Object-Relational Mapping (ORM)

Programmazione Distribuita - A.A. 2020/2021

Biagio Cosenza
Dipartimento di Informatica
Università di Salerno
http://cosenza.eu/
bcosenza@unisa.it

Organizzazione della Lezione

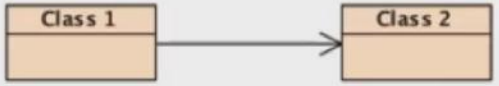
- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - tipi di query
- Ciclo di vita
  - callbacks
  - listeners
- Conclusioni

ORM: Mapping di relazioni

- Object-Relational Mapping è la maniera in cui il bridge tra OO e DB è più evidente
- Nel mondo object-oriented, ci sono le classi e le relazioni tra di esse
  - le relazioni possono essere di tipo unidirezionale (un oggetto può navigare verso un altro) oppure bidirezionale (si può navigare anche nell'altra direzione)
  - si usa il punto `.` per navigare tra oggetti
    - la notazione `.customer.getAddress().getCountry()` è una navigazione dall'oggetto customer al suo indirizzo e al paese
- In UML ci sono notazioni per esprimere queste associazioni, compreso l'associazione con molteplicità (o cardinalità)
  - in quel caso un oggetto della classe di partenza può riferire più oggetti della classe di destinazione

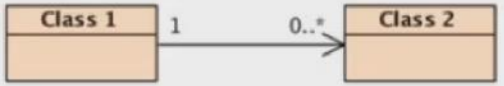
ORM: Esempi di relazioni

- Associazione unidirezionale fra due classi



- Associazione bidirezionale fra due classi
  - in Java: `Class1` che ha un attributo del tipo `Class2` e `Class2` che ha un attributo del tipo `Class1`

- Associazione con molteplicità
  - `Class1` si riferisce a zero o più instanze di `Class2`



Relazioni in database relazionali

- Nel mondo relazionale, un database relazionale è una collezione di relazioni, dette anche tabelle
  - ogni cosa viene modellata come una tabella
- Per modellare un'associazione... si usano le tabelle
- In JPA quando si ha una associazione fra una classe ed un'altra, nel database si ottiene una *table reference*
- Questa reference può essere modellata in due modi diversi:
  1. usando una foreign key (join column)
  2. usando una join table

Relazioni in database relazionali

- Esempio di join column
  - consideriamo l'esempio di un cliente che ha un indirizzo (one-to-one unidirezionale)
  - come si modella in Java: la classe Customer con un attributo Address
  - nel mondo relazionale: una tabella CUSTOMER che punta ad una tabella ADDRESS usando ad esempio una join column

Customer

Primary key	Firstname	Lastname	Foreign key
1	James	Rorisson	11
2	Dominic	Johnson	12
3	Maca	Macaron	13

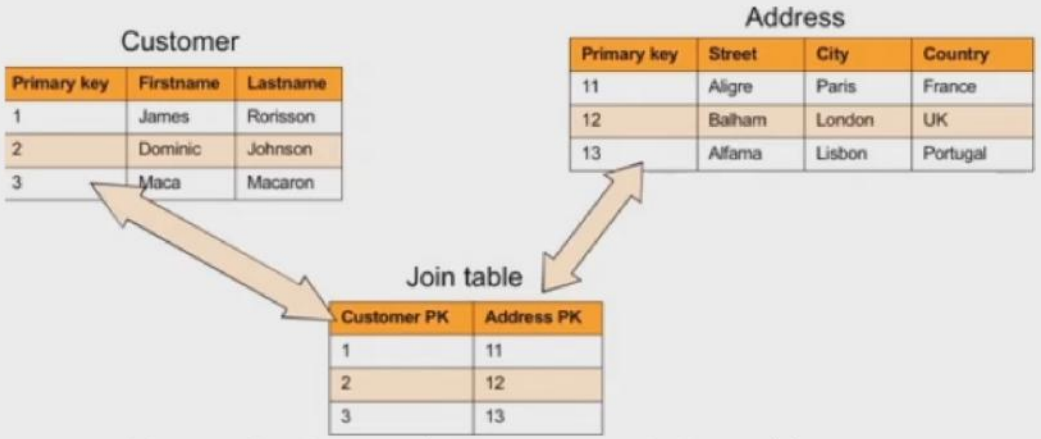
Address

Primary key	Street	City	Country
11	Allgre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Una relazione che usa una join column

