



# Java Persistence API

**Amleto Di Salle**

Email: [amleto.disalle@univaq.it](mailto:amleto.disalle@univaq.it)

Università degli studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Via Vetoio – 67100 Coppito L'Aquila

# Sommario (1)

- Cosa è persistenza
- Persistenza applicazioni OO
- ORM e Standards
- Mapping
  - Proprietà, componenti
  - Associazioni uno-a-uno, multi-a-1, 1-a-molti
  - Associazioni multi-a-molti
  - Ereditarietà

# Sommario (2)

- Architettura
- Persistence unit
- Configurazione
  - persistence.xml
  - Entity Manager
- Sessione
- Stati degli oggetti persistenti
- JPA Query Language

# Risorse

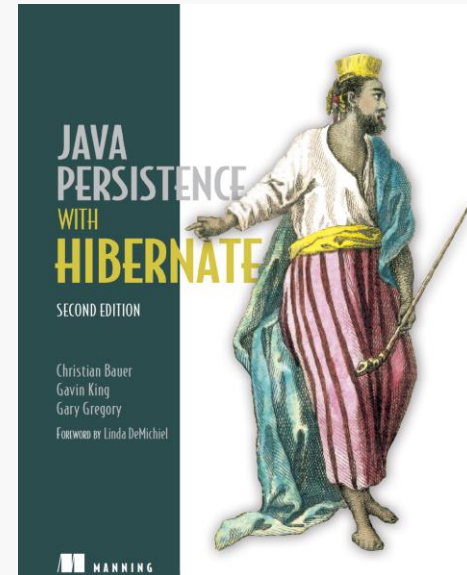
## Libro

Titolo: Java Persistence with Hibernate, 2° edition

Autori: Christian Bauer, Gavin King, Gary Gregory

ISBN: 9781617290459

Sito: <http://jpwh.org/>



## Specifica JPA

<https://jcp.org/en/jsr/detail?id=338>

# Cosa è la persistenza

- Applicazioni richiedono memorizzazione dati in maniera persistente
- Metodi
  - File binari (serializzazione) o di testo (XML)
  - DB Relazionali (usando SQL)
  - DB oggetti
  - DB grafi, documentali ecc

# Persistenza applicazioni OO

- Ammette ad un oggetto di sopravvivere al processo che lo ha creato
- Stato dell'oggetto memorizzato su disco e un oggetto con lo stesso stato può essere ricreato in futuro
- Non è limitato ad un solo oggetto ma all'intero grafo di oggetti interconnessi
- RDBMS forniscono una rappresentazione strutturata dati persistenti dandone la possibilità di manipolarli, cercarli, ordinarli ecc
- DBMS responsabili di gestire concorrenza e integrità dei dati; condividere dati tra i vari utenti/applicazioni

# Persistenza applicazioni OO

- Applicazioni con un *domain model* (classi e relazioni tra classi) non utilizza direttamente la rappresentazione tabulare delle entità
- Esempio
  - Tabelle ITEMS e BIDS
  - Classi Item e Bid
- Logica di business interagisce con domain model object-oriented e interconnessione tra classi
- Ovviamente nelle applicazioni semplici si potrebbe pensare di usare SQL e/o stored procedure senza il domain model
- Tuttavia domain model permette maggior riuso del codice e maggiore manutenibilità

# Cosa è un ORM

- Persistenza automatica (e trasparente) da oggetti (Java) a tabelle (database relazionali) utilizzando metadati che descrivono il mapping
  - Trasformazione dei dati da una rappresentazione ad un'altra (in modo reversibile)
    - Implica alcune penalità di prestazione
- Se è implementato come middleware, ci sono molte opportunità per l'ottimizzazione (a differenza della codifica manuale)
- Costituito da
  - API per eseguire CRUD di base su oggetti di classi persistenti
  - Linguaggio (o API) per specificare query che fanno riferimento a classi e proprietà
  - Struttura per specificare metadati
  - Tecnica per interagire con oggetti transazionali



# Vantaggi di un ORM

- Produttività
- Manutenibilità
- Performance
- Indipendenza dal vendor

# Hibernate, EJB3 e JPA

- Hibernate strumento object/relational mapping
  - Fornisce tutti i vantaggi ORM elencati in precedenza
  - API nativa
- Programmazione e persistenza del modello EJB ampiamente adottata nell'industria
  - Fattore importante per il successo di Java EE
- Standard EJB 2.1 doveva essere migliorato
- Sun definito nuova specifica Java EJB 3.0 (JSR 220) con obiettivo
  - Semplificare gli EJB
  - Prendere idee e concetti di prodotti e progetti di successo già esistenti (Hibernate)
  - Team di Hibernate ha contribuito definizione specifica

# Hibernate, EJB3 e JPA

- Specifica EJB 3.0 è disponibile in più parti
  - Prima parte: definisce nuovo modello programmazione EJB
    - Session, MDB, ecc
  - Seconda parte (definita in JavaPersistence API): riguarda persistenza tramite annotazioni
    - Object/relational mapping di metadati, linguaggio di query, ...
- Principi engine JPA
  - Pluggable (sostituibile)
  - Eseguibile al di fuori di ambiente di run-time EJB 3.0
- Hibernate implementa JPA

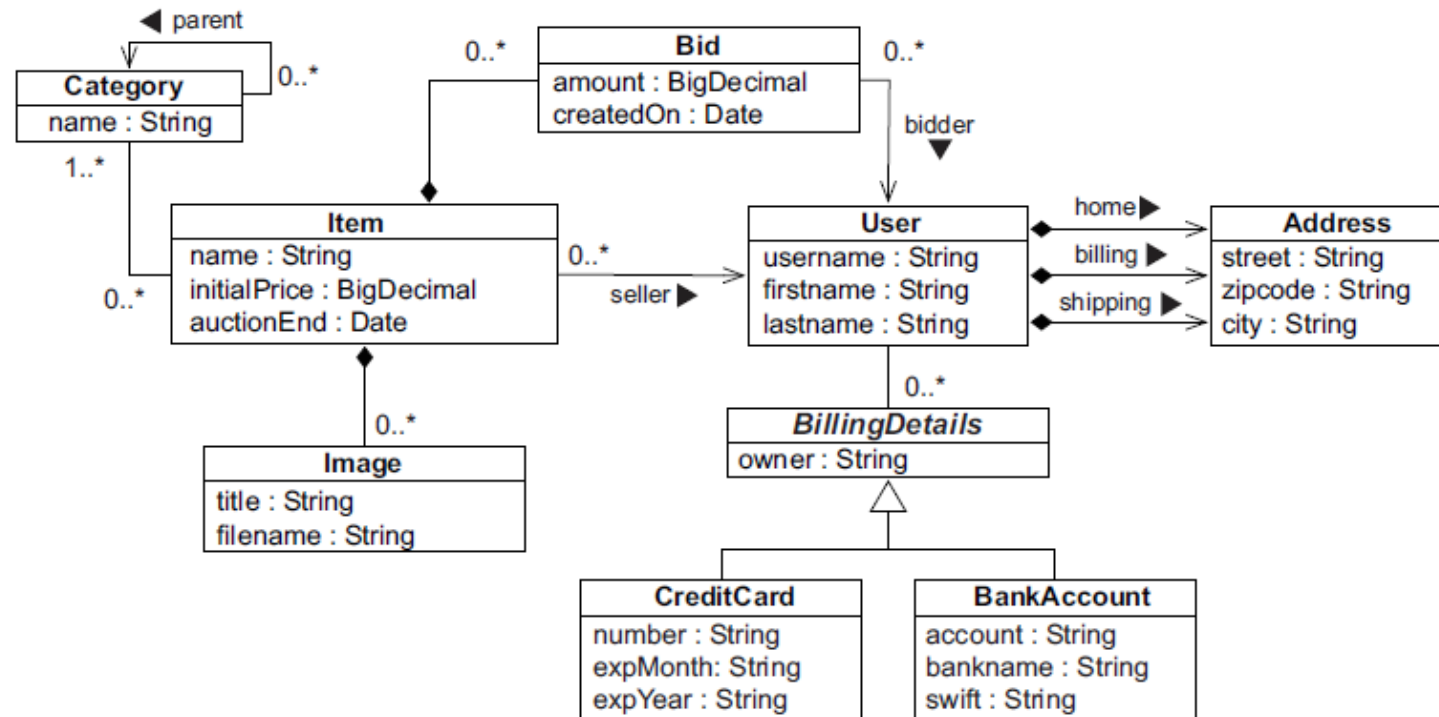
# Hibernate Nativo

- Conosciuto come Hibernate 3.2.x o Hibernate 4.x.x
- Servizio di base per la persistenza
  - API native
  - Mapping dei metadati tramite file XML
- Linguaggio di query HQL
- Centinaia di opzioni e funzioni disponibili
- Indipendente da qualsiasi framework o ambiente di run-time
- Funziona in ogni J2EE application server, in applicazioni Swing, in semplici servlet container, ecc.

# JPA

- Specifica Java per accedere, rendere persistente e gestire dati tra oggetti/classi Java e un DB relazionale
- Specifica e non un prodotto ha un insieme di interfacce e richiede una implementazione
- Implementazioni
  - Hibernate
  - TopLink
  - OpenJPA
- Mapping viene specificato tramite annotazioni oppure tramite file XML
- Definisce un EntityManager API per effettuare query e transazioni
- JPA definisce un object-level query language JPQL
- Ultima versione della specifica JPA 2.1

# Esempio (Caveat Emptor)



# Classe persistente

- Costruttore di default
- Per le proprietà coppia getter/setter
- Per le collezioni utilizzare il tipo dell'interfaccia

```
public class User implements java.io.Serializable {  
    private Long id;  
    .....  
    private String username;  
    private String password;  
    private Address homeAddress;  
    private Set<Role> roles = new HashSet<Role>();  
    .....  
}
```

dichiarazione di Serializable

dichiarazione di proprietà

# Classe persistente

```
public User() {}
```

 ← costruttore classe senza argomenti

```
public String getUsername() {  
    return username;  
}
```

```
public void setUsername(String username) {  
    this.username = username;  
}
```

```
public String getPassword() {  
    return password;  
}
```

```
public void setPassword(String password) {  
    this.password = password;  
}
```

```
public MonetaryAmount calcShippingCosts(Address fromLocation) {  
    .....  
} ← metodi di business
```

metodi accessori  
di proprietà



# Mapping: annotazione

```
@Entity
```

```
@Table(name = "USERS")
```

```
public class User implements java.io.Serializable {
```

```
.....
```

```
}
```

# Mapping: proprietà

```
@Column(name = "FIRSTNAME", nullable = false, length = 255)
```

```
private String firstname;
```

Type	Name	Description	Default
String	name	(Optional) The name of the column.	The property or field name.
boolean	unique	(Optional) Whether the column is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint corresponds to only a single column. This constraint applies in addition to any constraint entailed by primary key mapping and to constraints specified at the table level.	false
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL to create a column of the inferred type.
String	table	(Optional) The name of the table that contains the column. If absent the column is assumed to be in the primary table for the mapped object.	Column is in primary table.
int	length	(Optional) The column length. (Applies only if a string-valued column is used.)	255
int	precision	(Optional) The precision for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0 (Value must be set by developer.)
int	scale	(Optional) The scale for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0

# Mapping: id

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "USER\_ID", nullable = false)

private Long id;

→ Tipo della chiave primaria

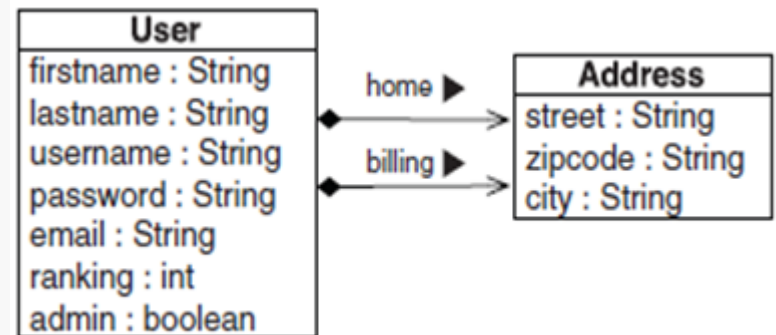
- Qualsiasi tipo primitivo oppure un tipo Wrapper
- Oppure `java.lang.String`, `java.util.Date`,  
`java.sql.Date`, `java.math.BigDecimal`,  
`java.math.BigInteger`

