UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Programmazione Object Oriented

Lezione – Persistenza in Java

Docente: Christian Esposito

Classe Resto 1 - Anno accademico 2020/21

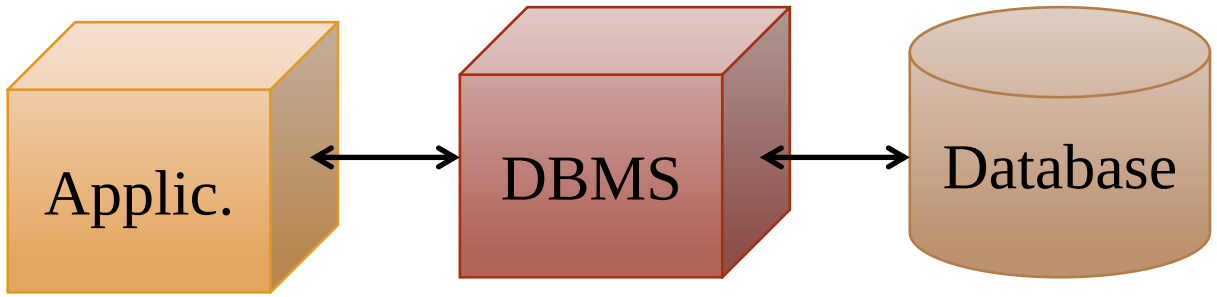
Basi di Dati

- Una base di dati è una collezione di dati organizzata in modo da poter essere facilmente consultata e aggiornata.
- In un database relazionale i dati sono organizzati in tabelle, strutturati in colonne (che rappresentano attributi) e in righe (che rappresentano le istanze dei dati memorizzati).

Nome	Cognome	Matricola	Email	Telefono
Esposito	Gennaro	58-0056	gennaro.esposito@studenti.unina.it	3456745334
Russo	Maria	45-33456	maria.russo@studenti.unina.it	3331245345
Rossi	Giovanni	21-4567	giovanni.rossi@studenti.unina.it	3209854678
Bugno	Valentina	32-56732	valentina.bugno@studenti.unina.it	3124312345

DBMS

- I database sono gestiti da un sistema software che prende il nome di DataBase Management System (DBMS), progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database.
- Le applicazioni possono accedere ai dati attraverso i servizi del DBMS.
- Un DBMS disaccoppia le applicazioni e i dati, in modo che i programmatori applicativi non debbano preoccuparsi dei dettagli della organizzazione, archiviazione e gestione dei dati. Ciò permette un elevato grado di indipendenza fra le applicazioni e la memorizzazione fisica dei dati.



SQL

- SQL (Structured Query Language) è il linguaggio usato nei sistemi di gestione di basi di dati per:
  - creare, modificare ed eliminare tabelle (Data Definition Language [DDL]):

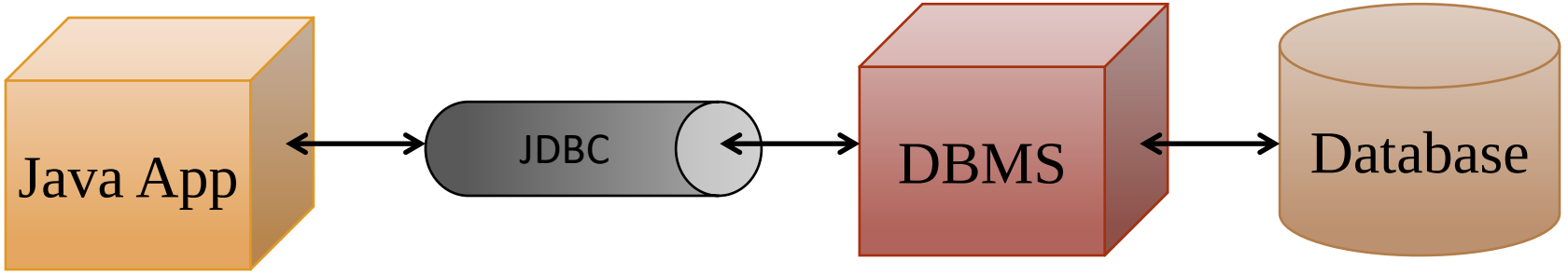
    CREATE TABLE Studenti(Cognome char(30), Nome char(20));
  - inserire, modificare, cancellare dati nelle tabelle (Data Maintenance Language [DML]):

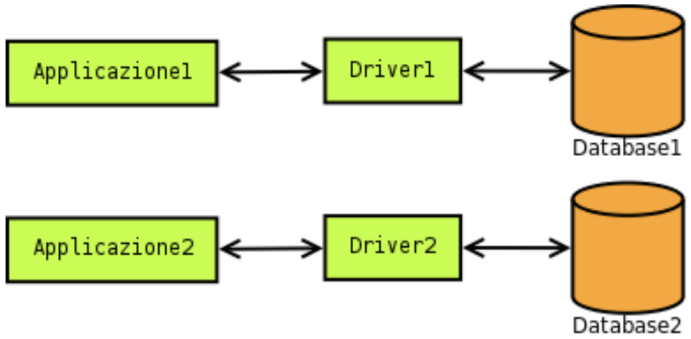
    INSERT INTO Studenti VALUE (‘Bianchi’, ‘Mario’, …);
  - interrogare una base di dati (Data Query Language [DQL]) DQL:

    SELECT * FROM Studenti WHERE Cognome = ‘Bianchi’.
- Essendo un linguaggio dichiarativo, SQL non richiede la stesura di sequenze di operazioni (come ad es. i Linguaggi imperativi), piuttosto di specificare le proprietà logiche delle informazioni risultati dalle istruzioni. Si concentra su cosa il programma deve fare, senza concentrarsi sul come o sullo specifico algoritmo adottato.

JDBC

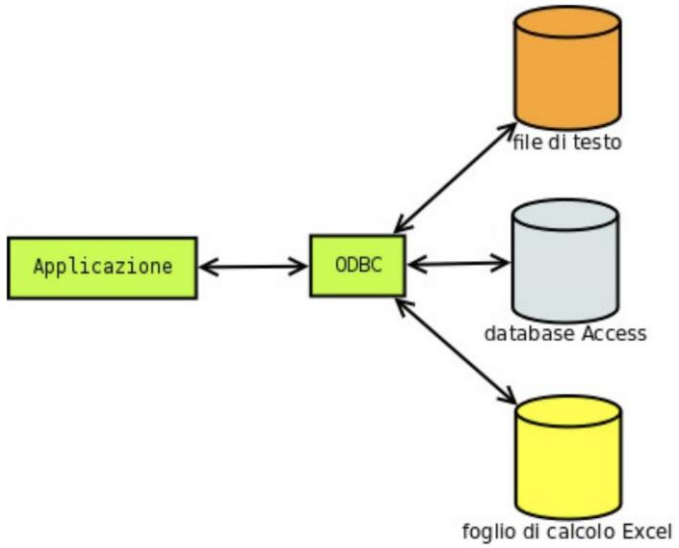
- Java offre la possibilità di interagire con un database attraverso il suo DBMS, inviando interrogazioni in SQL e ricevendone l'esito. Le comunicazioni tra un applicativo Java e un DBMS sono mediate attraverso un driver, una API chiamata "Java Database Connectivity" (JDBC) realizzata come una libreria di classi raggruppate nel package java.sql.



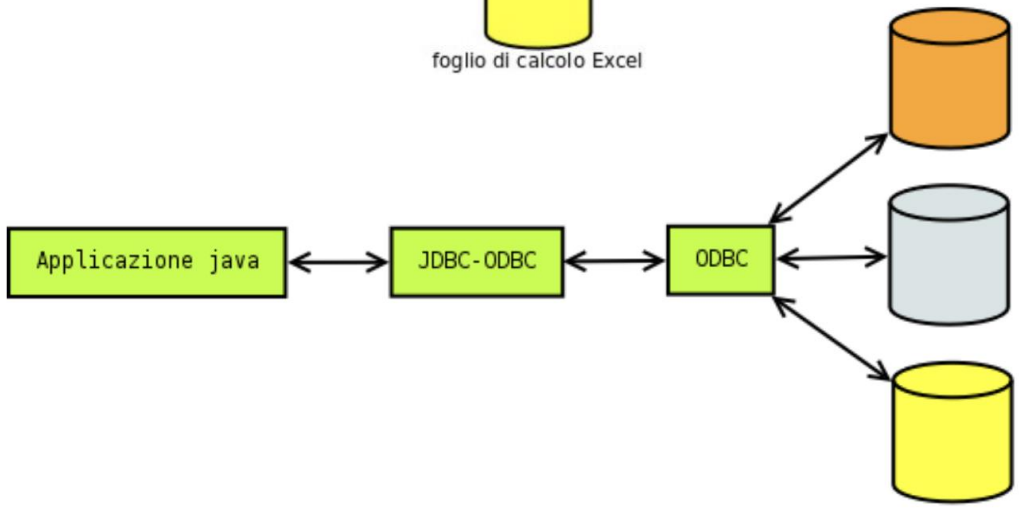


- Il dialogo fra applicazione e DBMS non è mai gestito direttamente ma passa in genere per un opportuno modulo software chiamato driver. Inizialmente, c'era infatti bisogno di un driver specifico, per poter accedere ad ogni diverso database.

