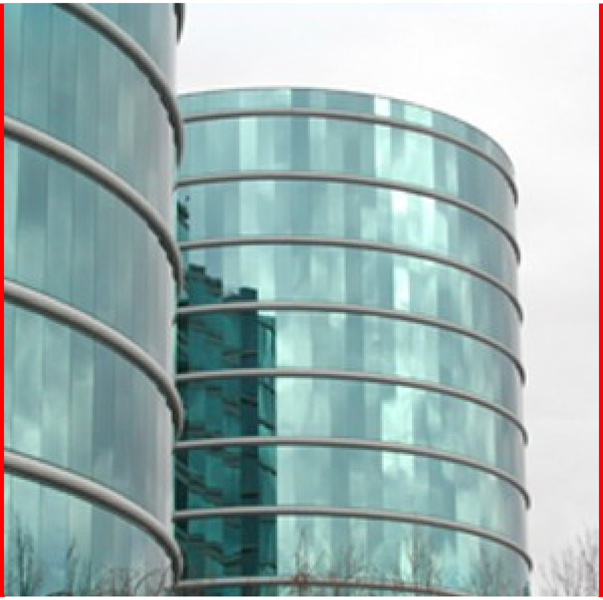


JPA Best Practices

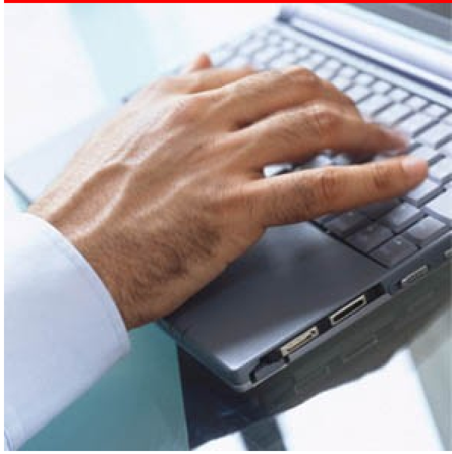
Lee Chuk Munn
chuk-munn.lee@oracle.com

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.

The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda

- Entities
- EntityManager
- Persistence Context
- Queries
- Transactions



Very Brief Overview of JPA

- Introduced as part of JavaEE 5
- POJO based persistence
  - No interface, convention over configuration, annotation based
- Support rich domain modelling
  - Inheritance and polymorphism
- Query language
- Standardize object/relationship mapping
- Usable in JavaEE and/or JavaSE
  - Unified persistence model across the Java platform

Entities

About Entities

- Are not `EntityBeans` !!!
  - Not threadsafe – not a problem if in JavaEE container
- Are POJOs
  - No remote calls involved, methods are executed locally
- Have states
  - New, managed, detached, removed
- Entities are detached (value objects) outside of transaction context
  - Must merge to update data

Example of an Entity

Annotated as “Entity”

Data are accessed as fields

Maps to “customer” table

```
@Entity(access=FIELD)
@Table(name = “customer”)
public class Customer {
   @Id public int id;
   ...
   public String name;
   @Column(name=”CREDIT”) public int c_rating;

   @LOB public Image photo;
   ...
}
```

@Id denotes primary key

Specify the table
column to map to

Primary Keys

- Annotated with @Id
- Simple use case @Id can be generated
  - TABLE – most portable
  - SEQUENCE, IDENTITY
    - Use database's sequence and/or identity column
    - May not be portable
  - AUTO – let persistence manager pick the best strategy

```
@TableGenerator(name=”mygen”, table=”ID_TABLE”
     , pkColumnName=”GEN_KEY”, pkColumnValue=”EMP_ID”
     , valueColumnName=”GEN_VALUE”)
@Id @GenerateValue(strategy=TABLE, generator=”mygen”)
long id;
```