→ GeneratedValue specifica algoritmo di generazione per PK

- SEQUENCE e IDENTITY specificano l'uso di una sequence o identity del DB (non sono portabili su tutti i DB)
- AUTO persistence provider deve adottare una particolare strategia a seconda del DB

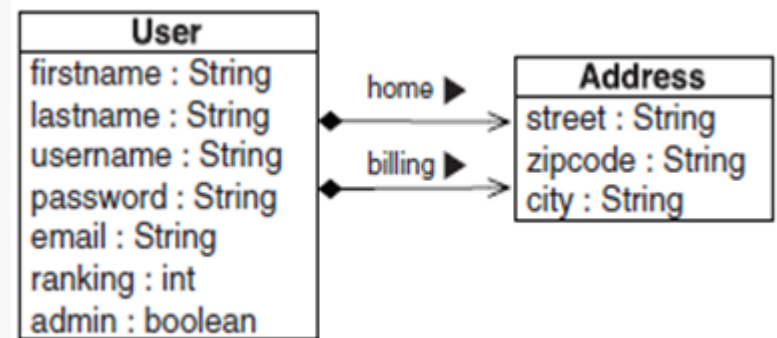
# Mapping: componenti

```
public class User implements java.io.Serializable {  
    private Long id;  
    private String firstname;  
    private String lastname;  
    private String username;  
    private String password;  
    private String email;  
    private int ranking;  
    private Address homeAddress;  
    private Address billingAddress;  
    .....  
}
```



# Mapping: componenti

```
public class Address implements java.io.Serializable {  
  
    private String street;  
  
    private String zipcode;  
  
    private String city;  
  
  
    public Address() {}  
  
    public Address(String street, String zipcode, String city) {  
  
        this.street = street;  
  
        this.zipcode = zipcode;  
  
        this.city = city;  
  
    }  
  
    .....  
  
}
```



# Mapping: componenti

@Embedded

```
@AttributeOverrides({
    @AttributeOverride(name = "street",
        column = @Column(name = "HOME_STREET")),
    @AttributeOverride(name = "zipcode",
        column = @Column(name = "HOME_ZIPCODE")),
    @AttributeOverride(name = "city",
        column = @Column(name = "HOME_CITY")) })

private Address homeAddress;
```

@Embedded

```
@AttributeOverrides({
    @AttributeOverride(name = "street",
        column = @Column(name = "BILLING_STREET")),
    @AttributeOverride(name = "zipcode", column = @Column(name = "BILLING_ZIPCODE")),
    @AttributeOverride(name = "city", column = @Column(name = "BILLING_CITY")) })

private Address billingAddress;
```

<< Table >> USERS	
FIRSTNAME LASTNAME USERNAME PASSWORD EMAIL ...	
HOME_STREET HOME_ZIPCODE HOME_CITY	Component Columns
BILLING_STREET BILLING_ZIPCODE BILLING_CITY	Component Columns

# Mapping: componenti

@Embeddable


```
public class Address implements java.io.Serializable {
```

```
    @OneToOne
```

```
    @JoinColumn(name="USER_ID")
```

```
    private User user;
```

Possono essere sovrascritte  
usando @AttributeOverrides  
nella classe Embedded



```
    @Column(name = "STREET", length = 255, nullable = false)
```

```
    private String street;
```

```
    @Column(name = "ZIPCODE", length = 16, nullable = false)
```

```
    private String zipcode;
```

```
    @Column(name = "CITY", length = 255, nullable = false)
```

```
    private String city;
```

```
.....
```

```
}
```

JPA

# Mapping: associazioni

- Associazioni tra entità singole
  - Uno-a-uno, molti-a-uno
- Associazioni tra collezioni di entità
  - Uno-a-molti, molti-a-molti



# Mapping: associazione 1-a-1

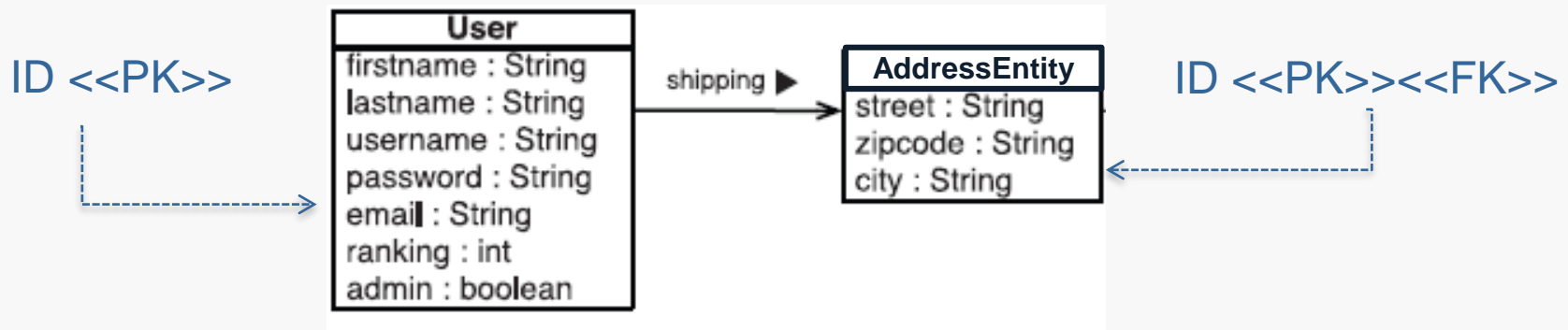
## → Condivisa

- Condivisione della chiave primaria
- Difficoltà principale è garantire che alle istanze associate viene assegnato lo stesso valore di chiave primaria quando gli oggetti vengono salvati
- Association bidirectional

## → foreign key associations

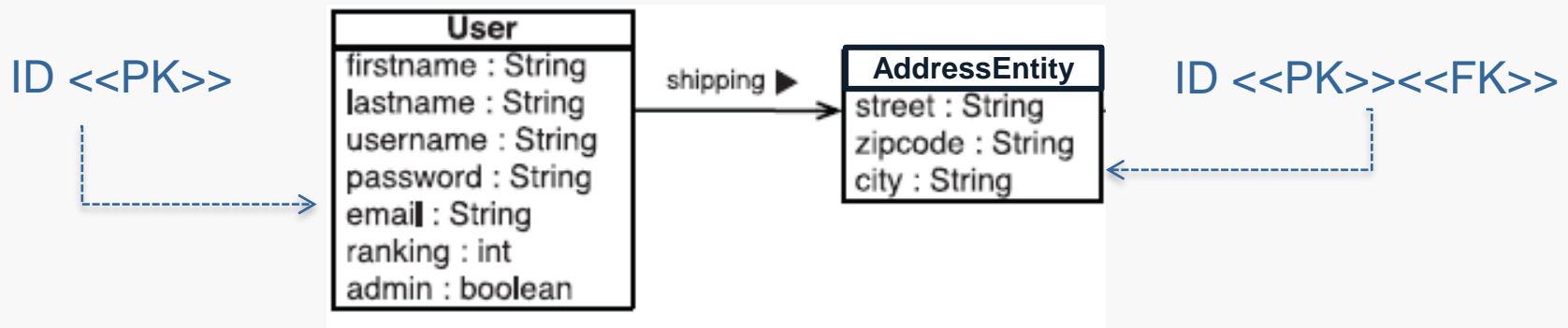
- Una tabella ha una chiave esterna che fa riferimento alla chiave primaria della tabella associata
- Source e Target della chiave esterna può essere anche la stessa tabella chiamato self-referencing relationship

# Mapping: associazione 1-a-1 (condivisa)



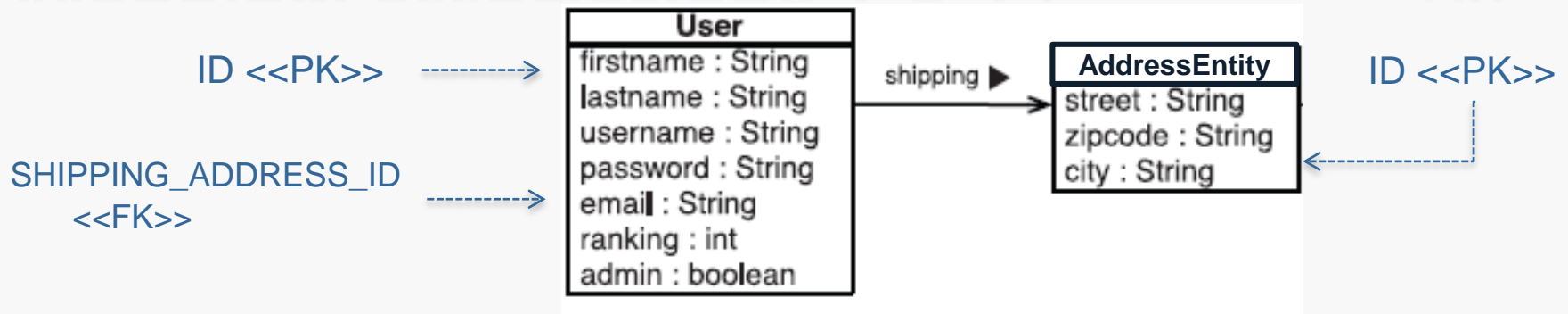
```
public class User implements java.io.Serializable {  
    private Long id;  
  
    .....  
    @OneToOne  
    @PrimaryKeyJoinColumn  
    private AddressEntity shippingAddress;  
  
    .....  
}
```

# Mapping: associazione 1-a-1 (condivisa)



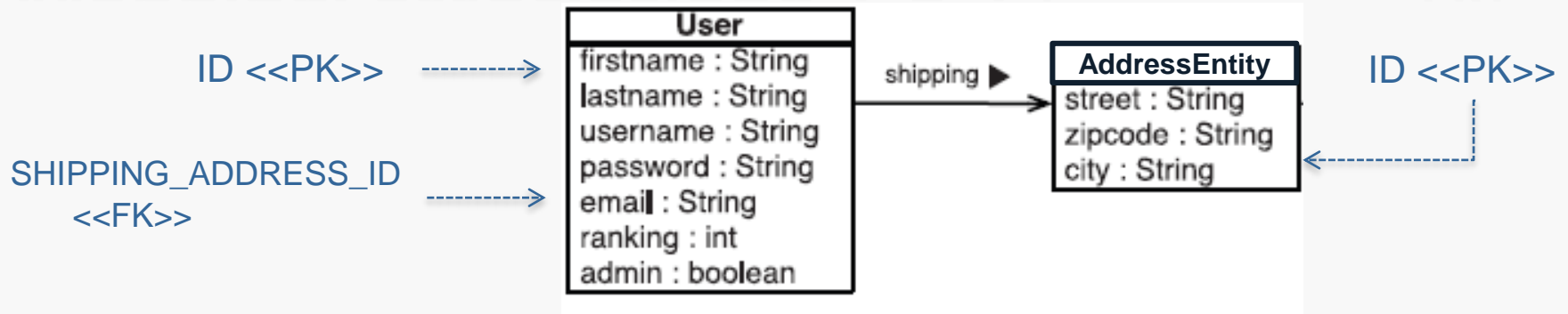
```
public class AddressEntity implements java.io.Serializable {
    @Id @GeneratedValue(generator = "myForeignGenerator")
    @org.hibernate.annotations.GenericGenerator(name = "myForeignGenerator",
        strategy = "foreign",
        parameters = @org.hibernate.annotations.Parameter(
            name = "property", value = "user")
    )
    @Column(name = "ADDRESS_ID")
    private User user;
    .....
}
```

# Mapping: associazione 1-a-1 (associazione FK)



```
public class User implements java.io.Serializable {  
    private Long id;  
    .....  
    @OneToOne(cascade = { CascadeType.PERSIST,  
                          CascadeType.MERGE })  
    @JoinColumn(name="SHIPPING_ADDRESS_ID")  
    private AddressEntity shippingAddress;  
    .....  
}
```

# Mapping: associazione 1-a-1 (associazione FK)



@Entity

@Table(name = "ADDRESSES")

```
public class AddressEntity implements java.io.Serializable {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "ADDRESS_ID", nullable = false)
```

```
    private Long id;
```

```
    @Column(name = "STREET", length = 255)
```

