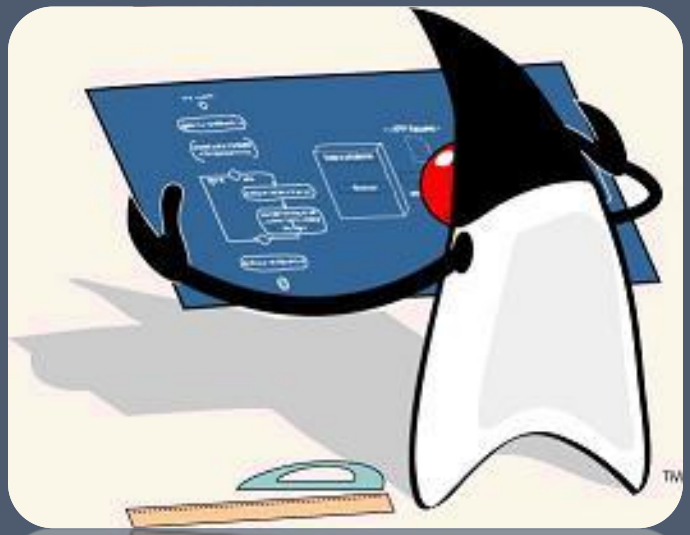
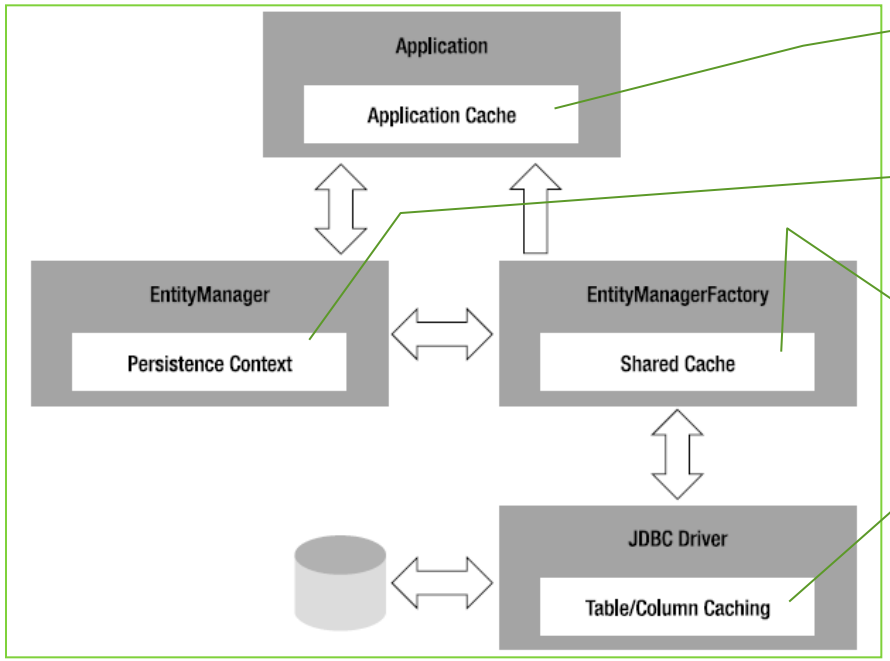


JPA
JAVA PERSISTENCE API

Tecniche avanzate

Caching

- Per caching si intende la capacità di JPA di mantenere un entity o lo stato di un entity nella cache
- In generale la cache è utilizzata in diversi livelli ed in diversi modi come illustrato dalla seguente figura



cache sviluppata dal programma (sconsigliata). NON fa parte di JPA

il persistence context è la cache di primo livello sempre presente in JPA

L'entityManagerFactory gestisce una shared cache (detta anche cache di secondo livello)

eventuale cache del driver. NON fa parte di JPA

Caching

- La cache di secondo livello è completamente trasparente alla applicazione
  - E' utile solo per aumentare le performance
- JPA non obbliga i provider ad avere una cache di secondo livello.
  - Normalmente, anche se supportata, i providers hanno un meccanismo per istallarla, abilitarla o disabilitarla
  - JPA definisce le regole per la cache → quindi indipendentemente dalla sua presenza l'applicazione resta portabile
- Se la cache non è supportata dal provider o non istallata, semplicemente non ha effetto
  - Tutti i metadati relativi alla cache di secondo livello vengono ignorati

Shared cache

- JPA supporta una shared cache di oggetti Entity attraverso l'interfaccia **javax.persistence.Cache.**
- Invocando il metodo EntityManagerFactory.getCache() si ottiene un oggetto che implementa l'interfaccia Cache
  - Se il provider non la supporta, viene comunque tornato un oggetto. In questo caso, le operazioni che fanno uso della cache non hanno effetto
- L'interfaccia Cache possiede i seguenti metodi :
  - **contains(Class cls, Object id)** → torna true se nella cache ci è un entity con un certo id istanziato da una certa precisa classe (no sottoclassi)
  - **evict(Class cls, Object id)** → rimuove dalla cache una istanza con un certo id. Agisce solo sulle classe e NON sottoclasse
  - **evict(Class)** → rimuove dalla cache tutte le istanze di una certa classe. Agisce su classe e sottoclasse
  - **evictAll()**
- NOTA: l'applicazione non dovrebbe manipolare la cache attraverso le API.

Configurazione statica della cache

- E' possibile configurare la shared cache attraverso
  - configurazione del persistence unit
  - configurazione della classe

- Nel file persistence.xml la configurazione si effettua tramite l'elemento shared-cache-mode
  - Si ottiene lo stesso effetto settando la proprietà javax.persistence.sharedCache al momento della creazione del EntityManagerFactiry

Shared cache

- La proprietà ha seguenti *5* possibili valori

Nome proprietà	descrizione
UNSPECIFIED (default)	Il provider è libero di abilitare o disabilitare la shared cache
ALL	Completamente abilitata (@Cacheable non ha effetto)
NONE	Completamente disabilitata (@Cacheable non ha effetto)
DISABLE_SELECTIVE	Abilita la cache per tutti gli entities; gli entity possono disabilitarla usando la annotazione @Cacheable(false)
ENABLE_SELECTIVE	Logica opposta a DISABLE_SELECTIVE

- Dire che il default è UNSPECIFIED significa che il provider è libero di abilitare la cache o non abilitarla di default

Shared cache

- DISABLE_SELECTIVE si utilizza in collaborazione con @Cacheable posta a livello di classe entity
- Di default tutti gli entities sono cacheabili; quindi si pone l'annotazione @Cacheable(false) sulle classi entity di cui non si vuole la cache
- Applicando @Cacheable(true/false) su una classe, questa ha effetto anche sulle sottoclassi
  - le sottoclassi possono a loro volta porre l'annotazione per cambiare comportamento
- Scrivere @Cacheable equivale a scrivere @Cacheable(true)

