

*Corso di Laurea in Ingegneria Informatica*

# Corso di Ingegneria del Software

---

## JDBC e DAO

# Sommario

- JDBC
- Impedance Mismatch e ORM

Riferimenti:

Craig Larman; *Applicare UML e i pattern. Analisi e progettazione orientata agli oggetti*, III edizione, Addison Wesley Professional. **Cap.** 38.

Documentazione Ufficiale JDBC:

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

<http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/index.html>

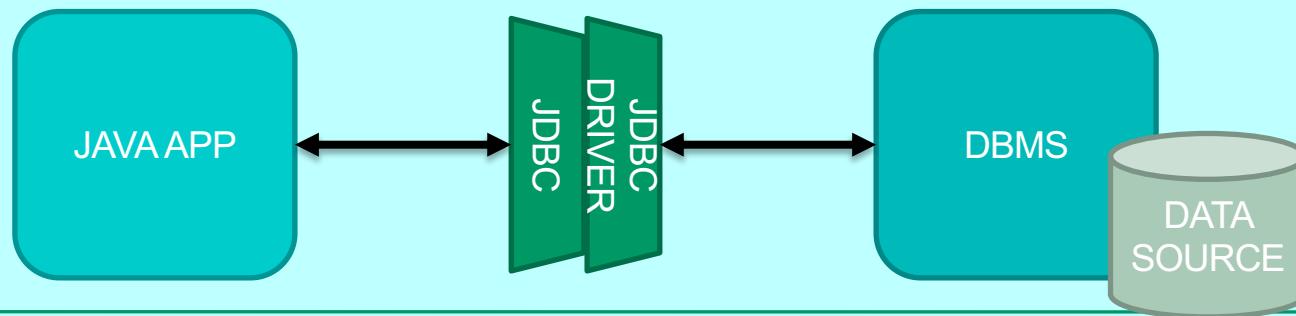
# JDBC

- ◆ Librerie Java che permettono l'accesso e l'interazione con database,  
in particolare i RDBMS
- ◆ Le API JDBC permettono di:
  - Connetersi ad una sorgente dati (e.g., un database)
  - Inviare interrogazioni (query) e aggiornare le istruzioni (statement) al database
  - Recuperare e processare i risultati ricevuti dal database in risposta alla query
- ◆ Permette ai programmi Java di interagire con i database con un'interfaccia comune, indipendente dal tipo di database e dalla piattaforma.
- ◆ Gli elementi JDBC sono contenuti nel *package java.sql*

# Architettura JDBC

## ◆ JDBC ha due tipi di interfacce

- Il primo è usato dagli sviluppatori di applicazioni: tramite queste le applicazioni Java possono interagire con i DBMS
- Il secondo è utilizzato dai produttori di DBMS: essi possono aggiungere moduli driver JDBC per permettere alle applicazioni Java di interagire con nuove sorgenti dati



