Inheritance Mapping Strategies in SAP JPA 1.0

Applies to:

SAP NetWeaver Composition Environment 7.1 EHP1. For more information, visit the Java homepage.

Summary

In Java it is trivial to have inheritance between classes. With JPA you can map Object Oriented inheritance to relational databases. This article explains the various inheritance mapping strategies supported by JPA with an example.

Author: Sampathi Gunda

Company: HCL Technologies

Created on: 22 September 2009

Author Bio

Sampath Gunda is working as a Java Developer in HCL Technologies, Chennai, India.

Table of Contents

Introduction

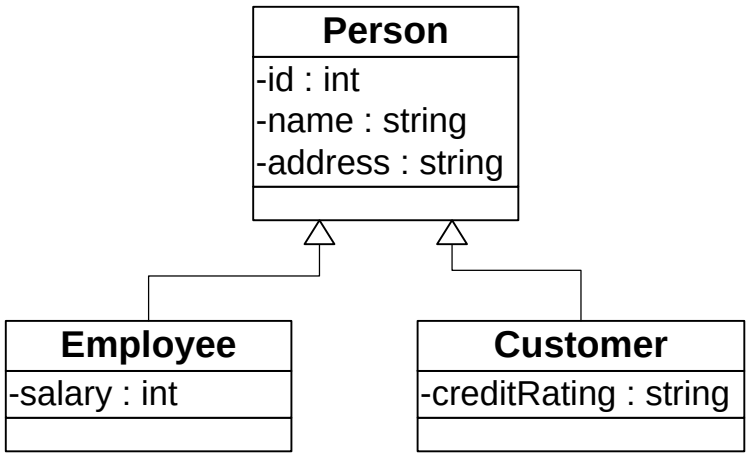
In Java it is trivial to have inheritance between classes. With JPA you can map Object Oriented inheritance to relational databases. This article explains the various inheritance mapping strategies supported by JPA with an example. We will also examine some of the advantages and disadvantages of each approach thereby helping you choose the best approach suitable to your requirements.

There are several mapping strategies available to support object oriented concept of inheritance in relational model. They are:

1. Single Table per Class Hierarchy
2. Joined Subclass
3. Separate Table per Concrete Class

SAP JPA 1.0 supports the first two strategies. In the next sections of this article, we will see how to implement these strategies along with their strengths and weaknesses.

These inheritance strategies are explained with the following simple example. There is a Person class that is the parent for Employee and Customer classes.



This article assumes that the reader is having a basic knowledge of SAP JPA 1.0 and he knows how to create a simple JPA application from scratch. You can refer to SAP help documentation or my article “SAP JPA 1.0, EJB 3.0 and Web Service -Modeling Your First JPA Entity in CE 7.1” to know how to create a simple JPA application from scratch.

Single Table per Class Hierarchy

This strategy is the default strategy in which a single table is used to represent the entire class hierarchy. A discriminator column is used to represent the individual sub classes. The discriminator column stores values that refer to the particular subclass of the row instance. The columns that correspond to subclass-specific states must be null-able. By default, the discriminator value for an entity is its entity name, but an entity may override this value using the **DiscriminatorValue** annotation.

Following listings show how the entities in the **Person** hierarchy are mapped using the “Single Table per Class Hierarchy” inheritance strategy.

```
@Entity
@Table(name="PERSON_SINGLE")
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=DiscriminatorType.STRING)
public abstract class Person implements Serializable {

   @Id
   private int id;
   private String name;
   private String address;
   private static final long serialVersionUID = 1L;

   public Person() {
    super();
   }
   public int getId() {
          return this.id;
   }
   public void setId(int id) {
          this.id = id;
   }
   public String getName () {
          return this.name;
   }
   public void setName(String name) {
          this.name = name;
   }
   public String getAddress() {
          return this.address;
   }
   public void setAddress(String address) {
          this.address = address;
   }
}
```

```
@Entity
@DiscriminatorValue("EMPLOYEE")
public class Employee extends Person implements Serializable {
   private int salary;
   private static final long serialVersionUID = 1L;

   public Employee() {
    super();
   }
   public int getSalary() {
          return this.salary;
   }
   public void setSalary(int salary) {
          this.salary = salary;
   }
}
```

```
@Entity
@DiscriminatorValue("CUSTOMER")
public class Customer extends Person implements Serializable {
   private String creditRating;
   private static final long serialVersionUID = 1L;

   public Customer() {
    super();
   }
   public String getCreditRating() {
          return this.creditRating;
   }
   public void setCreditRating(String creditRating) {
          this.creditRating = creditRating;
   }
}
```

Annotations @Inheritance, @DiscriminatorColumn and @DiscriminatorValue hints the application server that we are setting up a hierarchy using a single-table mapping strategy.