Shared cache

- Nel seguente esempio, ipotizzando che nel persistence.xml la proprietà shared-cache-mode è DISABLE_SELECTIVE

```
@Entity
@Cacheable(true)
public class Fornitore{ ... }

@Entity
@Cacheable(false)
public class Ordine{ ... }

@Entity
public class Prodotto{ ... }
```

DISABLE_SELECTIVE significa che tutte le classi sono abilitate per la cache

Ordine viene disabilitata

quali saranno le classi che abilitate per la cache di secondo livello? **Fornitore e Prodotto**

Gestione dinamica della cache

- Quanto configurato può essere sovrascritto dinamicamente attraverso due proprietà
  - **javax.persistence.cache.retriveMode** →indicando che un entity venga recuperato dalla cache al momento di eseguire una query
  - **javax.persistence.cache.storeMode** → indicando che un entity deve andare in cache dopo averlo recuperato dal database

- Le proprietà possono essere passate al metodo find() come hint ad una query

NOTA: per sovrascrivere dinamicamente la configurazione di un entity, per tale entity la cache deve essere abilitata via configurazione (statica)

Gestione dinamica della cache

- **javax.persistence.cache.retriveMode**
  - CacheRetrieveMode.USE (default) → per usare la cache quando un entity è letto dal database
  - CacheRetrieveMode.BYPASS → per bypassare la cache quando un entity è letto dal database. L'entity non viene cercato nella cache

Nota: l'unico motivo per cui esiste l'opzione USE è per tornare al default dopo aver usato l'opzione BYPASS

Gestione dinamica della cache

- **javax.persistence.cache.storeMode**
  - CacheStoreMode.USE (default) → pone gli oggetti in cache quando un entity è letto dal database o committato sul database
  - CacheStoreMode.BYPASS → per bypassare la cache quando un entity è letto dal database. L'entity non viene inserito nella cache
  - CacheStoreMode.REFRESH → fa in modo che gli oggetti già presente nella cache siano refreshati al successivo accesso ad DB.

Lifecycle Callbacks

- JPA gestisce alcuni metodi di callback in base ad eventi occorsi sul ciclo di vita
- Gli eventi posso essere suddivisi in 4 categorie: persisting, updating, removing, e loading
  - **gli eventi corrispondono alle effettive operazioni sul database**
- Per ogni categoria esiste una coppia di metodi pre/post.
  - Eccezione per la categoria loading che possiede solo il metodo post
- Possono essere presenti altri metodi di callback ma sono vendor specific

Operazioni consentite

- Non tutte le operazioni di persistenza sono consentite nei metodi di callback. Se scatta un metodo di callback su un entity A:
  - I metodi di callback non dovrebbero invocare i metodi dell'EntityManager ne eseguire Query, accedere ad altre istanze Entity
  - non dovrebbero modificare campi di relazione di A
  - possono modificare campi non di relazione del entity A
  - possono invocare JNDI, JDBC, JMS, e enterprise beans.
  - possono sollevare unchecked exception. Se accade in una transazione, la transazione è marcata per il rollback e la catena di invocazione dei metodi di callback si interrompe
- Il codice gira del metodo gira nello stesso contesto transazionale e di sicurezza del metodo che ha ha provocato l'esecuzione del metodo di callback

Persisting

- **PrePersist** → scatta quando viene invocato il metodo persist() o merge() e questi provoca un inserimento di un entity nel persistence context
  - viene invocato anche sugli oggetti che si rendono persistenti a cascata
  - se più oggetti vengono resi persistenti a cascata, non è possibile predire l'ordine con cui verranno invocati

- **PostPersist** → scatta quando l'entity viene effettivamente inserito nel DB (tipicamente alla chiusura della transazione)
  - il fatto che il metodo postPersist viene invocato NON vuol dire che la transazione è stata committata. Dopo l'esecuzione di postPersist la transazione può andare in rollback

Removing

- **PreRemove** → scatta quando viene invocato il metodo remove e questi predispone l'entity ad essere eliminato
  - viene invocato anche sugli oggetti che si rimuovono a cascata
  - se più oggetti vengono rimossi a cascata, non è possibile predire l'ordine con cui verranno invocati

- **PostRemove** → scatta quando l'entity viene effettivamente rimosso dal DB (tipicamente alla chiusura della transazione)
  - il fatto che il metodo postRemove viene invocato NON vuol dire che la transazione è stata committata. Dopo l'esecuzione di postRemove la transazione può andare in rollback

Updating

- **PreUpdate→** considerato che non vi è uno specifico metodo per la modifica e che la modifica di un entity può avvenire in qualsiasi momento, JPA garantisce soltanto che il metodo preUpdate parta prima della effettiva operazione sul DB

- **PostUpdate→** scatta quando l'entity viene effettivamente modificato sul DB (tipicamente alla chiusura della transazione)
  - il fatto che il metodo postUpdate viene invocato NON vuol dire che la transazione è stata committata. Dopo l'esecuzione di postUpdate la transazione può andare in rollback

- A causa della ottimizzazione interna al provider, non vi è nessuna garanzia che i metodi perUpdate e postUpdate vadano in esecuzione quando:
  - si modifica un entity e poi si rimuove nella stessa transazione
  - si rende persistente un entity e poi si modifica nella stessa transazione

Loading

- **PostLoad** →scatta quando i dati vengono letti dal db e l'istanza entity viene costruita
  - Qualsiasi operazioni che causa una lettura dal DB provocherà l'esecuzione del metodo postLoad (caricamento lazy, refresh(), query…)
  - viene invocato anche sugli oggetti che si leggono con cascata REFRESH
  - se più oggetti vengono letti a cascata, non è possibile predire l'ordine con cui verranno invocati

Annotazioni

- Le annotazioni sono:
  - @PrePersist
  - @PostPersist
  - @PreUpdate
  - @PostUpdate
  - @PreRemove
  - @PostRemove
  - @PostLoad
- si utilizzano per annotare i metodi delle classi Entity
- un metodo può essere annotato con più annotazioni di callback
- ci può essere solo una annotazione di callback di un certo tipo all'interno di una classe Entity

Metodi di callbak

- Un metodo di callback deve seguire le seguenti regole:
  - può avere qualsiasi nome
  - deve ritornare void
  - non deve avere nessun parametro
  - non può essere final o static
  - non può avere la clausola throws

- Considerato che non è possibile aggiungere la clausola throws, un metodo può sollevare solo una unchecked exception
  - Se si verifica una exception, l'invocazione dei successivi metodi di callback viene interrotta e la transazione corrente (se esiste) va in rollback

