**JPA Notes**

**Good overview**

http://en.wikibooks.org/wiki/Java\_Persistence

**1) @MappedSuperclass** = No Table exists. Mappings apply only to subclasses.

Egs abstract class HistorizedInfo.

**2) @Inheritance**. Used on the root class of a hierarchy.

Eg abstract class FundAnchor

<http://en.wikibooks.org/wiki/Java_Persistence/Inheritance#Example_single_table_inheritance_table_in_database>

**A single Table Inheritance** = Has ALL the classes of the inheritance hierarchy columns. Single table inheritance is the simplest and typically the best performing and best solution

**Joined, Multiple Table Inheritance** = Joined inheritance is the most logical inheritance solution because it mirrors the object model in the data model. This is what we have with BaseFundAnchor🡪FundAnchor.

**3) @DiscriminatorValue** = Table name. The DiscriminatorValue annotation can only be specified on a concrete entity class

4)**@PostLoad**. CallBack methods. Methods marked with this annotation will be invoked after all **eagerly fetched fields** of your class have been loaded from the datastore.

See <http://openjpa.apache.org/builds/1.0.0/apache-openjpa-1.0.0/docs/manual/jpa_overview_pc_callbacks.html>

See HistorizedInfo class. Method calls resetDirtyFlag() which calls this which which is like a DEEP COPY.

private HistorizedInfo<Anchor> createMemento(HistorizedInfo<Anchor> source) {

HistorizedInfo<Anchor> memento = newInstance();

copyHistorisedData(source, memento);

copyAuditData(source, memento);

copyBusinessData(source, memento);

return memento;

also AOP: Not sure how the below are linked up?

***HistorizedInfo class*** is annotated with

**@FilterDef**(name = "effectiveDate", parameters = @ParamDef(name = "effectiveDate", type = "date"), defaultCondition = "((:effectiveDate >= EFF\_FROM\_DT AND :effectiveDate < EFF\_TO\_DT) OR (:effectiveDate IS " + "NULL AND EFF\_TO\_DT = '9999-12-31'))

[")](mailto:\")And class TemporalFilterAdvice has around @Around(\"@annotation(bdsl.persistence.hods.services.common.aop.TemporalFilter)\"))

[And class](mailto:\")And class TemporalFilterAdvice has around @Around(\"@annotation(bdsl.persistence.hods.services.common.aop.TemporalFilter)\")) ***[TemporalFilterAdvice](mailto:\")And class TemporalFilterAdvice has around @Around(\"@annotation(bdsl.persistence.hods.services.common.aop.TemporalFilter)\"))*** [has around](mailto:\")And class TemporalFilterAdvice has around @Around(\"@annotation(bdsl.persistence.hods.services.common.aop.TemporalFilter)\")) **[@Around](mailto:\")And class TemporalFilterAdvice has around @Around(\"@annotation(bdsl.persistence.hods.services.common.aop.TemporalFilter)\"))**[("@annotation(bdsl.persistence.hods.services.common.aop.TemporalFilter)")](mailto:\")And class TemporalFilterAdvice has around @Around(\"@annotation(bdsl.persistence.hods.services.common.aop.TemporalFilter)\"))

***BaseFundClass*** has find(String code, *@EffectiveDate* Date effectiveDate) which might be the trigger for the aop?

**@TemporalFilter** on BaseFundService and ClientFundService classes.

# Pro JPA 1.2 Book

## Chapter 2 : Getting Started

P25: Looks like **Service classes** are used to access / operate on Entity classes. The entity glasses are pure POJO. N.B. M&G have services composed of spring Repos.

## Chapter 3 : Enterprise Apps

P32: Stateless session bean lifecycle **Calbacks**: In-container means during construction not all resources may be available so can use **@PostConstruct** etc..

P36: Stateless, stateful and singleton session beans. Not the same as servlet sessions (http).

**Singleton**: Once created, it will continue to exist until the container removes it, regardless of any exceptions that occur during business method execution. This is a key difference from other session bean types because the bean instance will never be re-created in the event of a system exception.

P38: DI in Java EE: A reference is declared using one of the resource reference annotations: Resource, @EJB, @PersistenceContext, or @PersistenceUnit. The context is an environment naming context. Which is a replacement for using jndi (cumbersome). Also Setter injection is better than field injection (as field can remain private).