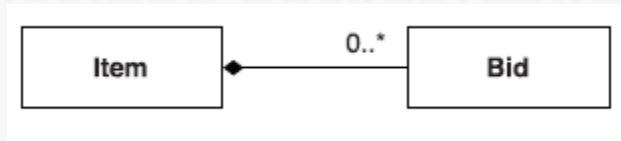
```
    .....
```

```
    @OneToOne(mappedBy = "shippingAddress")
```

```
    private User user;
```

```
    .....
```

# Mapping: associazione parent/children (multi-a-1)



@Entity

@Table(name = "BIDS")

```
public class Bid implements java.io.Serializable {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "BID_ID", nullable = false)
```

```
    private Long id;
```

```
    @ManyToOne
```

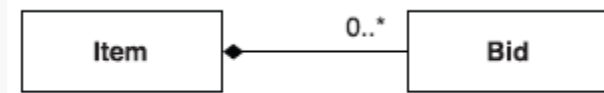
```
    @JoinColumn(name = "ITEM_ID", nullable = false, updatable = false,
                insertable = false)
```

```
    private Item item;
```

```
    .....
```

```
}
```

# Mapping: associazione parent/children (multi-a-1)



Associazione bidirezionale

```
@Entity
```

```
@Table(name = "ITEMS")
```

```
public class Item implements java.io.Serializable {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "ITEM_ID", nullable = false)
```

```
    private Long id;
```

```
    @OneToMany(mappedBy = "item", cascade = CascadeType.ALL)
```

```
    private Set<Bid> bids = new HashSet<Bid>();
```

```
    .....
}
```

## Mapping: associazione parent/children (1-a-molti)

- L'attributo `mappedBy` comunica a JPA che la collezione è una immagine `mirror` della associazione `@ManyToOne` ovvero è bidirezionale (Hibernate `inverse=true`)
- Senza l'attributo `mappedBy` (Hibernate `inverse=false`), JPA prova ad eseguire due SQL statements, entrambi per aggiornare la stessa foreign key quando vengono manipolati i link tra le due istanze
- Specificando `mappedBy`, viene detto esplicitamente a JPA quale è il link finale che non deve sincronizzare con il DB



# Mapping: associazione parent/children (1-a-molti)

Persistenza transitiva

```
Item newItem = new Item();  
Bid newBid = new Bid();  
newItem.addBid(newBid);  
em.persist(newItem);  
em.persist(newBid);
```

Doppio salvataggio

```
Item newItem = new Item();  
Bid newBid = new Bid();  
newItem.addBid(newBid);  
em.persist(newItem);
```

```
@OneToMany(mappedBy = "item",  
    cascade = {CascadeType.PERSIST,  
                CascadeType.MERGE})
```

Singolo salvataggio

# Mapping: associazione parent/children (1-a-molti)

## Cancellazione in cascata

```
Item anItem = ..... // load an item
```

```
for (Iterator<Bid> it = anItem.getBids().iterator(); it.hasNext();) {
```

```
    Bid bid = it.next();
```

```
    it.remove(); // remove reference from collection
```

```
    em.remove(bid); // delete it from the database
```

```
}
```

```
em.remove(anItem); // finally, delete the item
```

multipla cancellazione

```
Item anItem = ..... // load an item
```

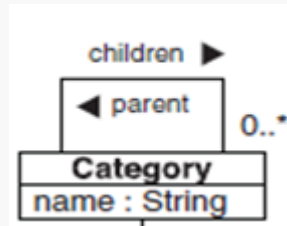
```
em.remove(anItem);
```

singola cancellazione

```
@OneToMany(mappedBy = "item", cascade =  
{CascadeType.PERSIST, CascadeType.MERGE,  
  CascadeType.REMOVE})
```

# Mapping: associazione parent/children

Esempio: Category



```
@Entity
```

```
@Table(name = "CATEGORIES")
```

```
public class Category implements java.io.Serializable {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "CATEGORY_ID", nullable = false)
```

```
    private Long id;
```

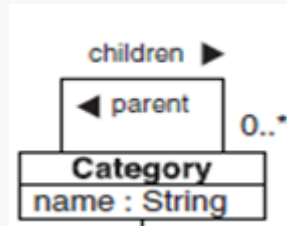
```
    @Column(name = "CATEGORY_NAME", nullable = false, length = 255)
```

```
    private String name;
```

```
    .....
```

# Mapping: associazione parent/children

Esempio: Category



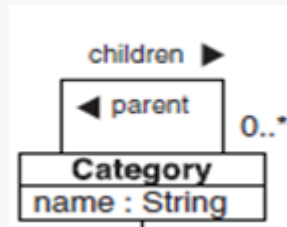
@JsonIgnore

```
@OneToMany(mappedBy = "parentCategory", cascade = {
    CascadeType.PERSIST})
@OrderBy(clause = "name asc")
private Set<Category> childCategories = new HashSet<Category>();
```

.....

# Mapping: associazione parent/children

Esempio: Category



```
@ManyToOne
```

```
@JoinColumn(name = "PARENT_CATEGORY_ID", nullable = true)
```

```
private Category parentCategory;
```

```
@Temporal(TemporalType.DATE)
```

```
@Column(columnDefinition = "timestamp", name = "CREATED",  
        nullable = false, updatable = false)
```

```
private Date created = new Date();
```

# Mapping: associazione 1-a-molti

- `java.util.Set` inizializzata con `java.util.HashSet`
  - L'ordine degli elementi non si conserva
  - Duplicati non ammessi
  - Collezione persistente più comune
- `java.util.SortedSet` inizializzata con `java.util.TreeSet`
  - Attributo sort impostato su un comparatore o su un ordinamento naturale
- `java.util.List` inizializzata con `java.util.ArrayList`
  - conserva la posizione di ciascun elemento con un indice di colonna aggiuntiva nella collection.
- `java.util.Collection` inizializzata con `java.util.ArrayList`
  - Ammessi duplicati e non viene conservato l'ordinamento degli elementi

# Mapping: associazione 1-a-molti

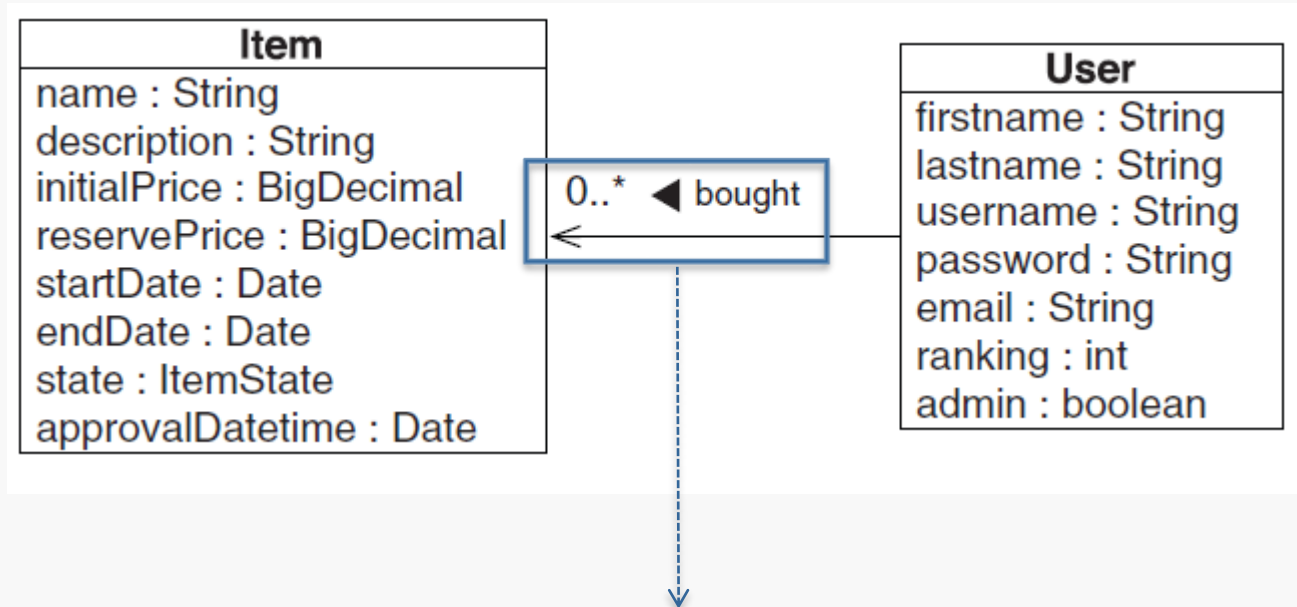
- `java.util.Map` inizializzata con `java.util.HashMap`
  - Preserva la chiave e la coppia di valori
- `java.util.SortedMap` inizializzata con `java.util.TreeMap`
  - Attributo `sort` impostato su un comparatore o su un ordinamento naturale
- Arrays sono supportati da Hibernate e non da JPA con
  - `<primitive-array>` per tipi primitivi Java
  - `<array>` per tutto il resto
  - Raramente utilizzati
  - Si perde il caricamento lazy

# Mapping: associazione 1-a-molti

```
@OneToMany(mappedBy = "parentCategory",  
            cascade = { CascadeType.PERSIST })  
@OrderBy("name asc")  
private Set<Category> childCategories =  
            new HashSet<Category>();
```

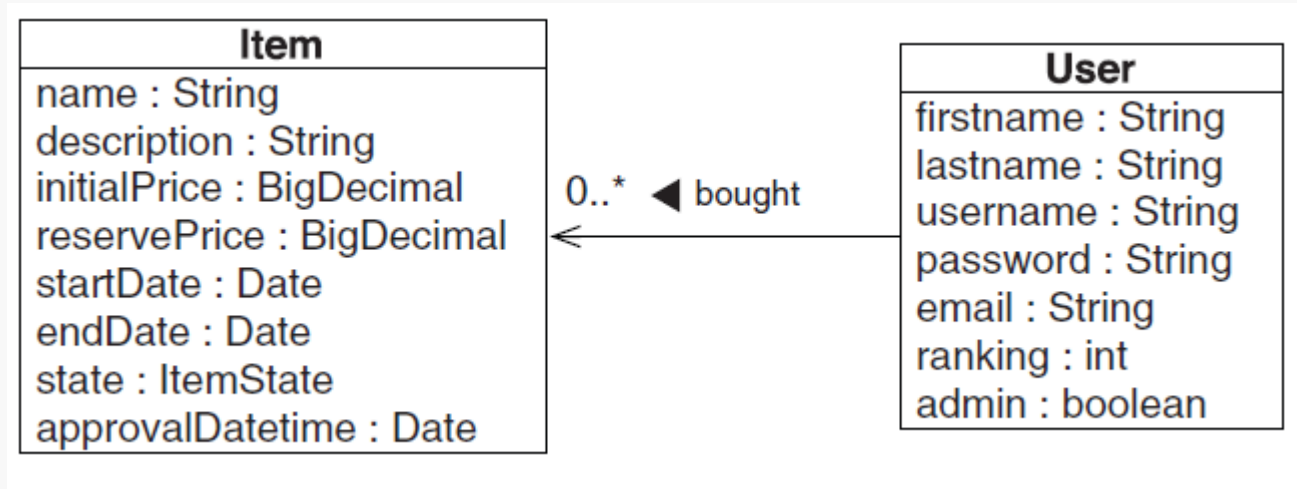


# Mapping: associazione 1-a-molti con join table

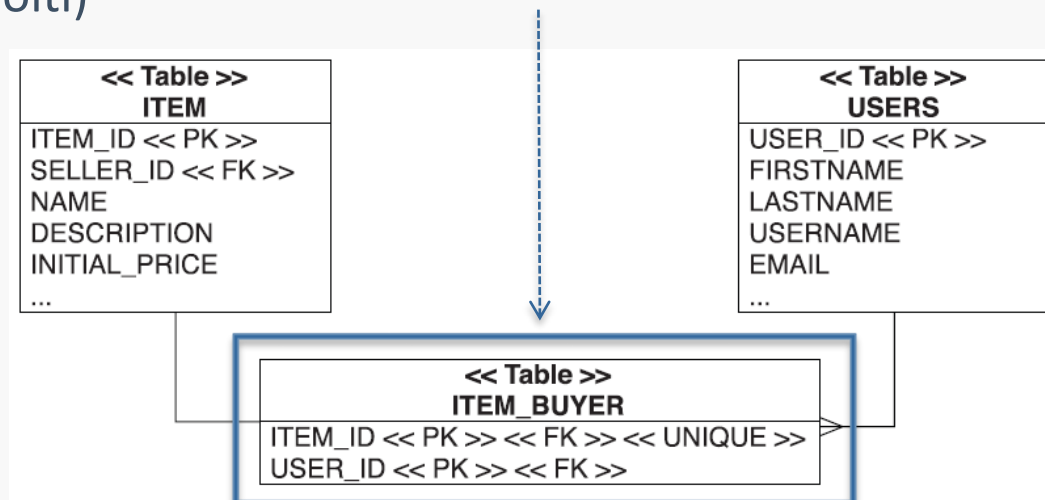


- Nel mondo O.O. non ha conseguenze sul modello
- Nel modello relazionale **SI** ovvero la FK (BUYER\_ID) dentro tabella ITEM è NULLABLE