JDBC

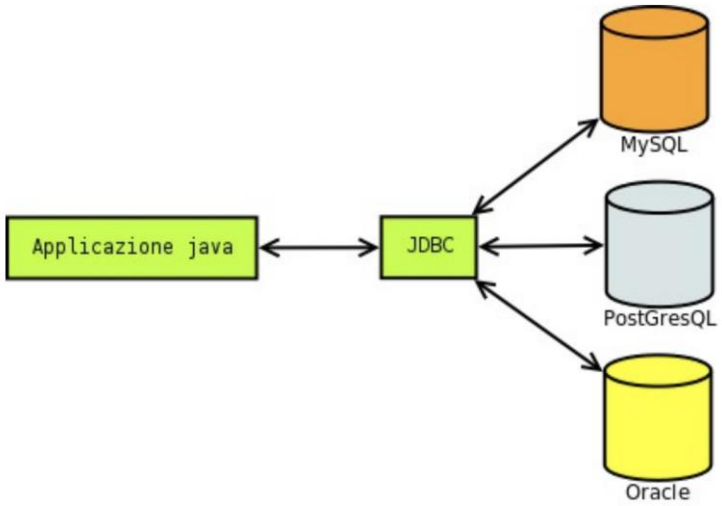


- Nel 1991, Microsoft progettò Open Database Connectivity (ODBC), una API standard per la connessione ai DBMS sviluppata inizialmente su Windows in linguaggio C e consiste di una dll (dynamic link library); altre release sono state scritte per Unix, OS/2 e Macintosh.

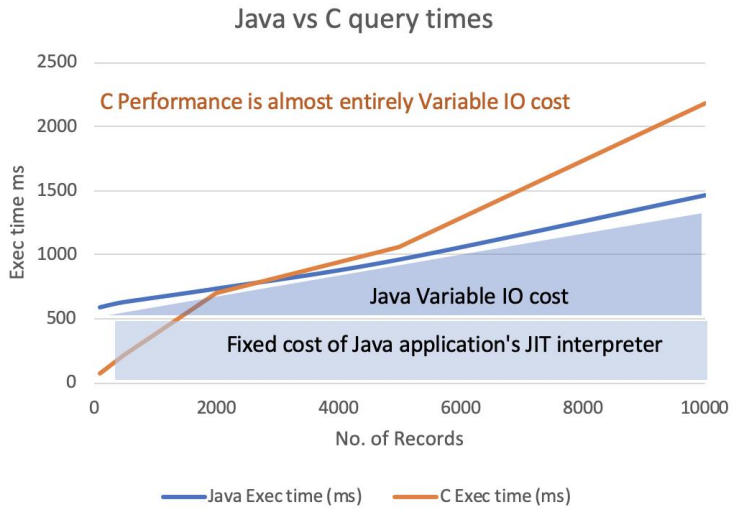


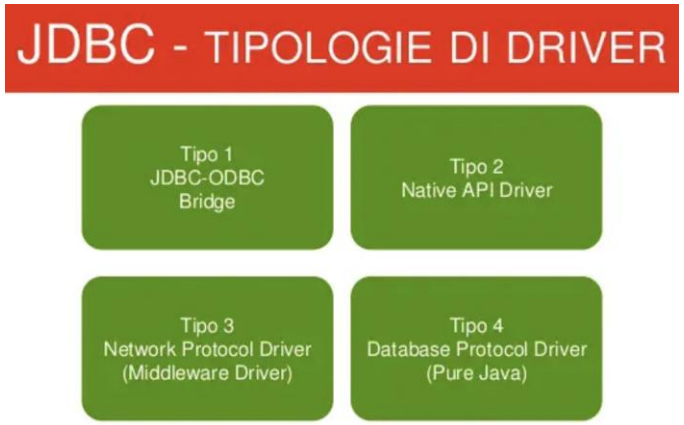
- ODBC è derivato da uno standard noto come X/Open SQL Call Level Interface, e non è direttamente impiegabile in Java.
- Sun Microsystems, in attesa di una soluzione “pure Java 100%”, introdusse una API intermedia tra ODBC e applicazioni denominandola JDBC-ODBC Bridge.

JDBC

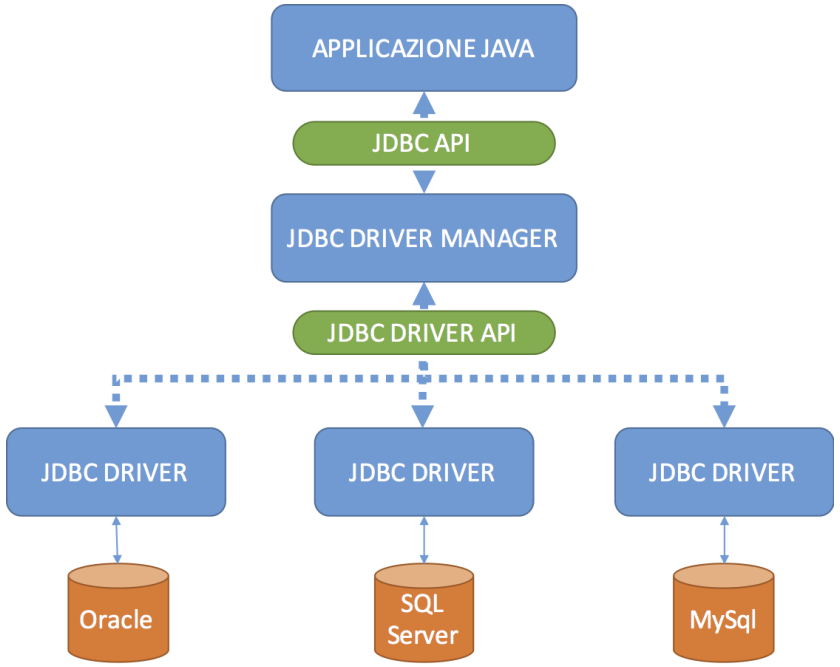


- JDBC è una API per database interamente scritta in java e fornisce metodi e interfacce per interrogare e modificare i dati in maniera Object Oriented. È ancora supportato un bridge JDBC-ODBC, per connettersi a database relazionali che supportano ODBC





JDBC



- Per poter utilizzare JDBC bisogna aver installato un DBMS e il relativo driver JDBC.
- Java offre la classe DriverManager per la gestione dei driver installati. Essa non fornisce un costruttore, ma è una collezione di metodi statici:
  - getDrivers() restituisce una lista di tutti i driver installati;
- getConnection(…) instaura una connessione con un database. Come parametro di ingresso prende l'URL di connessione, e in aggiunta due String che rappresentano l'username e la password per la connessione. Le URL di connessione a un database dipendono dallo specifico DBMS in uso, ma si presentano sempre nella forma:

jdbc:subprotocol:subname

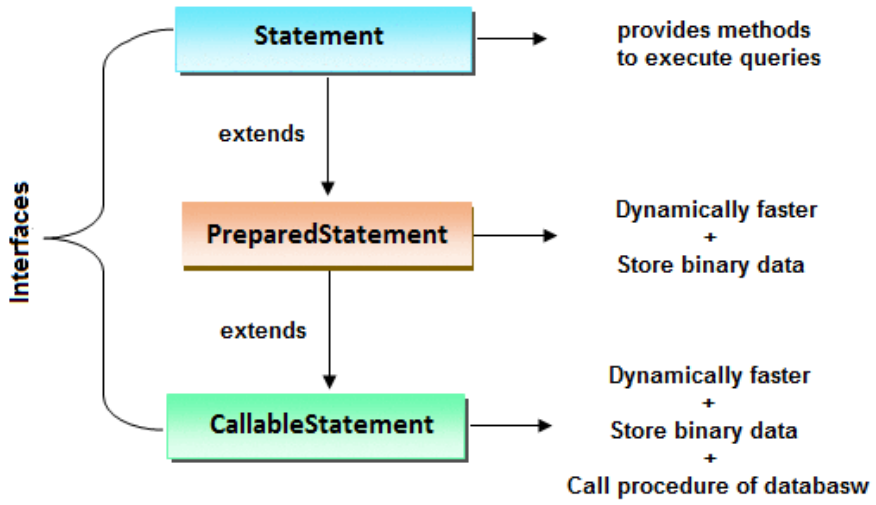
JDBC

- Il metodo getConnection(…) restituisce un oggetto Connection che implementa l'interfaccia con i metodi per interagire con il database. I principali metodi sono:
  - createStatement(): crea un oggetto di tipo Statement, che definisce i seguenti metodi:
  - executeQuery() esegue un'interrogazione SQL di query (DQL), e ritorna un oggetto di tipo ResultSet;
  - execute() esegue un'interrogazione SQL e restituisce un valore true se un oggetto ResultSet è stato creato;
  - executeUpdate() esegue un aggiornamento, cioè di tipo DML, e restituisce il numero di righe aggiornate o 0 se l'operazione non restituisce alcun risultato.
- close(): chiude la connessione.

