Like all object-oriented programing languages, Java supports the fundamental concepts of inheritance and composition. There is an important difference between both concepts. Inheritance enables you to model an is-a association between two classes by extending a superclass. Composition models a has-a association by referencing another class in an instance variable.

For plain Java classes, there have been hundreds of discussions about this question. And there is a clear winner: You should prefer composition unless the classes are specifically designed for extension.

But all these discussions and recommendations are for plain Java classes. Is this also valid for entity classes annotated with JPA annotations and mapped to tables in a relational database? And how should you implement the mapping?

Inheritance vs. Composition for Entity Classes

The general recommendation to use composition over inheritance is also valid for entity classes. In addition to all arguments that apply to plain Java classes, there is another reason to use composition to implement your entity classes. When you are working with entities, you always need to keep in mind that your classes will be mapped to database tables.

Relational databases don't support the inheritance concept. You have to use one of JPA's inheritance strategies to map your inheritance hierarchy to one or more database tables. When you choose one of these strategies, you need to decide:

- if you want to forgo constraints that ensure data consistency so that you get the best performance, or
- if you accept inferior performance so that you can use the constraints.

As you can see, both approaches have their disadvantages. You don't run into these problems if you use composition.

Using Composition with JPA and Hibernate

JPA and Hibernate support 2 basic ways to implement composition:

1. You can reference another entity in your composition by modeling an association to it. Each entity will be mapped to its database table and can be loaded independently. A typical example is a *Person* entity which has a one-to-many association to an *Address* entity.
2. You can also use an embeddable to include its attributes and their mapping information into your entity mapping. The attributes are mapped to the same table as the other attributes of the entity. The embeddable can't exist on its own and has no persistent identity. You can use it to define a composition of address attributes which become part of the *Person* entity.

Composition via association mapping

For a lot of developers, this is the most natural way to use composition in entity mappings. It uses concepts that are well-established and easy to use in both worlds: the database, and the Java application.

Associations between database records are very common and easy to model. You can:

- store the primary key of the associated record in one of the fields of your database record to model a one-to-many association. A typical example is a record in the *Book* table which contains the primary key of a record in the *Publisher* table. This enables you to store a reference to the publisher who published the book.
- introduce an association table to model a many-to-many relationship. Each record in this table stores the primary key of the associated record. E.g., a *BookAuthor* table stores the primary keys of records in the *Author* and *Book* tables to persist which authors wrote a specific book.

One of the main benefits of JPA and Hibernate is that you can easily map these associations in your entities. You just need an attribute of

the type of the associated entity or a collection of the associated entities and a few annotation. I explained these mappings in great details in one of my previous posts: Ultimate Guide – Association Mappings with JPA and Hibernate.

Composition with embeddables

Embeddables are another option to use composition when implementing your entities. They enable you to define a reusable set of attributes with mapping annotations. The main difference to the previously discussed association mapping is that an embeddable becomes part of the entity and has no persistent identity on its own.

Let's take a look at an example. The 3 attributes of the *Address* class store simple address information. The *@Embeddable* annotation tells Hibernate and any other JPA implementation that this class and its mapping annotations can be embedded into an entity. In this example, I rely on JPA's default mappings and don't provide any mapping information.

```
@Embeddable
public class Address {

    private String street;
    private String city;
    private String postalCode;

    ...
}
```

After you have defined your embeddable, you can use it as the type of an entity attribute. You just need to annotate it with *@Embedded*, and your persistence provider will include the attributes and mapping information of your embeddable in the entity. So, in this

example, the attributes *street*, *city* and *postalCode* of the embeddable *Address* will be mapped to columns with the same names of the *Author* table.

```
@Entity
public class Author implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Embedded
    private Address address;

    ...
}
```

If you want to use multiple attributes of same embeddable type, you need to override the column mapping of the attributes of the embeddable. You can do that with a collection of *@AttributeOverride* annotations. Since JPA 2.2, the *@AttributeOverride* is repeatable, and you no longer need to wrap it in an *@AttributeOverrides* annotation.

```
@Entity
public class Author implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Embedded
    private Address privateAddress;

    @Embedded
    @AttributeOverride(
        name = "street",
        column = @Column( name = "business_street" )
    )
    @AttributeOverride(
        name = "city",
        column = @Column( name = "business_city" )
    )
    @AttributeOverride(
        name = "postalCode",
        column = @Column( name = "business_postcalcode" )
    )
    private Address businessAddress;

    ...
}
```