Università di Bologna
CdS Laurea Magistrale in Ingegneria Informatica
I Ciclo - A.A. 2013/2014

Corso di Sistemi Distribuiti M

06 - Persistenza e Java Persistence API (JPA)

Docente: Paolo Bellavista
paolo.bellavista@unibo.it

http://lia.deis.unibo.it/Courses/sd1314-info/
http://lia.deis.unibo.it/Staff/PaoloBellavista/

Perché è necessario un mapping Object/Relational (O/R)?

- Una parte rilevante degli sforzi nello sviluppo di ogni applicazione distribuita di livello enterprise si concentra sul ***layer di persistenza***
  - Accesso, manipolazione e gestione di dati persistenti, tipicamente mantenuti in un DB relazionale

- Mapping O/R si occupa di risolvere il ***potenziale mismatch fra dati mantenuti in un db relazionale (table-driven)*** e il loro ***processamento tramite oggetti in esecuzione***
  - Database relazionali sono progettati per operazioni di query efficienti su dati di tipo tabellare
  - Necessità di lavorare invece tramite ***interazione fra oggetti***

Perché e che cos'é JPA?

- ***JPA è la specifica standard Java (sia JEE che JSE) del framework di supporto al mapping O/R***
- Inoltre, questo framework si occupa di gestire in modo trasparente la ***persistenza di POJO***
  - Libertà di lavorare senza i vincoli tipici del modello di dati dei db relazionali (table-driven) perché il ***potenziale mismatch O/R è gestito dal framework JPA***
- ***Oggetti persistenti*** possono seguire le ***usuali modalità di progettazione/programmazione***
- Fornisce un ***modello di persistenza "leggero"*** (capiremo più avanti l'esatto significato)
  - Modello di programmazione e di deployment
  - Performance a tempo di esecuzione
- Permette validazione e ***test al di fuori di container***
- ***Semplifica*** il modello di persistenza
  - Default e possibilità di configuraz., eliminazione deployment descriptor

Inoltre JPA…

- Supporta ricche funzionalità di modellazione di dominio
  - ***Ereditarietà e polimorfismo***
- Realizza un ***mapping O/R standard ed efficiente***
  - Ottimizzato per database relazionali
  - Con annotation standardizzate e file di configurazione XML standard
- Supporta ***capacità di querying*** ricche e flessibili
- Supporta ***l'integrazione con provider di persistenza di terze parti*** (*pluggable persistence provider*)
  - Attraverso le definizioni contenute nella cosiddetta unità di persistenza (*persistence unit*), rappresentata da persistence.xml

JPA comune a JSE e JEE

La funzionalità di persistenza è ora ***comune sia alla standard che alla enterprise edition di Java***

- Le API di persistenza sono state estese per includere ***anche l'utilizzo al di fuori di un container EJB***
- Evoluzione dalle prime versioni del supporto ad un insieme comune di API di persistenza per tutte le applicazioni Java SE e Java EE
  - Stessa JPA per applicazioni JSE, Web e EJB

Come saranno quindi supportati gli oggetti persistenti a runtime? C'è un container anche in JSE? Vedremo che si tratta di un vero e proprio caso di modello a container leggero...

Perché JPA (e non DAO)?

Prima di JPA, il tipico modo per accedere a dati era tramite ***metodi Java Data Access Object (DAO)***, ad esempio per la creazione di un nuovo dato, l'eliminazione di un dato, la ricerca su chiave primaria, …

Persistenza e transazionalità gestite a livello di programmazione

Esempio di "find by primary key"

```
public SampleDAO samplelookup(String id) {
   Connection c = null;
   PreparedStatement ps = null;
   ResultSet rs = null;
   SampleDAO dao = null;
try {
   c = getDataSource().getConnection();
   ps = c.prepareStatement("SELECT …");
   ps.setString(1, id);
   rs = ps.executeQuery();
   if (rs.first()) {
        dao = new SampleDAO(id, rs.getString(2), rs.getString(2));
   }
```

Perché JPA (e non DAO)?

```
    catch (SQLException se) {
         throw new SampleDAORuntimeException(se)); }
    } finally {
    if (rs != null) try {rs.close(); } catch (SQLException se) {}
    if (ps != null) try {ps.close(); } catch (SQLException se) {}
    if (c != null) try {c.close(); } catch (SQLException se) {}
}
return dao; }
```

Noioso e dipendente dal data-store

JPA come soluzione standard per riuscire a scrivere velocemente:

```
Sample entity = entityManager.find(Sample.class, id);
```

oppure

```
Sample entity = new Sample();
entity.setAttr1(attr1); entity.setAttr2(attr2); …
entity.setAttr(newValue);
entityManager.persist(entity);
```

Che cos'è un'entità in JPA?

- ***Plain Old Java Object (POJO)***
  - Creata attraverso l'invocazione di new() come per ogni usuale oggetto Java
  - ***Nessuna necessità di implementare interfacce*** come si doveva fare per gli Entity Bean EJB 2.x
- Può avere ***stato sia persistente che non persistente***
  - Stato non persistente (tramite ***annotazione @Transient***)
- Può fare ***sub-classing di altre classi***, sia Entity che non Entity
- È ***serializzabile***. Utilizzabile come *detached object* in altri tier (lo vedremo)
  - Non necessari oggetti specifici addizionali per il trasferimento di dati, ovvero mancata necessità di DTO espliciti

Entity (1)

Una Entity è un oggetto "leggero" appartenente a un dominio di persistenza. Usualmente rappresenta dati di un DB relazionale: ogni istanza di Entity corrisponde a una riga in una tabella

Lo ***stato persistente*** di una Entity è rappresentato ***da campi persistenti*** (o da proprietà persistenti). Questi campi/proprietà usano annotazioni per Object/Relational Mapping (OR/M): associazione con entità e relazioni per i dati mantenuti nel DB