# Tipologie di JDBC driver

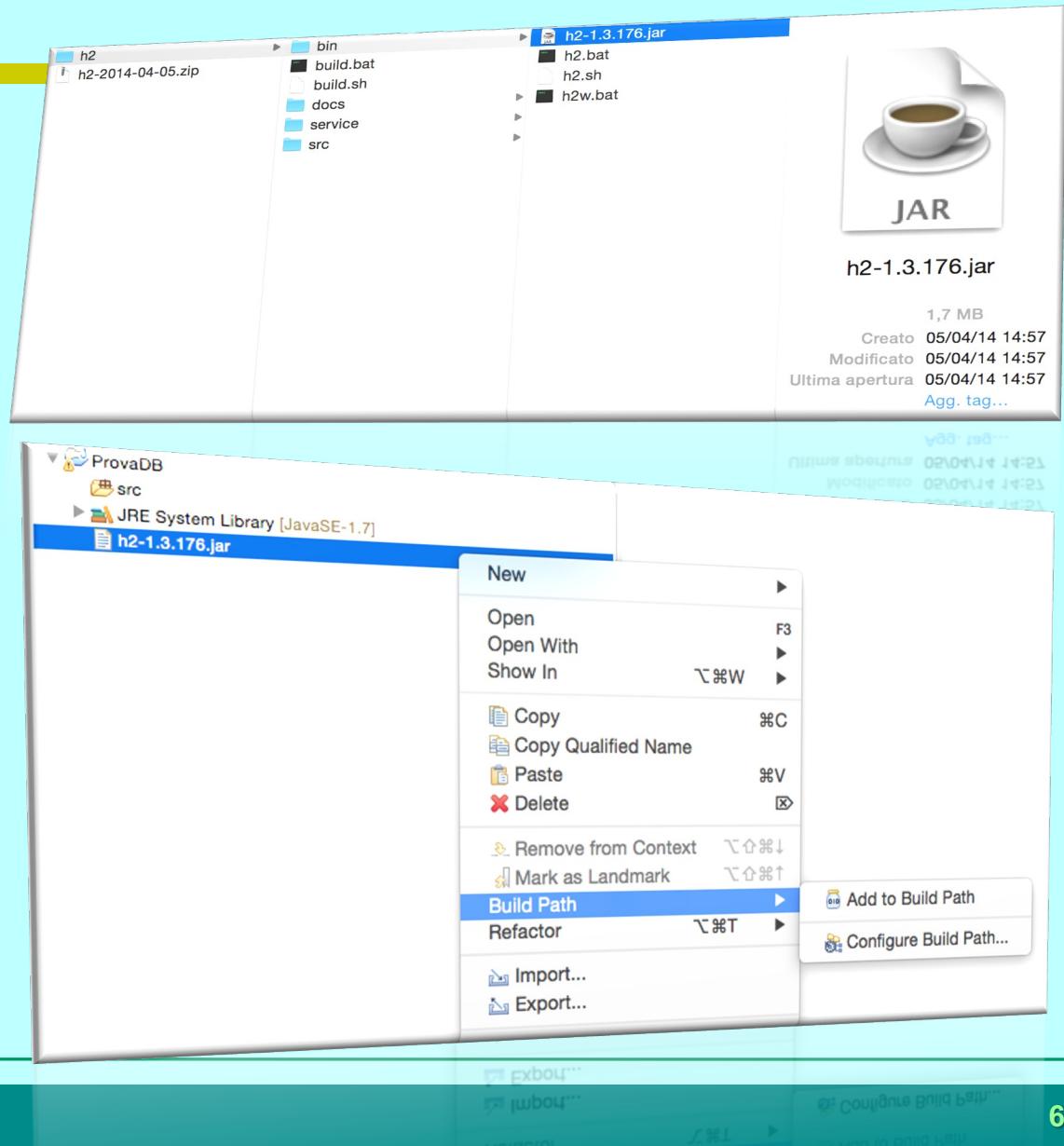
- ◆ Il driver per JDBC può essere indipendente dalla piattaforma. Ciò può limitare la portabilità delle applicazioni Java che usano JDBC.
- ◆ Possiamo distinguere quattro tipologie diverse di JDBC driver:
  1. Driver che implementano le JDBC API **incapsulando un'altra API** per l'accesso ai dati. Generalmente basati su **librerie native** e, quindi, poco portabili (es. il bridge JDBC-ODBC, «Open Database Connectivity»)
  2. Driver **in parte Java e in parte native**. Sono degli adattatori a librerie native e, ancora una volta, poco portabili.
  3. Driver puramente Java che comunicano con **un server middleware**, che a sua volta si preoccupa di inoltrare le richieste alla sorgente dati.
  4. Driver puramente Java che **implementano il protocollo per la specifica sorgente dati** e che, quindi, comunicano direttamente con essa. (es. **H2 database**)

Installare un driver JDBC generalmente consiste nel copiare il driver sul computer, quindi **aggiungere la sua posizione nel class path**. Driver diversi dalla tipologia 4 possono richiedere l'installazione di un client.