JDBC

- Degli appositi oggetti sono usati per l'interazione con il database attraverso una data connessione e l'esecuzione di apposite direttive SQL.

- Esistono tre tipi di oggetti statement:



- Statement: esegue semplici direttive SQL senza parametri;
- Prepared Statement: esegue direttive SQL precompilate con o senza parametri;
- Callable Statement: esegue una chiamata a una stored procedure del database.

JDBC

- I risultati di una query sono restituiti come una tabella organizzati per riga e per colonna, oggetto della classe ResultSet, che supporta metodi per l'accesso ai dati.

- ResultSet mantiene un puntatore, o cursore, a una riga dei dati tabellari. All'inizio il cursore punta alla riga precedente alla prima riga della tabella. Il metodo next() serve a far scorrere il cursore, che restituisce true se il cursore si è spostato, o false se si è arrivati all'ultima riga.

- Per accedere alle singole colonne, la classe ResultSet mette a disposizione un insieme di metodi get(). Inoltre ResultSet ha associato un oggetto di tipo ResultSetMetaData, con meta informazioni sullo schema dei dati restituiti.

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection c = DriverManager.getConnection(dbUrl, user, password);
         Statement s = c.createStatement();
         String queryString = “SELECT Nome, Cognome, Email FROM Studenti WHERE (Matricola='" + args[0] + "') “;
         boolean hasResults = s.execute(queryString);
         if(hasResults)
         {
            ResultSet r = s.getResultSet();
            while(r.next()) {
                System.out.println(r.getString(“Nome") + ", " + r.getString(“Cognome") + ": " + r.getString("Email") );
            }
         } else {
            System.out.println(“Nessun risultato);
         }
         s.close();
         c.close();
      } catch(Exception e) { e.printStackTrace();} } }
```

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forN                    odbc.JdbcOdbcDriver");
         Statement s = c.createStatement();
         String queryString = “SELECT Nome, Cognome, Email FROM Studenti WHERE (Matricola='" + args[0] + "') “;
         boolean hasResults = s.execute(queryString);
         if(hasResults)
         {
            ResultSet r = s.getResultSet();
            while(r.next()) {
                System.out.println(r.getString(“Nome") + ", " + r.getString(“Cognome") + ": " + r.getString("Email") );
            }
         } else {
            System.out.println(“Nessun risultato);
         }
         s.close();
         c.close();
      } catch(Exception e) { e.printStackTrace();} } }
```

Definizione dei parametri di connessione con il database.

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection c = DriverManager.getConnection(dbUrl, user, password);
         Statement s = c.createStatement();
         boolean hasResults = s.execute("SELECT * FROM Studenti WHERE (Matricola='" + args[0] + "') ";
         if(hasResults)
         {
            ResultSet r = s.getResultSet();
            while(r.next()) {
                System.out.println(r.getString("Nome") + ", " + r.getString("Cognome") + ": " + r.getString("Email") );
            }
         } else {
            System.out.println("Nessun risultato);
         }
         s.close();
         c.close();
      } catch(Exception e) { e.printStackTrace();} } }
```

Carico il driver per la connessione con il DBMS.

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection c = DriverManager.getConnection(dbUrl, user, password);
         Statement s = c.createStatement();
         String st = “SELECT Nome, Cognome, Email FROM Studenti WHERE (Matricola='" + args[0] + "') “;
         {
            ResultSet r = s.getResultSet();
            while(r.next()) {
                System.out.println(r.getString(“Nome") + ", " + r.getString(“Cognome") + ": " + r.getString("Email") );
            }
         } else {
            System.out.println(“Nessun risultato);
         }
         s.close();
         c.close();
      } catch(Exception e) { e.printStackTrace();} } }
```

Instauro la connessione con il Database.

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection c = DriverManager.getConnection(dbUrl, user, password);
         Statement s = c.createStatement();
         String queryString = “SELECT Nome, Cognome, Email FROM Studenti WHERE (Matricola='" + args[0] + "') “;
         boolean hasResu = s.execute(queryString);
         if(hasResult
         {
            while(r.next()) {
               System.out.println(r.getString(“Nome") + ", " + r.getString(“Cognome") + ": " + r.getString("Email") );
            }
         } else {
            System.out.println(“Nessun risultato);
         }
         s.close();
         c.close();
      } catch(Exception e) { e.printStackTrace();} } }
```

Creo uno Statement SQL e la stringa di interrogazione.

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection c = DriverManager.getConnection(dbUrl, user, password);
         Statement s = c.createStatement();
         String queryString = “SELECT Nome, Cognome, Email FROM Studenti WHERE (Matricola='" + args[0] + "') “;
         boolean hasResults = s.execute(queryString);
         if(hasResults)
         {
            ResultSet r = s.ge
            while(r.ne
                                                                    Email") );
         } else {
            System.out.println(“Nessun risultato);
         }
         s.close();
         c.close();
      } catch(Exception e) { e.printStackTrace();} } }
```

Eseguo l’interrogazione e verifico se è ritornato un risultato, in caso affermativo processo il risultato.

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection c = DriverManager.getConnection(dbUrl, user, password);
         Statement s = c.createStatement();
         String queryString = “SELECT Nome, Cognome, Email FROM Studenti WHERE (Matricola='" + args[0] + "') “;
         boolean hasResults = s.execute(queryString);
         if(hasResults)
         {
            ResultSet r = s.getResultSet();
            while(r.next()) {
                System.out.println(r.getString(“Nome") + ", " + r.getString(“Cognome") + ": " + r.getString("Email") );
            }
         } else {
            System.out.prin                                    to);
         }
```

Ottengo un oggetto ResultSet, scorro la tabella contenuta nell’oggetto riga per riga con next() e accedo alla colonna della riga corrente con getString() passando come parametro il nome della colonna da accedere.

JDBC

```
public class DatabaseApp {
   public static void main (String args[]) {
      String dbUrl = "jdbc:odbc:people"; String user = ""; String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection c = DriverManager.getConnection(dbUrl, user, password);
         Statement s = c.createStatement();
         String queryString = “SELECT Nome, Cognome, Email FROM Studenti WHERE (Matricola='" + args[0] + "') “;
         boolean hasResults = s.execute(queryString);
         if(hasResults)
         {
            ResultSet r = s.getResultSet();
            while(r.next()) {
                                                                    'Email") );
            System                                    risultato);
         }
         s.close();
         c.close();
      } catch(Exception e) { e.printStackTrace();} } }
```

Chiudo lo statement, che chiuderà automaticamente anche l’oggetto ResultSet, e rilascio la connessione con il database.

JDBC vs SQLJ

- La connessione ai database con JDBC è diversa dall'Embedded SQL (SQLJ):
  - Le istruzioni in Java e le interrogazioni SQL non si sovrappongono, ma esiste una netta separazione;
  - In JDBC, non esiste un preprocessore che verifica la validità sintattica e semantica delle istruzioni SQL.

JDBC

```
try {
  String queryString = "SELECT Nome " +
   "FROM Studenti " +
   "WHERE (Matricola='" + args[0] + "')";
  s.execute(queryString);
} catch (SQLException e) {
  System.out.println("Invalid");
  return; }
ResultSet r = s.getResultSet();
r.next();
System.out.println("\nNome="
        +r.getString("Nome"));
```

SQLJ

