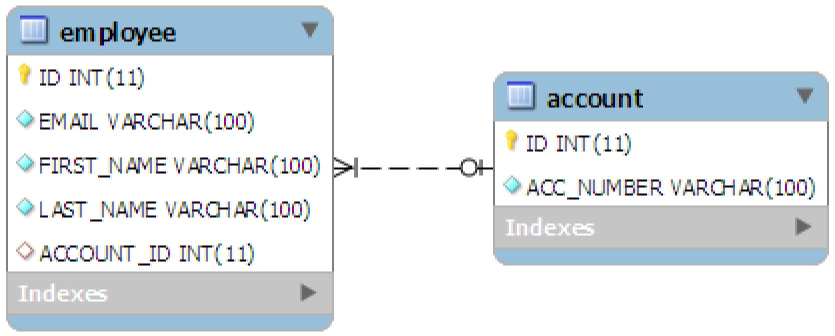
Hibernate Mapping and Inheritance annotations

- 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.



Association is managed by EmployeeEntity

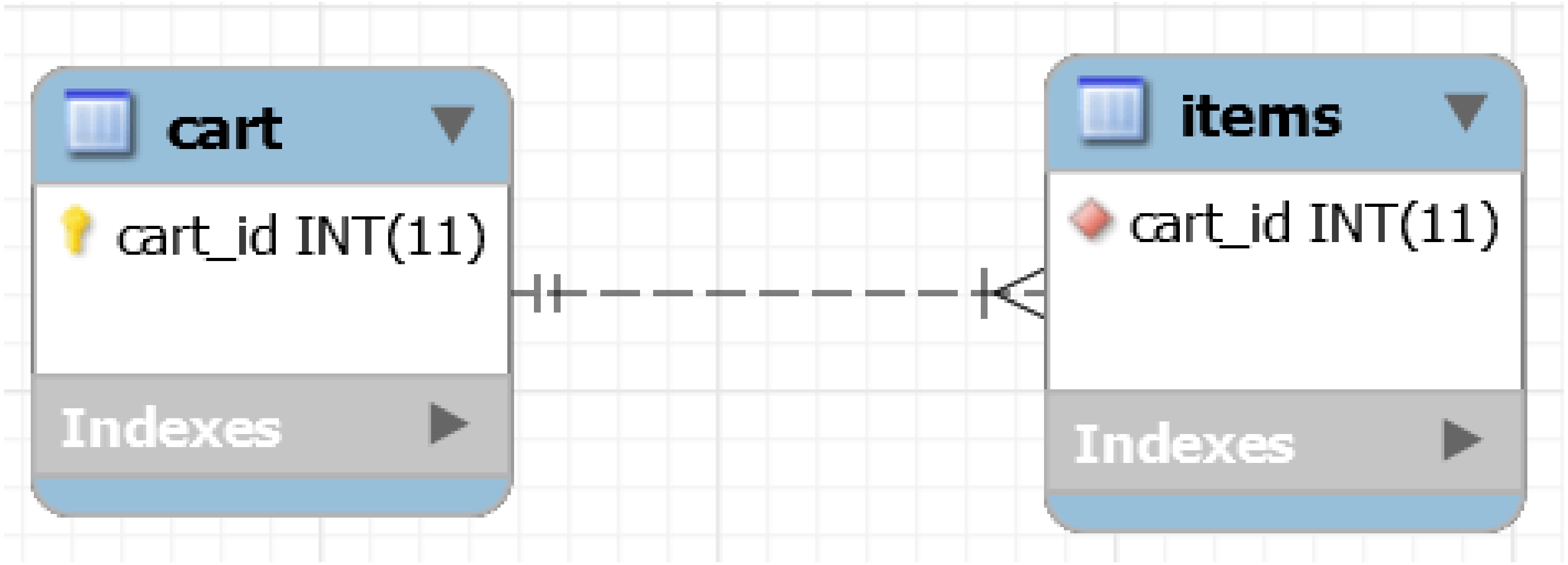
EmployeeEntity.java

```
@OneToOne
@JoinColumn(name="ACCOUNT_ID")
private AccountEntity account;
```

AccountEntity.java

```
@OneToOne(mappedBy="account")
private EmployeeEntity employee;
```

OneToMany



```
public class Cart {

    //...

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

    //...
}
```

- 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.