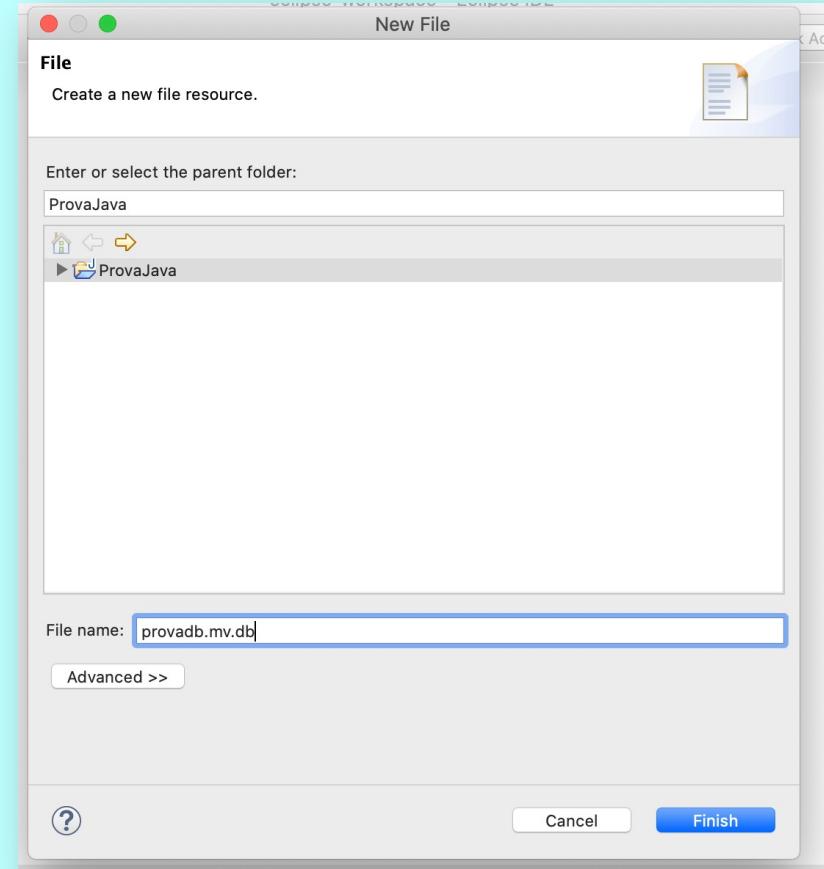
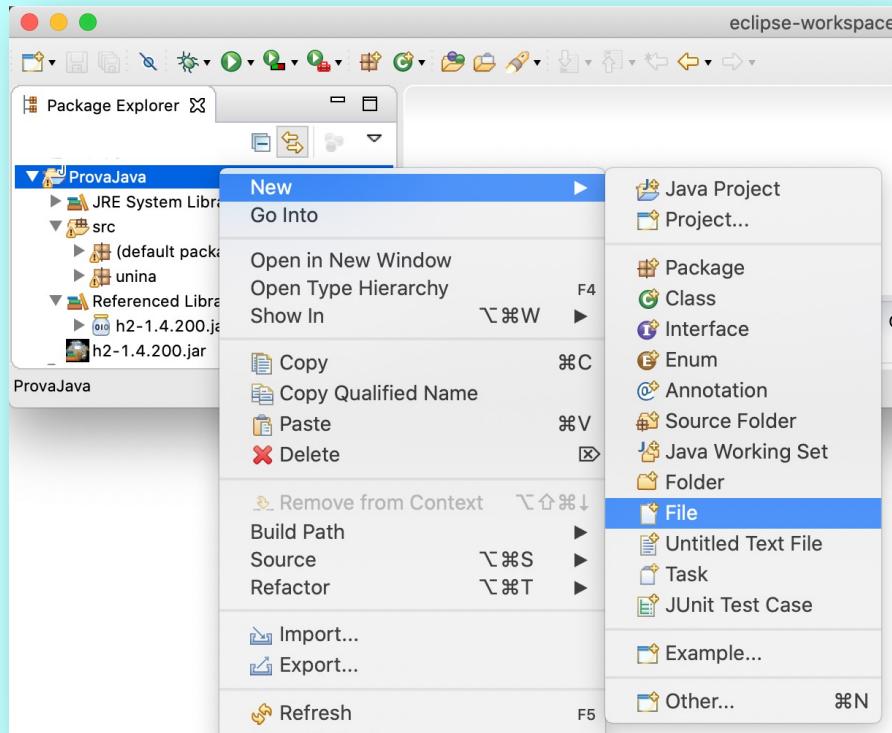
# Installazione H2 Driver

1. Download H2 (Last Stable, Platform-Independent Zip)
2. Scompattare H2
3. Copiare il file *.jar* nel progetto Eclipse
4. Aggiungere il file *.jar* al build path

Per inizializzare il database è possibile eseguire il file *.jar* e utilizzare l'interfaccia web