# Mapping: associazione 1-a-molti con join table



- Non desiderabile si inserisce tabella intermedia con relazione uno-a-uno (o uno-a-molti)



# Mapping: associazione 1-a-molti con join table

```
@OneToMany(mappedBy = "buyer")
```

```
private Set<Item> boughtItems = new HashSet<Item>();
```

← User

```
@ManyToOne
```

```
@JoinTable(name = "ITEMS_BUYER",
```

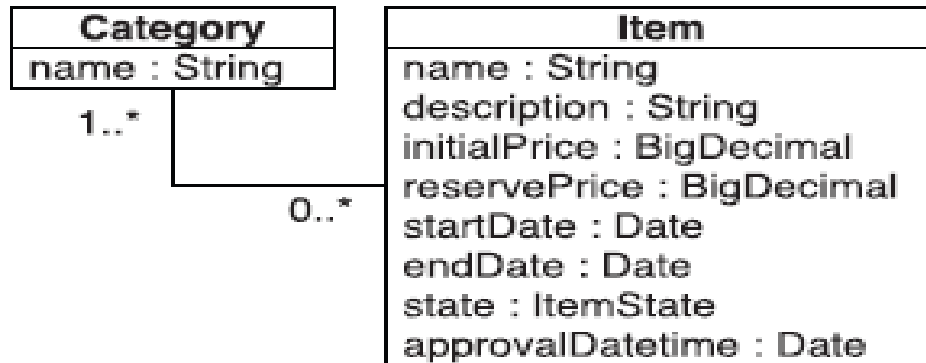
```
    joinColumns = { @JoinColumn(name = "ITEM_ID") },
```

```
    inverseJoinColumns = { @JoinColumn(name = "USER_ID") })
```

```
private User buyer;
```

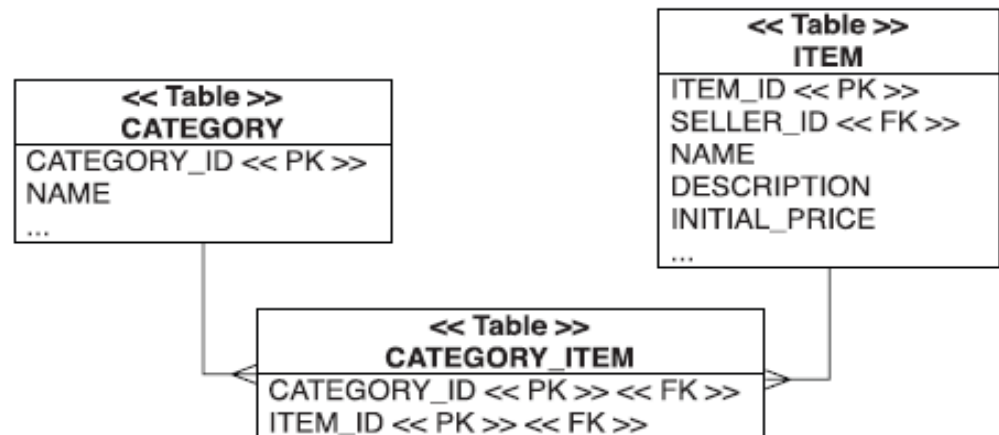
← Item

# Mapping: associazione multi-a-molti



←----- Modello oggetti

Modello relazionale ----->



# Mapping: associazione multi-a-molti

- Mapping mono direzionale ovvero presente solo ad un lato dell'associazione

```
@ManyToMany(fetch=FetchType.EAGER)  
@JoinTable(name = "CATEGORIZED_ITEMS",  
    joinColumns = { @JoinColumn(name = "ITEM_ID") },  
    inverseJoinColumns = { @JoinColumn(name = "CATEGORY_ID") })  
private Set<Category> categories = new HashSet<Category>();
```

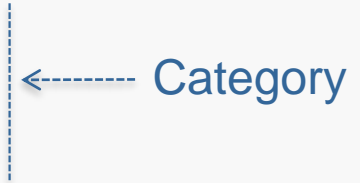
←----- Item

# Mapping: associazione multi-a-molti

- Se si vuole fare mapping bidirezionale si deve indicare uno dei due come *inverse* ovvero che l'aggiornamento avviene dall'altra parte

```
@ManyToMany(mappedBy = "categories")
```

```
private Set<Item> items = new HashSet<Item>();
```



```
@ManyToMany(fetch=FetchType.EAGER)
```

```
@JoinTable(name = "CATEGORIZED_ITEMS",
```

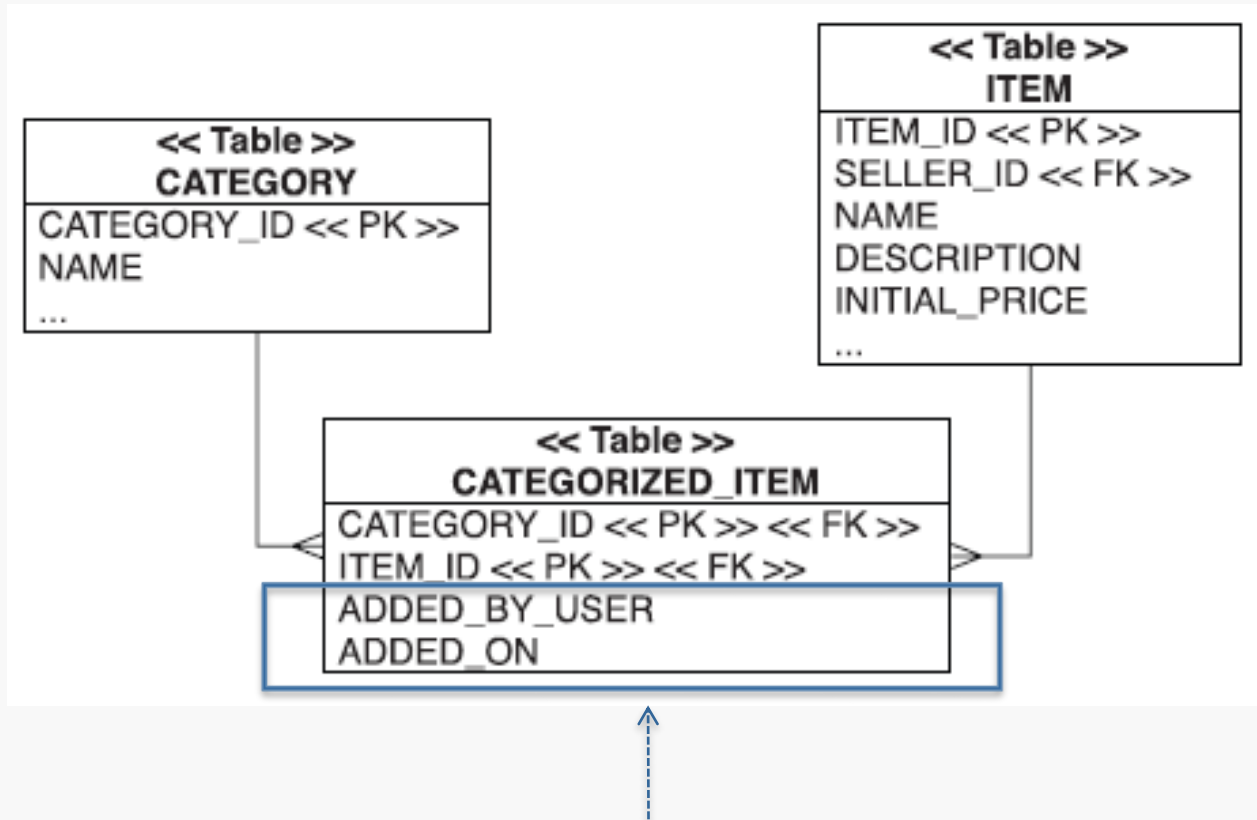
```
    joinColumns = { @JoinColumn(name = "ITEM_ID") },
```

```
    inverseJoinColumns = { @JoinColumn(name = "CATEGORY_ID") })
```

```
private Set<Category> categories = new HashSet<Category>();
```



# Mapping: associazione multi-a-molti



aggiungere colonne alla relazione

# Mapping: associazione multi-a-molti

```
@Entity
@Table(name = "CATEGORIZED_ITEM")
public class CategorizedItem {
    @Embeddable
    public static class Id implements Serializable {
        @Column(name = "CATEGORY_ID")
        private Long categoryId;
        @Column(name = "ITEM_ID")
        private Long itemId;
        public Id() {}
        public Id(Long categoryId, Long itemId) {
            this.categoryId = categoryId;
            this.itemId = itemId;
        }
        .....
    }
}
```



# Mapping: associazione multi-a-molti

```
public boolean equals(Object o) {  
    if (o != null && o instanceof Id) {  
        Id that = (Id) o;  
        return this.categoryId.equals(that.categoryId)  
            && this.itemId.equals(that.itemId);  
    } else {  
        return false;  
    }  
}  
  
public int hashCode() {  
    return categoryId.hashCode() + itemId.hashCode();  
}  
}
```

.....

# Mapping: associazione multi-a-molti

```
@EmbeddedId
private Id id = new Id();
@Column(name = "ADDED_BY_USER")
private String username;
@Column(name = "ADDED_ON")
private Date dateAdded = new Date();
@ManyToOne
@JoinColumn(name = "ITEM_ID", insertable = false, updatable = false)
private Item item;
@ManyToOne
@JoinColumn(name = "CATEGORY_ID", insertable = false, updatable = false)
private Category category;
public CategorizedItem() {}
public CategorizedItem(String username, Category category, Item item) {
    // Set fields
    .....
}
// Getter and setter methods
....
```

# Mapping: associazione multi-a-molti

Category e Item

```
@OneToMany(mappedBy = "category")
```

```
private Set<CategorizedItem> categorizedItems =new HashSet<CategorizedItem>();
```

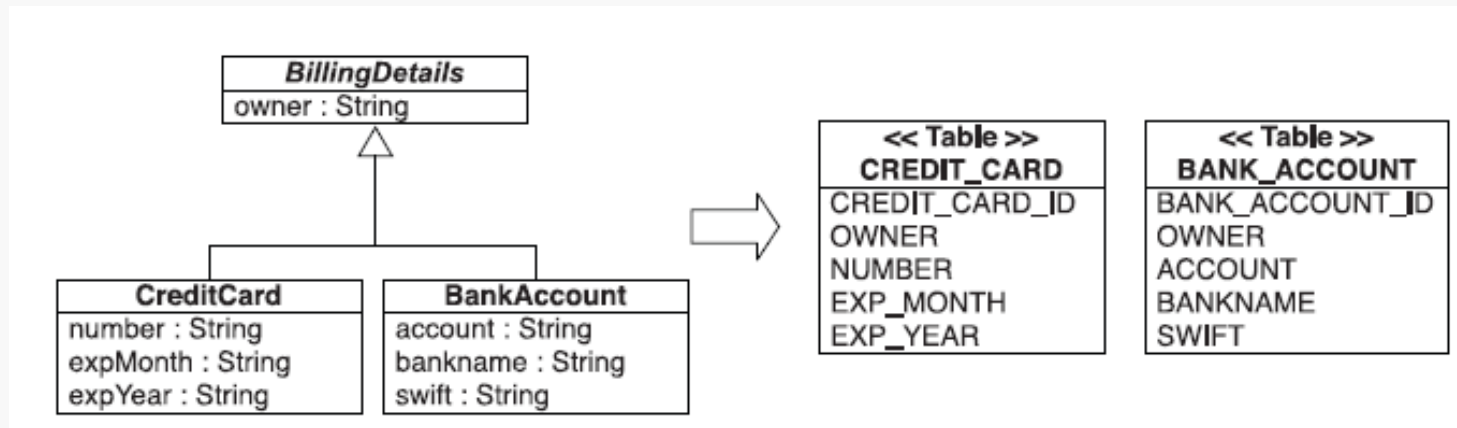
# Mapping: Ereditarietà

Quattro tipi approcci

- Tabella per classe concreta con polimorfismo implicito
  - Non vi è mapping esplicito dell'ereditarietà e comportamento polimorfico default a runtime
- Tabella per classe concreta
  - Scartati il polimorfismo e relazioni di ereditarietà dallo schema SQL
- Tabella per gerarchia di classe
  - Abilita il polimorfismo denormalizzando lo schema SQL
  - Utilizza una colonna discriminante che mantiene le informazioni sul tipo
- Tabella per sottoclasse
  - L'ereditarietà viene espressa mediante una relazione di *has a* (ovvero foreign key)