ID_TABLE

GEN_KEY	GEN_VALUE
EMP_ID	Last generated value

Example – Domain Model

```
@Entity public class Employee {
   @Id private int id;
   private String firstName;
   private String lastName;
   @ManyToOne(fetch=LAZY)
   private Department dept;
        ...
}
@Entity public class Department {
   @Id private int id;
   private String name;
   @OneToMany(mappedBy="dept", fetch=LAZY)
   private Collection<Employee> emps = new ...;
           ...
}
```

Example – Managing Relationship INCORRECT

```
public int addNewEmployee(...) {
   Employee e = new Employee(...);
   Department d = new Department(1, ...);

   e.setDepartment(d);
   //Reverse relationship is not set
   em.persist(e);
   em.persist(d);

   return d.getEmployees().size();
}
```

Example – Managing Relationship CORRECT

```
public int addNewEmployee(...) {
   Employee e = new Employee(...);
   Department d = new Department(1, ...);

   e.setDepartment(d);
   d.getEmployees().add(e);
   em.persist(e);
   em.persist(d);

   return d.getEmployees().size();
}
```

Navigating Relationships

- Data fetching strategy
  - EAGER – immediate
  - LAZY – load only when needed
- Lazy is good for large objects with deep relationship hierarchies
- Eager is automatic when operation is performed outside of a transaction
  - Entities are detached immediately
- Cascade specifies operations on relations
  - ALL, PERSIST, MERGE, REMOVE, REFRESH
  - Default is to do nothing
- Avoid MERGE with deep hierarchies
  - Or limit the scope of merge

Choosing Between EAGER and LAZY

- EAGER – too many joins

```
SELECT d.id, ..., e.id, ...
  FROM Department d left join fetch Employee e
      on e.deptid = d.id
```

- LAZY – N + 1

```
SELECT d.id, ... FROM Department d // 1 time
SELECT e.id, ... FROM Employee e
     WHERE e.deptId = ? // N times
```

Lazy Loading

- Lazy load fields and relationships that are not used frequently
- One-many/many-may relationships are lazy loaded by default
- Lazy load CLOB/BLOB if possible

LAZY Loading and Value Objects

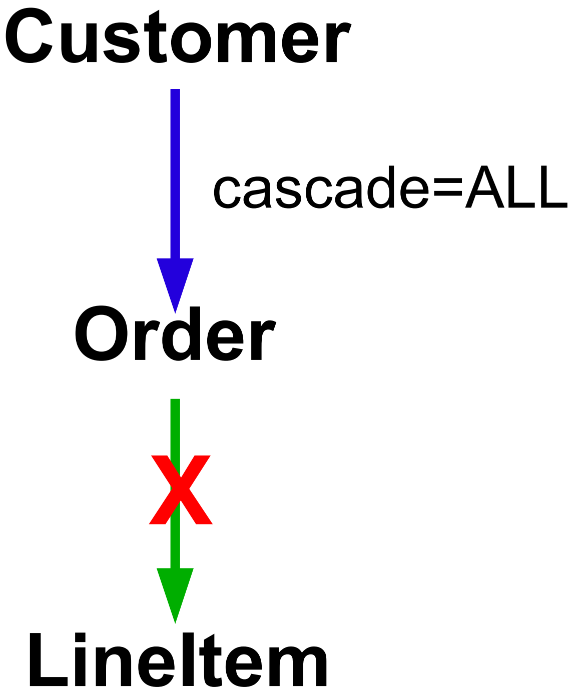
- Accessing a LAZY relationship from a detached entity
  - May get a null
  - May get a previously cached value
  - May get an exception
- Use JOIN FETCH for such objects
  - Specifying which field to pre-fetch – fetcy is like EAGER
  - Returns only Employees that matches WHERE

```
SELECT d FROM Department d
  JOIN FETCH d.employees WHERE ...
```

- Access the collection before entity is detached
    - Like a sync

```
//Forces all employees to be loaded
d.getEmployees().size();
```

