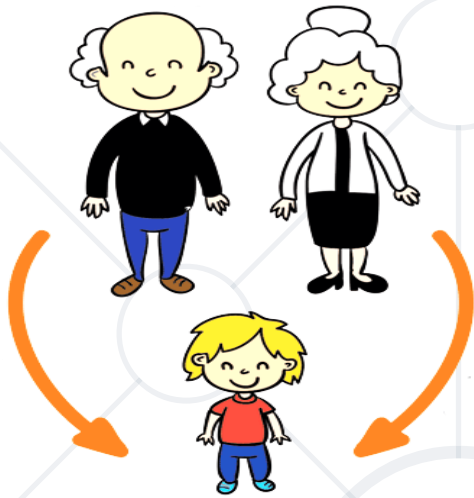


Hibernate (JPA) Code First Entity Relations

Advanced Mapping



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.org/>

Table of Contents

1. Java Persistence API Inheritance.
2. Table Relations.



sli.do

#



Java Persistence API Inheritance

Fundamental Inheritance Concepts

- Inheritance is a fundamental concept in most programming languages
 - SQL does not support this kind of relationships
- Implemented by any JPA framework by **inheriting** and **mapping Entities**

- Implemented by the `javax.persistence.Inheritance` annotation
- 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**
 - **"Join"** strategy – mapping common fields in a single table

- **Table creation for each entity**
 - A table defined for each concrete class in the inheritance
 - Allows inheritance to be used in the object model, when it does not exist in the data model
- Querying root or branch classes can be very difficult and **inefficient**

Table Per Class Strategy: Example (1)

Vehicle.java

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicle {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private long id;
    @Basic
    private String type;
    protected Vehicle() {}
    protected Vehicle(String type) {
        this.type = type;
    }
}
```

Inheritance type

A table generator is used for each table

Table Per Class Strategy: Example (2)

Bike.java

```
@Entity
@Table(name = "bikes")
public class Bike extends Vehicle {
    private final static String type = "BIKE";
    public Bike(){
        super(type);
    }
}
```

Table Name

Car.java

```
@Entity
@Table(name = "cars")
public class Car extends Vehicle {
    private final static String type = "CAR";
    public Car(){
        super(type);
    }
}
```

Table Name



Table Per Class Strategy: Example (3)

Main.java

```
..  
Vehicle bike = new Bike();  
Vehicle car = new Car();  
  
em.persist(bike);  
em.persist(car);
```

■ Result:

bikes	
id	type
1	"BIKE"

cars	
id	type
2	"CAR"

Table Per Class Strategy: Conclusion

- **Disadvantages:**

- Repeating information in each table
- Changes in super class involves changes in all subclass tables
- No foreign keys involved (unrelated tables)

- **Advantages:**

- No NULL values – no unneeded fields
- Simple style to implement inheritance mapping



Table Per Class: Joined

- Table is defined for each class in the inheritance hierarchy
 - Storing of that class **only the local attributes**
 - Each table must store object's **primary key**



Table Per Class Strategy: Example (1)

Vehicle.java

```
@Entity
@Table(name = "vehicles")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Vehicle {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;

    @Basic
    private String type;

    protected Vehicle() {}
    protected Vehicle(String type) {
        this.type = type;
    }
}
```

Inheritance type

A table generator is used for each table

Table Per Class Strategy: Example (2)

TransportationVehicle.java

@MappedSuperclass

```
public abstract class TransportationVehicle extends Vehicle {  
    private int loadCapacity;  
    public TransportationVehicle(){ }  
    public TransportationVehicle(String type,int loadCapacity) {  
        super(type);  
        this.loadCapacity = loadCapacity;  
    }  
  
    // Getters and setters  
}
```

Table Per Class Strategy: Example (3)

PassengerVehicle.java

@MappedSuperclass

```
public abstract class PassengerVehicle extends Vehicle {  
    private int numOfPassengers;  
  
    public PassengerVehicle() { }  
    public PassengerVehicle(String type,int numOfPassengers) {  
        super(type);  
        this.numOfPassengers = numOfPassengers;  
    }  
  
    // Getters and setters  
}
```

Table Per Class Strategy: Example (4)

Truck.java

```
@Entity
public class Truck extends TransportationVehicle {
    private final static String type = "TRUCK";
    private int numOfContainers;
    public Truck(){ }
    public Truck(String type, int numOfContainers,
int loadCapacity) {
        super(type,loadCapacity);
        this.numOfContainers = numOfContainers; }
    // Getters and setters
}
```

Car.java