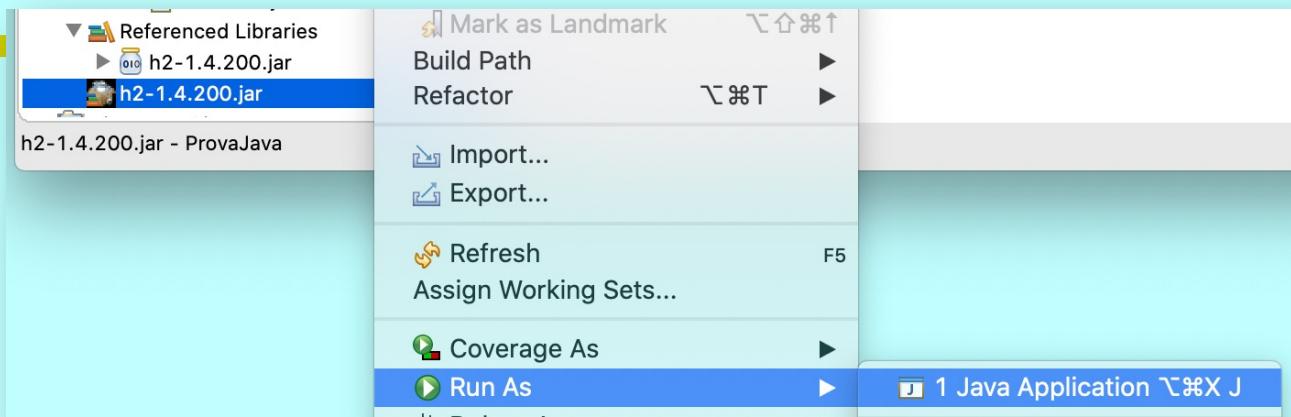


# Inizializzare il database

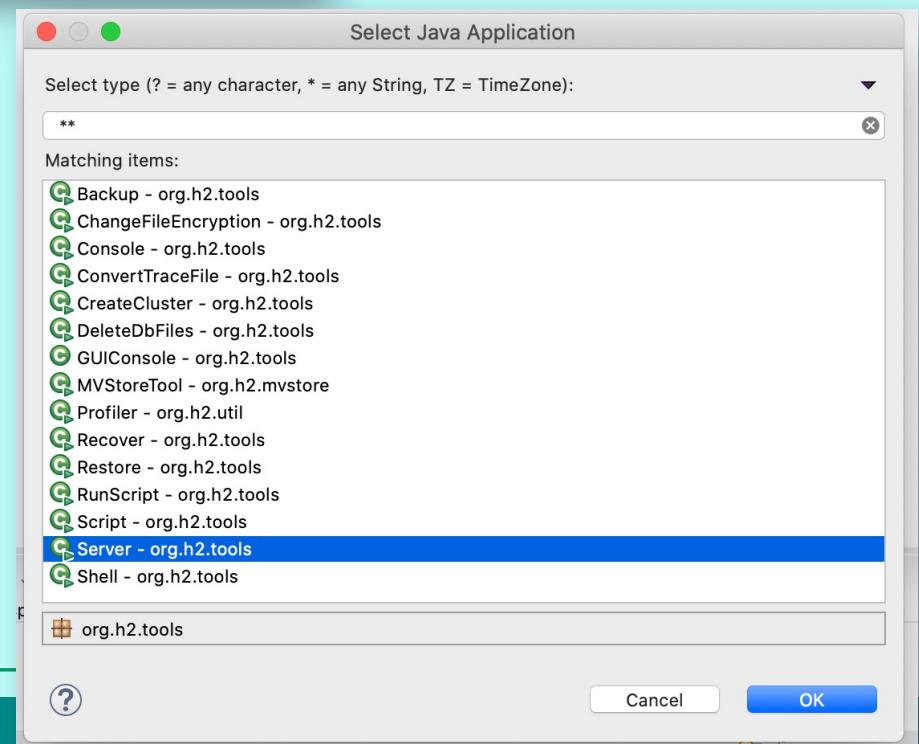


**Creare nella radice del progetto un file vuoto con estensione «.mv.db», che conterrà il database**

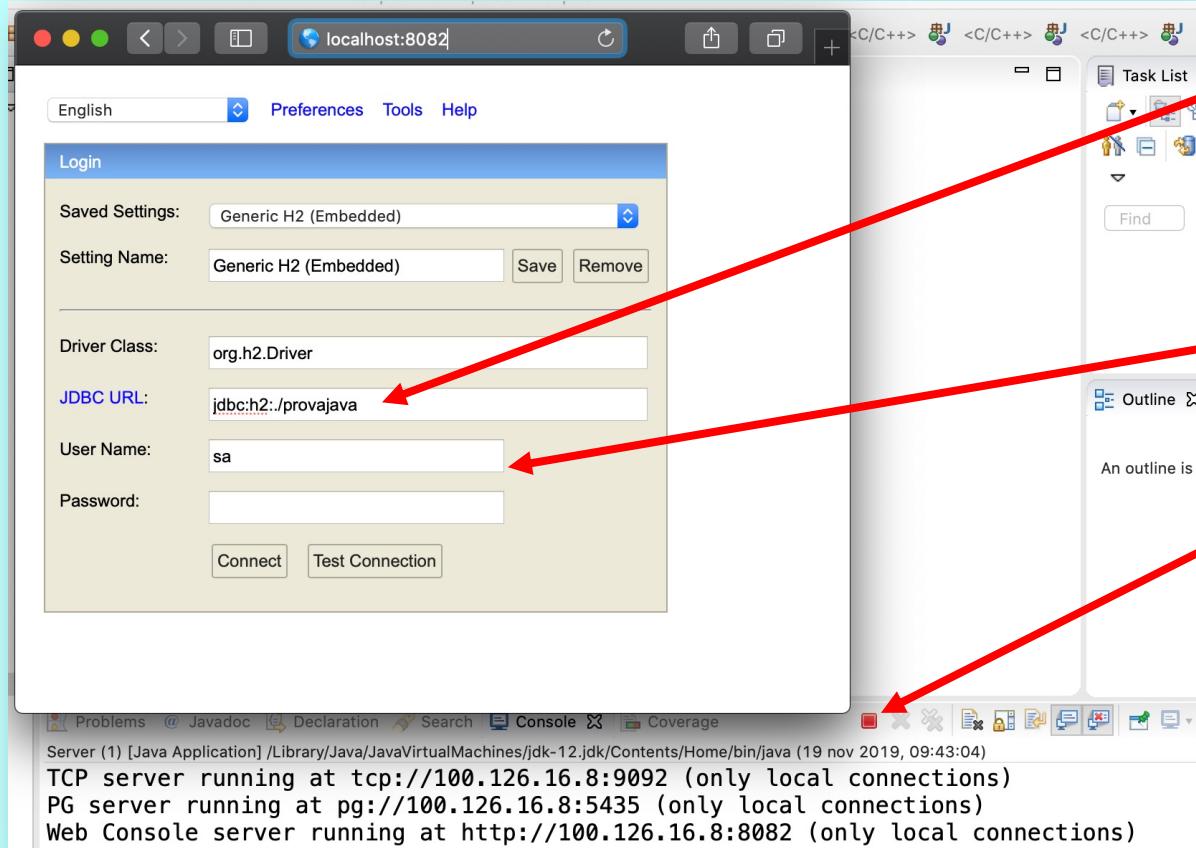
# Inizializzare il database



**Per utilizzare il database tramite Web Console, si avvii il server mediante il file .jar, e selezionando la classe «Server»**