Requisiti per classi Entity

- annotazione **javax.persistence.Entity**
- avere un ***costruttore senza argomenti***, public o protected (costruttori aggiuntivi sono ovviamente consentiti)
- non dichiarate final (nessun metodo o variabile di istanza persistente deve essere dichiarata final)

Entity (2)

Altre caratteristiche classi Entity

- ***Entity possono essere sottoclassi di classi entità oppure no***. Anche classi "normali" come sottoclassi di Entity
- ***Variabili di istanza persistenti*** devono essere dichiarate private, protected, o package-private, e ***possono essere accedute direttamente solo dai metodi della classe dell'Entity***
- Se una istanza di Entity è passata per valore (by value) come ***oggetto detached*** (vedremo che cosa vuol dire), ad esempio all'interfaccia remota di un session bean, allora la classe deve implementare l'interfaccia Serializable

Entity possono usare ***campi persistenti*** (annotazioni di mapping applicate a variabili di istanza) ***o proprietà persistenti*** (annotazioni di mapping applicate ai metodi getter per proprietà in stile JavaBean). ***NON*** si possono utilizzare annotazioni di ***entrambi i tipi*** in una singola Entity

Campi e Proprietà Persistenti

Campi persistenti

- Consentito accesso diretto alle variabili di istanza
- Tutti i campi NON annotati `javax.persistence.Transient` sono gestiti come persistenti verso il DB

Proprietà persistenti

- Si devono seguire le ***convenzioni sui metodi tipiche dei JavaBean***
- Metodi getter e setter: getProperty, setProperty, isProperty

Ad esempio, Customer entity con proprietà persistenti, con una variabile di istanza privata chiamata firstName

=> metodi `getFirstName()` e `setFirstName()`

Signature: `Type getProperty()` e `void setProperty(Type type)`

Per campi/proprietà persistenti con valori non singoli (*collection-valued*) => uso di Java Collection

Chiave Primaria e Entity

Ogni Entity deve avere un identificatore unico di oggetto. La chiave primaria di una Entity può essere ***semplice o composta***

Chiave primaria semplice: **javax.persistence.Id** annotation per indicare la proprietà o il campo chiave

Chiave primaria composita: anche un insieme di campi o proprietà persistenti. Annotazioni **javax.persistence.EmbeddedId** e **javax.persistence.IdClass**

Alcuni requisiti sulle classi di Chiave Primaria

- Costruttore di default deve essere public
- Implementare i metodi hashCode() e equals(Object other)
- Serializzabile
- Se la classe contiene campi/proprietà multiple della classe Entity, nomi e tipi dei campi/proprietà nella chiave devono fare match con quelli nella classe Entity

Chiave Primaria e Entity

Ad esempio, chiave composita costituita dai campi orderId e itemId:

```
public final class LineItemKey implements Serializable {
    public Integer orderId; public int itemId;
    public LineItemKey() {}
    public LineItemKey(Integer orderId, int itemId) {
        this.orderId = orderId;
        this.itemId = itemId; }
    public boolean equals(Object otherOb) {
        if (this == otherOb) { return true; }
        if (!(otherOb instanceof LineItemKey)) { return false; }
        LineItemKey other = (LineItemKey) otherOb;
        return ((orderId==null?other.orderId==null:
         orderId.equals(other.orderId)) &&(itemId ==
   other.itemId)); }
public int hashCode() {
        return ((orderId==null?0:orderId.hashCode())^((int)
   itemId));
    }
    public String toString() {
        return "" + orderId + "-" + itemId;
    } }
```

OR/M: Entity ed Ereditarietà

Supporto all'ereditarietà, all'associazione polimorfica e a query polimorfiche per le classi Entity

- Possono estendere classi non-Entity e classi non-Entity possono estendere entità
- Classi Entity possono essere sia astratte che concrete
- Se una query è effettuata su una Entity astratta, si opera su tutte le sue sottoclassi non astratte

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId; ... }
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary; ... }
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage; }
```

Entity ed Ereditarietà

Entity possono anche ***ereditare da superclassi che contengono stato persistente e informazioni su O/RM ma che non sono Entity*** (ovvero superclasse non decorata con @Entity). Si parla di ***Mapped Superclass***

Quando può essere utile?

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId; ... }
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary; ... }
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;  ... }
```

OR/M: Entity Inheritance e Strategie di Mapping

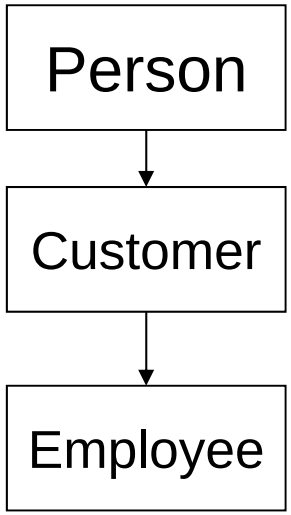
Possibile decidere ***come provider di persistenza debba fare mapping della gerarchia di classi Entity definita sulle tabelle del DB***. Annotazione javax.persistence.Inheritance

3 possibilità:

- SINGLE_TABLE - una tabella unica per l'intera gerarchia
- TABLE_PER_CLASS – come JOINED, ma ***ogni tabella contiene tutto lo stato*** per un'istanza della classe corrispondente. Una tabella per ogni classe Entity non astratta. ***Quindi rappresentazione normale?***
- JOINED – tabella differente per ogni classe. Ogni ***tabella include solo lo stato dichiarato nella sua classe*** (e riferimenti alle tabelle in gerarchia)

Default è InheritanceType.SINGLE_TABLE, usato se l'annotation @Inheritance non è specificata alla classe radice gerarchia di Entity