```
String mat = readEntry(args[o]);
try {
  #sql {select Nome
        into :nome
        from Studenti
        where Matricola = :mat };
} catch (SQLException e) {
  System.out.println("Invalid");
  return;
}
System.out.println("\n Nome= " + nome);
```

Approccio OO e Persistenza

- L’approccio Object-Oriented (OO) è basato sull'uso di linguaggi ad oggetti, e ad oggi è l'approccio più consolidato ed usato per la realizzazione di applicazioni.

- Le metodologie di progettazione OO prevedono l'uso di linguaggi di modellazione concettuale, che offrono un supporto per progettare la logica dell'applicazione e l’ interfaccia utente di un qualsiasi sistema informatico.

- Un sistema informativo che non preserva dati al suo spegnimento è di poca utilità. I linguaggi di programmazione OO non offrono soluzioni “native” al problema della persistenza, ovvero al problema della gestione di dati persistenti da parte di applicazioni OO.

Approccio OO e Persistenza

- I DBMS relazionali sono ad oggi la tecnologia più consolidata, efficiente e comunemente usata per la gestione di grandi quantità di dati.

- Offrono solo qualche ausilio alla programmazione “procedurale” di applicazioni (ES. stored procedure).

- Non sono specifici di alcun paradigma di programmazione, né in generale, di alcuna applicazione Questo principio più noto come data independence, ovvero in generale, i dati vivono più a lungo di qualsiasi applicazione.

Approccio OO e Persistenza

- Necessità di risolvere il problema della persistenza in applicazione OO mediante l'uso di DBMS relazionali.

- Problema: il paradigma OO si è sviluppato su principi propri dell'ingegneria del software, quello relazionale affonda le radici su principi matematici/logici.

- Impedance mismatch denota la mancata corrispondenza tra modello OO e relazionale:
  - Modello OO: descrive il dominio in termini di oggetti e loro proprietà, ovvero attributi o associazioni;
  - Modello relazionale: descrive il dominio in termini relazioni tra valori.

- Far colloquiare l'applicazione che parla in termini di oggetti con un DBMS che parla in termini di valori implica la risoluzione di questo problema.

Differenze Fondamentali

1. Coesione - In un oggetto, tutte le sue proprietà sono contenute nell'oggetto medesimo <> i dati relazionali corrispondenti ad una stessa entità possono essere suddivisi in più tabelle, e.g. a seguito della fase di ristrutturazione dello schema logico;
2. Incapsulamento - In un oggetto, la “business logic" risiede (parzialmente) nei metodi dell'oggetto stesso <> i dati relazionali e la logica che li governa sono implementati in maniera separata;
3. Granularità dei tipi di dato - Tipi di dati composti sono rappresentati mediante apposite classi <> l'SQL non prevede alcun meccanismo standard per la definizione di tipi di dato composti;
4. Ereditarietà e Polimorfismo - l'ereditarietà è implementata attraverso l'uso di super- e sotto-classi, e fornisce inoltre il polimorfismo <> il modello relazionale non ha la possibilità di rappresentare ereditarietà e associazioni polimorfiche: in particolare, l'ereditarietà è “simulata” attraverso la replicazione di dati e l'uso di vincoli di integrità, mentre non esiste il polimorfismo;

Differenze Fondamentali

5. Identità - esistono sia l'identità fisica di oggetti, sia l'identità semantica <> l'identità di una tupla è implementata attraverso l'uso della chiave primaria e non corrisponde direttamente a nessuna delle due nozioni di cui sopra;
6. Navigabilità - Nei linguaggi OO il dominio "si naviga" da un oggetto all'altro, attraverso le associazioni, ovvero secondo la responsabilità delle classi che partecipano all'associazione (Ricorda: a partire da un oggetto si possono accedere gli oggetti con cui partecipa ad una associazione solo se ha responsabilità su di essa) <> l'accesso ai dati relazionali avviene mediante l'uso di join tra tabelle.

Object-Relational Mapping

- La soluzione al problema della persistenza richiede un meccanismo per specificare la corrispondenza tra il dominio dell'applicazione e la base di dati.
- Ciò è chiamato Object-Relational Mapping (ORM) : si identificano le classi di oggetti dell'applicazione di tipo persistente e si fa in modo che i loro attributi e proprietà siano “mappati" su dati memorizzati in una base di dati relazionale.

- Strategie possibili:
  1. Forza bruta;
  2. Codifica manuale di uno strato per l'ORM di oggetti per l'accesso ai dati (Data Access Objects, DAO);
  3. Persistence framework.

ORM: Forza Bruta

- Prevede di equipaggiare le classi dell'applicazione con metodi che interagiscono direttamente con la base di dati, “ovunque” sia necessario.

- È ragionevole quando l'applicazione è sufficientemente semplice, e si può fare a meno di uno strato di incapsulamento (persistence classes) che renda persistente di dati.
- Un oggetto di una classe di business si popola con i dati presenti in un database, procedendo come segue:
  - costruisce da sé lo statement SQL necessario;
  - lo passa al driver;
  - riceve i risultati dal database;
  - li elabora opportunamente.

ORM: Forza Bruta

- Svantaggi:
  - Non è una strategia di incapsulamento;
  - Accoppia fortemente lo strato delle classi di dominio con quelle di controllo al database;
  - Richiede che chi progetta l'applicazione abbia conoscenza dettagliata del database.
- Nelle classi del dominio, si potrebbe avere un metodo per popolare con i dati del DB (Open()), o per rendere persistenti i cambiamenti effettuati (Save()), etc.
- Si collega la logica dell'applicazione al DB, che viola così:
  - interfacciamento esplicito: non è esplicitato il passaggio di dati tra l'applicazione ed il DB;
  - information hiding: non nasconde all'applicazione i dettagli della base dati.

ORM: DAO

- Prevede di realizzare uno strato software atto a gestire la comunicazione tra l'applicazione ed il DBMS. Anche in questo caso il mapping è realizzato manualmente attraverso l'uso di JDBC/SQL. L'accesso al DB viene, però, opportunamente incapsulato nelle classi DAO:
  - nasconde alla logica di business codice JDBC complesso e SQL non-portabile;
  - fornisce un interfacciamento esplicito del codice;
  - migliora la modularità, risolve problemi di accoppiamento tipici dell'approccio forza bruta.

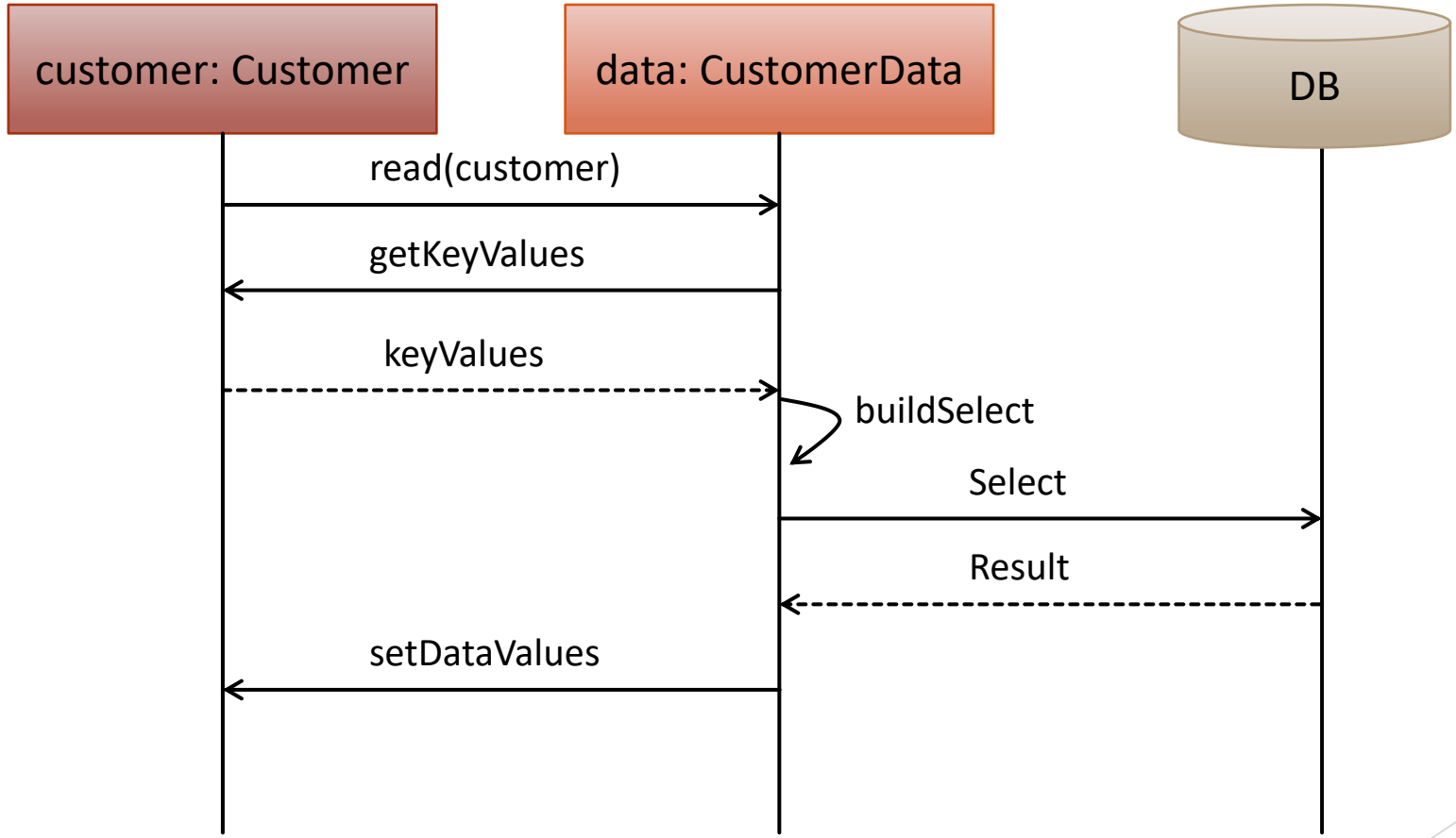
- Un oggetto invoca metodi di una classe DAO che costruisce lo statement SQL e lo passa al driver, riceve i risultati dal database e li inoltra alla classe di business che l'ha interrogata.

ORM: DAO

