# Chapter 1: Introduction

When you start to consider abstract superclasses or parent classes that are not persistent, inheritance rapidly becomes a complex issue in object-relational mapping. Not only is there a challenge with storage of the class data but the complex table relationships are also difficult to query efficiently

* JDO - java data object
* POJO - plain old java object

The Java Persistence API is a lightweight, POJO-based framework for Java persistence

New features in JPA 2.1

1) support for stored procedures

2) mapping converters

Queries use schema abstraction that is based on the entity model

# Chapter 2:

Objects that leave the persistence layer are called detached. Dattachement model provides a way of reconciling the state of entity being rettached with the state that it was in before it became detached

Characteristics of object that has been turned into an entity

1. Persistability
   1. Their state can be represented in a data store and can be accessed at a later time
   2. Persistent objects become persistent in a moment they are instantiated in a memory. If a persistent object exists, then by definition it is already persistent
2. Identity
   1. (it has object and database identity) Persistent identity is a key that uniquely identifies entity. An entity has a peristance identity when there exists a representation of it in the data store. (row in table)
3. Transactionality
   1. transactions are needed for the changes to be committed in the database. In-memory entities are simple Java objects that obey all of the rules and constraints that are applied by the JVM to other objects.
4. Granularity
   1. Entities are business domain objects that have specific meaning to the application that access them.

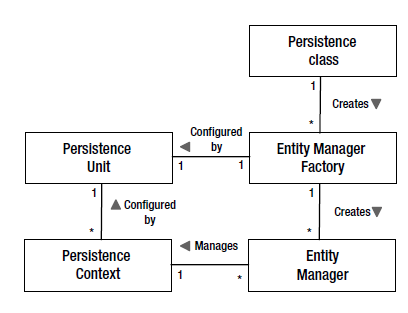
**Entity** - Java Bean with default constructor. Class turns to Entity by annotating with @Entity and adding @Id annotation for unique key (@Id annotation can be placed also on getter method)

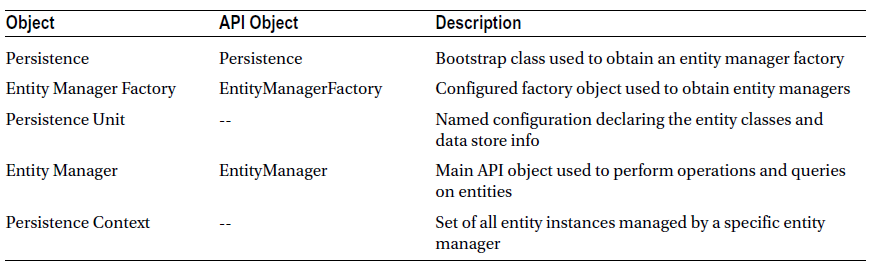
**EntityManager**

* interface, Until an Em is not used the entity is no more than a regular (nonpersistent) Java object.
* When EM obtains a reference to an entity, either by having it explicitly passed in as an argument to a method call or because it was read from Database - that object is managed by EM
* The set of managed entity instances within EM at any given time is called PERSISTANCE CONTEXT (only one java instance with the same persistent identity may exist in a persistence context at any time.)
* EM is implemented by a particular PERSISTANCE PROVIDER - supplies implementation for the entire JPA. (from EM to Query classes a SQL generation)
* Is not thread safe

Entity mangers come from EntityManagerFactory [EMF]=> is thread safe (Configuration for EM is template from the EMF that created it, it is defined separately as persistence unit)

**Persistence** **unit**

* dictates settings and entity classes used by all EM obtained from the unique EMF instance bound to that persistence unit. (EMF one-to-one correspondence to Persistence unit)
* allows differentiation of one EMF from another



Creation of EMF and EM in Java SE

EntityManagerFactory emf **=**

Persistence**.**createEntityManagerFactory**(**"EmployeeService"**);**

EntityManager em **=** emf**.**createEntityManager**();**

Persisting – transient entity or one that doesn’t yet have any representation in DB storing it into DB.

When EM is not properly instantiated throw PersistenceException.

If EM doesn’t find item in DB return null.

Correct delete statement –

Employee emp **=** em**.**find**(**Employee**.**class**,** id**);**

**if** **(**emp **!=** **null)** **{**

em**.**remove**(**emp**);**

**}**

When running this code in Java EE container is used JTA. The transaction model when running inside the container assumes that a transactional context is present when is required. It transaction context is not available than will be thrown Exception or changes we be never persisted

In Java SE environment exists EntityTransaction which serves to begin and commit transactions. Entity transaction is retrieved from EM.getTransaction() then {begin, em.persit, em.remove… commit}

JPQL (Java Persistence Query Language)

* is impelemented in code as a *Query* or *TypedQuery<X>* object.
* Query can be defined statically or dynamically
* To execute query it is needed to call i.e. (getResultList())

Transaction-type (inside persistence.xml) RESOURCE\_LOCAL – indicates that persistence unit uses resource-level EntityTransaction instead of JTA transactions.

<class> is needed for Java SE environment but for Java EE environment it is no needed. (Because container scan for entity classes annotated with @Entity)

Persistence artifacts are packaged in persistence archive (jar file which contains persistence.xml file in META-INF dir)

When we want to configure JPA in our project. It is important to differentiate between that a persistence-unit is managed by container or RESOURCE\_LOCAL.

When we are using container then it is enough to set up

<persistence-unit name="default" transaction-type="JTA">

But in case when we are not using container i. e. by tests. We have to set up

<persistence-unit name="default" transaction-type="RESOURCE\_LOCAL">

And also properties for connection.

<property name="hibernate.connection.username" value="sa"></property>  
<property name="hibernate.connection.password" value="sa"></property>  
<property name="hibernate.connection.driver\_class" value="org.hsqldb.jdbcDriver"></property>  
<property name="hibernate.connection.url" value="jdbc:hsqldb:mem:test"></property>

# Chapter 3: Enterprise Applications

## Session beans

* Component technology designed to encapsulate business services
* Bean class is a bit different from regular java Class because it is part of EJB component model
* Bean has access to wide array of container services
* The logical interface of the session bean consists of its public methods
* To allow both the server and the bean to achieve their initialization requirements, EJBs support lifecycle callback methods, that are invoked by server at various points in the bean’s lifecycle
* Communicate with client via proxy
* All beans can use interfaces (local, Remote)

Three types of session beans (Stateless, Stateful, Singletion)

**Stateless**

* no state is carries over from one business operation to the other.
* Callback methods –
  + PostConstruct – invoked as soon as are initialize all container services
  + PreDestory – invoked before the server releases the bean instances to be GC

**Stateful** – state is maintained across calls

* @Remove – methods that the client will use to end the conversation with the bean [Every stateful bean have to implement at least 1 @Remove method]
* Callback methods –
  + PostConstruct
  + PreDestroy
  + PrePassivate – before bean passivation is called [prepare bean for serialization]
  + PostActivate – after bean has been activated

Passivation is process by which server serializes the bean instances so that it can be stored offline

Activation is the process of deserializing a passivated session bean instance and making it active in the server.

**Singleton –**

* hybrid between stateless and stateful beans. All clients share the same singletion bean instance, so it becomes possible to share state across method invocations, but this bean lack conversational contract and mobility of stateful bean. (There are issues with concurrency).
* Bean instance will be never re-created in the event of the system exception
* Ideal for storing common application state (think about concurrency in different ways)
* By default all access to the bean are synchronized
* The lifecycle of stateless session bean is tied to the lifecycle of the overall application.
* Callbacks
  + PreDestroy – is invoked when application is shutting down (called only once)
  + PostConstruct

## Servlets

* HTTP session is a map of data associated with a session id.
* Cookies are used to link the session id with the client.
* Session beans, for example are easily accessible from anywhere within a Java EE application, making them perfect neutral.

## Dependency Management and CDI

Core Java EE has problems with injecting dependencies into a limited number of predefined server resources

References to resources

* reference is name link to resource that can be resolved dynamically at runtime (within app code or resolved automatically by the container)
* reference consists of two parts: Name and Target
  + Name – used by app code to resolve reference dynamically
  + Target – used by server to find resource that the app is looking for
* Reference is declared one of the following annotations (these annot. Can be placed on class[name attribute is mandatory], field, setter)
  + @Resource
  + @PersistentContext
  + @PersistentUnit
  + @EJB

1. Resource Injection can inject JNDI Resources directly whereas Dependency Injection cannot.
2. Dependency Injection can inject Regular Classes (managed bean) directly whereas Resource Injection cannot.
3. Resource Injection resolves by resource name whereas Dependency Injection resolves by type.
4. Dependency Injection is typesafe whereas Resource Injection is not.

**Dependency lookup**

Traditional form of dependency management in Java EE. App code is responsible for looking up named reference using JNDI. Each Java EE component has its own locally scoped JNDI naming context => environment naming context. “java:comp/env/” that was added to the reference name indicates to the server that the environment naming context should be searched to find the reference

@EJB**(**name**=**"deptAudit"**,** beanInterface**=**AuditService**.**class**)**

public class DeptService **{**

**try** **{**

Context ctx **=** **new** InitialContext**();**

audit **=** **(**AuditService**)** ctx**.**lookup**(**"java:comp/env/deptAudit"**);**

**}** **catch** **(**NamingException e**)** **{**

**throw** **new** EJBException**(**e**);**

**}**

EJB supports alternative to JNDI lookup – lookup method from EJBContext interface (SessionContext sub-interface)

SessionContext context**;**

public void setSessionContext**(**SessionContext context**)** **{**

**this.**context **=** context**;**

**}**

**(**AuditService**)** context**.**lookup**(**"deptAudit"**);**

EJB lookup has two advantages

* Name is exactly the same as it was specified in the resource reference
* Only RuntimeExceptions are thrown

**Dependency Injection**

The process of automatically looking up a resource and setting it into the class is called dependency injection. DI simplified testing.

1. Field Injection – server looks up the dependency in the environment naming context, it assigns the result directly into the annotated field of class. If I want to unit test there is needed add setter or make the field accessible to the unit tests. Environment naming context should be package.Class/fieldname
2. Setter Injection – Works well with unit testing

**Declaring dependencies**

**PersistanceContext**

@PersistenceContext**(**unitName**=**"EmployeeService"**)**

EntityManager em**;**

Is a managed set of entity instances. If the unitName is omitted , it is vendor-specific how the unit name for the persistence context is determined (required or not required). If EM is injected into stateless bean (EM is state field). The EM value is a container-managed proxy instance that acquires and releases persistence context on behalf of the app code. (In Spring it works in similar way but it needs additional configuration)

**PersistenceUnit**

@PersistenceUnit**(**unitName**=**"EmployeeService"**)**

private EntityManagerFactory emf**;**

A persistence unit is a named configuration of entity classes. It is vendor specific how unitName is determined. There is diff. between that how is EM is acquired.

**Referencing server resources**

@Resource

SessionContext context**;**

For resources that they don’t have dedicated annotations. resourceType specify type of resource if the server can’t it figured out automatically.

**CDI**

CDI bean is any class that qualifies for the CDI injection services. We can also use Constructor injection [good for testing]

**Scopes and Contexts**

5 pre-defined scopes exists, three of which(session, request, application) are defined by the servlet spec., by CDI was added conversation scope (spans a series of sequential JSF requests), by JTA was added Transaction scope (maps lifetime of the active JTA transaction).

Bean type is associated with the scope by annotation. Managed instances of that bean will have a lifecycle similar to the declared scope.

The context is a place where the scoped instances reside for the duration of the scope. There can be only one instance of each bean type in each context. Also exists dependent scope => it is really absence of scope. If no scope is specified, then the dependent scope is assumed. The @Dependent annotation can be used to explicitly mark a bean as being dependent.

beans.xml was required in CDI 1.0. In CDI 1.1 beans may be instead annotated.

**Qualified Injection**

Used to distinguish a bean type from other bean types that have the same inherited or implemented interface type. Helps container to decides which bean can be injected.

**Producer**

Producer method is a method that the CDI container will invoke to obtain a new bean instance. Producer method can decide in runtime which subclass could be returned. Producer fields work the same way as producer methods except that the container accesses the field to the instance instead of invoking a method.

A producer method acts as a source of objects to be injected, where:

* the objects to be injected are not required to be instances of beans, or
* the concrete type of the objects to be injected may vary at runtime, or
* the objects require some custom initialization that is not performed by the bean constructor

## Transactions

Basic transaction properties (ACID):

1. Atomicity – The success of every individual operation is tied to the success of the entire group.
2. Consistency – At the end of transaction data are legal and valid with the respect to the rest of the data in the system.
3. Isolation – Changes within transaction are visible only to the transaction that is making changes. After commit all transactions see changes.
4. Durability – changes after transaction endure beyond of the completion the transaction

Transactions live in enterprise application at different levels.

1. The basic and on the bottom level are transaction of the resource (relational db transactions) fronted be DataSource interface. => Resource local transaction. These types of transactions are manipulated directly by JDBC DataSource that is obtained from the app. Server. These transactions are demarcated by app (BMT- bean managed transaction).
2. Container transactions (uses JTA) – can enlist number of resources, including data sources as well as other types of transactional resources. Containers add custom level on top of the JDBC DataSource to perform functions such as connection management and pooling…. They are also called global transactions. These transactions can be demarcated by app. or by default they are demarcated by container(CMT – container managed transaction).

EJB can use both BMT and CMT . Servlet can use only BMT

Changing transaction management type in EJB from default CMT to BMT

@TransactionManagement**(**TransactionManagementType**.**BEAN**)**

public class ProjectService **{**

### Container –managed Transactions CMT

Transactional attributes:

* MANDATORY – it expected that transaction is already started and this transaction has to be active when method is called. If no transaction is active an exception is thrown.
* REQUIRED – (the most common) expected that method is already in transaction. The container provides a guarantee that transaction is active. If one is active, it is used. If one doesn’t exist the new one is created
* REQUIRES\_NEW – it is useful when we want in order to the method was in own transaction. The method should be committed or rolled back independently of others methods. It can lead to the excessive transaction overhead.
* SUPPORTS – methods are not dependent on a transaction, but will tolerate running inside one if it exists. No transactional resources are access in the method.
* NOT\_SUPPORTED – container suspends active transaction when method with this attribute is called inside transaction
* NEVER – method will cause thrown exception (use when it is expected that transaction has to be completed)

There are actually two different ways of specifying CMT, one for EJB and one for CDI, servlets, JAX-RS [use transactional interceptors]….

**EJB container-managed transactions**

By default REQUIRED attribute will be applied. In case when any bean wants in order to the transaction was rolled back, it can be done by setRollbackOnly() method on EJBContext. This will be not cause immediately rollback the transaction but after transaction is completed. It is same as container thrown Exception in transaction.

**Transactional interceptors**

The main difference to EJB is in using different annotation @Transactional [In Java EE 7] and instead of TransactionalAttributeType is used Trasactional.TxType. The main difference between EJB CMT and transactional interceptors-based components [TIBC] is that, that TIBC do not automatically get CMT but it has to declared by annotation @Transactional [interceptor] (can be used on managed beans or CDI but not by EJB)

### Bean-managed Transactions

Only EJBs are provided CMT by default. Method which begins transaction has to ensure in order to finish this transaction, in other case exception occurs. For EJB beans BMT transactions are not propagated. When other method is called from transaction this transaction is not propagated to this method [transaction is suspended] but after finishing this method transaction goes further, this is not true for none EJB components. Don’t use in EJB.

**UserTransaction**

The component in order to start or end the transaction needs an interface for it. UserTransaction JTA interface. UserTransaction is not current transaction instance, it is sort of proxy that provides the transaction API and represents current transaction. Can be injected by CDI @Inject or by @Resource. In environment naming context is reserved by name java:comp/UserTransaction

Only one transaction can be active at a given time. For BMT doesn’t exist possibility to suspend transaction. At a given time can be active only one transaction but multiple transactions can be associated with single thread. Transaction can be rolled back by different ways:

* setRollbackOnly() – transaction can end only with the rollback
* rollback() – transaction is immediately rolled back
* setTransactionTimeout(int second); - transaction is roll back when time limit is reached. It can not be changed during the transaction. It has to be set before transaction starts

Every transaction has a status – getStatus() If no transaction is active then value returned will be STATUS\_NO\_TRANSACTION [setRolledBack() => status STATUS\_MARKED\_ROLLBACK]

**How to pack it**

Persistence xml

<persistence>

<persistence-unit name="EmployeeService">

<jta-data-source>jdbc/EmployeeDS</jta-data-source>

</persistence-unit>

</persistence>

Persistence xml should be located in META-INF for jar or in the WEB-INF/classes/META-INF for war.

