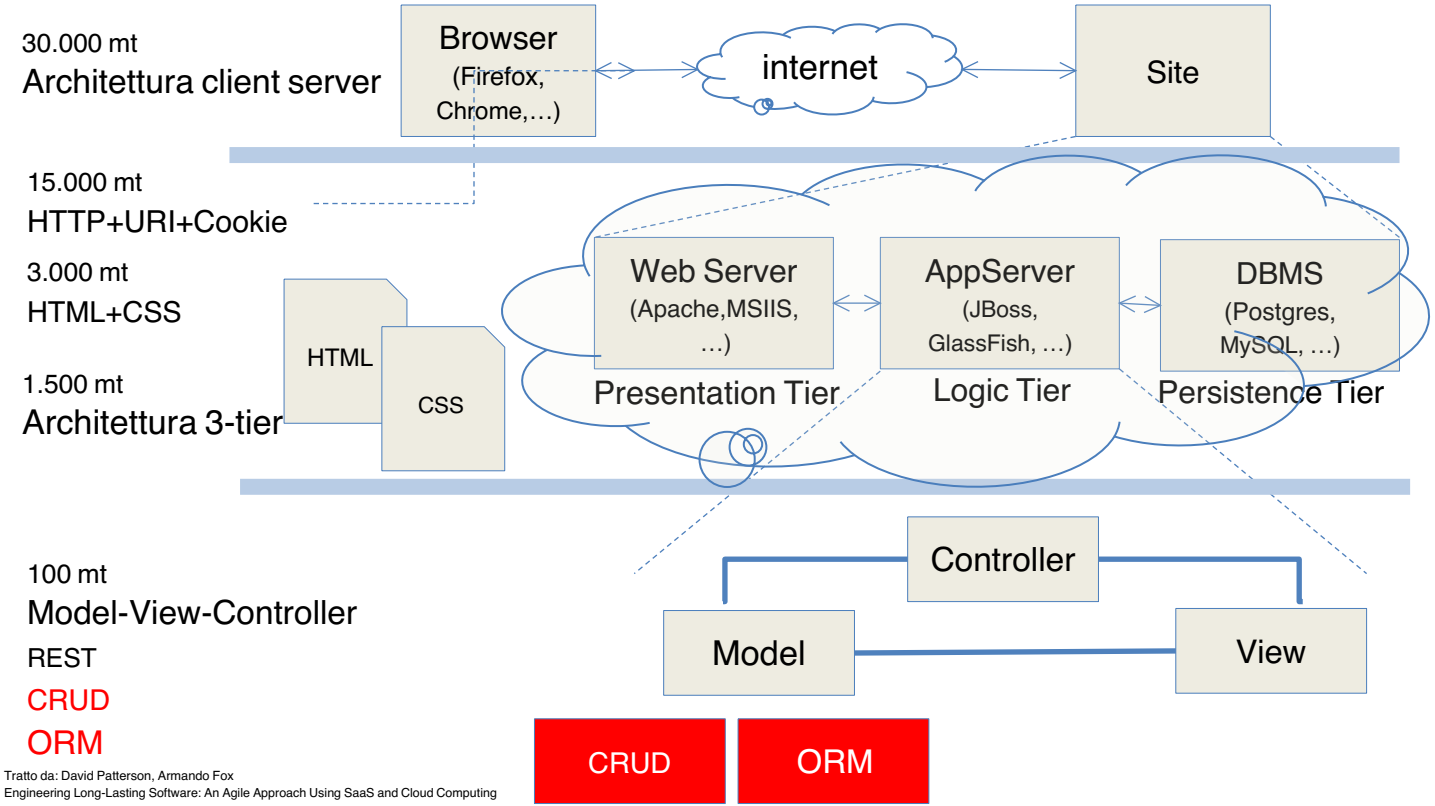
Sistemi informativi su Web

Gestione Persistenza:
Java Persistence API: Object Relational Mapping

aa 2024-2025

Paolo Merialdo
Università degli Studi Roma Tre

Anatomia
di un sistema informativo su Web



Tratto da: David Patterson, Armando Fox
Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing

Sommario

- Parte I: introduzione a JPA
  - Entità
  - Object-Relational-Mapping
- Parte II: Relationship Mapping
  - Associazioni: OO vs relazionale
  - Cardinalità e direzione
  - Mapping delle associazioni
  - Object-Relational-Mapping: approfondimenti
  - Esercizio
  - Mapping associazioni: approfondimenti

Introduzione

- Sistema informativo
  - logica applicativa
  - interazione con l'utente
  - interazione con altri sistemi
  - persistenza
- Persistenza in un database
  - tipicamente un database relazionale: noi ci concentriamo su questi (ma, a seconda delle esigenze, potrebbero essere usate anche altre tecnologie)

Introduzione

- Database relazionale
  - relazioni (tabelle) e ennuple (righe)
  - attributi (colonne)
  - chiavi primarie (primary key)
  - chiavi esterne (foreign key)
- Programmazione ad Oggetti
  - classi
  - oggetti
  - attributi (variabili di istanza di tipo primitivo o String)
  - riferimenti (variabili di istanza di tipo riferimento o collezione di riferimenti)

Introduzione

- Modello relazionale e Modello OO
  - molto diversi
- Object Relational Mapping (ORM)
  - strumenti per favorire il mapping tra i due mondi
  - offrono una vista OO su dati relazionali e viceversa
  - obiettivo: programmare OO affidando la persistenza ad un DBMS
- Java Persistence API (JPA)
  - standard definito a partire da Hibernate (ORM di successo) nel 2006 (JPA 1.0), successivamente rivisto nel 2009 (JPA 2.0) e nel 2013 (JPA 2.1). L'ultima versione (JPA 2.2) è del 2017

Nota terminologica

- Quando ci riferiamo al mondo relazionale, usiamo la seguente terminologia
  - tabella (o relazione)
  - riga (o ennupla o tupla)
  - colonna (o attributo)
  - chiave esterna (o vincolo di integrità referenziale)
- Quando ci riferiamo al mondo OO, usiamo la seguente terminologia
  - classe
  - oggetto
  - attributo (o variabile di istanza)
  - riferimento

Rappresentazioni Grafiche

Classe

Customer

-id: Long
-firstName: String
-lastName: String
-email: String
-phoneNumber: String

Schema Tabella

customer		
id	bigint	nullable=false PK
firstname	varchar(255)	nullable=false
lastname	varchar(255)	nullable=false
email	varchar(255)	nullable=true
phonenumber	varchar(255)	nullable=true

Rappresentazione testuale in forma compatta:
customer(<u>id</u>, firstname, lastname, email, phonenumber)

Tabella

customer

id	firstname	lastname	…
1	Robert	White	…
2	Julie	Brown	…
3	John	Smith	…
…	…	…	…

JPA

- Object Relational Mapping
  - modello di corrispondenza tra classi e tabelle
  - metadati (sintassi per definire i dettagli della corrispondenza)
  - linguaggio di interrogazione (simile a SQL, ma con dot-notation)
  - API e framework
- Varie implementazioni
  - OpenJPA
  - TopLink
  - **Hibernate**
  - EclipseLink
  - …

Entity

- Applicazione OOP: interazione tra oggetti
  - Oggetti: istanze di classi in memoria centrale
- Chiamiamo **entità** (entity) le classi corrispondenti ad oggetti che devono essere memorizzati in maniera persistente in un db
  - nome molto appropriato (cfr BD: modello ER)
  - in linea di massima corrispondono alle classi del modello di dominio (cfr APS)

Object-Relational-Mapping

- Obiettivo: creare una corrispondenza tra classi e tabelle
- Questa corrispondenza offre una "vista" OO agli sviluppatori, consentendogli di usare in maniera trasparente entità (oggetti persistiti) anziché tabelle

Object-Relational-Mapping

- Idea di base:
  - ogni classe corrisponde ad una tabella
    - ogni attributo (variabile di istanza di tipo primitivo o wrapper, String, tipo data/tempo di java.time) corrisponde ad una colonna
  - ogni oggetto corrisponde ad una riga della tabella

Object-Relational-Mapping

- *Configuration by exception*: il mapping segue regole di default:
  - nome classe nome tabella
  - nome attributo nome colonna
  - tipo attributo tipo colonna
- Eventuali eccezioni al default sono definite mediante metadati, sintatticamente rappresentati con annotazioni Java o mediante un descittore XML
- Nel corso useremo quasi sempre i valori di default, e sempre annotazioni Java

Object-Relational-Mapping

- Configurazione di default
  - Ogni entity viene mappata in una tabella
    - nome classe ? nome tabella
  - tutte le variabili corrispondenti ad un tipo primitivo (o corrispondente wrapper), String, java.time.LocalDate, java.time.LocalDateTime vengono mappate automaticamente in una colonna
    - nome attributo ? nome colonna
    - corrispondenza (naturale) tra i tipi primitivi Java (più java.lang.String e i tipi per date e orari di java.time) e i tipi SQL

Corrispondenza tra tipi Java e SQL

Java primitive primitive wrapper	Standard SQL type
boolean/java. lang. Boolean	BOOLEAN
byte/java. lang. Byte	SMALLINT (5)
short/java. lang. Short	SMALLINT (5)
char/java. lang. Character	CHARACTER (n)
int/java. lang. Integer	INTEGER (10)
long/java. lang. Long	BIGINT (19)
float/java. lang. Float	≅ DECIMAL (16,7)
double/java. lang. Double	≅ DECIMAL (32,15)
String	VARCHAR(255)

Esempio (1/3)

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.NamedQuery;
import jakarta.persistence.Column;

@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String name;

     private Float price;

    @Column(length = 2000)
    private String description;

    @Column(nullable = false)
    private String code;
```

Esempio (2/3)

