Hibernate Mapping and Inheritance annotations

To create SessionFactory

To create session

- openSession
- getCurrentSession

openSession and getCurrentSession.

- openSession

SessionFactory.openSession
- it always create new Session object and give it to you.
- You need to explicitly flush and close these session objects.
- session objects are not thread safe, you need to create one session object per request in multithreaded environment and one session per request in web applications too.

- getCurrentSession

SessionFactory. getCurrentSession
- it will provide you session object which is in hibernate context and managed by hibernate internally. It is bound to transaction scope.
- When you call SessionFactory. getCurrentSession , it creates a new Session if not exists , else use same session which is in current hibernate context. It automatically flush and close session when transaction ends, so you do not need to do externally.
- If you are using hibernate in single threaded environment , you can use getCurrentSession, as it is faster in performance as compare to creating new session each time.

- You need to add following property to hibernate.cfg.xml to use getCurrentSession method.

```
<session-factory>

<!-- Put other elements here -->

<property name="hibernate.current_session_context_class">

    thread

</property>

</session-factory>
```

- If you do not configure above property, you will get error as below.

```
Exception in thread "main" org.hibernate.HibernateException: No CurrentSessionContext
configured!
    at
org.hibernate.internal.SessionFactoryImpl.getCurrentSession(SessionFactoryImpl.java:1012)
        at com.arpit.java2blog.hibernate.HibernateSessionExamp
```

@Entity Annotation

```java
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEETab")
public class Employee {
  @Id
@GeneratedValue
 //@GeneratedValue(strategy=GenerationType.AUTO)
  @Column(name = "id")
  private int id;
 @Column(name = "name")
  private String Name;
    // Constructor and setter getter methods
}
```

@Column Annotation

- The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes –

- name attribute permits the name of the column to be explicitly specified.

- length attribute permits the size of the column used to map a value particularly for a String value.

- nullable attribute permits the column to be marked NOT NULL when the schema is generated.

- unique attribute permits the column to be marked as containing only unique values.

Session interface methods (CRUD operations)

- save(Object object) - save() method persist the given transient instance, first assigning a generated identifier. (Or using the current value of the identifier property if the assigned generator is used.) This operation cascades to associated instances if the association is mapped with cascade="save-update".

- saveOrUpdate(Object object) - This method either save(Object) or update(Object) the given instance, depending upon the resolution of the unsaved-value checks (see the manual for a discussion of unsaved-value checking).

- delete(Object object) - Remove a persistent instance from the datastore.

- get() - This method returns a persistence object of the given class with the given identifier. It will return null if there is no persistence object.

- Evict()- To detach the object from session cache,
  - After detaching the object from the session, any change to object will not be persisted.
 - The associated objects will also be detached if the association is mapped with cascade="evict".

    evict(Object object): Accept the object which is already synched with session.

- merge()
    - It is used to merge an object with persistent object on the basis of same identifier.
    - The object as an argument is not changed and method returns persistent object.
    - Associated objects are also merged if cascade type is as cascade="merge".

- 1. The state of object passed as an argument is copied to the object in the hibernate session with same identifier and return the persistent object.

- 2. If the object is not already present in the session with the same identifier as passed in the argument, first the session loads the object for the identifier of object passed as an argument then merge it and return the persistent object of the session.

- 3. If the persistent object for the identifier of object passed in argument, is not already in session and database, then a copy of object passed as an argument is persisted and is returned.

- merge(Object object) : Accepts the entity object and returns persistent object.