Relazioni in database relazionali

- Esempio di join table
  - consideriamo l'esempio di un cliente che ha un indirizzo (one-to-one unidirezionale)



Una relazione che usa una join table

Relazioni in database relazionali

- Esempio di join column
  - consideriamo l'esempio di un cliente che ha un indirizzo (one-to-one unidirezionale)
  - come si modella in Java: la classe Customer con un attributo Address
  - nel mondo relazionale: una tabella CUSTOMER che punta ad una tabella ADDRESS usando ad esempio una join column

Customer

Primary key	Firstname	Lastname	Foreign key
1	James	Rorisson	11
2	Dominic	Johnson	12
3	Maca	Macaron	13

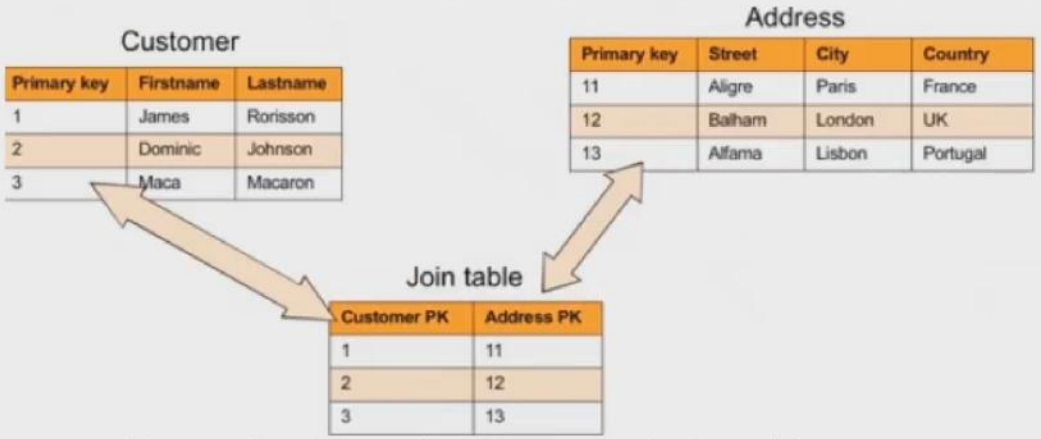
Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Una relazione che usa una join column

Relazioni in database relazionali

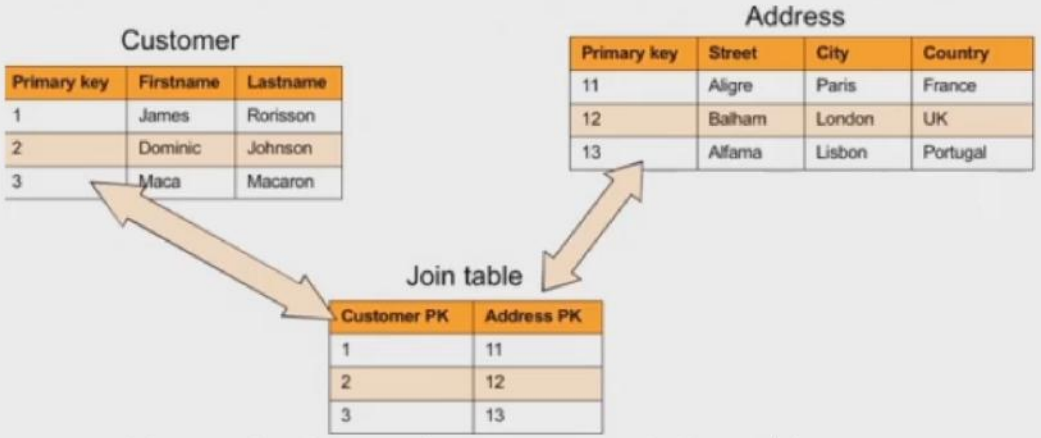
- Esempio di join table
  - consideriamo l'esempio di un cliente che ha un indirizzo (one-to-one unidirezionale)



Una relazione che usa una join table

Relazioni in database relazionali

- Esempio di join table
  - consideriamo l'esempio di un cliente che ha un indirizzo (one-to-one unidirezionale)
  - Per rappresentare una relazione one-to-one quale soluzione è preferibile fra le due presentate?