- Prevede di realizzare uno strato software atto a gestire la comunicazione tra l'applicazione ed il DBMS. Anche in questo caso il mapping è realizzato manualmente attraverso l'uso di JDBC/SQL. L'accesso al DB viene, però, opportunamente incapsulato nelle classi DAO:
  - nasconde alla logica di business codice JDBC complesso e SQL non-portabile;
  - fornisce un interfacciamento esplicito del codice;
  - migliora la modularità, risolve problemi di accoppiamento tipici dell'approccio forza bruta.

- Un oggetto invoca metodi di una classe DAO che costruisce lo statement SQL e lo passa al driver, riceve i risultati dal database e li inoltra alla classe di business che l'ha interrogata.

ORM: DAO


customer: Customer
data: CustomerData
DB
read(customer)
getKeyValues
keyValues
buildSelect
Select
Result
setDataValues

ORM: DAO

- Tutta la logica di accesso al DB è completamente incapsulata nelle classi DAO:
  - Cambiamenti del DB influenzano solo le DAO;
  - La classe CustomerData si fa carico di gestire il codice SQL, mentre tutto ciò è trasparente rispetto alla classe Customer, la quale invoca metodi indipendenti dal DB;

- L'approccio tipico è quello di avere un DAO per ciascuna classe di dominio.

- Svantaggi:
  - Il programmatore è vincolato a sviluppare interrogazioni SQL e di realizzare le classi DAO.

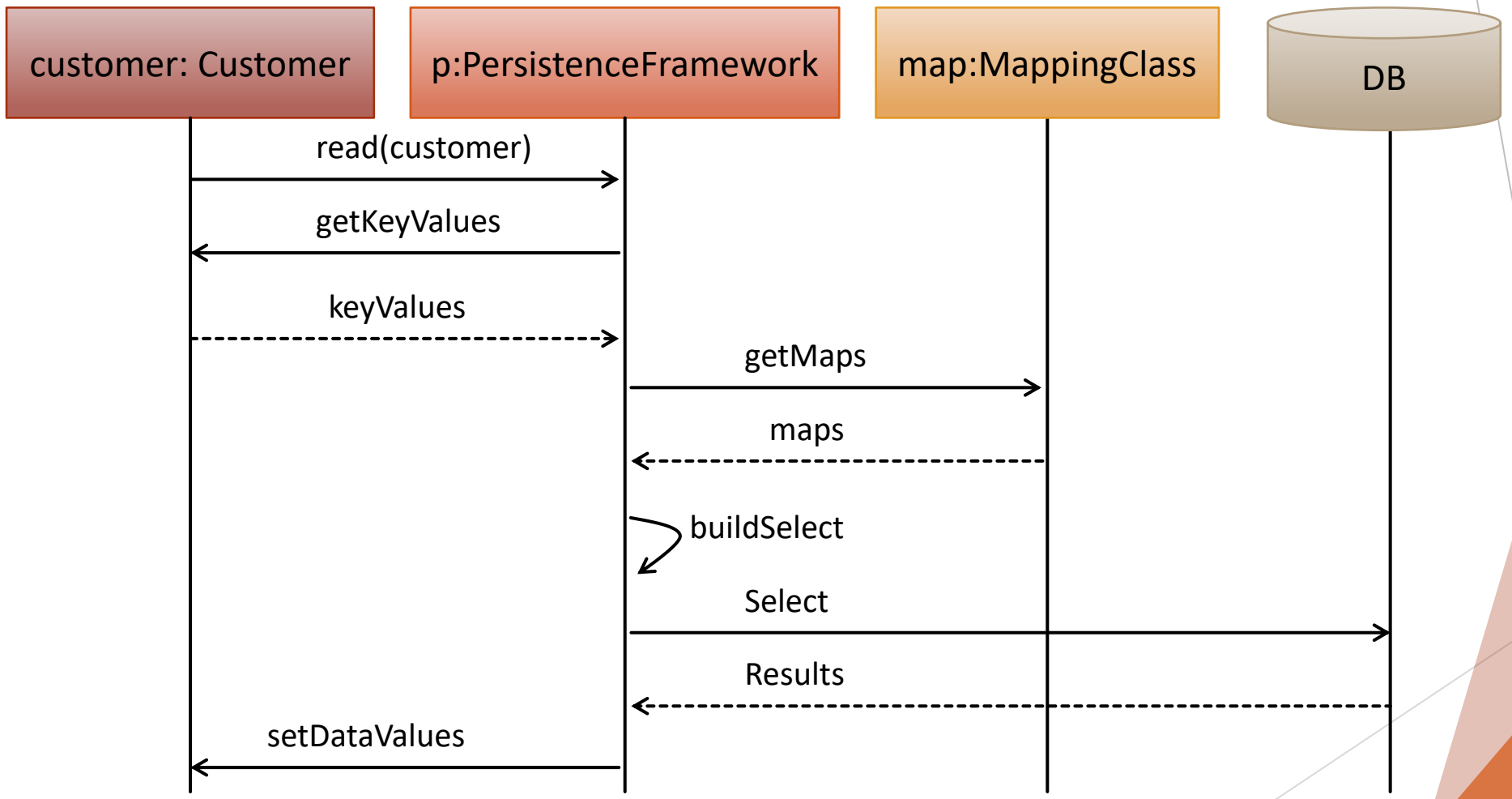
ORM: Persistence Framework

- Prevede l'utilizzo di un framework predefinito per la gestione della persistenza con l'obiettivo di liberare il programmatore quanto più possibile dalla necessità di scrivere codice SQL nella sua applicazione. Infatti, il programmatore vede il DB solo quando configura il framework.

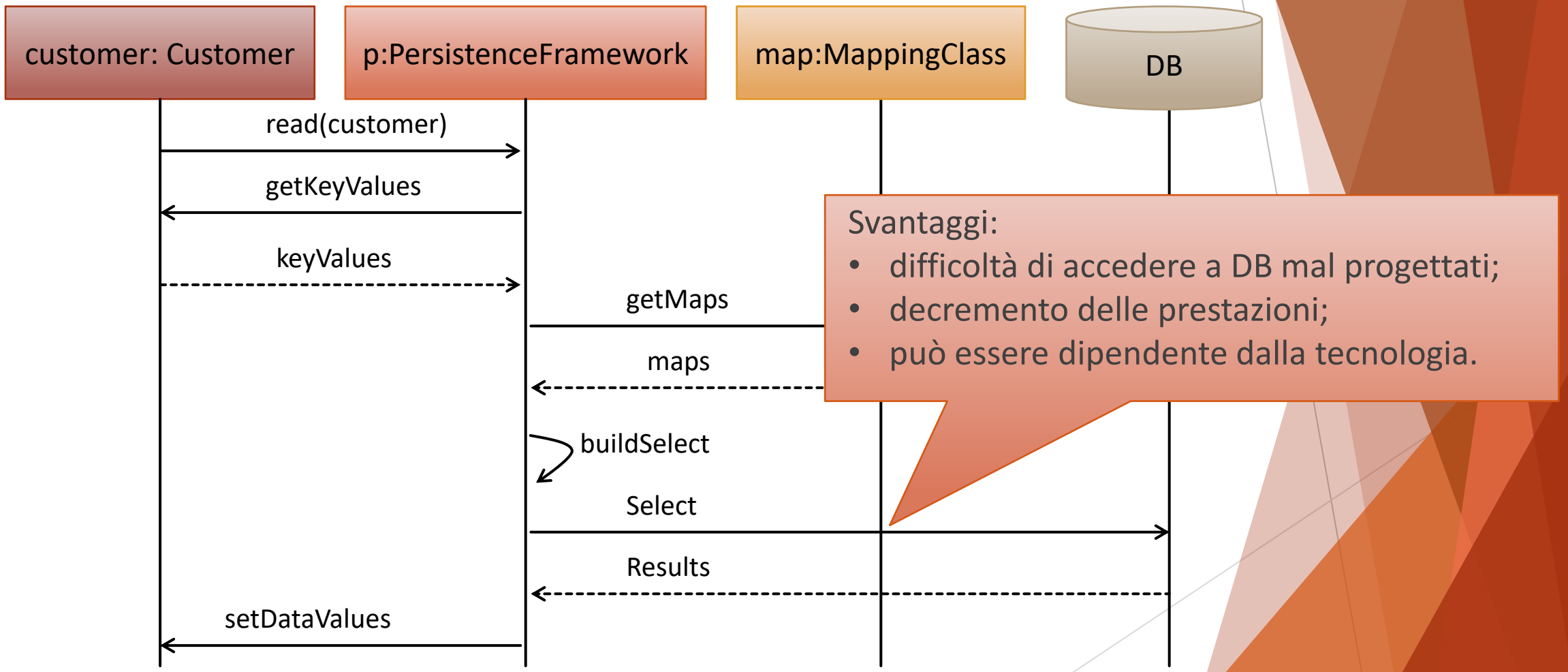
- Un persistence framework incapsula pienamente la logica di accesso al DB. Mentre, dei meta-dati rappresentano la corrispondenza tra domain classes e tabelle, nonché le associazioni tra le domain classes.

- Offrono funzionalita di base (CRUD) e altre:
  - transazioni;
  - gestione della concorrenza;
  - caching.

ORM: Persistence Framework

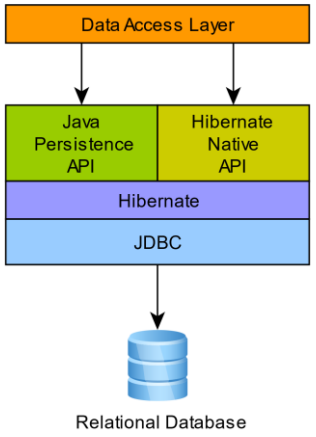


ORM: Persistence Framework



Hibernate Core

- Java prevede uno standard per la gestione della persistenza per mezzo di un framework ORM, denominato Java Persistence API, talvolta riferite come JPA. offre delle API per aiutare gli sviluppatori nelle operazioni di persistenza dei dati su un database relazionale. In particolare:
  - fornisce una mappatura tra classi Java e tabelle del database
  - fornisce un linguaggio per effettuare query SQL, chiamato JPQL (Java Persistence Query Language), che è indipendente dalla DBMS utilizzato
  - fornisce varie API per la gestione e manipolazioni degli oggetti Java che mappano le tabelle del database.
- In quanto API, JPA in realtà offre delle interfacce, implementate da vari provider tra cui il più famoso è Hibernate.



Hibernate Core

- Il mapping tra classi e tabelle sono specificati in appositi file XML, e lo sviluppatore è esonerato da:
  - richieste di esecuzione di chiamate SQL;
  - gestione manuale dei risultati di chiamate SQL e loro eventuale conversione in oggetti.
- L'applicazione rimane portabile in tutti i sistemi di gestione supportati, con pochissimo overhead. Può essere usato in maniera independente dal tipo di applicazione, di piattaforma e di ambiente runtime.
- I suoi approcci di utilizzo tipici sono:
  - Top down
  - Bottom up
  - Meet in the middle

Hibernate Core

Top down

- Si parte dalla specifica delle classi, o diagramma delle classi di dominio, e dalla sua implementazione in Java e si ha completa libertà rispetto allo schema della base di dati.
- Si specificano i file XML con i mapping ed eventualmente, si può usare Hibernate (in particolare lo strumento Hibernate hbm2ddl) per generare lo schema della base di dati.
  - Vantaggi: comodo quando non vi è alcun database preesistente.
  - Svantaggi: diventa laborioso se si vogliono sfruttare tutte le caratteristiche del DBMS, e.g. stored procedure, trigger.

Hibernate Core

Bottom up

- Si parte da un base di dati esistente e si ha completa libertà rispetto al dominio dell'applicazione.
- Si usano una serie di strumenti Hibernate per generare lo schema di base del codice Java per la gestione della persistenza, ovvero il codice delle classi persistenti:
  - jdbcconfiguration: si connette via JDBC ed accede ai metadati dal catalogo della base di dati;
  - hbm2hbmxml: strumento che prende in ingresso i metadati ottenuti dal catalogo e genera i file XML dei mapping;
  - hbm2cfgxml: strumento che prende in ingresso i metadati ottenuti dal catalogo e genera il file di configurazione di Hibernate.

Hibernate Core

Bottom up

- Problemi:
  - lo schema della base di dati deriva, nella migliore delle ipotesi, da un "degrado" dello schema ER ⇒ non si potrà mai ottenere un diagramma delle classi che sfrutti caratteristiche importanti della programmazione ad oggetti, e.g. ereditarietà, polimorfismo;
  - la logica dell'applicazione è influenzata dalla rappresentazione dei dati ⇒ il diagramma delle classi di dominio conterrà solamente le classi persistenti.

Hibernate Core

Meet in the middle

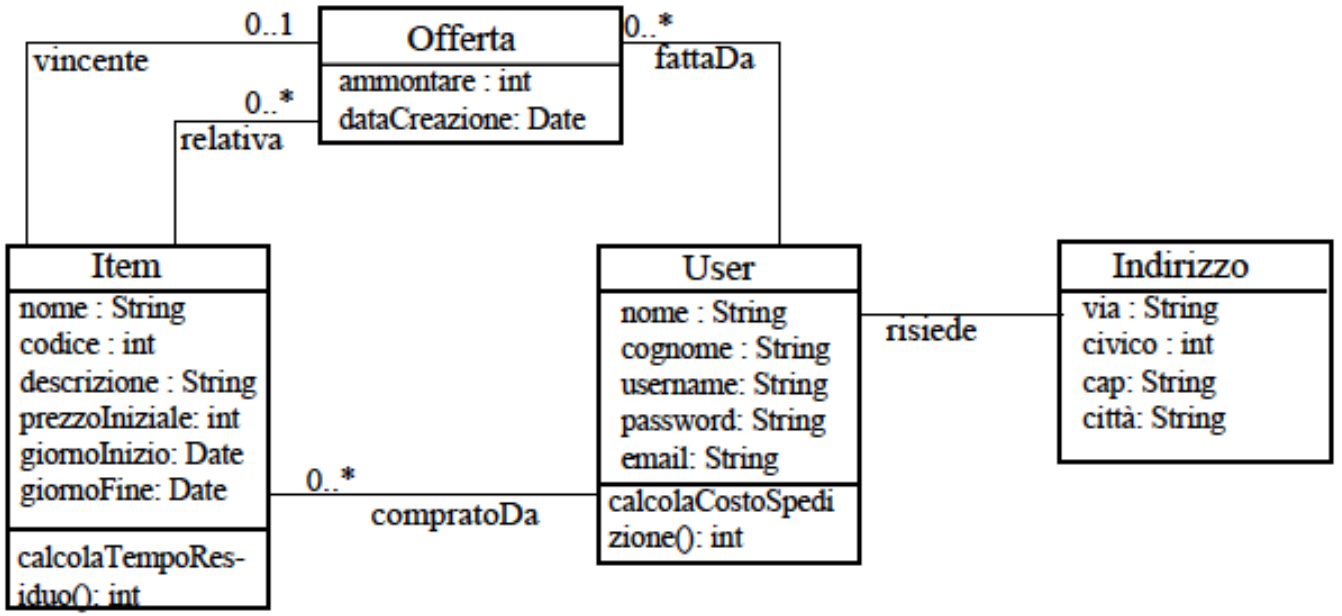
- Si parte da un diagramma delle classi di dominio dell'applicazione e da una base di dati preesistente .
- Non è in generale possibile mappare un insieme di classi di dominio su di una base di dati arbitrari.
- Si deve seguire un approccio generico all'ORM e ricorrere ad Hibernate quando possibile, ed in caso contrario, all'approccio generico DAO

Hibernate in Action

- 5 principali ingredienti di un'applicazione che fa uso di Hibernate per la gestione della persistenza:
  - Le classi di dominio realizzate in Java;
  - Una base di dati, per es. realizzata in Mysql;
  - Un file che definisce il mapping di ogni classe persistente;
  - Uno o più file di configurazione di Hibernate
  - Le interfacce Hibernate per l'accesso alla base di dati: Session, Transaction e Query – package org.hibernate.

Hibernate in Action

- Le classi di dominio sono definire come in qualsiasi applicazione Java:
  - Tipicamente, metodi set e get per l'accesso in scrittura e lettura delle proprietà degli oggetti della classe;
  - Metodi di business della classe.

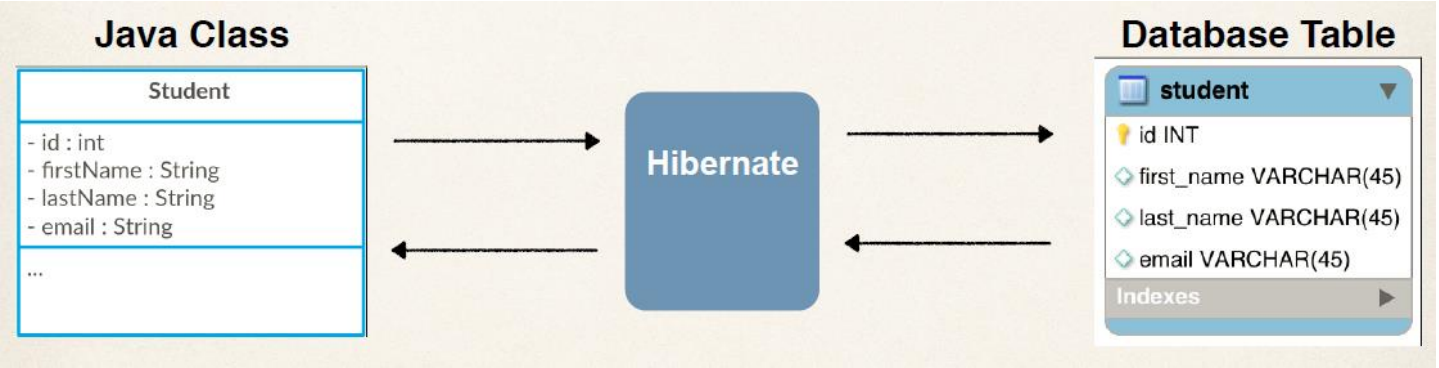


Hibernate in Action

- La base di dati è costituita da un insieme di:
  - Tabelle, Vincoli, Stored procedures e triggers.
- È contenuta all'interno di un certo DBMS accessibile per mezzo di JDBC da un determinato utente (caratterizzato da username e password).