- merge(String entityName, Object object) : Pass entity name and entity object.
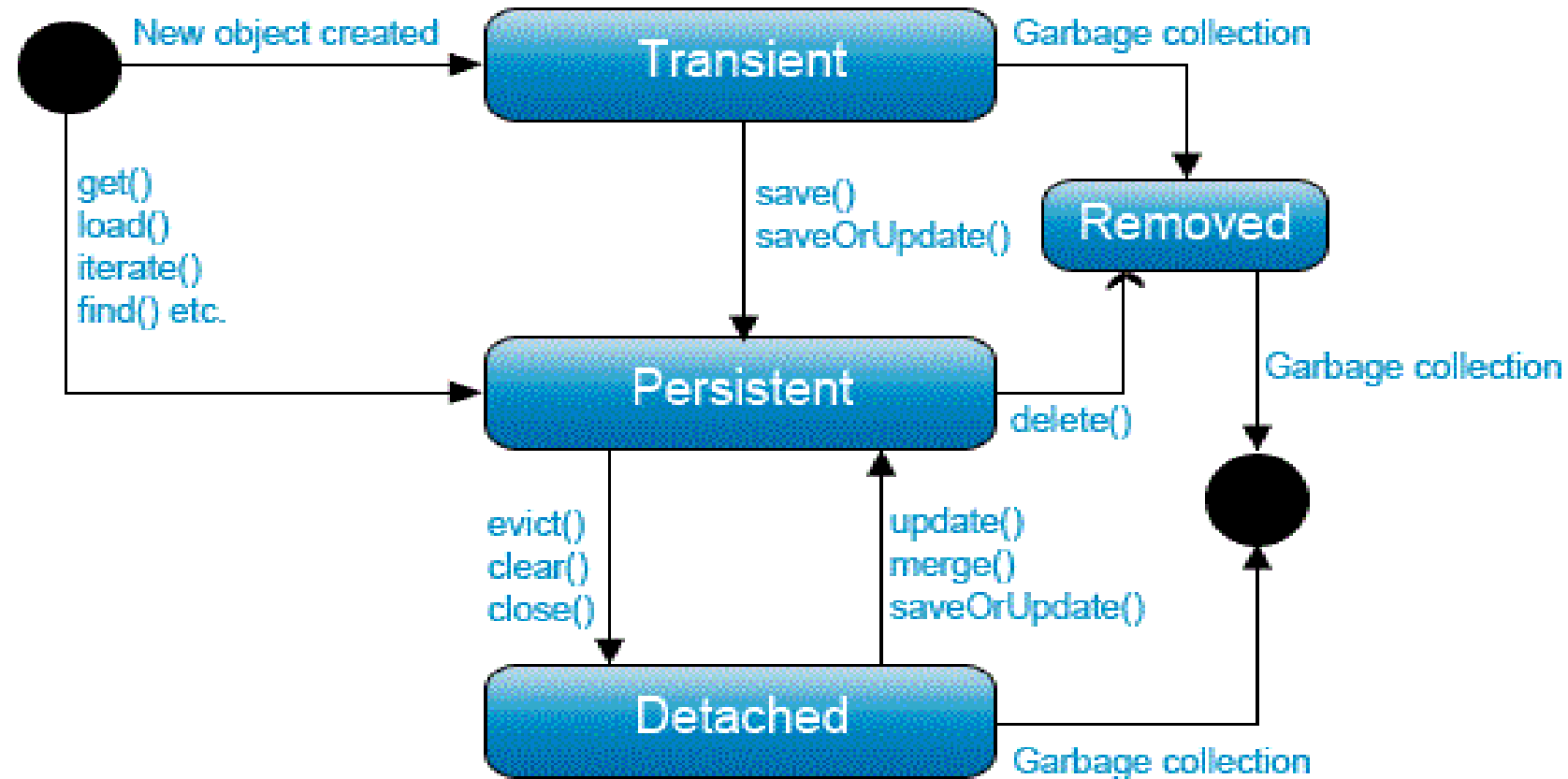
Difference between evict and delete methods ?

- evict() deletes the data from the session only,

- but delete() deletes the data from both the session as well as DataBase.

