Java EE™

Object-Relational Mapping (ORM)

Corso di Laurea in Informatica, Programmazione Distribuita
Delfina Malandrino, dmalandrino@unisa.it
http://www.unisa.it/docenti/delfinamalandrino

1

Organizzazione della lezione

2

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

1

2

Organizzazione della lezione

3

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

3

Mapping di relazioni

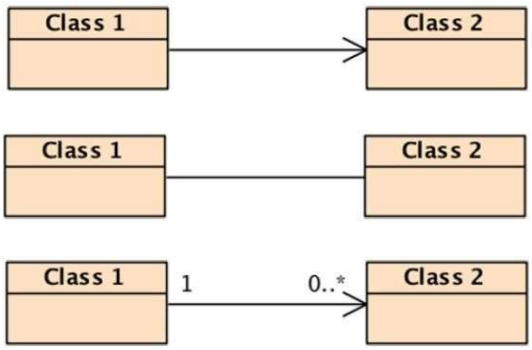
4

- 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 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 (ocardinalità)
  - in quel caso un oggetto della classe di partenza può riferire più oggetti della classe di destinazione

4

Esempi di relazioni

5



Associazione unidirezionale fra due classi

Associazione bidirezionale fra due classi

In Java, questa situazione è rappresentata con 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

5

Relazioni in Database relazionali

6

- 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:
  - usando una foreign key (join column)
  - usando una join table

6

Relazioni in Database relazionali

Esempio

7

- 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

Relazione che usa una join column

7

Relazioni in Database relazionali

Esempio

8

- Consideriamo l'esempio di un cliente che ha un indirizzo (one-to-one unidirezionale)

Customer

Primary key	Firstname	Lastname
1	James	Rorisson
2	Dominic	Johnson
3	Maca	Macaron

Address

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

Join table

Customer PK	Address PK
1	11
2	12
3	13

Relazione che usa una join table

Per rappresentare una relazione one-to-one quale soluzione è preferibile fra le due presentate?

8

Entity Relationships

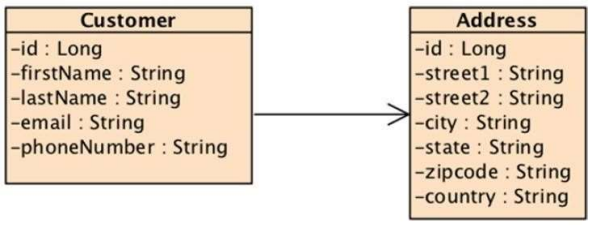
9

- 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 annotations
- Ogni annotazioni può essere usata in modo unidirezionale o bidirezionale

9

Un esempio di mapping unidirezionale

10



- In una relazione unidirezionale, una entità `Customer` ha un attributo di tipo `Address`
- La relazione è ***one-way***, navigating da un lato verso l'altro
- L'entità `Customer` rappresenta il proprietario della relazione
  - 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)

10

Come si cambia il nome di un attributo?

11

- 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 BOOK table)

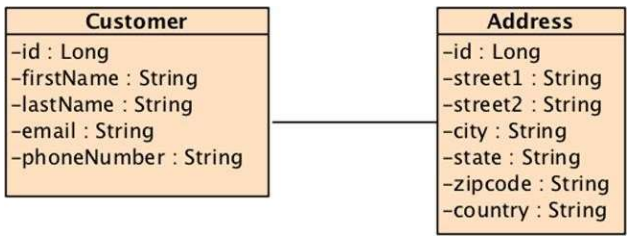
- Per cambiare il nome in t_book:

```
@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
}
```

11

Un esempio di mapping bidirezionale

12



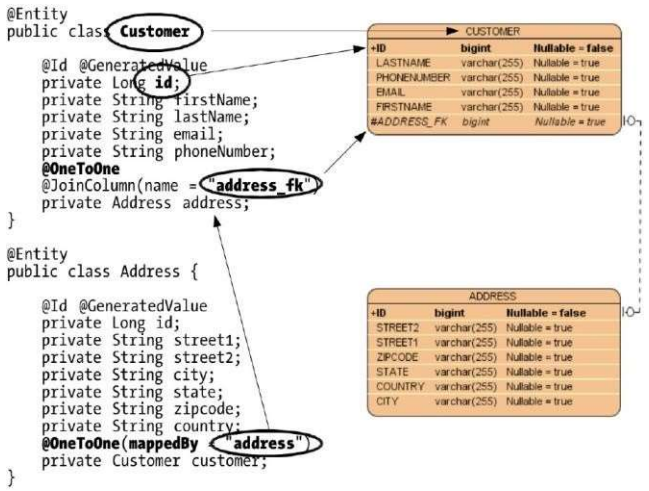
- Come si fa il mapping di una relazionebidirezionale?
- Chi è l'owner?
  - Bisogna dirlo esplicitamente con l'elemento **mappedBy**

12

Un esempio di mapping bidirezionale

13

- 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`**



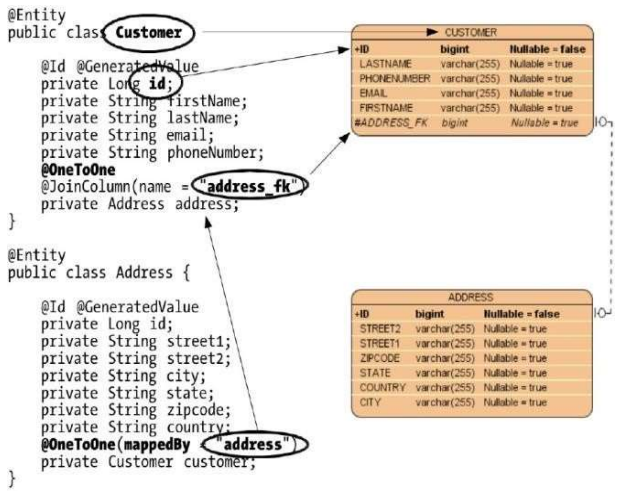
13

Un esempio di mapping bidirezionale

14

- Entrambe le entità hanno un collegamento verso l'altra
  - L'entità **`Address`** 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



14

Organizzazione della lezione

15

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

15

I metodi di un Entity Manager

16

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

16

I metodi di un Entity Manager

17

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

17

Esempio di riferimento

(one-way, relazione one-to-one fra **Customer** e **Address**)

Listing 6.5

```
@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

}
```

Un POJO che è una entità

9

18

Esempio di riferimento

(one-way, relazione one-to-one fra `Customer` e `Address`)

Listing 6.5

```
@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

}
```

- Un POJO che è una entità
- Nome della classe e della entità

19

Esempio di riferimento

(one-way, relazione one-to-one fra `Customer` e `Address`)

Listing 6.5

```
@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

}
```

- Un POJO che è una entità
- Nome della classe e della entità
- Chiave primaria

10

20

Esempio di riferimento

(one-way, relazione one-to-one fra `Customer` e `Address`)

Listing 6.5

```
@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

}
```

- Un POJO che è una entità
- Nome della classe e della entità
- Chiave primaria
- Campi vari

21

Esempio di riferimento

(one-way, relazione one-to-one fra `Customer` e `Address`)

Listing 6.5

```
@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

}
```

- 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`)

FetchType.LAZY = i dati devono essere caricati lazely, solo quando l'applicazione richiede le proprietà

22

Esempio di riferimento

(one-way, relazione one-to-one fra **Customer** e **Address**)

Listing 6.5

```
@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

}
```

- Un POJO che è una entità
- Nome della classe e della entità
- Chiave primaria
- Campi vari
- Definizione della relazione e del tipo
- Su questo campo **address** c'è il riferimento ad un'altra entità!