P42: @PersistenceContext: to acquire an entity manager through dependency injection into a stateless session bean.

@PersistenceUnit: The EntityManagerFactory for a persistence unit can be referenced by this.

P46: **@Qualifier:** Helps resolve which instance of a bean to inject.

@Target({METHOD, FIELD, PARAMETER, TYPE})

@Retention(RUNTIME)

public @interface Secure { }

p47: @Produces: Is like a factory method.

P49: DataSource tx is resource-local transaction and is equivalent to a database transaction. The broader container transaction uses the Java Transaction API (JTA) and can span multiple resource tx’s.

Java 7 @Transactional: Transactional **interceptor**.

P55: Session Façade DP: which a stateless session bean is used to shield clients from dealing with a particular persistence API.

## Chapter 4 : ORM

P60: Annotating field or getter/setter: If you annotate fields, the provider will get and set the fields of the entity using **reflection**. If the annotations are set on the getter methods of properties, those getter and setter methods will be invoked by the provider to access and set the state. M&G use field (although marked as private and there are getter/setters?). Non-annotated fields WILL also be persisted by default.

P64: An optional @Basic annotation can be placed on a field or property to explicitly mark it as being persistent. This annotation is mostly for documentation purposes and is not required for the field or property to be persistent.

P65: @Column specifies a physical column as opposed to @Basic. @Table: Only required if not using default name of @Entity.

P66: @Basic(fetch=**FetchType.LAZY**). This column loading may be deferred until it is referenced. This is only a hint and will be loaded as soon as column is referenced. *It is almost never a good idea to lazily fetch simple types.*

P67: @Lob indicates large object column (used by jdbc special calls). The Java types mapped to BLOB columns are byte[], Byte[], and Serializable types, while char[], Character[], and String objects are mapped to CLOB columns.

P69: To store enum values use @Enumerated(EnumType.STRING) will store name. can store ordinals but difficult with order changes.

Using java.util.Date and java.util.Calendar requires a @Temporal(TemporalType.DATE) to clarify mapping type. The java.sql are handled automatically.

P72: ID generating. The **AUTO** mode is really a generation strategy for **development or prototyping**. Use SEQ or Identity if supported.

P77: In some cases, unidirectional relationships in the object model can pose a problem in the database model. Source (Employee) \* 🡪 1 Target (Department). NB Source is many side. Ordinality refers to whether a relationship need exist (ie FK can be null).

P79. Mappings: ie the Ordinality on each side of rel \*🡪1 and 1🡪1 where TARGET = 1. The \* has the FK.

P80 Join Column – Note how Join column is separate to **@ManyToOne** ordinality. In almost every relationship, independent of source and target sides, one of the two sides will have the join column in its table. That side is called the owning side or the owner of the relationship. The side that does not have the join column is called the non-owning or inverse side. **So the FK table is the OWNER** and **@JoinColumn is always specified on the owner table.**

P83: Bi-directional 1🡪1 mapping. Either side can be the owner, but likely be decided based on the most frequent direction of traversal. Must add **@OneToOne**(mappedBy="parkingSpace") private Employee employee; to indicate Employee is owner. **NB Both classes contain ref vars to each other**.

P84: @OneToMany: Defines a Collection on 1 side. LIKE an AGGREGATION or parent. (Many is the owning side) as a single row on the 1 side cannot contain all the keys to the many.

@OneToMany(mappedBy="department") . One side is NOT the owner, but contains ref to \*.

private Collection<Employee> employees;

Q. I wonder why bidirectional are so prevalent as the object refs need to be stored on both sides?

P86: @ManyToMany (Collections on both sides), has definition on both sides. MUST pick one size to be owner (other side must have mappedBy). Requires a @JoinTable.

P89: **Collection**-valued relationships default to be **lazily loaded**. In bidirectional relationship cases, the fetch mode might be lazy on one side but eager on the other. This kind of configuration is actually quite common because relationships are often accessed in different ways depending on the direction from which navigation occurs.

P90: @Embeddable and @Embedded. An **embedded object** is one that is dependent on an entity for its identity. Eg ADDRESS of an EMPLOYEE. UML notation showing ownership: Emp<>----🡪Addr.

NB. It is not portable to define embedded objects as part of inheritance hierarchies