Ambiente Enterprise

- I metodi di callback possono essere definiti anche in ambiente enterprise
- JPA non interrompe o attiva eventuali altre operazioni in essere per eseguire un metodo di callback
  - operazioni come transazioni, naming o sicurezza

NOTA: fare attenzione a quando scattano i metodi: dipende dalle regole presenti in ambiente enterprise, soprattutto quelle legate alla apertura e chiusura della transazione

Entity Listener

- Si utilizzano quando si vuole gestire gli eventi di callback fuori dalle classi Entity in modo da condividerli tra più entities
- Una classe di tipo Entity Listener non è un entity
- Valgono le stesse regole definite per gli entity
  - un metodo può essere annotato con più annotazioni di callback
  - ci può essere solo una annotazione di callback di un certo tipo all'interno di una classe Listener

- La firma dei metodi deve avere come parametro un oggetto di tipo compatibile (classe, superclasse, interfaccia) con l'entity su cui è occorso l'evento
  - ad esempio public void fire(Object o) va sempre bene

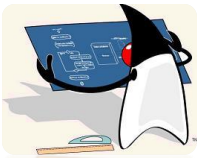
Entity Listener

- Una classe Entity Listener deve essere stateless
  - non è utile definire proprietà della classe
  - una singola istanza può essere condivisa tra più entity
- La classe deve avere un costruttore a zero parametri
- Per definire un Entity Listener per un certo entity si utilizza l'annotazione @EntityListeners
  - E' possibile definire più Entity Listeners per una classe Entity. I metodi di callback verranno invocati nell'ordine in cui sono scritte le annotazioni
  - Se vi sono metodi di callback nella classe Entity questi vengono invocati DOPO quelli presenti nelle classi Entity Listeners
  - Se un metodo di callback va in (runtime) exception, tutta la catena di invocazione si interrompe e gli altri metodi non verranno invocati

Default Listeners

- Nel caso una o più operazioni di callback vadano applicate a tutti gli entities, è possibile configurare un Entity Listener in modo che venga applicato a tutti gli entities.
  - Questi prendono il nome di Default Listener
- La configurazione è vendor specific e può essere fatta solo via XML
- E' possibile disabilitare i listeners di default su un entity attraverso l'annotazione @ExcludeDefaultListeners
  - i metodi di callback dei default listeners non scatteranno per tutte le istanze di quel tipo

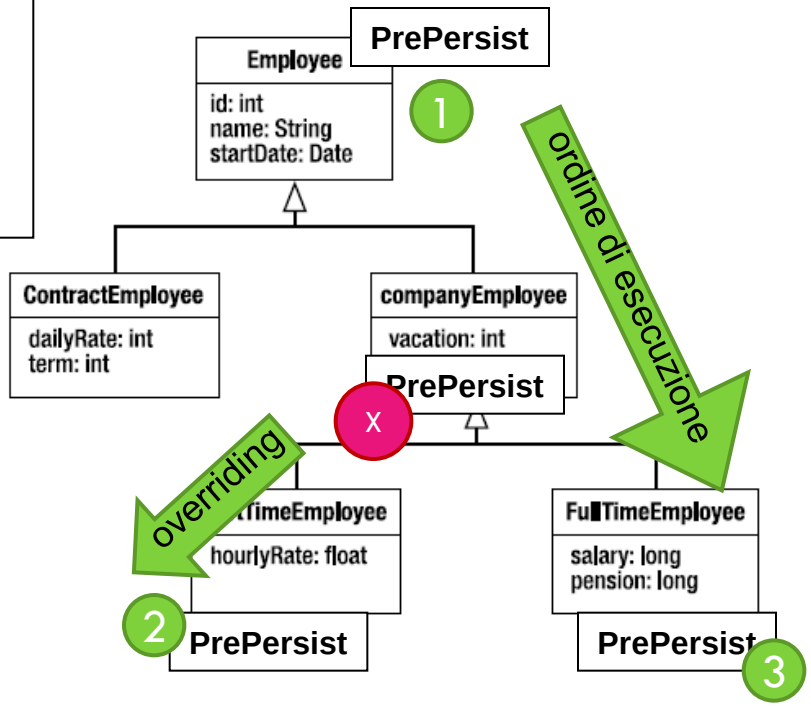
metodi di callback e ereditarietà



- I metodi di callback possono essere inclusi sia nelle classi entity che nelle classi mapped superclass sia concrete che astratte **ma NON nelle classi transient (non mappate)**

Se vi sono più metodi di callback dello stesso tipo lungo la catena, vengono eseguiti a partire dalla classe più generale fino alla classe più specifica

Nell'esempio viene eseguito prima il metodo in Employee, poi quello in CompanyEmployee e poi quello in FullTimeEmployee
Siccome in PartTimeEmployee vi è un metodo in overriding, quello di CompanyEmployee non viene eseguito a favore di quello in PartTimeEmployee



Entity listeners e gerarchie

- Come per i metodi di callback, i listeners possono essere inclusi sia nelle classi entity che nelle classi mapped superclass sia concrete che astratte

- Aggiungendo un listener all'interno di una gerarchia:
  - metodo in overriding → overriding del listener
  - metodo non in overriding → non si escludono gli altri listener dichiarati (no overraiding)

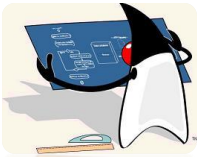
- Per escludere i listener presenti nelle superlcassi si utilizza l'annotazione @ExcludeSuperclassListeners
  - Questi provoca la mancata esecuzione dei listener presenti nelle superclassi relativamente ai metodi di callback presenti nella classe annotata.

Riepilogo invocazioni

- Se si verifica un evento X su un entity A, il provider:
  1. **Verifica se vi sono default listeners definiti.** Nel caso, itera su di loro nell'ordine in cui sono definiti e cerca i metodi che sono annotati per l'evento X. Se li trova li esegue.
  2. **Cerca i listener definiti nella classe entity o mapped superclass più alta nella gerarchia (@EntityListeners).** Nel caso, itera su di loro nell'ordine in cui sono definiti e cerca i metodi che sono annotati per l'evento X. Se li trova li esegue.
  3. Ripete il punto 2 per tutte le classi della gerarchia fino a raggiungere A. Poi esegue lo stesso processo per A stesso.
  4. **Cerca i metodi di callback relativi all'evento X definiti nella classe entity o mapped superclass più alta nella gerarchia.** Li esegue a meno che non vi sia un metodo di callback il overriding nella classe A.
  5. Ripete il punto 4 per tutte le classi della gerarchia fino a raggiungere A.
  6. **Cerca i metodi di callback relativi all'evento X nella classe A**

Esempio


```
public interface NamedEntity {
   public String getName();
}
```



cosa accade se viene invocato il metodo persist() su un oggetto dei vari tipi di questa gerarchia?