  Rinominata la chiave esterna

FetchType.LAZY = i dati devono essere caricati lazely, solo quando l'applicazione richiede le proprietà

The Customer Entity with a One-Way, One-to-One Address

Esempio di riferimento

(one-way, relazione one-to-one fra **Customer** e **Address**)

Listing 6.6

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

    //Constructors,getters,setters
}
```

- Un POJO che è una entità

Esempio di riferimento

(one-way, relazione one-to-one fra **Customer** e **Address**)

Listing 6.6

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

    //Constructors,getters,setters
}
```

- Un POJO che è una entità
- Il nome

25

Esempio di riferimento

(one-way, relazione one-to-one fra **Customer** e **Address**)

Listing 6.6

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

    //Constructors,getters,setters
}
```

- Un POJO che è una entità
- Il nome
- La chiave

26

Esempio di riferimento

(one-way, relazione one-to-one fra `Customer` e `Address`)

Listing 6.6

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

    //Constructors,getters,setters
}
```

- Un POJO che è una entità
- Il nome
- La chiave
- I vari campi

27

Cosa si è creato

28

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

CUSTOMER		
+ID	bigint	Nullable = false
LASTNAME	varchar(255)	Nullable = true
EMAIL	varchar(255)	Nullable = true
FIRSTNAME	varchar(255)	Nullable = true
#ADDRESS_FK	*bigint*	*Nullable = true*

ADDRESS		
+ID	bigint	Nullable = false
STREET1	varchar(255)	Nullable = true
ZIPCODE	varchar(255)	Nullable = true
COUNTRY	varchar(255)	Nullable = true
CITY	varchar(255)	Nullable = true

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`

28

Rendere persistente una entità

Listing 6-7

```
//...
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());

//...
```

- Si crea un oggetto cliente

29

Rendere persistente una entità

Listing 6-7

```
//...
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());

//...
```

- Si crea un oggetto cliente
- ...e un oggetto indirizzo

30

Rendere persistente una entità

Listing 6-7

```
//...
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());

//...
```

- 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

31

Rendere persistente una entità

Listing 6-7

```
//...
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());

//...
```

- Si crea un oggetto cliente
- ... e un oggetto indirizzo
- Li si lega (puro Java)
- Racchiuso da una transazione

32

Rendere persistente una entità

Listing 6-7

```
//...
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());

//...
```

- Si crea un oggetto cliente
- ... e un oggetto indirizzo
- Li si lega (puro Java)
- Racchiuso da una transazione
- Si rende persistente il cliente

33

Rendere persistente una entità

Listing 6-7

```
//...
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());

//...
```

- Si crea un oggetto cliente
- ... e un oggetto indirizzo
- Li si lega (puro Java)
- Racchiuso da una transazione
- Si rende persistente il cliente
- ... e l'indirizzo

34

Rendere persistente una entità

Listing 6-7

```
//...
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());

//...
```

- Si crea un oggetto client
- . . . e un oggetto indirizzo
- Li si lega (puro Java)
- Racchiuso da una Transazione
- Si rende persistente il cliente
- … e l'indirizzo
- Fino a questo punto, niente è fatto nel DB, solo con commit vengono inserite le righe nelle tabelle

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

35

Trovare una entità (Finding by ID)

Listing 6-8

36

```
//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
}
```

- Si ricerca per chiave primaria

36

Trovare una entità (Finding by ID)

Listing 6-8

37

```
//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) {
    //Entitynotfound
}
```

› Si ricerca per chiave primaria

› In caso l'oggetto non si trova, viene restituito null

37

Trovare una entità (Finding by ID)

38

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

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

Listing 6-8

Listing 6-9

› Si ricerca per chiave primaria

› In caso l'oggetto non si trova, viene restituito null

› 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)

Se l'entità non esiste, si lancia una eccezione, da gestire

38

Rimozione di una entità

Listing 6-10

```
//...
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);

//...
```

- Si creano i due oggetti e si linkano insieme

39

Rimozione di una entità

Listing 6-10

```
//...
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);

//...
```

- Si creano i due oggetti e si linkano insieme
- In una transazione . . .

20

40

Rimozione di una entità

Listing 6-10

```
//...
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);

//...
```

- Si creano i due oggetti e si linkano insieme
- In una transazione . . .
- . . . vengono resi persistenti

41

Rimozione di una entità

Listing 6-10

```
//...
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);

//...
```

- Si creano i due oggetti e si linkano insieme
- In una transazione . . .
- . . . vengono resi persistenti
- . . . e viene fatto il commit

42

Rimozione di una entità

Listing 6-10

```
//...
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);

//...
```

- Si creano i due oggetti e si linkano insieme
- In una transazione . . .
- … vengono resi persistenti
- … e viene fatto il commit
- In un'altra transazione

43

Rimozione di una entità

Listing 6-10

```
//...
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);

//...
```

- Si creano i due oggetti e si linkano insieme
- In una transazione . . .
- … vengono resi persistenti
- … e viene fatto il commit
- In un'altratransazione
- Viene cancellato il cliente: a seconda della politica di *cascading* l'indirizzo può essere o no cancellato

44

Rimozione di una entità

Listing 6-10

```
//...
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);

//...
```

› Si creano i due oggetti e si linkano insieme

› In una transazione . . .

› … vengono resi persistenti

› … e viene fatto il commit

› In un'altratransazione

› Viene cancellato il cliente: a seconda della politica di *cascading* l'indirizzo può essere o no cancellato

› Il cambio viene effettuato sul DB

Nota bene: il POJO esiste sempre!

Il cascading: rimozione degli orfani

Listing 6-11

46

Per problemi di data consistency i dati non referenziati nelle tabelle (orfani) sono da evitare

Bisogna pertanto provvedere alla rimozione automatica dell'entità **`Address`** quando il **`Customer`** viene rimosso


```
@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
}
```

Si definisce la relazione in modo che alla cancellazione del cliente **`(Customer)`** viene cancellato l'indirizzo **`(Address)`**

Sincronizzazione con il DB, flush e refresh

47

- La sincronizzazione del database avviene al commit time
  - L'entity manager è un 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

47

Sincronizzazione con il DB, flush e refresh

48

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

48

Sincronizzazione con il DB: flush e refresh

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

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

› Inizio transazione

49

Sincronizzazione con il DB: flush e refresh

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

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

› Inizio transazione
› Persistenza: due INSERT SQL

50

Sincronizzazione con il DB: flush e refresh

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

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

- Inizio transazione
- Persistenza: due INSERT SQL
- Commit della transazione

51

Sincronizzazione con il DB: flush e refresh

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

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

- Inizio transazione
- Persistenza: due INSERT SQL
- Commit della transazione
- Persistenza di un cliente

52

Sincronizzazione con il DB: flush e refresh

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

- Inizio transazione
- Persistenza: due INSERT SQL
- Commit della transazione

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

- Persistenza di un cliente
- Forzata l'esecuzione della INSERT precedente

Sincronizzazione con il DB: flush e refresh

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

- Inizio transazione
- Persistenza: due INSERT SQL
- Commit della transazione

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

- Persistenza di un cliente
- Forzata l'esecuzione della INSERT precedente
- Persistenza di address

PROBLEMA! Non funziona a causa di un integrity constraint sull'l'address foreign key (colonna **ADDRESS_FK** in **CUSTOMER**) che non è ancora committato

La transazione verrà annullata (roll back)

Sincronizzazione con il DB: flush e refresh

55

- 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

55

Sincronizzazione con il DB: flush e refresh

56 Refreshing the Customer Entity from the Database

Listing 6-12

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

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

› In memoria il nome è Antony

56

Sincronizzazione con il DB: flush e refresh

57 Refreshing the Customer Entity from the Database

Listing 6-12

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

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

- In memoria il nome è Antony
- Settiamo il nuovo nome

57

Sincronizzazione con il DB: flush e refresh

58 Refreshing the Customer Entity from the Database

Listing 6-12

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

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

- In memoria il nome è Antony
- Settiamo il nuovo nome
- Ora il valore è refresh dal DB! Quindi vale *Antony*

