Entity Inheritance

Entities support class inheritance, polymorphic associations, and polymorphic queries. Entity classes can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

The `roster` example application demonstrates entity inheritance, as described in Entity Inheritance in the `roster` Application.

Abstract Entities

An abstract class may be declared an entity by decorating the class with `@Entity`. Abstract entities are like concrete entities but cannot be instantiated.

Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity:

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information but are not entities. That is, the superclass is not decorated with the `@Entity` annotation and is not mapped as an entity by the Java Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the annotation `javax.persistence.MappedSuperclass`:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

Mapped superclasses cannot be queried and can't be used in `EntityManager` or `Query` operations. You must use entity subclasses of the mapped superclass in `EntityManager` or `Query` operations. Mapped superclasses can't be targets of entity relationships. Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from

the mapped superclass define the table mappings. For instance, in the preceding code sample, the underlying tables would be `FULLTIMEEMPLOYEE` and `PARTTIMEEMPLOYEE`, but there is no`EMPLOYEE` table.

Non-Entity Superclasses

Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete. The state of non-entity superclasses is nonpersistent, and any state inherited from the non-entity superclass by an entity class is nonpersistent. Non-entity superclasses may not be used in`EntityManager` or `Query` operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

Entity Inheritance Mapping Strategies

You can configure how the Java Persistence provider maps inherited entities to the underlying datastore by decorating the root class of the hierarchy with the annotation `javax.persistence.Inheritance`. The following mapping strategies are used to map the entity data to the underlying database:

- A single table per class hierarchy

- A table per concrete entity class

- A "join" strategy, whereby fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

The strategy is configured by setting the `strategy` element of `@Inheritance` to one of the options defined in the `javax.persistence.InheritanceType`enumerated type:

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

The default strategy, `InheritanceType.SINGLE_TABLE`, is used if the`@Inheritance` annotation is not specified on the root class of the entity hierarchy.

The Single Table per Class Hierarchy Strategy

With this strategy, which corresponds to the default`InheritanceType.SINGLE_TABLE`, all classes in the hierarchy are mapped to a single table in the database. This table has a **discriminator column**containing a value that identifies the subclass to which the instance represented by the row belongs.

The discriminator column, whose elements are shown in Table 32-2, can be specified by using the `javax.persistence.DiscriminatorColumn`annotation on the root of the entity class hierarchy.

Table 32-2 `@DiscriminatorColumn` Elements

Type	Name	Description
String	name	The name of the column to be used as the discriminator column. The default is `DTYPE`. This element is optional.
DiscriminatorType	discriminatorType	The type of the column to be used as a discriminator column. The default is`DiscriminatorType.STRING`. This element is optional.
String	columnDefinition	The SQL fragment to use when creating the discriminator column. The default is generated by the Persistence provider and is implementation-specific. This element is optional.
String	length	The column length for `String`-based discriminator types. This element is ignored for non-`String` discriminator types. The default is 31. This element is optional.

The `javax.persistence.DiscriminatorType` enumerated type is used to set the type of the discriminator column in the database by setting the`discriminatorType` element of `@DiscriminatorColumn` to one of the defined types. `DiscriminatorType` is defined as:

```
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
```

If @DiscriminatorColumn is not specified on the root of the entity hierarchy and a discriminator column is required, the Persistence provider assumes a default column name of DTYPE and column type of DiscriminatorType.STRING.

The javax.persistence.DiscriminatorValue annotation may be used to set the value entered into the discriminator column for each entity in a class hierarchy. You may decorate only concrete entity classes with @DiscriminatorValue.

If @DiscriminatorValue is not specified on an entity in a class hierarchy that uses a discriminator column, the Persistence provider will provide a default, implementation-specific value. If the discriminatorType element of @DiscriminatorColumn is DiscriminatorType.STRING, the default value is the name of the entity.

This strategy provides good support for polymorphic relationships between entities and queries that cover the entire entity class hierarchy. However, this strategy requires the columns that contain the state of subclasses to be nullable.

The Table per Concrete Class Strategy

In this strategy, which corresponds to InheritanceType.TABLE_PER_CLASS, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class's table in the database.

This strategy provides poor support for polymorphic relationships and usually requires either SQL UNION queries or separate SQL queries for each subclass for queries that cover the entire entity class hierarchy.

Support for this strategy is optional and may not be supported by all Java Persistence API providers. The default Java Persistence API provider in the GlassFish Server does not support this strategy.

The Joined Subclass Strategy

In this strategy, which corresponds to InheritanceType.JOINED, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that contains only those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table.

This strategy provides good support for polymorphic relationships but requires one or more join operations to be performed when instantiating entity subclasses. This may result in poor performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.

Some Java Persistence API providers, including the default provider in the GlassFish Server, require a discriminator column that corresponds to the root entity when using the joined subclass strategy. If you are not using automatic table creation in your application, make sure that the database table is set up correctly for the discriminator column defaults, or use the @DiscriminatorColumn annotation to match your database schema. For information on discriminator columns, see The Single Table per Class Hierarchy Strategy.