Using Cascade



Customer

cascade=ALL

Order

X

LineItem

```
public class Customer {
   @OneToMany(cascase=ALL,
        mappedby="customer")
   Set<Order> orders;


public class Order {
   @ManyToOne
   Customer customer;
   @OneToMany(mappedBy="order")
   List<LineItem> lineItems;


public class LineItem {
   @OneToMany
   Order order
```

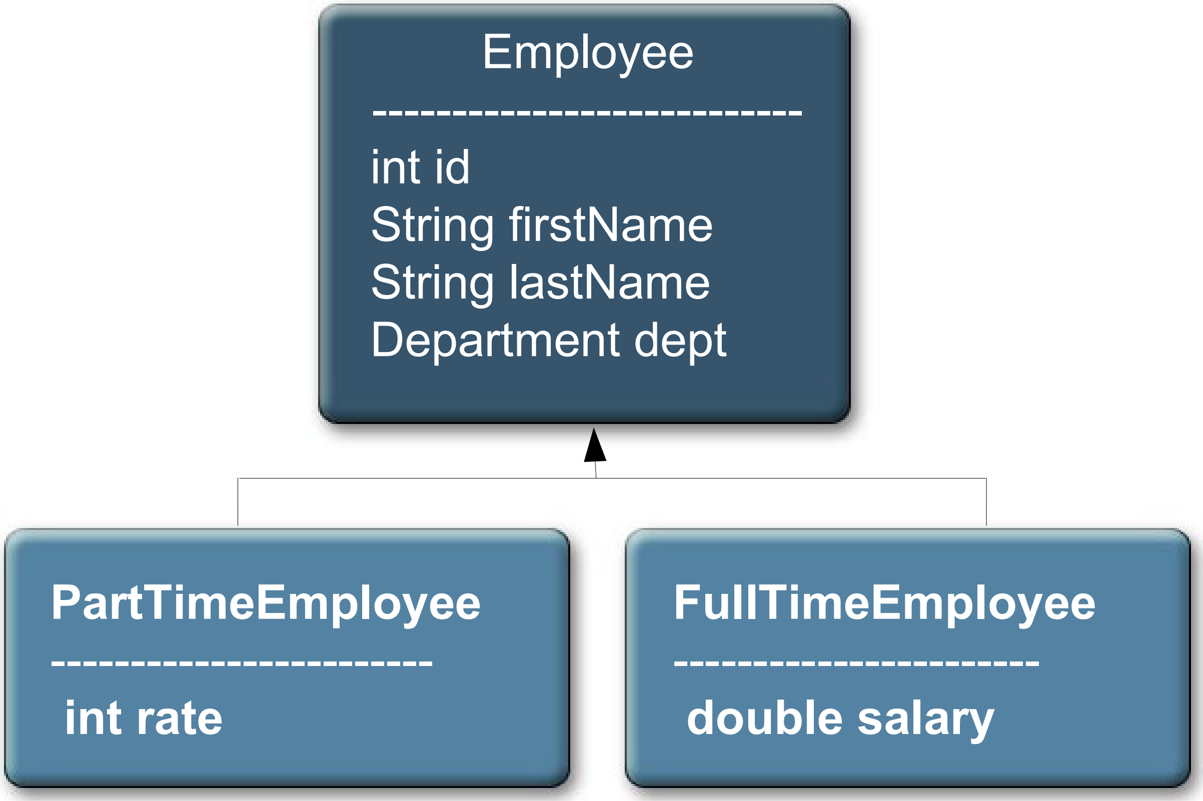
Cascade in Model or Schema

- Much faster as foreign key constraint but less apparent to developer

In Oracle PL/SQL

```
create table employee (
  ...
  constraint fk_dept_id
      foreign key (dept_id)
      references department(dept_id)
      on delete cascade
  ...
}
```