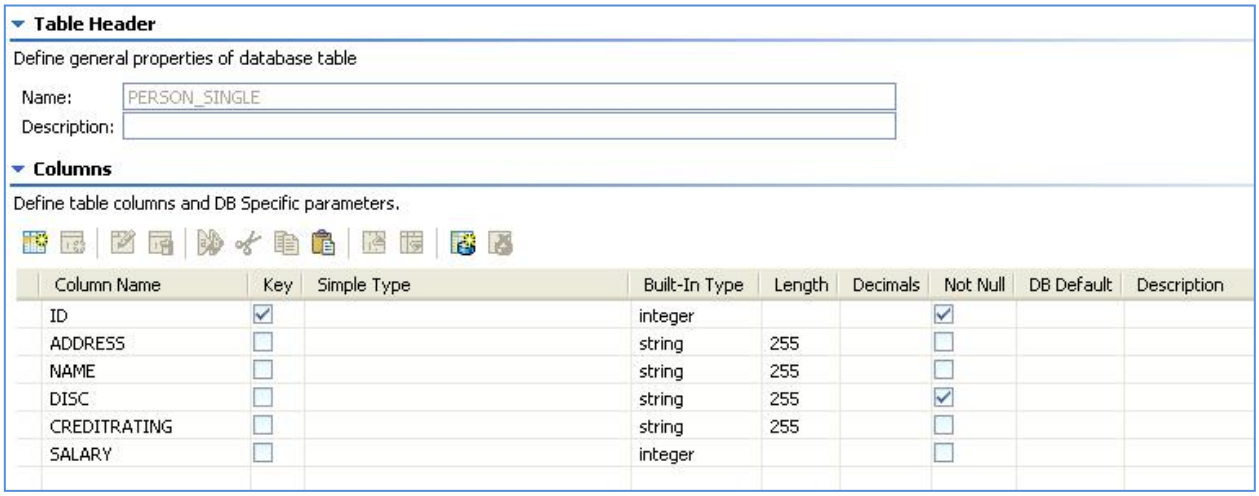
The strategy element of the annotation @Inheritance tells the application server what mapping strategy we are using, in this case single-table strategy.

The @DiscriminatorColumn annotation specifies the name of the discriminator column and its type. In our example, we have named the discriminator column DISC and declared its type String.

Each concrete entity declares a unique discriminator value that serves to identify the concrete entity type associated with each row in the table. The discriminator value defaults to the entity name, and in this example we've explicitly declared it using @DiscriminatorValue annotation.

Notice that @DiscriminatorValue annotation is not specified for Person entity because it is an abstract entity not a concrete entity. The only difference between an abstract entity and a concrete entity is that it can’t be instantiated.

The following figure shows the generated database table that maps our example entities using the “Single Table per Class Hierarchy” strategy.



This table contains all the attributes which are specified in our example class hierarchy. ID, ADRESS and NAME come from parent class Person, SALARY from Employee and CREDITRATING from Customer classes. It is having one extra attribute DISC which we declared using @DiscriminatorColumn annotation. This field in the table will contain different values, depending on the type of object being persisted to the table.

Let us assume that we have created and persisted Customer and Employee entities in the database. The data is as follows:

Customer:

Id: 1111

Name: Srinivas

Address: Chennai, India

Credit Rating: GOOD

Employee:

Id: 2222

Name: Prabhakar

Address: Hyderabad, India

Salary: 25000

The following table shows the records that are inserted into the PERSON_SINGLE table as a result of persisting Customer and Employee entities with the above sample data.

Table: PERSON_SINGLE

	ID	ADDRESS	NAME	DISC	SALARY	CREDITRATING
	1111	Chennai, India	Srinivas	CUSTOMER	(NULL)	GOOD
	2222	Hyderabad, India	Prabhakar	EMPLOYEE	25000	(NULL)
*		(NULL)	(NULL)		(NULL)	(NULL)

You can observe that the record representing Customer has NULL value for SALARY field since it not having this attribute. Similarly the record representing Employee has NULL value for CREDITRATING field.

