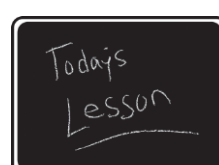
Corso di Laurea in Informatica
III Anno Triennale
Programmazione Distribuita

Java Persistence API (2)

Delfina Malandrino
dmalandrino@unisa.it
http://www.unisa.it/docenti/delfinamalandrino

Organizzazione della lezione

- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni



Organizzazione della lezione

- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

Java Persistence Query Language

- 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

```
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

```
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

Esempi di JPQL

```
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

Organizzazione della lezione

- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

Le query possibili con JPA

- Ci sono 5 tipi di query che permettono in contesti diversi di integrare JPQL nelle applicazioni 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.

Come ottenere una query dall'EM

`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

Query API

- Eseguire una query ed ottenere risultati

```
public interface Query {

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

Query API

- 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);
```

Query API

- 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);
```

Query API

```
// 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);
}
```

Le query possibili con JPA... descriviamole

- 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

Le query possibili con JPA... descriviamole

- 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

Query dinamiche

```
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()**

Query dinamiche

```
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**

Query dinamiche

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

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

Query dinamiche

- 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();
```

Query dinamiche

```
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();
```

Query dinamiche

- 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();
```

Le query possibili con JPA... descriviamole

- 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

Named queries

- 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

Named queries

- 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")
})
```

Named queries

```
@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)

Named queries

```
@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

Named queries

```
@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

Named queries

```
@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

Named queries

```
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

Alcuni commenti sulle query named

- 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);
```

Alcuni commenti sulle query named

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

- **Restrizione: il nome della 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!)

Le query possibili con JPA... descriviamole

- 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

Criteria API (or Object-Oriented Queries)

- 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

Criteria API (or Object-Oriented Queries)

- Esempio: Vogliamo una query che restituisce tutti i customerscon 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();
```

Criteria API (or Object-Oriented Queries)

```
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

Le query possibili con JPA... descriviamole

- 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

Native Queries

- 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();
```

Native Queries

- 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 {...}
```

Le query possibili con JPA... descriviamole

- 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

Stored procedure

- 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)

Stored procedure: esempio pratico

- 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

Stored procedure: un esempio

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
```

Stored procedure: un esempio


```
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
```

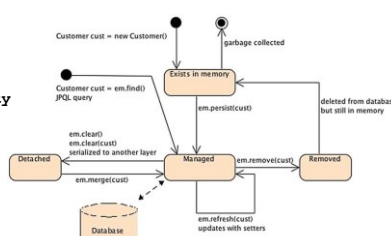

- La stored procedure è compilata nel database e può essere invocata attraverso il suo nome **`sp_archive_books`**
- La stored procedure accetta dati nella forma di parametri di input e di output (@archiveDate and @warehouseCode nel nostro esempio)

Organizzazione della lezione

- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni

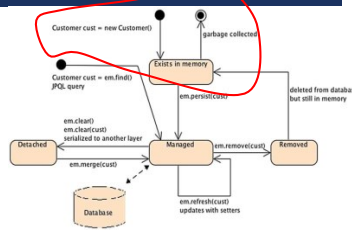
Il ciclo di vita di una entità

Customer Entity



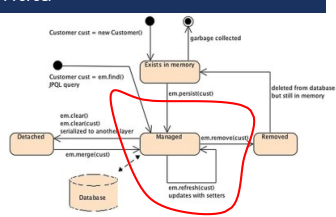
Il ciclo di vita di una entità

- 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



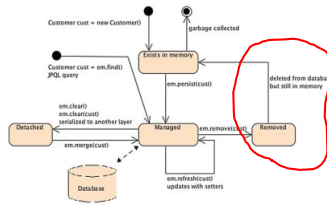
Il ciclo di vita di una entità

- 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



Il ciclo di vita di una entità

- 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

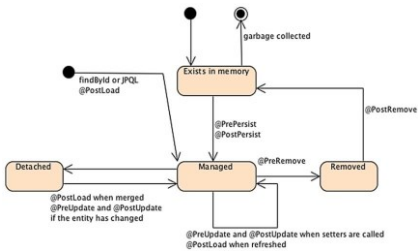


I diversi tipi di Callback

- 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.

Il ciclo di vita con gli eventi



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");
    }

    //...
```

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"

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

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)

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

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

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

Esempio con annotazioni di callback

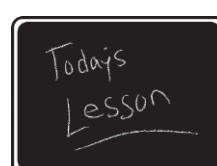
```
//...
@PostLoad
@PostPersist
@PostUpdate
public void calculateAge(){
 if(dateOfBirth == null) {
      age = null;
      return;
 }
 Calendar birth = new GregorianCalendar();
 birth.setTime(dateOfBirth);
 Calendar now = new GregorianCalendar();
 now.setTime(new Date());
 int adjust = 0;
 if(now.get(DAY_OF_YEAR) - 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 customere …
- ….la memorizza nel campo transiente age

Organizzazione della lezione

- JPQL
  - Tipi di query
- Ciclo di vita
  - Callbacks
  - Listeners
- Conclusioni



I Listeners come generalizzazione di callback

- I metodi di callback sono inglobati all'interno della definizione della entità
  - Ad esempio, nella 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`

I Listeners come generalizzazione di callback

- 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)

I Listeners come generalizzazione di callback

- 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

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

Classe standard(POJO)

```
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);
    }
```

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

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)

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

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

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**

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**

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 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

Come registrare i Listeners ad una Entità

- 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!**

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>
...
```

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>
...
```

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>
...
```

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>
...
```

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>
...
```

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>
...
```

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

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)

Corso di Laurea in Informatica
III Anno Triennale
Programmazione Distribuita

Java Persistence API (2)

Delfina Malandrino
dmalandrino@unisa.it
http://www.unisa.it/docenti/delfinamalandrino