# Mapping: Ereditarietà

Tabella per classe concreta con polimorfismo implicito



- Sono viste come due tabelle separate
- Problema principale che non supporta le associazioni polimorfiche (ovvero no foreign key)
- Query polimorfiche sono problematiche

# Mapping: Ereditarietà

Tabella per classe concreta con polimorfismo implicito

@MappedSuperclass

```
public abstract class BillingDetails {  
    @Column(name = "OWNER", nullable = false)  
    private String owner;  
    // ...  
}
```

←----- una superclasse astratta o un'interfaccia

Proprietà ereditata  
da tutte le  
classi concrete

@Entity

@Table(name = "CREDIT\_CARD")

@AttributeOverride(name = "owner", column = @Column(name = "CC\_OWNER", nullable = false))

```
public class CreditCard extends BillingDetails {  
    @Id @GeneratedValue  
    @Column(name = "CREDIT_CARD_ID")  
    private Long id = null;  
    @Column(name = "NUMBER", nullable = false)  
    private String number;  
    // ...  
}
```

←----- Una sottoclasse  
concreta è mappata  
nella tabella

// ...

JPA

# Mapping: Ereditarietà

## Tabella per classe concreta con unioni

@Entity

@Inheritance(strategy = InheritanceType.**TABLE\_PER\_CLASS**) ← una superclasse astratta o un'interfaccia

```
public abstract class BillingDetails {  
    @Id  
    @GeneratedValue  
    @Column(name = "BILLING_DETAILS_ID")  
    private Long id = null;  
    @Column(name = "OWNER", nullable = false)  
    private String owner;  
    // ...  
}
```

@Entity

```
@Table(name = "CREDIT_CARD")  
public class CreditCard extends BillingDetails {  
    @Column(name = "NUMBER", nullable = false)  
    private String number;  
    // ...  
}
```

← Una sottoclasse  
concreta è mappata  
nella tabella  
Non si replicano proprietà

# Mapping: Ereditarietà

Tabella per classe concreta con unioni

```
Select BILLING_DETAILS_ID, OWNER, NUMBER, EXP_MONTH, EXP_YEAR, ACCOUNT,  
        BANKNAME, SWIFT CLAZZ_  
  
from ( select BILLING_DETAILS_ID, OWNER, NUMBER, EXP_MONTH, EXP_YEAR,  
            null as ACCOUNT, null as BANKNAME, null as SWIFT, 1 as CLAZZ_  
        from CREDIT_CARD  
        union select BILLING_DETAILS_ID, OWNER, null as NUMBER,  
            null as EXP_MONTH, null as EXP_YEAR, ...  
            ACCOUNT, BANKNAME, SWIFT, 2 as CLAZZ_  
        from BANK_ACCOUNT  
    )
```

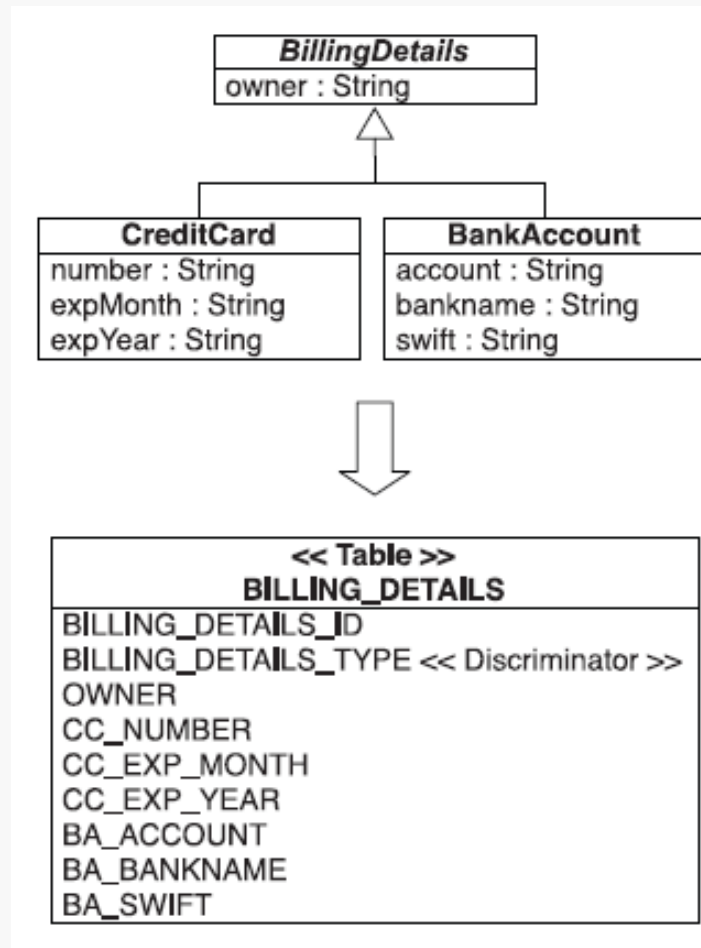
Vantaggi

- Nelle query polimorfiche
- Possibilità di gestire associazioni polimorfiche (esempio mapping tra User e BillingDetails possibile)



# Mapping: Ereditarietà

Tabella per gerarchia di classe



# Mapping: Ereditarietà

Tabella per gerarchia di classe

```
@Entity
@Table(name = "BILLING_DETAILS")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "BILLING_DETAILS_TYPE",
                    discriminatorType = DiscriminatorType.STRING)
public abstract class BillingDetails {
    @Id
    @GeneratedValue
    @Column(name = "BILLING_DETAILS_ID")
    private Long id = null;
    @Column(name = "OWNER", nullable = false)
    private String owner;
    // ...
}
```

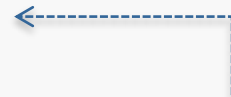


usato per distinguere  
le classi persistenti

# Mapping: Ereditarietà

Tabella per gerarchia di classe

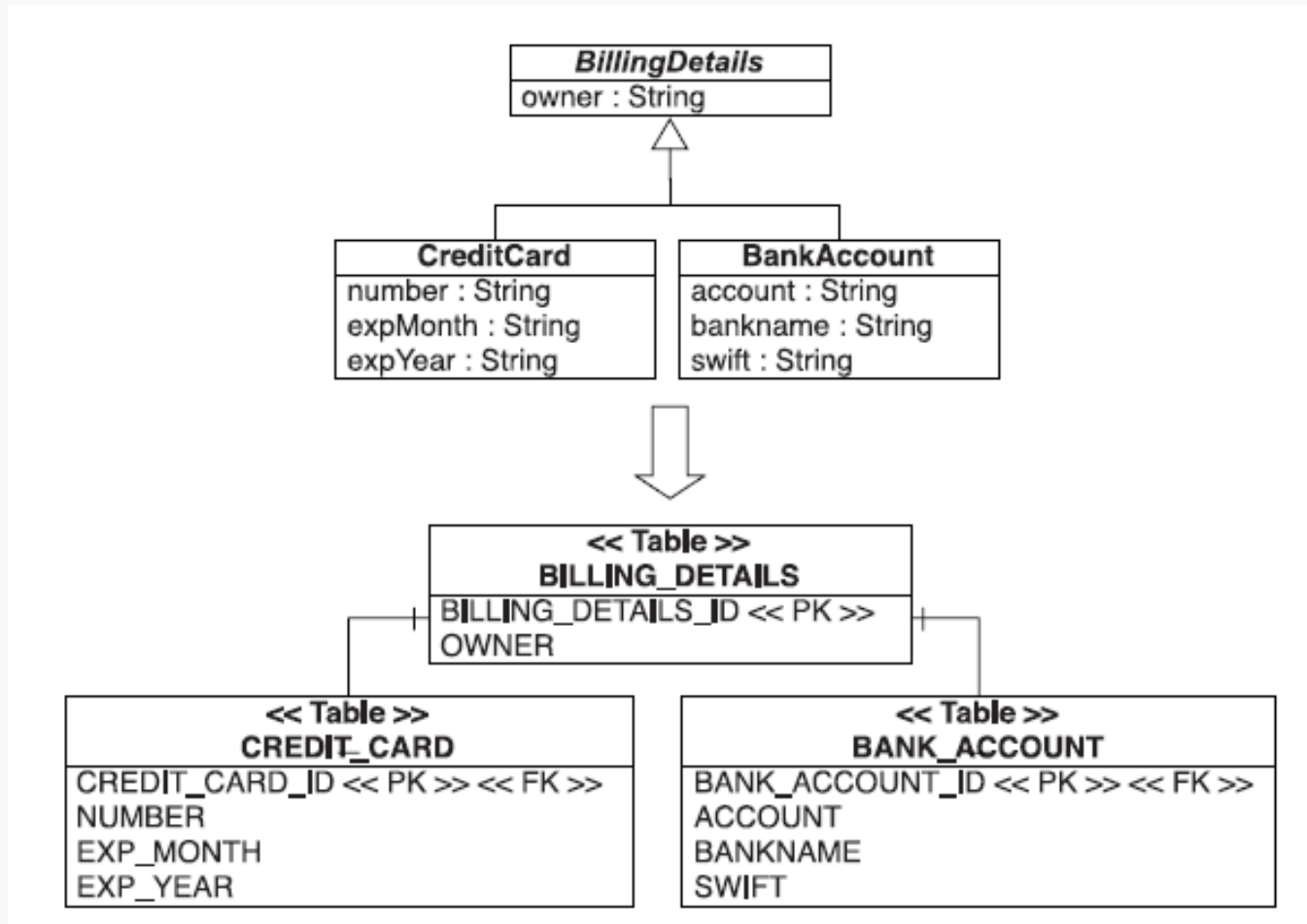
```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
    @Column(name = "CC_NUMBER")
    private String number;
    // ...
}
```



proprietà di una  
sottoclasse sono  
mappate in colonne  
della tabella  
BILLING\_DETAILS

# Mapping: Ereditarietà

Tabella per sottoclasse



# Mapping: Ereditarietà

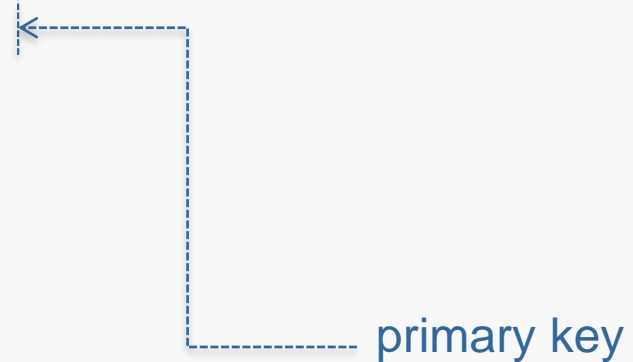
## Tabella per sottoclasse

```
@Entity
@Table(name = "BILLING_DETAILS")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
    @Id
    @GeneratedValue
    @Column(name = "BILLING_DETAILS_ID")
    private Long id = null;
    // ...
}
```