# Inizializzare il database



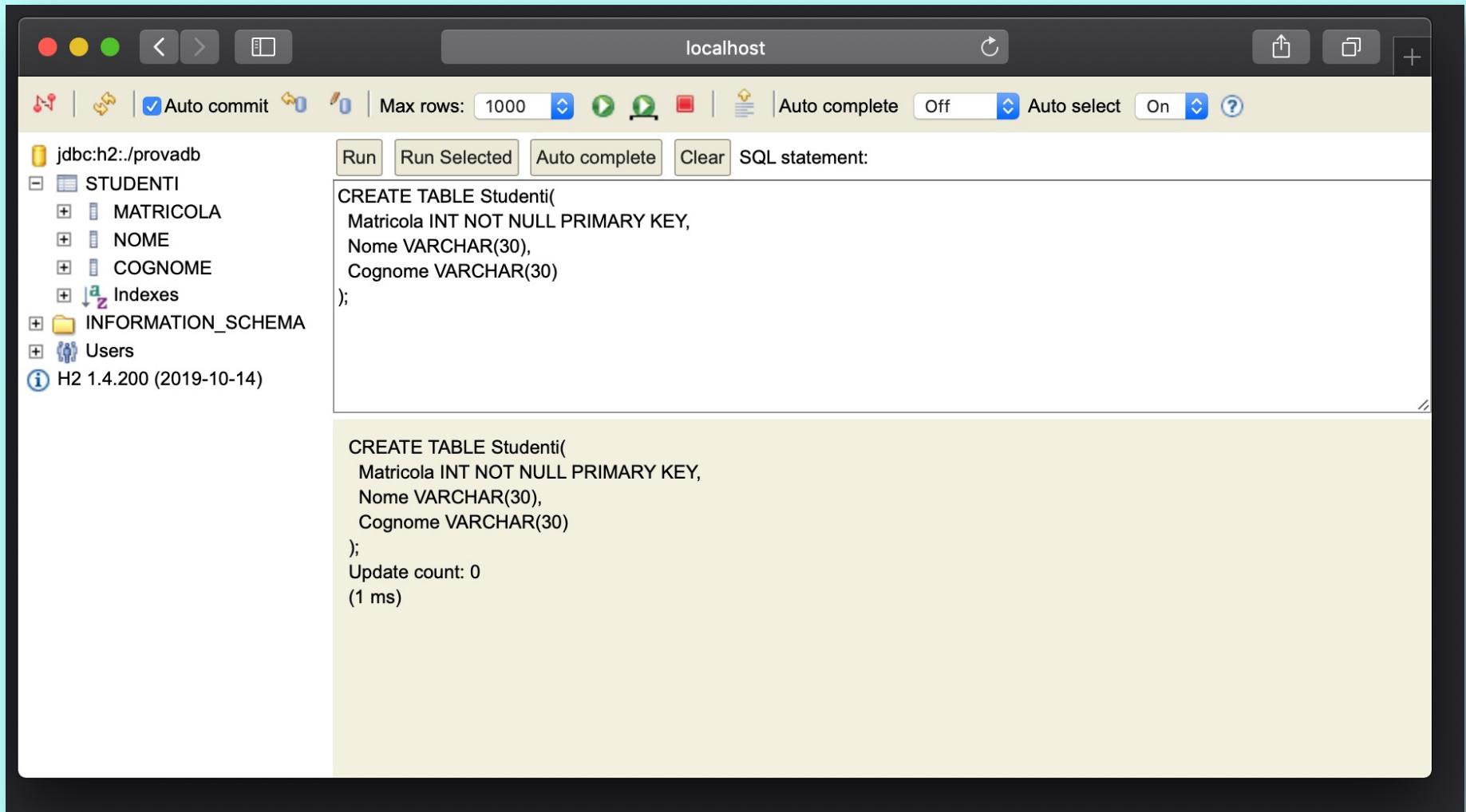
Indicare il nome del file che è stato creato (senza estensione «.mv.db»)

User e pass di default

Per interrompere il server (da utilizzare quando si smette di usare la Web Console, prima di avviare il nostro programma Java)

Il driver avvia un thread che rimane in ascolto sul port 8082 (Web Console server, da aprire con il browser)

# Inizializzare il database



The screenshot shows the H2 Database Tools interface running in a macOS window. The title bar says "localhost". The left sidebar shows the database structure:

- jdbc:h2:./provadb
- STUDENTI
  - MATRICOLA
  - NOME
  - COGNOME
  - Indexes
- INFORMATION\_SCHEMA
- Users
- H2 1.4.200 (2019-10-14)

The main area contains the SQL statement for creating the "Studenti" table:

```
CREATE TABLE Studenti(
    Matricola INT NOT NULL PRIMARY KEY,
    Nome VARCHAR(30),
    Cognome VARCHAR(30)
);
```

Below the SQL statement, the results of the execution are shown:

```
CREATE TABLE Studenti(
    Matricola INT NOT NULL PRIMARY KEY,
    Nome VARCHAR(30),
    Cognome VARCHAR(30)
);
Update count: 0
(1 ms)
```