58

Interazione con il Persistence Context: Contains e Detach

59

- 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

59

Interazione con il Persistence Context: Contains e Detach

60 Test Case for Whether the Customer Entity Is in the Persistence Context

Listing 6-13

```
//...
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));
```

› Persist di una entità…

30

60

Interazione con il Persistence Context: Contains e Detach

61 Test Case for Whether the Customer Entity Is in the Persistence Context

Listing 6-13

```
//...
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));
```

› Persist di una entità…

› …facciamo l'eliminazione dal PC

61

Interazione con il Persistence Context: Contains e Detach

62 Test Case for Whether the Customer Entity Is in the Persistence Context

Listing 6-13

```
//...
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));
```

› Persist di una entità…

› …facciamo l'eliminazione dal PC

› e controlliamo che sia cosi

62

Interazione con il Persistence Context:
Clear, Merge, Update

63

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

63

Interazione con il Persistence Context:
Clear, Merge, Update

```
//..
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();
```

Entità persistente

Listing 6-15

32

64

Interazione con il Persistence Context:
Clear, Merge, Update

```
//..
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();
```

- Entità persistente
- Tutte le entità, quindi anche l'entity Customer, vengono eliminate dal PC esistono ma non sono sincronizzate con il DB)

65

Interazione con il Persistence Context:
Clear, Merge, Update

```
//..
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();
```

- Entità persistente
- Tutte le entità, quindi anche l'entity Customer, vengono eliminate dal PC esistono ma non sono sincronizzate con il DB)
- Ciò accade con questa istruzione è che il cambiamento non viene sincronizzato con il DB (perché eseguito su una entità detached)

66

Interazione con il Persistence Context:
Clear, Merge, Update

```
//..
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();
```

- Entità persistente
- Tutte le entità, quindi anche l'entity Customer, vengono eliminate dal PC esistono ma non sono sincronizzate con il DB)
- Ciò accade con questa istruzione è che il cambiamento non viene sincronizzato con il DB (perché eseguito su una entità detached)

› . . . solo dopo il merge, il cambiamento viene replicato sul database

67

Interazione con il Persistence Context:
Clear, Merge, Update

68

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

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

68

Cascading events

69

- 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*

69

Cascading events

70

- 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();
```

  - Istanziamo un Customer ed un Address entity, le colleghiamo (customer.setAddress(address)), e le rendiamo persistenti

70

Cascading events

71

- 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)`**

71

Cascading events

```
@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
}
```

Entità

Listing 6-18

```
//...
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();
```

36

72

Cascading events

```
@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();
```

- Entità
- Nome della classe e della entità

Listing 6-18

73

Cascading events

```
@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();
```

- Entità
- Nome della classe e della entità
- Chiave primaria

Listing 6-18

74

Cascading events

```
@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();
```

Listing 6-18

- Entità
- Nome della classe e della entità
- Chiave primaria
- Relazione

Cascading events

```
@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();
```

Listing 6-18

- Entità
- Nome della classe e della entità
- Chiave primaria
- Relazione
- Definizione del cascading per le operazioni indicate

Cascading events

```
@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();
```

Listing 6-18

- 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`

77

Organizzazione della lezione

78

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

39

78

Java Persistence Query Language

79

- JPQL permette di interrogare entità persistenti indipendentemente dal database utilizzato
- Simile alla sintassi di SQL, con la differenza che JPQL restituisce non tabelle (con righe e colonne) ma una entità o una collezione di entità
  - POJOs facili da gestire nel linguaggio
- JPQL traduce la query in SQL (usando JDBC per collegarsi)
- Le query possono essere di tipo diverso, molto espressive come per SQL

Esempi di JPQL

80

```
SELECT <select clause>
    FROM <from clause>
    [WHERE <where clause>]
    [ORDER BY <order by clause>]
    [GROUP BY <group by clause>]
    [HAVING <having clause>]
```

- SELECT: definisce il formato dei risultati (entità o loro attributi)
- FROM: definisce una entità o le entità da cui si vogliono ottenere dei risultati
- WHERE: istruzione condizionale per restringere il risultato
  - Possibile usare parametri posizionali: WHERE c.firstName = ?1 AND c.address.country = ?2
- ORDER: in ordine decrescente (DESC) o crescente (ASC)
- GROUP: possibile raggruppare (per contare ad esempio) selezionando con il filtro HAVING

Esempi di JPQL

81

```
SELECT <select clause>
    FROM <from clause>
    [WHERE <where clause>]
    [ORDER BY <order by clause>]
    [GROUP BY <group by clause>]
    [HAVING <having clause>]
```

- Selezionare tutte le istanze di una singola entità

```
SELECT b
FROM Book b
```

- La clausola FROM è usata anche per definire un alias all'entity:
  - b è un alias **Book**
- La clausola SELECT indica che il tipo del risultato della query è l'entity b (**Book**)
  - Il risultato sarò una lista di 0 o più **Book** instances

81

Esempi di JPQL

82

```
SELECT <select clause>
    FROM <from clause>
    [WHERE <where clause>]
    [ORDER BY <order by clause>]
    [GROUP BY <group by clause>]
    [HAVING <having clause>]
```

- Restringiamo il risultato usando la clausola WHERE

```
SELECT b
FROM Book b
WHERE b.title = 'H2G2'
```

- Il risultato sarò una lista di 0 o più Book instances che hanno un titolo = H2G2

41

82

Organizzazione della lezione

83

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

83

Le query possibili con JPA

84

- Ci sono 5 tipi di query che permettono in contesti diversi di integrare JPQL nella applicazione Java
  - Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
  - Named Query: query statiche definite e non modificabili
  - Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
  - Query native: per eseguire SQL nativo invece di JPQL
  - Query da Stored Procedure: introdotte da JPA 2.1
- Tramite metodi dell'Entity Manager si ottiene una query di un certo tipo
  - Dalla quale si vanno a prelevare risultato/risultati, etc.

84

Come ottenere una query dall'EM

85

`Query createQuery(String jpqlString)`	Creates an instance of Query for executing a JPQL statement for dynamic queries
`Query createNamedQuery(String name)`	Creates an instance of Query for executing a named query (in JPQL or in native SQL)
`Query createNativeQuery(String sqlString)`	Creates an instance of Query for executing a native SQL statement
`Query createNativeQuery(String sqlString, Class resultClass)`	Native query passing the class of the expected results
`Query createNativeQuery(String sqlString, String resultSetMapping)`	Native query passing a result set mapping
`<T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery)`	Creates an instance of TypedQuery for executing a criteria query
`<T> TypedQuery<T> createQuery(String jpqlString, Class<T> resultClass)`	Typed query passing the class of the expected results
`<T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass)`	Typed query passing the class of the expected results
`StoredProcedureQuery createStoredProcedureQuery(String procedureName)`	Creates a StoredProcedureQuery for executing a stored procedure in the database
1. `StoredProcedureQuery createStoredProcedureQuery(String procedureName, Class... resultClasses)`	Stored procedure query passing classes to which the result sets are to be mapped
2. `StoredProcedureQuery createStoredProcedureQuery(String procedureName, String... resultSetMappings)`	Stored procedure query passing the result sets mapping
`StoredProcedureQuery createNamedStoredProcedureQuery(String name)`	Creates a query for a named stored procedure

EntityManager methods per la creazione di query

Table 6-4

85

Query API

86

- Eseguire una query ed ottenere risultati

```
public interface Query {

  // Executes a query and returns a result
  List getResultList();
  Object getSingleResult();
  int executeUpdate();
```


86

Query API

87

- Settare parametri per una query