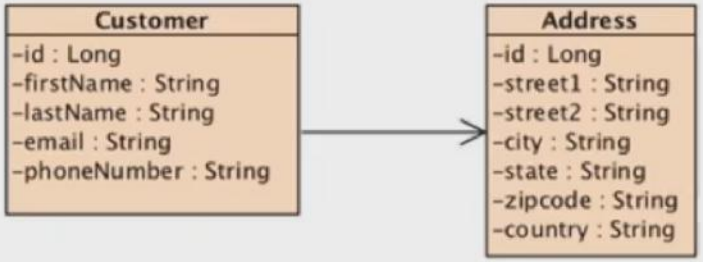
Una relazione che usa una join table

Entity Relationships

- La maggior parte delle entità hanno necessità di referenziare, o avere relazioni con altre entità
  - JPA permette di mappare associazioni cosicché una entity possa essere linkata ad un'altra in un modello relazionale
- La cardinalità fra due entità può essere:
  - one-to-one
  - one-to-many
  - many-to-one
  - many-to-many
- Con rispettive annotazioni:
  - `@OneToOne`
  - `@OneToMany`
  - `@ManyToOne`
  - `@ManyToMany`
- Ogni annotazioni può essere usata in modo unidirezionale o bidirezionale

Esempio di mapping unidirezionale

- In una relazione unidirezionale, una entità `Customer` ha un attributo di tipo `Address`
  - la relazione è `one-way`, da un lato verso l'altro



- L'entità `Customer` rappresenta il proprietario della relazione (*ownership*)
  - in termini di database, la tabella CUSTOMER avrà una foreign key (join column) che punta ad ADDRESS
  - ha la possibilità di personalizzare il mapping di questa relazione
    - se si vuole cambiare il nome della *foreign key*, il mapping andrà fatto nella entity `Customer` (l'owner)
- Nota: *i diagrammi UML non mostrano gli attributi che rappresentano una relazione*

Esempio di mapping unidirezionale

```java
@Entity
public class Customer {

  @Id
  @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  private Address address;

  public Customer() {}

 // Getters & setters
 …
}
```

```java
@Entity
public class Address {

  @Id
  @GeneratedValue
  private Long id;
  private String street1;
  private String street2;
  private String city;
  private String state;
  private String zipcode;
  private String country;

  public Address() {}
 // Getters & setters
 …
}
```

Come si cambia il nome di un attributo?

- Dato l'oggetto `Book`, il default è una tabella di nome BOOK
- Rules for configuration-by-exception mapping: nome dell'entità e nome della tabella coincidono
  - Book entity mappata in una tabelle BOOK
- Per cambiare il nome in `t_book`:

```
@Entity
@Table(name = "t_book")
public class Book {

@Id
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;

  // Constructors, getters, setters
}
```

Esempio di mapping bidirezionale

- Come si fa il mapping di una relazione bidirezionale?
- Chi è l'owner?
  - bisogna dirlo esplicitamente con l'elemento `mappedBy`

Esempio di mapping bidirezionale

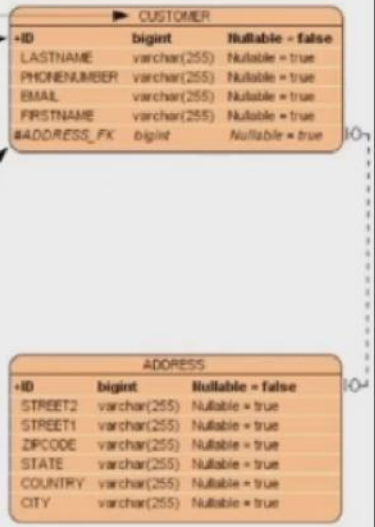
- Entrambe le entità hanno un collegamento verso l'altra
  - `Customer` ha un attributo `address` annotato con `@OneToOne`
  - l'entità `Address` ha un attributo customer annotato con `@OneToOne`
  - usa l'elemento `mappedBy` sulla sua annotazione
    - inverse owner della relazione