# Chapter 4: Object Relational Mapping

The mapping annotations can be categorized as being in one of two categories:

1. Logical annotations: describe the entity model from object modelling view. Tight to Domain Model
2. Physical annotations: relate to concreate Data model in Database. They deal with tables , columns

Access Mode: the way how is state accessed in the entity

* Field Access: all fields has to be declared as either, package, default or protected access modifier
* Property Access – must exist getter and setter methods. Return type of getter must be same as parameter into setter. Both methods must be either protected or public visibility. The mapping must to be on getter method.
* Mixed access: @Access annotation with specific access mode on the subclass entity will be cause the default access type to be overridden for the entity subclass. Also useful when you need to perform a simple transformation to the data when reading or writing to the db. In Case when we want to mixed property and field Accesstypes. (Don’t forget on transient annotation on field !!!!)

@Entity

@Access**(**AccessType**.**FIELD**)**

public class Employee **{** **...** **}**

@Transient private String phoneNum**;**

@Access**(**AccessType**.**PROPERTY**)** @Column**(**name**=**"PHONE"**)**

protected String getPhoneNumberForDb**()** **{** **...** **}**

## Mapping to a Table

When table name doesn’t match with entity name add @Table annotation @Table(name="EMP")

@Table(name="EMP", schema="HR", catalog="HR")– we can set table name, schema name [@Table(name="HR.EMP")] – none standard

Schema is used to differentiate one set of tables from different set.

Catalogs are named collections of schemas in an SQL-environment. An SQL-environment contains zero or more catalogs. A catalog contains one or more schemas, but always contains a schema named INFORMATION\_SCHEMA that contains the views and domains of the Information Schema.

## Mapping Simple Types

If type of JDBC layer cannot be converted to Java type Exception is thrown but it is not guaranteed. If Database type cannot be converted to JDBC type, It can be on provider or JDBC driver might perform conversation.

@Basic – is optional, can be placed on property or field and explicitly mark it as persistent.

## Column mappings

@Column(name="COMM") – if default name doesn’t correspond to db name

**Fetching type**

In @Basic annotation we can choose fetch type @Basic(fetch = FetchType.Lazy/Eager). Default behavior is load data eagerly. Lazy data are not loaded by default. Lazy attribute can be returned from the query as empty value. Not good idea lazy fetch simple types.

**Large Objects (LOB)**

Character large objects (CLOB) Byte Large Objects(BLOB). When is in database large object Lob than we have to mark column with annotation @Lob.

* Database BLOB is equivalent Java byte[], Byte[] and Serializable types
* Database CLOB is equivalent Java char[], Character[], String

**Enumerated types**

Each enum has an ordinal value which we can don’t change at runtime and this value is specified by order in enum, first value is 0. Enum is mapped with database integer. This solution works by default without any annotations. Problem with adding values in the middle.

Better approach is store the value as a string. We can do this be adding @Enumerated annotation. Default value of @Enumerated(Ordinal). When we want to map string values @Enumerated(EnumType.STRING). Here is problem with renaming enumarion values.

Enum in Java can have also state, but this property is not supported by JPA.

**Temporal types**

Set of times: java.sql.Data, java.sql.Time, java.sql.Timestamp, java.util.Date, java.util.Calendar.

Util types need additional metadata in order to indicate how communicate with jdbc @Temporal => TemporalType.TIME/DATE/TIMESTAMP

**Transient state**

@Transient- attribute marked with this annotation will be not mapped. Also can be used java *transient*

## Mapping primary key

Primary key not nullable or updatable and should be insertable. When we want to automatically generate unique values: @GeneratedValue. This values is visible for persisted entity and when newly entity is store to db and flush action is performed or transaction is completed.

Exists four types of generation strategies: GenerationType.AUTO/TABLE/ SEQUENCE/IDENTITY. Table and Sequence generators can be reused by multiple entities. These generators are named and are accessible to all entities in persistence unit.

1. **AUTO** usually create integer. It is good practice to use Long for id. Provider pick own generation strategy . Provider can chooses from three different types of generation strategies. Some strategies need for generation some resources (Table needs create table….). Therefore there is needed some creation phase or schema generation to initialize these resources.
2. **TABLE** – the most generic and portable. Can be specified on table or attribute. You can store multiple sequences in one table. Multiple identifiers for multiple entities in one table. Id generation table should have two columns
   * First is string – used to identify the particular generator sequence. Primary key for all generators.
   * Second is integral type – represents last generated value using jpa generator strategy. Every sequence represents a separate row in the table

@TableGenerator**(**name **=** "employee\_gen"**,** table **=** "id\_gen"**,** pkColumnName "gen\_name"**,** valueColumnName **=** "gen\_val"**,** allocationSize **=** 100**)**

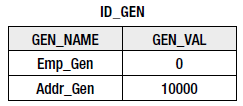
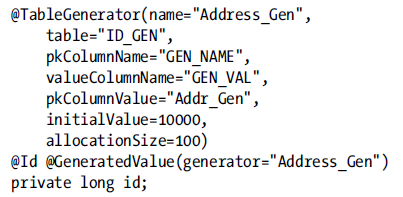
@GeneratedValue**(**strategy **=** GenerationType**.**TABLE**,** generator **=** "employee\_gen"**)**

If schema generation is choosing it will be created; if not the default table assumed by the provide must be known and must exist in the db.

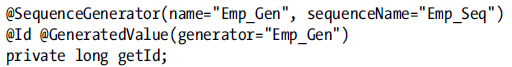
We can also explicitly specify which table will be used for id storage. @TableGenerator(name=”Emp\_gen”)

@GeneratorValue(generator=”Emp\_gen”)

The default value for generated values is 0. This value can be also explicitly set on custom value. This value is used only during schema generation. Identifier value is incremented by the allocation size (By default 50).

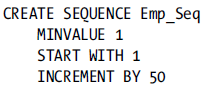


1. **ID Generation using DB Sequence** – If more generator are used and these generators are not named. There is no guarantee whether they use the same default sequence or different ones.



It is good practice to name sequence and we can create sequence in Db with following command.

allocationSize in annotation has to respond to the Increment by

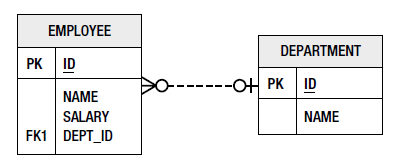


1. **ID Generation using DB Identity –** Is available only when database supports it. For Object-Relational identifier they are less efficient => identifier is available after commit time and they cannot be allocated in the blocks. For others generator there exist possibility to eagerly allocate identifier

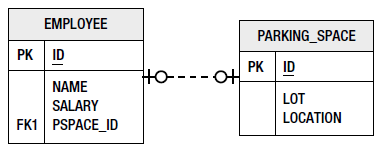
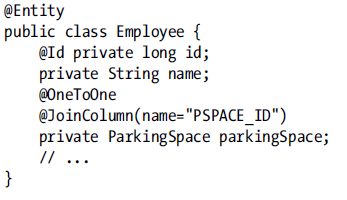
## Relationships

That side which has JoinColum is owning side. MappedBy means that the entity is on inverse side of the relationship. It is good practice that logical mapping should be first followed by physical

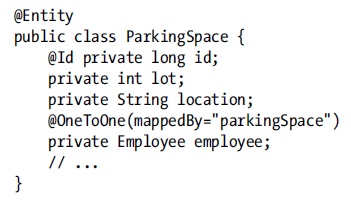
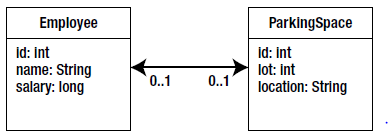
1. **Many-To-One** – is always on the owning side of the relationship. To specify the name of the join column, the name element is used. If no @JoinColumn is used the default column name will be assumed. The name that is used as the default is formed from a combination of both the source and target entities. DEPARTMENT\_ID (name of relationship attribute in source entity[which is not owning] unserscore \_ plus name of the primary key of owning entity)



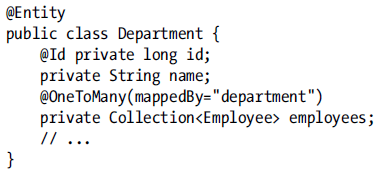
1. **One-To-One –** 
   1. **Unidirectional**

* 1. **Bidirectional –** Either side can be the owner of the relationship => JoinColumn can be on one side or the other. We must add a mappedBy element to indicate which side is owner of the relationship(This should be in attribute that has not JoinColumn annotation

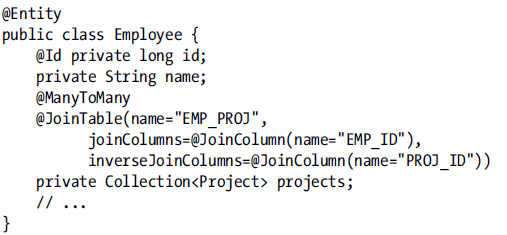
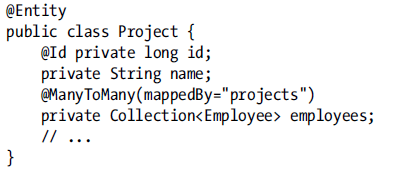


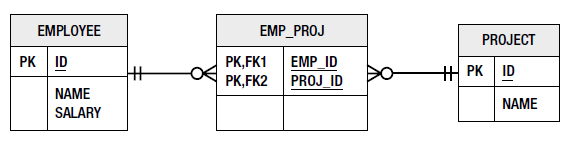
1. **One-To-Many –** 
   1. Bidirectional – in older Version Java there was not implemented collection and we must define target type of entity in @OneToMany(targetType=Employee.class, mappedBy=”..”)



Many-To-One should be the owning side, One-To-Many – inverse side =>mappedBy. Without mappedBy => provider will that handle it as unidirectional One-To-Many

1. **Many-To-Many –** is expressed on both the source and the target entities as @ManyToMany. There is no JoinColumn. When i. e. persistence provider generate schema for us JoinTable can has default name <Owner>\_<Inverse> (names of entities). Ids in join table will be with default names. Same approach as default names by One-To-Many. ([attribute\_name + \_ + entity id name)

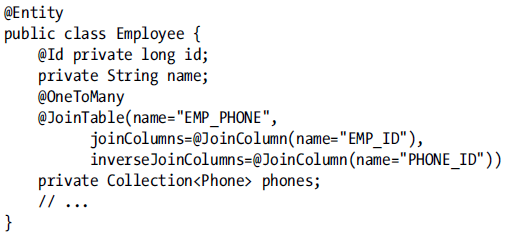
 



**Unidirectional relationships**

By many-to-many relationship when one side doesn’t have mapping to the other side => unidirectional (in that case Join table still be used, only one entity load data and store entity associations). There is not attribute in target entity to reference the source entity. The join table must now be specified as part of the mapping.

Uniderctional One-to-Many



The default naming is slightly different in the unidirectional case

1. The name of the join table is EMPLOYEE\_PHONE
2. JOIN column => PHONES\_ID

Lazy relationships

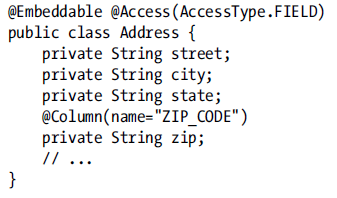
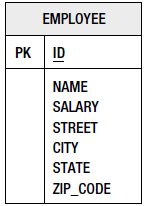
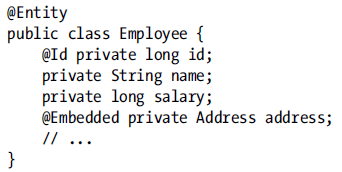
By default in all the relationships is EAGER in the single valued relationships @OneToOne(feth=FetchType.Lazy). Collection-valued relationships by default LAZY loaded

## Embedded Objects

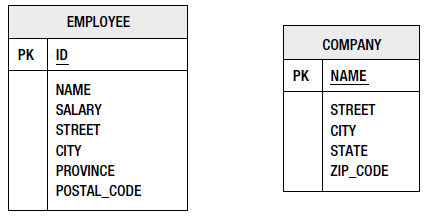
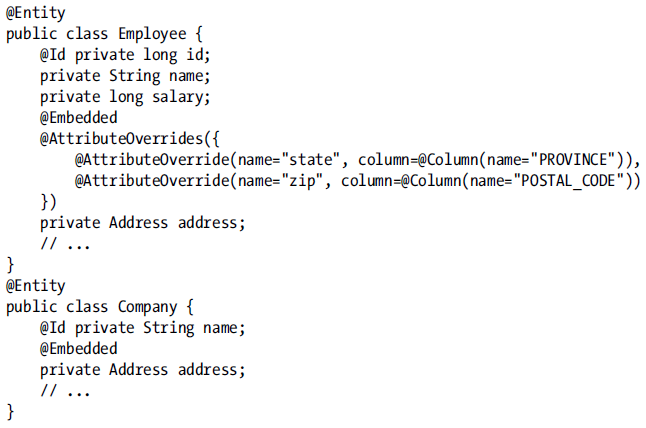
Embedded objects cannot be used in inheritance hierarchy.

@Embeddable annotation to the class definition.

@Embedded is optional

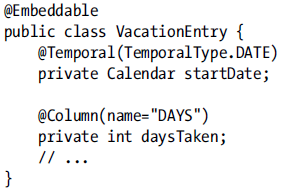
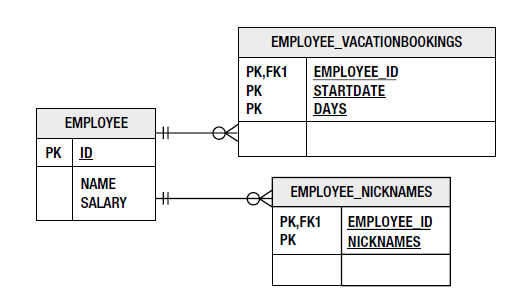
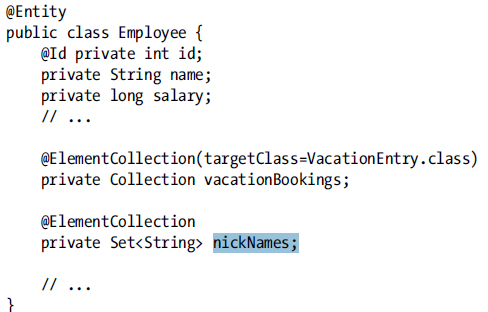
 

@AttributeOverride – used for each each attribute of the emedded object that we want to override in the entity. The column element allows us to specify the column that the attribute is being mapped to in the entity table. We indicate this in the form of a nested @Column annotation.

# Chapter 5: Collection Mapping

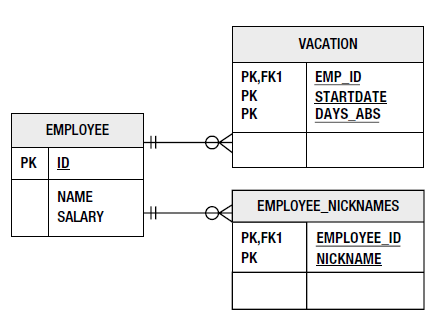
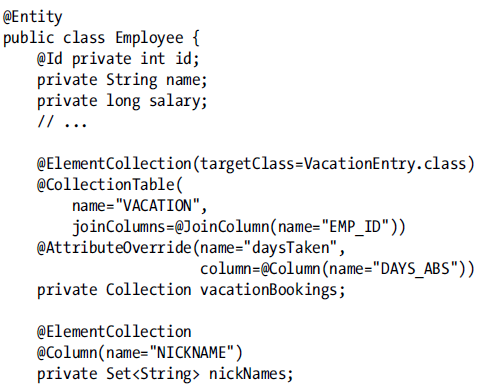
We can map collections of entities, embeddables, or basic types. Collections of embeddable and basic types are not relationships. Relationships needs minimally annotations @ManyToMany, @OneToMany. Elements collections are annotated with @ElementCollection(targetType= VacationEntry.class, fetchtype = ) and required separate table called a **collection table.** Every collection table has to have join column that refers to the containing entity table.