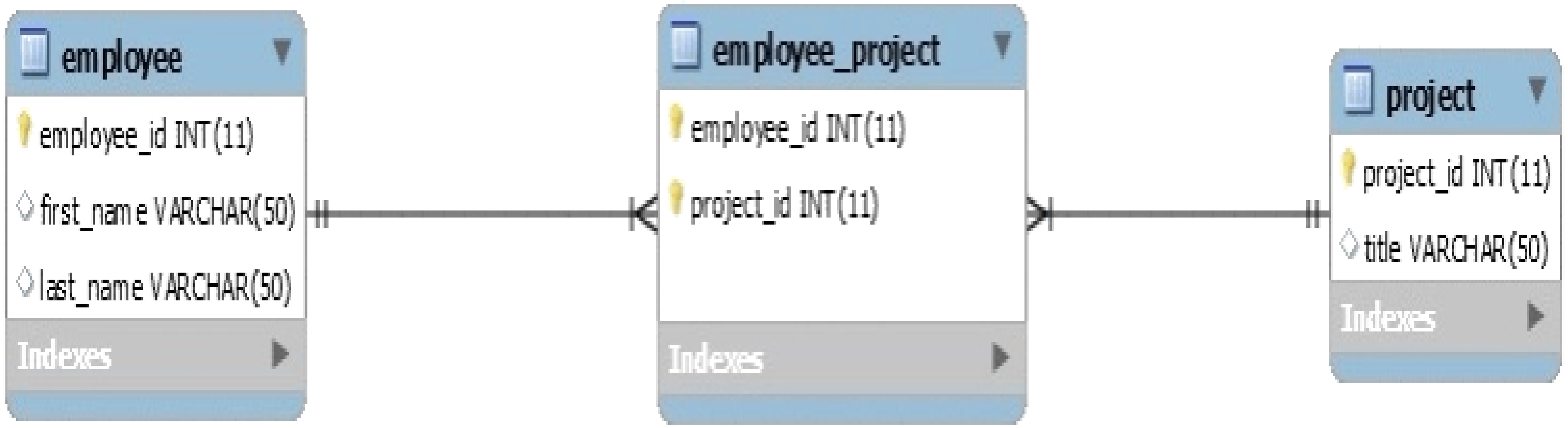
```
• @Entity
• @Table(name="CART")
• public class Cart {
•
•     //...
•
•     @OneToMany(mappedBy="cart")
•     private Set<Items> items;
•
•     // getters and setters
• }
```

- 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

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

ManyToMany relation



- In order to map a many-to-many association, we use the @ManyToMany, @JoinTable and @JoinColumn annotations
- **This association has two sides i.e. the owning side and the inverse side.** In our example, 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.*

```
• @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
• }
```

- @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:

- MappedSuperclass – the parent classes, can’t be entities
- 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

- MappedSuperclass
- Using the MappedSuperclass strategy, inheritance is only evident in the class, but not the entity model.

- Let’s start by creating a Person class which will represent a parent class:

- @MappedSuperclass
- public class Person {
-
-     @Id
-     private long personId;
-     private String name;
-
-     // constructor, getters, setters
- }
- Notice that this class no longer has an @Entity annotation, as it won’t be persisted in the database

- let’s add an Employee sub-class:

- @Entity
- public class MyEmployee extends Person {
-     private String company;
-     // constructor, getters, setters
- }

- Since the records for all entities will be in the same table, Hibernate needs a way to differentiate between them.

- By default, this is done through a discriminator column called DTYPE which has the name of the entity as a value.

- 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.

- Next, we need to tell Hibernate what value each sub-class record will have for the product_type column:

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

- Joined Table
- Using this strategy, 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.