```
@Entity
public class Car extends PassengerVehicle {
    private final static String type = "CAR";

    public Car(){ }

    public Car(String type,int numOfPassengers){
        super(type, numOfPassengers);
    }
    // Getters and setters
}
```



Results – Joined Strategy

- After persist:

cars	
id	numOfPassengers
1	2

vehicles	
id	type
1	CAR
2	TRUCK

trucks		
id	numOfContainers	loadCapacity
1	2	5

Results – Joined Strategy

- **Disadvantages:**

- Multiple JOINS - for deep hierarchies it may give poor performance

- **Advantages:**

- No NULL values
- No repeating information
- Foreign keys involved
- Reduced changes in schema on superclass changes



- **Simplest** and typically the best performing and best solution
 - A single table is used to store all the instances of the **entire inheritance hierarchy**
 - A column for every attribute of every class
 - A **discriminator column** is used to determine to which class the particular row belongs to

SINGLE TABLE: Example

Vehicle.java

```
@Entity
@Table(name = "vehicles")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type")
public abstract class Vehicle {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    @Basic
    @Column(insertable = false, updatable = false)
    private String type;
    protected Vehicle() {}
    protected Vehicle(String type) {
        this.type = type;
    }
}
```

Inheritance type

A table generator is used for each table

Table Per Class Strategy: Example (1)

TransportationVehicle.java

@MappedSuperclass

```
public abstract class TransportationVehicle extends Vehicle {  
    private int loadCapacity;  
    public TransportationVehicle() { }  
    public TransportationVehicle(String type, int loadCapacity) {  
        super(type);  
        this.loadCapacity = loadCapacity;  
    }  
    // Getters and setters  
}
```

Table Per Class Strategy: Example (2)

PassengerVehicle.java

```
@MappedSuperclass
public abstract class PassengerVehicle extends Vehicle {
    private int noOfpassengers;

    public PassengerVehicle() { }

    public PassengerVehicle(String type, int noOfpassengers) {
        super(type);
        this.noOfpassengers = noOfpassengers;
    }

    // Getters and setters
}
```

Table Per Class Strategy: Example (3)

Truck.java

```
@Entity
@DiscriminatorValue(value = "truck")
public class Truck extends TransportationVehicle {
    private final static String type = "TRUCK";
    private int noOfContainers;

    // Constructors
    // Getters and setters
}
```

Car.java

```
@Entity
@DiscriminatorValue(value = "car")
public class Car extends PassengerVehicle {
    private final static String type = "CAR";
    public Car() { }
    public Car(){
        super(type);
    }
}
```



Results – Joined Strategy

- After persist:

vehicles				
id	type	loadCapacity	noOfPassengers	noOfContainers
1	truck
2	car

Discriminator column



Table Relations

One-to-One, One-to-Many, Many-to-Many

- There are several types of database relationships:
 - **One to One** Relationships
 - **One to Many** and **Many to One** Relationships
 - **Many to Many** Relationships
 - **Self Referencing** Relationships

One-To-One – Unidirectional (1)



One-To-One – Unidirectional (2)

BasicShampoo.java

```
@Entity
@Table(name = "shampoos")
public abstract class BasicShampoo implements Shampoo {

//...

    @OneToOne(optional = false)
    @JoinColumn(name = "label_id",
        referencedColumnName = "id")
    private BasicLabel label;

