# 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

# Esempio 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

# Esempio 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

# Esempio 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

- .

# Le query possibili con JPA 2.0

- Ci sono 5 tipi di query che permettono in contesti diversi di integrare JPQL nella applicazione Java
  1. Query Dinamiche: specificate a run-time (costose in termini di prestazioni)
  2. Named Query: query statiche definite e non modificabili
  3. Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
  4. Query native: per eseguire SQL nativo invece di JPQL
  5. 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

- Metodi di EntityManager per la creazione di query

| | |
|---|---|
| `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 |

# 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

```
    // 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

- Rivediamole
  1. Query dinamiche: specificate a run-time (costose in termini di prestazioni)
  2. Named query: query statiche definite e non modificabili
  3. Criteria API: un nuovo tipo di query object-oriented (da JPA 2.0)
  4. Query native: per eseguire SQL nativo invece di JPQL
  5. Query da stored procedure: introdotte da JPA 2.1

# Query JPA: 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()` 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()`
- Il metodo `getResultList()` restituisce una lista di *untyped objects*
  - se vogliamo una lista del tipo Customer? Bisogna usare una `TypedQuery`

# Query Dinamiche con TypedQuery

- Con Query

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

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

- Con TypedQuery

```
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 con Parametri

- Nell'esempio precedente abbiamo fatto una SELECT specificando "Betty" come `firstName`

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

- Alternative con parametri
  - volendo parametrizzare la SELECT (1)
  - oppure volendo usare un parametro di posizione (2)

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

// query (2)
query = em.createQuery("SELECT c FROM Customer c where c.firstName = ?1");
query.setParameter(1, "Betty");
List<Customer> customers = query.getResultList();
```

# Query Dinamiche con Paginazione

- 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();
```

# Named Queries

- Named Query: query statiche e non modificabili
- Meno flessibili ma più efficienti
  - 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: un esempio

- 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: un esempio

- Entità con query statiche (efficienti)
  - Query che seleziona tutti i customer dal DB
  - Query per un certo utente
  - Query con parametro

```
@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
}
```

# Named Queries con Parametri

- Come si usa
  - si crea una query dall'EM
  - si setta un parametro
  - si definisce il massimo numero di risultati (e.g., 3)
  - si esegue

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

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

# Named Query: Commenti

- 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 il parametro una stringa, errori di query sono riconosciuti a runtime
  - perdiamo la *type safety*!

# Criteria API

- 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 i customers con nome “Vincent”

1. In JPQL:

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

2. 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 una rappresentazione nella API attraverso i metodi `select()`, `from()`, e `where()`
  - questa regola è valida per ogni JPQL keyword

# Query Native

- Query native: per eseguire SQL nativo invece di JPQL
- 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();
```

# Query Native con nome

- Come le *named queries*, le *native queries* possono usare le annotazioni per definire SQL queris 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 {...}
```

# Query da Stored Procedure

- Query da Stored Procedure: introdotte da JPA 2.1
- Tutte le query viste finora 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, 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

- 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

```
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

- 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
  - i tipi di tutti i paramentri (`Date.class` and `String.class`)
  - i loro corrispondenti parameter modes (IN, OUT, INOUT, ecc)

# Stored procedure: un esempio

- Definizione della procedura, compilata nel DB
  - complessa, riguarda diverse tabelle
- Definizione di una *named stored procedure* per un'entità `Book`
  - i parametri
  - ciascuno con il loro tipo:
    - una data e una stringa