- Let’s create a super-class that uses this strategy:

- @Entity
- @Inheritance(strategy = InheritanceType.JOINED)
- public class Animal {
-     @Id
-     private long animalId;
-     private String species;
-
-     // constructor, getters, setters
- }

- Then, we can simply define a sub-class:

- 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:

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

Table Per Class

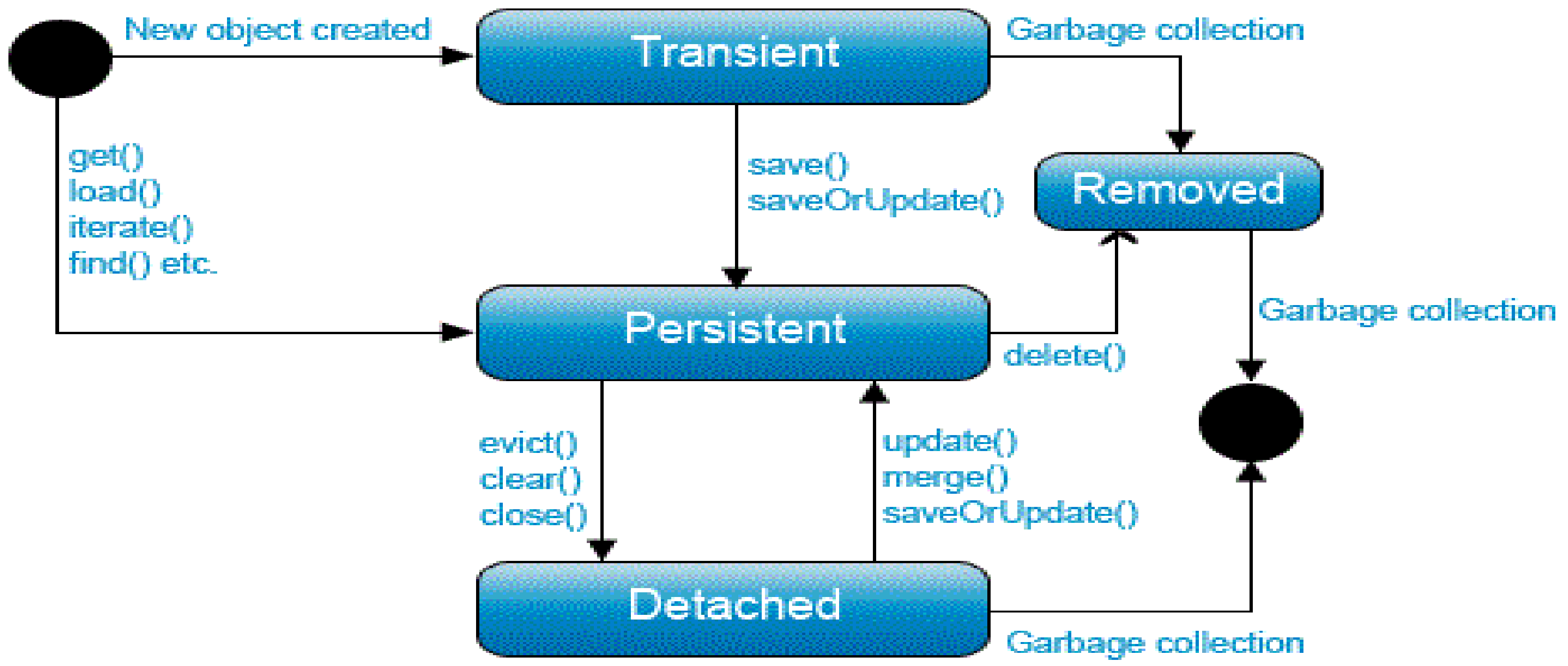
- The Table Per Class strategy maps each entity to its table which contains all the properties of the entity, including the ones inherited.
- The resulting schema is similar to the one using @MappedSuperclass, but unlike it, table per class will indeed define entities for parent classes, allowing associations 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
}
```

Diagram :

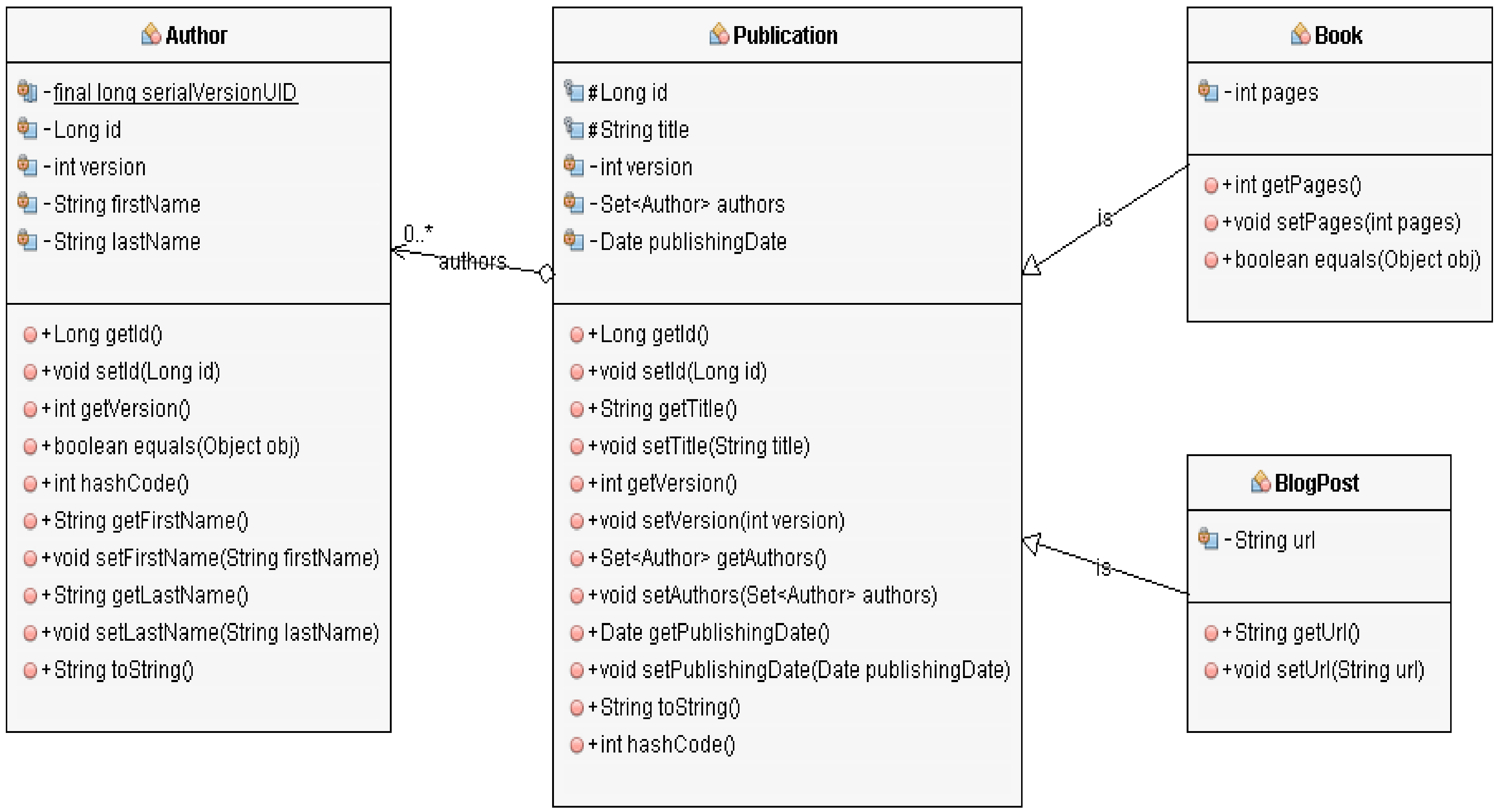

New object created
Transient
Garbage collection
get()
load()
iterate()
find() etc.
save()
saveOrUpdate()
Removed
Persistent
delete()
Garbage collection
evict()
clear()
close()
update()
merge()
saveOrUpdate()
Detached
Garbage collection

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()
0..*
authors
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()
is
is
Book
- int pages
+ int getPages()
+void setPages(int pages)
+ boolean equals(Object obj)
BlogPost
- String url
+ String getUrl()
+void setUrl(String url)

Mapped Superclass

- The mapped superclass strategy is the simplest approach to mapping an inheritance structure to database tables. It maps each concrete class to its own table.

- mappedSuperClass

- That allows you to share the attribute definition between multiple entities. But it also has a huge drawback. A mapped superclass is not an entity, and there is no table for it.

- That means that you can’t use polymorphic queries that select all Publication entities and you also can’t define a relationship between an Author entity and all Publications. You either need to use uni-directional relationship from the Publication to the Author entity, or you have to define a relationship between an Author and each kind of Publication. In general, if you need these relationships, you should have a look at the other inheritance strategies. They are most likely a better fit for your use case.

- If you just want to share state and mapping information between your entities, the mapped superclass strategy is a good fit and easy to implement. You just have to set up your inheritance structure, annotate the mapping information for all attributes and add the @MappedSuperclass annotation to your superclass. Without the @MappedSuperclass annotation, Hibernate will ignore the mapping information of your superclass.

- You can see an example of such a mapping in the following code snippets. the Publication class is annotated with @MappedSuperclass and provides the shared attributes with their mapping annotations. As you can see, Publication has no @Entity annotation and will not be managed by the persistence provider.