Update and merge
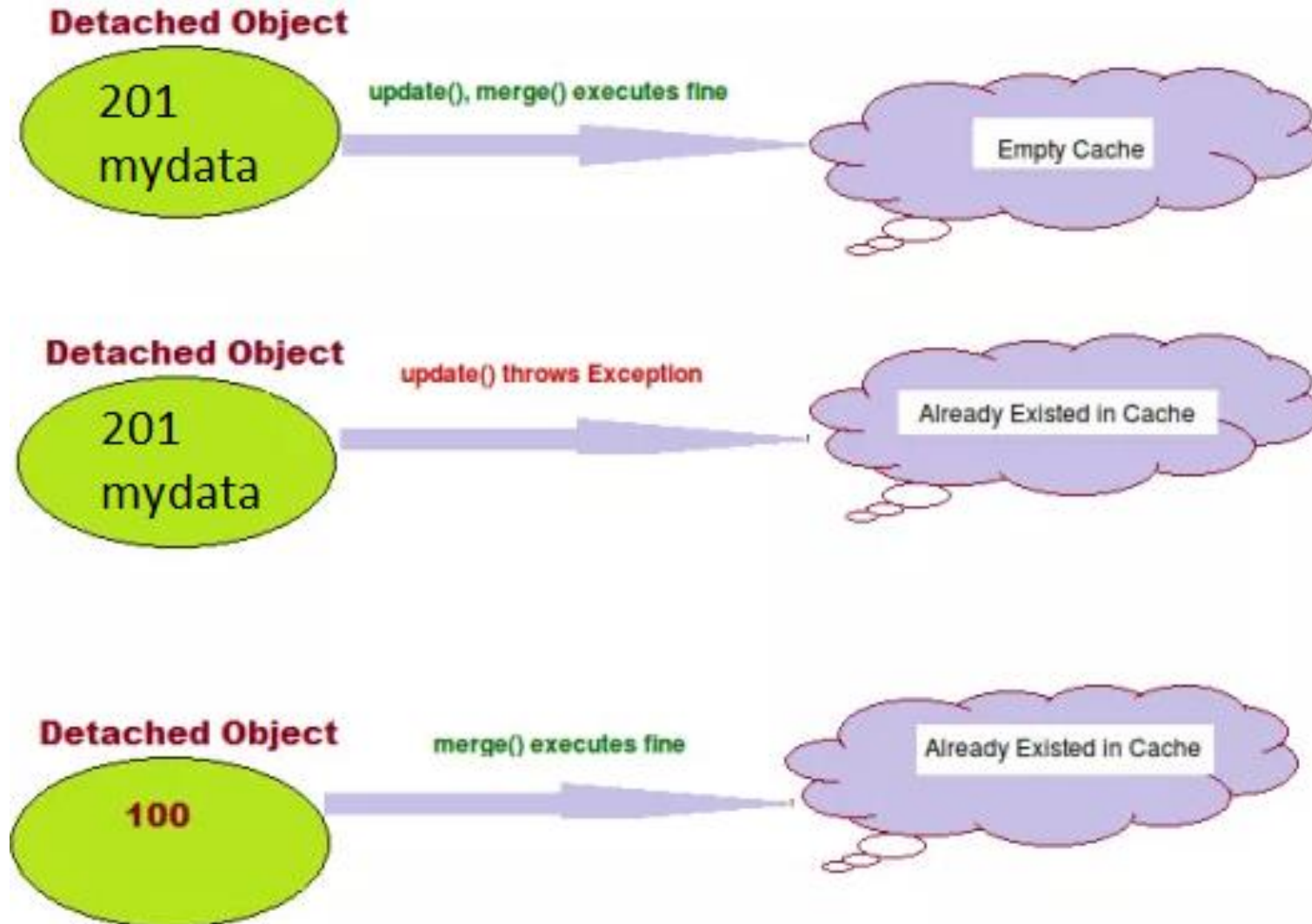
Object Lifecycle

Relations in Hibernate and default FetchType

- OneToMany: LAZY
- ManyToOne: EAGER
- ManyToMany: LAZY
- OneToOne: EAGER

Association

- In this type of association, we will define the association as below using @OneToOne annotation.

- Foreign key column will get added in both tables to avoid that use mappedBy

    //Inside EmployeeEntity.java
    @OneToOne
    AccountEntity   account;

    //Inside AccountEntity.java
    @OneToOne
    EmployeeEntity   employee;

- With above association, both ends are managing the association so both must be updated with information of each other using setter methods defined in entity java files. If you fail to do so, you will not be able to fetch the associated entity information and it will be returned as null.