//...
}
```

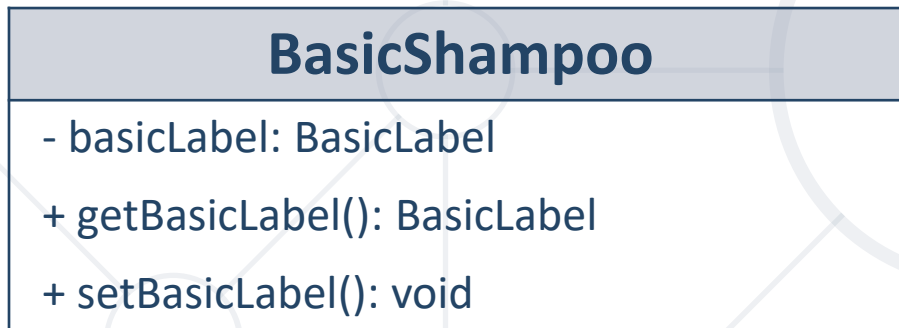
One-To-One relationship

Runtime evaluation

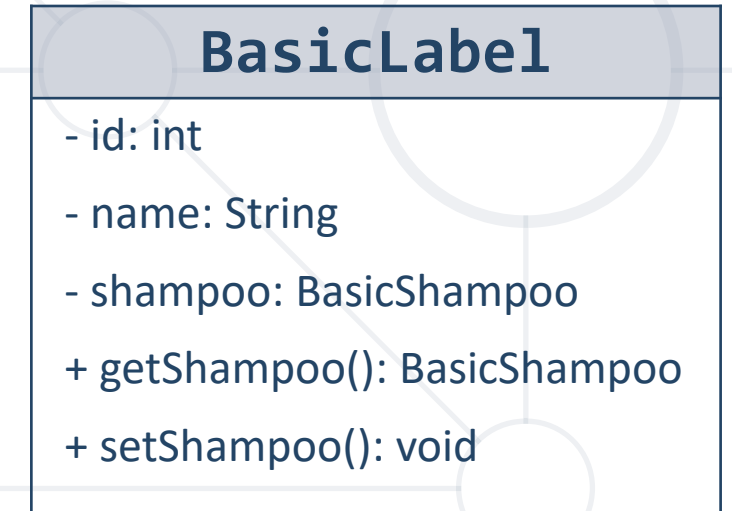
Column name in
table shampoos

Column name in
table label

One-To-One – Bidirectional (1)



One-to-one



One-To-One – Bidirectional (2)

BasicLabel.java

```
@Entity
@Table(name = "labels")
public class BasicLabel implements Label{
//...
```

Field in entity BasicShampoo

```
@OneToOne(mappedBy = "label",
targetEntity = BasicShampoo.class)
private BasicShampoo basicShampoo;
```

Entity for the mapping

```
//...
}
```

Many-To-One – Unidirectional (1)



Many-To-One – Unidirectional (2)

BasicShampoo.java

```
@Entity
@Table(name = "shampoos")
public abstract class BasicShampoo implements Shampoo {
```

```
//...
```

Many-To-One relationship

Runtime evaluation

```
@ManyToOne(optional = false)
@JoinColumn(name = "batch_id", referencedColumnName = "id")
private ProductionBatch batch;
```

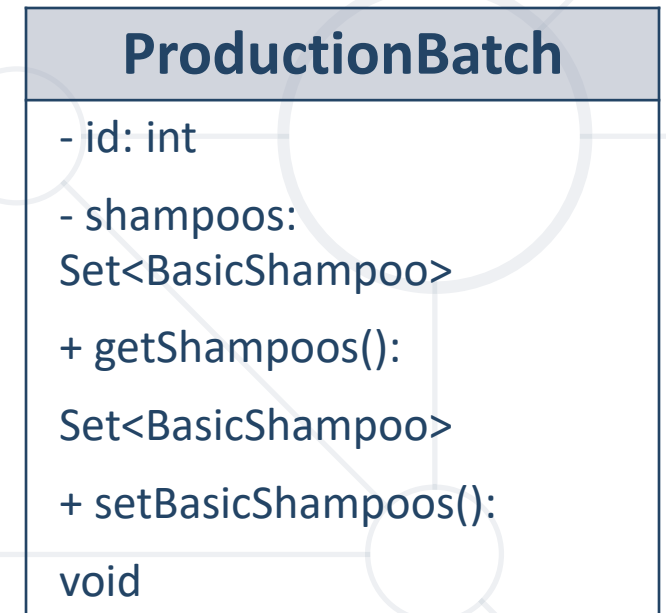
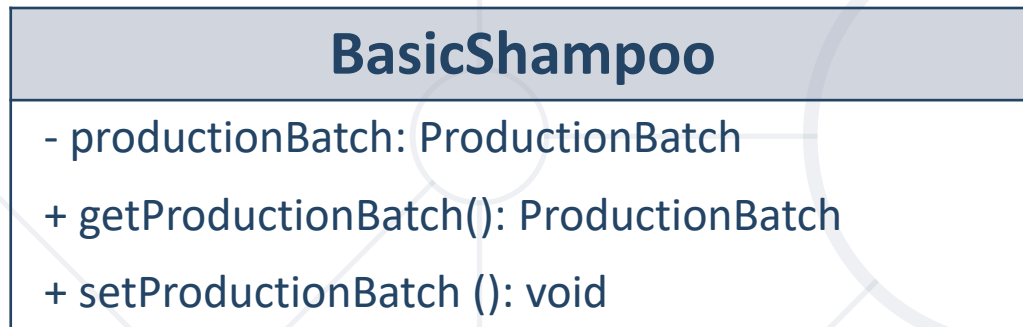
```
//...
```

```
}
```

Column name in
table shampoos

Column name in
table batches

One-To-Many – Bidirectional (1)



One-To-Many – Bidirectional (2)

ProductionBatch.java

```
@Entity
@Table(name = "batches")
public class ProductionBatch implements Batch {
    //...