```
@MappedSuperclass
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;

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

        …
}
```

- The subclasses Book and BlogPost extend the Publication class and add their specific attributes with their mapping annotations. Both classes are also annotated with @Entity and will be managed by the persistence provider.

- @Entity(name = “Book”)
- public class Book extends Publication {

- @Column
- private int pages;

- …
- }

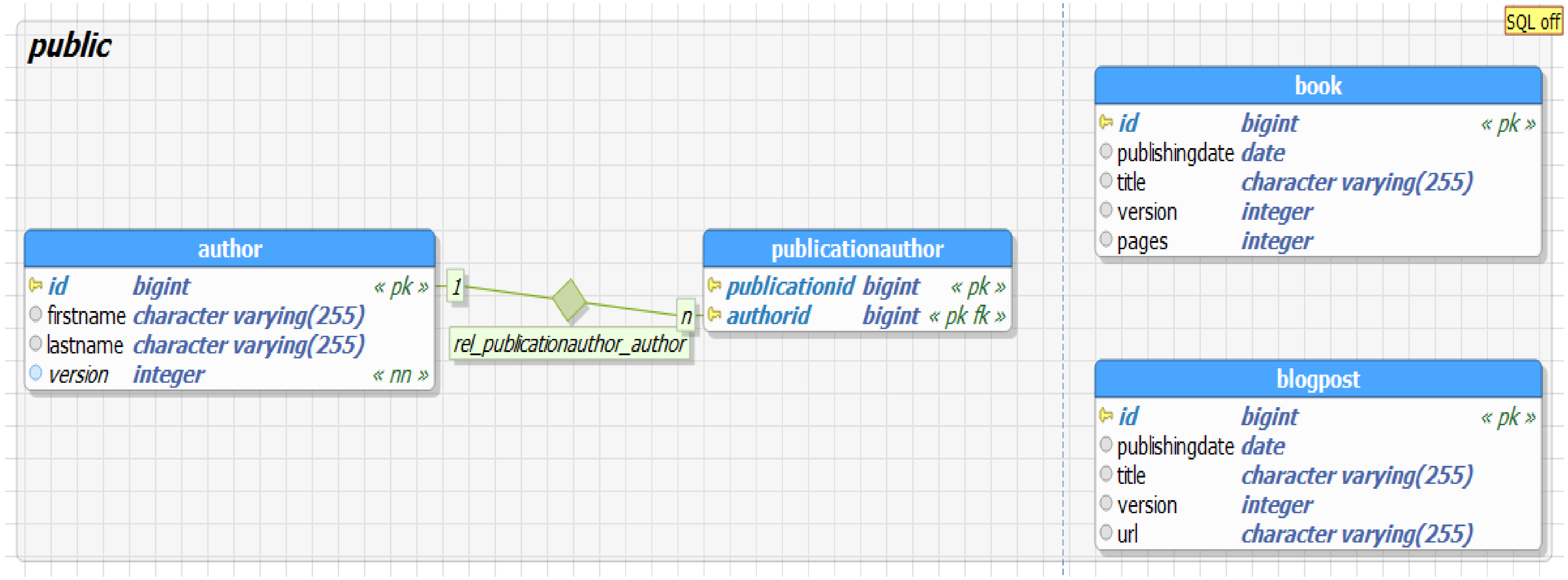
- @Entity(name = “BlogPost”)
- public class BlogPost extends Publication {

- 	@Column
- 	private String url;

- 	…
- }
- As explained at the beginning of this section, you can’t use the inheritance structure for polymorphic queries or to define relationships. But you can, of course, query the entities as any other entity.

- 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
SQL off
author
id bigint « pk »
firstname character varying(255)
lastname character varying(255)
version integer « nn »
1
rel_publicationauthor_author
n
publicationauthor
publicationid bigint « pk »
authorid bigint « pk fk »
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.

- @Entity(name = “Book”)
- public class Book extends Publication {

- @Column
- private int pages;

- …
- }
- view rawBook_TablePerClass.java hosted with ♡ by GitHub
- @Entity(name = “BlogPost”)
- public class BlogPost extends Publication {

- @Column
- private String url;

- …
- }
- view rawBlogPost_TablePerClass.java hosted with ♡ by GitHub

- 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.

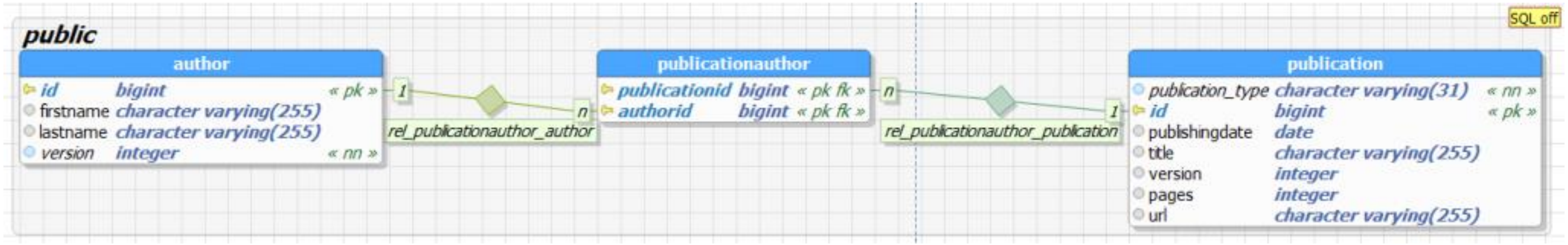