OneToOne Mapping
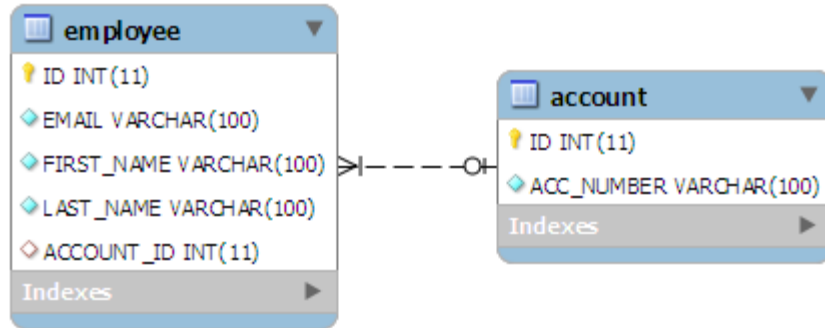
Association is managed by EmployeeEntity
So account_id will be added in EmployeEntity class

```
EmployeeEntity.java
@OneToOne
@JoinColumn(name="ACCOUNT_ID")
private AccountEntity account;
```
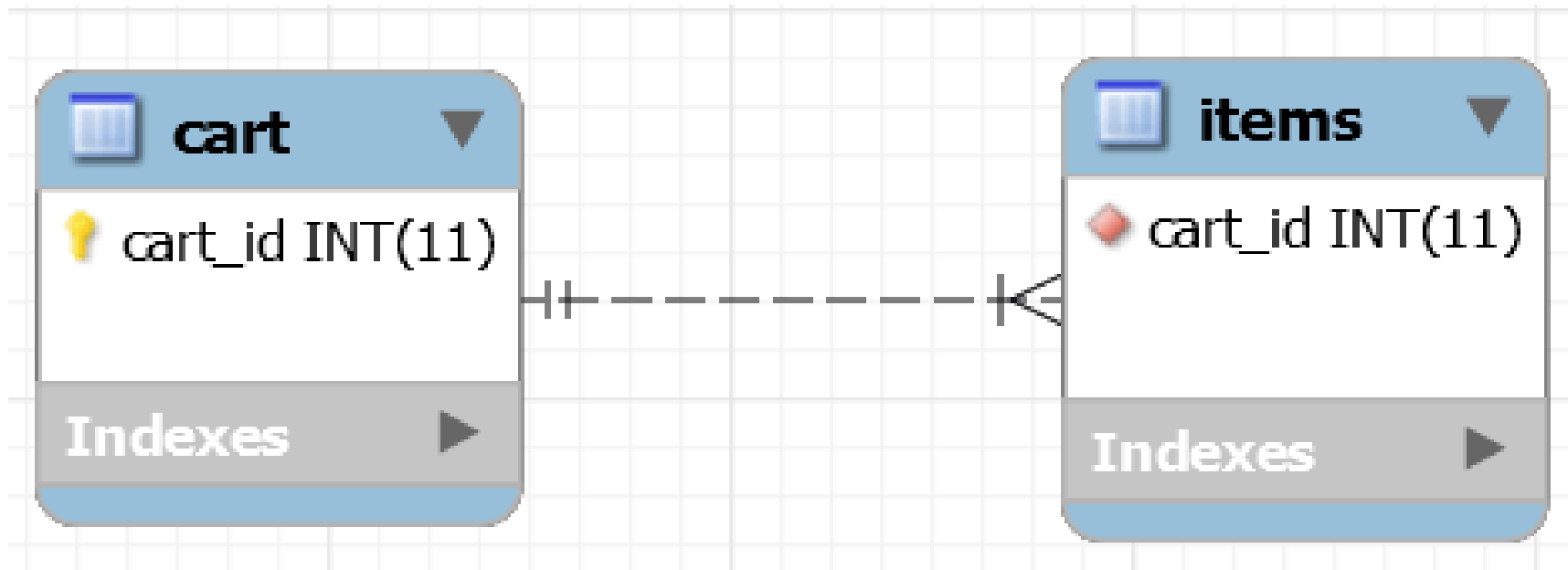
```
AccountEntity.java
@OneToOne(mappedBy="account")
private EmployeeEntity employee;
```

OneToMany

```java
@Entity
@Table(name="CART")
public class Cart {

    //...

    @OneToMany(mappedBy="cart")
    private Set<Items> items=new HashSet<>();

    // getters and setters
}
```

- also add a reference to Cart in Items using @ManyToOne, making this a bidirectional relationship. Bidirectional means that we are able to access items from carts, and also carts from items.