@CollectionTable – (contain JoinColumn, name)

* default name => name of referencing entity + \_ + name of the entity attribute that contains the element collection. [EMPLOYEE\_VACATIONBOOKINGS] [EMPLOYEE\_NICKNAMES]
* Default join column name =>name of referec. Entity + \_ + name of the primary key of the entity. [EMPLOYEE\_ID]

@AttributeOverride – for simple embedded mappings, the column specified is actually applies to the collection table, not to the entity table



## Different collection types

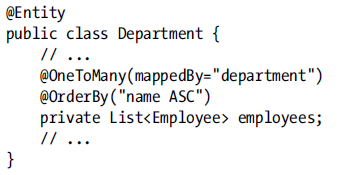
We can use Collection, Set, List, Map

### Lists

Retrieve entity in pre-defined order

1. **Ordering by entity or element attribute - @OrderBy –** we can order more columns separated by comma. Column should be comparable. (ASC is default value it is not needed to write it down). We can also order by embedded fields.

* Employee has a field info and this field is type of EmbeddableInfo then => @OrderBy("info.name ASC").
* Multilevel ordering @OrderBy("status DESC, name ASC")

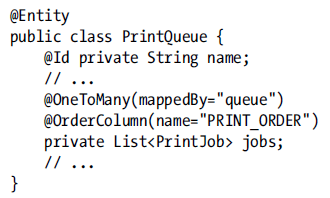
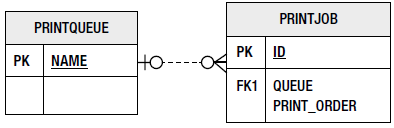


Simple changing the order of the items in a list in memory will not cause that order to be stored in the db at commit time.

1. **Persistently ordered lists - @OrderColumn**

Name is optional => default value is attribute name + \_ORDER [JOBS\_ORDER]

In many-to-many relations is order column in joinTable. There is problem due to update each row.



### Maps

1. If the values are entities maps has to be mapped one-to-many or many-to-many relationship
2. The values of map are either embeddable or basic types, map is mapped as an element collection
3. Map can be used for any collection mapping One-To-Many, Many-To-Many, ElementCollection,

JPA 2.0 allows for a Map where the key is not part of the target object to be persisted. The Map key can be any of the following:

1. A **Basic value**, stored in the target's table or join table.
2. An **Embedded object**, stored in the target's table or join table.
3. A foreign key to **another Entity**, stored in the target's table or join table.

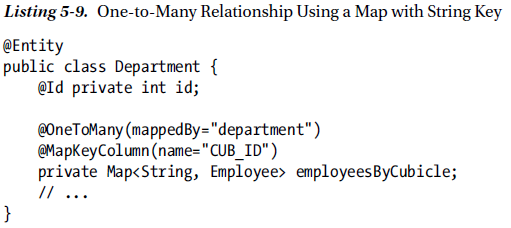
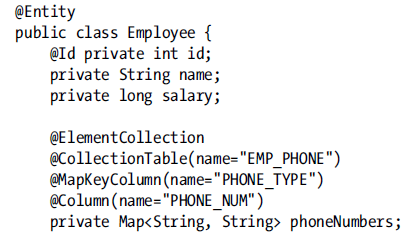
The type of mapping used is always determined by the value of the Map, not the key. So if the key is a Basic type and value is an entity => One-To-Many, in case value is Basic type and key is entity then ElementCollection is used.

1. For Many-To-Many the key is always stored in JoinTable.
2. For One-To-Many the key is stored in JoinTable if defined, otherwise is stored in target Entity
3. For ElementCollection the key is stored in the elements table.

* The @MapKeyColumn annotation or <map-key-column> XML element is used to define a map relationship where the key is a Basic value,
* The @MapKeyEnumerated and @MapKeyTemporal can also be used with this for Enum or Calendar types.
* The @MapKeyJoinColumn annotation or <map-key-join-column> XML element is used to define a map relationship where the key is an Entity value,
* The @MapKeyJoinColumns can also be used with this for composite foreign keys.
* The annotation @MapKeyClass or <map-key-class> XML element can be used when the key is an Embeddable or to specify the target class or type if generics are not used.

1. **Keying by basic type**

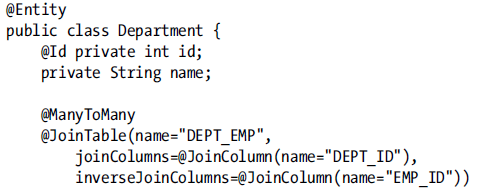
The keys will be stored in the same collection table in which the Map values are stored. Likewise, if it is a one-to-many relationship, and the foreign key is in the target entity table, the keys will be in the target entity table. If the relationship mapping uses a join table, the keys will be in the join table



@MapKeyColumn – indicate the column in the collection table that stores the basic key. When this annotation is not specified the key is stored in a column named after the mapped collection attribute appended with the \_KEY suffix [PHONENUMBERS\_KEY].

@MapKeyEnumarated(EnumType.String), @MapKeyTemporal – are applicable to keys that are of a basic type

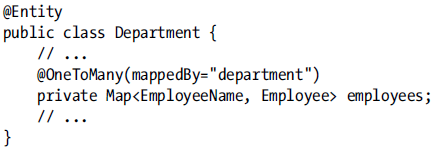
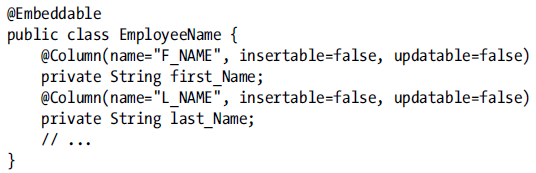
Many-to-Many relationship using a Map With strings, (CUB\_ID will be stored in join table). Map can be used only on one side of a many-to-many relationship, it make no difference on which side.



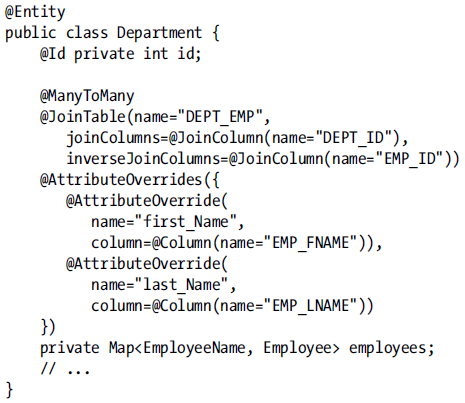
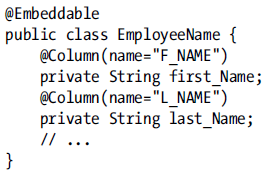
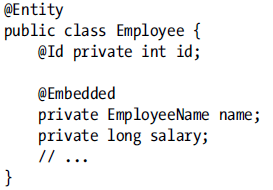


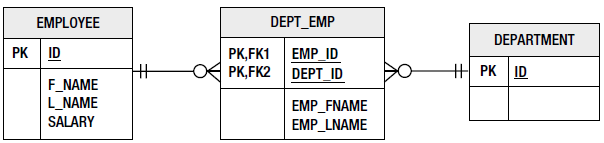
1. **Keying by Entity Attribute –** Similar as previous sample we can use attribute of entity as key in the similar way as basic type. It is need in order to was key unique in scope of relationship. @MapKeyColumn is not used
2. **Keying by Embeddable Type –** (Not good idea). Embeddables are not queryable. Embeddable has not uniqueness. There must be uniqueness defined (applicable within map). Uniqueness constraint is on the combination of the embedded attributes and the FK column to the source entity.

**Sharing Embeddable Key Mappings with Values**

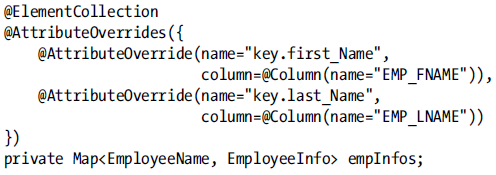


**Overriding Embeddable Attributes (Many-to-Many)**



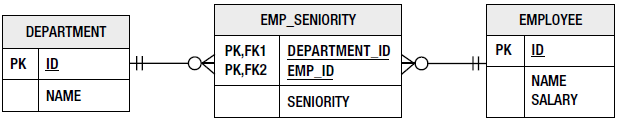
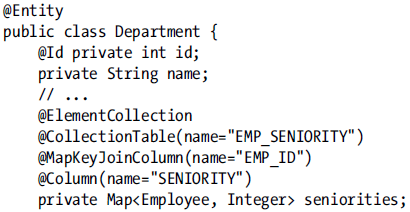


We has @ElementCollection instead Many-To-Many. We want to differentiate between key and values with prefixes

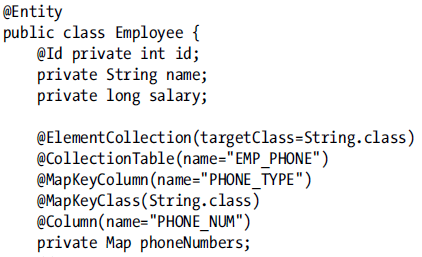


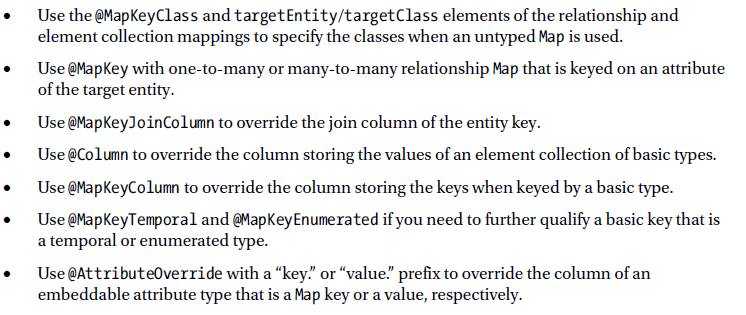
1. **Keying by Entity –** Non-parameterized style of maps(Entities are globally unique- within PU)

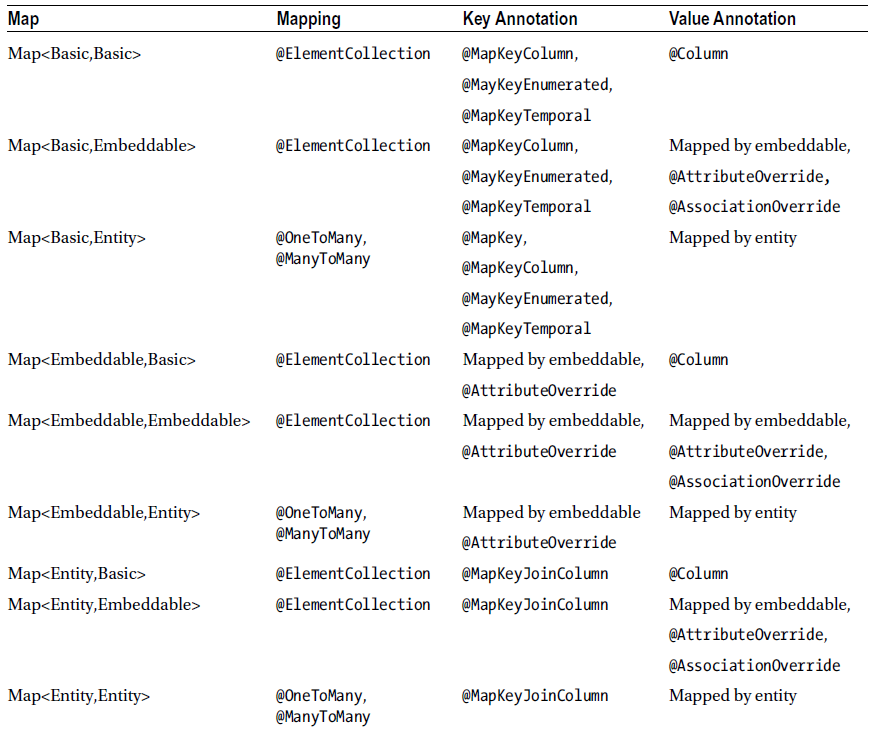
@MapKeyJoinColumn – override the join column defaults for an entity key. Default = [name of relationship or element collection attribute + \_KEY]

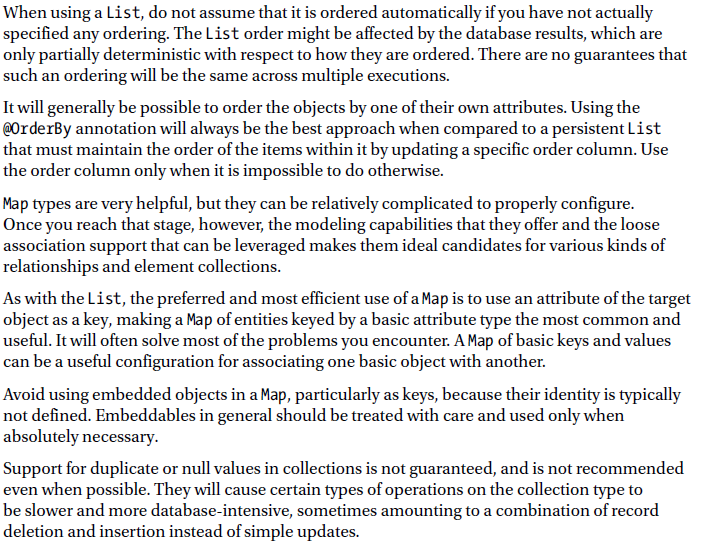


1. **Untyped Maps –** (Untyped Element collection of Strings with String Keys. @MapKeyClass – might be used when attribute is not of target Entity or when the key class cannot be deduced from the attribute definition or the other mapping metadata.









# Chapter 6: Entity manager [EM]

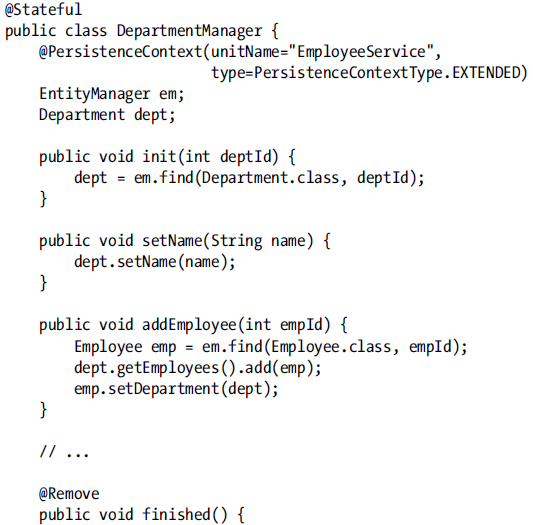
EM manages a persistence context. Persistent context is accessed indirectly by EM.

## Container-managed EM [CMEM]

In java EE app usually @PersistenceContext is used to acquire EM => CMEM. Container manages the lifecycle of EM. CMEM in two varieties:

* Transaction-scoped – persistence contexts managed by EM are scoped by the active JTA transaction, new persistence context is created when doesn’t exist. EM created with @PersistenceContext annotation. Is a stateless. When the transaction is over, the persistence context goes away.
* Extended – works with single persistent context => is tied to the lifecycle of stateful session bean (potentially spanning multiple transactions)

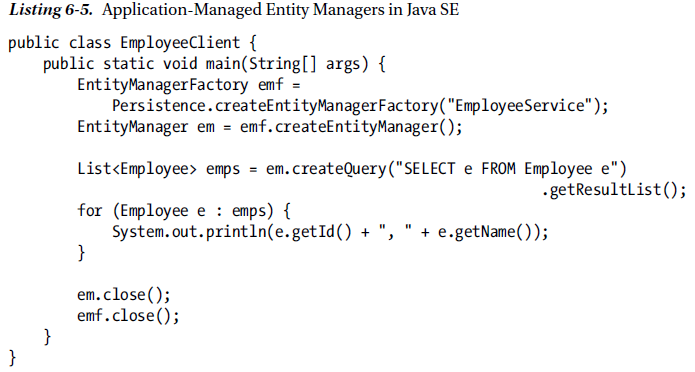
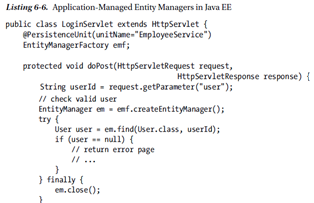
Extended



PersistenceContextType (Extended, Transaction). Persistence context is created when stateful session bean instance is created and it lasts until the bean is removed (any changes will be automatically stored in db).