```
• List authors= em.createQuery(“SELECT a FROM Author a”,
  Author.class).getResultList();
• for (Author a : authors) {
•         for (Publication p : a.getPublications()) {
•                 if (p instanceof Book)
•                 log(p.getTitle(), “book”);
•                 else
•                 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.

-
  ```
  15:57:16,722 DEBUG [org.hibernate.SQL] – select author0_.id as id1_0_, author0_.firstName as firstNam2_0_, author0_.lastName as lastName3_0_, author0_.version as version4_0_ from Author author0_
  ```
-
  ```
  15:57:16,765 DEBUG [org.hibernate.SQL] – select publicatio0_.authorId as authorId2_4_0_, publicatio0_.publicationId as publicat1_4_0_, publicatio1_.id as id1_3_1_, publicatio1_.publishingDate as publishi2_3_1_, publicatio1_.title as title3_3_1_, publicatio1_.version as version4_3_1_, publicatio1_.pages as pages1_2_1_, publicatio1_.url as url1_1_1_, publicatio1_.clazz_ as clazz_1_ from PublicationAuthor publicatio0_ inner join ( select id, publishingDate, title, version, null::int4 as pages, null::varchar as url, 0 as clazz_ from Publication union all select id, publishingDate, title, version, pages, null::varchar as url, 1 as clazz_ from Book union all select id, publishingDate, title, version, null::int4 as pages, url, 2 as clazz_ from BlogPost ) publicatio1_ on publicatio0_.publicationId=publicatio1_.id where publicatio0_.authorId=?
  ```
- Effective Java is a book.
- view rawPublicationsOfAuthor_TablePerClass.log hosted with ♡ by GitHub
- 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.