# Package `java.sql`

- ♦ Relativamente ai Driver JDBC il package possiede
  - **DriverManager** (Class)
    - ◆ Il servizio di base per la gestione di un set di driver JDBC
    - ◆ Le applicazioni Java devono interagire con esso per instaurare una connessione con una sorgente dati
  - **Driver** (Interface)
    - ◆ L'interfaccia che ogni Driver JDBC deve realizzare

Altri elementi nel package sono:

- **Connection** (Interface)
  - ◆ Definisce una connessione (sessione) con uno specifico database.
  - ◆ Nel contesto di una connessione vivono oggetti Statement e PreparedStatement.
- **Statement** e **PreparedStatement** (Interface)
  - ◆ Metodi per eseguire istruzioni SQL, transazioni e interrogazioni al DBMS all'interno di una sessione.
- **ResultSet** (Interface)
  - ◆ Rappresenta una tabella di dati che contiene i risultati di una interrogazione posta al DBMS.
  - ◆ Un oggetto ResultSet mantiene un cursore alla riga (tupla) corrente di dati letti e scorrendolo si possono ottenere i valori delle varie righe risultato della query.

# Passi per l'esecuzione di una query SQL

---

1. Stabilire una connessione
2. Creare uno statement
3. Eseguire la query
4. Processare i risultati, se previsti
5. Rilasciare le risorse

I passi 1. e 2. non devono essere eseguiti necessariamente ogni volta che si interroga il database. Basta anche eseguirli solo una volta all'inizio dell'esecuzione dell'intero software.

---

# 1. Stabilire una connessione

- ◆ A partire da JDBC 4.0, ogni driver presente nel **build path** viene automaticamente caricato.
- ◆ Prima di interrogare il database bisogna stabilire una connessione:

*Connection conn = DriverManager.getConnection(url, login, password);*

- ◆ Il primo parametro di *getConnection* è una **stringa di connessione** che dipende dallo specifico driver in uso. In genere si presenta nella forma «*jdbc:subprotocol:subname*»

```
String dbName = "./test";
String username = "admin";
String password = "";
Connection conn =
    DriverManager.getConnection("jdbc:h2:"+dbName,
                               username, password);
```

## 2. Creare uno Statement

- ◆ Gli oggetti Statement consentono di eseguire istruzioni SQL sul database nel contesto di una connessione
  - *Statement stmt = conn.createStatement();*
- ◆ Uno statement può eseguire una query passata tramite un oggetto String, **concatenando sottostringhe** con «SELECT», «FROM», etc.
- ◆ L'operazione di composizione dell'istruzione SQL mediante concatenazione di sottostringhe è **sconsigliata**, poiché può portare facilmente a vulnerabilità ad attacchi di **«SQL Injection»**.
- ◆ Gli oggetti PreparedStatement sono una alternativa che consente di creare statement SQL «parametrizzati»
  - Garantiscono istruzioni SQL ben formattate
  - Possono essere precompilati, migliorando le prestazioni se una istruzione SQL è eseguita più volte

# PreparedStatement

- ◆ In una istruzione SQL **PreparedStatement** si inseriscono dei **placeholder** («?») in luogo dei parametri effettivi da utilizzare per completare lo statement.

*PreparedStatement stmt = conn.prepareStatement(*

*"UPDATE STUDENTI SET VALUES Name=? WHERE ID=?");*

- ◆ Si utilizzano i **metodi di set** per sostituire i placeholder con i parametri effettivi, che devono essere **compatibili con i tipi SQL** definiti per i parametri di input
- ◆ Un **PreparedStatement** può essere riutilizzato per ripetere la query più volte. Esso conserva i valori dei parametri finché non sono modificati con operazioni di **set** oppure o di **clearParameters**.
  - stmt.setString(1, "Mario");
  - stmt.setInt(2, 102);
  - stmt.executeUpdate();
- ◆ Nota: Gli indici partono da 1

# 3. Eseguire una query

- Per eseguire una query, si chiama uno dei metodi **execute\*** della classe *Statement*:
  - execute**: Utilizzabile con qualunque tipo di query. Restituisce «true» se la query genera dei risultati. I risultati sono recuperabili invocando il metodo *Statement.getResultSet()* (restituisce **ResultSet**)
  - executeQuery**: Utilizzabile con query di tipo SELECT, restituisce un *ResultSet*
  - executeUpdate**: Utilizzabile per query di tipo INSERT, DELETE e UPDATE. Restituisce un intero che indica il numero di righe modificate dallo statement SQL.

```
Statement stmt = conn.createStatement();
ResultSet r = stmt.executeQuery("SELECT * FROM TEST");
```