- note that the @OneToMany annotation is used to define the property in Items class that will be used to map the mappedBy variable. That's why we have a property named "cart" in the Items class:

@Entity

@Table(name="ITEMS")

public class Items {

  //...

  @ManyToOne

  @JoinColumn(name="cart_id", nullable=false)

  private Cart cart;

  public Items() {}

  // getters and setters

}

- It is important to note that the @ManyToOne annotation is associated with Cart class variable. @JoinColumn annotation references the mapped column

ManyToMany relation
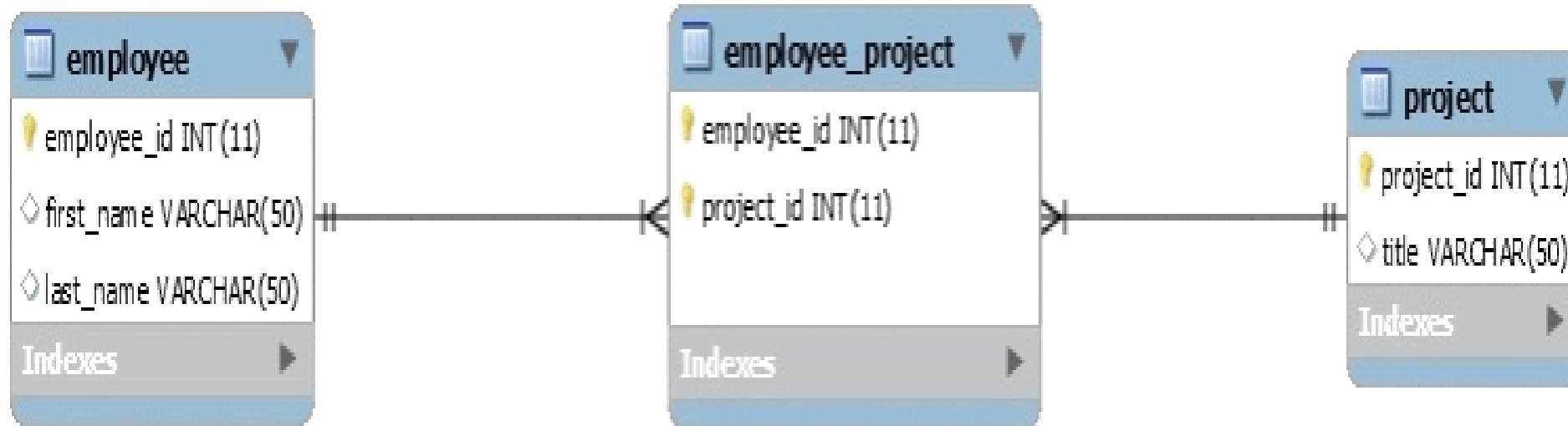
many-to-many association

- use @ManyToMany, @JoinTable and @JoinColumn annotations
- **This association has two sides i.e. the owning side and the inverse side.**
- The owning side is *Employee* so the join table is specified on the owning side by using the *@JoinTable* annotation in *Employee* class. The *@JoinTable* is used to define the join/link table. In this case, it is *Employee_Project.*

```java
@Entity
@Table(name = "Employee")
public class Employee {
    // ...

    @ManyToMany(cascade = { CascadeType.ALL })
    @JoinTable(
        name = "Employee_Project",
        joinColumns = { @JoinColumn(name = "employee_id") },
        inverseJoinColumns = { @JoinColumn(name = "project_id") }
    )
    Set<Project> projects = new HashSet<>();

    // standard constructor/getters/setters
}
```

```java
@Entity
@Table(name = "Project")
public class Project {
    // ...

    @ManyToMany(mappedBy = "projects")
    private Set<Employee> employees = new HashSet<>();

    // standard constructors/getters/setters
}
```

Inheritance Annotation

- To address this, the JPA specification provides several strategies:

- Single Table – the entities from different classes with a common ancestor are placed in a single table

- Joined Table – each class has its table and querying a subclass entity requires joining the tables

- Table-Per-Class – all the properties of a class, are in its table, so no join is required

Single table for hirarchy

- To customize the discriminator column, we can use the @DiscriminatorColumn annotation:

- @Entity(name="products")
- @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
- @DiscriminatorColumn(name="product_type",
- discriminatorType = DiscriminatorType.INTEGER)
- public class MyProduct {
-     // …
- }
- Here we've chosen to differentiate MyProduct sub-class entities by an integer column called product_type.