Strategie di Mapping: Esempi



- SINGLE_TABLE – ***tutte le proprietà in un'unica tabella***, insieme a un ***discriminator*** per stabilire tipo di entity *(perché può servire?)*
- TABLE_PER_CLASS – ogni tabella ha ***colonne con valore per ogni proprietà***, comprese quelle ereditate dalle superclassi; nessun bisogno di discriminator; schema normalizzato?
- JOINED – ogni tabella ha colonne con valore per le ***sole proprietà definite nella classe specifica***, ovvero come TABLE_PER_CLASS ma con ***schema normalizzato***

Qualche esempio di considerazioni su performance…

- Ad esempio, la strategia TABLE_PER_CLASS offre una ***scarsa efficienza nel supporto a relazioni (e query) polimorfiche***; di solito richiede query separate per ogni sottoclasse per coprire l'intera gerarchia

- Invece, utilizzando JOINED, ogni sottoclasse ha una tabella separata che contiene i soli campi specifici per la sottoclasse (la tabella non contiene colonne per i campi e le proprietà ereditati)

  => buon ***supporto a relazioni polimorfiche ma richiede operazioni di join (anche multiple) quando si istanziano sottoclassi di Entity (scarsa performance per gerarchie di classi estese)***. Analogamente, query che intendono coprire l'intera gerarchia richiedono operazioni di join fra tabelle sottoclassi

<u>Conclusione</u>: la scelta della strategia ottimale presuppone una buona conoscenza del tipo di query che si faranno sulle Entity

OR/M: Molteplicità nelle Relazioni

4 tipologie di molteplicità E/R

One-to-one: ogni istanza di Entity è associata a una singola istanza di un'altra Entity. Annotazione **javax.persistence.OneToOne** sul corrispondente campo/proprietà persistente

One-to-many: ad esempio un ordine di vendita con associati oggetti multipli. Annotazione **javax.persistence.OneToMany**

Many-to-one: viceversa, uno degli oggetti contenuti nell'ordine di vendita. Annotazione **javax.persistence.ManyToOne**

Many-to-many: annotation javax.persistence.ManyToMany

Perché questa informazione può essere utile per il provider JPA?

OR/M: Direzionalità nelle Relazioni

Una relazione può essere ***monodirezionale o bidirezionale***; determina anche come il gestore di persistenza aggiorna la relazione nel DB

Relazione bidirezionale:

Ogni entità ha un campo o una proprietà che riferisce l'altra entità, ad esempio ordine per gli oggetti in esso contenuti e oggetto per l'ordine che lo include

Devono essere rispettate alcune regole sulle annotazioni per indicare ***chi è il proprietario della relazione*** (elemento **mappedBy** delle annotazioni)

A che cosa può servire per il gestore di persistenza?

Query e direzionalità nelle relazioni:

determina se una query può navigare o meno da una entità a un'altra

Anche cascade delete relationship (dipendenza dell'esistenza, cancellare un oggetto parte di un ordine se viene cancellato l'ordine)

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

Più rilevante per noi:
Gestione Runtime di Entity

Entity sono gestite da un EntityManager, istanza di **javax.persistence.EntityManager**

Ogni istanza di EntityManager è associata con un contesto di persistenza, che definisce lo scope all'interno del quale le istanze di Entity sono create, gestite e rimosse

Contesto di Persistenza

Stessa cosa di DB?

Insieme di istanze di Entity che esistono in un particolare data-store e che sono gestite da un singolo EntityManager. ***L'interfaccia dell'EntityManager definisce i metodi che sono usati per interagire con il contesto di persistenza*** (creazione e rimozione di istanze di Entity persistenti, ritrovamento di Entity tramite chiave primaria ed esecuzione di query)

EntityManager con ***gestione a livello di container o a livello di applicazione***

Container-managed EntityManager

EntityManager a livello di Container (***Container-Managed***)

Contesto di persistenza automaticamente propagato dal container a tutti i componenti applicativi che usano l'istanza di EntityManager al'interno di una singola transazione Java Transaction Architecture (JTA)

Transazioni JTA eseguono generalmente chiamate fra componenti applicativi che, per completare la transazione, devono avere accesso a un contesto di persistenza

Succede quando ***injection di EntityManager*** tramite **@PersistenceContext**

Propagazione automatica significa che i componenti applicativi non hanno bisogno di passare riferimenti a EntityManager per produrre modifiche all'interno della singola transazione. ***È il container che si occupa del ciclo di vita degli EntityManager container-managed***

```
@PersistenceContext
EntityManager em;
```

Application-managed EntityManager

Invece, ***contesto di persistenza NON propagato ai componenti applicativi*** e ciclo di vita delle istanze di EntityManager gestito direttamente dall'applicazione

Usato quando l'applicazione necessita di ***diversi contesti di persistenza*** e di diverse istanze di EntityManager correlate