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```



Esempio di mapping bidirezionale

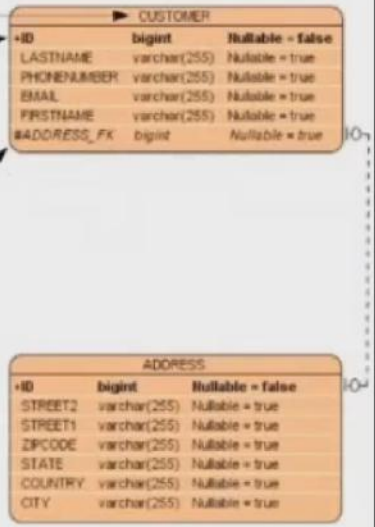
- Entrambe le entità hanno un collegamento verso l'altra
  - `Customer` ha un attributo `address` annotato con `@OneToOne`
  - l'entità `Address` ha un attributo customer annotato con `@OneToOne`
  - usa l'elemento `mappedBy` sulla sua annotazione
    - inverse owner della relazione
- L'elemento `mappedBy`
  - indica che la join column (address) è specificata all'altro lato della relazione
  - infatti, nell'altro lato, l'entità `Customer` definisce la join column usando l'annotazione `@JoinColumn` e rinomina la foreign key in `address_fk`

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```



Come si manipolano le entità con un EM

- I metodi di un Entity Manager

Method	Description
`void persist(Object entity)`	Makes an instance managed and persistent
`<T> T find(Class<T> entityClass, Object primaryKey)`	Searches for an entity of the specified class and primary key
`<T> T getReference(Class<T> entityClass, Object primaryKey)`	Gets an instance, whose state may be lazily fetched
`void remove(Object entity)`	Removes the entity instance from the persistence context and from the underlying database

Come si manipolano le entità con un EM

- I metodi di un Entity Manager

Method	Description
`<T> T merge(T entity)`	Merges the state of the given entity into the current persistence context
`void refresh(Object entity)`	Refreshes the state of the instance from the database, overwriting changes made to the entity, if any
`void flush()`	Synchronizes the persistence context to the underlying database
`void clear()`	Clears the persistence context, causing all managed entities to become detached
`void detach(Object entity)`	Removes the given entity from the persistence context, causing a managed entity to become detached
`boolean contains(Object entity)`	Checks whether the instance is a managed entity instance belonging to the current persistence context

Esempio di riferimento: one-way, con relazione one-to-one tra `Customer` e `Address`

- Un POJO che è una entità
- Nome della classe e della entità
  - chiave primaria
  - campi vari
  - definizione della relazione e del tipo
    - lazy fetch di `Address`
- Sul campo `address` c'è il riferimento ad un'altra entità!
  - rinominata la chiave esterna
- FetchType.LAZY:
  - i dati devono essere caricati solo quando l'applicazione richiede le proprietà

```
@Entity
public class Customer {
   @Id @GeneratedValue
   private Long id;
   private String firstName;
   private String lastName;
   private String email;
  @OneToOne (fetch = FetchType.LAZY)
  @JoinColumn(name ="address_fk")
  private Address address;

  // Constructors, getters, setters

}
```

Customer Entity con un Address one-way, one-to-one

Esempio di riferimento: one-way, con relazione one-to-one tra `Customer` e `Address`

- Un POJO che è una entità
  - nome
  - la chiave
  - i vari campi

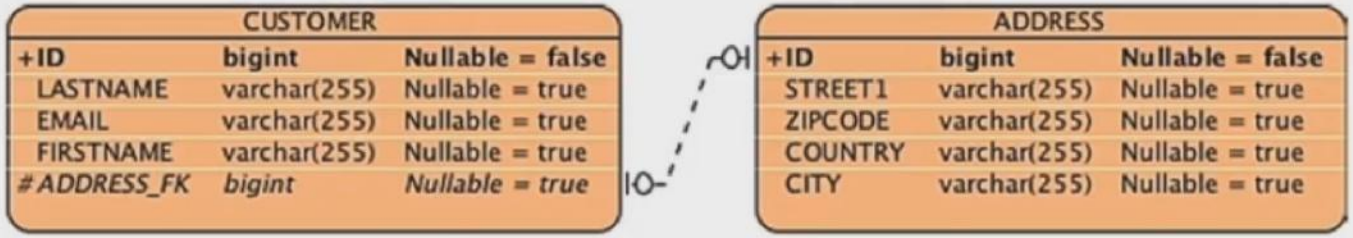
```
@Entity
public class Address {
  @Id @GeneratedValue
  private Long id;
  private String street1;
  private String city;
  private String zipcode;
  private String country;

 // Constructors, getters, setters

}
```

Esempio di riferimento: Cosa si e' creato

- La relazione che si viene a creare tra le due tabelle (non sono POJOs, sono entità!)