```
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
```

Procedura in SQL

```
@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.
}
```

Entity che dichiara la stored procedure

# Stored procedure: un esempio

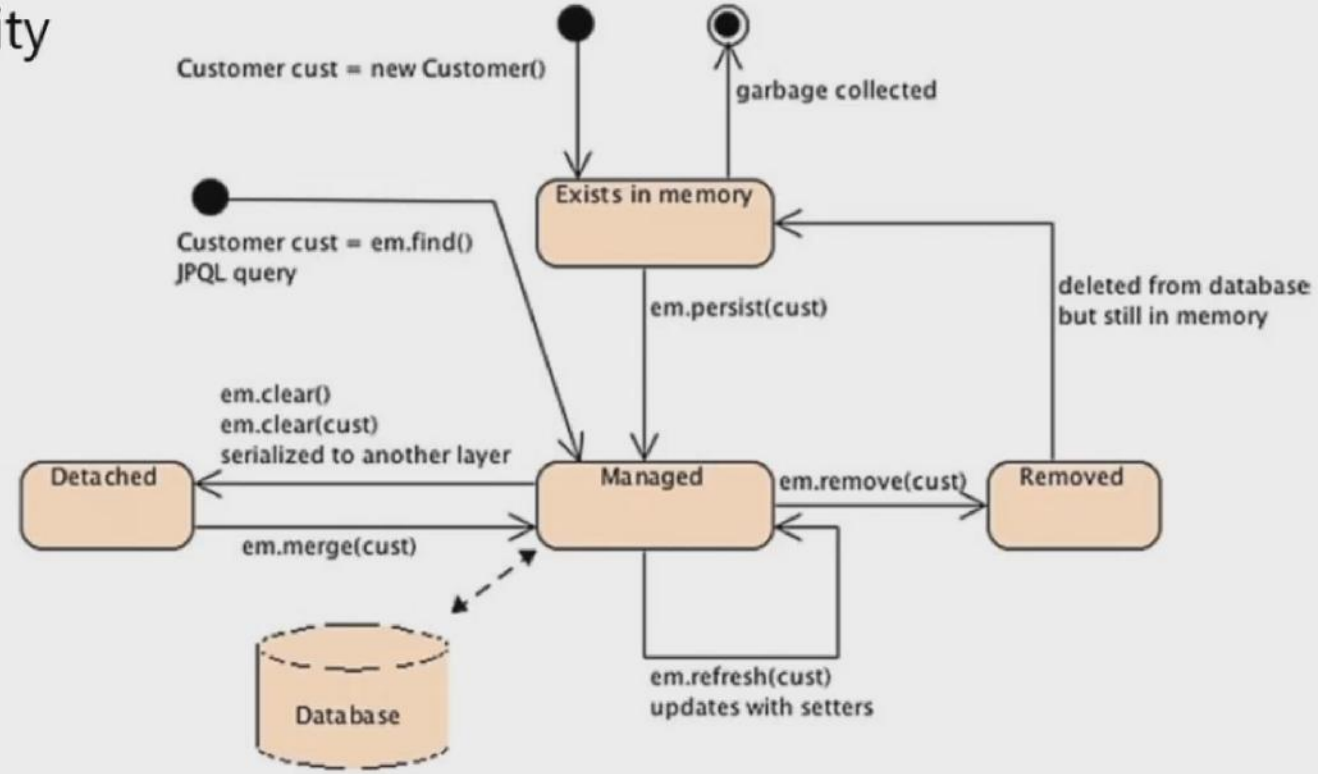
- Per invocare la stored procedure `sp_archive_books` è necessario
  - usare l'entity manager
  - creare una named stored procedure query passando il suo nome (archiveOldBooks)
  - questo restituisce una StoredProcedureQuery query sulla quale settare i parametri ed eseguire

***Listing 6-31.*** Calling a StoredProcedureQuery

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

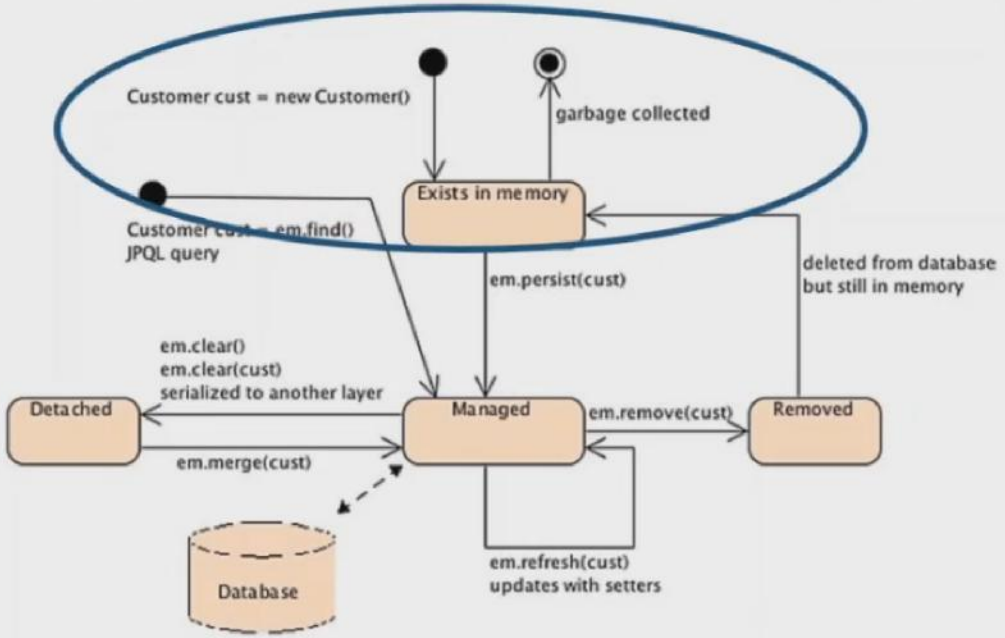
# Ciclo di Vita di un Entita

- Esempio: Customer Entity



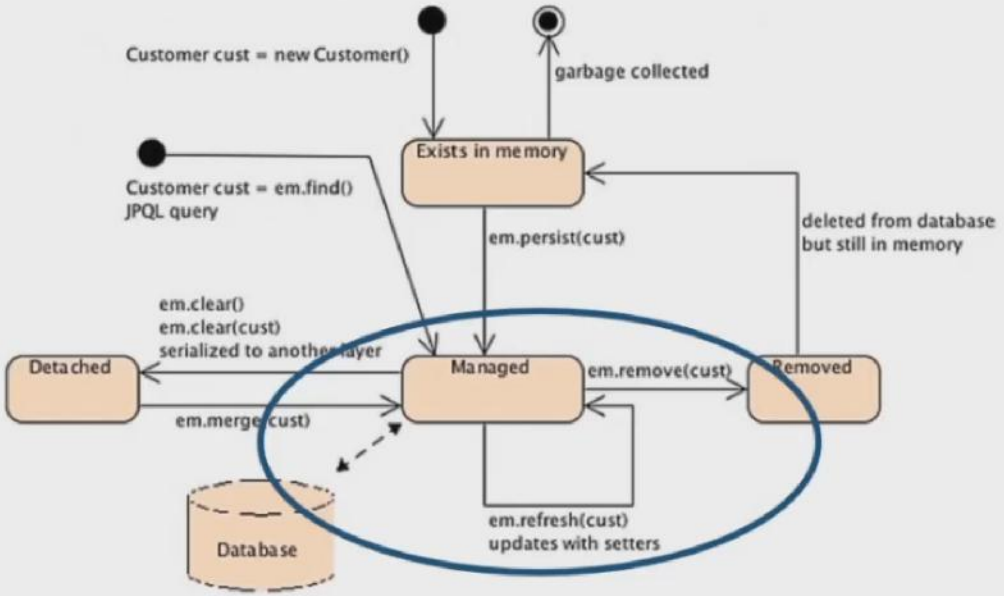
# Ciclo di Vita di un Entita

- 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



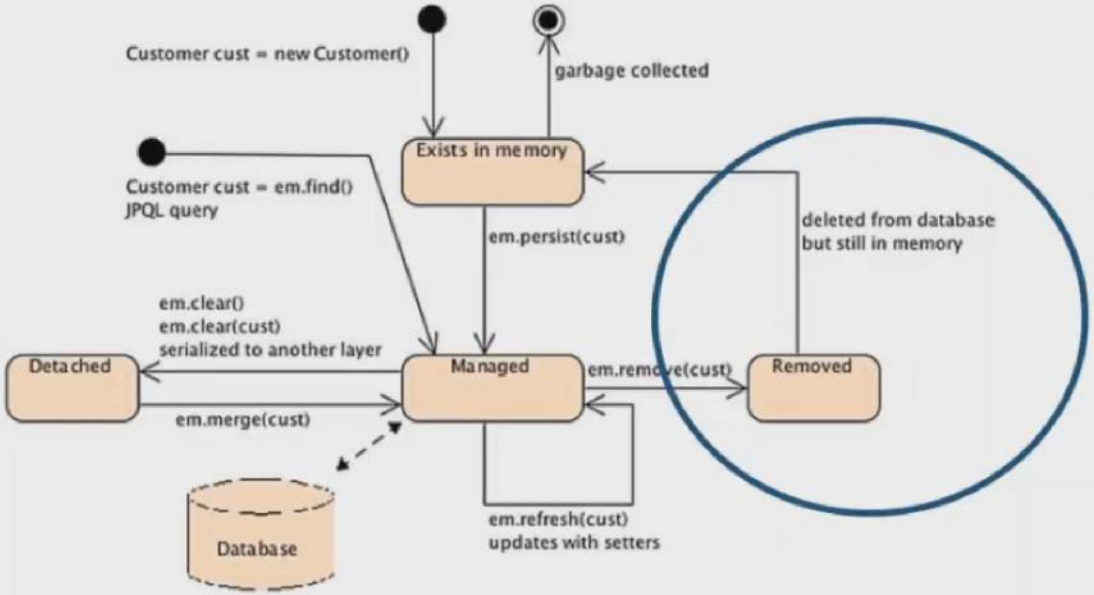
# Ciclo di Vita di un Entita

- 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



# Ciclo di Vita di un Entita

- 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



# 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` ed eventi `post` che possono essere intercettati dall'entity manager quando si deve invocare un metodo di business

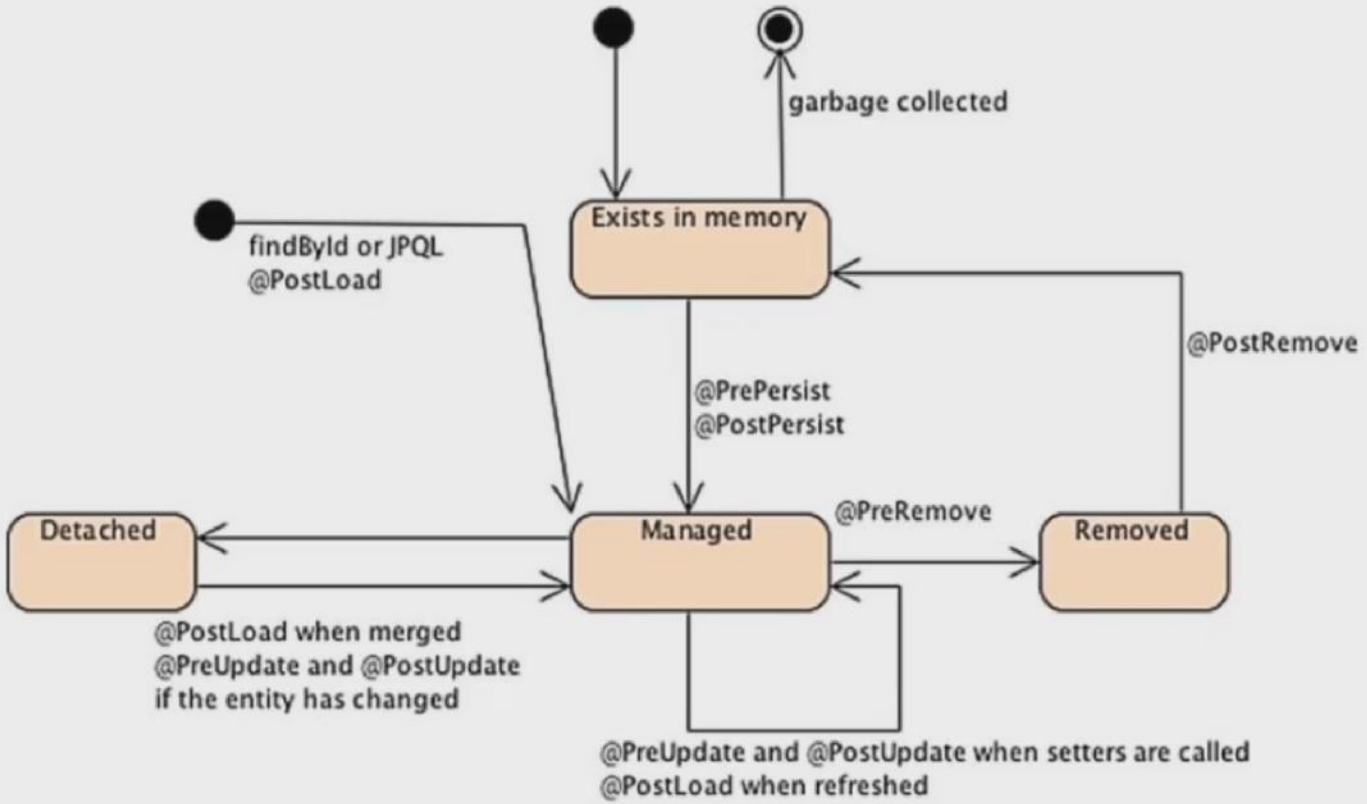