## Application managed EM

It is created by using EntityManagerFactory. Persistence.createEntityMangerFactory can be with and without parameters. Method without parameters by default uses setting from persistence.xml. Method with parameters overrides persistence.xml settings. In Java EE EMfactory is closed by container by default.

## Transaction Management

Transaction type is defined for a persistence unit and is configure using the persistence.xml file.Two types of transaction management supported by JPA

* Resource local transactions of the JDBC drivers (referenced by persistence unit)
* JTA transactions, transactions of the Java EE server (support multiple participating resources, transaction lifecycle management, distributed XA transactions)

CMEM always use JTA transactions.

Application managed EM [APEM] can use either type.

### JTA transaction management

1. Transaction synchronization – is a process by which a persistence context is registered with transaction (persistence context can be notified when a transaction commits => provider uses this notification to ensure that a given persistence context is correctly flushed to the db).
2. Transaction association – is an act of binding persistence context to the transaction.
3. Transaction propagation – is a sharing persistence context between multiple-managed EM in a single transaction.

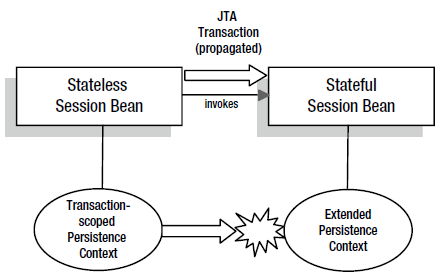
**Transaction-Scoped Persistence contexts**

Lazy loaded persistence context be EM’s when is not created. Transaction association is in time when EM call some method (em.find(..)). When is created PC EM uses this PC.

**Extended PC**

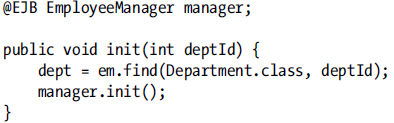
Lifecycle is bounded to the stateful session bean. Extended EM always use the same PC. Transaction association is eager (automatically when method bean is called, In case of bean managed transactions asap as UserTransaction.begin() is invoked). It is possible to share extended PC with other transaction-scoped EM. This context would always try to make itself the active PC.

**PC collision**



**PC inheritance**

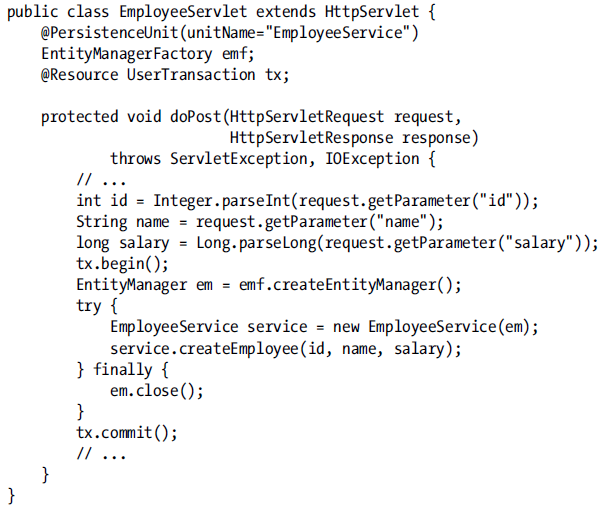
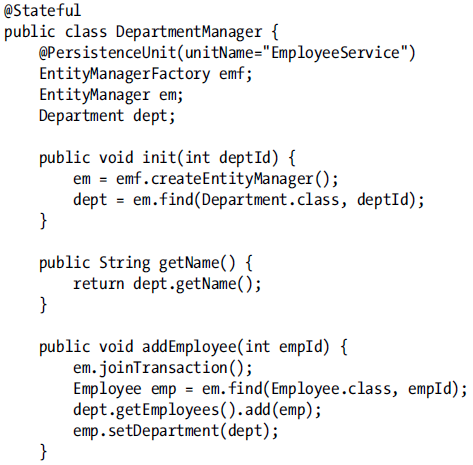
One stateful session bean inherits PC from the other. (EmployeeManager is stateful and inherits PC from DepartmentManger stateful bean, this PC is inherited when EmployeeManger is ingected into DepartmentManager bean)



**Application Managed PC**

It can be synchronized with JTA transactions. There is no restriction to number of application-managed PC’s but container managed PC can be only one. JTA transaction can be managed in two ways:

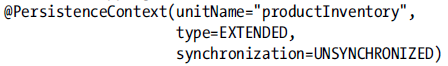
* If the PC is created inside the transaction => persistence provider automatically synchronize PC with the transaction
* If the PC was created earlier => the PC can be manually synchronized with transaction by calling EM method joinTransaction()



This EM doesn’t propagate transaction only way how to deals with it is using is passing EM as method parameter. Changes can be written to the database after then EM is closed. (EM is closed but PC is still alive). Best practice is use only one PC synchronized with transaction.

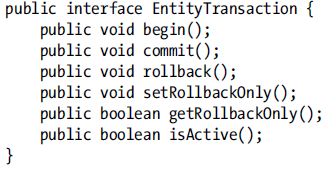
**Unsynchronized PC**

Added in JPA 2.1. It is suitable for conversational use case. When unsynchronized PC has not been joined to a transaction, no writes to the db,(will be thrown exception). An unsynchronized persistence context is never propagated into synchronized PC.



**Resource-Local Transactions**

By explicitly are controlled by application (app has no part on management). Entity transaction is acquired from EM by calling getTransaction().



When transaction is running and we attempt to begin new then IllegalStateException will be thrown. PersistenceException will be thrown in case when some error occurs by rollback. RollbackException will be thrown to indicate that commit failure. If transaction is marked for rollback but commit occurs then will be thrown RollbackException. getRollbackOnly() method can be called to determine whether the transaction is in failed state.

**Transaction Rollback and Entity State**

By rollback can become the next obscureness:

* By creating new entity with automatic generated primary key. Clear the primary key before attempting to persist the entity again. And do not relay on the primary key value in the detached entity.
* Be version field which is used for locking purposes can be set incorrect value.

In case when a want to use data from detached entities after rollback => consider selectively copying the changed data into new managed enitites.

## Choosing an EM

* Container-managed, transaction-scoped is appropriate for many applications.
* Container-managed, Extended persistence contexts – is not suitable for all applications.
* Application-managed if they need to be accessed by unmanaged classes (lack of propagation)

## EM Operations

### Persisting an Entity

After persisting new Entity this entity becomes managed (managed doesn’t mean that it is persisted right away). We can check whether entity is already managed with contains() method. If we want to persist entity which already exists EntityExistsException will be thrown. When persist() is invoked outside of transaction, the behavior depends on the type of EM.

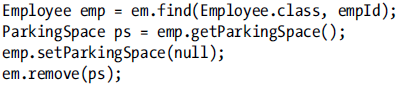
* Transaction-scoped EM will throw TransactionRequiredException.
* Application-managed and extended will accept the persist request => causing that entity to become managed, but no actions will be taken until the new transaction begins and PC becomes synchronized with the transaction.

### Finding an Entity

Return managed entity in all cases except when is invoked outside of transaction on transaction-scoped EM ( in that case entity is returned in detached state).

getReference() removes the need the retrieve the target entity instance, when is invoked provider returns a proxy to Reference entity. Drawbacks => proxy can be used, It might throw EntityNotFoundException in case when it is unable to locate real entity instance and we want to access other field then PK(use only in case when you are sure that the reference to primary key exist).

### Removing an Entity

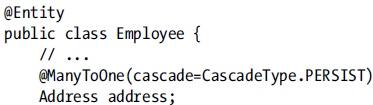


### Cascading Operations

CascadeType enum with values (PERSIST, REFRESH, REMOVE, MERGE, DETACH, ALL) -> this value indicate which operation can be cascaded. By default is empty cascade set. Cascade settings are unidirectional.

**Cascade Persist**

If you want to persist only address entity, Employee entity would not to be managed.



**Cascade Remove**

In gives sense only for one-to-one and one-to-many relationships. If it is save to use remove cascade then it is also save to use persist cascade. Remove() can be safely cascaded only from parent to child. It is needed to set cascaded entity or collection to null !!!!!

**Clearing the PC**

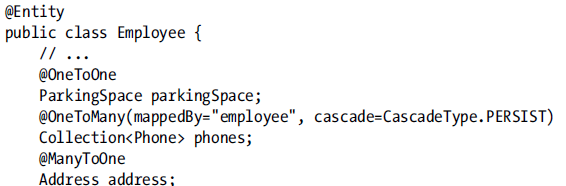
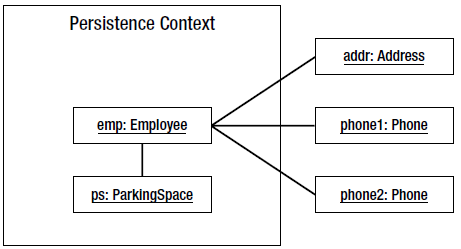
Needed only for application-managed and extended PC. Use clear() method of EM. Selectively cancelling the management of any particular entity instance while the persistence context is still open is achieved via detach() (is working cascading if cascade is set to DETACH). Clearing when there are uncommitted changes is dangerous. When you are in transaction and changes were stored to db and was invoked clear(). Changes in db would not be rollbacked.

### Synchronization with DB

Flush – storing data to db.

When a flush occurs, the EM cascade over the managed entities and looks for new entities that have been added to relationships with cascade persist enabled.

Any time a detached object that is the target of the one-to-one or many-to-one relationship is encountered and the source entity is owner, flush will be processed because owning entity doesn’t depend on the target entity (the owning entity has FK column and needs to store only PK value of the target entity). => look on the Address

If the PERSIST cascade is not set and target is not managed will be thrown IllegalStateException (except One-to-One, Many-to-one => previous text)

## Detachment and Merging

Detached entity cannot be used with any EM operation that requires managed entities. Opposite to detachment is merging. By merging any changes to entity that were made on the detached entity overwrite the current values in the PC.

### Detachment

Situation which leads to detachment:

* When the transaction that a transaction-scoped persistence context is associated with commits, all the entities managed by the persistence context become detached.
* If an application-managed persistence context is closed, all its managed entities become detached.
* If a stateful session bean with an extended persistence context is removed, all its managed entities become detached.
* If the clear() method of an entity manager is used, it detaches all the entities in the

persistence context managed by that entity manager.

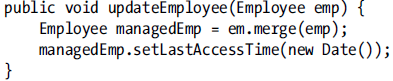
* If the detach() method of an entity manager is used, it detaches a single entity instance from

the persistence context managed by that entity manager.

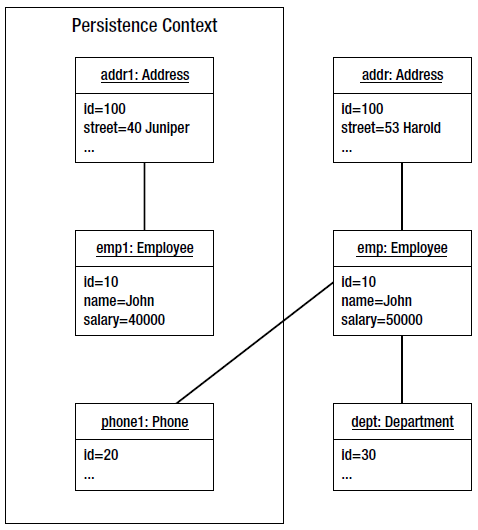
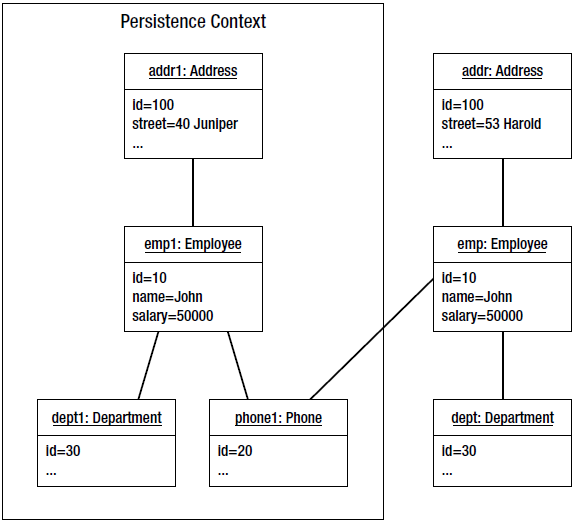
* When transaction rollback occurs, it causes all entities in all persistence contexts associated with the transaction to become detached.
* When an entity is serialized, the serialized form of the entity is detached from its persistence context.

By lazyLoading occur some problems. Because when entity is detached with lazyLoad attribute and this attribute had not be retrieved yet => behavior is not defined. In the presence of lazy loading attributes care must be taken to ensure that all the information you need to access offline is available

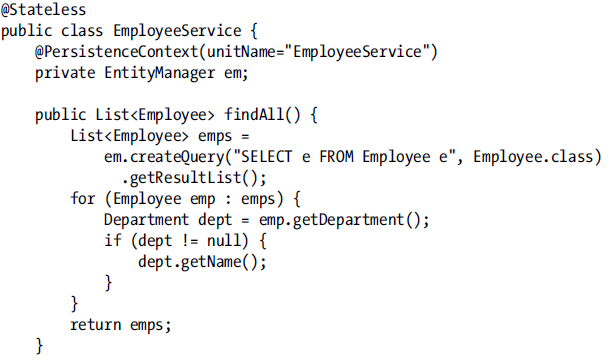
**Merging Detached Entities**



It creates new copy of entity. If the entity has a relationship to an object that has no persistent identity, the outcome of the merge operation is undefined. If merge on removed Entity will be throw IllegalArgumentException. LazyLoading => if entity was not load (ignored) or in case when entity attribute was loaded but set to null relationship will be cleared during the merge.

**Triggering Lazy Loading**

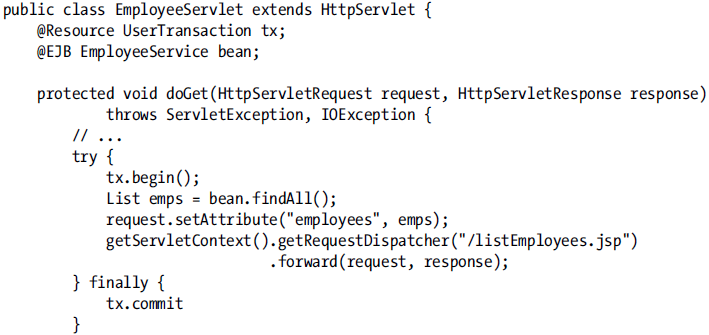


**Avoiding Detachment**

In case when we do not use entities but we copy entity data into different data structure.[Transfer Object] or use projection queries

1. **Transaction view**

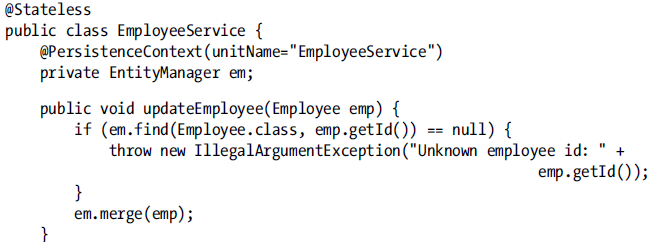
It includes both session bean invocation, and JPS page rendering[Transaction View Pattern]



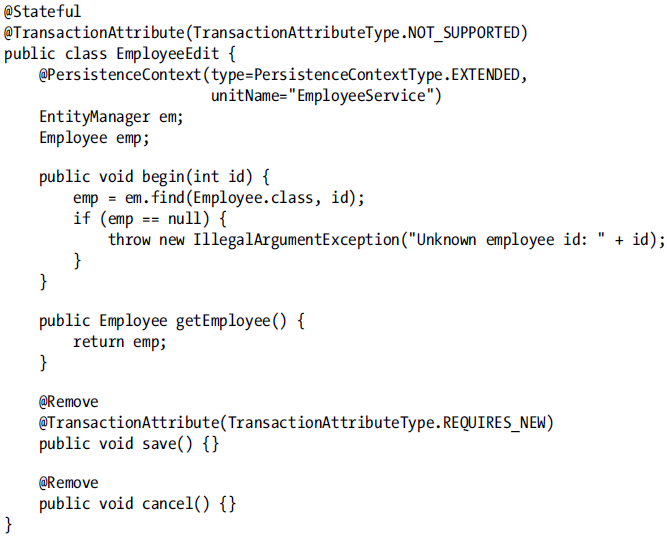
1. **Entity Manger per Request**

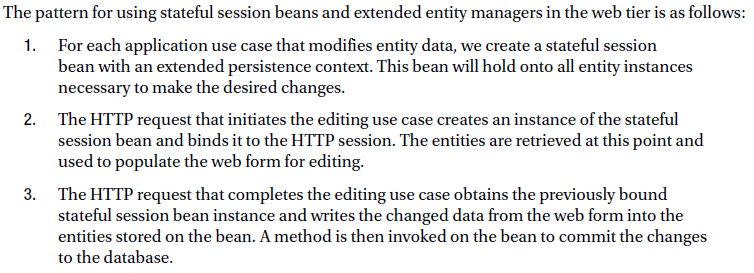
**Merge Strategies**

1. **Session Façade**

1. **Edit Session**





1. **Conversation**

# Chapter 7: Using queries

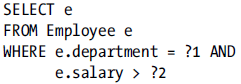
## JPQL

### Language

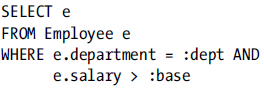
It is built on EJB QL. Entity expressions are used instead of column references. Result can be only single value as an entity instance or persistent field type. Five aggregate functions (Avg, Min, Max, Count, Sum). We can use many aggregate functions in one query. SELECT d, COUNT(e), MAX(e.salary)…

JPQL supports two types of query parameters binding

1. Positional binding



1. Named parameters



## Defining Queries

JPA provides the Query and TypedQuery interface to configure and execute quries.

