## Inheritance mapping

Java is an object oriented language. It is possible to implement Inheritance in Java. Inheritance is one of the most visible facets of Object-relational mismatch. Object oriented systems can model both **“is a”** and **“has a”** relationship. Relational model supports only “has a” relationship between two entities. Hibernate can help you map such Objects with relational tables. But you need to choose certain mapping strategy based on your needs.

Hibernate supports the three basic inheritance mapping strategies:

* table per class hierarchy
* table per subclass
* table per concrete class

In addition, Hibernate supports a fourth, slightly different kind of polymorphism:

* implicit polymorphism

## One Table Per Class Hierarchy example

Suppose we have a class Person with subclass Employee. The properties of each class are:

\* class Person

- firstname

- lastname

\* class Employee

- joining\_date

- department\_name

In One Table per Class Hierarchy scheme, we store all the class hierarchy in a single SQL table. A discriminator is a key to uniquely identify the base type of the class hierarchy.

Following are the advantages and disadvantages of One Table per Class Hierarchy scheme.

#### Advantage

This hierarchy offers the best performance even for in the deep hierarchy since single select may suffice.

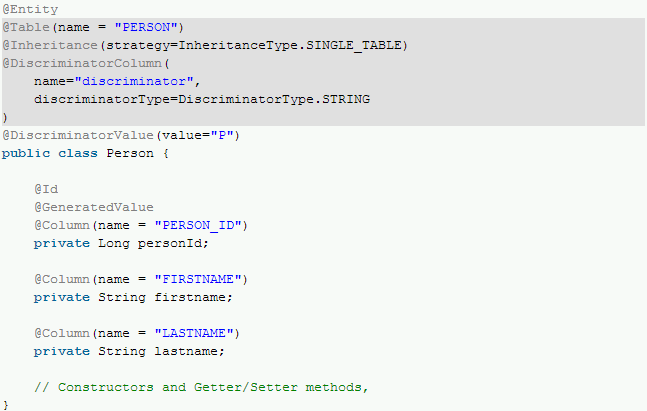
#### Disadvantage

Changes to members of the hierarchy require column to be altered, added or removed from the table.

## Create Database Table to persist Class Hierarchy

|  |
| --- |
|  |

The PERSON table will store both Employee and Person objects.

Following is the example where we map Employee and Person entity classes using JPA Annotations. File: Person.javaThe Person class is the root of hierarchy. Hence we have used some annotations to make it as the root.

**@Inheritance** – Defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity class that is the root of the entity class hierarchy.

**@DiscriminatorColumn** – Is used to define the discriminator column for the **SINGLE\_TABLE** and JOINED inheritance mapping strategies. The strategy and the discriminator column are only specified in the root of an entity class hierarchy or sub hierarchy in which a different inheritance strategy is applied

If the @DiscriminatorColumn annotation is missing, and a discriminator column is required, the name of the discriminator column defaults to "DTYPE" and the discriminator type to DiscriminatorType.STRING.

**@DiscriminatorValue** – Is used to specify the value of the discriminator column for entities of the given type. The DiscriminatorValue annotation can only be specified on a concrete entity class. If the DiscriminatorValue annotation is not specified and a discriminator column is used, a provider-specific function will be used to generate a value representing the entity type. If the DiscriminatorType is STRING, the discriminator value default is the entity name.

The inheritance strategy and the discriminator column are only specified in the root of an entity class hierarchy or subhierarchy in which a different inheritance strategy is applied. The discriminator value, if not defaulted, should be specified for each entity class in the hierarchy.



Employee class is child of Person class. Thus while specifying the mappings, we used@DiscriminatorValue to specify discriminator value. In our case “E” will be persisted in discriminator column.

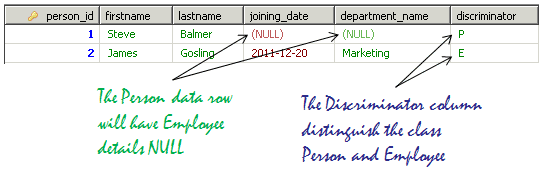
## Main class

|  |
| --- |
|  |

The Main class is used to persist Person and Employee object instance. Note that both these classes are persisted in same table PERSON. The discriminator column is used to distinguished between the entities.