- To tell Hibernate what value each sub-class record will have for the product_type column:

```java
@Entity
@DiscriminatorValue("1")
public class Book extends MyProduct {
    // ...
@Entity
@DiscriminatorValue("2")
public class Pen extends MyProduct {
    // ...
}
```

Joined Table

- each class in the hierarchy is mapped to its table. The only column which repeatedly appears in all the tables is the identifier, which will be used for joining them when needed.

- super-class that uses this strategy:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
    @Id
    private long animalId;
    private String species;

    // constructor, getters, setters
}
```

- define a sub-class:

```java
public class Pet extends Animal {
    private String name;

    // constructor, getters, setters
}
```

- Both tables will have an animalId identifier column. The primary key of the Pet entity also has a foreign key constraint to the primary key of its parent entity. To customize this column, we can add the @PrimaryKeyJoinColumn annotation:

```java
@Entity
@PrimaryKeyJoinColumn(name = "petId")
public class Pet extends Animal {
    // ...
}
```

Table Per Class

- The Table Per Class strategy maps each entity to its table which contains all the properties of the entity, including the ones inherited.

- The resulting schema, table per class will define entities for parent classes, allowing associations and polymorphic queries as a result.

- To use this strategy, we only need to add the @Inheritance annotation to the base class:

@Entity

@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

public class Vehicle {

  @Id

  private long vehicleId;

   private String manufacturer;

   // standard constructor, getters, setters

}

Inheritance

- Inheritance is one of the key concepts in Java, and it's used in most domain models. That often becomes an issue, if you try to map these models to a relational database. SQL doesn't support this kind of relationship and Hibernate, or any other JPA implementation has to map it to a supported concept.

- You can choose between 4 strategies which map the inheritance structure of your domain model to different table structures. Each of these strategies has its advantages and disadvantages. It's, therefore, important to understand the different concepts and to choose the one that fits best.

- simple domain model in all of the examples to show you the different inheritance strategies. It consists of an author who has written different kinds of publications. A publication can either be a book or a blog post. Both of them share most of their attributes, like the id, a title, and a publishing date. In addition to the shared attributes, the book also stores the number of pages, and the blog post persists its URL.

Author

- #- final long serialVersionUID
- #- Long id
- #- int version
- #- String firstName
- #- String lastName

- ●+Long getId()
- ●+void setId(Long id)
- ●+int getVersion()
- ●+boolean equals(Object obj)
- ●+int hashCode()
- ●+String getFirstName()
- ●+void setFirstName(String firstName)
- ●+String getLastName()
- ●+void setLastName(String lastName)
- ●+String toString()

Publication

- #Long id
- #String title
- #- int version
- #- Set<Author> authors
- #- Date publishingDate

- ●+Long getId()
- ●+void setId(Long id)
- ●+String getTitle()
- ●+void setTitle(String title)
- ●+int getVersion()
- ●+void setVersion(int version)
- ●+Set<Author> getAuthors()
- ●+void setAuthors(Set<Author> authors)
- ●+Date getPublishingDate()
- ●+void setPublishingDate(Date publishingDate)
- ●+String toString()
- ●+int hashCode()

Book

- #- int pages

- ●+int getPages()
- ●+void setPages(int pages)
- ●+boolean equals(Object obj)

BlogPost

- #- String url

- ●+String getUrl()
- ●+void setUrl(String url)

0..* authors

is

is

```java
@MappedSuperclass
public abstract class Publication {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", updatable = false, nullable = false)
    protected Long id;

    @Column
    protected String title;
```

```java
@Version
        @Column(name = "version")
        private int version;

        @Column
        @Temporal(TemporalType.DATE)
        private Date publishingDate;