    @OneToMany(mappedBy = "batch", targetEntity = BasicShampoo.class,
               fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    private Set<Shampoo> shampoos;

    //...
}
```

Field in entity BasicShampoo

Entity for the mapping

Fetching type

Cascade type

Many-To-Many – Unidirectional

BasicShampoo.java

```
@Entity
@Table(name = "shampoos")
public abstract class BasicShampoo implements Shampoo {

    //...
    @ManyToMany
    @JoinTable(name = "shampoos_ingredients",
        joinColumns = @JoinColumn(name = "shampoo_id", referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "ingredient_id",
            referencedColumnName = "id"))
    private Set<BasicIngredient> ingredients;

    //...
}
```

Many-To-Many relationship

Mapping table

Column in shampoos

Column in ingredients

Column in mapping table

Many-To-Many – Bidirectional

BasicIngredient.java

```
@Entity
@Table(name = "ingredients")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type", discriminatorType = DiscriminatorType.STRING)
public abstract class BasicIngredient implements Ingredient {
    //...

    @ManyToMany(mappedBy = "ingredients",
        targetEntity = BasicShampoo.class)
    private Set<BasicShampoo> shampoos;

    //...
}
```

Field in entity BasicShampoo

Entity for the mapping

- Fetching – retrieve objects from the database
 - Fetched entities are stored in the **Persistence Context** as cache
- Retrieval of an entity object might cause automatic retrieval of **additional** entity objects

- Fetching Strategies
 - EAGER – retrieves all entity objects reachable through fetched entity
 - Can cause **slowdown** when used with a big data source
 - **LAZY** – retrieves all reachable entity objects **only when fetched entity's getter method is called**