```
// Sets parameters to the query
Query setParameter(String name, Object value);
Query setParameter(String name, Date value, TemporalType temporalType);
Query setParameter(String name, Calendar value, TemporalType temporalType);
Query setParameter(int position, Object value);
Query setParameter(int position, Date value, TemporalType temporalType);
Query setParameter(int position, Calendar value, TemporalType temporalType);
<T> Query setParameter(Parameter<T> param, T value);
Query setParameter(Parameter<Date> param, Date value, TemporalType temporalType);
Query setParameter(Parameter<Calendar> param, Calendar value, TemporalType temporalType);
```

87

Query API

88

- Ottenere parametri da una query

```
// Gets parameters from the query
Set<Parameter<?>> getParameters();
Parameter<?> getParameter(String name);
Parameter<?> getParameter(int position);
<T> Parameter<T> getParameter(String name, Class<T> type);
<T> Parameter<T> getParameter(int position, Class<T> type);
boolean isBound(Parameter<?> param);
<T> T getParameterValue(Parameter<T> param);
Object getParameterValue(String name);
```


88

Query API

89

```
// Constrains the number of results returned by a query
Query setMaxResults(int maxResult);
int getMaxResults();
Query setFirstResult(int startPosition);
int getFirstResult();

// Sets and gets query hints
Query setHint(String hintName, Object value);
Map<String, Object> getHints();

// Sets the flush mode type to be used for the query execution
Query setFlushMode(FlushModeType flushMode);
FlushModeType getFlushMode();

// Sets the lock mode type to be used for the query execution
Query setLockMode(LockModeType lockMode);
LockModeType getLockMode();

// Allows access to the provider-specific API
<T> T unwrap(Class<T> cls);
}
```

89

Le query possibili con JPA… descriviamole

90

- Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
- Named Query: query statiche definite e non modificabili
- Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
- Query native: per eseguire SQL nativo invece di JPQL
- Query da Stored Procedure: introdotte da JPA 2.1

90

Le query possibili con JPA… descriviamole

91

- Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
- Named Query: query statiche definite e non modificabili
- Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
- Query native: per eseguire SQL nativo invece di JPQL
- Query da Stored Procedure: introdotte da JPA 2.1

91

Query dinamiche

92

```
Query query = em.createQuery("SELECT c FROM Customer c");

List<Customer> customers = query.getResultList();
```

- Restituito un oggetto Query
- Il risultato della query è una lista
- Il metodo getResultList() method restituisce una lista di `Customer` entities (`List<Customer>`)
- Se però è noto che il risultato è una singola entità allora bisogna usare il metodo `getSingleResult()`

92

Query dinamiche

92

```
Query query = em.createQuery("SELECT c FROM Customer c");

List<Customer> customers = query.getResultList();
```

- Il metodo `getResultList()` restituisce una lista di untyped objects
- Se vogliamo una lista del tipo `Customer`?
  - Bisogna usare una **`TypedQuery`**

93

Query dinamiche

93

```
//…
TypedQuery<Customer> query = em.createQuery("SELECT c FROM Customer c", Customer.class);

List<Customer> customers = query.getResultList();
```


94

Query dinamiche

95

- La query può essere creata dall'applicazione
- String concatenation usata per costruire una query a seconda di uno specifico criterio

```
String jpqlQuery = "SELECT c FROM Customer c";

if (someCriteria)
      jpqlQuery += " WHERE c.firstName = 'Betty'";

query = em.createQuery(jpqlQuery);

List<Customer> customers = query.getResultList();
```

95

Query dinamiche

96

```
String jpqlQuery = "SELECT c FROM Customer c";
if (someCriteria)
    jpqlQuery += " WHERE c.firstName = 'Betty'";
query = em.createQuery(jpqlQuery);
List<Customer> customers = query.getResultList();
```

- Nell'esempio precedente abbiamo fatto una SELECT specificando "Betty" come firstName
- Volendo parametrizzare la SELECT (1)…. Oppure volendo usare un parametro di posizione (2)…

(1)

```
//…
query = em.createQuery("SELECT c FROM Customer c where c.firstName = :fname");
query.setParameter("fname", "Betty");
List<Customer> customers = query.getResultList();
```

(2)

```
//…
query = em.createQuery("SELECT c FROM Customer c where c.firstName = ?1");
query.setParameter(1, "Betty");
List<Customer> customers = query.getResultList();
```

96

Query dinamiche

97

- Se vogliamo paginazione dei risultati a gruppi di 10 alla volta:

```
query = em.createQuery("SELECT c FROM Customer c", Customer.class);

query.setMaxResults(10);

List<Customer> customers = query.getResultList();
```

97

Le query possibili con JPA… descriviamole

98

- Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
- Named Query: query statiche definite e non modificabili
- Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
- Query native: per eseguire SQL nativo invece di JPQL
- Query da Stored Procedure: introdotte da JPA 2.1

98

Named queries

99

- Le Named queries sono statiche e non modificabili
- Meno flessibili ma più efficienti dal momento il persistence provider può tradurre la stringa JPQL in SQL una sola volta quando l'applicazione parte, e non ogni volta che la query deve essere eseguita
- Si utilizza l'annotazione **`@NamedQuery`**
- Esempio:
  - Cambiamo l'entità **`Customer`** e staticamente definiamo 3 queries usando l'annotazione richiesta

99

Named queries

100

- Esempio:
  - Cambiamo l'entità **`Customer`** e staticamente definiamo 3 queries usando l'annotazione richiesta

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent", ➥
              query="select c from Customer c where c.firstName = 'Vincent'"),
  @NamedQuery(name = "findWithParam", ➥
              query="select c from Customer c where c.firstName = :fname")
})
```

100

Named query

101

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent",
        query="select c from Customer c where c.firstName = 'Vincent'"),
  @NamedQuery(name = "findWithParam",
        query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
 @Id @GeneratedValue
 private Long id;
 private String firstName;
 private String lastName;
 private Integer age;
 private String email;
 @OneToOne
 @JoinColumn(name = "address_fk")
 private Address address;

 // Constructors, getters, setters
}
```

Entità con query statiche (efficienti)

101

Named query

102

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent",
        query="select c from Customer c where c.firstName = 'Vincent'"),
  @NamedQuery(name = "findWithParam",
        query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
 @Id @GeneratedValue
 private Long id;
 private String firstName;
 private String lastName;
 private Integer age;
 private String email;
 @OneToOne
 @JoinColumn(name = "address_fk")
 private Address address;

 // Constructors, getters, setters
}
```

Entità con query statiche (efficienti)

Query che seleziona tutti i customer dal DB

102

Named query

103

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent",
        query="select c from Customer c where c.firstName = 'Vincent'"),
  @NamedQuery(name = "findWithParam",
        query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
 @Id @GeneratedValue
 private Long id;
 private String firstName;
 private String lastName;
 private Integer age;
 private String email;
 @OneToOne
 @JoinColumn(name = "address_fk")
 private Address address;

 // Constructors, getters, setters
}
```

Entità con query statiche (efficienti)

Query che seleziona tutti i customer dal DB

Query per un certo utente

103

Named query

104

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent",
        query="select c from Customer c where c.firstName = 'Vincent'"),
  @NamedQuery(name = "findWithParam",
        query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
 @Id @GeneratedValue
 private Long id;
 private String firstName;
 private String lastName;
 private Integer age;
 private String email;
 @OneToOne
 @JoinColumn(name = "address_fk")
 private Address address;

 // Constructors, getters, setters
}
```

Entità con query statiche (efficienti)

Query che seleziona tutti i customer dal DB

Query per un certo utente

Query con parametro

104

Named query

105

```
Query query = em.createNamedQuery("findWithParam");
query.setParameter("fname", "Vincent");

query.setMaxResults(3);
List<Customer> customers = query.getResultList();
```

- Come si usa
  - Si crea una query dall'EM
  - Si setta un parametro
  - Si definisce il Max numero di risultati (3)
  - Si esegue

105

Alcuni commenti sulle query named

106

- Sono utili per migliorare le prestazioni
- API flessibile: quasi tutti i metodi restituiscono una Query
  - quindi permettono di scrivere eleganti shortcut:

```
Query query =
   em.createNamedQuery("findWithParam").setParameter("fname","Vincent")
                                       .setMaxResults(3);
```

- Restrizione: il nome delle query ha scope relativo al persistence unit e deve essere univoco all'interno di questo scope
  - Una findAll query per i customer ed una findAll query per gli address devono avere nomi differenti
- Essendo una stringa il parametro, errori di query sono riconosciuti a runtime (typesafety bye bye!)