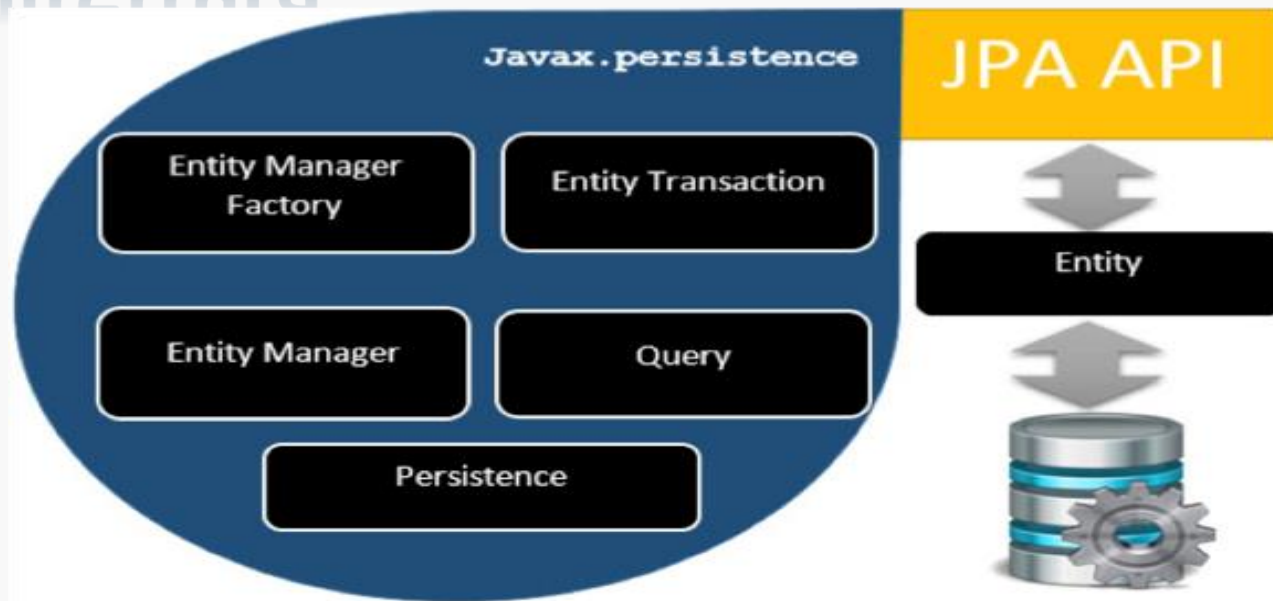
# Mapping: Ereditarietà

Tabella per sottoclasse

```
@Entity
@Table(name = "CREDIT_CARDS")
@PrimaryKeyJoinColumn(name = "CREDIT_CARD_ID")
public class CreditCard {
    //...
}
```



# Architettura



`EntityManagerFactory`: Factory per istanze di `EntityManager`

`EntityManager`: Interfaccia che gestisce operazioni su oggetti. Agisce come factory per istanze di `Query`

`Entity`: Oggetti persistenti “agganciati” al DB

`EntityTransaction`: Relazioni one-to-one con `EntityManager`. Operazioni su `EntityManager` sono gestite da `EntityTransaction`

`Persistence`: Metodi statici per ottenere istanze di `EntityManagerFactory`

`Query`: Interfaccia per eseguire query

# Persistence Unit

Unità logica che racchiude

- Entity Manager Factory e relative Entity Managers con informazioni di configurazione
- Insieme delle classi (entità) gestite dall'Entity Manager della relativa Entity Manager Factory
- Mapping metadati (con annotazioni e/o XML) che specifica mapping classi con tabelle nel database
- E' possibile definire un numero qualsiasi con un nome univoco di persistence unit all'interno ambiente Java EE



# Persistence Unit

- Definito tramite il file `persistence.xml` presente cartella `META-INF`
- Possono far parte di un WAR o un EJB JAR oppure JAR che fa parte di un WAR o un EAR
- Se fa parte di un EJB JAR `persistence.xml` viene posto nella cartella `META-INF`
- Se fa parte di un WAR `persistence.xml` viene posto cartella `WEB-INF/classes/META-INF`
- Se fa parte di un JAR il file deve essere posto
  - WAR: `WEB-INF/lib` cartella
  - EAR: dentro library cartella

# persistence.xml

- Utilizzato per specificare le classi di persistenza incluse nel persistence unit con relativo mapping object/relational
- Scripts per la generazione dello schema
- Entity Manager Factory
- ecc
- Mapping object/relational può avvenire
  - Annotazioni
  - File `orm.xml` presente dentro cartella `META-INF`
  - Uno o più file XML presenti nel classpath e specificati all'interno del file `persistence.xml`
- Elemento root dentro il file `persistence.xml` è `persistence`

# persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                                  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
              version="2.1">
  <persistence-unit name="caveatemptor" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <non-jta-data-source>jdbc/caveatemptor</non-jta-data-source>

    <class>it.univaq.disim.caveatemptor.business.model.AddressEntity</class>
    <class>it.univaq.disim.caveatemptor.business.model.Bid</class>
    <class>it.univaq.disim.caveatemptor.business.model.BillingDetails</class>
    <class>it.univaq.disim.caveatemptor.business.model.Category</class>

    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <property name="hibernate.use_identifier_rollback" value="true"/>
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
      <property name="hibernate.show_sql" value="false"/>
    </properties>
  </persistence-unit>
</persistence>
```

# persistence.xml

→ persistence-unit

- name: definisce il nome della persistence unit

Utilizzato all'interno delle annotazioni

PersistenceContext e PersistenceUnit  
per creare una entity manager factory

- transaction-type: Utilizzato per specificare  
se gli entity manager devono essere JTA o  
RESOURCE\_LOCAL

Se non specificato in un ambiente Java EE default  
JTA, in un ambiente Java SE default  
RESOURCE\_LOCAL

# persistence.xml

- provider
  - Specifica il nome del provider (esempio: `org.hibernate.ejb.HibernatePersistence`)
- jta-data-source, non-jta-data-source
  - Ambienti Java EE `jta-data-source` e `non-jta-data-source` sono utilizzati per specificare il nome JNDI del JTA e/o non-JTA data source
- mapping-file, jar-file, class, exclude-unlisted-classes
  - Insieme classi persistenti che sono gestite da una persistence unit sono definite
    - Classi annotate (`Entity`, `Embeddable`, `MappedSuperclass`, o `Converter`) contenute nella root (a meno che non viene utilizzato il tag `exclude-unlisted-classes`)
    - Uno o più file XML object-relational
    - Uno o più files jar dove verranno cercati le classi
    - Una lista esplicita di classi

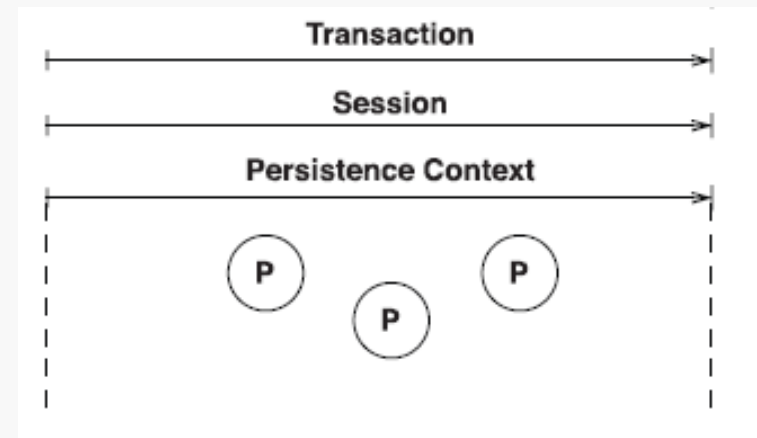
# Come creare la EntityManagerFactory

- provider
  - Specifica nome del provider
    - Esempio: `org.hibernate.ejb.HibernatePersistence`
- `jta-data-source`, `non-jta-data-source`
  - Ambienti Java EE `jta-data-source` e `non-jta-data-source` sono utilizzati per specificare il nome JNDI del JTA e/o non-JTA data source
- `mapping-file`, `jar-file`, `class`, `exclude-unlisted-classes`
  - Insieme classi persistenti che sono gestite da una persistence unit sono definite
    - Classi annotate (`Entity`, `Embeddable`, `MappedSuperclass`, o `Converter`) contenute nella root (a meno che non viene utilizzato `exclude-unlisted-classes`)
    - Uno o più file XML object-relational
    - Uno o più files jar dove verranno cercate classi
    - Una lista esplicita di classi

# Session

## Esempio