- Single Table
- The single table strategy maps all entities of the inheritance structure to the same database table. This approach makes polymorphic queries very efficient and provides the best performance.

- But it also has some drawbacks. The attributes of all entities are mapped to the same database table. Each record uses only a subset of the available columns and sets the rest of them to null. You can, therefore, not use not null constraints on any column that isn’t mapped to all entities. That can create data integrity issues, and your database administrator might not be too happy about it.


public
SQL off
author
id bigint « pk »
firstname character varying(255)
lastname character varying(255)
version integer « nn »
1
rel_publicationauthor_author
n
publicationauthor
publicationid bigint « pk fk »
authorid bigint « pk fk »
n
rel_publicationauthor_publication
1
publication
publication_type character varying(31) « nn »
id bigint « pk »
publishingdate date
title character varying(255)
version integer
pages integer
url character varying(255)

- When you persist all entities in the same table, Hibernate needs a way to determine the entity class each record represents. This is information is stored in a discriminator column which is not an entity attribute. You can either define the column name with a @DiscriminatorColumn annotation on the superclass or Hibernate will use DTYPE as its default name.

```
• @Entity
• @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
• @DiscriminatorColumn(name = “Publication_Type”)
• 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;

- 		…
- }

- The definition of the subclasses is again similar to the previous examples. But this time, you should also provide a @DiscriminatorValue annotation. It specifies the discriminator value for this specific entity class so that your persistence provider can map each database record to a concrete entity class.

- The @DiscriminatorValue annotation is optional if you use Hibernate. If you don’t provide a discriminator value, Hibernate will use the simple entity name by default. But this default handling isn’t defined by the JPA specification, and you shouldn’t rely on it.

- @Entity(name = “Book”)
- @DiscriminatorValue(“Book”)
- public class Book extends Publication {
- @Column
- private int pages;
- …
- }

- @Entity(name = “BlogPost”)
- @DiscriminatorValue(“Blog”)
- public class BlogPost extends Publication {
-
- 	@Column
- 	private String url;
-
- 	…
- }

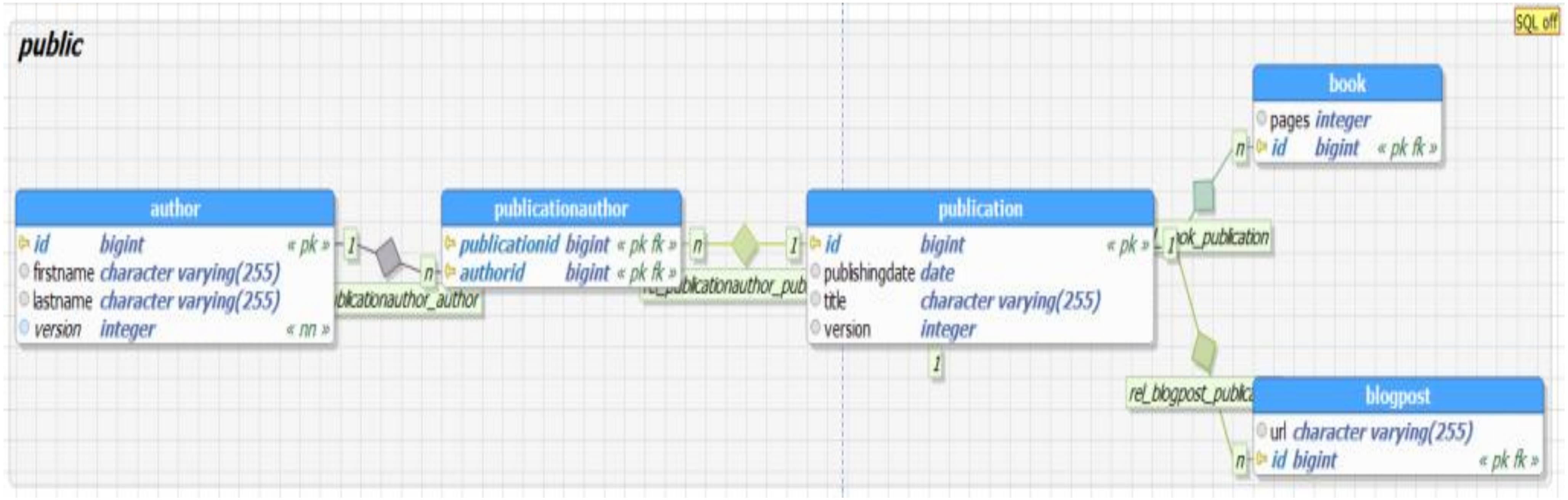
- As I explained at the beginning of this section, the single table strategy allows easy and efficient data access. All attributes of each entity are stored in one table, and the query doesn’t require any join statements. The only thing that Hibernate needs to add to the SQL query to fetch a particular entity class is a comparison of the discriminator value. In this example, it’s a simple expression that checks that the column publication_type contains the value ‘Book‘.

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

- The previously discussed inheritance strategies had their issues with polymorphic queries. They were either not supported or required complex union and join operations. That’s not the case if you use the single table strategy. All entities of the inheritance hierarchy are mapped to the same table and can be selected with a simple query. The following code and log snippets show an example for such a query. As you can see in the log messages, Hibernate selects all columns, including the discriminator column publication_type, from the publication table. It then uses the discriminator value to select the right entity class and to map the database record. This query is much easier than the one created by the table per class strategy, and you don’t need to worry about performance problems.