106

Le query possibili con JPA… descriviamole

107

- Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
- Named Query: query statiche definite e non modificabili
- Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
- Query native: per eseguire SQL nativo invece di JPQL
- Query da Stored Procedure: introdotte da JPA 2.1

107

Criteria API (or Object-Oriented Queries)

108

- Il vantaggio di scrivere concisamente con le stringhe, è accoppiato al problema della mancanza di controlli a tempo di compilazione
  - Errori tipo SLECT invece di SELECT oppure Custmer invece di Customer sono scoperti a runtime
- Da JPA 2.0 ci sono le CRITERIA API che permettono di scrivere query in maniera sintatticamente corretta
- L'idea è che tutte le keywords JPQL sono definite in questa API
  - API che supportano tutto quello che può fare JPQL ma in maniera Object-Oriented

108

Criteria API (or Object-Oriented Queries)

109

- Esempio: Vogliamo una query che restituisce tutti i customers con nome named "Vincent"
- In JPQL:

```
SELECT c FROM Customer c WHERE c.firstName = 'Vincent'
```

- Con le Criteria API:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Vincent"));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

- SELECT, FROM, e WHERE hanno un API representation attraverso i metodi **`select`**(), **`from`**(), e **`where`**()
  - Questa regola è valida per ogni JPQL keyword

109

Le query possibili con JPA… descriviamole

110

- Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
- Named Query: query statiche definite e non modificabili
- Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
- Query native: per eseguire SQL nativo invece di JPQL
- Query da Stored Procedure: introdotte da JPA 2.1

110

Native Queries

111

- Native queries prendono una native SQL statement (SELECT, UPDATE, o DELETE) come parametro e restituiscono una Query instance
- Non sono portabili

```
Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.class);
List<Customer> customers = query.getResultList();
```

111

Native Queries

112

- Come le named queries, native queries posono usare le annotazioni per definire SQL queries statiche
- Le Named native queries sono definite usando l'annotazione @NamedNativeQuery (posizionata sull'entità)
- Il nome della query deve essere unico all'interno del persistence unit

```
@Entity
@NamedNativeQuery(name = "findAll", query="select * from t_customer")
@Table(name = "t_customer")
public class Customer {...}
```

112

Le query possibili con JPA… descriviamole

113

- Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
- Named Query: query statiche definite e non modificabili
- Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
- Query native: per eseguire SQL nativo invece di JPQL
- Query da Stored Procedure: introdotte da JPA 2.1

113

Stored procedure

114

- Tutte le query finora viste sono simili in comportamento
- Le query stored sono invece esse stesse definite nel database
- Utili per compiti ripetitivi ed ad alta intensità di uso dei dati
- Diversi vantaggi (anche se si perde di portabilità):
  - migliori prestazioni per la precompilazione
  - permette di raccogliere statistiche, per ottimizzare le prestazioni
  - evita di dover trasmettere dati (codice sul server)
  - codice centralizzato e usabile da diversi programmi (non solo Java)
  - ulteriore possibilità di controlli di sicurezza (accesso alla stored procedure)

114

Stored procedure: esempio pratico

115

- Servizio di archiviazione di libri e CD
  - Dopo una certa data books e CDs devono essere archiviati
    - Fisicamente trasferiti dal magazzino al rivenditore
  - Il servizio è time-consuming e diverse tabelle devono essere aggiornate
    - Inventory, Warehouse, Book, CD, Transportation tables, etc.)
  - Soluzione: scriviamo una stored procedure che raggruppi diverse istruzioni SQL per migliorare le performance
- La stored procedure **`sp_archive_books`**
  - ha due argomenti in ingresso: archive date ed un warehouse code
  - aggiorna le tabelle T_Inventory e T_Transport

115

Stored procedure: un esempio

116

Procedura in SQL

Definizione della procedura, compilata nel DB (complessa, riguarda diverse tabelle)

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS
  UPDATE T_Inventory
  SET Number_Of_Books_Left - 1
  WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;

  UPDATE T_Transport
  SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```


116

Stored procedure: un esempio

117

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS
  UPDATE T_Inventory
  SET Number_Of_Books_Left - 1
  WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;

  UPDATE T_Transport
  SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

- La stored procedure è compilata nel database e può essere invocata attraverso il suo nome sp_archive_books
- La stored procedure accetta dati nella forma di parametri di input e di output (@archiveDate and @warehouseCode nel nostro esempio)

117

Stored procedure: un esempio

117

- E' possibile invocare una stored procedure con annotazioni (@NamedStoredProcedureQuery) o dinamicamente
- Vediamo un esempio di Book entity che dichiara la sp_archive_books stored procedure usando named query annotations
- L'annotazione NamedStoredProcedureQuery specifica il nome della stored procedure da invocare e i tipi di tutti i paramentri (Date.class and String.class), i loro corrispondenti parameter modes (IN, OUT, INOUT, ecc)

118

Stored procedure: un esempio

118

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS
  UPDATE T_Inventory
  SET Number_Of_Books_Left - 1
  WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;

  UPDATE T_Transport
  SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

```
@Entity
@NamedStoredProcedureQuery(name ="archiveOldBooks",
            procedureName ="sp_archive_books",
    parameters = {
        @StoredProcedureParameter(name ="archiveDate",
                            mode = IN, type = Date.class),
        @StoredProcedureParameter(name ="warehouse",
                            mode = IN, type = String.class)
    }
)
public class Book {
    @Id @GeneratedValue
    private Long id;
    private String title;
    //...etc.etc.
}
```

- Definizione della procedura, compilata nel DB (complessa, riguarda diverse tabelle)
- Definizione di una named stored procedure per una entità **Book**

Entity che dichiara la stored procedure

119

Stored procedure: un esempio

Procedura in SQL

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE,
                            @warehouseCode VARCHAR AS
    UPDATE T_Inventory
    SET Number_Of_Books_Left - 1
    WHERE Archive_Date < @archiveDate AND
                    Warehouse_Code = @warehouseCode;
    UPDATE T_Transport
    SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

```
@Entity
@NamedStoredProcedureQuery(name ="archiveOldBooks",
            procedureName ="sp_archive_books",
    parameters = {
        @StoredProcedureParameter(name ="archiveDate",
                            mode = IN, type = Date.class),
        @StoredProcedureParameter(name ="warehouse",
                            mode = IN, type = String.class)
    }
)
public class Book {
    @Id @GeneratedValue
    private Long id;
    private String title;
    //...etc.etc.
}
```

- Definizione della procedura, compilata nel DB (complessa, riguarda diverse tabelle)
- Definizione di una named stored procedure per una entità **Book**
- I parametri

Entity che dichiara la stored procedure

120

Stored procedure: un esempio

Procedura in SQL

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE,
                                  @warehouseCode VARCHAR AS
    UPDATE T_Inventory
    SET Number_Of_Books_Left - 1
    WHERE Archive_Date < @archiveDate AND
                         Warehouse_Code = @warehouseCode;
    UPDATE T_Transport
    SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

```
@Entity
@NamedStoredProcedureQuery(name ="archiveOldBooks",
            procedureName ="sp_archive_books",
    parameters = {
        @StoredProcedureParameter(name ="archiveDate",
                                  mode = IN, type = Date.class),
        @StoredProcedureParameter(name ="warehouse",
                                  mode = IN, type = String.class)
    }
)
public class Book {
    @Id @GeneratedValue
    private Long id;
    private String title;
    //...etc.etc.
}
```