```
public Product() {
}

public Product(String name, Float price, String description, String code) {
   this.name = name;
   this.price = price;
   this.description = description;
   this.code = code;
}

public String getDescription() {
   return description;
}

public void setDescription(String description) {
   this.description = description;
}

public String getCode() {
   return code;
}

public void setCode(String code) {
   this.code = code;
}

public void setId(Long id) {
   this.id = id;
}
```

Esempio (3/3)

```
  public Long getId() {
    return id;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Float getPrice() {
    return price;
  }

  public void setPrice(Float price) {
    this.price = price;
  }
  public boolean equals(Object obj) {
    Product product = (Product)obj;
    return this.getCode().equals(product.getCode());
  }

  public int hashCode() {
    return this.code.hashCode();
   }

  public String toString() {
    final StringBuilder sb = new StringBuilder();
    sb.append("Product");
    sb.append("{id=").append(id);
    sb.append(", name='").append(name);
    sb.append(", price=").append(price);
    sb.append(", description='").append(description);
    sb.append(", code='").append(code);
    sb.append("}\n");
    return sb.toString();
  }
}
```

Esempio (dettagli)

```
+import jakarta.persistence.Entity;[…]

    @Entity
    public class Product {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;

        private String name;

        private Float price;

        @Column(length = 2000)
        private String description;