Metodo createEntityManager() di javax.persistence.EntityManagerFactory

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em = emf.createEntityManager();
```

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

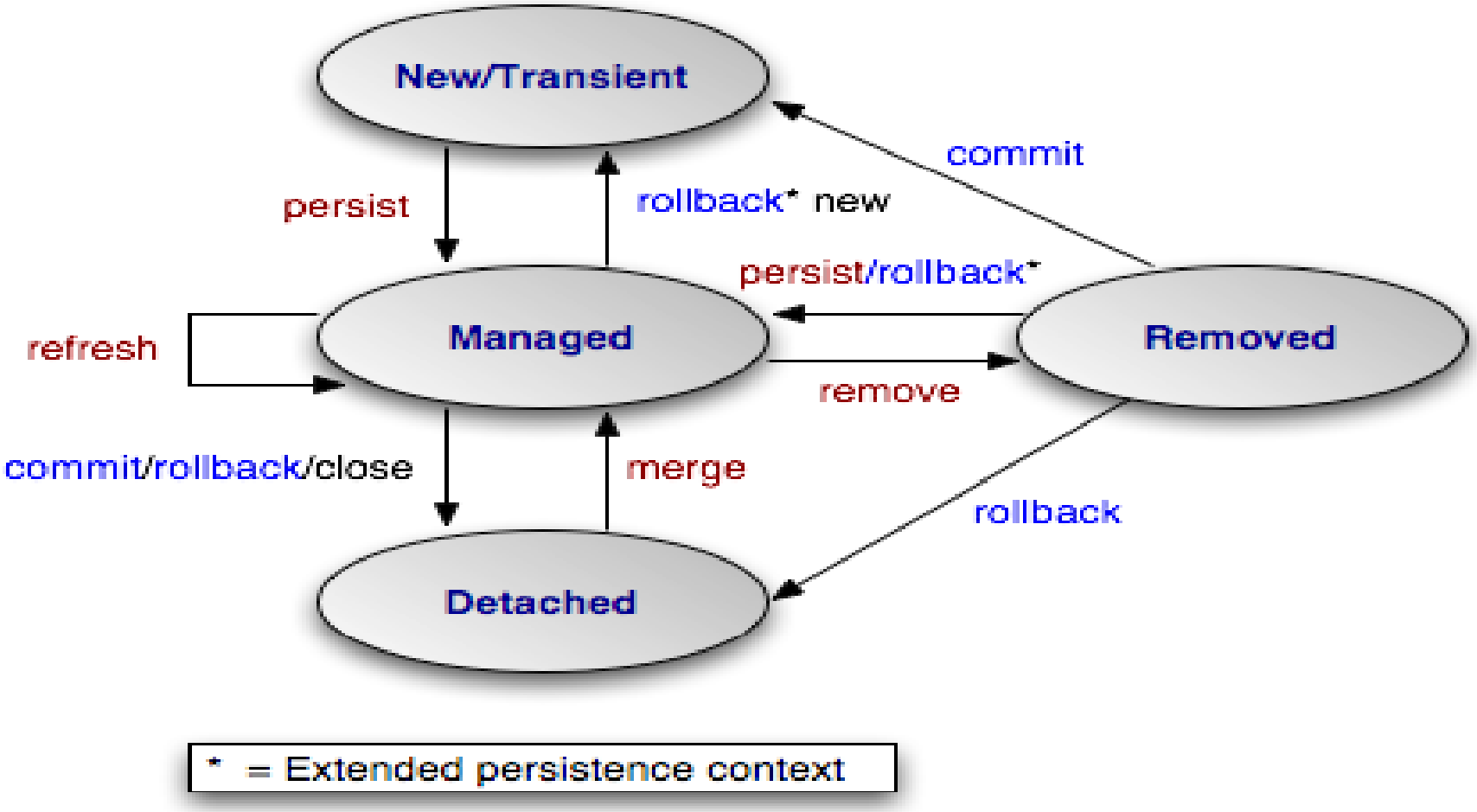
Ciclo di Vita di Entity

La gestione delle istanze di Entity viene sempre fatta tramite l'invocazione di operazioni su EntityManager

4 stati consentiti per Entity:

- ***New/Transient entity*** – nuove istanze non hanno ancora identità di persistenza e non sono ancora associate ad uno specifico contesto di persistenza
- ***Managed entity*** – istanze con identità di persistenza e associate con un contesto di persistenza
- ***Detached entity*** – istanze con identità di persistenza e correntemente (possibilmente temporaneamente) disassociate da contesti di persistenza
- ***Removed entity*** – istanze con identità di persistenza, associate con un contesto e la cui eliminazione dal data-store è stata già schedulata

Ciclo di Vita di una Entity



Ciclo di Vita di Entity

Nuove istanze di Entity diventano gestite e persistenti

- o invocando il ***metodo persist() di EntityManager***
- o tramite un'operazione persist() invocata da Entity correlate con cascade=PERSIST o cascade=ALL nelle annotation di relazione

Questo comporta che i dati ***saranno memorizzati nel DB quando la transazione associata a persist() sarà completata***

- Se **`persist()`** è invocato per una istanza di removed entity, questa ritorna nello stato di managed
- Se viene fatto su una detached entity, IllegalArgumentException o fallimento della transazione

```
@PersistenceContext
EntityManager em; ...
public LineItem createLineItem(Order order, Product product, … {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);     return li; }
// persist propagata a tutte le Entity in relazione con
// cascade element = ALL o PERSIST
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems; }
```

Ciclo di Vita di Entity

Rimozione di istanze di Entity

Managed entity possono essere rimosse tramite remove() o attraverso operazione di rimozione in cascata da Entity correlate con cascade=REMOVE o cascade=ALL

- Se **`remove()`** è invocato su new entity => operazione viene ignorata
- Se **`remove()`** è invocato su detached entity => IllegalArgumentException o fallimento del commit della transazione
- Se **`remove()`** è invocato su Entity già in stato di removed => operazione ignorata

I dati relativi alla Entity sono ***effettivamente rimossi dal DB solo a transazione completata*** o come risultato di una operazione esplicita di flush

```
public void removeOrder(Integer orderId) {
    try {
        Order order = em.find(Order.class, orderId);
        em.remove(order); }
```

Tutte le Entity oggetto associate con l'ordine sono rimosse perché cascade=ALL set nell'annotazione di relazione

Ciclo di Vita di Entity

Sincronizzazione con DB

Lo stato di entità persistenti è sincronizzato con il DB quando la ***transazione associata compie il commit***

Se una managed entity è in una relazione bidirezionale, i dati sono resi persistenti sulla base del lato di possesso (ownership) della relazione

Per forzare la sincronizzazione con DB, possibilità di invocare il ***metodo flush()***

- Con solito effetto cascade

Creazione di Query

Si utilizzano i ***metodi*** **createQuery()** ***e*** **createNamedQuery()** di ***EntityManager***, costruendo ***query conformi a Java Persistence query language***

createQuery() permette la ***costruzione di query dinamiche***, ovvero definite all'interno della business logic:

```
public List findWithName(String name) {
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

Il metodo createNamedQuery si utilizza invece per creare ***query statiche***, ovvero definite a livello di annotation. Annotazione **@NamedQuery**

Creazione di Query

Il metodo createNamedQuery si utilizza invece per creare ***query statiche***, ovvero definite a livello di annotation. Annotazione **@NamedQuery**

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName" )
@PersistenceContext
public EntityManager em; ...
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

I ***parametri con nome*** (*named parameters*) sono parametri di query preceduti da (:). Sono legati ad un valore dal metodo **javax.persistence.Query. setParameter(String name, Object value)**

```
public List findWithName(String name) {
    return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE
  :custName")
        .setParameter("custName", name)
        .getResultList(); }
```

Unità di Persistenza

Una ***unità di persistenza*** (*persistence unit*) definisce ***l'insieme di tutte le classi Entity potenzialmente gestite da EntityManager in una applicazione***. Come si differenzia da contesto di persistenza?

In altre parole, rappresenta l'insieme dei dati, di interesse per quella applicazione, contenuto in un data-store singolo

Unità di persistenza sono definite all'interno di un file XML **persistence.xml**, distribuito insieme al file EJB JAR o WAR, a seconda dell'applicazione sviluppata e secondo diverse specifiche di naming

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description> Questa unità gestisce ordini e clienti
                </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

Unità di Persistenza

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description> Questa unità gestisce ordini e clienti
               </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

Ad esempio il file sopra definisce una unità di persistenza chiamata **OrderManagement**, che usa una sorgente dati **jdbc/MyOrderDB** che è “consapevole di JTA” (*JTA-aware*)

Gli elementi jar-file e class specificano le classi relative all’unità di persistenza: classi Entity, embeddable, e superclassi mapped

L’elemento jta-data-source specifica il nome JNDI globale della sorgente dati che deve essere utilizzata dal container

Loading Lazy/Eager

Possibilità di ***controllare il caricamento*** dei dati di entità correlate ad una applicazione:

- ***eager*** - le entità correlate (ad es. indirizzo per una entity Customer che lo contenga come campo) sono ***caricate quando viene caricata l'entità "padre"***, ad es. perché coinvolta in una query

```
@OneToMany(cascade=ALL, mappedBy="owner", fetch=EAGER)
```

- ***lazy*** – entità correlate sono caricate solo quando si "naviga" effettivamente su di esse

```
@OneToMany(cascade=ALL, mappedBy="owner", fetch=LAZY)
```

=> Linea guida per massime performance: EAGER su set di dati piuttosto ristretti, LAZY altrimenti

Java Persistence Query Language

JPA definisce un ***proprio linguaggio di querying per le Entity e il loro stato persistente***. Obiettivo: scrivere ***query portabili e indipendenti dallo strumento di data-store***

Il linguaggio di query JPA usa gli schemi (*schema*) ottenuti dalla definizione delle Entity persistenti (relazioni incluse) per il modello dei dati, e definisce operatori/espressioni basati su questo modello

Lo ***scope di una query include tutte le entity correlate all'interno della stessa unità di persistenza*** (*persistence unit*)

Sintassi simile a SQL

Dettagli e sintassi del linguaggio di querying non sono di interesse per questo corso ☺…

Vista Cliente: ad es. da uno Stateless Session Bean

```
@Stateless
public class OrderEntry {

// Dependency injection di Entity Manager per l'unità
// di persistenza di interesse
@PersistenceContext
EntityManager em;

public void enterOrder(int custID, Order newOrder){

      // usa il metodo find() per trovare customer entity
      Customer c = em.find(Customer.class, custID);
      // aggiunge un nuovo ordine
      c.getOrders().add(newOrder);
      newOrder.setCustomer(c);
   }

// altri metodi di business
}
```

Vista Cliente: ad es. da un cliente Java SE

```
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntity-
 ManagerFactory("EmployeeService");
EntityManager em = emf.createEntityManager();

Collection emps = em.createQuery("SELECT e FROM
 Employee e").getResultList();
```

…

Dal punto di vista delle API, quindi, EntityManager…

- ***EntityManager di JPA*** svolge funzionalità simili a Hibernate Session, Java Data Objects (JDO) PersistenceManager, …

- ***Controlla e gestisce il ciclo di vita di oggetti Entity***
  - persist() – inserisce una Entity nel DB
  - remove() – rimuove una Entity dal DB
  - merge() – sincronizza lo stato di entità cosiddette detached
  - refresh() – ricarica lo stato dal DB

Esempio: Operazione Persist

```
public Order createNewOrder(Customer customer) {
  // Crea una nuova istanza dell'oggetto - entity è in
   // stato New/Transient
   Order order = new Order(customer);

     //  Dopo l'invocazione del metodo persist(),
     //  lo stato viene cambiato in managed. Al prossimo
     //  flush o commit, le nuove istanze persistenti saranno
     //  inserite nella tabella del DB

   entityManager.persist(order);
   return order;
}
```

Esempio: Operazioni Find&Remove

```
public void removeOrder(Long orderId) {
   Order order =
           entityManager.find(Order.class, orderId);

   entityManager.remove(order);
   // Le istanze verranno eliminate dalla tabella
   // al prossimo flush o commit. L'accesso a una
   // entity già rimossa restitisce risultati non definiti

}
```

