, pr.getDataPrenotazione());
            ResultSet rs = statement.executeQuery();
            rs.next();
            int idprenotazione = rs.getInt(1);
            pr.setIdPrenotazione(idprenotazione);
        }
        catch (SQLException e) {
            throw new PersistenceException(e.getMessage());
        }
        finally {
            try {
                if (statement != null)
                    statement.close();
                if (connection!= null){
                    connection.close();
                    connection = null;
                }
            }
            catch (SQLException e) {
                throw new PersistenceException(e.getMessage());
            }
        }
}
```

Tricky Elements

- In questo esercizio d'esame c'era un elemento più o meno esplicito…
- il metodo che rende persistente una prenotazione, non accetta tra i parametri l'ID surrogato della prenotazione presente nel rispettivo oggetto **`PrenotazioneRistorante`**
  - Ovviamente non è possibile aggiungere il parametro ID alla signature del metodo perché in questo modo non verrebbe rispettata la specifica del testo… occorre trovare un altro modo…

Tricky Elements: ID auto-incrementale

- Come possiamo inserire una nuova tupla senza preoccuparci di esplicitarne il valore?
  - Attraverso l'uso di ID auto-incrementali
- Come si implementa un ID auto-incrementale?
  - È un ID generato automaticamente dal DB all'inserimento di una nuova

```
// create table
private static String create =
    "CREATE " +
        "TABLE " + TABLE +" ( " +
            ID + " INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1), " +
            COGNOME + " VARCHAR(20) NOT NULL, " +
            DATA + " DATE NOT NULL, " +
            NUMEROPERSONE + " INT, " +
            CELLULARE + " VARCHAR(10), " +
            TAVOLO + " INT NOT NULL REFERENCES tavolo, " +
            "PRIMARY KEY ("+ID+"), "+
            "CONSTRAINT pt_PrenotazioneID UNIQUE ("+COGNOME+","+DATA+"), " +
            "CONSTRAINT pr_PranotazioneTavoloID UNIQUE ("+DATA+","+TAVOLO+") " +
        ") "
    ;
```

! Una volta inserita la tupla nella tabella, occorre allineare il valore dell'ID generato con quello del corrispondente oggetto JavaBean