Mapping Inheritance



Employee

----

int id

String firstName

String lastName

Department dept

PartTimeEmployee

----

int rate

FullTimeEmployee

----

double salary

Single Table Per Class

- Benefits
  - Simple
  - No joins
- Drawbacks
  - Not normalized
  - Requires a discriminator field for subclass
  - Table may have too many columns

```
              EMPLOYEE
     ---------------------------
ID                   Int PK,
FIRSTNAME            varchar(255),
LASTNAME             varchar(255),
DEPT_ID              int FK,
RATE                 int NULL,
SALARY               double NULL,
DISCRIM              varchar(30)
```

```
@Inheritance(strategy=SINGLE_TABLE)
```

Joined Subclass

- Benefits
  - Normalized database
  - Database view same as domain model
  - Easy to evolve domain model
- Drawbacks
  - Poor performance in deep hierarchies
  - Poor performance for polymorphic queries and relationships

EMPLOYEE

ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
DISCRIM	varchar(30)

PARTTIMEEMPLOYEE

ID	int PK FK,
RATE	int NULL

FULLTIMEEMPLOYEE

ID	int PK FK,
SALARY	
double NULL	

```
@Inheritance(strategy=JOINED)
```

Table Per Class

- Benefits
  - No need for joins if only leaf class are entities
- Drawbacks
  - Not normalized
  - Poor performance when querying non-leaf entities-union
  - Poor support for polymorphic relationships
- This is not mandatory in the specs

EMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK

PARTTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
RATE	int NULL

FULLTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
SALARY	double NULL

```
@Inheritance(strategy=TABLE_PER_CLASS)
```

Entity Manager

Container vs Application

- Container managed entity manager
  - Injected into application
  - Automatically closed
  - JTA transaction – propagated
- Application managed entity managers
  - Used outside of the JavaEE 5 platform
  - Need to be explicitly created
    - `Persistence.createEntityManagerFactory()`
  - RESOURCE_LOCAL transactions
    - Not propagated
  - Need to explicitly close

Threading Model and Injections

- JPA components
  - EntityManager is not threadsafe
  - EntityManagerFactory is threadsafe
- Field injection is only supported for instance variable
  - Not threadsafe
- Dangerous to inject non threadsafe objects into stateless components
  - Inconsistent data
  - Data viewable by other threads

Injecting EntityManagers

```
public class ShoppingCartServlet extends HttpServlet {
  @PersistenceContext EntityManager em;
  protected void doPost(HttpServlet req, ...) {
    Order order order = ...;
    em.persist(order);
  }
```

WRONG

```
public class ShoppingCartServlet extends HttpServlet {
  @PersistenceUnit EntityManagerFactory factory;
  protected void doPost(HttpServlet req, ...) {
    EntityManager em = factory.createEntityManager();
    Order order order = ...;
    em.persist(order);
  }
```

CORRECT

Persistence Context

Persistence Context

- Acts like a cache for entities
- Two types of persistence context
- Transaction scoped
  - Used in stateless components
  - Typically begins/ends at method entry/exit points
- Extended scoped persistence context
  - Used with business transactions spans multiple request
  - Ideal place is to create extended PC at the beginning of business process or session
  - Supported in
    - `StatefulSessionBean`
    - Application managed `EntityManager`

Persistence Context and Caching

```
String empId = “12345”;
. . .
```

Meanwhile empId 12345 have been changed **in another thread**

```
//Query the data
Query query = em.createQuery(“SELECT e FROM Employee e “
     + “WHERE e.ID = :ID”).setParameter(“ID”, empId);
employee = (Employee)query.getSingleResult();
```

Will I get the new data for employee?