- Definizione della procedura, compilata nel DB (complessa, riguarda diverse tabelle)
- Definizione di una named stored procedure per una entità **Book**
- I parametri
- …ciascuno con il loro tipo (una data)

Entity che dichiara la stored procedure

121

Stored procedure: un esempio

Procedura in SQL

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE,
                                  @warehouseCode VARCHAR AS
    UPDATE T_Inventory
    SET Number_Of_Books_Left - 1
    WHERE Archive_Date < @archiveDate AND
                         Warehouse_Code = @warehouseCode;
    UPDATE T_Transport
    SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

```
@Entity
@NamedStoredProcedureQuery(name ="archiveOldBooks",
            procedureName ="sp_archive_books",
    parameters = {
        @StoredProcedureParameter(name ="archiveDate",
                                  mode = IN, type = Date.class),
        @StoredProcedureParameter(name ="warehouse",
                                  mode = IN, type = String.class)
    }
)
public class Book {
    @Id @GeneratedValue
    private Long id;
    private String title;
    //...etc.etc.
}
```

- Definizione della procedura, compilata nel DB (complessa, riguarda diverse tabelle)
- Definizione di una named stored procedure per una entità **Book**
- I parametri
- …ciascuno con il loro tipo (una data)
- …e qui una stringa

Entity che dichiara la stored procedure

122

Stored procedure: un esempio

123

- Per invocare la stored procedure sp_archive_books è necessario
  - Usare l'entity manager
  - Creare una named stored procedure query passando il suo nome (archiveOldBooks)
  - Questo restituisce una StoredProcedureQuery **`query`** sulla quale settare i parametri ed eseguire

Listing 6-31. Calling a StoredProcedureQuery

```
StoredProcedureQuery query = em.createNamedStoredProcedureQuery("archiveOldBooks");
query.setParameter("archiveDate", new Date());
query.setParameter("maxBookArchived", 1000);
query.execute();
```


Organizzazione della lezione

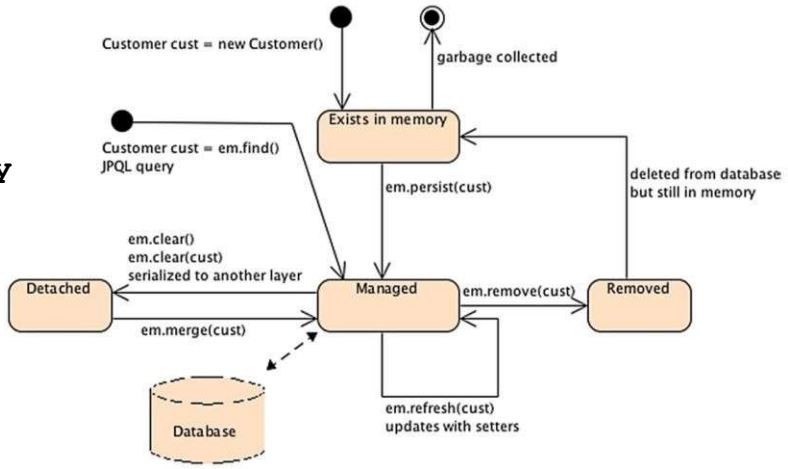
124

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

Il ciclo di vita di una entità

125

`Customer Entity`

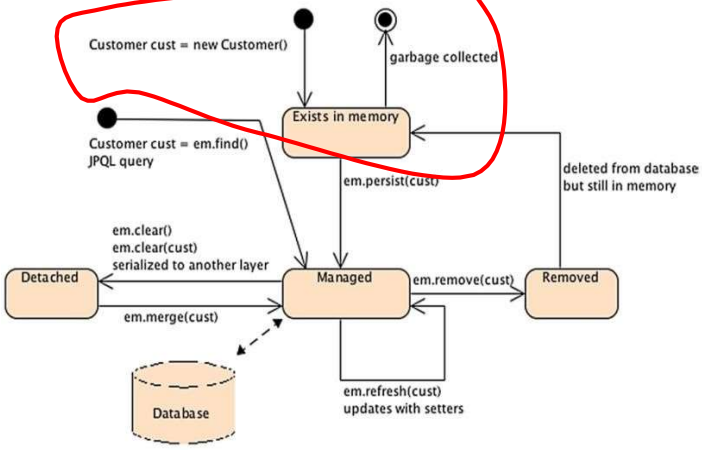


125

Il ciclo di vita di una entità

126

- Per creare una istanza della Customer entity, usiamo l'operatore **new**
- Questo oggetto esiste in memoria anche se JPA non ne è ancora a conoscenza
- Se l'oggetto non viene usato, verrà liberato dal garbage collector ed il cilco di vita termina

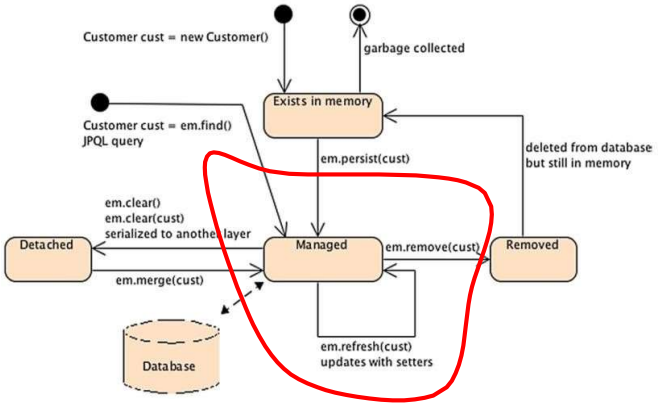


126

Il ciclo di vita di una entità

127

- Quando viene invocato il metodo `EntityManager.persist()`, l'entità diventa 'managed', ed il suo stato sincronizzato con il database
- In questa fase (managed state), è possibile settare attributi (usando setter methods come customer.setFirstName()) oppure fare refresh del contenuto con il metodo EntityManager.refresh()
- Tutti questi cambiamenti verranno sincronizzati con il database

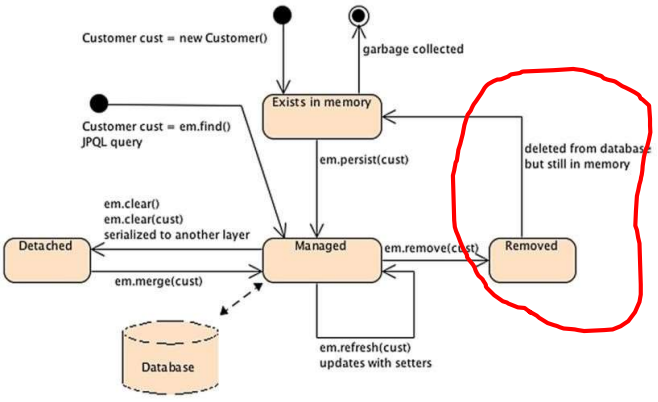


127

Il ciclo di vita di una entità

128

- Nel managed state, è possibile invocare il metodo `EntityManager.remove()` e l'entità verrà cancellata dal database
- L'entità non sarà più gestita, ma l'oggetto Java continua a risiedere in memoria fin quando non interverrà il garbage collector