```
Item item = new Item();  
item.setName("test");  
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("helloworld");  
EntityManager em = emf.createEntityManager();  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
em.persist(item);  
tx.commit();  
em.close();
```



# Sessione

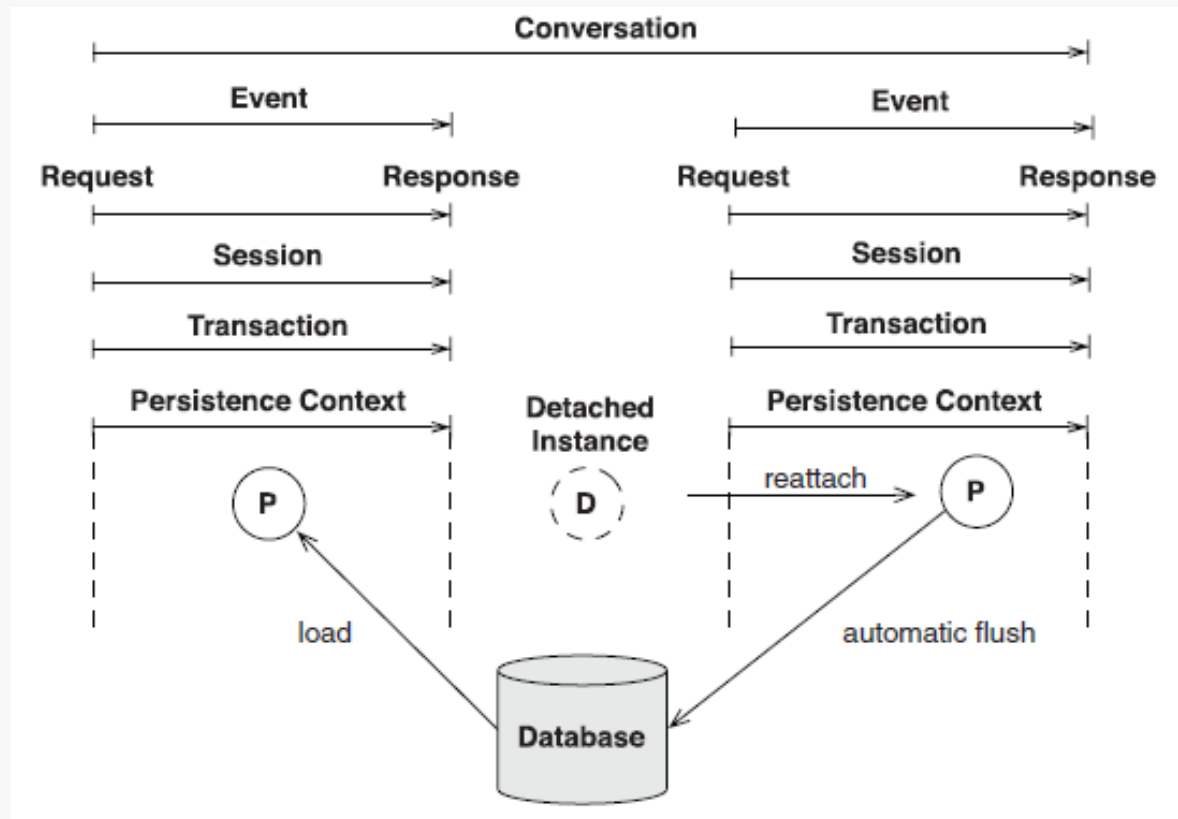
- Oggetto poco costoso e NON thread-safe
- Non stabilisce una connessione verso il DB a meno che non è necessario (ovvero quando si effettua una query)
- Per ridurre al minimo accesso al DB una transazione deve essere più breve possibile
- E' possibile avere diverse transazioni all'interno della stessa sessione



# Sessione

- Non utilizzare l'anti-pattern *session-per-operation*
  - Non aprire e chiudere una sessione per ogni singola operazione sul DB in un singolo thread
- Utilizzare il pattern *session-per-request*
  - Viene aperta una sessione e tutte le operazioni sul DB sono eseguite in tale unità di lavoro
  - Quando il lavoro viene completato la sessione viene svuotata (flush) e chiusa
  - Relazione uno-a-uno tra client-thread-sessione-transazione

# Sessione: scenario tipo web application



Pattern session-per-request

# Sessione: scenario tipo web application

*Quando aprire e chiudere la sessione?*

1. All'inizio/fine della richiesta/risposta (pattern open-session-in-view)

→ Vantaggi

- Semplice realizzazione
- Nella vista si possono navigare oggetti persistenti senza incappare in eccezioni (`LazyInitializationException`)

→ Svantaggi

- Se il ciclo richiesta/risposta è lungo → si mantiene aperta una transazione per lungo tempo (non è desiderabile)
- N+1 select problem in caso di navigazioni in relazioni uno-a-molti con lazy fetching dei dati

# Sessione: scenario tipo web application

2. All'inizio/fine del metodo che *esprime* la logica di business

→ Problema: `LazyInitializationException` se le entità/collezioni sono lazy

– Soluzioni

■ `lazy=false` (rischio recuperare tutto il db)

■ *Tirare su* esplicitamente oggetti che servono specificandoli espressamente nella query

# Persistence context

- Può essere visto come una cache delle istanze delle entità gestite
- All'interno dell'applicazione non viene utilizzata esplicitamente tramite alcuna API
- `EntityManager` di JPA ha un persistence context (`Session` di Hibernate)
- Entità in uno stato persistente e gestite in una unità di lavoro sono “cached” nel persistence context
- Persistence context è utile per
  - Automatic dirty checking e transazioni
  - Cache di primo livello
  - .....

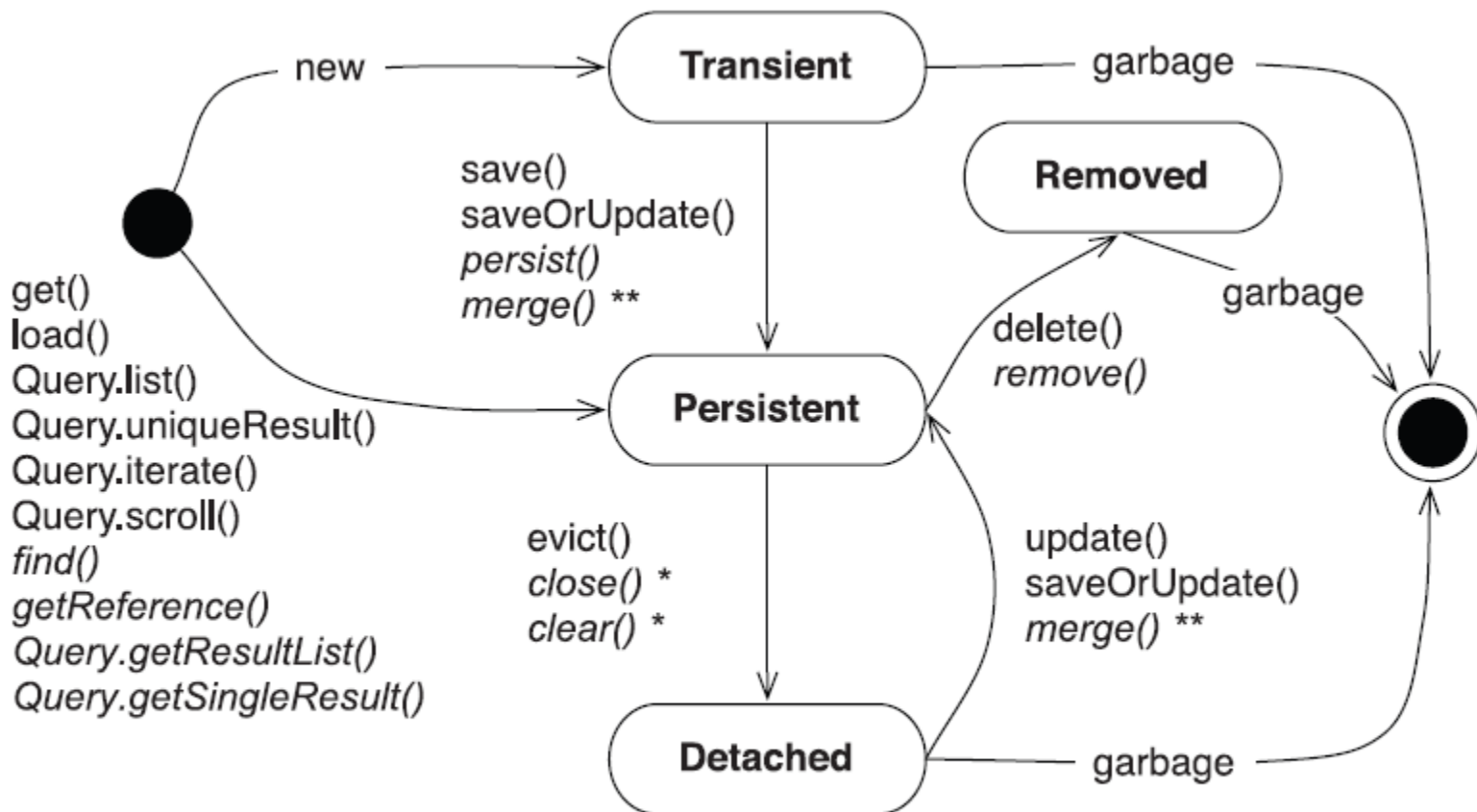
# Automatic dirty checking

- Istanze persistenti sono gestite all'interno di un persistence context e loro stato è sincronizzato con il database alla fine dell'unità di lavoro (esecuzione di SQL INSERT, UPDATE, DELETE)
- ORM hanno una strategia per determinare quando lo stato degli oggetti persistenti è stato modificato dall'applicazione (automatic dirty checking)
- Strategia cerca minimizzare numero di accessi al DB (minimizzare lock sul DB) mantenendo consistente lo stato dell'oggetto rispetto al DB

# Persistence context cache

- Persistence context effettua “cache” istanze entità persistenti
- Ricorda tutte istanze persistenti gestite all’interno di una particolare unità di lavoro
- Automatic dirty checking beneficia di tale cache
- Altro beneficio è una lettura ripetibile per le entità gestite
  - Esempi
    - Lettura entità tramite primary key: Viene controllata se entità presente già nel persistence context. Se si viene utilizzata altrimenti accesso al DB
    - Query effettuato cache degli oggetti
- Gestisce anche reference circolari nel grafo degli oggetti
- Non vi sono conflitti che rappresentano stessa riga del DB in una unità di lavoro

# Ciclo di vita oggetti nella persistenza



\* Hibernate & JPA, affects all instances in the persistence context

\*\* Merging returns a persistent instance, original doesn't change state



# Ciclo di vita oggetti nella persistenza: Stati

## → **Transient**

- Oggetti istanziati mediante l'operatore `new` non sono persistenti
- Non sono associati con alcuna riga in una tabella del DB
- Perdono il loro stato non appena non sono più referenziati

## → **Persistent**

- Istanza di entità con una identità nel DB (ovvero ha il valore di una chiave primaria settato nel suo identificatore)
- Istanze persistenti sono associate sempre ad un **contesto persistente**
- All'interno di tale contesto qualsiasi modifica all'istanza si *riflette* sulla riga della tabella

# Ciclo di vita oggetti nella persistenza: Stati

## → **Removed**

- Un oggetto è nello stato removed se è stato *schedulato* per la cancellazione alla fine di una unità di lavoro

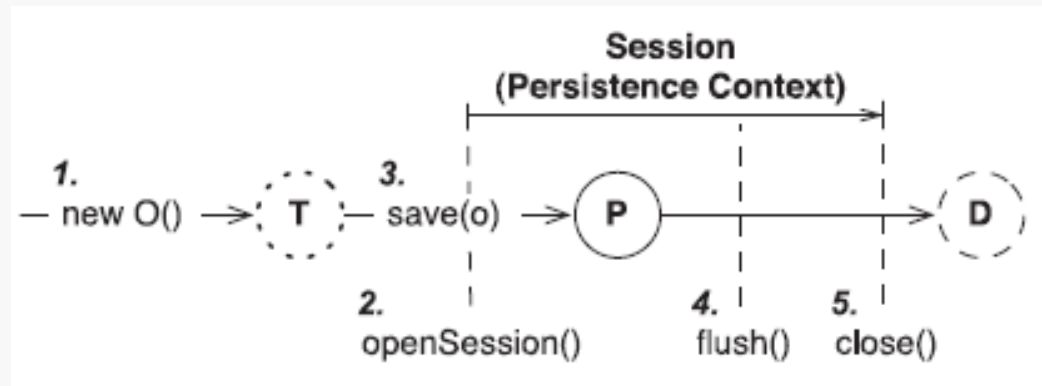
## → **Detached**

- Si trova nello stato detached dopo che dallo stato persistente è stato chiuso il contesto persistente
- Qualsiasi modifica allo stato dell'oggetto non ha effetto sulla riga della tabella

# Rendere un oggetto persistente

```
Item item = new Item(); ← 1
item.setName("Playstation3 incl. all accessories");
item.setEndDate( ... );

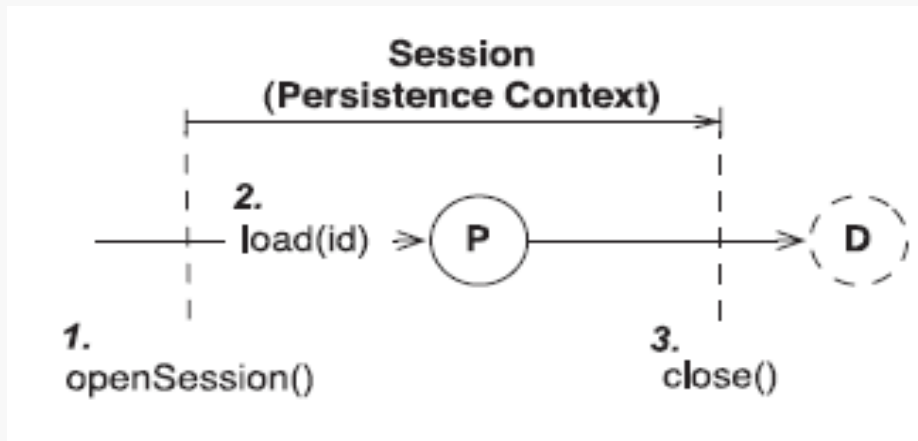
EntityManagerFactory emf = Persistence.createEntityManagerFactory("caveatemptor");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction(); ← 2
tx.begin();
em.persist(item); ← 3
tx.commit(); ← 4
em.close(); ← 5
```



# Recuperare un oggetto persistente

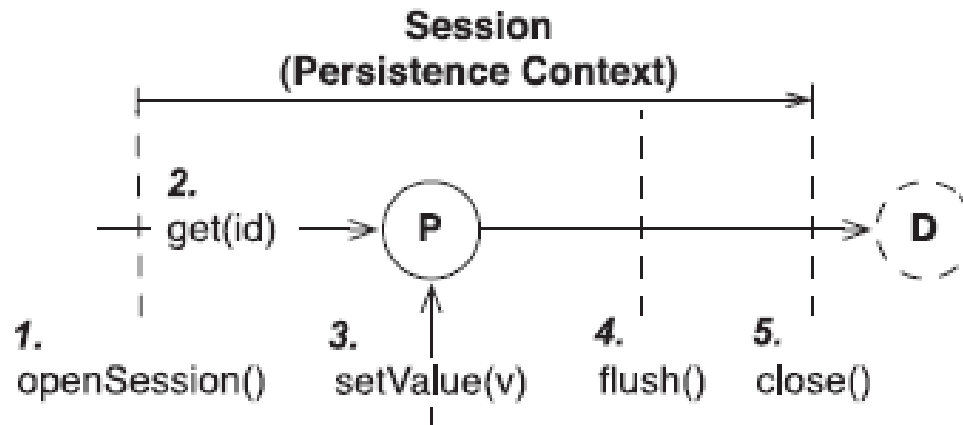
```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("caveatemptor");
EntityManager em = emf.createEntityManager();
EntityManagerTransaction tx = em.getTransaction();
tx.begin();
Item item = em.find(Item.class, new Long(1234));
tx.commit();
em.close();
```

→ find ritorna null se non esiste l'oggetto



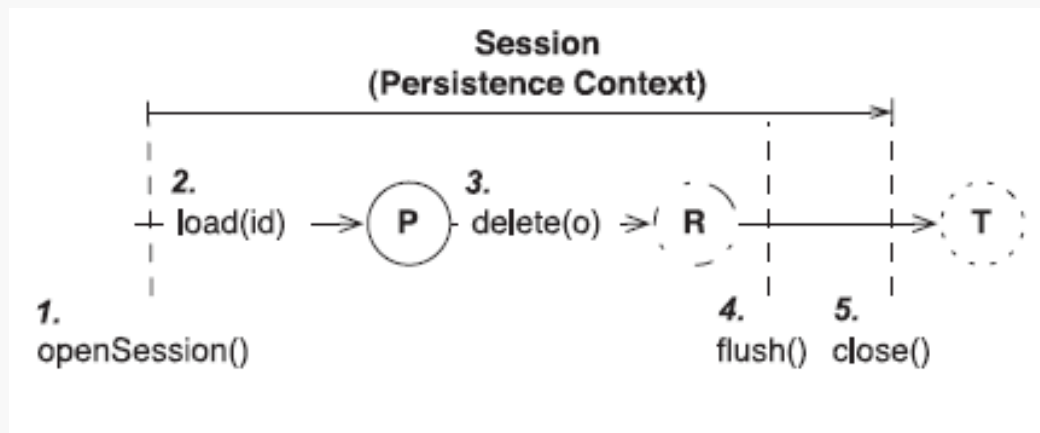
# Modificare un oggetto persistente

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("caveatemptor");  
EntityManager em = emf.createEntityManager();  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
Item item = em.find(Item.class, new Long(1234));  
item.setDescription("This Playstation is as good as new!");  
tx.commit();  
em.close();
```