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@EntityListeners(NameValidator.class)
public class Employee implements NamedEntity {
   @Id private int id;
   private String name;
   @Transient private long syncTime;
   public String getName() { return name; }
   @PostPersist
   @PostUpdate
   @PostLoad
   private void resetSyncTime() {
      syncTime =  System.currentTimeMillis();
   }
   // ...
}
```

Supponiamo non vi siano default listeners

```
@Entity
@ExcludeSuperclassListeners
@EntityListeners(LongNameValidator.class)
public class ContractEmployee extends Employee {
   private int dailyRate;
   private int term;
   @PrePersist
   public void verifyTerm() { ... }
   // ...
}
```

```
@MappedSuperclass
@EntityListeners(EmployeeAudit.class)
public abstract class CompanyEmployee extends Employee {
   protected int vacation;
   // ...
   @PrePersist
   @PreUpdate
   public void verifyVacation() { ... }
   // ….
}
```

Nota per l'esame

- Le domande basate sulle invocazioni dei metodi di callback fanno riferimento all'uso di JPA **in ambiente non gestito**
  - **Quindi, alla chiusura della transazione il persistence context non si chiude ne si svuota.**
- Ad esempio, nel seguente scenario:

1. Creates the entity
2. Begins a transaction
3. Persists the entity
4. Commits the transaction
5. Begins another transaction
6. Removes the entity
7. Merges the entity
8. Commits the transaction.

al punto 5, l'entity si trova nello stato managed

Locking

- Con locking si intende la possibilità di acquisire locks sulle righe delle tabelle al fine di gestire correttamente ed efficacemente le operazioni sul DB

- Tuttavia, in JPA vi sono alcune tecniche che si basano sull'idea di ridurre in numero di locks sulle tabelle a favore di tecniche di locking in memoria
  - Tali tecniche prevedono la possibilità di conflitti a livello tabellare.

- L'incremento o la riduzione del numero di locks sulle tabelle dipende dal livello di isolamento della transazione sul DB

Optimistic locking

- Il lock ottimistico è una tecnica che si basa sulla premessa che un entity modificato in una transazione, difficilmente verrà modificato contemporaneamente da un'altra transazione

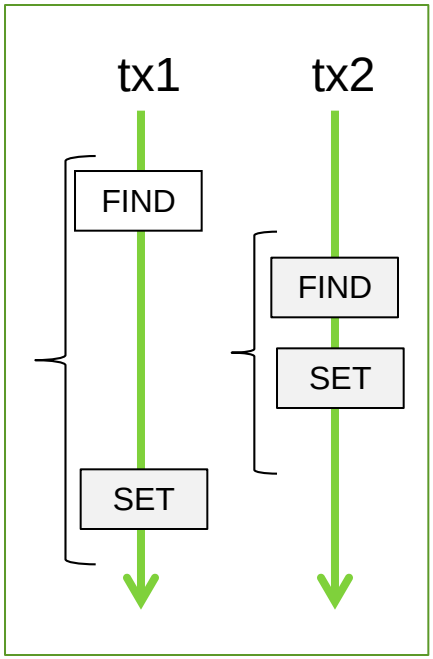
- Questo porta a evitare di acquisire locks sulla tabella fino al momento di sincronizzare i dati sul DB, tipicamente alla fine della transazione
  - al momento della sincronizzazione (al flush() o alla chisura della transazione) si controlla che non siano occorse variazioni ai dati sul db mentre l'entity veniva modificato.
  - Se questo è avvenuto, viene sollevata una exception di tipo OptimisticLockException

Esempio

- Di seguito un esempio che richiede l'uso del lock ottimistico

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
   @PersistenceContext(unitName="EmployeeService")
   EntityManager em;

   public void deductEmployeeVacation(int id, int days) {
       Employee emp = em.find(Employee.class, id);
       int currentDays = emp.getVacationDays();
       // Do some other stuff like notify HR system, etc.
       // ...
       emp.setVacationDays(currentDays - days);
   }
}
```



- tx1 non appone locks e quindi tx2 è libero di leggere e modificare i dati. In tal caso l'ultimo che committa sovrascrive i dati → **LAST COMMIT WIN**
  - Il lock ottimistico prevede invece che tx2 committa e tx1 solleva una exception pur non apponendo locks →**FIRST COMMIT WIN**

Versioning

- La tecnica prevede la presenza di un campo aggiuntivo alla classe Entity mappato con una colonna nella tabella
  - Questo campo prende il nome di version e l'entity si dice versionato

```
@Entity
public class Employee {
   @Id private int id;
   @Version private int version;
   // ...
}
```

e' fondamentale la presenza di una colonna aggiuntiva nella tabella altrimenti la tecnica non è implementabile

- Il campo può essere di tipo int, short, long, i corrispondenti tipi wrapper e java.sql.Timestamp
- Il campo può essere letto ma non deve mai essere modificato dall'applicazione

Campo @Version

- per il campo usato per il versioning valgono le seguenti regole:
  - non può essere pubblico
  - non può essere static o final
  - deve essere mappato nella tabella primaria dell'entity
  - deve essere una campo singolo. Usare più campi per il versioning rende l'applicazione non portabile

Esempio di bulk operation

- In caso di bulk operation
  - il campo di versione non viene modificato (ne nell'oggetto ne sulla riga)
    - Alcuni provider forniscono il supporto alle bulk operations modificando il campo dell'entity ma ciò non è garantito.
    - Dovrebbe essere quindi la bulk operation stessa a modificare il campo di versione
  - opera su classi e sottoclassi
  - gli oggetti coinvolti non vengono portati nel persistence context per operazioni di update e delete

```
UPDATE Employee e
SET e.salary = e.salary + 1000, e.version = e.version + 1
WHERE EXISTS (SELECT p FROM e.projects pWHERE p.name = 'Release2')
```

campo @Version

- Il campo di versionamento viene automaticamente modificato soltanto in questi due casi:
  - uno dei campi non di relazione viene modificato
  - la propria chiave esterna della relazione (lato owner) viene modificata (cioè relazioni uno-a-uno e molti-a-uno)
- E' possibile fare in modo che il campo venga modificato anche per le relazioni uno-a-molti e negli altri casi ma bisogna attuare tecniche apposite (optimistic write locking)

Tipi di lock

- Nell'ambito del lock ottimistico (e dunque l'uso del campo di versione) esistono diversi livelli in base alle necessità

- In sostanza, rinunciando all'uso di un livello di isolamento appropriato (e quindi ai locks espliciti sul DB) passando a simulare tali livelli attraverso tecniche ottimistiche basate sul numero di versione