Persistence Context as Cache

- It depends
- Entities managed by persistence context
  - Are not refreshed from database until `EntityManager.refresh()` is invoked
  - Are not synchronized with database until `EntityManager.flush()` is explicitly invoked or implicitly when PC closes
- Entities remains managed by PC until
  - `EntityManager.clear()` is invoked
  - Transaction commits

Flush Mode

- Controls whether the state of managed entities are synchronized before a query
- Types of flush mode
  - AUTO – immediate, default
  - COMMIT – flush only when a transaction commits
  - NEVER – need to invoke `EntityManager.flush()` to flush
- Querying data you know that has not change or don't care if result includes changes, set flush to COMMIT

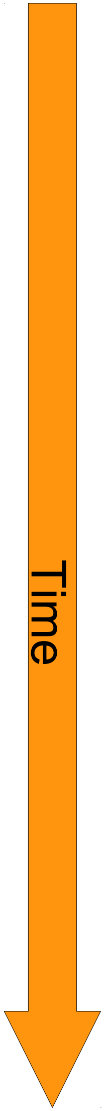
```
Query q = em.createNamedQuery(“findAllOrders”);
q.setParameter(“id”, orderNumber);
q.setFlushMode(FlushModeType.AUTO);
//Ensure that the query gets the latest results
List list = q.getResultList();
```

Stale Data and Parallel Updates

- JPA simplifies persistence but does not guard against parallelism
- Introduce `@Version` for optimistic locking
  - Can be `int`, `Integer`, `short`, `Short`, `long`, `Long`, `Timestamp`
  - Not used by application
  - Updated when transaction commits, merged or acquiring a write lock

```
public class Employee {
  @ID int id;
  @Version Timestamp timestamp;
  ...
```

Preventing Parallel Updates – 1

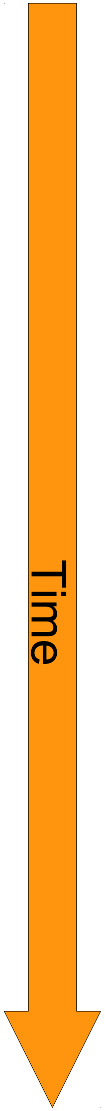


```
tx1.begin();                  tx2.begin();
//Joe's employee id is 5      //Joe's employee id is 5
//e1.version == 1             //e1.version == 1
e1 = findPartTimeEmp(5);      e1 = findPartTimeEmp(5);
                              //Series of expensive
//Current rate is $9          //to follow
e1.raiseByTwoDollar();
//Current rate is $11



tx1.commit();
//e1.version == 2 in db
                              //e1.version == 1 in db?
                              tx2.commit();
                              ~~//Joe's rate will be $14~~
                              //OptimisticLockException
```

Preventing Parallel Updates – 2



```
tx1.begin();
//Joe's employee id is 5
//e1.version == 1
e1 = findPartTimeEmp(5);

//Current rate is $9
e1.raiseByTwoDollar();
//Current rate is $11



tx1.commit();
//e1.version == 2 in db
```

```
tx2.begin();
//Joe's employee id is 5
//e1.version == 1
e1 = findPartTimeEmp(5);
em.lock(d1, WRITE);
//version++ for d1
em.flush();
//Series of expensive
//to follow


//e1.version == 1 in db?
tx2.commit();
~~//Joe's rate will be $14~~
//OptimisticLockException
```

Lock Modes

- Five lock modes
  - OPTIMISTIC – provides repeatable read isolation
  - OPTIMISTIC_FORCE_INCREMENT – repeatable read but updates version field
  - PESSIMISTIC_READ – pessimistic repeatable read
  - PESSIMISTIC_WRITE – serialized access
  - PESSIMISTIC_FORCE_INCREMENT – pessimistic but also updates version field, optional
- OPTIMISTIC and OPTIMISTIC_FORCE_INCREMENT are the new names for READ and WRITE respectively

Bulk Updates

- Update directly against the database
  - By passes `EntityManager`
  - `@Version` will not be updated
  - Entities in persistence context may be outdated
- Avoid updating individual entities
  - Use bulk updates

SLOW

```
//Terminate all contract employees
List<Employee> empList = query.getResultList();
for (Employee e: empList)
   e.status(“ContractEnd”);
```