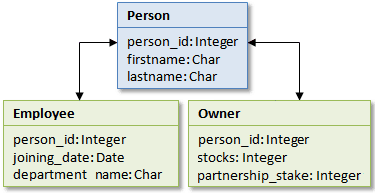
#### Output

|  |
| --- |
| Hibernate: insert into EMPLOYEE (FIRSTNAME, LASTNAME, discriminator) values (?, ?, 'P')  Hibernate: insert into EMPLOYEE (FIRSTNAME, LASTNAME, department\_name, joining\_date, discriminator) values (?, ?, ?, ?, 'E') |



## One Table Per Subclass example

Suppose we have a class Person with subclass Employee and Owner. Following the class diagram and relationship of these classes.



In One Table per Subclass scheme, each class persist the data in its own separate table. Thus we have 3 tables; PERSON, EMPLOYEE and OWNER to persist the class data. Note that a foreign key relationship exists between the subclass tables and super class table. Thus the common data is stored in PERSON table and subclass specific fields are stored in EMPLOYEE and OWNER tables.

Following are the advantages and disadvantages of One Table per Subclass scheme.

#### Advantage

* Using this hierarchy, does not require complex changes to the database schema when a single parent class is modified.
* It works well with shallow hierarchy.

#### Disadvantage

* As the hierarchy grows, it may result in poor performance.
* The number of joins required to construct a subclass also grows.

CREATE TABLE `person` (

    `person\_id` BIGINT(20) NOT NULL AUTO\_INCREMENT,

    `firstname` VARCHAR(50) NOT NULL DEFAULT '0',

    `lastname` VARCHAR(50) NOT NULL DEFAULT '0',

    PRIMARY KEY (`person\_id`)

)

CREATE TABLE `employee` (

    `person\_id` BIGINT(10) NOT NULL,

    `joining\_date` DATE NULL DEFAULT NULL,

    `department\_name` VARCHAR(50) NULL DEFAULT NULL,

    PRIMARY KEY (`person\_id`),

    CONSTRAINT `FK\_PERSON` FOREIGN KEY (`person\_id`) REFERENCES `person` (`person\_id`)

)

CREATE TABLE `owner` (

    `person\_id` BIGINT(20) NOT NULL DEFAULT '0',

    `stocks` BIGINT(11) NULL DEFAULT NULL,

    `partnership\_stake` BIGINT(11) NULL DEFAULT NULL,

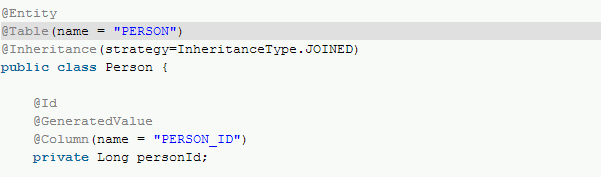
    PRIMARY KEY (`person\_id`),

    CONSTRAINT `FK\_PERSON2` FOREIGN KEY (`person\_id`) REFERENCES `person` (`person\_id`)

)

Following is the example where we map Employee and Person entity classes using JPA Annotations.

*File: Person.java*

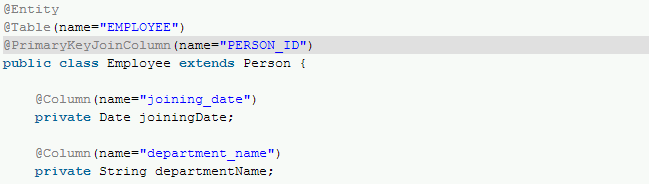


The Person class is the root of hierarchy. Hence we have used some annotations to make it as the root.