- quindi lasciamo il livello di isolamento **read committed** (cioè basso) e al suo posto usiamo varie strategie di lock ottimistico

Lock mode

- E' possibile passare ai seguenti metodi il lock mode:
  - EntityManager.lock()— pone un lock esplicito sull'oggetto già presente nel persistence context
  - EntityManager.refresh()—pone un lock ad un oggetto nel persistence context prima di refresh
  - EntityManager.find()—pone un lock sull' oggetto ritornato (e presente nel persistence context)
  - Query.setLockMode()—pone il lock mode che avrà effetto durante l'esecuzione della query
- Questi metodi di EntityManager devono essere eseguiti in una transazione.
- refresh() e lock() devono essere applicati su oggetti già presenti nel p.c.
- Il metodo Query.setLockMode() può essere invocato in qualsiasi momento ma la query deve essere invocata all'interno di una transazione

Optimistic read locking

- Il livello di isolamento successivo a read committed è repeteable reads: una transazione che legge due volte gli stessi dati non li trova modificati tra una lettura e un'altra a causa della modifica di un'altra transazione
  - Questo comporta che se una transazione accede a dei dati che poi vengono modificati da un'altra transazione, ad almeno una delle due deve essere impedito di committare

- L'alltributo LockModeType.OPTIMISTIC attua l'optimistic read locking che fornisce questo livello di isolamento.
  - pone un lock in lettura sull'oggetto

NOTA: LockModeType.OPTIMISTIC è stato introdotto con JPA2 in sostituzione di LockModeType.READ (che comunque è ancora supportato)

Esempio

- Questo è il caso in cui una transazione legge un entity e un' altra transazione lo modifica.

```
public SalaryReport generateDepartmentsSalaryReport(List<Integer> deptIds) {
   SalaryReport report = new SalaryReport();
   long total = 0;
   for (Integer deptId : deptIds) {
      long deptTotal = totalSalaryInDepartment(deptId);
      report.addDeptSalaryLine(deptId, deptTotal);
      total += deptTotal;
   }
   report.addSummarySalaryLine(total);
   return report;
}
```

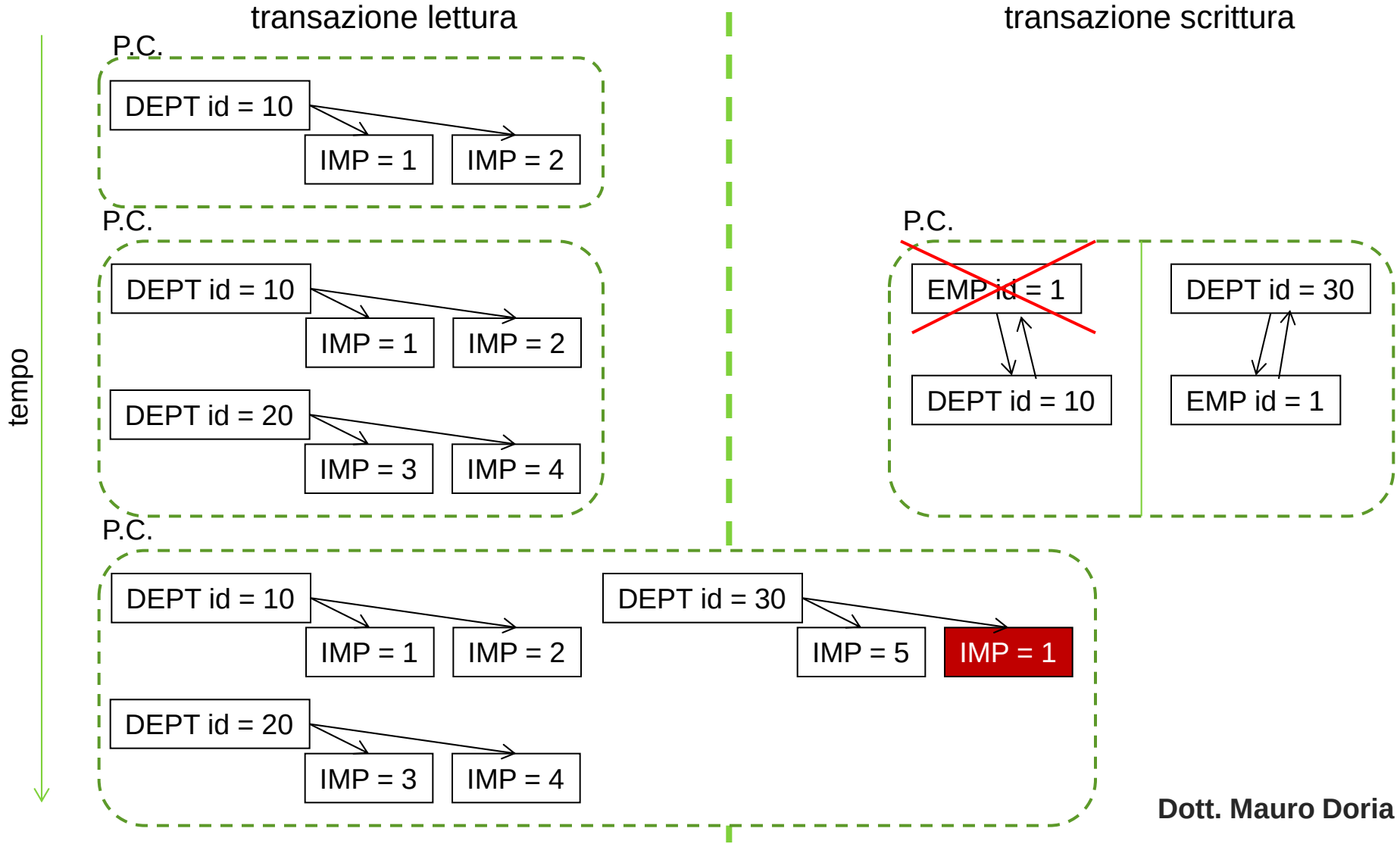
in questo esempio una transazione calcola i salari degli impiegati di n dipartimenti mentre un'altra transazione sposta gli impiegati da un dipartimento all'altro

il metodo totalSalaryIn Department(id) effettua un find al suo interno

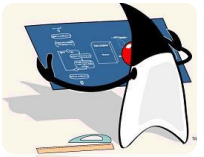
```
public void changeEmployeeDepartment(int deptId, int empId) {
   Employee emp = em.find(Employee.class, empId);
   emp.getDepartment().removeEmployee(emp);
   Department dept = em.find(Department.class, deptId);
   dept.addEmployee(emp);
   emp.setDepartment(dept);
}
```

- Cosa succede se mentre si fa il calcolo del salario un impiegato cambia dipartimento?
  - Nel calcolo, il salario di tale impiegato potrebbe essere conteggiato due volte