```
• 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.



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	

book

pages	integer	
id	bigint	« pk fk »

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;
-
-

- @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;

-    …
- }

- As I already explained, 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();

- 15:56:21,463 DEBUG [org.hibernate.SQL] – select book0_.id as id1_3_, book0_.publishingDate as publishi2_3_, book0_.title as title3_3_, book0_.version as version4_3_, book0_.pages as pages1_2_ from Book book0_

- 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”);
•         }
• }
```

- 17:16:05,244 DEBUG [org.hibernate.SQL] – select author0_.id as id1_0_, author0_.firstName as firstNam2_0_, author0_.lastName as lastName3_0_, author0_.version as version4_0_ from Author author0_
- 17:16:05,280 DEBUG [org.hibernate.SQL] – select publicatio0_.authorId as authorId2_4_0_, publicatio0_.publicationId as publicat1_4_0_, publicatio1_.id as id1_3_1_, publicatio1_.publishingDate as publishi2_3_1_, publicatio1_.title as title3_3_1_, publicatio1_.version as version4_3_1_, publicatio1_1_.pages as pages1_2_1_, publicatio1_2_.url as url1_1_1_, case when publicatio1_1_.id is not null then 1 when publicatio1_2_.id is not null then 2 when publicatio1_.id is not null then 0 end as clazz_1_ from PublicationAuthor publicatio0_ inner join Publication publicatio1_ on publicatio0_.publicationId=publicatio1_.id left outer join Book publicatio1_1_ on publicatio1_.id=publicatio1_1_.id left outer join BlogPost publicatio1_2_ on publicatio1_.id=publicatio1_2_.id where publicatio0_.authorId=?
- Effective Java is a book.

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.