spring

HIBERNATE

SUNBEAM

Trainer: Nilesh Ghule

Wake up from Hibernate, Spring up!!!

Agenda

- HQL Joins
- Hibernate Inheritance
  - @MappedSuperclass
  - @Inheritance
- Forward engineering vs Reverse engineering
- Using Hibernate in Java EE application
- Hibernate caching
  - SessionFactory cache

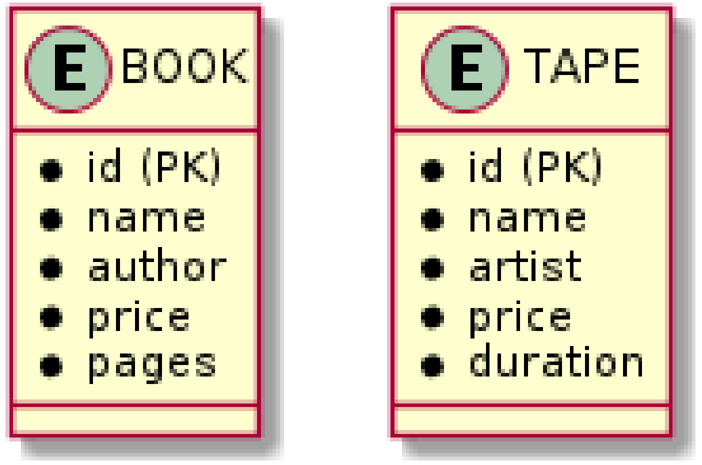
HQL Join

- SQL joins are used to select data from two related tables.
- Types of joins: Cross, Inner, Left outer, Right outer, Full outer.
- SQL Join syntax:
  - SELECT e.ename, d.dname FROM EMP e INNER JOIN DEPT d ON e.deptno = d.deptno;
  - SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;
- HQL joins are little different than SQL joins. They can be used on association so that condition will be generated automatically. Alternatively property of composite object can be given in select or may use traditional join syntax.
- HQL Join syntax:
  - SELECT e.ename, d.dname FROM EMP e INNER JOIN e.dept d;
  - SELECT e.ename, e.dept.dname FROM EMP e;
  - SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;
- Note that joins are executed automatically when Hibernate associations are used.

Hibernate Relations (inheritance)

- @MappedSuperclass
  - Different tables for sub-class entities.
  - Easy to implement.
  - Super class is not entity.
  - No polymorphic queries.
  - Relations/mapping in super class are applicable only to sub-class table (as super-class doesn't have table).

```
@MappedSuperclass
class Product {
   @Id
   @Column private int id;
   @Column private String name;
   @Column private double price;
    // ...
}
```



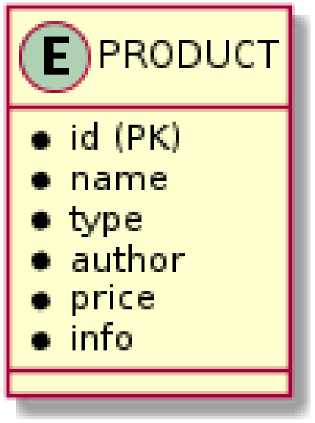
```
@Entity
@Table(name="BOOK")
class Book extends Product {
    @Column private String author;
    @Column private int pages;
    // ...
}
```

```
@Entity
@Table(name="TAPE")
class Tape extends Product {
    @Column private String artist;
    @Column private int duration;
     // ...
}
```

Hibernate Inheritance – SINGLE_TABLE strategy

- SINGLE_TABLE
  - Single table for all sub-class entities.
  - Easy to implement.
  - All classes including super class are entity classes.
  - Polymorphic queries, High performance.
  - Relations in super class..

```
@Entity
@Table(name="PRODUCT")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE")
class Product {
    @Id
    @Column private int id;
    @Column private String name;
    @Column private double price;
    // ...
}
```