Esempio: Operazione Merge

```
public OrderLine updateOrderLine(OrderLine orderLine)
   {

    // Il metodo merge restituisce una copia managed
    // della data entity di tipo detached.
    // Le modifiche fatte allo stato persistente
    // dell'entity detached sono applicate a questa
    // istanza managed
   return entityManager.merge(orderLine);
}
```

Listener di Entità

- Si possono definire ***listener o metodi di callback*** che saranno ***invocati dal provider di persistenza alla transizione*** fra diversi stati del ciclo di vita delle Entity
- Per usare metodi di callback
  - Occorre ***annotare i metodi di callback desiderati nella classe della Entity*** o definirli all'interno di una ***classe listener separata***
  - Annotazioni correlate
    - PrePersist / PostPersist
    - PreRemove/ PostRemove
    - PreUpdate / PostUpdate
    - PostLoad

Esempio di Listener per Entity

```
@Entity
@EntityListener(com.acme.AlertMonitor.class)
public class AccountBean implements Account {
Long accountId;
Integer balance;
boolean preferred;
@Transient ClassA obj1;

public Long getAccountId() { ... }
public Integer getBalance() { ... }
   public boolean isPreferred() { ... }
public void deposit(Integer amount) { ... }
public Integer withdraw(Integer amount) throws
NSFException {... }
```

a che cosa serve?

Esempio di Listener per Entity

```
@PrePersist
public void validateCreate() {
   if (getBalance() < MIN_REQUIRED_BALANCE)
      throw new AccountException("Insufficient
         balance to open an account");    }

@PostLoad
public void adjustPreferredStatus() {
   preferred =(getBalance() >=
               AccountManager.getPreferredStatusLevel());   }
}
```

Altra classe

```
public class AlertMonitor {

@PostPersist
public void newAccountAlert(Account acct) {
   Alerts.sendMarketingInfo(acct.getAccountId(),
                            acct.getBalance());    }
}
```

Giusto per fissare i concetti in modo pratico…

Qualche esempio concreto di caso d'uso e motivazioni per:

- ***singolo*** entity manager?
- entity manager ***multipli verso differenti*** db?
- entity manager ***multipli verso stesso*** db?

E inoltre:

- ***Invalidazione di entity*** nel contesto di persistenza - esattamente quando?
- ***Find di entity removed*** - che cosa succede?

Esempi e Demo

Oltre agli esempi nelle esercitazioni guidate,
vedi anche utili esempi e demo all'URL (specialmente esercizi 1 e 4):

http://download.oracle.com/javaee/5/tutorial/doc/bnbpy.html
http://www.jpox.org/docs/1_2/tutorials/jpa_tutorial.html
http://wiki.eclipse.org/EclipseLink

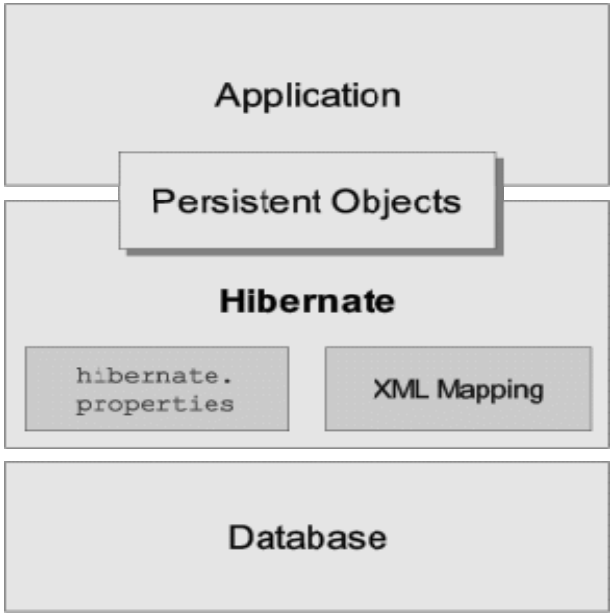
e ovviamente tanti altri esempi di applicazioni che sono disponibili su libri vari e Web…

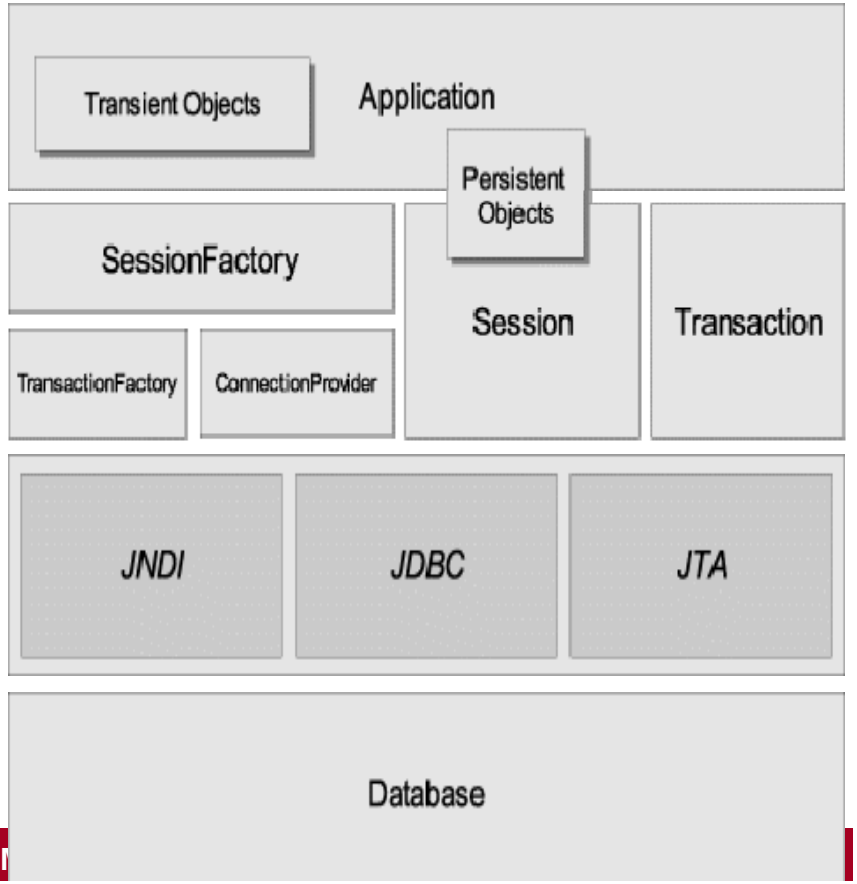
E per quanto riguarda ***Hibernate***, la buona notizia è che, data la vostra profonda conoscenza di JPA ☺, rimane poco da aggiungere…