Esempio


transazione lettura
transazione scrittura
P.C.
DEPT id = 10
IMP = 1
IMP = 2
P.C.
DEPT id = 10
IMP = 1
IMP = 2
DEPT id = 20
IMP = 3
IMP = 4
P.C.
EMP id = 1
DEPT id = 10
DEPT id = 30
EMP id = 1
P.C.
DEPT id = 10
IMP = 1
IMP = 2
DEPT id = 20
IMP = 3
IMP = 4
DEPT id = 30
IMP = 5
IMP = 1
tempo

Esempio



- Notare che questo scenario di errore non capita sempre
  - se la transazione che legge precarica tutti gli entities Department prima di avviare il calcolo questo non succede
  - se la transazione che scrive, cambia il dipartimento di un impiegato non ancora conteggiato, non si verifica nessun problema

NOTA: il problema si può risolvere in molti modi, primo fra tutti alzare il livello di isolamento da read committed a repeteable reads. Ma i metodi di tipo lock di JPA sono utili proprio nel caso questo non si volglia o non si possa fare.

Esempio

- La soluzione è porre un lock read su ogni impiegato letto nella transazione in modo da bloccare l'oggetto.
  - Se una seconda transazione modifica l'oggetto, la prima solleva una exception

```
protected long totalSalaryInDepartment(int deptId) {
   long total = 0;
   Department dept = em.find(Department.class, deptId);
      for (Employee emp : dept.getEmployees()) {
                em.lock(emp, LockModeType.OPTIMISTIC);
                total += emp.getSalary();
      }
      return total;
}
```

  - da notare che questo comporta la repeteable read: se si legge lo stesso oggetto due volte nella stessa transazione, si ottengono gli stessi dati
    - si fa per dire, considerato che la cache di primo livello non consente di caricare due oggetti con lo stesso id.

Optimistic write locking

- LockModeType.OPTIMISTIC_FORCE_INCREMENT consente di avere le caratteristiche di optimistic read locking ma aggiunge una maggiore restrizione
  - intuitivamente, pone un lock in scrittura sull'oggetto
  - In sostanza, il numero di versione viene incrementato preventivamente, cioè indipendentemente da come usiamo l'oggetto (lettura o scrittura)
    - Da qui la parola FORCE_INCREMENT

NOTA: LockModeType.OPTIMISTIC_FORCE_INCREMENT è stato introdotto con JPA2 in sostituzione di LockModeType.WRITE (che comunque è ancora supportata)

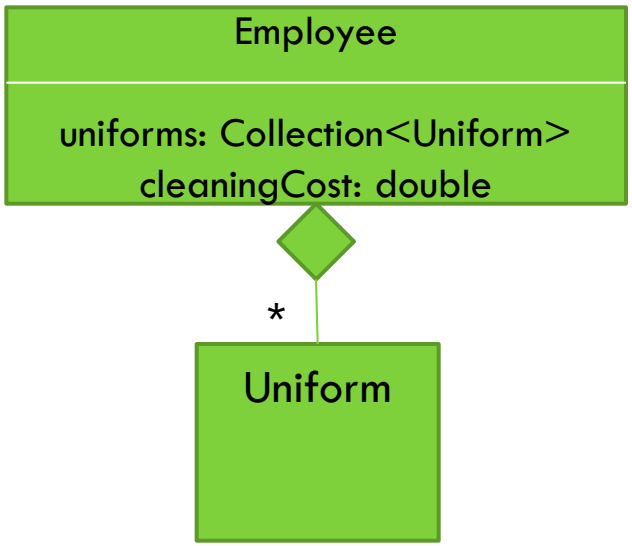
Optimistic write locking

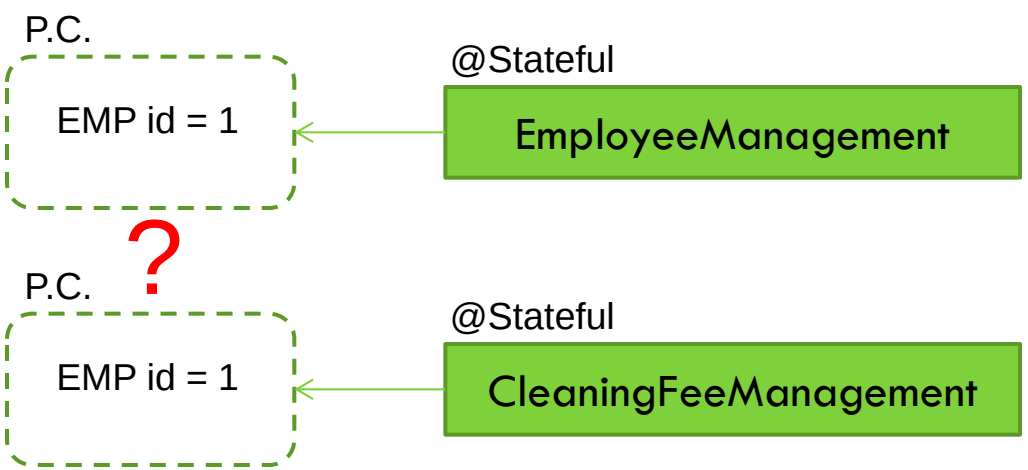
- L'optimistic write locking si utilizza nel caso si voglia un incremento del numero di versione anche quando si modifica un elemento di una relazione (ed esempio uno-a-molti con la chiave esterna lato target)
  - quando i puntatori nella collection cambiano mentre nella tabella no (chiave esterna dall'altro lato)

- Questo perché l'oggetto entity ha un lock in scrittura e qualsiasi altra transazione che agisce su di esso andrà in exception

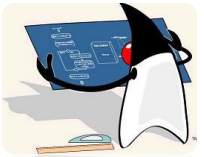
Esempio

- In questo esempio, un impiegato possiede diverse uniformi le quali vengono mandate a lavare regolarmente.
  - Il campo numerico cleaningCost rappresenta i costi per il servizio lavanderia che l'impiegato deve corrispondere a fine mese.
  - A fine mese viene calcolato il costo di pulizia di tutte le uniformi e quindi viene modificato il campo cleaningCost
- Supponiamo di avere 2 Stateful bean con extended persistence context: uno gestisce gli impiegati e l'altro i costi del servizio di pulizia