Query -> when result type is object or dynamic queries when result type may not be known ahead of time.

TypedQuery -> result type is known. Extends Query.

Three approaches exist how to define query:

* Dynamically specified at runtime (Strings).
* Configured in persistence unit metadata (annotation or XML) and referenced by name (named queries)
* Dynamically specified and saved to be later referenced by name

### Dynamic Query Definition

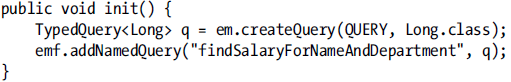
Using for complex queries but there can be issue with processing dynamic queries. (In some cases queries are cached). Sql injection !!! em.createQuery(

### Named Query Definition

@NamedQuery – may be placed on the class definition or entity. Name of named query has to be unique in persistence unit scope. Good practice use prefixes “Employee.findAll”. When we want to define more named queries for one class use => @NamedQueries( array of @NamedQuery). These queries can be also defined or redefined in XML files. Query cannot be redefined at runtime. Em.createNamedQuery.

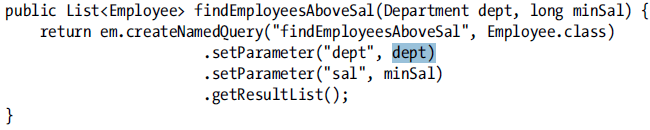
### Dynamic Named Queries

Dynamically created query and then stored as named in the EMF. If name already exists the old one would be overwritten with new one. JPA 2.1



## Parameter Types

With the query parameters is that the same parameter can be used multiple times in the query string yet only to be bound only once by using setparamter().



Binding Date parameters



## Executing Queries

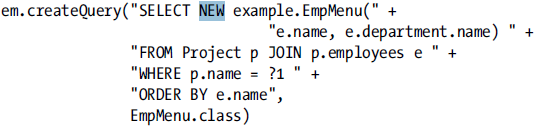
Each interface provides three different ways to execute a query. getSingleResult() [throws NoResultException, NoneUniqueResultExceptoin], getResultList()[when no items collection is empty], executeUpdate() -> invoke bulk update and delete queries. Query and TypedQuery objects can be reused as often as PC that was used to create the query is still active.

**Optimizing Read-Only Queries**

Use TransactionAttributeType.NOT\_SUPPORTED – any active transaction will be suspended

**Special Result Type**

Class which use used in select is a bit specific. It has to have Constructor with arguments which match exact type and order to select result.

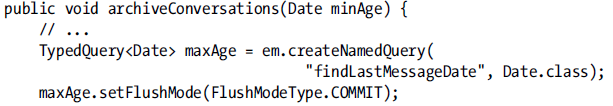


**Query paging**

setMaxResults(), setFirstResults() [should be not used with queries that join across collection relationship (one-to-many, many-to-many -> because there can occurs duplicate values] => getFirstResult(), getMaxResults() [see Stateful report pager]

**Queries and Uncomitted changes**

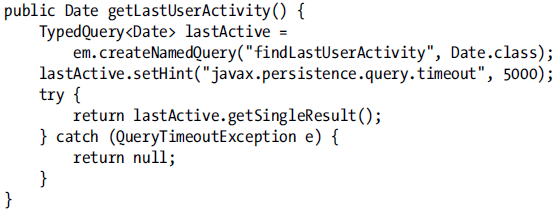
Changes are flushed to db on commit. When some changes are made meanwhile how transaction is not committed provider relies on internal cache synchronization. Database executes queries and not provider can be problem.

These two queries are dependent to each other and flushing changes to db could be very expensive operation -> because hundreds of active conversations are being checked. EM and Query interface support setFlushMode(),getFlushMode() ->FlushModeType.AUTO/COMMIT [default is AUTO]-> pending changes are included in query results. COMMIT -> queries don’t overlap with changed data in the persistence context. Flush mode is set to EM but it is property for PC. For transaction-scoped EM has to be changed for every transaction. Extended and application-manged EM will preserve their flush mode setting across transactions. Flush mode will be not applied to non-synchronized PC with transactions. Setting the flush mode on EM will applies to all queries, while setting the flush mode for a query limits the setting to that scope.

**Query timeouts**

Setting query timeout may be not supported by all database platforms or persistence provider.



## Bulk Update and Delete

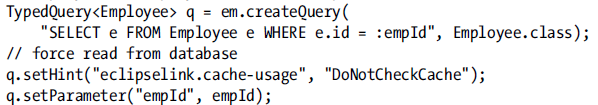
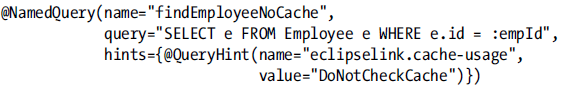
Can update only one entity type or embeddable objects. PC is not updated to reflect changes performed by this operation. Bulk operations are issued as SQL against the db. Should be performed in one transaction[TransactionAttributeType.REQUIRES\_NEW] or should be first in a transaction. A typical strategy for persistence providers dealing with bulk operations is to invalidate any in-memory cache of data related to the target entity. This forces data to be fetched from the database the next time it is required. Bulk operations and Extended PC are dangerous combination.

### Bulk Delete and Relationships

remove() is applied only on one entity, while DELETE statement can be applied on more entities. DELETE statements do not cascade to related entities -> first update related entities and set values of FK to primary entities to null and then DELETE main entities.

## Query Hints

Query hints are the JPA extension point for query features. Hint is a simple string name and object value. Hints can be used with Query, TypedQuery, @NamedQuery(hints can be only in string form) or set in xml file. Keep hints decoupled from the code.

## Query BestPractices

When it is possible use @NamedQueries. Named queries can be overridden with SQL queries or even with the vendor specific. Host queries in stateless session beans.

Consider

* Pagination is not exactly by JPA specified (look on your provider)
* When and how often persistence provider flush PC (look on your provider)

**Report Queries**

When you want to fetch data without intention to modify these data -> consider executing queries using transaction-scoped EM but outside of a transaction.

# Chapter 8: Query Language

The set of entities and embeddables on which operate queries and are defined by a persistence unit is known as the abstract persistence schema. Queries are not case sensitive only Entity names, and entity attributes. Alias [single identification variable] identification is mandatory in JPQL

## Select Clause

SELECT OBJECT(d) FROM Department d -> it is the same as without OBJECT , but OBJECT(e.department) is illegal

* SELECT e.department FROM Employee e -> return also duplicates
* SELECT DISTINCT e.department FROM Employee e -> without duplicates
* SELECT d.employess FROM Department d => Is illegal (is a collection value path).

Selecting embeddables will be not managed. By selecting (SELECT e FROM Employee e) from this employee we retrieved managed embeddable.

**Constructor Expressions -** The result object type must be referred to by using fully qualified name of the object



## From Clause

<entity\_name> AS <identifier> Using AS is optional. The identifier -> standard java naming convention (case-insensitive).

In absence of JOIN conditions => Cartesian product

**JOIN Operator**

SELECT p FROM Employee e JOIN e.phones p

Three the same selects

* SELECT d FROM Employee e JOIN e.department d
* SELECT e.department FROM Employee e
* SELECT DISTINCT d FROM Department d, Employee e WHERE e.department = d

**Map Joins**

* SELECT e.name, p FROM Employee e JOIN e.phones p
* SELECT e.name, VALUE(p) FROM Employee e JOIN e.phones p
* SELECT e.name, KEY(p), VALUE(p) FROM Employee e JOIN e.phones p WHERE KEY(p) IN (‘Work’,’Cell’)

We can use ENTRY [can be used only in Select clause] keyword.

**Outer Join**

Only one side of the relationship is required to be complete. Return also Employee which has no department. ON(JPA 2.1) has to be used only with right side object in JOIN statement.

**Fetch Joins**

It is used to indicate that some data can be fetched eagerly. (Employee with lazy address) . In case on figure address is fetched eagerly. FETCH join retrieve data in cartersian and there exists a lot of duplicates. To remove duplicates use DISTINCT



## WHERE Clause

BETWEEN – values are inclusively.

LIKE – Wildcards (\_) – single character; (%) – multiple characters

COLLECTION OPERATIONS

The IS EMPTY operator is the logical equivalent of IS NULL, but for collections.

|  |  |  |
| --- | --- | --- |
|  | JPQL | SQL |
| SUBQUERIES |  |  |
| IS EMPTY |  |  |
| MEMBER OF |  |  |
| ALL(all has to match), ANY(only one has to match),SOME |  |  |
| SUBCLASS  DISCRIMINATION |  |  |
| DOWNCASTING (Was added in JPA 2.1 |  |  |

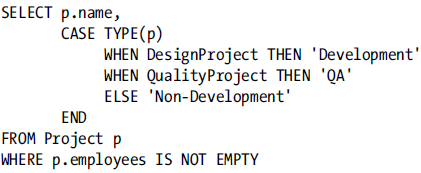
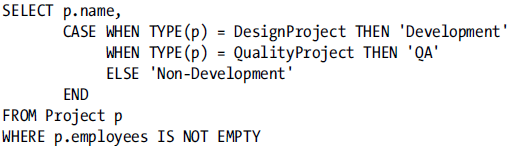
## Scalar Expressions

Using ENUM in JPQL – you have to write fully qualified name [WHERE key(p) = com.PhoneType.Home]

Native Database functions can be accessed using the FUNCTION expression. (since JPA 2.1)



CASE EXPRESSION – 4 Forms



COALESCE – first no-null expression will be returned

NULLIF – if first argument match second result will be NULL, if these two don’t match result will be first argument



## ORDER BY Clause

Default value is ASC.



This is not valid because we are retrieving only e.name and e.salary is not available.

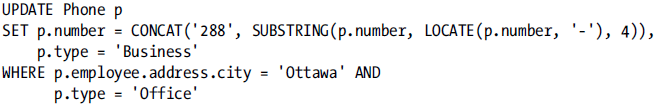
## Aggregate Queries

When is not defined GROUP BY data are grouped by entities.

AVG – returned type is double. COUNT – returned type is long, SUM – if are summed double values return will be double, if are summed long values result will be long.

HAVING – primary use case is for restrict results based on the aggregate function.

## UPDATE Queries

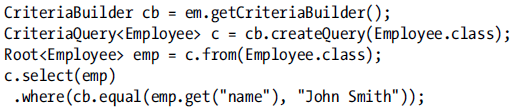


## DELETE Queries

It is not cascaded.



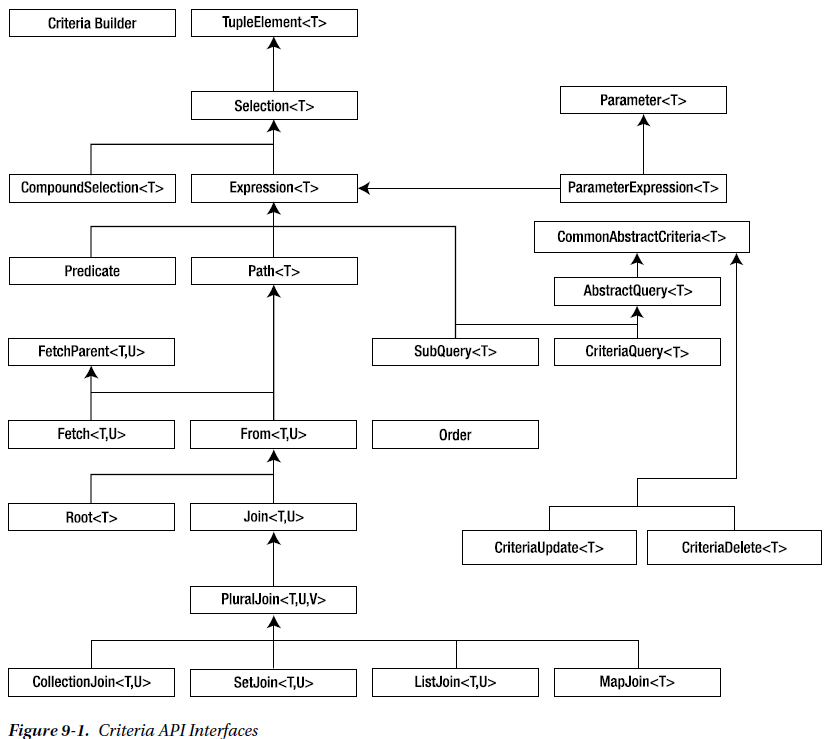
# Chapter 9: Criteria API



@SuppresWarnings(“unchecked”) – disable compiler warnings by type checking.

It is nice to use by generic queries or when we want to build parameters in loop.

CriteriaBuilder – createTypedQuery, createQuery, createTupleQuery = [createQuery(Tupple.class)] (Tuple is interface that contains an assortment of objects or data and applies to the aggregate parts.)



The majority of objects created through the Criteria API => immutable. CriteriaBuilder objects created by calling methods ( select,from, where) are mutable.

## Query Roots

We can create more roots (without where clause it behaves like Cartesian). Calls the from method are additive (add new root). Good practice to store roots locally and refers to them.

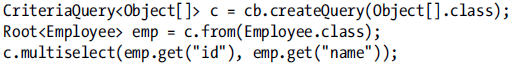


## SELECT Clause

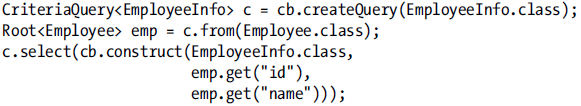
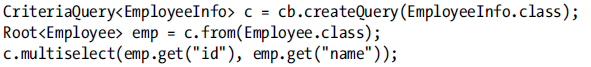
It is from CriteriaQuery interface. To argument to select() must be a type that is compatible with the result type of the query definition. When we want to select tuple then CompundSelection<Tuple> has to be passed to select(). If we want to create object via constructor then CompoundSelection<T> has to be passed to select() [T class type]. When result type is an array of object to select we have to pass CompoundSelection<Object[]> => these objects are created with the tuple(), construct(), array() methods of CriteriaBuilder interface.

Multi select – can have three forms.

* First form is for queries that have Object or Object[] as their result type. Result is always an instance of Object, but if multiple arguments are passed into multiselect() then the result has to be cast to Object[]

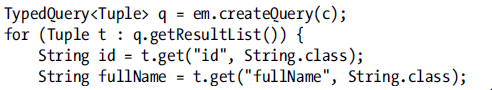


* Result is Tuple CriteriaQuery<Tuple> c = cb.createTupleQuery();
* Result is non-presistent type-> (same solutions)



### Using Aliases

It is good practice to use by tuples.