Hibernate in un paio di slide…

- Framework O/RM per realizzare la ***persistenza in modo trasparente per POJO***
  - Nessun vincolo derivante dal modello orientato alle tabelle dei DB relazionali; gestione mismatch O/R
- Permette la costruzione e l'utilizzo di oggetti persistenti seguendo gli usuali concetti di programmazione OO
  - Associazione/utilizzo e composizione
  - Ereditarietà e polimorfismo
  - Collection per relazioni "multiple"
- ***Performance e Object caching***
- Supporto sofisticato a querying
  - Criteria, Query By Example (QBE)
  - Hibernate Query Language (HQL) e SQL nativo

Hibernate in un paio di slide…





L'architettura di Hibernate permette di ***astrarre dalle API JDBC/JTA sottostanti***: livello di applicazione può essere trasparente a questi dettagli

Principali Oggetti del Framework Hibernate

- ***SessionFactory***
  - Classe **org.hibernate.SessionFactory**
  - Factory per oggetti ***Session*** e cliente di **ConnectionProvider**
  - ***Tipicamente una factory per ogni DB***
  - Si occupa di mantenere una ***cache di primo livello*** associata all'oggetto Session e usata di default
  - Può mantenere una ***cache opzionale di secondo livello***, associata all'oggetto SessionFactory, con dati riusabili in diverse transazioni, anche a livello di cluster
- ***Session***
  - Classe **org.hibernate.Session**
  - Rappresenta un ***contesto di persistenza*** e la sua vita è delimitata ***dall'inizio e dalla fine di una transazione logica***
  - Gestisce le ***operazioni del ciclo di vita*** degli oggetti persistenti
  - Fa da factory per oggetti **Transaction**

Principali Oggetti del Framework Hibernate

- ***Oggetti persistenti***
  - ***Oggetti single-threaded*** che contengono ***stato persistente e logica di business***
  - Possono essere normali JavaBean o POJO – unico aspetto peculiare è che ***devono essere associati con un (esattamente uno) oggetto Session***
  - Modifiche fatte sugli oggetti persistenti sono automaticamente riportate sulle tabelle DB (al loro commit)
  - Appena si chiude la sessione, gli stessi oggetti diventano automaticam. “detached” e possono essere usati liberamente
- ***Oggetti transient e detached***
  - Istanze di classi persistenti che ***non sono correntemente associati a nessuna Session***
  - Possono essere stati istanziati dall’applicazione e non essere ancora diventati persistenti, oppure derivare da una Session chiusa
  - …

Principali Oggetti del Framework Hibernate

- ***Oggetti transient e detached***
  - Modifiche fatte su questi oggetti ***NON si riflettono sul DB***
  - Operazioni di persist o merge per farli tornare "persistenti" (con modifiche correlate ad aggiornamenti tabelle DB)
- ***Transazioni***
  - Classe **org.hibernate.Transaction**
  - Oggetto single-threaded usato dall'applicazione per specificare ***unità atomiche di lavoro***
  - Permette di ***astrarre dai dettagli dei sottostanti meccanismi transazionali*** (JDBC, JTA, CORBA, …)
  - Un oggetto Session può essere coinvolto in diverse Transaction
  - La ***demarcazione delle transazioni deve comunque essere specificata esplicitamente***

Stato degli Oggetti Persistenti

Una istanza di una classe persistente può assumere in ogni istante uno fra ***tre stati possibili***, definiti all'interno di un ***contesto di persistenza***

- ***transient*** (non appartenente a un contesto di persistenza)
- ***persistent*** (appartenente a un contesto di persistenza)
- ***detached*** (usualmente appartenente a un contesto di persistenza ma non in questo momento corrente)

- ***Stato transient***
  - Istanza ***MAI associata con una sessione*** (contesto di persistenza)
  - ***NON ha identità di persistenza*** (valore per primary key)
  - Non ha righe corrispondenti nel DB
  - Ad es. quando un'istanza POJO viene creata fuori da una sessione

Stato degli Oggetti Persistenti

- ***Stato persistent***
  - Istanza ***correntemente associata con una sessione***
  - Ha un'identità di persistenza e usualmente una riga corrispondente in una tabella DB
  - Ad es. quando un oggetto viene creato in una sessione o un oggetto transient viene fatto diventare persistente
- ***Stato detached***
  - Istanza che ***è stata associata a un contesto di persistenza in passato***, ma quella sessione è stata chiusa oppure l'istanza è stata trasferita tramite serializzazione a un altro processo
  - Ha identità di persistenza e forse una riga corrispondente in una tabella DB
  - Ad es. quando un oggetto deve essere inviato ad un altro processo, che lo utilizzerà senza necessità di avere un contesto di persistenza associato