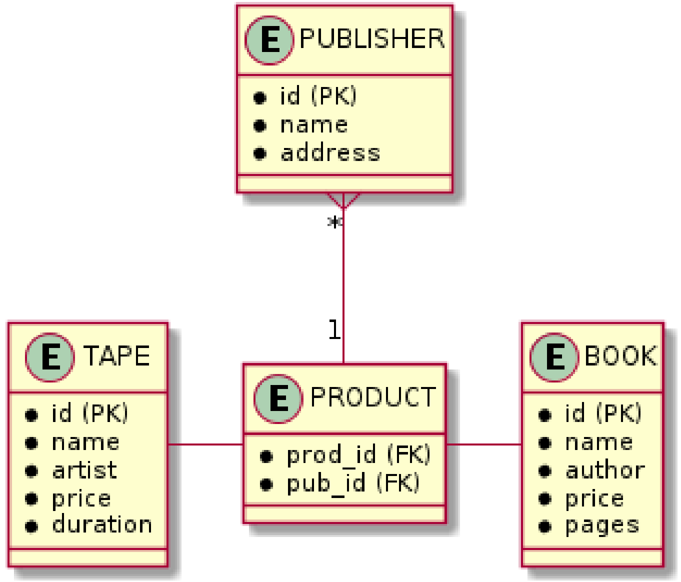
```
@Entity
@DiscriminatorValue("BOOK")
class Book extends Product {
    // …
    @Column(name="author")
    private String author;
    @Column(name="info")
    private int pages;
}
```

```
@Entity
@DiscriminatorValue("TAPE")
class Tape extends Product {
    // …
    @Column(name="author")
    private String artist;
    @Column(name="info")
    private int duration;
}
```

Hibernate Inheritance – TABLE_PER_CLASS strategy

- TABLE_PER_CLASS
  - Different table for each sub-class & separate table for super-class. Super-class table is very small.
  - All classes including super class are entity classes.
  - Joins are fired to fetch data of sub-class entities.
  - Polymorphic queries, less performance, complex join queries.
  - Relations in super class.

```
@Entity
@Table(name="PUBLISHER")
class Publisher {
    @Id
    @Column private int id;
    @Column private String name;
    @Column private String address;
    @OneToMany(mappedBy="pubId")
    private List<Product> productList;
}
```



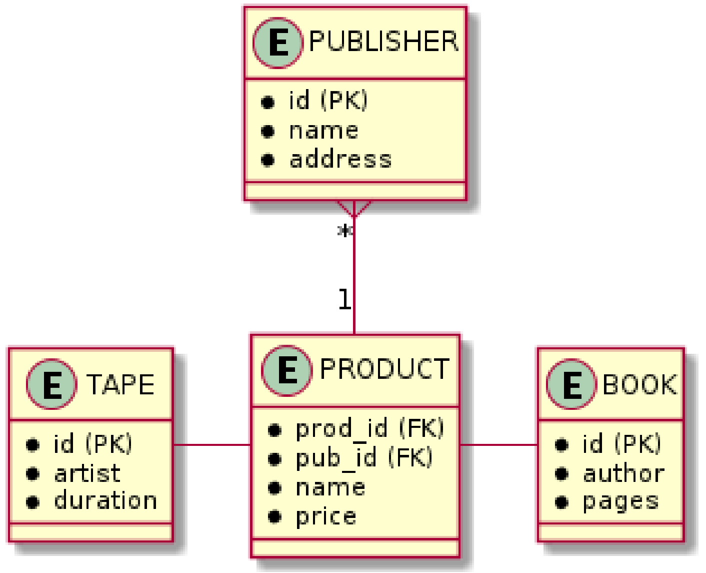
```
@Entity
@Table(name="PRODUCT")
@Inheritance(strategy = TABLE_PER_CLASS)
class Product {
    @Id
    @Column(name="prod_id") private int id;
    @Column(name="pub_id") private int pubId;
    @Column private String name;
    @Column private double price;
}
```

```
@Entity @Table(name="BOOK")
class Book extends Product {
    @Column private String author;
    @Column private int pages;
}
@Entity @Table(name="TAPE")
class Tape extends Product {
    @Column private String artist;
    @Column private int duration;
}
```

Hibernate Inheritance – JOINED strategy

- JOINED
  - Different table for each sub-class & separate table for super-class. Super-class table contains common data. Sub-class tables contains only specific data.
  - All classes including super class are entity classes.
  - Joins are fired to fetch data of sub-class entities.
  - Polymorphic queries, Join queries.
  - Relations in super class.

```
@Entity
@Table(name="PUBLISHER")
class Publisher {
    @Id
    @Column private int id;
    @Column private String name;
    @Column private String address;
    @OneToMany(mappedBy="pubId")
    private List<Product> productList;
}
```



```
@Entity
@Table(name="PRODUCT")
@Inheritance(strategy = JOINED)
class Product {
    @Id
    @Column(name="prod_id") private int id;
    @Column(name="pub_id") private int pubId;
    @Column private String name;
    @Column private double price;
}
```

```
@Entity @Table(name="BOOK")
class Book extends Product {
    @Column private String author;
    @Column private int pages;
}
@Entity @Table(name="TAPE")
class Tape extends Product {
    @Column private String artist;
    @Column private int duration;
}
```