Generate lots of SQL

FAST

```
//Terminate all contract employees
TypedQuery<Employee> query = em.createQuery(
     “UPDATE Employee e SET i.status = 'ContractEnd'
     + “WHERE ...”);
query.executeQuery();
```

Queries

Queries

- Prefix query names with class being returned (JPA 1)

```
@NamedQuery(name="Employee.findByName", ...)
```

- Dynamic query
  - Beware of SQL injection, better to use with named parameters
  - Use named query instead of dynamic query where possible – enforce parametrized query

```
q = em.createQuery("select e from Employee e WHERE "
     + "e.empId LIKE '" + id + "'");
```

NOT GOOD

```
q = em.createQuery("select e from Employee e WHERE "
     + "e.empId LIKE ':id'");
q.setParameter("id", id);
```

GOOD

Typed Queries

- Specify the type that the query will return
  - Works with named, native and dynamic queries
- Alternatively, use criteria – same effect

```
TypedQuery<Employee> q = em.createQuery(
    “select e from Employee e WHERE ”
    + “e.empId LIKE ':id'”, Employee.class);
q.setParameter(“id”, id);

List<Employee> list = q.getResultList();
```

Polymorphic Queries

- May return too many results
  - Eg. Employee → PartTime, FullTime, Intern – return 2 of 3
- Use type expression to restrict query polymorphism

```
select e from Employee e
    where type(e) in (PartTime, Intern)
```

Criteria API

- Currently JPQLs are string based
  - Easier to use but not cannot perform compile time checking on query and entity attribute name typos
- Dynamically creates query without out string manipulation
  - Parity with string based query
- Strongly type, compiler validation during development
- Optionally can generate metamodel over entities
  - Provided by ORM tools

JPA Queries

“*SELECT o FROM ORDER o WHERE o.total > 100*”

OK

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Order> o = cb.createQuery(Order.class);
Root<Order> ord = o.from(Order.class);
Predicate cond = cb.gt(ord.get(“total”), 100);
o.select(ord).where(cond);
TypeQuery<Order.class> q = en.createQuery(o);
List<Person> result = q.getResultList();
```

BETTER

```
CriteriaQuery<Order> o = cb.createQuery(Order.class);
Root<Order> ord = o.from(Order.class);
o.select(ord).where(cb.gt(ord.get(Order_.total), 100));
TypeQuery<Order.class> q = en.createQuery(o);
List<Person> result = q.getResultList();
```

BEST

Generated metamodel

Transactions

Transactions

- Do not perform expensive and unnecessary operations that are not part of a transaction
  - Hurt performance, eg. logging
- Keep the code in the transaction to a minimum and close it when not needed
- Eliminate transaction for “read-only” data
  - Eg. Department names

```
@Stateless public ... {
  @TransactionAttribute(NOT_SUPPORTED)
  public List<Deparment> getAllDepartments() {
    return (em.createQuery(
         “SELECT e FROM Department e”)
         .getResultList());
  }
```

Transaction Type

- Container managed EntityManager can be JTA or `RESOURCE_LOCAL`
  - `RESOURCE_LOCAL` is non JTA
- `RESOURCE_LOCAL` `EntityManager` are created from `EntityManagerFactory`

JTA From Non JTA `EntityManager`

- Create `EntityManager` inside a JTA transaction
  - Get an injected instance of JTA from container or client container (for JavaSE)

```
@Resource UserTransaction utx;
. . .
utx.begin();
EntityManager em = emf.createEntityManager();
//em is now JTA
```

- Join a JTA transaction