Transizioni di Stato

- Oggetti Transient possono diventare persistenti tramite chiamate ai metodi **save(), persist() o saveOrUpdate()**, ovviamente dell'oggetto Session associato
- Oggetto Persistent possono diventare transienti tramite l'invocazione di **delete()**
- Ogni istanza di oggetto persistente restituita da **get() o load()** è Persistent
- Oggetti Detached possono diventare persistenti tramite chiamate ai metodi **update(), saveOrUpdate(), lock() o replicate()**
- Lo stato di una istanza Transient o Detached può transitare in Persistent anche tramite **merge()** e istanziazione di un nuovo oggetto persistente

Hibernate Caching

Utilizzato per motivi di performance

Idea di base:

- Rendere l'accesso al DB necessario solo quando si devono reperire dati non disponibili già sulla cache
- L'applicazione può avere bisogno di ***svuotare (invalidare)*** la cache se il DB viene aggiornato o se non è possibile sapere se la cache mantiene ancora copie aggiornate

- ***First-level cache***
  - Associata con l'oggetto Session
- ***Second-level cache***
  - Associata con l'oggetto SessionFactory

Cache di Primo e di Secondo Livello

- ***Cache di primo livello***

Usata da Hibernate all'interno dei ***confini di una singola transazione*** (spesso associata a singola session)

  - Principalmente al fine di ***ridurre il numero di query SQL generate all'interno di una transazione***
  - Ad es. se un oggetto è modificato diverse volte all'interno della medesima transazione, Hibernate genera un unico statement SQL UPDATE alla fine della transazione, con tutte le modifiche

- ***Cache di secondo livello***

Mantiene dati ***a livello di Session Factory, utilizzabili da diverse transazioni*** *(across transaction boundaries)*

  - Oggetti persistenti disponibili per l'intera applicazione, non solo per l'utente che sta eseguendo le query e per il SessionBean associato

Quale protocollo per aggiornamento/invalidazione?

Optimistic Concurrency Control

Hibernate sfrutta Optimistic Concurrency Control (OCC) per motivi di performance

- OCC si basa sull'assunzione che ***la maggior parte delle transazioni verso DB non sono in conflitto*** con altre transazioni; questo permette di essere piuttosto "permissivi" nel rilasciare la possibilità di esecuzione
- Se i conflitti sono rari, ***post-validazioni possono essere svolte efficientemente*** => alto throughput
- OCC sfrutta ***versioning dei dati*** per ottenere elevati gradi di concorrenza e alta scalabilità
- Version checking sfrutta numeri di versione o timestamp per fare la detection di aggiornamenti che possono determinare conflitti

Hibernate Version Checking

Hibernate (e ora anche JPA) implementa il concetto di ***version-based OCC***; ***non del tutto automatico***

Una proprietà può essere marcata con @Version

Darà origine a una colonna speciale nel datastore

```
@Entity
@Table(name = "orders")
public class Order {
   @Id private long id;
   @Version private int version;
   private String description; private String status;  … }
```

Quando il gestore di persistenza "salva" questa entity in datastore, la proprietà version viene posta automaticamente a valore iniziale.

Ogni volta Hibernate aggiorna l'entità, ***query effettivamente eseguita dal persistence manager*** è:

```
update orders set description=?, status=?, version=? where id=? and
   version=?
```

Hibernate Version Checking

Ad es. se due utenti (Alice e Bob) caricano un ordine con versione 1…

Alice decide di ***approvare l'ordine*** e lancia tale azione => lo ***stato*** è ***aggiornato*** sul datastore

```
update orders set description=?, status=?, version=2 where id=? and
   version=1
```

Il fatto di persistere l'aggiornamento del dato incrementa version counter a 2

Nel frattempo Bob, ad esempio nella sua GUI, ha ancora la ***vecchia versione dei dati (version=1)***

Quando lancia un update all'ordine, ***query eseguita risulta essere***:

```
update orders set description=?, status=?, version=2 where id=? and
   version=1
```

Il risultato è che questa seconda update ***NON fa match con alcuna riga (no match con clausola WHERE)***

Hibernate lancia una eccezione `org.hibernate.StaleObjectStateException` (wrapped in `javax.persistence.OptimisticLockException`)

Hibernate Fetching

- ***Hibernate può adottare diverse strategie di fetching***. Una strategia di fetching determina ***come e quando i dati vengono effettivamente caricati dal DB*** per un'applicazione che usa gli oggetti di persistenza associati
- Strategia di fetching adottata ha ovviamente impatto sulle ***performance*** ottenibili
- La strategia adottata viene di solito dichiarata in un file di mapping o ne viene fatto l'override tramite specifiche query
- Modalità di fetching
  - ***FetchMode.DEFAULT*** (configurazione del mapping file)
  - ***FetchMode.JOIN*** (Hibernate recupera i dati associati, anche collezioni, utilizzando un `outer join` all'interno della stessa `select`)
  - ***FetchMode.SELECT*** (Hibernate effettua una seconda `select` separata per recuperare le entity o collection associate)

  Lazy fetching è il default per SELECT: la seconda `select` viene eseguita solo quando l'applicazione accede veramente ai dati associati

Funzionalità Avanzate di Querying: l'Esempio di QBE

Query By Examples (QBE)

- Stile drasticamente differente da SQL per la ricerca dati nel DB
- ***Idea di base – Ricerca Associativa***
  - Popolare parzialmente una istanza di un oggetto
  - Permettere a Hibernate di costruire trasparentemente un criterio di scelta (criteria) ***usando l'istanza come esempio***

Ad esempio, se la classe **org.hibernate.criterion.Example** implementa l'interfaccia **Criterion**

```
// Cerca gli oggetti persona tramite un oggetto di esempio
    Criteria crit = sess.createCriteria(Person.class);
    Person person = new Person();
    person.setName("Shin");
    Example exampleRestriction = Example.create(person);
    crit.add( exampleRestriction  );
    List results = crit.list();
```