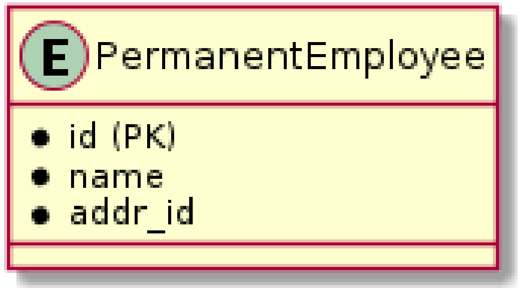
@AssociationOverride

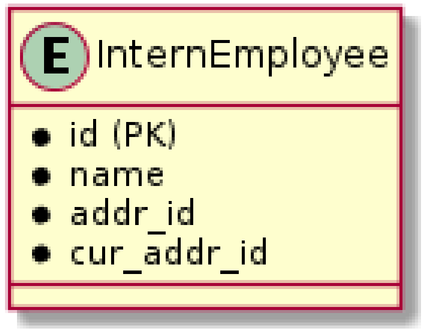
- Used to override a mapping for an entity relationship.
- May be applied to an entity that extends a mapped superclass to override a relationship mapping defined by the mapped superclass.
- If not specified, the association is mapped the same as in the original mapping. When used to override a mapping defined by a mapped superclass, AssociationOverride is applied to the entity class.
- May be used to override a relationship mapping from an embeddable within an entity to another entity when the embeddable is on the owning side of the relationship.
- When used to override a relationship mapping defined by an embeddable class (including an embeddable class embedded within another embeddable class),
- AssociationOverride is applied to the field or property containing the embeddable.

@AssociationOverride

- The default mapping of address (with addr_id column) has been changed into inherited class InternEmployee.
- InternEmployee considers its current address (cur_addr_id) over its permanent address (addr_id).

```
@MappedSuperclass
public class Employee {
    @OneToOne
    @JoinColumn(name="addr_id")
    protected Address address;
    // …
}
@Entity
public class PermanentEmployee extends Employee {
    // inherited "address" refer to addr_id (Permanent address)
}
@Entity
@AssociationOverride(name="address",
joincolumns=@JoinColumn(name="cur_addr_id"))
public class InternEmployee extends Employee {
    // inherited "address" refer to cur_addr_id (Temporary address)
}
```

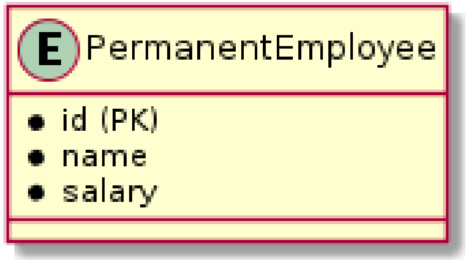


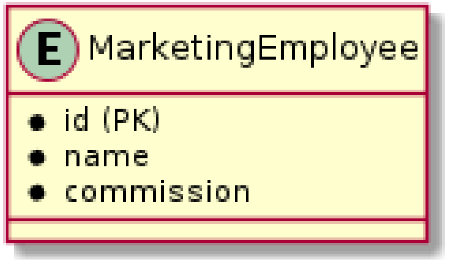


@AttributeOverride

- Used to override the mapping of a basic (whether explicit or default) property or field or property.
- May be applied to an entity that extends a mapped superclass or to an embedded field or property to override a basic mapping or id mapping defined by the mapped superclass or embeddable class (or embeddable class of one of its attributes).

```
@MappedSuperclass
public class Employee {
    protected double salary;
}
@Entity
public class PermanentEmployee extends Employee {
    // inherited "salary" refer to salary column
}
@Entity
@AttributeOverride(name="salary",
column=@Column(name="commission"))
public class MarketingEmployee extends Employee {
    // inherited "salary" refer to commission column
}
```





Forward and Reverse Engineering

- There are two approaches
  - Database first approach (reverse engg)
  - Entity first approach (forward engg)
- Reverse engg
  - 1. Design database and create tables with appropriate relations.
  - 2. Implement entity classes with appropriate ORM annotations
- Two ways to implement reverse engg
  - Manual approach
  - Wizard/tools (Eclipse -> JPA project)