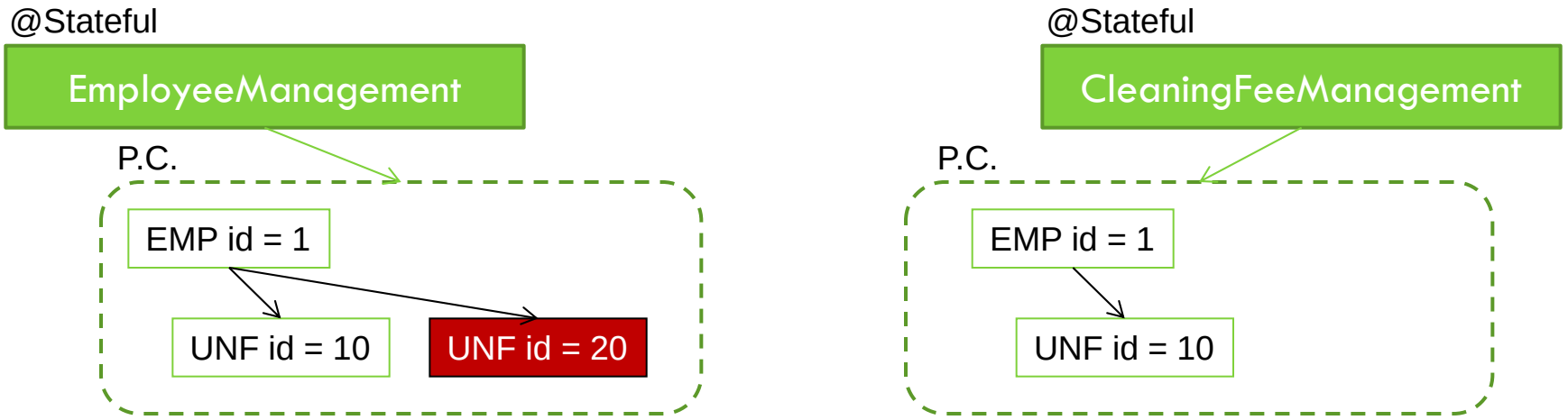




Esempio



- che succede se entrambi agiscono sullo stesso impiegato caricandoli (uno a testa) nel proprio persistence context?



- inizialmente entrambi leggono l'impiegato ID=1 con una uniforme ID=10. Che succede se EmployeeManagement aggiunge l' uniforme ID=20?
  - EmployeeManagement aggiunge l'uniforme alla collectione e committa la transazione. L'operazione si riflette sul DB
  - CleaningFeeManagement calcola i costi basati su una sola uniforme e modifica il campo cleaning cost e committa la transazione.

Esempio

- Non vi verificano eccezioni di versione perché l'aggiunta dell'uniforme comporta una modifica all'oggetto uniforme e quindi il numero di versione dell'impiegato con ID=10 non viene modificato
  - questo perché la chiave esterna si trova nella tabella Uniforme
  - Come risultato, l'impiegato paga per la pulizia di una uniforme invece che per due
- La soluzione è adottare un oprimistic write locking che modifica il numero di versione dell'impiegato a monte
  - la transazione che calcola i costi fallirà non appena tenterà di modificare il campo cleaningCost

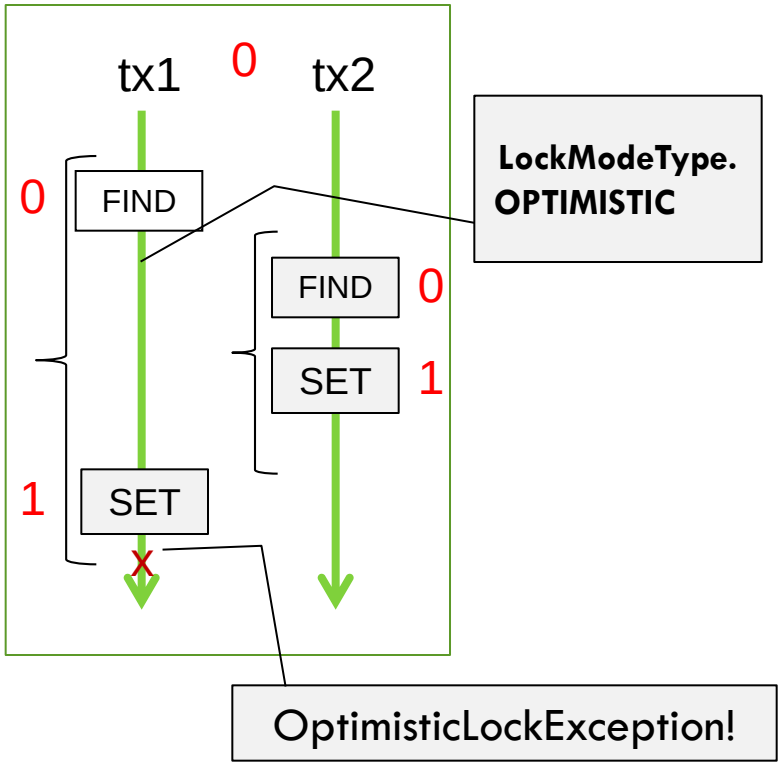
Esempio


```
@Stateful
public class EmployeeManagementBean implements EmployeeManagement {
  @PersistenceContext(unitName="EmployeeService",type=PersistenceContextType.EXTENDED)
   EntityManager em;
   public void addUniform(int id, Uniform uniform) {
      Employee emp = em.find(Employee.class, id);
      em.lock(emp, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
      emp.addUniform(uniform);
      uniform.setEmployee(emp);
   } // ...
}
```

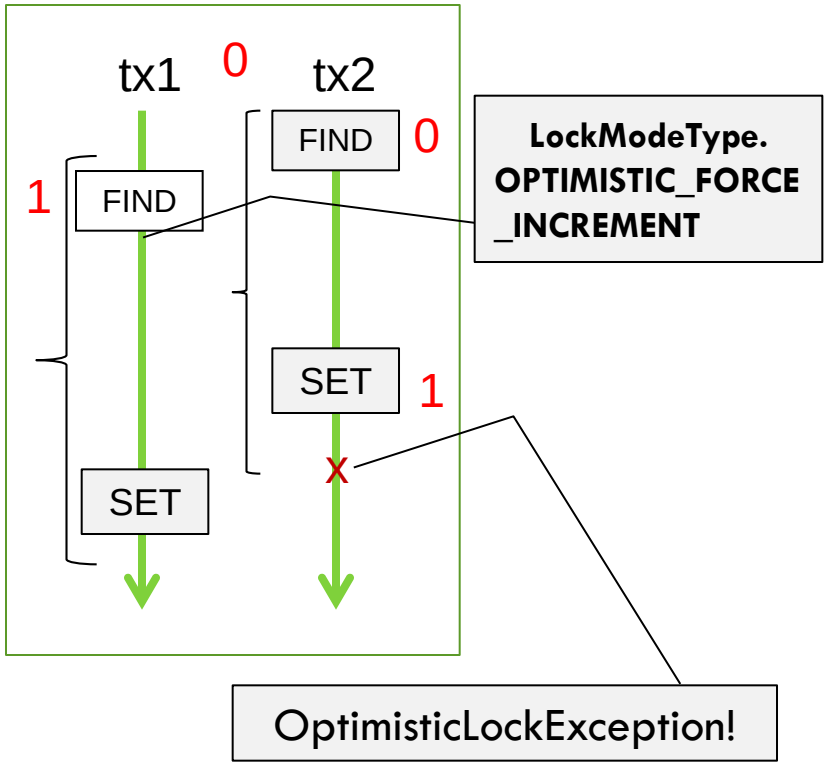
```
@Stateful
public class CleaningFeeManagementBean implements CleaningFeeManagement {
  static final Float UNIFORM_COST = 4.7f;
 @PersistenceContext(unitName="EmployeeService", type=PersistenceContextType.EXTENDED)
 EntityManager em;
 public void calculateCleaningCost(int id) {
   Employee emp = em.find(Employee.class, id);
   Float cost = emp.getUniforms().size() * UNIFORM_COST;
   emp.setCost(emp.getCost() + cost);
 } // ...
}
```