        @Column(nullable = false)
        private String code;
```

In rosso le annotazioni JPA per il mapping object relational

Entity: restrizioni sul codice Java

- Entità
  - La classe deve avere il costruttore no-arg publico (o protected) e può avere anche altri costruttori
  - La classe non deve essere dichiarata final, nè avere metodi final
- Attributi
  - Tipi primitivi, classi wrapper, java.lang.String, java.time
  - Metodi getter e setter
    (nelle slide successive sono omessi per leggibilità, ma sono obbligatori)

Annotazioni principali

- **@Entity**
  specifica che la classe è un entità (oggetto persistente)
- **@Id**
  specifica che l'attributo è un'identificatore: una chiave primaria nel db
- **@GeneratedValue**
  specifica in che modo deve essere generato il valore della chiave primaria
  - strategy=GenerationType.AUTO indica che il valore dell'id deve essere generata automaticamente (un valore incrementale assegnato dal db)
  - Il valore dell'id viene creato (automaticamente) la prima volta che l'oggetto viene reso persistente nel db
- **@Column**
  specifica eventuali proprietà diverse da quelle di default sulla colonna che corrisponde all'attributo che segue
  - **nullable = false** indica che la colonna (nel db) non ammette valori nulli
  - **length = 2000** indica che la lunghezza della colonna varchar corrispondente all'attributo che segue
  - altre proprietà
  - Va usata solo se serve a imporre proprietà specifiche del mapping. Ogni variabile di tipo primitivo (o wrapper corrispondente), String, java.time viene automanticamente mappata in una colonna della tabella corrispondente all'entità, con un tipo opportuno. Se non dobbiamo specificare nessuna proprietà nel mapping della variabile non è necessario usare l'annotazione (nell'esempio, vedi variabile Float price)

Equals - HashCode

- Su quali attributi verifichiamo l'identità?
  - id assume un valore solo dopo che l'oggetto viene persistito
  - l'identità va verificata sull'identificatore naturale
- Nel nostro esempio l'identificatore naturale di un prodotto è l'attributo code

Esempio

Classe

<<Entity>> Product
- id: Long
- name: String
- price: Float
- description: String
- code: String

mapping

Tabella

product		
id	bigint	nullable=false PK
name	varchar(255)	nullable=false
price	double	nullable=true
description	varchar(2000)	nullable=true
code	varchar(255)	nullable=false

Rappresentazione testuale in forma compatta:
Product(id, name, price, description, code)

Object-Relational-Mapping

- Attraverso un sistema molto ricco di metadati è possibile definire tutti i dettagli del mapping, specificando eccezioni ai valori di default. Esempi:
  - @Table(name = "product_table")
  - @Column(name = "product_code", nullable = false, length = 16)
- E' anche possibile definire
  - chiavi primarie composte da uno o più attributi
  - vincoli di unicità (su uno o più attributi)

Mettiamo tutto in pratica

Esercizio 1

L'obiettivo di questo esercizio è capire come funziona il mapping delle associazioni: scriviamo codice Java e verifichiamo come JPA genera il database corrispondente. Non faremo nessuna operazione sul database.

1. Creare su postgres il database **ecommerce**
2. Creare un progetto Spring Boot **siw-jpa-es1**

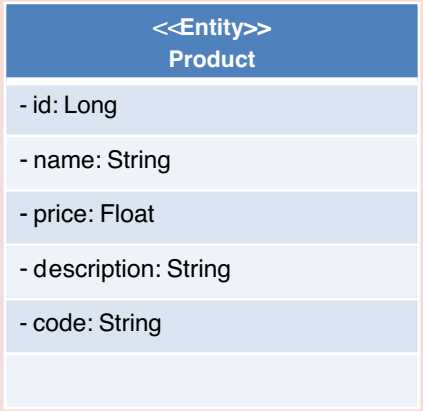
   Come librerie aggiungere solo **spring data jpa** e quella del driver **postgres**

   impostare i parametri di connessione al DBMS nel file **application.properties** (servono solo i parametri di connessione al DBMS)

Esercizio 1

class diagram (vedi prossima slide)

3. Creare il package **it.uniroma3.siw.siw-jpa-es1**
4. Nel package **it.uniroma3.siw.siw-jpa-es1**, scrivere il codice della classi **Product** rappresentata nel class diagram



5. Eseguire il programma (come Spring Boot App)
6. Controllare lo schema della tabella creata da JPA nel database **ecommerce**

Sommario

- Parte I: introduzione a JPA
  - Entità
  - Object-Relational-Mapping
- Parte II: Relationship Mapping
  - Associazioni: OO vs relazionale
  - Cardinalità e direzione
  - Mapping delle associazioni
  - Object-Relational-Mapping: approfondimenti
  - Esercizio
  - Mapping associazioni: approfondimenti

Relationship Mapping

- Su una singola classe il mapping è semplice e intuitivo
- Vediamo ora come vengono mappate le associazioni tra classi
- In questo caso le differenze tra il modello relazionale e il modello ad oggetti rendono il mapping più articolato e vanno tenuti in considerazione diversi aspetti

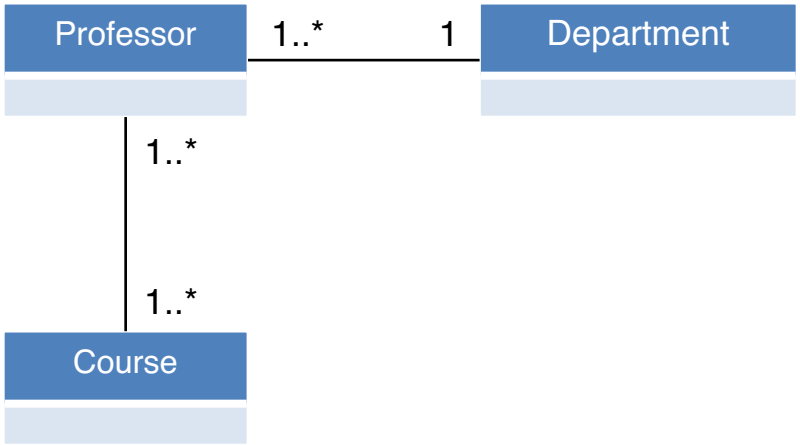
Associazioni nel mondo OO

- Associazioni tra classi
  - sono espresse per mezzo di riferimenti
  - hanno una direzione (che dipende dai requisiti) e possono essere:
    - unidirezionali
    - bidirezionali
  - hanno una cardinalità (o molteplicità):
    - specifica quanti oggetti sono coinvolti nella associazione

Associazioni nel mondo OO: direzione e molteplicità

- Supponiamo di progettare il sistema informativo di una università
- Assumiamo che una parte dei requisiti riportino quanto segue:
  - Ogni docente insegna uno o più corsi e afferisce ad un dipartimento
  - Ogni corso è insegnato da uno o più docenti
  - Ad ogni dipartimento afferiscono uno o più docenti
- Questi requisiti ci parlano delle molteplicità delle associazioni tra le classi. Possiamo iniziare a definire il class diagram che segue

Associazioni nel mondo OO: direzione e molteplicità



Associazioni nel mondo OO: direzione e molteplicità

- Assumiamo che i requisiti riportino anche quanto segue:
  - Il sistema deve po… …ttura alle informazioni di docenti, corsi, e dipartim…
    - Per ogni **cor**… …formazioni sul corso e l'elenco dei docenti c…
    - Per ogni **docen**… …o di afferenza e l'elenco dei corsi insegnati…
    - Per ogni **dipartimento** d… …trate solo le informazioni sul dipartimento

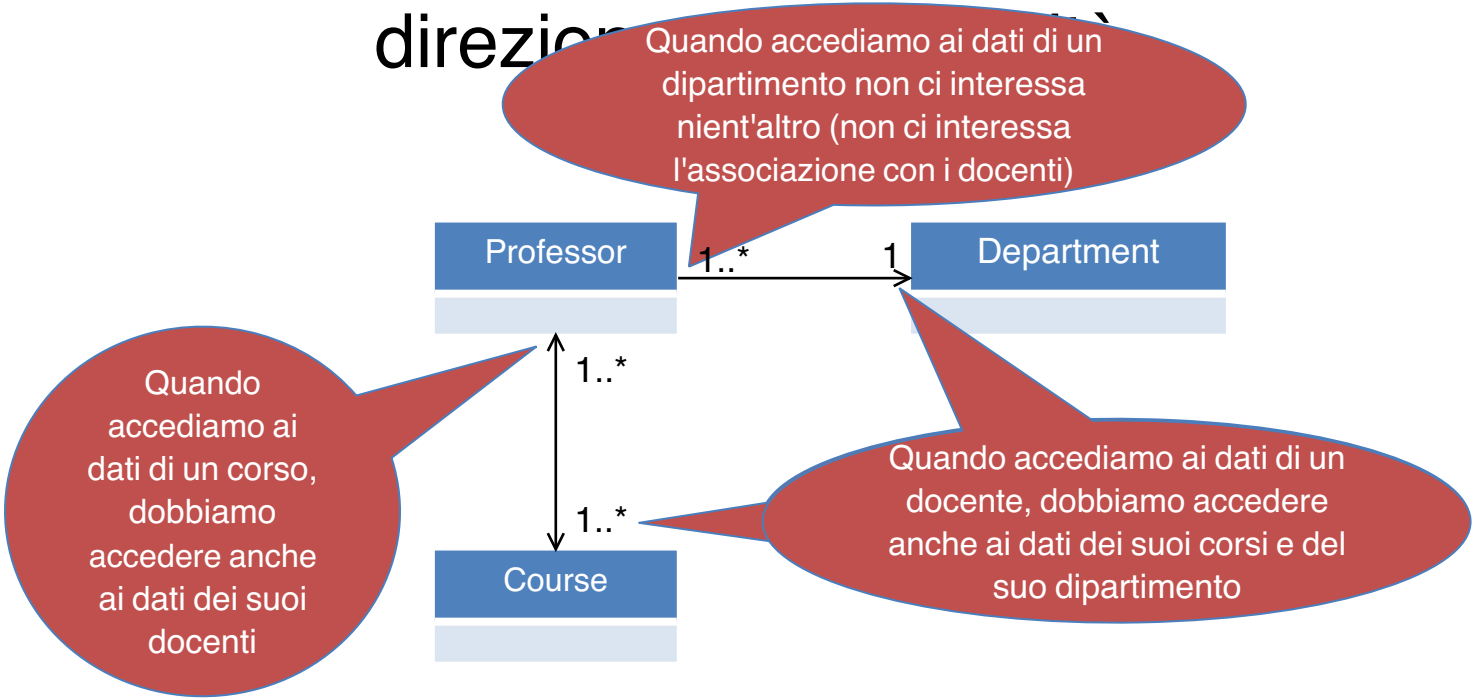
Quando accediamo ai dati di

Quando accediamo ai dati di un docente, dobbiamo accedere

Quando accediamo ai dati di un dipartimento non ci interessa nient'altro (non ci interessa l'associazione con i docenti)

- Questi requisiti ci parlano della direzione delle associazioni tra le classi. Con queste informazioni, possiamo completare il class diagram come segue

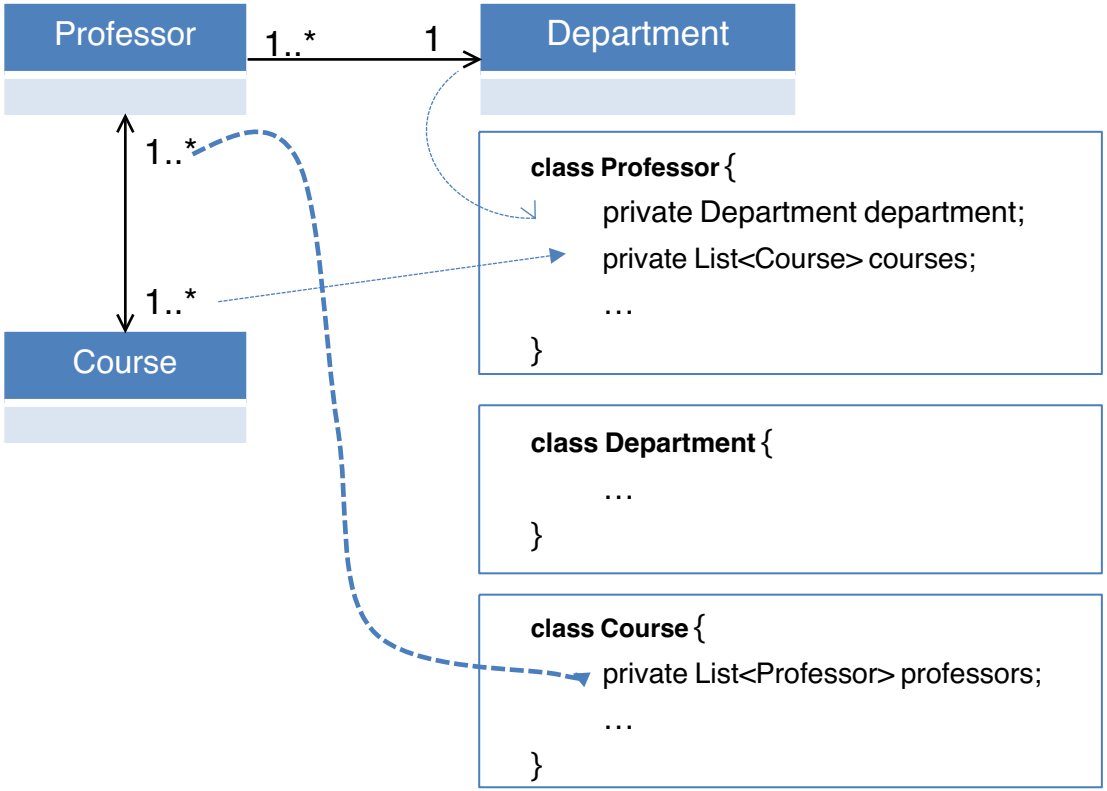
Associazioni nel mondo OO: direzio



Associazioni nel mondo OO: direzione e molteplicità

- Come si traduce il class diagram precedente in un linguaggio orientato agli oggetti?
- Le associazioni si rappresentano tramite riferimenti.
  In ogni classe:
  - ogni associazione rappresentata con un freccia uscente con molteplicità di destinazione 1 (*one*) si traduce in un riferimento ad un oggetto della classe destinazione
  - ogni associazione rappresentata con un freccia uscente con molteplicità di destinazione 1..* (*many*) si traduce in una collezione di riferimenti ad oggetti della classe destinazione

Associazioni nel mondo OO: direzione e molteplicità



Associazioni nel mondo OO: direzione e molteplicità

- Possiamo avere associazioni
  - one-to-many
  - one-to-one
  - many-to-one
  - many-to-many
- E queste possono essere
  - monodirezionali (la freccia è in uno dei due estremi)
  - bidirezionali (la freccia è in entrambi gli estremi)

Associazioni nel mondo relazionale

- Nel mondo relazionale le associazioni sono espresse attraverso chiavi esterne (vincoli di integrità referenziale)
- Non hanno una direzione
- Hanno una cardinalità
  - le associazioni *many-to-many* sono espresse da una "join-table", con chiavi esterne verso le tabelle che rappresentano le classi (entità) che partecipano alla associazione
  - Le associazioni *many-to-one* (che è inutile distinguere dalle associazioni one-to-many, non essendoci direzione) sono espresse da una join table oppure da una chiave esterna nella tabella che rappresenta la classe (entità) dalla parte *many* della associazione
  - Le associazioni *one-to-one* sono espresse da una chiave esterna in una delle due tabelle che rappresentano le classi (entità) che partecipano alla associazione

Associazioni nel mondo relazionale

professor		
id	bigint	nullable=false PK
name	varchar(255)	nullable=false
department_id	bigint	nullable=false FK department.id

department		
id	bigint	nullable=false PK
name	varchar(255)	nullable=false

Questa Foreign Key rappresenta l'associazione molti-a-uno tra professore e dipartimento

Questa tabella di join (con le FK) rappresenta l'associazione molti-a-molti tra corso e professore

teaching		
professor_id	bigint	nullable=false FK professor.id
course_id	bigint	nullable=false FK course.id

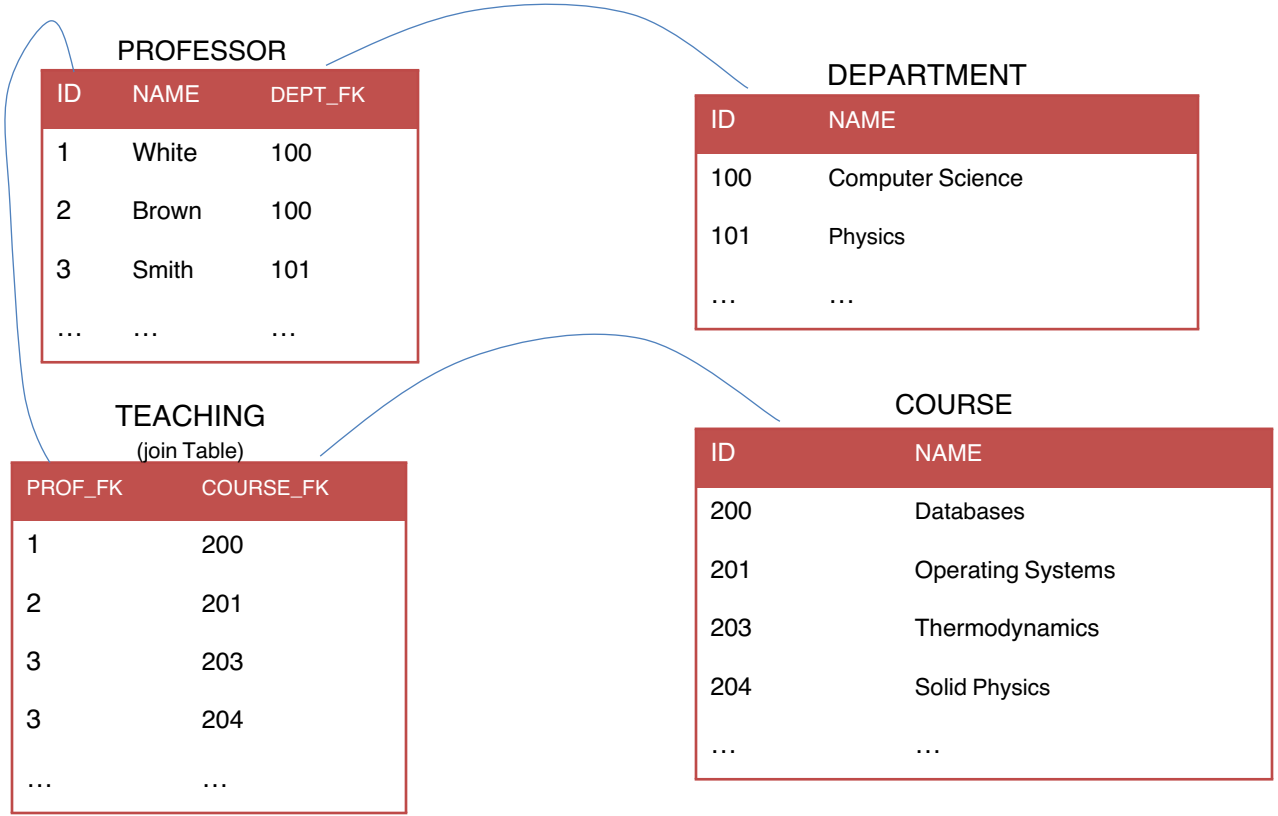
course		
id	bigint	nullable=false PK
name	varchar(255)	nullable=false

Rappresentazione testuale in forma compatta:

Professore (id, name, dept_id) department(id, name) FK: department.id
professor.department_id

Teaching(prof_id, course_id) course(id, name) FK: course.id teaching.course_id

Associazioni nel mondo relazionale



Mapping delle Associazioni

- Il mapping delle associazioni deve coniugare i due mondi (OO e relazionale) in maniera trasparente
- Abbiamo visto che il mapping delle entità sulle tabelle è semplice e intuitivo
- Il mapping delle associazioni è più delicato perchè nelle associazioni il modello OO e il modello relazionale sono più distanti

Mapping delle Associazioni

- Da un punto di vista sintattico, per definire il mapping di una associazione e usiamo speciali annotazioni che vanno poste prima delle variabili riferimento che rappresentano l'associazione
- Le annotazioni specificano la cardinalià dell'associazione:

  @OneToOne

  @OneToMany

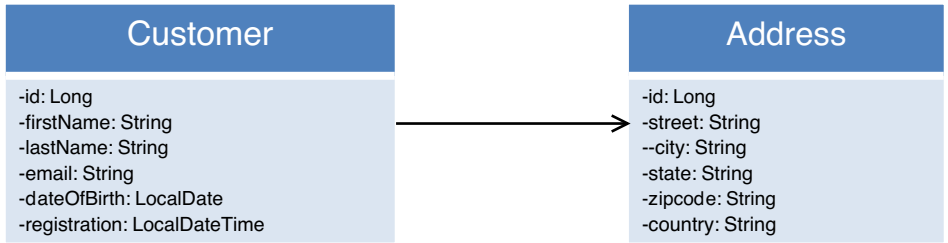
  @ManyToOne

  @ManyToMany
- Ma per definire correttamente il mapping dobbiamo considerare anche la **direzione** dell'associazione

Associazioni unidirezionali

- Le associazioni unidirezionali sono semplici
- In questo caso è sufficiente indicare l'associazione specificando correttamente la cardinalità

Associazioni unidirezionali
OneToOne: Esempio



Associazioni unidirezionali OneToOne: Esempio (cont.)