        …
}
```

List books = em.createQuery("SELECT b FROM Book b", Book.class).getResultList();

- The Book entity and all its attributes are mapped to the book table. This makes the generated query simple and efficient. It just has to select all columns of the book table.

- Table per Class

- The table per class strategy is similar to the mapped superclass strategy. The main difference is that the superclass is now also an entity. Each of the concrete classes gets still mapped to its own database table. This mapping allows you to use polymorphic queries and to define relationships to the superclass. But the table structure adds a lot of complexity to polymorphic queries, and you should, therefore, avoid them.

public

author

🔑 *id*	*bigint*	« pk »
○ firstname	*character varying(255)*	
○ lastname	*character varying(255)*	
○ *version*	*integer*	« nn »

publicationauthor

🔑 *publicationid*	*bigint*	« pk »
🔑 *authorid*	*bigint*	« pk fk »

1 ———◇——— n

rel_publicationauthor_author

book

🔑 *id*	*bigint*	« pk »
○ publishingdate	*date*	
○ title	*character varying(255)*	
○ version	*integer*	
○ pages	*integer*	

blogpost

🔑 *id*	*bigint*	« pk »
○ publishingdate	*date*	
○ title	*character varying(255)*	
○ version	*integer*	
○ url	*character varying(255)*	

- The definition of the superclass with the table per class strategy looks similar to any other entity definition. You annotate the class with @Entity and add your mapping annotations to the attributes. The only difference is the additional @Inheritance annotation which you have to add to the class to define the inheritance strategy. In this case, it's the InheritanceType.TABLE_PER_CLASS.

- @Entity
- @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
- public abstract class Publication {

- @Id
- @GeneratedValue(strategy = GenerationType.AUTO)
- @Column(name = "id", updatable = false, nullable = false)
- protected Long id;

- @Column
- protected String title;

- @Version
- @Column(name = "version")
- private int version;

-

- @ManyToMany

-     @JoinTable(name = "PublicationAuthor", joinColumns = { @JoinColumn(name = "publicationId", referencedColumnName = "id") }, inverseJoinColumns = { @JoinColumn(name = "authorId", referencedColumnName = "id") })

-     private Set authors = new HashSet();

-     @Column

-     @Temporal(TemporalType.DATE)

-     private Date publishingDate;

-     …

- }

- view rawPublication_TablePerClass.java hosted with ♡ by GitHub

- The definitions of the Book and BlogPost entities are identical to the previously discussed mapped superclass strategy. You just have to extend the Publication class, add the @Entity annotation and add the class specific attributes with their mapping annotations.

```java
@Entity(name = "Book")
public class Book extends Publication {
        @Column
        private int pages;

        …
}
@Entity(name = "BlogPost")
public class BlogPost extends Publication {
        @Column
        private String url;