Forward and Reverse Engineering

- Forward engg
  - 1. Implement entity classes with appropriate ORM annotations.
  - 2. Database tables (with corresponding relations) will be automatically created when prorgram is executed.
- Hibernate.cfg.xml – hibernate.hbm2ddl.auto
  - create: Db dropping will be generated followed by database creation.
  - create-only: Db creation will be generated.
  - create-drop: Drop the schema and recreate it on SessionFactory startup. Additionally, drop the schema on SessionFactory shutdown.
  - update: Update the database schema.
  - validate: Validate the database schema
  - none: No action will be performed.

Using hibernate in Java EE application

- While dealing with single database, whole Java EE application should have single SessionFactory instance.
- This can be created as Singleton object in ServletContext.
- For each request new thread gets created.
- Hibernate session can be attached to thread session context, so that it can be accessed from any component.
- ORM is done using annotations on Entity objects and DAO layer is to be written using Hibernate (instead of JDBC).

Online Bookshop application

JNDI

- JNDI is Java API used to discover and lookup resources by name in form of Java object.
- JNDI is independent of underlying implementation.
- JNDI is commonly used for RMI, JDBC (Web server), EJB and JMS (App server).
- JNDI names are in hierarchical fashion.
- Name is bound and looked up in some context.
- Typical example is web server bind some name with the JDBC connection pool and it can be looked up from the web applications.

Hibernate caching

- Hibernate caches are used to speed up execution of the program by storing data (objects) in memory and hence save time to fetch it from database repeatedly.
- There are two caches
  - Session cache (L1 cache)
  - SessionFactory cache (L2 cache)

Hibernate session cache

- Collection of entities per session – persistent objects.
- Hibernate keep track of state of entity objects and update into database.
- Session cache cannot be disabled.
- If object is present in session cache, it is not searched into session factory cache or database.
- Use refresh() to re-select data from the database forcibly.

Hibernate session factory cache

- Collection of entities per session factory (usually single in appln).
- By default disabled, but can be enabled and configured into hibernate.cfg.xml
- Need to add respective additional second cache jars in project and optional cache config file (e.g. ehcache.xml).

```
<property name="hibernate.cache.region.factory_class">
        org.hibernate.cache.ehcache.EhCacheRegionFactory
</property>
<property name="hibernate.cache.use_second_level_cache"> true </property>
```

Hibernate session factory cache

- Four types of second level caches: EHCache, Swarm Cache, OS Cache, Tree Cache
- Use @Cache on entity class to cache its objects.
- Decide cache policy for those object and specify into its usage attribute: READ_ONLY, READ_WRITE, NONSTRICT_READWRITE, TRANSACTIONAL
- Stores the objects into the map (with its id as key). So lookup by key is very fast.

Hibernate session factory cache

- Cache Strategies (enum CacheConcurrencyStrategy)
    - READ_ONLY: Used only for entities that never change (exception is thrown if an attempt to update such an entity is made). It is very simple and performant. Very suitable for some static reference data that don’t change.
    - NONSTRICT_READ_WRITE: Cache is updated after a transaction that changed the affected data has been committed. Thus, strong consistency is not guaranteed and there is a small time window in which stale data may be obtained from cache. This kind of strategy is suitable for use cases that can tolerate eventual consistency.
    - READ_WRITE: This strategy guarantees strong consistency which it achieves by using ‘soft’ locks: When a cached entity is updated, a soft lock is stored in the cache for that entity as well, which is released after the transaction is committed. All concurrent transactions that access soft-locked entries will fetch the corresponding data directly from database
    - TRANSACTIONAL: Cache changes are done in distributed XA transactions. A change in a cached entity is either committed or rolled back in both database and cache in the same XA transaction.

Hibernate session factory cache

- Cache Types
  - EHCache: It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.
  - Swarm Cache: A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache.
  - OSCache (Open Symphony Cache): Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support.
  - JBoss Tree Cache: A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking.
  - Note that not all cache support all strategies.

Hibernate session factory cache

Cache	Read Only	NonStrict Read-Write	Read-Write	Transactional
EH Cache	Yes	Yes	Yes	No
OS Cache	Yes	Yes	Yes	No
Swarm Cache	Yes	Yes	No	No
Jboss Cache	Yes	No	No	Yes

SUNBEAM

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>