```
+import javax.persistence.Column;[…]

@Entity
public class Customer {

   @Id
   @GeneratedValue(strategy = AUTO)
   private Long id;
   @Column(nullable = false)
   private String firstName;
   @Column(nullable = false)
   private String lastName;
   private String email;
   private LocalDate dateOfBirth;
   private LocalDateTime registration;
   @OneToOne
   private Address address;
…
// getter, setter
}
```

(nel seguito, getter e setter omessi per leggibilità)

```
package it.uniroma3.siw.model;

+import javax.persistence.Column; […]
i
@Entity
public class Address {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String street;
   @Column(nullable = false)
    private String city;
    private String state;
   @Column(nullable = false)
    private String zipcode;
   @Column(nullable = false)
   private String country;
…
// getter, setter
}
```

Object-Relational-Mapping
Tipi info temporali

- Da Java 8 le info temporali sono rappresentate con le classi del package java.time
  - LocalDate rappresenta una data (giorno, mese, anno)
  - LocalDateTime rappresenta una data complete di orario (giorno, mese, anno, ora, minuti, secondi)
- In SQL le informazioni temporali sono rappresentate da DATE e TIMESTAMP, che hanno diversa granularità (date, date+orario)
- A partire da JPA 2.2, c'é una corrispondenza diretta tra i tipi SQL e le classi di java.time

Object-Relational-Mapping
Tipi info temporali

- A partire da JPA 2.2, c'é una corrispondenza diretta tra i tipi SQL e le classi di java.time

JAVA TYPE	SQL TYPE
java.time.Duration	BIGINT
java.time.Instant	TIMESTAMP
java.time.LocalDateTime	TIMESTAMP
java.time.LocalDate	DATE
java.time.LocalTime	TIME
java.time.OffsetDateTime	TIMESTAMP
java.time.OffsetTime	TIME
java.time.ZonedDateTime	TIMESTAMP

Object-Relational-Mapping
Tipi info temporali

- Nella classe Customer:
  - per la data di nascita:

    private LocalDate dateOfBirth;
  - per data e ora di registrazione:

    private LocalDateTime registration;
- I due tipi sono mappati rispettivamente nei tipi SQL
  - DATE (giorno, mese, anno)
  - TIMESTAMP (giorno, mese, anno, ora, minuto, secondo, msec)

Mettiamo tutto in pratica

Esercizio 2 (1/12)

L'obiettivo di questo esercizio è capire come funziona il mapping delle associazioni: scriviamo codice Java e verifichiamo come JPA genera il database corrispondente. Non faremo nessuna operazione sul database.

1. Continuiamo a lavorare sul progetto **siw-jpa-es1**

Esercizio 2 (2/12)

2. Scrivere il codice delle classi **Customer** e **Address** rappresentate nel class diagram

3. Quante tabelle ci aspettiamo che saranno create da JPA?
   Scrivere il numero di un foglio di carta. Sullo stesso foglio scrivere lo schema delle tabelle che ci aspettiamo saranno create da JPA.
4. Eseguire il programma (come Spring Boot App)
5. Controllare lo schema delle tabelle create dal JPA nel database **products** e confrontarle con lo schema che abbiamo scritto al punto 4

Object-Relational-Mapping
Approfondimenti

- Nel mapping dal mondo OO a quello relazionale ci dobbiamo preoccupare anche delle corrispondenze tra:
  - Vincoli di unicità

Object-Relational-Mapping
Vincoli di Unicità

- Si specificano attraverso un attributo della annotazione @Table
- Per gli attributi semplici (una colonna) è sufficiente aggiungere l'annotazione
  - @Column(**unique=true**)
- Per gli attributi composti (>1 colonne) la sintassi è un po' articolata
- Vediamola attraverso due esempi

Object-Relational-Mapping
Vincoli di Unicità

```
@Entity
public class Product {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String name;
   …
   @Column(unique=true, nullable = false)
   private String code;
…
}
```

```
CREATE TABLE product(
  id bigint NOT NULL,
  name character varying(255) NOT NULL,
  code character varying(255) NOT NULL,
  description character varying(2000),

  CONSTRAINT product_pkey PRIMARY KEY (id),
  CONSTRAINT u_product_code UNIQUE (code)
)
```

Object-Relational-Mapping
Vincoli di Unicità

```
@Entity
@Table(uniqueConstraints=@UniqueConstraint(columnNames={"firstname","lastname"}))
public class Customer {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String firstName;
   @Column(nullable = false)
   private String lastName;
   private String email;
   ...
}
```

```
CREATE TABLE customer (
  id bigint NOT NULL,
  firstname character varying(255) NOT NULL,
  lastname character varying(255) NOT NULL,
  email character varying(255),

  CONSTRAINT customer_pkey PRIMARY KEY (id),
  CONSTRAINT u_customer_firstname UNIQUE (firstname, lastname)
)
```

Mettiamo tutto in pratica

Esercizio 2

- Modificare le classi dell'Esercizio 2 affinchè siano espressi i seguenti vincoli:
  - non possono esserci due clienti che abbiano stesso nome, cognome, data di nascita

Associazioni unidirezionali
OneToOne: Esempio (cont.)

Nel DB vengono generate le tabelle e il vincolo di chiave esterna (FK) che rappresenta l'associazione come indicato in figura

customer		
id	bigint	nullable=false PK
firstname	varchar(255)	nullable=false
lastname	varchar(255)	nullable=false
email	varchar(255)	nullable=true
dateOfBirth	date	nullable=true
registration	timestamp	nullable=true
address_id	bigint	nullable=true FK address.id

address		
id	bigint	nullable=false PK
street	varchar(255)	nullable=false
city	varchar(255)	nullable=true
state	varchar(255)	nullable=true
zipcode	varchar(8)	nullable=true
country	varchar(255)	nullable=true

Configuration by Exception

- Il mapping di default per la colonna che rappresenta la foreign key ha il nome costruito come segue:
  - nome tabella corrispondente alla entità destinazione, seguito dal simbolo _ seguito da chiave primaria tabella corrispondente alla entità destinazione
  - Nel nostro esempio: **address_id**

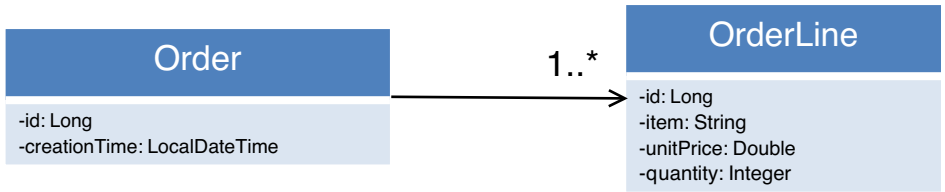
  - *Configuration by exception*: è possibile (**ma sconsigliato**) modificare il nome di questa colonna usando l'annotazione @JoinColumn(name="nome colonna") che va posta immediatamente dopo l'annotazione che indica la cardinalità della associazione.
    Nel nostro esempio:
    @OneToOne
    @JoinColumn(name = "address_fk")

Associazioni unidirezionali
OneToMany: Esempio

Esempio (cont.)