```
CREATE TABLE utente (
username VARCHAR(10) PRIMARY KEY,
nome VARCHAR(15),
cognome VARCHAR(15),
cod_fis VARCHAR(16) UNIQUE,
password VARCHAR(8),
email VARCHAR(20),
via VARCHAR(20),
civico INT,
cap VARCHAR(5),
citta VARCHAR(15));
```

Hibernate in Action



- Il mapping è un file XML che definisce come si mappano le proprietà delle classi Java persistenti sulle tabelle della base di dati.
- Deve soddisfare la grammatica specificata all'interno di un apposito DTD, chiamato hibernate-mapping-3.0.dtd.
- Per verificare la correttezza sintattica del file di mapping, Hibernate cercherà il DTD all'interno del classpath e lo troverà nella libreria .jar di Hibernate. Qualora non lo trovasse, Hibernate lo cercherà all'indirizzo specificato nella dichiarazione del DOCTYPE.

Hibernate in Action

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="User" table="UTENTE">
    <id name="username" column="username"></id>
    <property name="nome" column="NOME"></property>
    <property name="cognome" column="COGNOME"></property>
    <property name="email"></property>
    <property name="password"></property>
  </class>
</hibernate-mapping>
```

Hibernate in Action

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hi
   <
     <id name="username" column="username"></id>
     <property name="nome" column="NOME"></property>
     <property name="cognome" column="COGNOME"></property>
     <property name="email"></property>
     <property name="password"></property>
   </class>
</hibernate-mapping>
```

Specifica del DTD per la verifica del file di Mapping.

Hibernate in Action

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="User" table="UTENTE">
     <id name="user          lumn="username"></id>
     <property name="email"></property>
     <property name="password"></property>
   </class>
</hibernate-mapping>
```

Dichiara il nome della classe persistente e della tabella relativa. Ogni istanza della classe è rappresentata da una tubla della tabella.