# 4. Elaborare i risultati

- ◆ Quando un **ResultSet** punta ad una riga valida, è possibile leggere gli attributi della tupla mediante un insieme di **metodi get** che ResultSet espone.
- ◆ Esiste un **metodo get per almeno ogni tipo di dato primitivo Java** verso cui si desidera convertire un attributo della tupla
  - `getInt`, `getShort`, `getBoolean`, `getDouble`, ...
- ◆ Le funzioni membro `get` sono overloaded: si può specificare la colonna da leggere in base al nome o all'indice numerico (i-esima colonna).
  - Gli indici di colonna partono da 1...
- ◆ I metodi `get` gestiscono differentemente il caso di campo `NONE` (vedi Javadoc).
  - Per sapere se nell'ultima operazione di `getTIPO` il campo era `NONE` è possibile servirsi del metodo `wasNull()`

```
while (r.next()) {
    int id = r.getInt("ID");
    if (r.wasNull()) System.out.println("id is null");
    String s = r.getString("NAME");
    if (r.wasNull()) System.out.println("name is null");
    System.out.println(id + ":" + s);
}
```

# 5. Rilasciare le Risorse

- ◆ I ResultSet sono associati agli statement
  - La chiusura di uno Statement chiude il ResultSet associato
  - Per ogni Statement al più un unico ResultSet può essere aperto allo stesso tempo.  
Se uno Statement è utilizzato per eseguire due query, la seconda query chiude il primo ResultSet.
- ◆ È consigliabile rilasciare appena possibile le risorse impegnate dagli oggetti ResultSet, Statement e Connection mediante i metodi close() esposti dalle classi.
- ◆ Nelle versioni successive JDBC 4.1 è possibile utilizzare il costrutto **try-with-resources**

```
try(Statement stmt = conn.createStatement()){  
    //...  
}catch(SQLException e) {  
    //...  
}
```

# OBJECT-RELATIONAL MAPPING

# Il paradigma OO e il diagramma ER

- ◆ I diversi componenti delle architetture presentate sono implementate, generalmente, con tecnologie object-oriented (OO) oramai consolidate, basate ad es. sull'uso dei linguaggi C++, Java, etc.
- ◆ Le metodologie di progettazione OO prevedono l'uso di linguaggi di modellazione concettuale, come UML.
- ◆ I dati sono memorizzati in memoria secondaria in un DBMS relazionale (organizzati in una base di dati, eventualmente ottenuta a partire da uno schema ER).

# Impedance mismatch

- ◆ Il binomio **applicazione OO - DBMS relazionale** è spesso una scelta obbligata.
- ◆ Mentre, però, il paradigma OO si è sviluppato su principi propri dell'ingegneria del software, quello relazionale affonda le radici su principi matematici, e le differenze fra i due risultano considerevoli.
- ◆ ***Impedance mismatch*** è un termine per denotare la mancata corrispondenza tra modello OO e relazionale ( termine preso in prestito dall'ingegneria elettrica )
- ◆ Nel modello OO, le relazioni tra classi sono realizzate con associazioni, mentre nel modello relazionale sono realizzate con i valori.
- ◆ In un oggetto, tutti i dati sono contenuti nell'oggetto medesimo (coesione); i corrispondenti dati relazionali possono essere invece contenuti in più tabelle, occorre conoscere la struttura del DB per accedervi.
- ◆ In un oggetto, la “business logic” risiede (parzialmente) nei metodi dell’oggetto stesso, mentre i dati relazionali sono esclusivamente statici e la logica deve essere implementata altrove.

# Object-Relational Mapping (ORM)

- ◆ Con **object-relational mapping** (ORM) si intende il meccanismo che permette la corrispondenza fra le entità una applicazione e le entità di un database relazionale
  - Nello specificare tale mappatura, è necessario risolvere opportunamente il problema dell'impedance mismatch.
  - Permette di utilizzare un RDBMS per memorizzare gli oggetti dell'applicazione.
- ◆ Lo stato degli oggetti persistenti viene memorizzato e ripristinato dalla base dati relazionale in base alle regole del mapping
  - Possiamo invocare operazioni di creazione, modifica, cancellazione su tali oggetti con l'obiettivo di effettuare letture e modifiche sulla base dati sottostante.

# Possibili soluzioni per la persistenza

1. Forza Bruta
2. Oggetti per accesso ai dati (DAO)
3. Persistence framework

# 1. Forza Bruta

- ◆ Prevede di equipaggiare le classi dell'applicazione (in particolare le classi di dominio) con metodi che interagiscono direttamente con la base di dati, “ovunque” questo sia necessario.
- ◆ Il mapping è realizzato attraverso la definizione di opportuni statement SQL (eventualmente raccolti in transazioni) scritti manualmente dal programmatore.
- ◆ È adatto per applicazioni semplici perché non incapsula la logica di accesso al DB.
- ◆ Per quanto possa essere la tecnica più semplice (con minore scrittura di codice) per applicazioni semplici, crea una dipendenza eccessiva tra la logica del programma e il database scelto
  - Difficile riusare codice in caso di cambio di tipologia di database