```
+import javax.persistence.Entity;[…]

@Entity
@Table(name = "orders")
public class Order {

   @Id
   @GeneratedValue(strategy =
GenerationType.AUTO)
   private Long id;
   private LocalDateTime creationTime;
   @OneToMany
   private List<OrderLine> orderLines;

…
}
```

```
+import javax.persistence.Entity;[…]

@Entity
public class OrderLine {

   @Id
   @GeneratedValue(strategy =
GenerationType.AUTO)
   private Long id;
   private int quantity;
   private Float price;
```



Associazioni unidirezionali OneToMany

- Per le associazioni unidirezionali OneToMany JPA crea di default una tabella di join
  - una tabella che contiene coppie di chiavi esterne delle due tabelle corrispondenti alle entità che partecipano alla associazione.
- Nel nostro esempio, vedi prossima slide

Esempio (cont.)

Queste sono le tabelle generate dal mapping

orders		
id	bigint	nullable=false
creationtime	timestamp	nullable=false

orderline		
id	bigint	nullable=false
quantity	int	nullable=false
price	real	nullable=true

orders_orderline		
orders_id	bigint	nullable=false **FK orders.id**
orderslines_id	bigint	nullable=false **FK orderline.id**

Configuration by Exception

- Abbiamo cambiato il nome di default della tabella corrispondente all'entità Order perchè in Postgres Order è una parola chiave
- Il nome (di default) della tabella di join è formato dalla concatenazione dei nomi delle due tabelle, separati dal simbolo _

Associazioni unidirezionali
OneToMany

- Dalla progettazione logica studiata nel corso "Basi di dati" sappiamo che per le associazioni OneToMany la tabella di join non è necessaria: l'associazione si può rappresentare con una chiave esterna nella tabella dalla parte "many"
- Per forzare la creazione della foreign key nella tabella va specificato il nome della colonna di join attraverso l'annotazione **@JoinColumn(name = "nome colonna FK")** che va messa ubito dopo l'annotazione **@OneToMany**

Esempio

```
+ import java.util.ArrayList;[…]

@Entity
@Table(name = "orders")
public class Order {

   @Id
   @GeneratedValue(strategy =
GenerationType.AUTO)
   private Long id;
   private LocalDateTime creationTime;
   private LocalDateTime deliveryDate;
   @OneToMany
   @JoinColumn(name = "orders_id")
   private List<OrderLine> orderLines;

…
```

```
+import javax.persistence.Entity;[…]

@Entity
public class OrderLine {

   @Id
   @GeneratedValue(strategy =
GenerationType.AUTO)
   private Long id;
   private int quantity;
   private Float price;
```

Esempio (cont.)

Queste sono le tabelle generate dal mapping

orders		
id	bigint	nullable=false
creationtime	timestamp	nullable=false

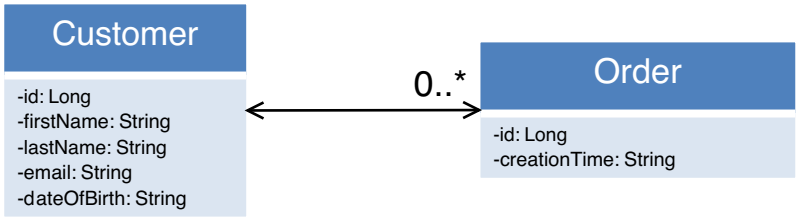
orderline		
id	bigint	nullable=false
item	varchar(255)	nullable=false
quantity	int	nullable=false
price	real	nullable=true
orders_id	bigint	nullable=false **FK orders.id**

Associazioni bidirezionali

- In una associazione bidirezionale, le due entità coinvolte hanno un riferimento (o una collezione di riferimenti) l'una all'altra
- E' fondamentale esprimere la reciprocità dei riferimenti: i riferimenti nelle due entità rappresentano la stessa associazione
- Nel caso delle associazioni unidirezionali il problema non si pone, perchè il riferimento è uno solo (quello nella entità sorgente)

Esempio

```
public class Customer {

   private Long id;
   private String firstName;
   private String lastName;
   private String email;
   private LocalDate dateOfBirth;

   private Address address;
   private List<Order> orders;
…
}
```




```
public class Order {

   private Long id;
   private LocalDateTime creationTime;
   private Customer customer;
…
}
```

Esprimono la stessa associazione

Associazioni bidirezionali

- In JPA, per esprimere la reciprocità dei riferimenti di una associazione bidirezionale adorniamo l'annotazione della parte One-to-Many con l'attributo **mappedBy**:

  @OneToMany**(mappedBy = "nome della variabile della parte inversa che implementa l'associazione")**

Esempio (cont.)

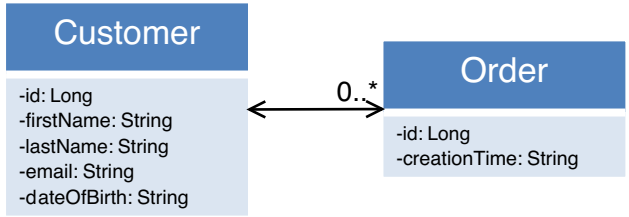
- Parte proprietaria: Order
- Parte inversa: Customer
- Quindi:
  - nella classe Order

    @ManyToOne

    private Customer customer;
  - nella classe Customer

    @OneToMany(mappedBy = "customer")

    private List<Order> orders;



Esempio (cont.)

```
+import javax.persistence.Entity;[…]

@Entity
public class Customer {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String firstName;
   @Column(nullable = false)
   private String lastName;
   private String email;
   private LocalDate dateOfBirth;
   @OneToOne
   private Address address;
   @OneToMany(mappedBy = "customer")
   private List<Order> orders;
…
}
```

```
+import javax.persistence.Entity;[…]

@Entity
@Table(name = "orders")
public class Order {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private LocalDateTime creationTime;
   @ManyToOne
   private Customer customer;
…
}
```

Esempio (cont.)

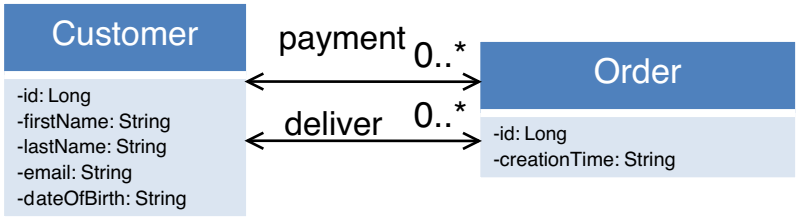
Cosa succede se togliamo il mapping?

```
+import javax.persistence.Entity;[…]

@Entity
public class Customer {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String firstName;
   @Column(nullable = false)
   private String lastName;
   private String email;
   private LocalDate dateOfBirth;
   @OneToOne
   private Address address;
   @OneToMany
   private List<Order> orders;
…
}
```

```
+import javax.persistence.Entity;[…]

@Entity
@Table(name = "orders")
public class Order {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private LocalDateTime creationTime;
   @ManyToOne
   private Customer customer;
…
}
```

Perchè serve il mapping

- Consideriamo il seguente schema



- Ci sono due associazioni tra Customer e Order
- Vediamo il codice corrispondente (prossima slide)

Perchè serve il mapping

```
+import javax.persistence.Entity;[…]