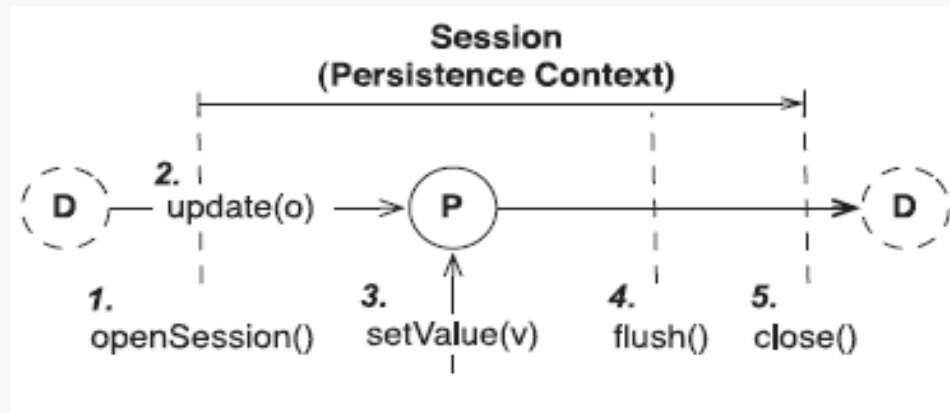
# Rendere un oggetto persistente transiente

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("caveatemptor");  
EntityManager em = emf.createEntityManager();  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
Item item = em.find(Item.class, new Long(1234));  
em.remove(item);  
tx.commit();  
em.close();
```



# Merging istanza detached modificata

```
EntityManager em = emf.createEntityManager();  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
Item item = em.find(Item.class, new Long(1234));  
tx.commit();  
em.close();  
  
item.setDescription(...); // Loaded in previous Session  
  
EntityManager em2 = emf.createEntityManager();  
EntityTransaction tx2 = em2.getTransaction();  
em2.merge(item);  
tx2.commit();  
em2.close();
```



# Metodi di interrogazione dati

## → JPA Query Language (JPA QL)

- Linguaggio simil SQL che permette di astrarsi dal dialetto del DB utilizzato
- Utilizza concetti O.O. per effettuare query

## → Altri metodi

- Query tramite `Criteria`
  - Comprende anche query by example
- Si può sempre utilizzare l'SQL nativo (perdita di portabilità)



# JPA Query Language

## Creazione

```
Query query = entityManager.createQuery("from User");  
  
Query query = entityManager.createQuery("from User u order by u.name asc");  
entityManager.setMaxResults(10);  
entityManager.setFirstResult(40);  
  
String queryString = "from Item i where i.description like '" + search + "'";  
List result = entityManager.createQuery(queryString).list();  
  
String queryString = "from Item item"  
    + " where item.description like :search"  
    + " and item.date > :minDate";  
  
Query q = entityManager.createQuery(queryString)  
    .setParameter("search", searchString)  
    .setParameter("minDate", mDate, TemporalType.DATE);
```

alias

ordinamento

paginazione

parametri

binding parametri

chain

# JPA QL: Named Query

```
em.createNamedQuery("findItemsByDescription")  
    .setParameter("desc", description);
```

← chiamata

```
@NamedQueries({  
    @NamedQuery(name = "findItemsByDescription",  
        query = "select i from Item i where i.description like :desc"  
    ),  
    //...  
})  
@Entity
```

← Annotazioni  
dentro classe  
Item

# JPA QL: Esecuzione

```
Bid maxBid = (Bid) em.createQuery(  
    "select b from Bid b order by b.amount desc")  
    .setMaxResults(1)  
    .getSingleResult();
```

←----- risultato unico

Se ritorna più di un  
valore viene lanciata  
un'eccezione

←-----  
Se non trova nulla  
viene lanciata  
un'eccezione

```
List result = myQuery.getResultList();
```

←----- lista i risultati

# JPA QL: Restrizioni

## Esempi

```
select u from User u  
where u.email = 'foo@hibernate.org'
```

```
select u.USER_ID, u.FIRSTNAME,  
u.LASTNAME, u.USERNAME, u.EMAIL  
from USER u  
where u.EMAIL = 'foo@hibernate.org'
```

```
select i from Item i where i.isActive = true
```

 ←----- accesso proprietà

```
select bid from Bid bid where bid.amount between 1 and 10
```

```
select bid from Bid bid where bid.amount > 100
```

```
select u from User u where u.email in ('foo@bar',  
'bar@foo')
```

←----- utilizzo operatori  
Simili SQL

# JPA QL: Restrizioni

## Esempi

```
select u from User u where u.email is null
```

```
select i from Item i where i.successfulBid is  
not null
```

← uso di NULL

```
select u from User u where u.firstname like 'G%'
```

```
select u from User u where u.firstname not like  
'%Foo B%'
```

← uso di LIKE

```
select user from User user
```

```
where user.firstname like 'G%' and user.lastname like 'K%'
```

```
select u from User u
```

```
where ( u.firstname like 'G%' and u.lastname like 'K%' )
```

```
or u.email in ('foo@hibernate.org', 'bar@hibernate.org' )
```

← espressioni  
complesse

# JPA QL: Restrizioni

## Operatori e precedenza

Operator	Description
.	Navigation path expression operator
+, -	Unary positive or negative signing (all unsigned numeric values are considered positive)
*, /	Regular multiplication and division of numeric values
+, -	Regular addition and subtraction of numeric values
=, <>, <, >, >=, <=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL,	Binary comparison operators with SQL semantics
IS [NOT] EMPTY, [NOT] MEMBER [OF]	Binary operators for collections in HQL and JPA QL
NOT, AND, OR	Logical operators for ordering of expression evaluation

# JPA QL: Restrizioni

```
select i from Item i where i.bids is not empty
```

```
select i from Item i, Category c where  
    i.id = '123' and i member of c.items
```

```
select i from Item i where size(i.bids) > 3
```

```
select u from User u where  
    lower(u.email) = 'foo@hibernate.org'
```

```
select user from User user where  
    concat(user.firstname, user.lastname)  
    like 'G% K%'
```

←----- collezioni

←----- funzioni

# JPA QL : Restrizioni

Function	Applicability
UPPER(s), LOWER(s)	String values; returns a string value
CONCAT(s1, s2)	String values; returns a string value
SUBSTRING(s, offset, length)	String values (offset starts at 1); returns a string value
TRIM( [[BOTH LEADING TRAILING] char [FROM]] s)	Trims spaces on BOTH sides of s if no char or other specification is given; returns a string value
LENGTH(s)	String value; returns a numeric value
LOCATE(search, s, offset)	Searches for position of ss in s starting at offset; returns a numeric value
ABS(n), SQRT(n), MOD(dividend, divisor)	Numeric values; returns an absolute of same type as input, square root as double, and the remainder of a division as an integer
SIZE(c)	Collection expressions; returns an integer, or 0 if empty



# JPA QL: Ordinamento

```
select u from User u order by u.username desc
```

Proprietà singola

Possibile ordinare su più proprietà

```
select u from User u order by u.lastname asc, u.firstname asc
```



# JPA QL : Proiezione

→ Prodotto cartesiano

```
Query q = em.createQuery("select i, b from Item i, Bid b");  
Iterator pairs = q.getResultList().iterator();  
while ( pairs.hasNext() ) {  
    Object[] pair = (Object[]) pairs.next();  
    Item item = (Item) pair[0];  
    Bid bid = (Bid) pair[1];  
}
```

# JPA QL : Proiezione

→ Clausola distinct

```
select distinct item.description from Item item
```

→ Invocazioni funzioni

```
select item.startDate, current_date() from Item item
```

```
select item.startDate, item.endDate, upper(item.name) from Item item
```

# JPA QL : Join impliciti

```
select u from User u where u.homeAddress.city = 'Bangkok'
```

```
select distinct u.homeAddress.city from User u
```

```
select bid from Bid bid where bid.item.description like '%Foo%'
```

```
select bid from Bid bid where bid.item.category.name like 'Laptop%'  
and bid.item.successfulBid.amount > 100
```

# JPA QL : Join espliciti

```
select i, b from Item i join i.bids  
b where i.description like '%Foo%'  
and b.amount > 100
```

contiene b ----->

```
select i.DESCRPTION,  
       i.INITIAL_PRICE, ...  
       b.BID_ID, b.AMOUNT,  
       b.ITEM_ID, b.CREATED_ON  
from ITEM i inner join BID b  
       on i.ITEM_ID = b.ITEM_ID  
where i.DESCRPTION like '%Foo%'  
and b.AMOUNT > 100
```

```
select i  
from Item i join i.bids b  
where i.description like '%Foo%'  
and b.amount > 100
```

-----> seleziona solo i

# JPA QL : Join Left e fetch

→ Left

```
select i from Item i left join i.bids b with b.amount > 100
where i.description like '%Foo%'
```

→ Fetch

```
select i from Item i left join
fetch i.bids
where i.description like '%Foo%'
```



```
select i.DESCRPTION,
       i.INITIAL_PRICE, ...
       b.BID_ID, b.AMOUNT, b.ITEM_ID,
       b.CREATED_ON
from ITEM i left outer join BID b on
       i.ITEM_ID = b.ITEM_ID
where i.DESCRPTION like '%Foo%'
```

# JPA QL : Left join e fetch

→ Relazioni uno-a-uno e multi-a-uno

```
from Bid bid
    left join fetch bid.item
    left join fetch bid.bidder
where bid.amount > 100
```



```
select b.BID_ID, b.AMOUNT, b.ITEM_ID,
       b.CREATED_ON i.DESCRPTION,
       i.INITIAL_PRICE, ... u.USERNAME,
       u.FIRSTNAME, u.LASTNAME, ...
from BID b left outer join ITEM i on
           i.ITEM_ID = b.ITEM_ID
       left outer join USER u on
           u.USER_ID = b.BIDDER_ID
where b.AMOUNT > 100
```

# JPA QL : Funzioni aggregazione

```
Long count = (Long) em.createQuery("select count(i) from Item i")  
                        .getSingleResult();
```

```
select sum(i.successfulBid.amount) from Item I
```

```
select min(bid.amount) , max(bid.amount) from Bid bid where bid.item.id = 1
```



# JPA QL : group by e having

→ Clausola group by

```
select u.lastname, count(u) from User u group by u.lastname
```

```
select bidItem.id, count(bid), avg(bid.amount)
```

```
from Bid bid join bid.item bidItem
```

```
where bidItem.successfulBid is null group by bidItem.id
```

→ Clausola having

```
select item.id, count(bid), avg(bid.amount)
```

```
from Item item join item.bids bid
```

```
where item.successfulBid is null group by item.id having count(bid) > 10
```