| Annotation | Description |
|---|---|
| `@PrePersist` | Marks a method to be invoked before `EntityManager.persist()` is executed. |
| `@PostPersist` | Marks a method to be invoked after the entity has been persisted. If the entity autogenerates its primary key (with `@GeneratedValue`), the value is 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. |

# 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` ed 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 annotazioni callback



# Esempio con annotazioni di callback (1)

- Entità
- Campo "temporale"
- Campo non mappato su 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

```
@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 (2)

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

```
// …
@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
```

# Listeners

- I listeners sono una generalizzazione di callback
- 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`

# Listener: un esempio per il calcolo dell'eta di un customer

- 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`

```
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);

}
```

# Listener: un esempio per la validazione

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

```
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");

}
```

# Listener: come registrare i listeners su una entita

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

```
@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
```

# Listener: un Listener per diverse entita: DebugListener

- 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!

# Listener: un Listener per diverse entita: DebugListener

- 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`)

```
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");
  }
```

- Nel file `persistence.xml`
  - Definizione del listener
    - per tutte le entità

```
...
<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>
...
```

# Riassumendo

- 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 per 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>
```

# Riassumendo

- Quando si dichiara una lista di default entity listeners, ogni listener verrà invocato nell'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`

***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>
```

# Conclusioni

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