        …
}
```

- The table per class strategy maps each entity to its own table which contains a column for each entity attribute. That makes the query for a specific entity class easy and efficient.

- List books = em.createQuery("SELECT b FROM Book b", Book.class).getResultList();

- The superclass is now also an entity and you can, therefore, use it to define a relationship between the Author and the Publication entity. This allows you to call the getPublications() method to get all Publications written by that Author. Hibernate will map each Publication to its specific subclass.

- List authors= em.createQuery("SELECT a FROM Author a", Author.class).getResultList();
- for (Author a : authors) {
-   for (Publication p : a.getPublications()) {
-     if (p instanceof Book)
-     log(p.getTitle(), "book");
-     else
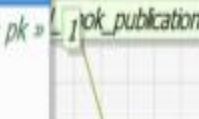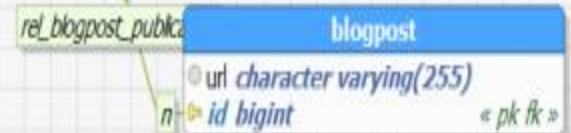-   log(p.getTitle(), "blog post");
-   }
- }

- The Java code looks easy and comfortable to use. But if you have a look at the generated SQL statement, you recognize that the table model makes the required query quite complicated.

- Hibernate has to join the author table with the result of a subselect which uses a union to get all matching records from the book and blogpost tables. Depending on the amounts of records in both tables, this query might become a performance issue. And it gets even worse if you add more subclasses to the inheritance structure. You should, therefore, try to avoid these kinds of queries or choose a different inheritance strategy.

- List authors= em.createQuery("SELECT a FROM Author a", Author.class).getResultList();
- for (Author a : authors) {
-      for (Publication p : a.getPublications()) {
-           if (p instanceof Book)
-           log(p.getTitle(), "book");
-           else
-           log(p.getTitle(), "blog post");
-      }
- }

- Joined

- The joined table approach maps each class of the inheritance hierarchy to its own database table. This sounds similar to the table per class strategy. But this time, also the abstract superclass Publication gets mapped to a database table. This table contains columns for all shared entity attributes. The tables of the subclasses are much smaller than in the table per class strategy. They hold only the columns specific for the mapped entity class and a primary key with the same value as the record in the table of the superclass.

public

book

pages	integer	
id	bigint	« pk fk »

author

id	bigint	« pk »
firstname	character varying(255)	
lastname	character varying(255)	
version	integer	« nn »

publicationauthor

publicationid	bigint	« pk fk »
authorid	bigint	« pk fk »

publication

id	bigint	« pk »
publishingdate	date	
title	character varying(255)	
version	integer	

rel_publicationauthor_author

rel_publicationauthor_pub

ok_publication

rel_blogpost_publica

blogpost

url	character varying(255)	
id	bigint	« pk fk »

- Each query of a subclass requires a join of the 2 tables to select the columns of all entity attributes. That increases the complexity of each query, but it also allows you to use not null constraints on subclass attributes and to ensure data integrity. The definition of the superclass Publication is similar to the previous examples. The only difference is the value of the inheritance strategy which is InheritanceType.JOINED.

- @Entity
- @Inheritance(strategy = InheritanceType.JOINED)
- public abstract class Publication {

- @Id
- @GeneratedValue(strategy = GenerationType.AUTO)
- @Column(name = "id", updatable = false, nullable = false)
- protected Long id;

- @Column
- protected String title;

-

```java
@Version
    @Column(name = "version")
    private int version;


    @ManyToMany
    @JoinTable(name = "PublicationAuthor", joinColumns = { @JoinColumn(name = "publicationId", referencedColumnName = "id") }, inverseJoinColumns = { @JoinColumn(name = "authorId", referencedColumnName = "id") })
    private Set authors = new HashSet();


    @Column
    @Temporal(TemporalType.DATE)
    private Date publishingDate;


    …
}
```

- The definition of the subclasses doesn't require any additional annotations. They just extend the superclass, provide an @Entity annotation and define the mapping of their specific attributes.
- @Entity(name = "Book")
- public class Book extends Publication {

- 	@Column
- 	private int pages;

- 	…
- }

- @Entity(name = "BlogPost")
- public class BlogPost extends Publication {

-     @Column
-     private String url;

-     …
- }

- the columns mapped by each subclass are stored in 2 different database tables. The publication table contains all columns mapped by the superclass Publication and the book table all columns mapped by the Book entity. Hibernate needs to join these 2 tables by their primary keys to select all attributes of the Book entity. This is an overhead that makes these queries slightly slower than the simpler queries generated for the single table strategy.

- List books = em.createQuery("SELECT b FROM Book b", Book.class).getResultList();

- Hibernate has to use a similar approach for polymorphic queries. It has to left join the publication table with all tables of the subclasses, to get all Pubications of an Author

- List authors= em.createQuery("SELECT a FROM Author a", Author.class).getResultList();
- for (Author a : authors) {
-      for (Publication p : a.getPublications()) {
-           if (p instanceof Book)
-           log(p.getTitle(), "book");
-           else
-           log(p.getTitle(), "blog post");
-      }
- }

Choosing a Strategy

- Choosing the right inheritance strategy is not an easy task. As so often, you have to decide which advantages you need and which drawback you can accept for your application. Here are a few recommendations:

- If you require the best performance and need to use polymorphic queries and relationships, you should choose the single table strategy. But be aware, that you can't use not null constraints on subclass attributes which increase the risk of data inconsistencies.

- If data consistency is more important than performance and you need polymorphic queries and relationships, the joined strategy is probably your best option.

- If you don't need polymorphic queries or relationships, the table per class strategy is most likely the best fit. It allows you to use constraints to ensure data consistency and provides an option of polymorphic queries. But keep in mind, that polymorphic queries are very complex for this table structure and that you should avoid them.