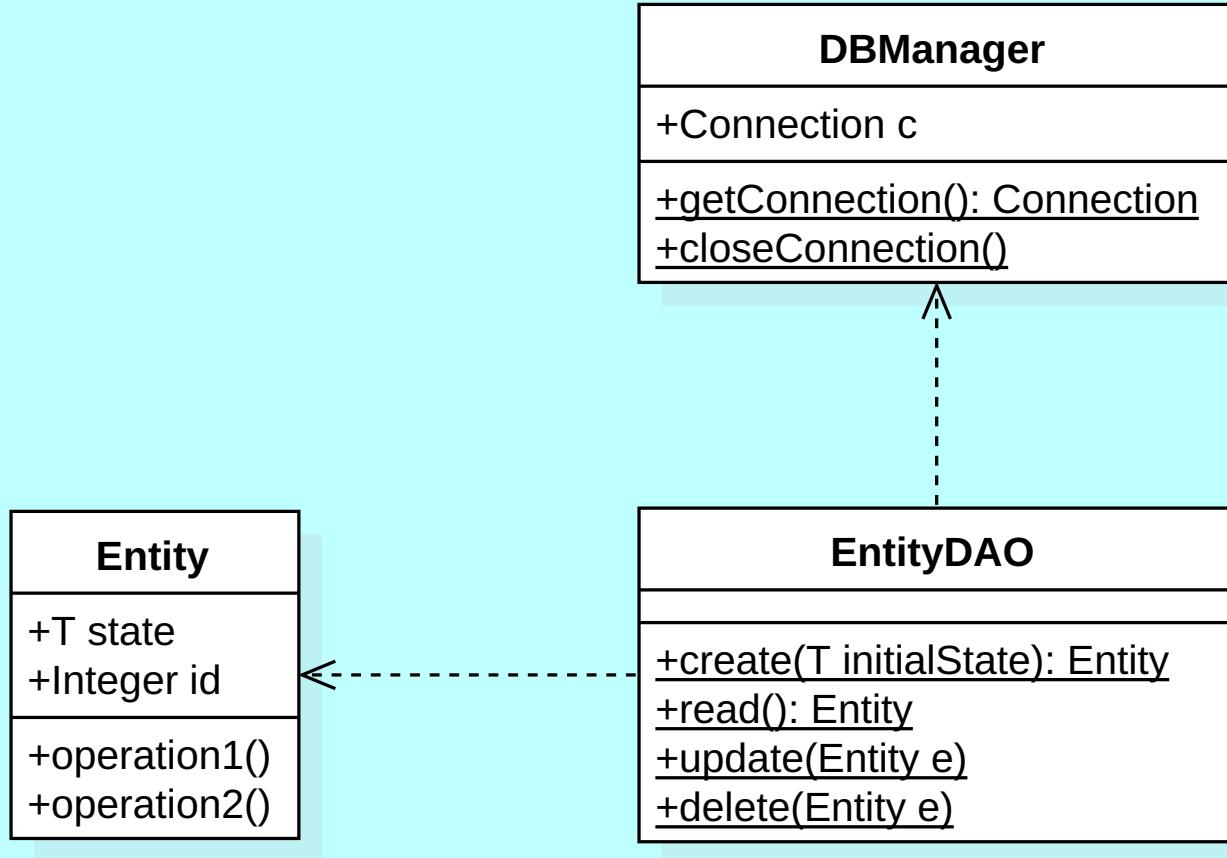
## 2. Direct Access Object (DAO)

- ◆ Prevede di realizzare uno strato dell'applicazione (chiamato appunto DAO) demandato completamente a gestire la comunicazione fra l'applicazione ed il DBMS.
- ◆ Anche in questo caso il mapping è realizzato manualmente attraverso l'uso di SQL.
- ◆ L'accesso al DB viene però opportunamente **incapsulato**, migliorando la modularità e l'interfacciamento esplicito del codice, e fornendo uno schema di riferimento in cui è possibile migliorare problemi di accoppiamento tipici dell'approccio forza bruta.
- ◆ Tutta la logica di accesso al DB è completamente encapsulata nelle Data Access Classes.
- ◆ Cambiamenti del DB influenzano **solo** le Data Access Classes.
- ◆ L'approccio tipico **per le operazioni che coinvolgono una sola entità** è quello di avere un DAO per ciascuna domain class.

# Progettazione DAO

- ◆ **Come raggruppare le operazioni sul db?** Cioè, quante classi DAO realizziamo in base alle operazioni da implementare?
- ◆ Si crea una classe DAO per ogni classe che rappresenta entità del dominio di applicazione.
- ◆ Questa classe DAO conterrà i metodi di interrogazione e manipolazione della corrispondente classe di dominio In particolare conterrà le funzionalità CRUD
  - Create
  - Read
  - Update
  - Delete

# DAO (Data Access Object) pattern



# 3. Persistence Framework

- ◆ Prevede l'utilizzo di un *framework* predefinito per la gestione della persistenza.
- ◆ L'obiettivo è liberare il programmatore quanto più possibile dalla necessità di scrivere codice SQL nella sua applicazione.
- ◆ Il codice SQL viene generato automaticamente sulla base di informazioni di meta-livello fornite dal programmatore (ad es. all'interno di file di configurazione).
- ◆ Incapsulamento completo: il programmatore vede il DB solo quando configura il *framework*
  - Esempi: EJB 3.0 e Hibernate per Java; NDO (.Net Data Objects) per .NET