The advantage of this strategy is that it is very efficient and supports polymorphism. Since all classes are mapped to a single table, no table joins are required by Persistence framework.

The disadvantage is that the mapping table must have a column for each attribute in the hierarchy and every field related to a subclass must be null-able.

Joined Subclass

The joined subclass strategy uses one-to-one relationships to map the object oriented inheritance. It represents the class hierarchy by multiple tables. It will have a separate table for each entity class in the hierarchy. The table representing the super class contains primary key, columns for all other attributes of the super class and an extra discriminator column. The table representing a sub class contains columns for the attributes specific to the sub class and a foreign key to the parent table (i.e. the table representing the super class).

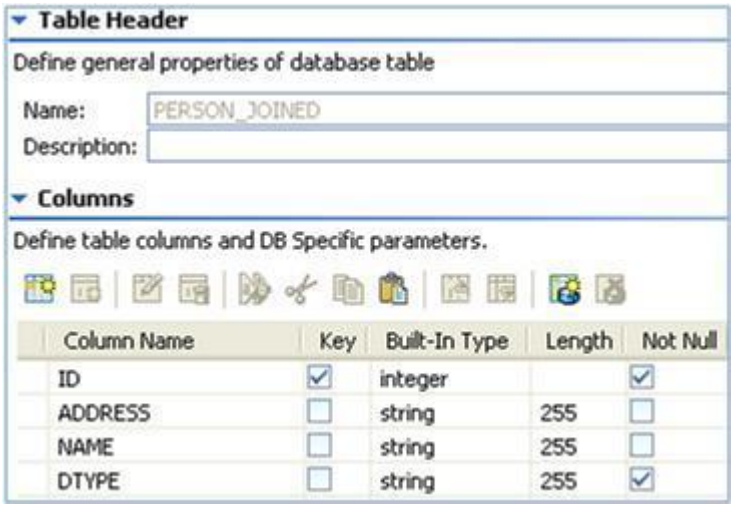
Following listings show how the entities in the **Person** hierarchy are mapped using the “Joined Subclass” inheritance strategy.

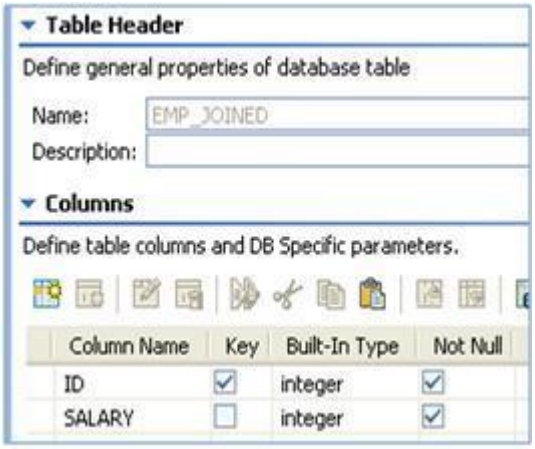
```
@Entity(name="PersonJoined")
@Table(name="PERSON_JOINED")
@Inheritance(strategy=JOINED)
public abstract class Person implements Serializable {
   @Id
   private int id;
   private String name;
   private String address;
   private static final long serialVersionUID = 1L;
   public Person() {
    super();
}

 //setters and getters go here …
}


@Entity(name="EmployeeJoined")
@Table(name="EMP_JOINED")
public class Employee extends Person implements Serializable {

   private int salary;
   private static final long serialVersionUID = 1L;
   public Employee() {
    super();
   }
   // setters and getters go here ...

}
```