128

I diversi tipi di Callback

129

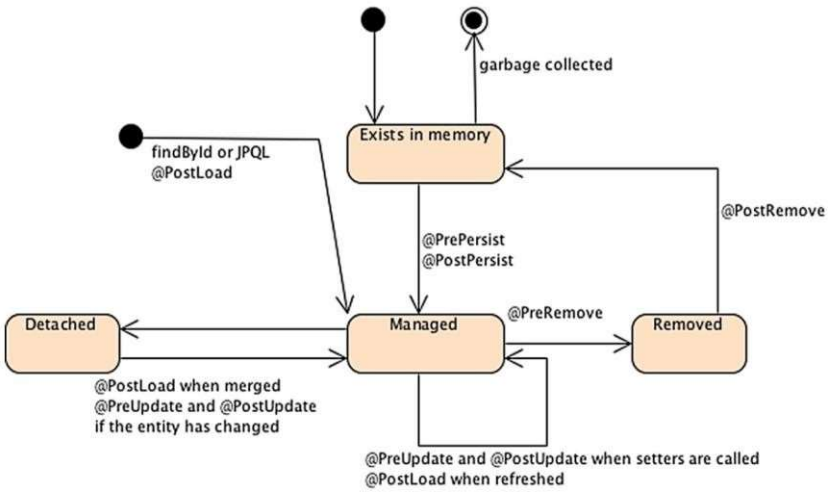
- Il ciclo di vita delle entità ricade in 4 categorie di stati nel ciclo di vita:
  - persisting, updating, removing e loading
- Per ogni categoria, ci sono eventi `pre` e eventi `post` che possono essere intercettati dall'entity manager quando si deve invocare un metodo di business

Annotation	Description
`@PrePersist`	Marks a method to be invoked before `EntityManager.persist()` is executed.
`@PostPersist`	Marks a method to be invoked after the entity has been persisted. If the entity autogenerates its primary key (with `@GeneratedValue`), the value is available in the method.
`@PreUpdate`	Marks a method to be invoked before a database update operation is performed (calling the entity setters or the `EntityManager.merge()` method).
`@PostUpdate`	Marks a method to be invoked after a database update operation is performed.
`@PreRemove`	Marks a method to be invoked before `EntityManager.remove()` is executed.
`@PostRemove`	Marks a method to be invoked after the entity has been removed.
`@PostLoad`	Marks a method to be invoked after an entity is loaded (with a JPQL query or an `EntityManager.find()`) or refreshed from the underlying database. There is no `@PreLoad` annotation, as it doesn't make sense to preload data on an entity that is not built yet.

129

Il ciclo di vita con gli eventi

130



130

Esempio con annotazioni di callback

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @PrePersist
    @PreUpdate
    private void validate() {
          if(firstName == null||"".equals(firstName))
              throw new IllegalArgumentException(
                                        "Invalidfirstname");
        if(lastName ==null||"".equals(lastName))
              throw new IllegalArgumentException(
                                        "Invalidlastname");
    }

    //...
```

- Entità

131

Esempio con annotazioni di callback

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @PrePersist
    @PreUpdate
    private void validate() {
          if(firstName == null||"".equals(firstName))
              throw new IllegalArgumentException(
                                        "Invalidfirstname");
        if(lastName ==null||"".equals(lastName))
              throw new IllegalArgumentException(
                                        "Invalidlastname");
    }

    //...
```

- Entità
- Campo "temporale"

66

132

Esempio con annotazioni di callback

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @PrePersist
    @PreUpdate
    private void validate() {
          if(firstName == null||"".equals(firstName))
              throw new IllegalArgumentException(
                                        "Invalidfirstname");
        if(lastName ==null||"".equals(lastName))
              throw new IllegalArgumentException(
                                        "Invalidlastname");
    }

    //...
```

- Entità
- Campo "temporale"
- Campo non mappato su DB, ma calcolato per il POJO

133

Esempio con annotazioni di callback

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @PrePersist
    @PreUpdate
    private void validate() {
          if(firstName == null||"".equals(firstName))
              throw new IllegalArgumentException(
                                        "Invalidfirstname");
        if(lastName ==null||"".equals(lastName))
              throw new IllegalArgumentException(
                                        "Invalidlastname");
    }

    //...
```

- Entità
- Campo "temporale"
- Campo non mappato su DB, ma calcolato per il POJO
- Campo "temporale", ma timestamp (tick successivi)

134

Esempio con annotazioni di callback

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @PrePersist
    @PreUpdate
    private void validate() {
          if(firstName == null||"".equals(firstName))
              throw new IllegalArgumentException(
                                        "Invalidfirstname");
        if(lastName ==null||"".equals(lastName))
              throw new IllegalArgumentException(
                                        "Invalidlastname");
    }

    //...
```

- Entità
- Campo "temporale"
- Campo non mappatosu DB, ma calcolato per il POJO
- Campo "temporale",ma timestamp (tick successivi)
- Annotazioni per un metodo da chiamare prima di scrivere o di aggiornare nel database

135

Esempio con annotazioni di callback

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @PrePersist
    @PreUpdate
    private void validate() {
          if(firstName == null||"".equals(firstName))
              throw new IllegalArgumentException(
                                        "Invalidfirstname");
        if(lastName ==null||"".equals(lastName))
              throw new IllegalArgumentException(
                                        "Invalidlastname");
    }

    //...
```

- Entità
- Campo "temporale"
- Campo non mappatosu DB, ma calcolato per il POJO
- Campo "temporale",ma timestamp (tick successivi)
- Annotazioni per un metodo da chiamare prima di scrivere o di aggiornare nel database
- Effettua dei controlli di validità

136

Esempio con annotazioni di callback

```
//...
@PostLoad
@PostPersist
@PostUpdate
public void calculateAge(){
 if(dateOfBirth == null) {
     age = null;
     return;
 }
 Calendar birth = new GregorianCalendar();
 birth.setTime(dateOfBirth);
 Calendar now = new GregorianCalendar();
 now.setTime(new Date());
 int adjust = 0;
 if(now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
     adjust = -1;
 }
 age = now.get(YEAR) - birth.get(YEAR) + adjust;

//Constructors,getters,setters
```

Metodo da eseguire dopo aver caricato, reso persistente o fatto update

137

Esempio con annotazioni di callback

```
//...
@PostLoad
@PostPersist
@PostUpdate
public void calculateAge(){
 if(dateOfBirth == null) {
     age = null;
     return;
 }
 Calendar birth = new GregorianCalendar();
 birth.setTime(dateOfBirth);
 Calendar now = new GregorianCalendar();
 now.setTime(new Date());
 int adjust = 0;
 if(now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
     adjust = -1;
 }
 age = now.get(YEAR) - birth.get(YEAR) + adjust;

//Constructors,getters,setters
```

- Metodo da eseguire dopo aver caricato, reso persistente o fatto update
- Calcola l'età del customer e …
- ….la memorizza nel campo transiente age

138

Organizzazione della lezione

139

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

139

I Listeners come generalizzazione di callback

140

- I metodi di callback sono inglobati all'interno della definizione della entità
  - la definizione di Customer
- Nel caso in cui si voglia estrapolare questa logica per applicarla a diverse entità, condividendo il codice, si deve definire un entity listener
- Un Entity Listener è un POJO su cui è possibile definire metodi di callback
- L'entità interessata provvederà a registrarsi a questi listeners usando l'annotazione @EntityListeners

140

I Listeners come generalizzazione di callback

140

- Usando l'esempio del Customer
  - Estraiamo i metodi calculateAge() e validate() per separarli in classi listener separate
  - AgeCalculationListener (Listing 6-39) e DataValidationListener (Listing 6-40)

141

I Listeners come generalizzazione di callback

141

- Alcune regole per i listener dettano come vanno eseguiti:
  - costruttore pubblico senza argomenti
  - i metodi di callback devono avere un parametro del tipo dell'entità (che viene passato automaticamente)
  - se ha parametro Object può essere chiamato su diverse entità, altrimenti ha il tipo specifico

142

Una classe Listener per il calcolo dell'età di un customer

```
public class AgeCalculationListener
{   @PostLoad
    @PostPersist
    @PostUpdate
    public void calculateAge(Customer customer) {
        if(customer.getDateOfBirth() == null) {
            customer.setAge(null);
            return;
        }
        Calendar birth = new GregorianCalendar();
        birth.setTime(customer.getDateOfBirth());
        Calendar now = new GregorianCalendar();
        now.setTime(new Date());
        int adjust = 0;if(now.get(DAY_OF_YEAR) -
              birth.get(DAY_OF_YEAR) < 0) {
            adjust = -1;
        }
        customer.setAge(now.get(YEAR) - birth.get(YEAR) +
              adjust);
    }
```