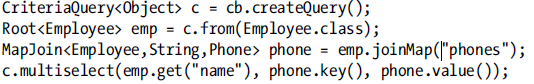
## From Clause

## Inner and Outer Joins

Joins are created by using join() method of the From interface. We can specify precise type of join by JoinType.INNER or JoinType.LEFT (By default inner join)



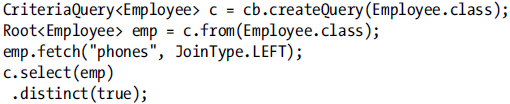
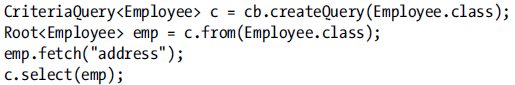
For joining to map we has to use special method joinMap.



Collection, Set and List relationships are handled with the joinCollection(), joinSet(), joinList()

## Fetch Joins

Result from the fetch is always type Fetch not Join.



## Where Clause

When we write second where this where overwrite the first one.

## Predicates

Conjunciton, Disjunction functions. There are type of gretherThan() and gt() methods. gt() is specific for numeric values and are strongly typed to work with number types.

Literals – must be wrapped with literal() method null literals are created with nullLiteral()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Criteria Query | | JPQL query | |
| Subqueries the AbstractQuery interface provides subquery(), correlate() –reuse root???? |  | |  | |
| IN |  | |  | |
| CASE – may not be supported by all databases. selectCase() |  | |  | |
| COALESCE |  | |  | |
| FUNCTION |  | | | |
| DOWNCASTING (JPA 2.1) |  |  | |
| OuterJOIN  (JPA 2.1) |  |  | | |
| Group BY,  HAVING |  |  | | |

Case expressions are non-mutating each when add new conditional expression and each value method adds an additional value to coalesce.

## BULK Update and Delete

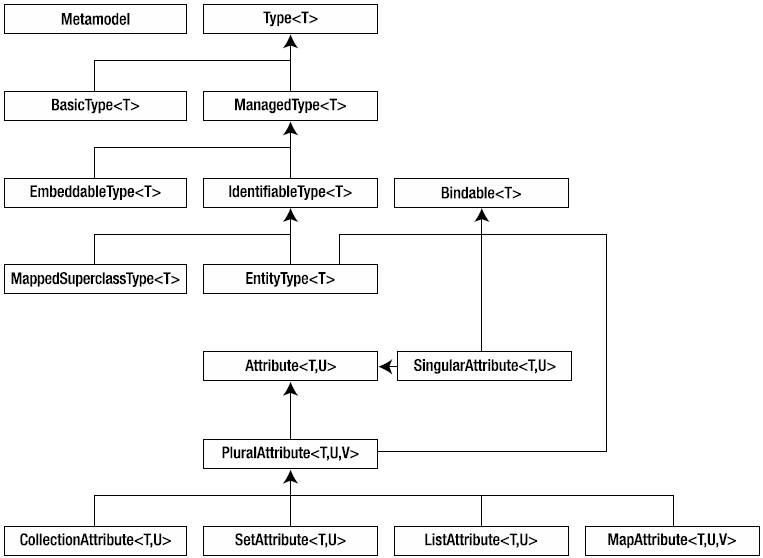
CriteriaDelete, CriteriaUpdate interfaces with specific methods. Bulk update can be created by using the createCriteraUpdate(), createCriteriaDelete() -> from CriteriaBuilder (JPA 2.1 in criteria API)

|  |  |  |
| --- | --- | --- |
|  | |  |
|  |  | |

## Strongly Typed Query Definitions

### The Metamodel API

The metamodel of persistence is a description of the persistence type, state and relationships of entities, embeddables, and managed classes. getMetamodel() from EM. It is initialized when EMF for the persistence unit was created

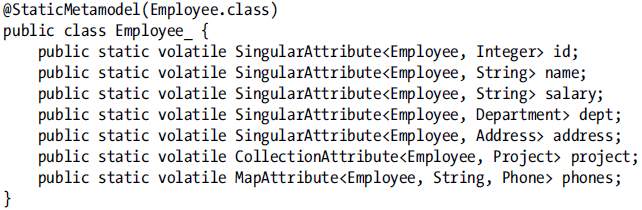


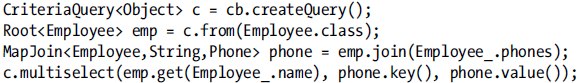


When input for entity is not pre-existing persistence type -> IllegalArgumentException

### The Canonical Metamodel

It consists of dedicated classes, typically generated, one per persistence class. It is generated in the same package as its associated persistent class and has the same name with the underscore. Canonical model can be generated but it depends on JPA provider.

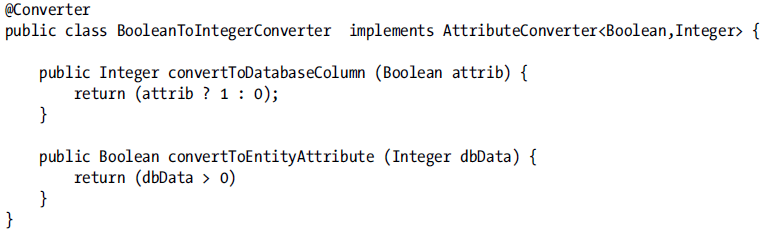




# Chapter 10: Advanced Object-Relational Mapping

Databases are case insensitive, but in case we want to use names EMPLOYEE and Employee we have to distinguish these two by following approach @Table(name=”\”Employee\””) @Table(name =”\”EMPLOYEE””). Xml escaping is a bit different that the one used in java (&) – is used to escape in xml. By including empty **delimited-identifiers** in the Xml mapping file all identifiers in the persistence unit will be treated as delimited and quotes will be passed to them when they are passed to the driver. (This setting cannot be overridden)

## Converting Entity State (JPA 2.1)

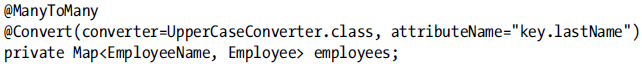
We can introduce custom converters -> by implementing AttributeConverter<X,Y> => first type is entity attribute and the second represents JDBC type. Converters are managed classes. In java SE converter class should be included in a class element in the persistence.xml descriptor.  

When bonded is attribute of embeddable object SecurityInfo then we ->



Converting collections

For list with primitive types it is simple but for map where key is embeddable i.e.



Converters cannot be used on identifier, version, relationship @Enumarated, @Temporal[with these annotations you cannot use converter, because it is useless] attributes. We can set in @Converter(autoApply =true). That means that every attribute with the specific will be converted using this converter. We can disable conversion by setting @Convert(disableConversion = true)

### Converters in Queries

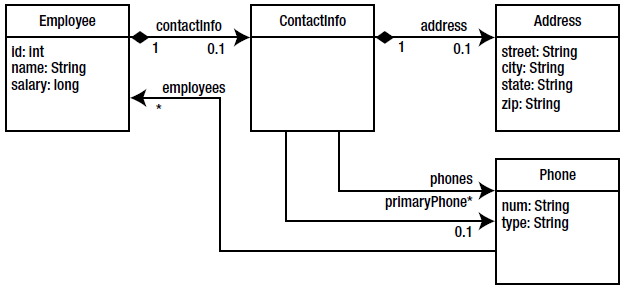
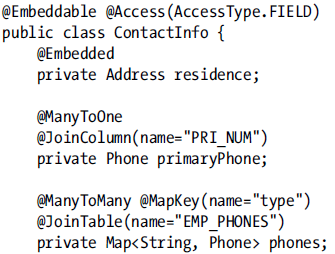
In some cases the converters can work properly

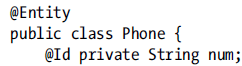
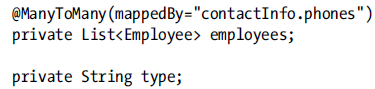


But in the some other case the converter doesn’t work properly (in some functions UPPER, MOD)



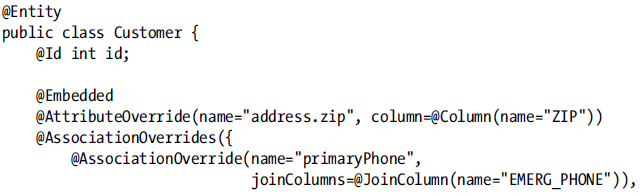
## Complex Embedded Queries

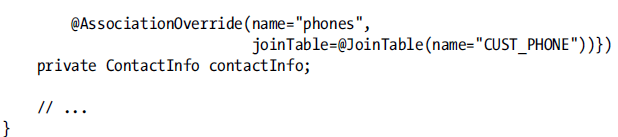
 

When bidirectional relationships exist within an embedded object, they are treated as though they exist in the owning entity, and the target entity points back to the owning entity, not to the embedded object. It must first contain the attribute within Employee that contains the embeddable, as well as a dot separator character (.)[contactInfo.phones] and the relationship attribute within the embeddable. Embeddable class cannot contain primary keys of joined entities, (embedabbles)

@AttributeOverride is used to override attributes but for relationships we need @AssociationOverride

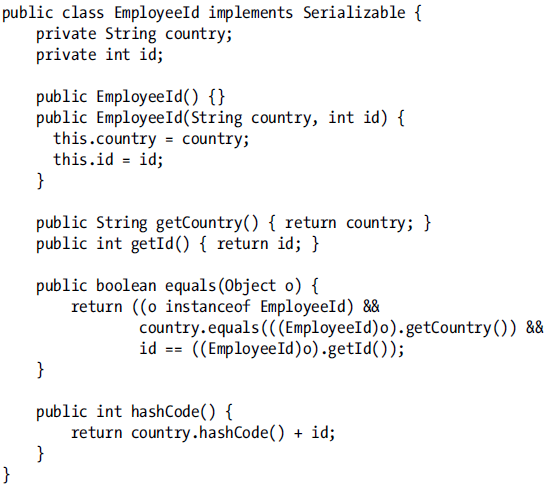




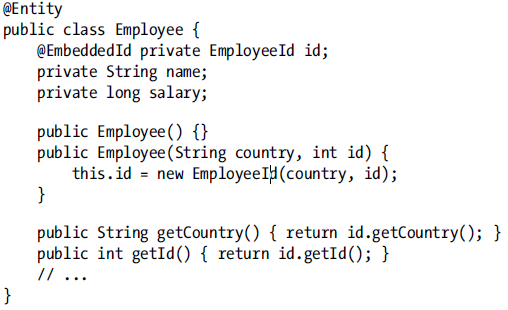
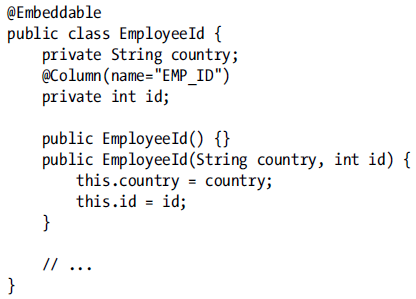
## Compound Primary Keys

Primary key class muss contain definition for hashCode, equals methods. Exists two types:

1. **Id Class** – Each field of entity that makes primary key is marked with @Id annotation. The primary key class is defined separately and associated with the entity by using the @IcClass annotation on the class definition

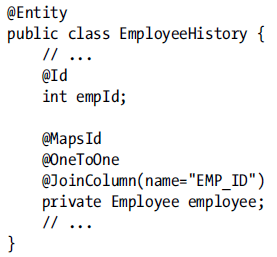
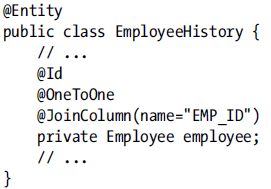


1. **Embedded ID class** – embedded id class access type can differ from that of that the entity use it.

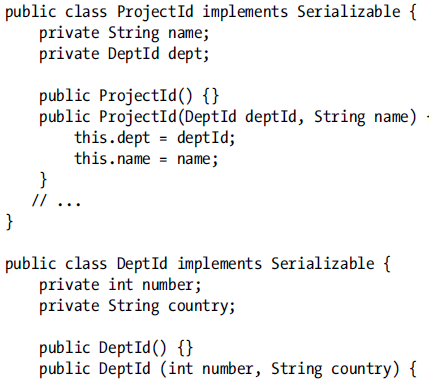
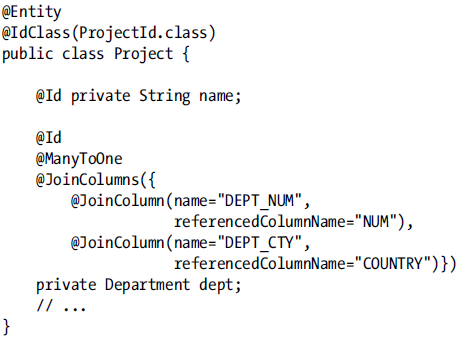


## Derived Identifiers

Identifier in one entity includes a foreign key to another entity. First has to be persisted primary object after then referential object.



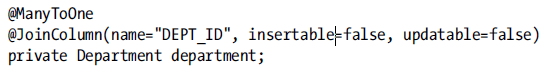
### Multiple mapped Attributes



## Advanced Mapping elements

### Read-only mappings

We want to ensure that data will be not modified or persisted if they are already in DB. Entita can be changed in memory, but changes will be not persisted to the DB. But entities can be deleted.



### Optionality

For @Basic, @ManyToOne and @OneToOne(optional =false) annotations. This annotation servers by generating schema if column has to be null or not null.

## Advanced Relationships

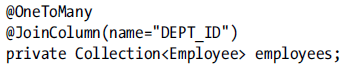
### Join Tables

One-To-One or Many-To-One relationships usually doesn’t need @JoinTable but in case where table already exists. By relationships many-to-many or one-to-many joinTable is physical annotation and must be defined on the owning side of the relationship ( the many side has to be always owner)



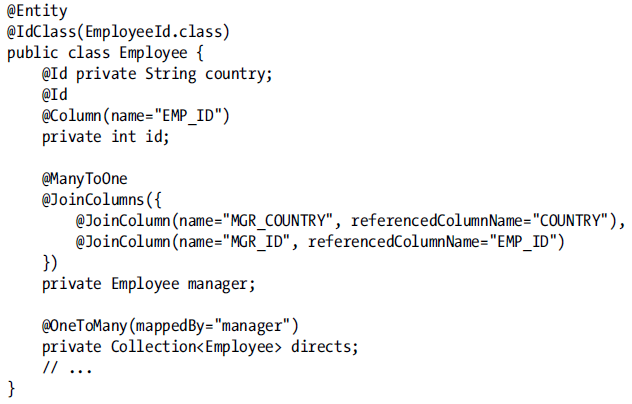
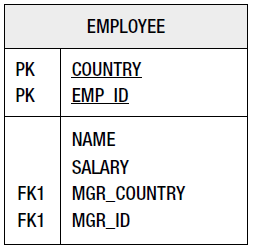
### Avoiding Join Tables

Unidirectional One-to-Many Mapping Using a Target Foreign Key. The none owning side of relationship does not change base upon whether the relationship is mapped using a join table or not. By this implementation there is a problem with performance (when state of Employee is changed and also Department to which employee belongs . When writing out the Employee state, the FK to the Department is not known because Employee entity does not have any reference to it. In this case, the Employee might have to be written twice (one for Employee change, and one for Department change -> and also FK from Employee to Department has to be updated)

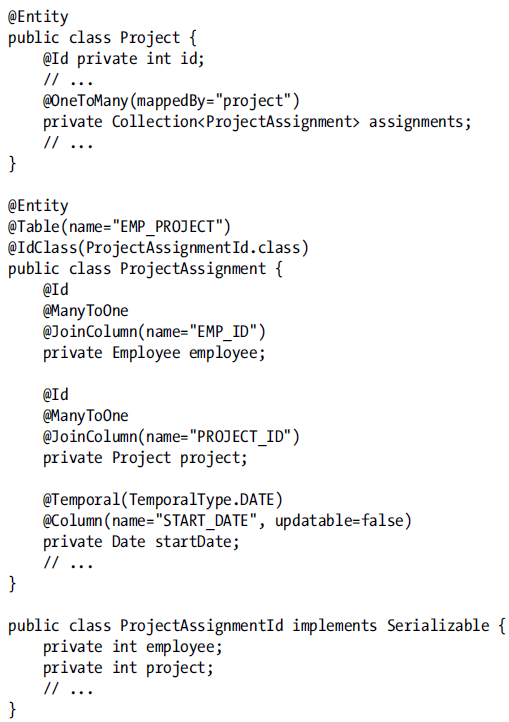


### Compound Join Columns

There are no default values for join column names when we have multiple join colums (Good practice is to set names and also referencedColumnName).