Differenze tra read e write

optimistic read locking



optimistic write locking



NOTA: OPTIMISTIC_FORCE_INCREMENT potrebbe non riuscire a scrivere nella colonna sul DB a causa di locks già presenti sulla riga o versioni del DBMS

Pessimistic Locking

- Si passa ad una soluzione pessimistica quando quella ottimistica non è utilizzabile
  - I conflitti sono molto frequenti e comportano "costi" di recovery dalle OptimisticLokingException troppo elevati
- Lock pessimistico significa che viene posto un lock in modo che le operazioni su quel oggetto saranno esclusive e non potranno subire interferenze da nessuna altra transazione
  - Vengono posti locks sulle tabelle
  - Una transansazione che usa lock pessimistico NON DOVREBBE MAI andare in exception a causa di un conflitto di concorrenza sull'oggetto
  - non dovremmo mai aver bisogno di gestire una exception di tipo OptimisticLokingException

Pessimistic write loking

- Si ottiene invocando il metodo LockModeType.PESSIMISTIC_WRITE
- Equivale ad una SELECT FOR UPDATE in SQL
- javax.persistence.lock.timeout è vendor specific

- Se si specifica una modalità lock pessimistico su un oggetto versionabile il persistence provider deve attuare i controlli di versione nel momento in cui ottiene il lock.
  - Se I controlli falliscono si solleva una OptimisticLockException

Esempio

Consideriamo il seguente esempio

```
@Stateless
public class VacationAccrualBean implements VacationAccrualService {

  @PersistenceContext(unitName="Employee")
  EntityManager em;

  public void accrueEmployeeVacation(int id) {
    Employee emp = em.find(Employee.class, id);
    // Find amt according to union rules and emp status
    EmployeeStatus status = emp.getStatus();
    double accruedDays = calculateAccrual(status);
    if accruedDays > 0 {
            em.lock(emp, LockModeType.PESSIMISTIC_WRITE);
            emp.setVacationDays(emp.getVacationDays() + accruedDays);
    }
  }}
```

letto un certo impiegato, si verifica il suo stato e nel caso si modificano i suoi giorni di ferie

è corretto acquisire il lock solo quando necessario?

Esempio

- Sembra ragionevole acquisire il lock solo quando serve per minimizzare la quantità di tempo in cui i lock solo attivi sulle tabelle e sull'oggetto. Questo però è sbagliato
- Nell'esempio, dal momento in cui leggiamo l'impiegato a quando lo modifichiamo, un'altra transazione potrebbe aver modificato l'impiegato rendendolo STALE (obsoleto)
- Se esiste un campo di versione sull'entity, si ottiene un OptimisticLockingException anche in questo caso.
  - Ma si utilizza un lock pessimistico proprio per evitare di gestire problemi di oggetti STALE!
- La soluzione è acquisire il lock subito dopo la find!

Recap locking

- Lock ottimistico
  - Nessun lock sulle tabelle e livello di isolamento basso
  - Si sollevano exception di tipo OptimisticLockingException
  - LockModeType.OPTIMISTIC
    - Oggetto leggibile ma non modificabile da altre transazioni (se modifica l'oggetto non riesce a committare)
  - LockModeType.OPTIMISTIC_FORCE_INCREMENT
    - Oggetto non leggibile o modificabile da altre transazioni (che li legge e li modifica nel persistence context ma non riesce a committare)

Recap locking

- Lock pessimistico
  - Lock esclusivo sulla riga e sulle righe collegate via chiave esterna (owner della associazione)
    - Con la proprietà EXTENDED il lock viene applicato anche alle righe collegate senza una chiave esterna.
  - Lock sulle tabelle sempre e immediatamente
  - L'oggetto su cui si applica il lock può essere letto e modificato dalla transazione corrente

Recap locking

- LockModeType.PESSIMISTIC_READ
  - Supporto anche per gli oggetti non versionati
  - Da usare nelle query che leggono dati. Semantica repeteable read
  - Oggetto lockato leggibile da altre transazioni
- LockModeType.PESSIMISTIC_WRITE
  - Supporto anche per gli oggetti non versionati
  - Oggetto lockato non leggibile da altre transazioni (isolamento serialized)
  - Oggetto lockato non modificabile o cancellabile da altre transazioni
- LockModeType.PESSIMISTIC_FORCE_INCREMENT
  - Supporto non garantito per gli oggetti non versionati
  - Versione incrementata subito indipendentemente dalla operazione sul DB effetuata
  - Oggetto lockato non modificabile o cancellabile da altre transazioni
  - Oggetto lockato non leggibile da altre transazioni
  - Versioning attivo per prevenire sovrascritture

controllo di versione SEMPRE per gli oggetti versionati

Recap locking

- Se si acquisisce un pessimistic lock READ e poi si modifia l'oggetto, il persistence provider converte il lock da pessimistic READ a pessimistic WRITE

Exception

- Con lock pessimistico, il provider chiede al DB di acquisire immediatamente un lock. se il DB non si riesce ad acquisire il lock si può sollevare una exception del tipo:
  - PessimisticLockException (rollback transazione)
  - LockTimeoutException (annullamento istruzione corrente ma **NON** rollback dellta transazione)

- Exception che provocano l'annullamento della istruzione in esecuzione (NON DELLA TRANSAZIONE)
  - QueryTimeoutException
  - LockTimeoutException

Exception

- Exception che provocano una rollback
  - EntityExistsException
  - EntityNotFoundException
  - OptimisticLockException
  - PessimisticLockException.

- Exception che NON provocano una rollback
  - NoResultException
  - NonUniqueResultException
  - LockTimeoutException
  - QueryTimeoutException