- Classe standard (POJO)

Una classe Listener per il calcolo dell'età di un customer

```
public class AgeCalculationListener
{   @PostLoad
    @PostPersist
    @PostUpdate
    public void calculateAge(Customer customer) {
        if(customer.getDateOfBirth() == null) {
            customer.setAge(null);
            return;
        }
        Calendar birth = new GregorianCalendar();
        birth.setTime(customer.getDateOfBirth());
        Calendar now = new GregorianCalendar();
        now.setTime(new Date());
        int adjust = 0;if(now.get(DAY_OF_YEAR) -
              birth.get(DAY_OF_YEAR) < 0) {
            adjust = -1;
        }
        customer.setAge(now.get(YEAR) - birth.get(YEAR) +
              adjust);
    }
```

- Classe standard (POJO)
- Definizione di un metodo annotato come una callback: unica differenza il parametro

Una classe Listener per il calcolo dell'età di un customer

```
public class AgeCalculationListener
{   @PostLoad
    @PostPersist
    @PostUpdate
    public void calculateAge(Customer customer) {
        if(customer.getDateOfBirth() == null) {
            customer.setAge(null);
            return;
        }
        Calendar birth = new GregorianCalendar();
        birth.setTime(customer.getDateOfBirth());
        Calendar now = new GregorianCalendar();
        now.setTime(new Date());
        int adjust = 0;if(now.get(DAY_OF_YEAR) -
              birth.get(DAY_OF_YEAR) < 0) {
            adjust = -1;
        }
        customer.setAge(now.get(YEAR) - birth.get(YEAR) +
              adjust);
    }
```

- Classe standard (POJO)
- Definizione di un metodo annotato come una callback: unica differenza il parametro
- Logica di business solita: calcola il campo age di un Customer

145

Una classe Listener per la validazione

```
public class DataValidationListener
 @PrePersist
 @PreUpdate
 private void validate(Customer customer) {
   if(customer.getFirstName() == null||
                        "".equals(customer.getFirstName()))
      throw new IllegalArgumentException(
                              "Invalidfirstname");
   if(customer.getLastName() == null||
                         "".equals(customer.getLastName()))
      throw new IllegalArgumentException(
                               "Invalidlastname");
 }
```

- Classe standard (POJO)

146

Una classe Listener per la validazione

```
public class DataValidationListener
 @PrePersist
 @PreUpdate
 private void validate(Customer customer) {
   if(customer.getFirstName() == null||
                        "".equals(customer.getFirstName()))
      throw new IllegalArgumentException(
                                "Invalidfirstname");
   if(customer.getLastName() == null||
                         "".equals(customer.getLastName()))
      throw new IllegalArgumentException(
                                 "Invalidlastname");
 }
```

- Classe standard (POJO)
- Definizione di un metodo annotato come una callback: unica differenza il parametro

147

Una classe Listener per la validazione

```
public class DataValidationListener
 @PrePersist
 @PreUpdate
 private void validate(Customer customer) {
   if(customer.getFirstName() == null||
                        "".equals(customer.getFirstName()))
      throw new IllegalArgumentException(
                                "Invalidfirstname");
   if(customer.getLastName() == null||
                         "".equals(customer.getLastName()))
      throw new IllegalArgumentException(
                                 "Invalidlastname");
 }
```

- Classe standard (POJO)
- Definizione di un metodo annotato come una callback: unica differenza il parametro
- Logica di business solita: valida un Customer

148

Come registrare i Listeners ad una Entità

```
@EntityListeners({DataValidationListener.class,
                  AgeCalculationListener.class})
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    //Constructors,getters,setters
```

› Registrazione come listener della classe `DataValidationListener`

149

Come registrare i Listeners ad una Entità

```
@EntityListeners({DataValidationListener.class,
                  AgeCalculationListener.class})
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    //Constructors,getters,setters
```

› Registrazione come listener della classe `DataValidationListener`

› … e di `AgeCalculatorListener`

150

Come registrare i Listeners ad una Entità

```
@EntityListeners({DataValidationListener.class,
                  AgeCalculationListener.class})
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    //Constructors,getters,setters
```

- Registrazione come listener della classe `DataValidationListener`
- … e di `AgeCalculatorListener`
- Definizione entità

151

Come registrare i Listeners ad una Entità

```
@EntityListeners({DataValidationListener.class,
                  AgeCalculationListener.class})
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    //Constructors,getters,setters
```

- Registrazione come listener della classe `DataValidationListener`
- … e di `AgeCalculatorListener`
- Definizione entità
- … come prima (check di validità e calcolo dell'età del customer

152

Come registrare i Listeners ad una Entità

153

- Nell'esempio appena visto, l'entità Customer definisce due listener, ma…
- … un singolo listener può essere definito da più di una entità
  - Un listener che fornisce una logica generale, utilizzabile da diverse entità
    - Un debug!

153

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

Prima dell'operazione di persistenza…

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

77

154

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

- Prima dell'operazione di persistenza . . .
- Viene chiamato con qualsiasi tipo (`Object`)

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

155

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

- Prima dell'operazione di persistenza . . .
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di update . . .

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

156

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

- Prima dell'operazione di persistenza . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di update . ..
- Viene chiamato con qualsiasi tipo (`Object`)

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

157

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

- Prima dell'operazione di persistenza . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di update . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di cancellazione . ..

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

158

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

- Prima dell'operazione di persistenza . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di update . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di cancellazione . ..
- Viene chiamato con qualsiasi tipo (Object)

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

159

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

- Prima dell'operazione di persistenza . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di update . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di cancellazione . ..

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

- Definzione nel file persistence.xml

160

Un Listener per diverse Entità: Debug

```
public class DebugListener {
    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }
    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }
    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
```

- Prima dell'operazione di persistenza . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di update . ..
- Viene chiamato con qualsiasi tipo (`Object`)
- Prima dell'operazione di cancellazione . ..

Nel file `persistence.xml`

```
...
<persistence-unit-metadata>
 <persistence-unit-defaults>
  <entity-listeners>
   <entity-listener
    class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
  </entity-listeners>
 </persistence-unit-defaults>
  persistence-unit-metadata>
...
```

- Definzione nel file `persistence.xml`
- Definizione del listener (per tutte le entità)

161

Riassumendo...

162

- Il tag <persistence-unit-metadata> definisce tutti i metadata che non hanno una notazione equivalente
- Il tag <persistence-unit-defaults> definisce tutti i defaults del persistence unit
- Il tag <entity-listener> definisce il default listener
- Al deployment il DebugListener sarà automaticamente invocato for ogni singola entità

Listing 6-43. A Debug Listener Defined as the Default Listener

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
                 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
                 http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd"
                 version="2.1">

   <persistence-unit-metadata>
      <persistence-unit-defaults>
         <entity-listeners>
            <entity-listener class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
         </entity-listeners>
      </persistence-unit-defaults>
   </persistence-unit-metadata>

</entity-mappings>
```


162

Riassumendo…

163

Listing 6-43. A Debug Listener Defined as the Default Listener

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
                 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
                 http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd"
                 version="2.1">

   <persistence-unit-metadata>
      <persistence-unit-defaults>
         <entity-listeners>
            <entity-listener class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
         </entity-listeners>
      </persistence-unit-defaults>
   </persistence-unit-metadata>

</entity-mappings>
```

- Quando si dichiara una lista di default entity listeners, ogni listener verrà invocatonell'ordine in cui è listato nel file XML
- I Default entity listeners sono sempre invocati prima di ogni altro entity listeners listato nell'annotazione @EntityListeners

163

Organizzazione della lezione

164

- Object-relational Mapping
- Come si manipolano le entità con un EM
- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

164