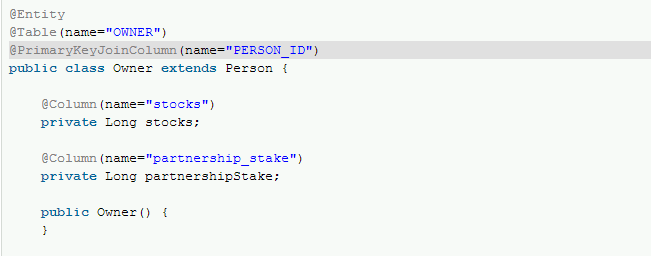
**@Inheritance** – Defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity class that is the root of the entity class hierarchy.

**@InheritanceType** – Defines inheritance strategy options. **JOINED** is a strategy in which fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class, and a join is performed to instantiate the subclass. Thus fields of Employee (joining\_date, department) and Owner (stocks etc) are mapped to their respective tables.

*File: Employee.java*



File: Owner.java



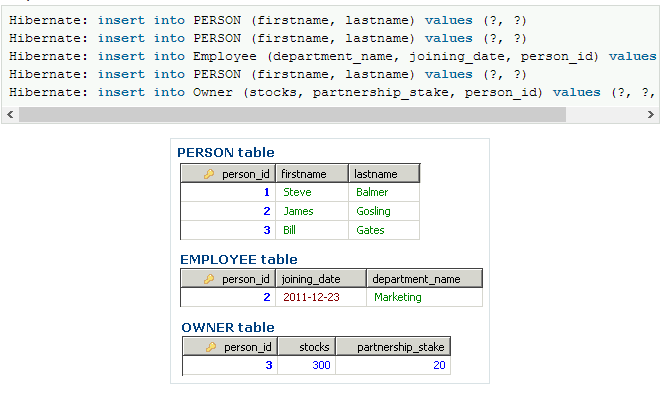
Both Employee and Owner classes are child of Person class. Thus while specifying the mappings, we used @PrimaryKeyJoinColumn to map it to parent table.

**@PrimaryKeyJoinColumn** – This annotation specifies a primary key column that is used as a foreign key to join to another table.

It is used to join the primary table of an entity subclass in the JOINED mapping strategy to the primary table of its superclass; it is used within a SecondaryTable annotation to join a secondary table to a primary table; and it may be used in a OneToOne mapping in which the primary key of the referencing entity is used as a foreign key to the referenced entity.

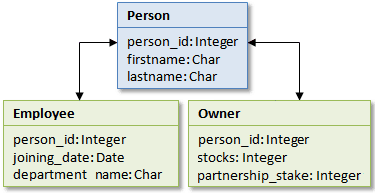
If no PrimaryKeyJoinColumn annotation is specified for a subclass in the JOINED mapping strategy, the foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass





## *One Table per Concrete Class example*

Suppose we have a class Person with subclasses Employee and Owner. Following the class diagram and relationship of these classes.



In One Table per Concrete class scheme, each concrete class is mapped as normal persistent class. Thus we have 3 tables; PERSON, EMPLOYEE and OWNER to persist the class data. In this scheme, the mapping of the subclass repeats the properties of the parent class.

Following are the advantages and disadvantages of One Table per Subclass scheme.

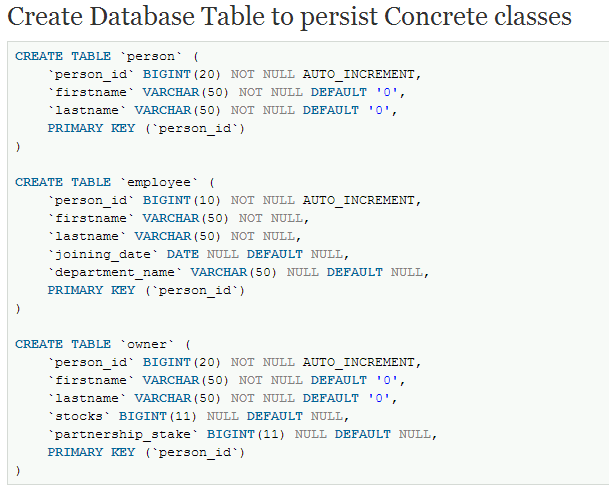
#### Advantages

* This is the easiest method of Inheritance mapping to implement.