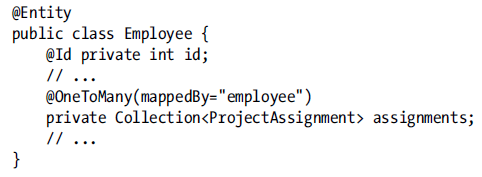


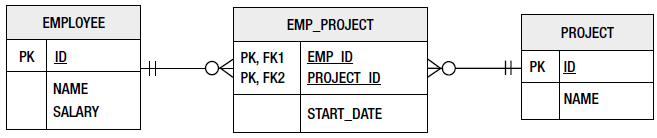
### Orphan removal

Only relationships with single cardinality on the source side can enable orphan removal @OneToOne, @OneToMany. It causes that child entity will be removed when the relationship between the parent and the child is broken. Setting orphan removal on a relationship automatically causes the relationship tho have the REMOVE operation option added to its cascade list. In case we have map keyed by different entity type, then orphan removal would only apply to the entity values in the map, not to keys.



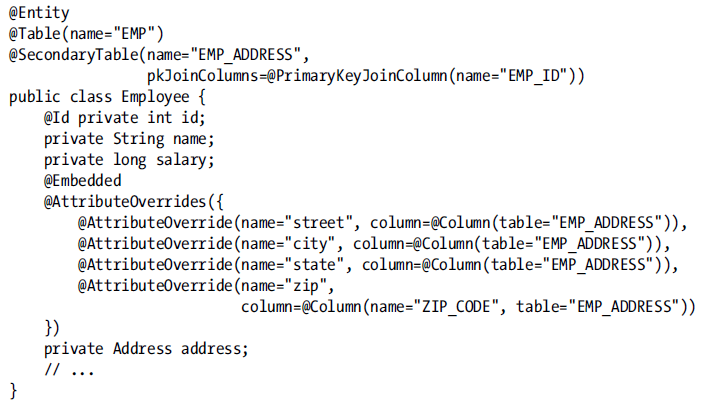
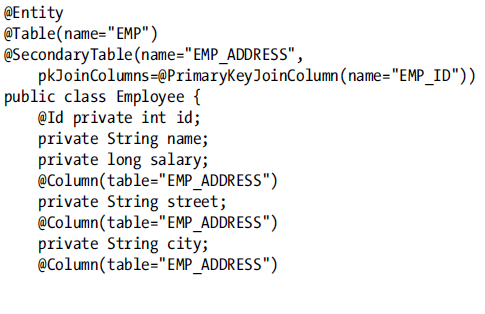
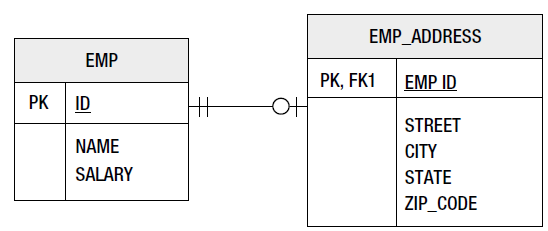
### Mapping relationship state

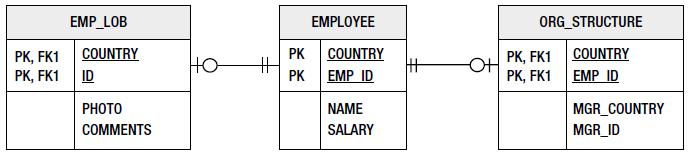




# Multiple tables

We can distribute data from entity in more tables





## Inheritance

Superclass is not a persistent class. It cannot be queried and annotations such as @Table are not permitted on mapped superclasses because the state defined in them applies only to its entity subclasses. Mapped superclass don’t count as levels in hierarchy they contribute only the entities beneath them. It cannot be used in relationships as target. => @MappedSuperclass

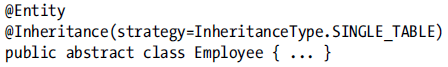
Some classes in hierarchy whose are not entities are called as transient. Entity can extend transient class also state will be inherited but this state will be not managed by persistence provider. Entita can be defined as an Abstract class.

### Inheritance models

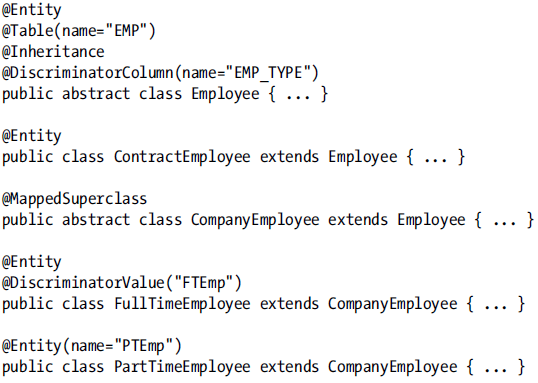
JPA provides three different data representations. The root entity class must signify the inheritance hierarchy by being annotated with the @Inheritance annotation. The inheritance strategy is inherited also by children.

### Single-Table Strategy

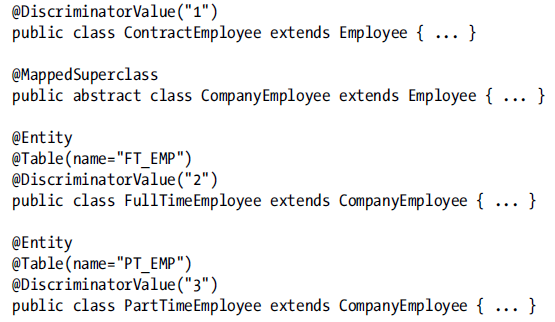
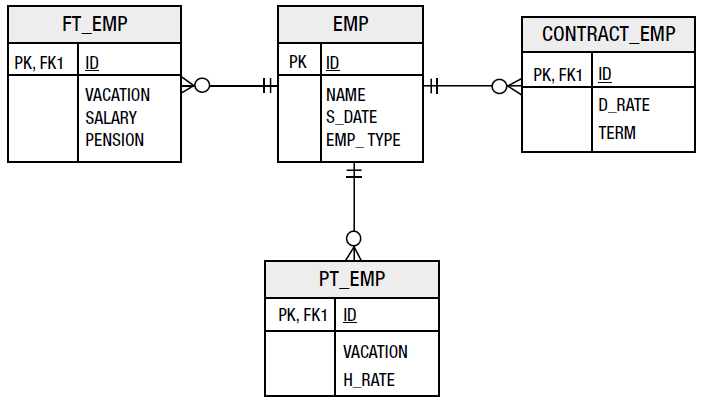
Default strategy.



Discriminator column by default is of type String and default Discriminator values are entity names (allowed types are String, Integer, Char)



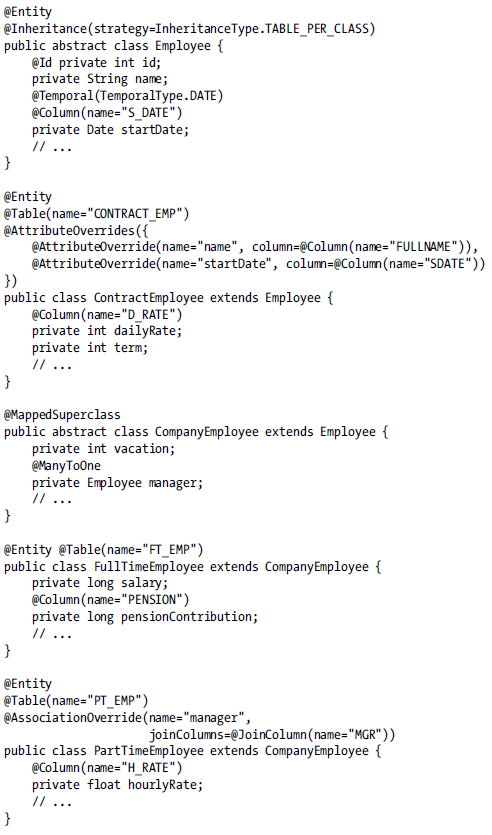
### Joined Strategy



### Table per concreate class strategy

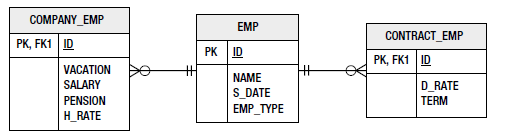
Each concreate entity is mapped to separate table. Problem with this strategy is that it must execute separate queries across each of the subclass tables. Rendundacy of columns, because in case one concreate entity inherit from another it inherits also some fields but these fields has to be in both db tables. Bright side is that when we want to make query with on concreate class we can do it. Discriminator is not needed.

If we want to override attribute => @AttributeOverride but if we want to override is association => @AssociationOverride (one-to-one, many-to-one)



### Mixed Inheritance

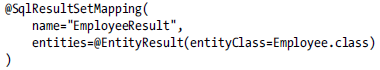
It is out of specification. It makes sense to mixes only single-table and joined inheritance types.



# Chapter 11: Advanced Queries

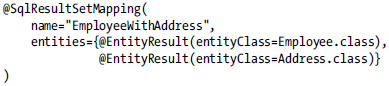
## SQL Queries

### SQL Result Set Mapping

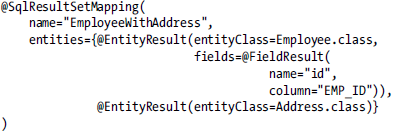


### Multiple Result Mappings

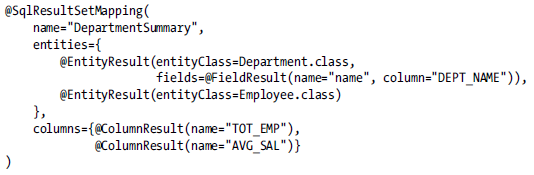
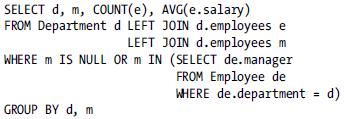
This only suitable for one-to-one relationships (otherwise there will be duplicate entities).



### Mapping column aliases

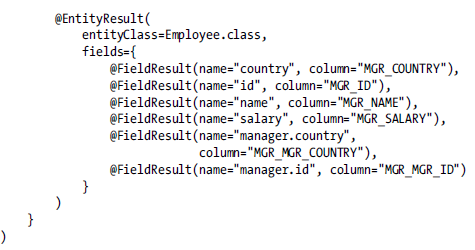
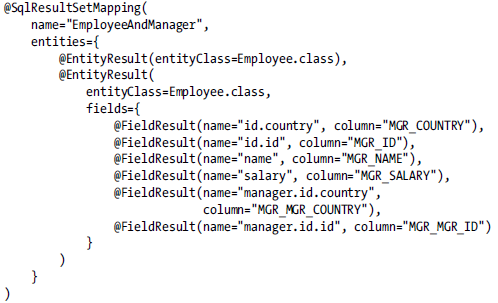


### Mapping scalar results



### Mapping compound keys

Embedded key Id class



### Mapping inheritance

If result includes more than one entity type, each of the columns for all the possible entity types must be presented in the query.

### Mapping to none-entities types (JPA 2.1)

@SqlResultSetMapping(  
 name="EmployeeDetailMapping",  
 classes={  
 @ConstructorResult(  
 targetClass=example.EmployeeDetails.class,  
 columns={  
 @ColumnResult(name="name"),  
 @ColumnResult(name="salary", type=Long.class),  
 @ColumnResult(name="deptName")  
 })})

JPQL solution

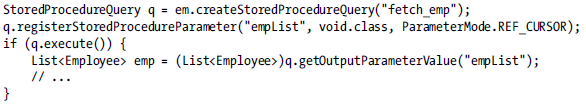


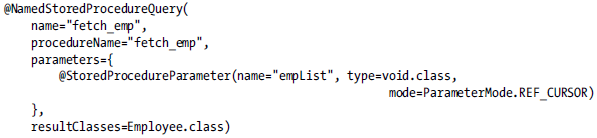
### Parameter Binding

JPA provides only positional parameters for native queries.

### Stored Procedures (JPA 2.1)

Defined new query type StoredProcedureQuery extends Query. Queries can be created programmatically from the EM or they may be defined



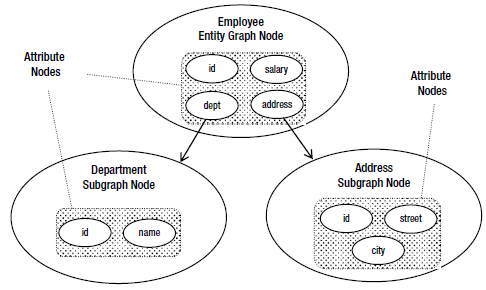
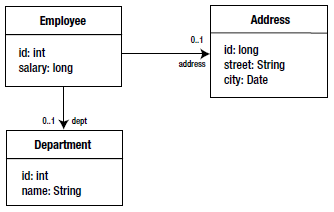


## Entity Graphs

It is a template for specifying entity and embeddable attributes. It serves as pattern that can be passed into a find method or query to specify which entity and embeddable attributes should be fetched in query result.

They contain only three types of objects.

1. **Entity graph nodes –** one entity graph node for every entity graph. It is root. It contains all of the attributes nodes and also all subgraph nodes
2. **Attribute nodes –** represents entity attribute or embeddable type.
3. **Subgraph nodes –** It contains subgraph nodes and attribute nodes, but represents a graph of attributes for an entity or embeddable type that is not the root type.



Entity graphs can be defined statically in annotation or dynamically through an API.

### Entity Graphs Annotations

Default name is by entity name. When you use all attributes you can use

@NamedEntityGraph(  
 attributeNodes={  
 @NamedAttributeNode("street"),  
 @NamedAttributeNode("city"),  
 @NamedAttributeNode("state"),  
 @NamedAttributeNode("zip")}  
)

@NamedEntityGraph(includeAllAttributes=true) => equivalent

@NamedEntityGraph(name="Employee.graph2",  
 attributeNodes={  
 @NamedAttributeNode("name"),  
 @NamedAttributeNode("salary"),  
 @NamedAttributeNode(value="address"),  
 @NamedAttributeNode(value="phones", subgraph="phone"),  
 @NamedAttributeNode(value="manager", subgraph="namedEmp"),  
 @NamedAttributeNode(value="department", subgraph="dept")},  
 subgraphs={  
 @NamedSubgraph(name="phone",  
 attributeNodes={  
 @NamedAttributeNode("number"),  
 @NamedAttributeNode("type"),  
 @NamedAttributeNode(value="employee", subgraph="namedEmp")}),  
 @NamedSubgraph(name="namedEmp",  
 attributeNodes={  
 @NamedAttributeNode("name")}),  
 @NamedSubgraph(name="dept",  
 attributeNodes={  
 @NamedAttributeNode("name")})  
 })

**Entity graphs with Inheritance**

@NamedEntityGraph(name="Employee.graph4",  
 attributeNodes={  
 @NamedAttributeNode("name"),  
 @NamedAttributeNode("address"),  
 @NamedAttributeNode(value="department", subgraph="dept")},  
 subgraphs={  
 @NamedSubgraph(name="dept",  
 attributeNodes={  
 @NamedAttributeNode("name")})},  
 subclassSubgraphs={  
 @NamedSubgraph(name="notUsed", type=ContractEmployee.class,  
 attributeNodes={  
 @NamedAttributeNode("hourlyRate")})  
 })

**Map Key Subgraphs**

@Embeddable  
public class EmployeeName {  
 private String firstName;  
 private String lastName;  
// ...  
}  
@Entity  
@NamedEntityGraph(name="Department.graph1",  
 attributeNodes={  
 @NamedAttributeNode("name"),  
 @NamedAttributeNode(value="employees",  
 subgraph="emp",  
 keySubgraph="empName")},  
 subgraphs={  
 @NamedSubgraph(name="emp",  
 attributeNodes={  
 @NamedAttributeNode(value="name",  
 subgraph="empName"),  
 @NamedAttributeNode("salary")}),  
 @NamedSubgraph(name="empName",  
 attributeNodes={  
 @NamedAttributeNode("firstName"),  
 @NamedAttributeNode("lastName")})  
 })  
public class Department {  
 @Id private int id;  
 private String name;  
 @OneToMany(mappedBy="department")  
 @MapKey(name="name")  
 private Map<EmployeeName, Employee> employees;  
// ...  
}

### Entity graph API

EntityGraph<Department> graph = em.createEntityGraph(Department.class);  
graph.addAttributeNodes("name");  
graph.addSubgraph("employees").addAttributeNodes("salary");  
graph.addKeySubgraph("employees").addAttributeNodes("firstName", "lastName");

## Managing Entity Graphs

### Accessing Named Entity Graphs

List<EntityGraph<? super Employee>> egList = em.getEntityGraphs(Employee.class);  
for (EntityGraph<? super Employee> graph : egList) {  
 System.*out*.println("EntityGraph: " + graph.getName());  
 List<AttributeNode<?>> attribs = graph.getAttributeNodes();  
 for (AttributeNode<?> attr : attribs) {  
 System.*out*.println(" Attribute: " + attr.getAttributeName());  
 }  
}

Can be fetched also some superclass of Employee (i.e. Person). We can add named entity graph also in code. When we specify name which already exist, name will be overwritten.