Hibernate in Action

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="User" table="UTENTE">
    <id name="username" column="username"></id>
    <property name="pa          mn="NOME"></property>
    <pr
    <pr
    <property name="password"></property>
  </class>
</hibernate-mapping>
```

Rappresenta la proprietà della classe che è caratterizzabile (non modificabile dell'utente) e mappata sulla chiave primaria della tabella.

Hibernate in Action

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="User" table="UTENTE">
    <id name="username" column="username"></id>
    <property name="nome" column="NOME"></property>
    <property name="cognome" column="COGNOME"></property>
    <property name="email"></property>
    <property name="password"></property>
  </class>
</hiber
```

Rappresenta la proprità della classe che è caratterizzabile (non modificabile dell'utente) e mappata sulla chiave primaria della tabella.

Hibernate in Action

- Hibernate costituisce la parte dell'applicazione che gestisce la persistenza, ovvero che si connette alla base di dati:
  - Ha bisogno di avere le informazioni necessarie ad effettuare la connessione, quali il DBMS, il driver JDBC, la base di dati, utente/password, etc.
- Un file XML fornisce tutte queste informazioni e deve soddisfare la grammatica specificata nel DTD hibernate-configurazion-3.0.dtd.

Hibernate in Action

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
<!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">
         com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">
         jdbc:mysql://localhost/bid</property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>
<!-- SQL dialect -->
```

Hibernate in Action

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
<!-- Database con        settings -->
        <property name="hibernate.connection.url">
         jdbc:mysql://localhost/bid</property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>
<!-- SQL dialect -->
```

Definisce le informazioni per l'accesso ad una particolare base di dati.

Hibernate in Action

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">
         com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">
         jdbc:mysql://localhost/bid</property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>
<!-- SQL dialect -->
```

Le proprietà hibernate.connection.* definiscono i parametri per la connessione ad un dato database.

Hibernate in Action

```
        <property name="dialect">
         org.hibernate.dialect.MySQL5InnoDBDialect</property>
<!-- JDBC connection pool (use C3P0) -->
        <property name="c3p0.min_size">5</property>
        <property name="c3p0.max_size">20</property>
        <property name="c3p0.timeout">300</property>
        <property name="c3p0.max_statement">50</property>
<!-- Show and print nice SQL on stdout -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>
<!--  List of XML mapping files -->
        <mapping resource="bid/User.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Hibernate in Action

```
        <property name="dialect">
         org.hibernate.dialect.MySQL5InnoDBDialect</property>
<!-- JDBC connection                    C3P0) -->
        <p
        <property name="c3p0.max_size">20</property>
        <property name="c3p0.timeout">300</property>
        <property name="c3p0.max_statement">50</property>
<!-- Show and print nice SQL on stdout -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>
<!--  List of XML mapping files -->
        <mapping resource="bid/User.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Specifica quale variante del linguaggio SQL il DBMS impiegato fa uso.

Hibernate in Action

```
        <property name="dialect">
         org.hibernate.dialect.MySQL5InnoDBDialect</property>
<!-- JDBC connection pool (use C3P0) -->
        <property name="c3p0.min_size">5</property>
        <property name="c3p0.max_size">20</property>
        <property name="c3p0.timeout">300</property>
        <property name="c3p0.max_statement">50</property>
<!-- Show and print
        <p
        <property name=
<!--  List of XML mapping files -->
        <mapping resource="bid/User.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Specifica i parametri per il pooling di connessioni.

Hibernate in Action

```
        <property name="dialect">
         org.hibernate.dialect.MySQL5InnoDBDialect</property>
<!-- JDBC connection pool (use C3P0) -->
        <property name="c3p0.min_size">5</property>
        <property name="c3p0.max_size">20</property>
        <property name="c3p0.timeout">300</property>
        <property name="c3p0.max_statement">50</property>
<!-- Show and print nice SQL on stdout -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>
<!--  List of XML mapping files -->
        <mapping resource="bid/User.hbm.xml" />
    </session-factor
</hibernat
```

Specifica il file di mapping da usare per identificare la relazione tra oggetti e tabelle.

Hibernate in Action

- Hibernate fornisce tre principali interfacce per l'accesso alla base di dati, tutte appartenenti a org.hibernate:
  - Session, ogni istanza rappresenta una sessione di comunicazione tra applicazione e base di dati. Contiene i metodi per caricare/salvare oggetti nella/dalla base di dati;
  - Transaction, ogni istanza rappresenta una transazione, e fornisce un alto disaccoppiamento perché per definire una transazione non si deve far ricorso all'API di JDBC;
  - Query, permette di creare ed eseguire query sia nel linguaggio di Query di Hibernate (HQL), che in SQL.

Hibernate in Action

```
import java.util.*;
import org.hibernate.*;
import persistence.HibernateUtil;

public class Bid {
public static void main(String[] args) {

//First unit of work
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
User user = new User("pippo");
String userId = (String) session.save(user);
tx.commit();
session.close();
```

Hibernate in Action

```
import java.util.*;
import org.hibernate.*;
import persistence.HibernateUtil;

public class Bid {
public static void main(String[] args) {

//First unit of work
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Use
Str
tx.commit();
session.close();
```

L'istanza di sessione è ottenibile per mezzo di un apposito metodo factory, sulla classe Factiry di Session.

Hibernate in Action

```
import java.util.*;
import org.hibernate.*;
import persistence.HibernateUtil;

public class Bid {
public static void main(String[] args) {

//First unit of work
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
User user = new User("pippo");
String userId = (String) session.save(user);
tx.commit();
ses
```

Un'istanza di Transaction è ottenibile da un apposito metodo factory su un'istanza di Session. Si realizza il salvataggio di un oggetto sul database.

Hibernate in Action

```
import java.util.*;
import org.hibernate.*;
import persistence.HibernateUtil;

public class Bid {
public static void main(String[] args) {

//First unit of
Session session
Transaction tx =
User user = new
String userId =
tx.commit();
ses
```

Lo statement SQL per il salvataggio è generato da Hibernate:

```
insert into UTENTE(USERNAME,NOME,COGNOME,COD_FIS,
PASSWORD,EMAIL,VIA,CIVICO,CAP,CITTA)
values ('pippo', null, null, null,
null, null, null, null, null, null);
```

Un'istanza di Transaction è ottenibile da un apposito metodo factory su un'istanza di Session. Si realizza il salvataggio di un oggetto sul database.

Hibernate in Action

```
import java.util.*;
import org.hibernate.*;
import persistence.HibernateUtil;

public class Bid {
public static void main(String[] args) {

//First unit of work
Sessio
Transa
User u
String userId                    session.save(user);
tx.commit();
session.close();
```

Di default, Transaction ha l'autocommit settato a false, quindi bisogna invocare commit per approvare la transazione.