#### Disadvantages

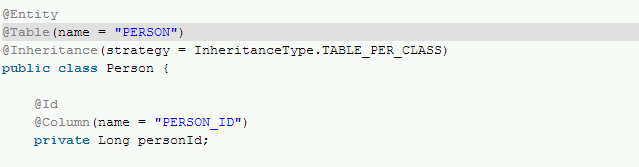
* Data that’s belongs to a parent class is scattered across a number of subclass tables, which represents concrete classes.
* This hierarchy is not recommended for most cases.
* Changes to a parent class is reflected to large number of tables
* A query couched in terms of parent class is likely to cause a large number of select operations

This strategy has many drawbacks (esp. with polymorphic queries and associations) explained in the JPA spec, the Hibernate reference documentation, Hibernate in Action, and many other places. Hibernate work around most of them implementing this strategy using SQL UNION queries. It is commonly used for the top level of an inheritance hierarchy:



Following is the example where we map Person, Employee and Owner entity classes using JPA Annotations.

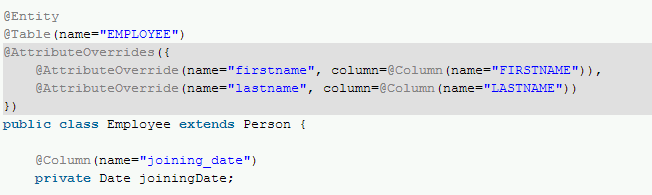
*File: Person.java*



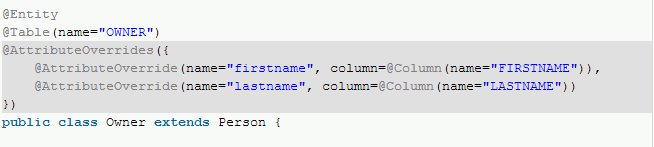
**@Inheritance** – Defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity class that is the root of the entity class hierarchy.

**@InheritanceType** – Defines inheritance strategy options. **TABLE\_PER\_CLASS** is a strategy to map table per concrete class.

*File: Employee.java*



File: Owner.java



Both Employee and Owner classes are child of Person class. Thus while specifying the mappings, we used @AttributeOverrides to map them.

**@AttributeOverrides** – This annotation is used to override mappings of multiple properties or fields.

**@AttributeOverride** – The AttributeOverride annotation is used to override the mapping of a Basic (whether explicit or default) property or field or Id property or field.

The AttributeOverride annotation may be applied to an entity that extends a mapped superclass or to an embedded field or property to override a basic mapping defined by the mapped superclass or embeddable class. If the AttributeOverride annotation is not specified, the column is mapped the same as in the original mapping.

Thus in over case, firstname and lastname are defined in parent class Person and mapped in child class Employee and Owner using @AttributeOverrides annotation.

This strategy supports one-to-many associations provided that they are bidirectional. This strategy does not support the IDENTITY generator strategy: the id has to be shared across several tables. Consequently, when using this strategy, you should not use AUTO nor IDENTITY. Note that in below Main class we specified the primary key explicitly.



The Main class is used to persist Person, Employee and Owner object instances. Note that these classes are persisted in different tables and parent attributes (firstname, lastname) are repeated across all tables.

#### Output

|  |
| --- |
| Hibernate: insert into PERSON (FIRSTNAME, LASTNAME, PERSON\_ID) values (?, ?, ?)  Hibernate: insert into EMPLOYEE (FIRSTNAME, LASTNAME, department\_name, joining\_date, PERSON\_ID) values (?, ?, ?, ?, ?)  Hibernate: insert into OWNER (FIRSTNAME, LASTNAME, partnership\_stake, stocks, PERSON\_ID) values (?, ?, ?, ?, ?) |