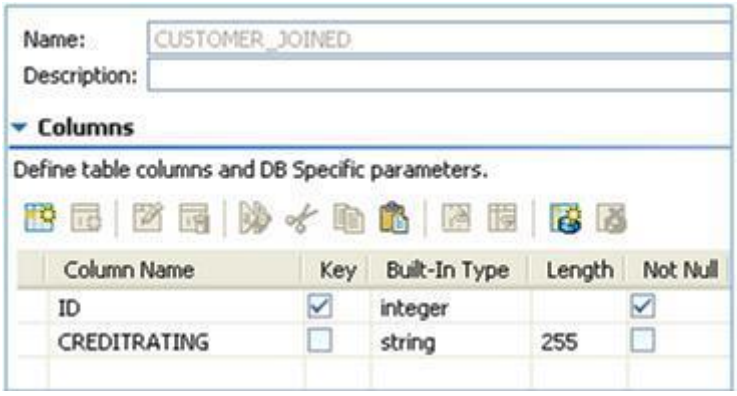
```
@Entity(name="CustomerJoined")
@Table(name="CUSTOMER_JOINED")
public class Customer extends Person implements Serializable {

   private String creditRating;
   private static final long serialVersionUID = 1L;

   public Customer() {
    super();
   }
   // setters and getters go here ...
}
```

This time we have changed the inheritance strategy to JOINED in superclass person. We have suffixed the entity and table names with Joined strategy to differentiate from that of previous example using @Entity and @Table annotations respectively. We have removed the @DicriminatorColumn annotation from the root class Person and @DiscriminatorValue annotation from the sub classes.

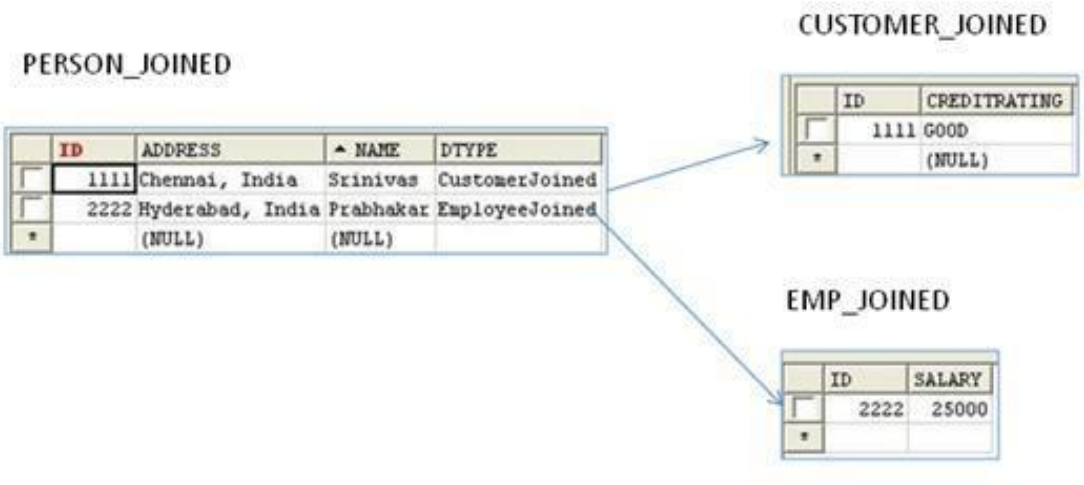
Following figure illustrates the schema that maps our entities using the JOINED inheritance strategy.







For the same sample data of previous section, following figure shows the records inserted into the database tables PERSON_JOINED, CUSTOMER_JOINED and EMP_JOINED.



The disadvantage of this approach is that table joins are required to get all the properties of a sub class.

This strategy supports polymorphism. This is an excellent approach when your class hierarchy is not too deep.

Table per Concrete Class

In this strategy, every concrete class in the hierarchy has its own table and no relationships exist between any of the tables. Each table contains all of the properties found in the inheritance chain up to the parent class. Since this strategy is not supported by SAP JPA 1.0 implementation we will not go into the details of it.

Related Content

Getting Started with Java Persistence API and SAP JPA 1.0

Basics of the Java Persistence API – Understanding the Entity Manager

Basics of the Java Persistence API – Defining and Using Relationships

SAP NetWeaver Composition Environment 7.1

Java Development

SAP Library – Developing Java EE Applications

Disclaimer and Liability Notice

This document may discuss sample coding or other information that does not include SAP official interfaces and therefore is not supported by SAP. Changes made based on this information are not supported and can be overwritten during an upgrade.

SAP will not be held liable for any damages caused by using or misusing the information, code or methods suggested in this document, and anyone using these methods does so at his/her own risk.

SAP offers no guarantees and assumes no responsibility or liability of any type with respect to the content of this technical article or code sample, including any liability resulting from incompatibility between the content within this document and the materials and services offered by SAP. You agree that you will not hold, or seek to hold, SAP responsible or liable with respect to the content of this document.