@Entity
public class Customer {

   @Id
   @GeneratedValue(strategy =
GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String firstName;
   @Column(nullable = false)
   private String lastName;
   private String email;
   private LocalDate dateOfBirth;
   @OneToOne
   private Address address;
   @OneToMany(mappedBy = "payer")
   private List<Order> payedOrders;
   @OneToMany(mappedBy = "deliver")
   private List<Order> orders;
…
}
```

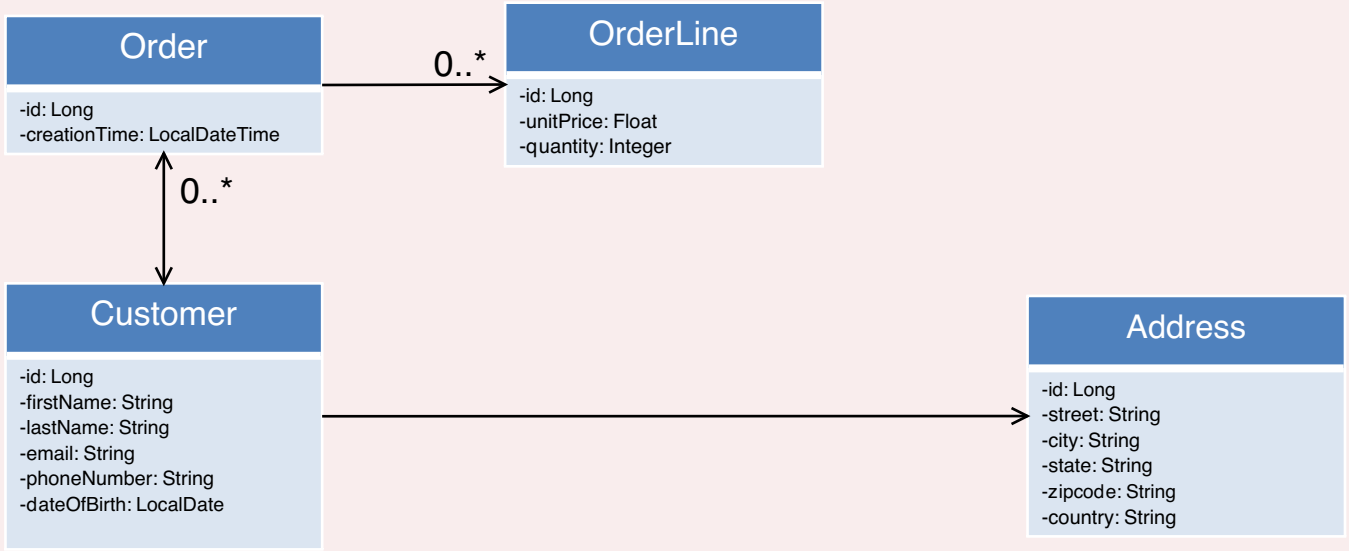
```
package it.uniroma3.siw;

+import javax.persistence.Entity;[…]

@Entity
@Table(name = "orders")
public class Order {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private LocalDateTime creationTime;
   @ManyToOne
   private Customer payer;
   @ManyToOne
   private Customer deliver;

…
}
```

Senza specificare mappedBy non sarebbe possibile capire la semantica dei riferimenti

Mettiamo tutto in pratica

Esercizio 2 (3/12)

Continuiamo a lavorare sul progetto siw-jpa-es1



Esercizio 2 (4/12)

7. Aggiungere il codice delle classi **Order** e **OrderLine** e modificare il codice della classe **Customer** rappresentate nel class diagram della slide precedente
8. Quante tabelle ci aspettiamo che saranno create da JPA? Scrivere il numero di un foglio di carta. Sullo stesso foglio scrivere lo schema delle tabelle che ci aspettiamo saranno create da JPA.
9. Eseguire il programma (come Spring Boot App)
10. Controllare lo schema delle tabelle create dal JPA nel database **products** e confrontarle con lo schema che abbiamo scritto al punto 8

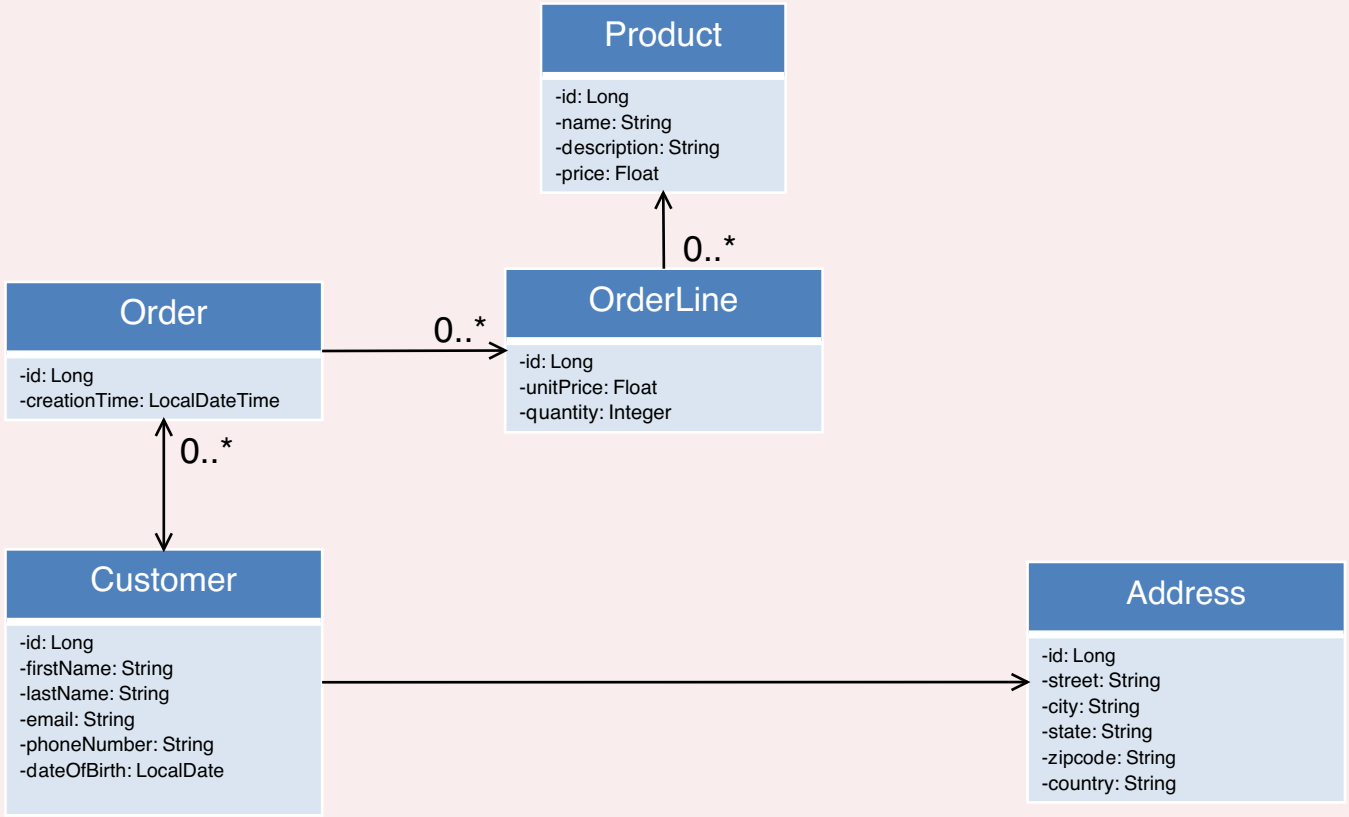
Per i passi successivi, vedi le prossime slide

Esercizio 2 (5/12)

11. Se abbiamo svolto correttamente il mapping della associazione tra **Customer** e **Order**, nella classe **Customer** abbiamo usato l'attributo mappedBy nella relazione one-to-many. Provare a rimuovere l'attributo mappedBy
12. Quante tabelle ci aspettiamo che saranno create da JPA? Scrivere il numero di un foglio di carta. Sullo stesso foglio scrivere lo schema delle tabelle che ci aspettiamo saranno create da JPA. Confrontare questo schema con quello precedente
13. Eseguire il programma (come Spring Boot App)
14. Controllare lo schema delle tabelle create dal JPA nel database **products** e confrontarle con lo schema che abbiamo scritto al punto 12. Aggiungere l'attributo mappedBy (soluzione corretta).

Per i passi successivi, vedi le prossime slide

Esercizio 2 (6/12)



Esercizio 2 (7/12)

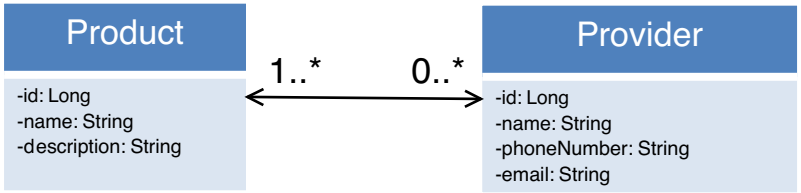
15. Aggiungere il codice della classe **Product** e modificare il codice della classe **OrderLine** rappresentate nel class diagram della slide precedente
16. Quante tabelle ci aspettiamo che saranno create da JPA? Scrivere il numero di un foglio di carta. Sullo stesso foglio scrivere lo schema delle tabelle che ci aspettiamo saranno create da JPA
17. Eseguire il programma (come Spring Boot App)
18. Controllare lo schema delle tabelle create dal JPA nel database **products** e confrontarle con lo schema che abbiamo scritto al punto 16

Per i passi successivi, vedi le prossime slide di esercizi

Associazioni bidirezionali ManyToMany

- Nel mondo OO abbiamo una collezione di riferimenti in entrambe le entità
- Nel mondo relazionale abbiamo una tabella di join (coppie di foreign key alle tabelle corrispondenti alle due entità)
- Una associazione ManyToMany nel db è perfettamente simmetrica. Ma anche in questo caso è necessario specificare la reciprocità dei riferimenti: se non lo facciamo JPA interpreta le due collezioni di riferimenti come due associazioni indipendenti
- Si usa la stessa sintassi vista per le associazioni bidirezionali one-to-many. Si adorna una delle due annotazioni che esprimono la relazione con l'attributo mapped-by:
  (mappedBy = "attributo della owner side che implementa l'associazione")

Esempio


Product
-id: Long
-name: String
-description: String
1..*
0..*
Provider
-id: Long
-name: String
-phoneNumber: String
-email: String

Esempio (cont.)