- Nullable = false: si rifiuta il valore `null` per rendere obbligatoria la relazione

- Negli esempi che seguono si assume che `em` sia un `EntityManager` mentre `tx` sia un `EntityTransaction`

Rendere persistente un'entita

- si crea un oggetto cliente
  - e un oggetto indirizzo
  - li si lega (puro Java)
    - nel database, la riga customer viene collegata ad address attraverso una foreign key
  - racchiuso da una transazione
  - si rende persistente il cliente
- fino a questo punto, niente è fatto nel DB
  - solo con `commit` vengono inserite le righe nelle tabelle

```
//...
Customer customer = new Customer("Antony","Balla",
        "tballa@mail.com");
Address address = new Address("RitherdonRd", "London",
        "8QE","UK");

customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());
//...
```

- Le espressioni `assertNotNull` verificano che entrambe le entità abbiano ricevuto un identifier
  - grazie al persistent provider e alle annotazioni @Id @GeneratedValue

Trovare una entità (Finding by ID)

- Si ricerca per chiave primaria
  - in caso l'oggetto non si trova, viene restituito `null`

```
//Per ID
Customer customer = em.find(Customer.class, 1234L)
if(customer!=null) {
  //Process the object
}

//Per riferimento
try{
 Customer customer = em.getReference(Customer.class, 1234L)
   //Process the object
   //...
}catch(EntityNotFoundException ex) {
  //Entity not found
}
```

- `getReference()` simile al `find()`
  - ma qui siamo interessati al riferimento di una entità non ai suoi dati
  - il fetching è lazy: solo la chiave viene acceduta, altre info non vengono usate
    - di fatto e' un proxy ad un entity, non l'entity stesso
  - se l'entità non esiste, si lancia una eccezione, da gestire

Rimozione di un entita

- Si creano i due oggetti e si linkano insieme
- In una transazione
  - vengono resi persistenti
  - e viene fatto il `commit`
- In un'altra transazione
  - viene cancellato il cliente
  - a seconda della politica di *cascading* l'indirizzo può essere o no cancellato
  - al `commit` il cambio viene effettuato su DB
- Nota bene: il POJO esiste sempre!

```
//...
Customer customer = new Customer("Antony","Balla",
    "tballa@mail.com");
Address address = new Address("RitherdonRd","London",
    "8QE","UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

tx.begin();
em.remove(customer);
tx.commit();

assertNotNull(customer);

//...
```

Cascading: rimozione degli orfani

- Evitare gli orfani
  - problemi di data consistency
  - equivalgono a righe di tabelle non referenziate

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne (fetch = FetchType.LAZY,orphanRemoval = true)
  private Address address;

  // Constructors, getters, setters
}
```

- JPA supporta la rimozione automatica degli orfani
  - nell'esempio: dell'entità `Address` quando il `Customer` viene rimosso
- Si definisce la relazione in modo che alla cancellazione del cliente (`Customer`) viene cancellato l'indirizzo (`Address`)

Sincronizzazione con il DB, flush e refresh

- La sincronizzazione del database avviene al momento del commit
  - l'entity manager è in first-level cache, ed attende che la transazione sia committata per il flush dei dati nel database
- Cosa accade quando un customer ed un address devono essere inseriti?

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

- Due istruzioni INSERT vengono prodotte e rese permanenti solo al commit della transazione

Sincronizzazione con il DB, flush e refresh

- Con il metodo `EntityManager.flush()` il persistence provider potrebbe essere esplicitamente forzato a fare il flush dei dati nel database
  - ma non a fare `commit` della transazione

- Vediamo un esempio…

Sincronizzazione con il DB, flush e refresh

- Primo esempio:
  - persistenza: due INSERT SQL
  - commit della transazione
- Secondo esempio:
  - persistenza di un cliente
  - forzata l'esecuzione della INSERT precedente
  - persistenza di address

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

Sincronizzazione con il DB, flush e refresh

- Primo esempio:
  - persistenza: due INSERT SQL
  - commit della transazione
- Secondo esempio:
  - persistenza di un cliente
  - forzata l'esecuzione della INSERT precedente
  - persistenza di address

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

- Problema!
  - non funziona a causa di un integrity constraint sull'address foreign key che non è ancora committato
    - la colonna ADDRESS_FK in CUSTOMER
  - la transazione verrà annullata (roll back)

Sincronizzazione con il DB, flush e refresh

- Il metodo `EntityManager.refresh()` è usato per la sincronizzazione dei dati nella direzione opposta del `flush()`
  - overwrite dello stato di un entity con i dati presenti nel database

- Esempio

1. In memoria il nome è Antony
2. Settiamo il nuovo nome
3. Ma poi il valore è in refresh dal DB!
4. Quindi vale Antony