### Creating entity Graphs From Named Ones

In case when we want to use similar graph like existing entity graph and we need to change only some parts.

EntityGraph<?> graph = em.createEntityGraph("Employee.graph2");  
graph.addAttributeNodes("projects");  
em.getEntityManagerFactory().addNamedEntityGraph("Employee.newGraph", graph);

### Using entity graphs

Depending upon which property is used, the entity graph will take on the role of either a fetch graph or a load graph.

1. **Fetch Graphs –** is setting as a value for property => javax.persistence.fetchgraph

All attributes included in the graph are treated as having a fetch type EAGER. Attributes that are not included. We could simply pass graph to find method or as a query hint. To make a single attribute be lazy, you will need to specify all of the eager attributes. In other words, you cannot just override the fetch mode for a single attribute; when you create a fetch graph for an entity or embeddable type, you are effectively overriding all of the attributes of the type, either by including them or excluding them from the fetch graph.

Map<String,Object> props = new HashMap<String,Object>();  
props.put("javax.persistence.fetchgraph", em.getEntityGraph("Employee.graph2"));  
Employee emp = em.find(Employee.class, empId, props);

1. **Load Graphs -** is setting as a value for property -> javax.persistence.loadgraph

Difference with fetch graph is in that how are treated the missing attributes. In load graph all missing attributes are treated as they are defined in the mappings. The value of using load graph is in the ability to cause one or more attributes to be treated as Eager even though they were statically defined to be LAZY

EntityGraph<Employee> graph = em.createEntityGraph(Employee.class);  
graph.addAttributeNodes("department");  
TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e", Employee.class);  
query.setHint("javax.persistence.loadgraph", graph);

We can test loading behavior of provider by using the PersistenceUnitUtil.isLoaded() on lazy attribute before it is accessed.

# Chapter 12: Other Advanced Topics

## Callbacks

Lifecycle fall into four categories: persisting, updating, removing, loading. Every has post and pre events except load that has only PostLoad event. Prepersist event can occurs on merge() call when new entity has been merged into the PC. PostPerist occurs during the transaction completion phase. PreUpdate depends on implementation because when are updated one or more entities the method can be called once or after every update event. PostLoad callback occurs after data loading from db and when entity is already created. It can be triggered by any operation which load data from db (lazyloading, REFRESH)

Callback signature – return type void without params and without checked exception. Method can be annotated with multiple lifecycles event annotations. There are none-permited operations (operations on EM,

### Entity Listeners

Valid callback -> void foo(Object o) {}. Entity listeners has to be stateless => without fields. If CDI is enabled in container then listeners can be also CDI bean (JPA 2.1) but they are not contextual (instances are not stored in specific context) but they can have state. Listeners are invoked in order in which they are written in annotation. When exception occurs on one listener others will be not invoked.

@EntityListeners({EmployeeDebugListener.class, NameValidator.class})  
public class Employee implements NamedEntity{

}

public class NameValidator {  
 static final int *MAX\_NAME\_LEN* = 40;  
 @PrePersist  
 public void validate(NamedEntity obj) {  
 if (obj.getName().length()) > *MAX\_NAME\_LEN*)  
 throw new ValidationException("Identifier out of range");  
 }  
}

We can define default listeners in Xml file. These listeners will be used for all entities but we can exclude some entities with annotation @ExcludeDefaultListeners

First are invoked default listeners the next step is traverse through the hierarchy looking for entity listeners. Callbacks methods can be overridden.

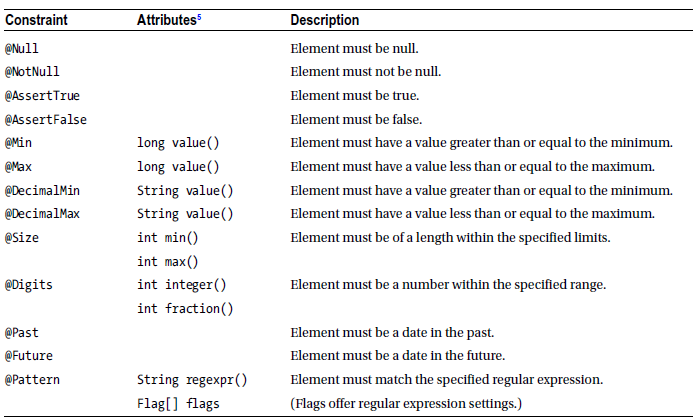
### Validation

Validation is invoked by calling validate() method from Validator class.

@NotNull(message="Employee name must be specified")  
@Size(max=40)  
private String name;

In container Validator instance may be injected into any Java EE component that supports injection. In none container environment Validator is obtained from ValidatorFactory.

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();  
Validator validator = factory.getValidator();



**Validation Groups**

They are used for determining what and when validation occurs. Field without groups is explicitly defined like the Default group and will be checked whenever either the Default group. Or no group is passed to the validate() method. PrePersist and PreUpdate lifecycle events trigger validation by Default validation group. No group will be validated during the PreRemove phase. We can change default settings with

* javax.persistence.validation.group.pre-persist: Set the groups to validate
* javax.persistence.validation.group.pre-update
* javax.persistence.validation.group.pre-remove

or with map which is passed as argument to createEntityManagerFactory()

<property name="javax.persistence.validation.group.pre-remove" value="Remove"/>

@Size(groups=Remove.class,min=0,max=0)  
private long vacationDays;

**Creating New Constraints**

New constraint is composed from two parts: the annotation definition and the implementation or validation classes. 3 mandatory elements (message, groups and paylod element [used for passing context specific data])

@Constraint(validatedBy={EvenNumberValidator.class})  
@Target({METHOD, FIELD})  
@Retention(RUNTIME)  
public @interface Even {  
 String message() default "Number must be even";  
 Class<?>[] groups() default {};  
 Class<? extends Payload>[] payload() default {};  
 boolean includeZero() default true;  
}

Constraint implementation class – first type is the constraint annotation and second is the type of value

public class EvenNumberValidator implements ConstraintValidator<Even,Integer> {  
 boolean includesZero;  
 public void initialize(Even constraint) {  
 includesZero = constraint.includeZero();  
 }  
 public boolean isValid(Integer value, ConstraintValidatorContext ctx) {  
 if (value == null)  
 return true;  
 if (value == 0)  
 return includesZero;  
 return value % 2 == 0;  
 }  
}

**JPA validation**

When a @Valid annotation is present on a field or property, the validation process proceeds to the object stored in that field or property. When we annotated embeddabeles validation would be traversed into embeddable object.

@Embedded  
@Valid  
private EmployeeInfo info;

**Enable Validation**

By default validation is enabled but we can enable or disable validation by using validation-mode in persistence.xml. This setting can be configured with three values

* Auto – turn on validation when validation provider is present on the classpath(default)
* CALLBACK – Turn on validation but in case when provider is not available thrown exception
* NONE – turn of validation

We can also configure java.persistence.validation.mode. persistence property. This property may be specified in the Map passed to the createEntityManagerFactory() method and overrides the validation-mode setting if present [string values “AUTO”, “CALLBACK”, “NONE”]

## Concurency

### Refreshing Entity State

It is useful for extended and application managed PC. If entity is not managed => IllegalArgumentException will be thrown. Refresh will discard all changes on the current entity => it will load data from database.

## Locking

### Versioning

The version column is absolute necessity whenever an entity gets modified as a detached entity and merged back into a persistence context. Version can be long, int, short or java.sql.Timestamp. Update of version column can be not supported by all providers. In some cases we have to update seldom this column. Version column would be updated automatically only when either the non-relationship files or the owning FK relationship files( many-to-one, one-to-one) are modified. If we want to cause the version update for the non-owned, collection-valued => use one of the locking mechanism.

Locking options can be specified by means of a number of different calls:

* EM.lock() – in the PC lock the object
* EM.refresh(), find() – permits lock mode to be passed in and applies to the object being returned/refreshed.
* Query.setLockMode() – effect in during the execution of query.

EM calls have to be executed during the transaction

### Optimistic (JPA 1.0 Read)

Perform a version check on locked Entity before commit, throw an OptimisticLockException if Entity version mismatch.

### OPTIMISTIC\_FORCE\_INCREMENT (JPA 1.0 Write)

Perform a version check on locked Entity before commit, throw an OptimisticLockException if Entity version mismatch, force an increment to the version at the end of the transaction, even if the entity is not modified.

### PESIMISTIC

Lock the database row when reading.

### PESSIMISTIC\_FORCE\_INCREMENT

Lock the database row when reading, force an increment to the version at the end of the transaction, even if the entity is not modified.

**Recovering from optimistic failures.**

**Pessimictic locking recovering**

## Caching

### Shared cache

Static configuration of the cache

# JPA CERTIFICATION TEST ISSUES

1. The cascade=REMOVE specification can only be applied to associations that are specified as OneToOne or OneToMany.
2. The lock modes OPTIMISTIC and OPTIMISTIC\_FORCE\_INCREMENT are used for optimistic locking. The lock mode type values READ and WRITE are synonymous with OPTIMISTIC and OPTIMISTIC\_FORCE\_INCREMENT respectively
3. An entity is considered to be loaded if all attributes with FetchType.EAGER—whether explicitly specified or by default—(including relationship and other collection-valued attributes) have been loaded from the database or assigned by the application. Attributes with FetchType.LAZY may or may not have been loaded.
4. The primary key class must define equals and hashCode methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.
5. The primary key must be defined on the entity that is the root of the entity hierarchy or on a mapped superclass of the entity hierarchy. It is not necessary to be a root class or an Entity class.
6. The primary key class must be public and must have a public no-arg constructor.
7. A persistence unit that is defined at the level of the EAR is generally visible to all components in the application. However, if a persistence unit of the same name is defined by an EJB-JAR, WAR, or application jar file within the EAR, the persistence unit of that name defined at EAR level will not be visible to the components defined by that EJB-JAR, WAR, or application jar file unless the persistence unit reference uses the persistence unit name # syntax to specify a path name to disambiguate the reference. When the # syntax is used, the path name is relative to the referencing application component jar file. For example, the syntax ../lib/persistenceUnitRoot.jar#myPersistenceUnit refers to a persistence unit whose name, as specified in the name element of the persistence.xml file, is myPersistenceUnit and for which the relative path name of the root of the persistence unit is ../lib/persistenceUnitRoot.jar. The # syntax may be used with both the unitName annotation element or persistence-unit-name deployment descriptor element to reference a persistence unit defined at EAR level
8. The extended persistence context exists from the point at which the entity manager has been created using EntityManagerFactory.createEntityManager until the entity manager is closed by means of EntityManager.close

The Bean Provider must do one of the following to ensure data integrity before throwing an application exception from an enterprise bean instance:

1. Ensure that the instance is in a state such that a client’s attempt to continue and/or commit the transaction does not result in loss of data integrity. For example, the instance throws an application exception indicating that the value of an input parameter was invalid before the instance performed any database updates.
2. If the application exception is not specified to cause transaction rollback, mark the transaction for rollback using the EJBContext.setRollbackOnly method before throwing the application exception. Marking the transaction for rollback will ensure that the transaction can never commit.

If the enterprise bean method encounters a system-level exception or error that does not allow the method to successfully complete, the method should throw a suitable non-application exception that is compatible with the method’s throws clause. While the EJB specification does not prescribe the exact usage of the exception, it encourages the Bean Provider to follow these guidelines:

* if the bean method encounters a system exception or error, it should simply propagate the error from the bean method to the container (i.e., the bean method does not have to catch the exception).
* If the bean method performs an operation that results in a checked exception that the bean method cannot recover, the bean method should throw the javax.ejb.EJBException that wraps the original exception.
* Any other unexpected error conditions should be reported using the javax.ejb.EJBException.

Note that the javax.ejb.EJBException is a subclass of the java.lang.RuntimeException, and therefore it does not have to be listed in the throws clauses of the business methods.

JPA 2.0 supports the use of a second-level cache by the persistence provider. (the specification supports the use of caching and defines caching annotations as well. However, it does not require the support for caching from the persistence provider.) The second-level cache underlies the persistence context, and is largely transparent to the application.

All exceptions defined in javax.persistence inherit from PersistenceException and that exception inherit from .

NoResultException, NonUniqueResultException, LockTimeoutException, and QueryTimeoutException will cause no rollback.

EntityExistsException, EntityNotFoundException, OptimisticLockException, PessimisticLockException will cause rollback.

The onMessage method of a MDB can only have Required or NotSupported transactions.

Whether the entities and entity-related state of a persistence unit will be cached is determined by the value of the shared-cache-mode element of the persistence.xml file.

The shared-cache-mode element has five possible values: ALL, NONE, ENABLE\_SELECTIVE, DISABLE\_SELECTIVE, UNSPECIFIED.

* A value of ALL causes all entities and entity-related state and data to be cached.
* A value of NONE causes caching to be disabled for the persistence unit. Persistence providers must not cache if NONE is specified.
* The values ENABLE\_SELECTIVE and DISABLE\_SELECTIVE are used in conjunction with the Cacheable annotation (or XML element). The Cacheable annotation specifies whether an entity

should be cached if caching is enabled by the persistence.xml shared-cache-mode element.

The Cacheable element is specified on the entity class. It applies to the given entity and its subclasses unless subsequently overridden by a subclass.

* Cacheable(false) means that the entity and its state must not be cached by the provider.
* A value of ENABLE\_SELECTIVE enables the cache and causes entities for which Cacheable(true) (or its XML equivalent) is specified to be cached. Entities for which Cacheable(true) is not specified or for which Cacheable(false) is specified must not be cached.
* A value of DISABLE\_SELECTIVE enables the cache and causes all entities to be cached except those for which Cacheable(false) is specified. Entities for which Cacheable(false) is specified must not be cached.

If either the shared-cache-mode element is not specified in the persistence.xml file or the value of the shared-cache-mode element is UNSPECIFIED, and the javax.persistence.

sharedCache.mode property is not specified, the behavior is not defined, and provider specific defaults may apply. If the shared-cache-mode element and the javax.persistence.sharedCache.mode property are not specified, the semantics of the Cacheable annotation (and XML equivalent) are undefined.

The persistence provider is not required to support use of a second-level cache. If the persistence provider does not support use of a second-level cache or a second-level cache is not installed, this element will be ignored and no caching will occur.

with BMTD (bean managed transaction demarcation) can only use the UserTransaction interface. See detailed explanation to see the methods of UserTransaction

TypedQuery can have one or more roots. The query must have a Root if you want to navigate through the returned object graph. It must always have at least one Root. Think of the root as the from clause of SQL/JPQL. It specifies the table/abstract schema from which you want to select data. You can't have a query without a from clause.

EJB and JPA => Since isolation levels are implemented differently in various databases, the EJB specification has not standardized its usage. Entities don't manage transactions anyway. Session beans or Message Driven Beans work with transactions.