```
+import java.util.ArrayList;[…]

@Entity
public class Product {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String name;
   private String description;
   @ManyToMany(mappedBy = "products")
   private List<Provider> providers;
…
}
```

```
+import java.util.ArrayList;[…]

@Entity
public class Provider {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(nullable = false)
   private String name;
   private String email;
   @ManyToMany
   private List<Product> products;
…
}
```

Esempio (cont.)

product		
id	bigint	nullable=false
name	varchar(255)	nullable=false
description	varchar(255)	nullable=true

provider		
id	bigint	nullable=false
name	varchar(255)	nullable=false
phonenumber	varchar(255)	nullable=true
email	varchar(255)	nullable=true

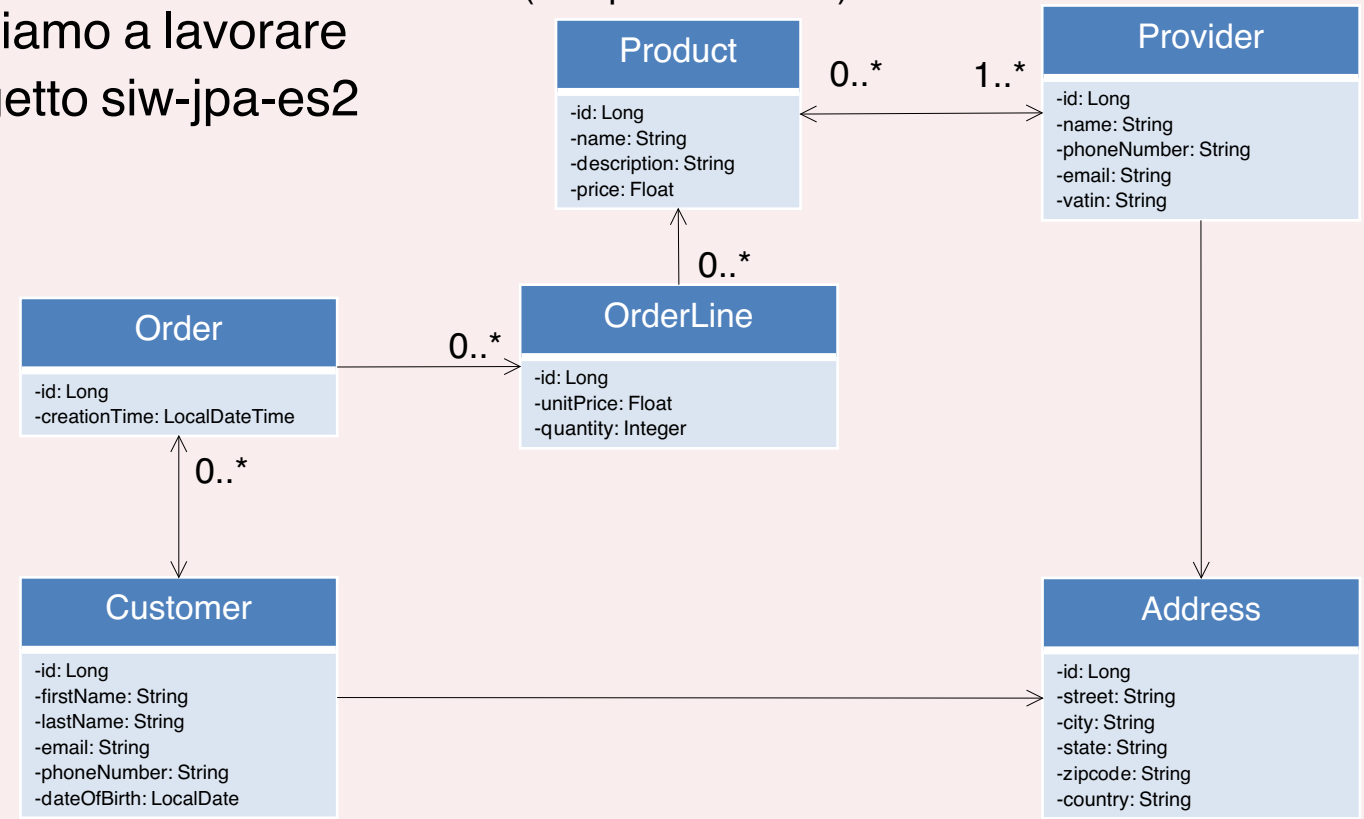
product_provider		
product_id	bigint	nullable=false **FK product.id**
provider_id	bigint	nullable=false **FK provider.id**

Mettiamo tutto in pratica

Esercizio 2 (7/12)

class diagram per il punto 19
(vedi prossima slide)

Continuiamo a lavorare
sul progetto siw-jpa-es2



Esercizio 2 (8/12)

19. Aggiungere il codice della classe **Provider** e modificare il codice della classe **Product** rappresentate nel class diagram della slide precedente
20. Quante tabelle ci aspettiamo che saranno create da JPA? Scrivere il numero di un foglio di carta. Sullo stesso foglio scrivere lo schema delle tabelle che ci aspettiamo saranno create da JPA
21. Eseguire il programma (come Spring Boot App)
22. Controllare lo schema delle tabelle create dal JPA nel database **products** e confrontarle con lo schema che abbiamo scritto al punto 20

Esercizio 2 (9/12)

19-bis. Se abbiamo svolto correttamente il mapping della associazione tra Product e Provider, in una delle due classi abbiamo usato l'attributo mappedBy nella relazione many-to-many. Provare a rimuovere l'attributo mappedBy

20-bis. Quante tabelle ci aspettiamo che saranno create da JPA? Scrivere il numero di un foglio di carta. Sullo stesso foglio scrivere lo schema delle tabelle che ci aspettiamo saranno create da JPA. Confrontare questo schema con quello dell'esercizio 19.

21-bis. Eseguire il programma (come Spring Boot App)

22-bis. Controllare lo schema delle tabelle create dal JPA nel database **products** e confrontarle con lo schema che abbiamo scritto al punto 20-bis.
Aggiungere l'attributo mappedBy (soluzione corretta).

FINE

Cardinalità e direzione tra entità

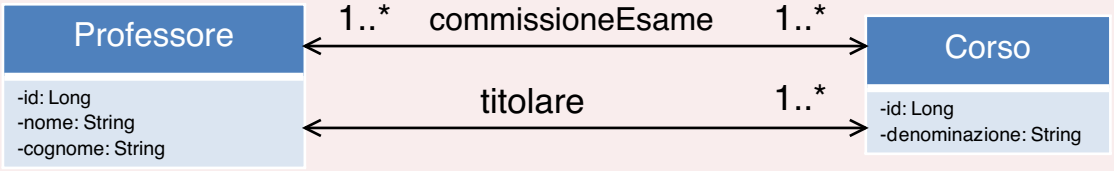
- I concetti visti fino ad ora si applicano nelle diverse combinazioni di cardinalità e direzione

CARDINALITA'	DIREZIONE
OneToOne	unidirectional
OneToOne	bidirectional
OneToMany	unidirectional
ManyToOne/OneToMany	bidirectional
ManyToOne	unidirectional
ManyToMany	unidirectional
ManyToMany	bidirectional

Mettiamo tutto in pratica

Esercizio 3

- Il class diagram in figura modella la seguente realtà:
  - Un professore è titolare di uno o più corsi
  - Un corso ha uno ed un solo professore titolare
  - Un professore può essere nella commissione di esame di uno o più corsi
  - Un corso ha una commissione di esame alla quale partecipano uno o più professori



Esercizio 3

1. Avviare postgres. Creare il database **university**
2. Creare un progetto Eclipse **siw-jpa-es3** e impostare opportunamente le configurazioni nel file **application.properties**
3. Come per l'esercizio 2, l'obiettivo di questo esercizio è capire come funziona il mapping delle associazioni. Anche qui scriviamo codice Java e verifichiamo come JPA genera il database corrispondente.

Esercizio 3

4. Scrivere il codice delle classi **Corso** e **Professore** rappresentate nel classe diagram della slide precedente
5. Quante tabelle ci aspettiamo che saranno create da JPA? Scrivere il numero di un foglio di carta. Sullo stesso foglio scrivere lo schema delle tabelle che ci aspettiamo saranno create da JPA
6. Eseguire il programma (come Spring Boot App)
7. Controllare lo schema delle tabelle create dal JPA nel database **university** e confrontarle con lo schema che abbiamo scritto al punto 5

Per i passi successivi, vedi le prossime slide

Esercizio 3

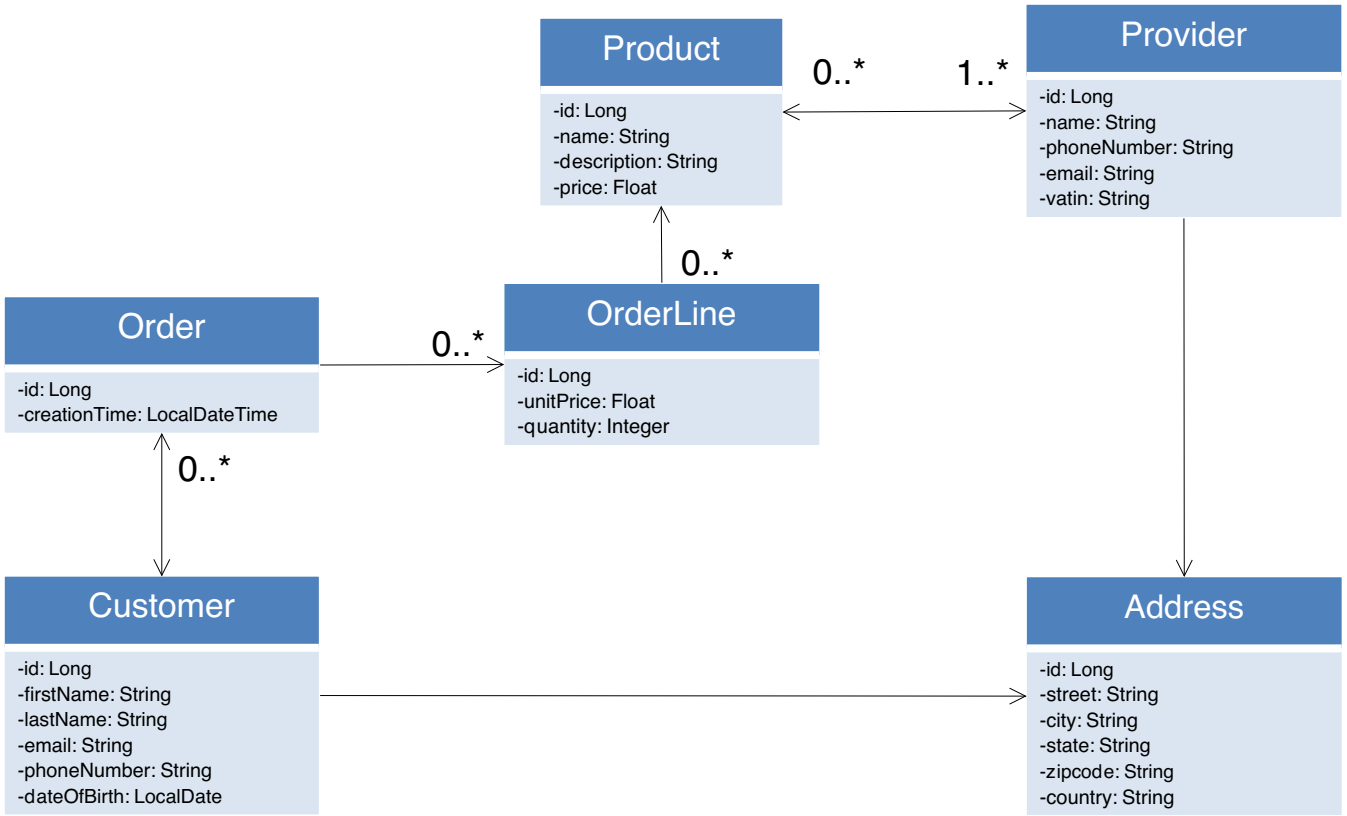
8. Modifichiamo il codice: togliamo l'attributo mappedBy
   che abbiamo usato (in una delle due classi) per rappresentare
   la reciprocità dei riferimenti nelle associazioni
9. Quante tabelle ci aspettiamo che saranno create da JPA?
   Scrivere il numero di un foglio di carta. Sullo stesso foglio
   scrivere lo schema delle tabelle che ci aspettiamo
   saranno create da JPA
10. Eseguire il programma (come Spring Boot App)
11. Controllare lo schema delle tabelle create dal JPA
    nel database **university** e confrontarle con lo schema che
    abbiamo scritto al punto 9

FINE

Approfondimenti
Mapping Associazioni

- Strategie di fetch
- Mapping classi estese (cenni)

Class Diagram Esercizio 2



Strategia di *Fetch*

- Quando JPA recupera dati dal database e crea oggetti in memoria, come si comporta con I riferimenti?
- Ad esempio, consideriamo il class diagram dell'Esercizio 2:
  - supponiamo di dover recuperare i dati di un oggetto Customer (ci servono in una operazione del sistema che prevede di recuperare nome cognome e numero di telefono di un cliente)
  - un oggetto Customer ha anche una lista di riferimenti ai suoi ordini: dobbiamo recuperare anche i dati degli ordini, costruire gli oggetti Order e riempire la lista dei riferimenti agli oggetti Order di Customer?
  - e ogni oggetto Order ha una lista di riferimenti ad oggetti OrderLine, ogni OrderLine ad un Product, ogni Product ha una lista di riferimenti a oggetti Provider
  - Dobbiamo recuperare dal database tutti questi dati per costruire un oggetto Customer? Quanti join (operazione costosa) deve fare il DBMS?
  - Ci servivano nome, cognome e numero di telefono di un cliente e siamo arrivati a prendere l'indirizzo di tutti i fornitori di tutti I prodotti ordinati dal cliente!

Strategia di *Fetch*

- Per evitare operazioni dispendiose (e magari non necessarie) sul database è possibile specificare come deve essere fatto il caricamento degli oggetti "collegati" dalle associazioni
  - **Lazy Load** (caricamento pigro): viene caricato il solo oggetto richiesto; gli oggetti collegati vengono richiesti solo se l'applicazione chiede di seguire un collegamento
  - **Eager** (impaziente): vengono caricati subito anche gli oggetti collegati

Strategia di *Fetch*

- Default — Strategia applicata
  @OneToOne — EAGER
  @ManyToOne — EAGER
  @OneToMany — LAZY
  @ManyToMany — LAZY
- Per modificare la strategia di fetch, l'annotazione della associazione viene adornata con l'attributo **fetch**
  @OneToOne(fetch = FetchType.EAGER)
  @ManyToMany(fetch = FetchType.LAZY)

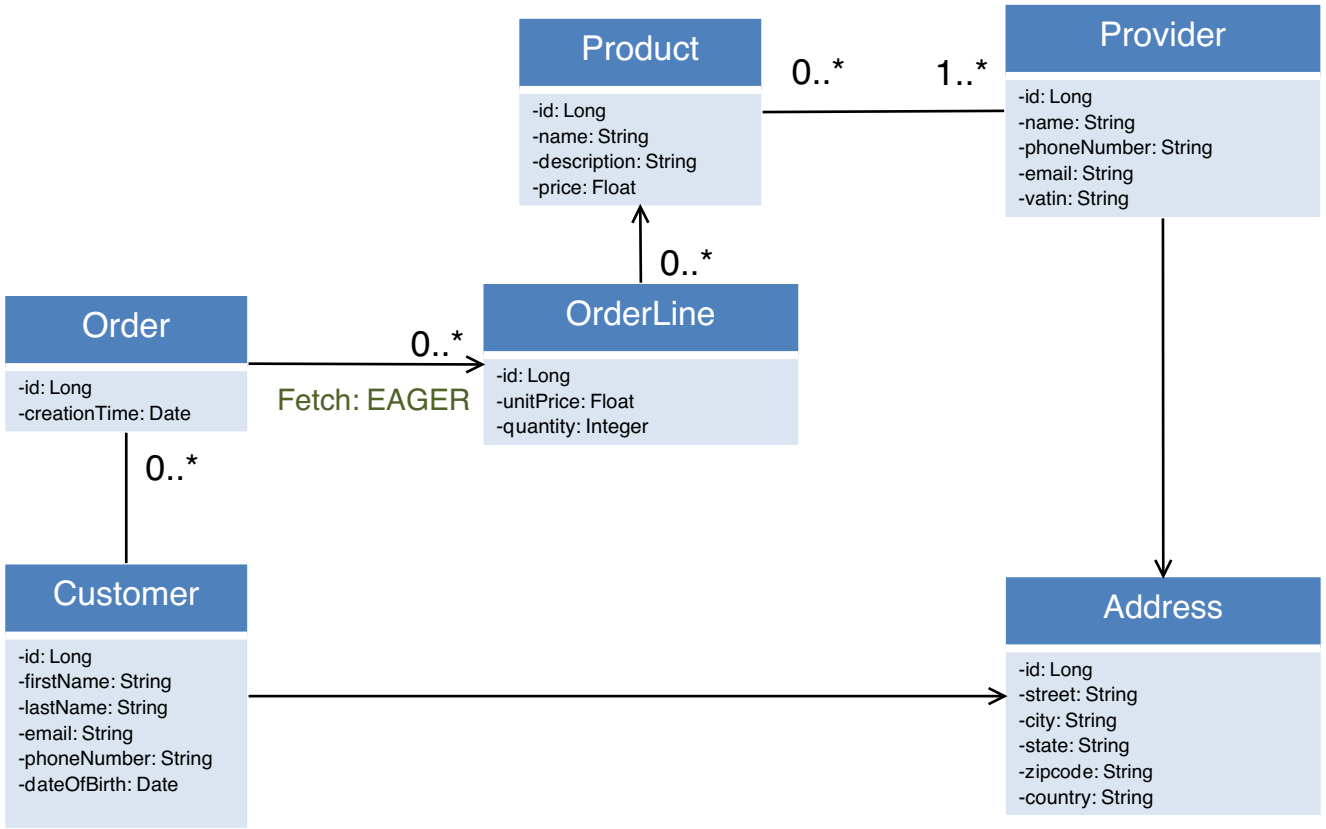
Strategia di *Fetch*

- In generale conviene usare i valori di default
  - ci possono essere significative perdite di prestazioni se si usa in maniera impropria EAGER
- Eventuali eccezioni al default devono essere giustificate dalle operazioni specificate nei casi d'uso

Progettazione

- Strategie di Fetch sono decisioni che vengono prese in fase di progettazione
- Le indichiamo nel class diagram attraverso opportune annotazioni (vedi esempio nella slide successiva)
  - Se non viene specificato nulla vale il default

Esempio


Product
-id: Long
-name: String
-description: String
-price: Float
0..*
1..*
Provider
-id: Long
-name: String
-phoneNumber: String
-email: String
-vatin: String
0..*
Order
-id: Long
-creationTime: Date
0..*
Fetch: EAGER
OrderLine
-id: Long
-unitPrice: Float
-quantity: Integer
0..*
Customer
-id: Long
-firstName: String
-lastName: String
-email: String
-phoneNumber: String
-dateOfBirth: Date
Address
-id: Long
-street: String
-city: String
-state: String
-zipcode: String
-country: String

Mettiamo tutto in pratica

Esercizio 5

- Modificare le classi dell'Esercizio 2 affinchè siano rispettate le strategie di fetch

Mapping Classi Estese (cenni)

- Diverse Strategie:
  - Single Table Strategy
  - Joined Strategy
  - Table-per-Class Strategy
- Principi simili a quelli visti nel corso di Basi di dati per la traduzione delle gerarchie da ER a relazionale
- Non approfondiamo l'argomento