```
University university = em.find((long) 1); // collection students = null  
  
// The collection holding the students is populated when the getter is called  
university.getStudents();
```

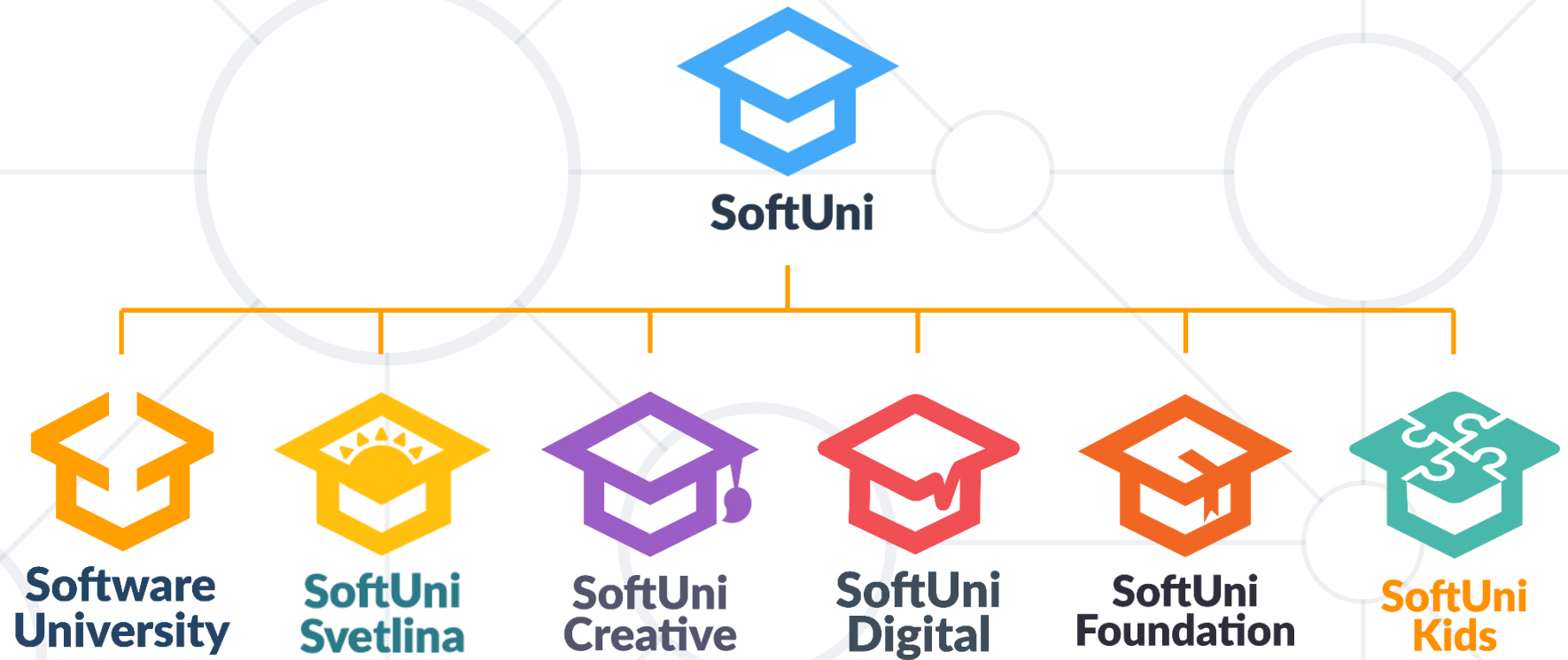
- JPA translates **entity state transitions** to database **DML** statements
 - This behavior is configured through the **CascadeType** mappings
- **CascadeType.PERSIST**: means that `save()` or `persist()` operations cascade to related entities
- **CascadeType.MERGE**: means that related entities are merged into managed state when the owning entity is merged
- **CascadeType.REFRESH**: does the same thing for the `refresh()` operation

- **CascadeType.REMOVE**: removes all related entities association with this setting when the owning entity is deleted
- **CascadeType.DETACH**: detaches all related entities if a "manual detach" occurs
- **CascadeType.ALL**: is shorthand for all of the above cascade operations

- Relational databases don't support inheritance
- It is implemented by JPA:
 - **SINGLE_TABLE**
 - **TABLE_PER_CLASS**
 - **JOINED**
- Table relations are Un/Bidirectional
- **One-to-One**
- **Many-to-One**
- **Many-to-Many**



Questions?



SoftUni Diamond Partners



SCHWARZ



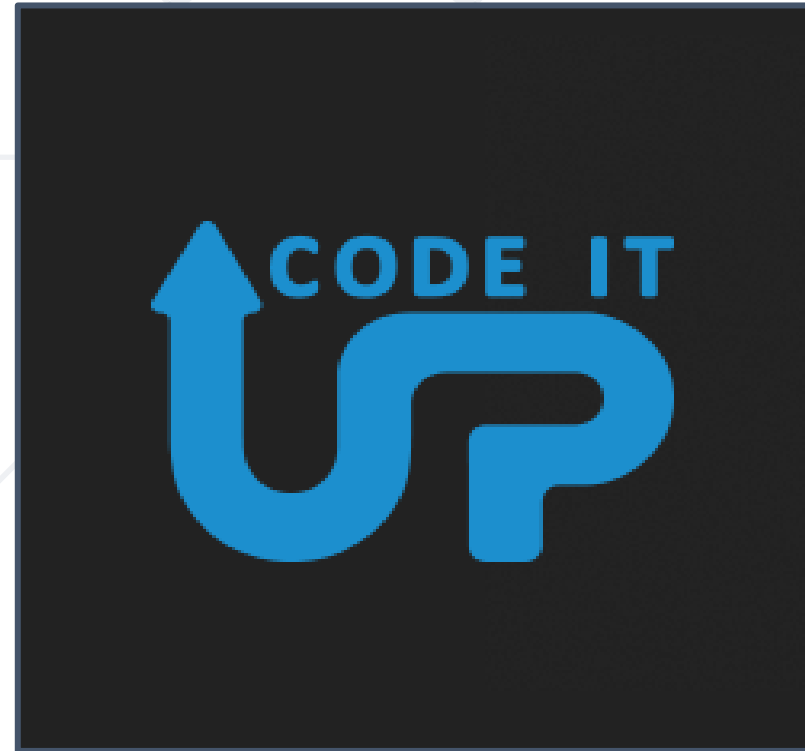
**SUPER
HOSTING
.BG**



INDEAVR
Serving the high achievers

Bosch.io





Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- SoftUni Global
 - softuni.org



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>
- © SoftUni Global – <https://softuni.org>