```
@Resource UserTransaction utx;
. . .
EntityManager em = emf.createEntityManager();
//em is is RESOURCE_LOCAL
utx.begin();
em.joinTransaction();
```

SOFTWARE. HARDWARE. COMPLETE.

We encourage you to use the newly minted corporate tagline "Software. Hardware. Complete." at the end of all your presentations. This message should replace any reference to our previous corporate tagline "Oracle Is the Information Company."

ORACLE®

For More Information

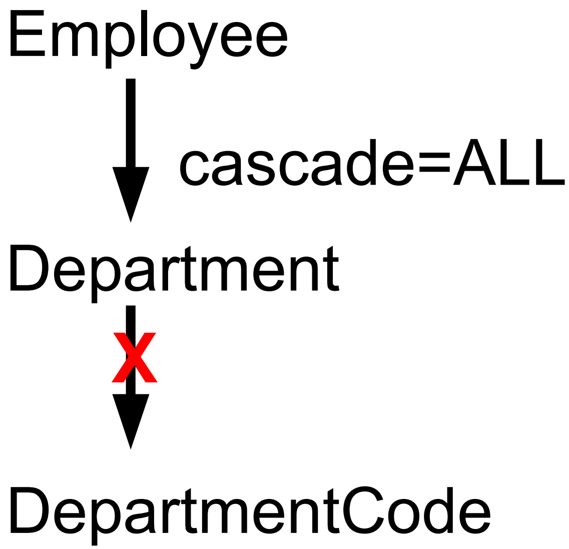
search.oracle.com

or

oracle.com

Using Cascade

```
@Entity public class Employee {
  @Id private int id;
  private String firstName;
  private String lastName;
  @ManyToOne(cascade=MERGE, fetch=LAZY)
  private Department dept;
      ...
}
@Entity public class Department {
  @Id private int id;
  private String name;
  @OneToMany(mappedBy = "dept"
            cascade=MERGE, fetch=LAZY)
  private Collection<Employee> emps = new ...;
  @OneToMany
  private Collection<DepartmentCode> codes;
        ...
 }
```



Employee

cascade=ALL

Department

X

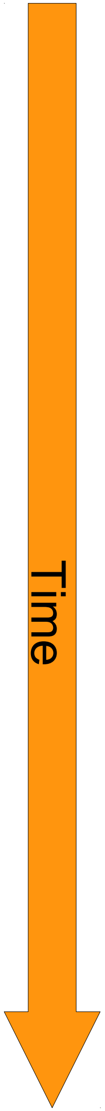
DepartmentCode

Transient Fields

- Used on fields that are not persisted
  - Eg. computed fields, temporary values, cached values

```
@Entity public class Employee {
   @Id private int id;
   private String firstName;
   private String lastName;
   @ManyToOne(fetch=LAZY)
   private Department dept;
   @Transient float yearEndBonus = 0f;
       ...
}
```

Preventing Stale Data

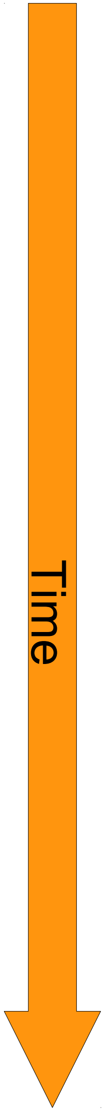


```
tx1.begin();                        tx2.begin();
d1 = findDepartment(dId);
                                    e1 = findEmp(eId);
//d1's original name is             d1 = e1.getDepartment();
//”Engrg”                           em.lock(d1, READ);
d1.setName(“MarketEngrg”);          if(d1's name is “Engrg”)
                                       e1.raiseByTenPercent();
tx1.commit();




                                    //Check d1.version in db
                                    tx2.commit();
                                    ~~//e1 gets the raise he does~~
                                    ~~//not deserve~~
                                    //Transaction rolls back
```

Pessimistic Locks on Update



```
tx1.begin();
e1 = findDepartment(dId);
em.lock(e1, PESSIMISTIC_WRITE);




//d1's original name is
//”Engrg”
d1.setName(“MarketEngrg”);

tx1.commit();
```

```
tx2.begin();
props.put(“javax.persistance
.lock.timeout”, 5000);
e1 = findEmp(eId);
//Continue or timeout
em.lock(e1
, PESSIMISTIC_WRITE, props);
```