```
Customer customer = em.find(Customer.class, 1234L)
assertEquals(customer.getFirstName(),"Antony");
customer.setFirstName("William");

em.refresh(customer);
assertEquals(customer.getFirstName(),"Antony");
```

Interazione con il Persistence Context: Contains e Detach

- Il metodo `EntityManager.contains()` restituisce un Boolean e permette di controllare se una istanza è *managed* dall'entity manager all'interno del persistence context
- Il metodo `clear()` azzera il persistence context, e tutte le entità *managed* diventano *detached*
- Il metodo `detach(Object entity)` rimuove una entità dal persistence context

Interazione con il Persistence Context: Contains e Detach

- Persist di una entità
- facciamo l'eliminazione dal *persistence context*
- e verifichiamo che sia cosi

```
//...
Customer customer = new Customer("Antony","Balla",
                "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();

assertTrue(em.contains(customer));

em.detach(customer);
assertFalse(em.contains(customer));
```

Interazione col Persistence Context: Clear, Merge, Update

- Una entità di cui è stato fatto *detach* non è più associato ad un *persistence context*
- Per ri-gestirlo, occorre farne il *reattach* (ovvero: *merge*)

Interazione col Persistence Context: Clear, Merge, Update

- Entità persistente
  - `clear`: tutte le entità, quindi anche l'entity Customer, vengono eliminate dal PC
    - esistono ma non sono sincronizzate col DB
  - qui il cambiamento non viene sincronizzato con il DB
    - perché eseguito su una entità *detached*
  - solo dopo il `merge`, il cambiamento viene replicato sul database

```
Customer customer = new Customer("Antony","Balla",
                "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
em.clear();

customer.setFirstName("William");
tx.begin();
em.merge(customer);
tx.commit();
```

Interazione col Persistence Context: Clear, Merge, Update

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
customer.setFirstName("William");
tx.commit();
```

- Cosa accade in questo caso?

Interazione col Persistence Context: Clear, Merge, Update

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
customer.setFirstName("William");
tx.commit();
```

- Cosa accade in questo caso?

1. Persist di un customer il cui nome è «Antony»
2. Vogliamo settare il nuovo nome («William»)
3. Poiché l'entity è *managed*, i cambiamenti vengono apportati anche al database

Cascading Events

- Di default ogni entity manager operation si applica esclusivamente all'entità passata come argomento dell'operazione
- Spesso si desidera che una modifica apportata su una entità sia propagata in cascata a tutte le sue associazioni
- Questa operazione è conosciuta come *cascading an event*

Cascading Events

- Nell'esempio che segue

```
Customer customer   new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

- Per creare un customer, istanziamo un `Customer` ed un `Address` entity
- le colleghiamo con `customer.setAddress(address)` e le rendiamo persistenti

Cascading Events

- Poiché esiste una relazione fra `Customer` e `Address`, si può mettere in cascata l'azione `persist` dal customer all'address
- Una chiamata a `em.persist(customer)` comporterà in cascata la persistenza dell'`Address` entity se permette questo tipo di propagazione
- È quindi possibile ridurre il codice ed eliminare `em.persist(address)`

Esempio di Cascading Events

- Entità
  - nome della classe e della entità
  - chiave primaria
  - relazione
- Definizione del cascading per le operazioni indicate
- Questa persist va a cascata anche su address

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne (fetch = FetchType.LAZY,
             cascade = {CascadeType.PERSIST,
                        CascadeType.REMOVE})
  @JoinColumn(name ="address_fk")
  private Address address;

  // Constructors, getters, setters
}
```

```
//...
Customer customer = new Customer("Antony","Balla",
     "tballa@mail.com");
Address address = new Address("RitherdonRd","London",
     "8QE","UK");
customer.setAddress(address);
tx.begin();
em.persist(customer);
tx.commit();
```