Hibernate in Action

```
//Second unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction();
user = (User) session.get(User.class,userId);
user.setNome("Filippo");
tx.commit();
session.close();


//Third unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction()
List users = session.createSQLQuery("select * from utente
order by username").addEntity(User.class).list();
System.out.println(users.size()+" user(s) found: ");
for (Iterator iter= users.iterator(); iter.hasNext(); ) {
User userId = (User) iter.next();
System.out.println (userId.getNome());
}
tx.commit();
```

Hibernate in Action

```
//Second unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction();
user = (User) session.get(User.class,userId);
user.setNome("Fili
```

Ottenere dalla base di dati un'istanza della classe persistente identificata dal un certo User Id; se non trovata restituisce null.

```
//Third unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction()
List users = session.createSQLQuery("select * from utente
order by username").addEntity(User.class).list();
System.out.println(users.size()+" user(s) found: ");
for (Iterator iter= users.iterator(); iter.hasNext(); ) {
User userId = (User) iter.next();
System.out.println (userId.getNome());
}
tx.commit();
```

Hibernate in Action

```
//Second unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction();
user = (User) session.get(User.class,userId);
user.setNome("Filippo");
tx.commit();
```

Realizza il dirty checking automatico: Hibernate si accorge di eventuali modifiche agli oggetti e aggiorna la base di dati.

```
//Third unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction()
List users = session.createSQLQuery("select * from utente
order by username").addEntity(User.class).list();
System.out.println(users.size()+" user(s) found: ");
for (Iterator iter= users.iterator(); iter.hasNext(); ) {
User userId = (User) iter.next();
System.out.println (userId.getNome());
}
tx.commit();
```

Hibernate in Action

```
//Second unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction();
user = (User) session.get(User.class,userId);
user.setNome("Filippo");
tx.commit();
session.close();
```

Query SQL applicata alla base di dati (approccio DAO) per ottenere una lista di oggetti di una data classe di dominio.

```
sionFactory().openSession();
tx =  .beginTransaction()
List users = session.createSQLQuery("select * from utente
order by username").addEntity(User.class).list();
System.out.println(users.size()+" user(s) found: ");
for (Iterator iter= users.iterator(); iter.hasNext(); ) {
User userId = (User) iter.next();
System.out.println (userId.getNome());
}
tx.commit();
```

Hibernate in Action

```
session.close();

//Shutting down the application
HibernateUtil.shutdown();
}

}
```

```
//Second unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction();
user = (User) session.get(User.class,userId);
user.setNome("Filippo");
tx.commit();
session.close();


//Third unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction()
List users = session.createSQLQuery("select * from utente
order by username").addEntity(User.class).list();
System.out.println(users.size()+" user(s) found: ");
for (Iterator iter= users.iterator(); iter.hasNext(); ) {
User userId = (User) iter.next();
System.out.println (userId.getNome());
}
tx.commit();
```

Hibernate in Action

```
session.close();

//Shutting down the application
HibernateUtil.shutdown();
}

}
```

```
//Second unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction();
user = (User) session.get(User.class,userId);
user.setNome("Filippo");
tx.commit();
session.close();


//Third unit of work
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction()
List users = session.createSQLQuery("select * from utente
order by username").addEntity(User.class).list();
ystem.out.println(users.size()+" user(s) found: ");
(Iterator iter= users.iterator(); iter.hasNext(); ) {
rId = (User) iter.next();
println (userId.getNome());
```

Chiudere sempre le sessioni aperte, e terminare la piattaforma Hibernate.

Hibernate in Action

- Le query sono una parte cruciale e onerosa. Hibernate offre un mezzo potente per esprimere quasi tutto quello che comunemente serve di esprimere in SQL, ma in termini object-oriented – usando classi e proprietà.

- Esistono tre modi diversi di esprimere query:
  - HQL : session.createQuery("from Category c where c.name like 'Laptop%'");
  - Query by criteria (QBC) e Query by example (QBE) : session.createCriteria(Category.class).add(Expression.like("name", "Laptop%"));
  - SQL con un mapping automatico degli insiemi dei risultati su oggetti: session.createSQLQuery("select {c.*} from CATEGORY {c} where NAME like 'Laptop%'", "c", Category.class);

Hibernate in Action

- Le annotazioni di Hibernate sono il modo più nuovo per definire le mappature senza l'uso del file XML. È possibile utilizzare le annotazioni in aggiunta o in sostituzione dei metadati di mappatura XML.
- Le annotazioni vengono inseriti nel file della classe Java le cui istanze si vogliono rendere persistenti, e questo aiuta l'utente a comprendere la struttura della tabella e della classe contemporaneamente durante lo sviluppo.

```
create table EMPLOYEE ( id INT NOT NULL
auto_increment, first_name VARCHAR(20) default NULL,
last_name VARCHAR(20) default NULL, salary INT default
NULL, PRIMARY KEY (id) );
```

Hibernate in Action

```
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;
   @Column(name = "first_name")
   private String firstName;
   @Column(name = "last_name")
   private String lastName;
   @Column(name = "salary")
   private int salary;
   public Employee() {}

   …
}
```

Hibernate in Action

```
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;
   @Column(name = "first_name")
   private String firstNar
   @Column(name = "la
   private String lastNan
   @Column(name = "sa
   private int salary;
   public Employee() {}

   ...
}
```

@Entity contrassegna questa classe persistente, quindi deve avere un costruttore senza argomenti visibile con almeno un ambito protetto.

@Table consente di specificare i dettagli della tabella che verrà utilizzata per rendere persistente l'entità nel database.

Hibernate in Action

```
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
  @Id @GeneratedValue
  @Column(name = "id")
  private int id;
  @Column(name = "first_name")
  private String firstNar
  @Column(name = "la
  private String lastNan
  @Column(name = "salary")
  private int salary;
  public Employee() {}

  ...
}
```

L'annotazione @Id determinerà automaticamente la strategia di generazione della chiave primaria più appropriata da utilizzare, ma è possibile sovrascriverla applicando l'annotazione @GeneratedValue.

Hibernate in Action

```
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
  @Id @GeneratedValue
  @Column(name = "id")
  private int id;
  @Column(name = "first_name")
  private String firstNar
  @Column(name = "la
  private String lastName;
  @Column(name = "salary")
  private int salary;
  public Employee() {}

  ...
}
```

L'annotazione @Column viene utilizzata per specificare i dettagli della colonna a cui verrà mappato un campo o una proprietà.

Hibernate in Action

```
public class ManageEmployee {
  private static SessionFactory factory;
  public static void main(String[] args) {

    try {
      factory = new AnnotationConfiguration().configure("/cfg.xml").
            addAnnotatedClass(Employee.class).buildSessionFactory();
    } catch (Throwable ex) {
      System.err.println("Failed to create sessionFactory object." + ex);
      throw new ExceptionInInitializerError(ex);
    }

    ManageEmployee ME = new ManageEmployee();

    ...
```

Hibernate in Action

```
public class ManageEmployee {
  private static SessionFactory factory;
  public static void main(String[] args) {

    try {
      factory = new AnnotationConfiguration().configure("/cfg.xml").
          addAnnotatedClass(Employee.class).buildSessionFactory();
    } catch (Throwable ex) {
      System.err.println("Failed to c      sionFactory object." + ex);
      throw new ExceptionInInitializer
    }

    ManageEmployee ME =

    ...
```

Come parte della configurazione si passa il nome del file xml con la specifica dei dettagli di connessione al database, e anche la classe annotata che ne rappresenta il mapping con le tabelle nel database.

Hibernate in Action

```
...
   /* Add few employee records in database */
   Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
   Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
   Integer empID3 = ME.addEmployee("John", "Paul", 10000);

   /* List down all the employees */
   ME.listEmployees();

   /* Update employee's records */
   ME.updateEmployee(empID1, 5000);
   /* Delete an employee from the database */
   ME.deleteEmployee(empID2);

   /* List down new list of the employees */
   ME.listEmployees();
}
```

Hibernate in Action

```
/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
   Session session = factory.openSession();
   Transaction tx = null;   Integer employeeID = null;
   try {
      tx = session.beginTransaction();  Employee employee = new Employee();
      employee.setFirstName(fname);  employee.setLastName(lname);
      employee.setSalary(salary);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   } finally {
      session.close();       }
   return employeeID;
}
```

Hibernate in Action

```
  /* Method to  READ all the employees */
public void listEmployees( ){
   Session session = factory.openSession();      Transaction tx = null;
   try {
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator = employees.iterator(); iterator.hasNext();){
         Employee employee = (Employee) iterator.next();
         System.out.print("First Name: " + employee.getFirstName());
         System.out.print("  Last Name: " + employee.getLastName());
         System.out.println("  Salary: " + employee.getSalary());         }
      tx.commit();
   } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   } finally {
      session.close();   }   }
```

Hibernate in Action

```
  /* Method to  READ all the employees */
public void listEmployees( ){
   Session session = factory.openSession();      Transaction tx = null;
   try {
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator = employees.iterator(); iterator.hasNext();){
         Employee employee = (Employee) iterator.next();
         System.out.print("First Name: " + employee.getFirstName());
         System.out.print("  Last Name: " + employee.getLastName());
         System.out.println("  Salary: " + employee.getSalary());
      }
      tx.commit();
   } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   } finally {
```

Hibernate in Action

```
   /* Method to UPDATE salary for an employee */
  public void updateEmployee(Integer EmployeeID, int salary ){
    Session session = factory.openSession();    Transaction tx = null;
          try {
       tx = session.beginTransaction();
       Employee employee = (Employee)session.get(Employee.class,
EmployeeID);
       employee.setSalary( salary );
              session.update(employee);
       tx.commit();
    } catch (HibernateException e) {
       if (tx!=null) tx.rollback();
       e.printStackTrace();
    } finally {
       session.close();
    }
  }
```

Hibernate in Action

```
 /* Method to DELETE an employee from the records */
  public void deleteEmployee(Integer EmployeeID){
    Session session = factory.openSession();   Transaction tx = null;

    try {
      tx = session.beginTransaction();
      Employee employee = (Employee)session.get(Employee.class,
EmployeeID);
      session.delete(employee);          tx.commit();
    } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }
  }
}
```