Don't use unidirectional one-to-many associations

In your table model, you normally use a foreign key column on the to-many side of the association to store a reference to the associated record. Hibernate uses the same approach when you model a bidirectional one-to-many or an unidirectional many-to-one relationship. It uses the foreign key column to map the association.

But it can't do that if you don't model the relationship on the entity, which represents the to-many side of the relationship. So, Hibernate introduces an association table to store the foreign keys.

You can avoid this table if you specify the foreign key column with a *@JoinColumn* annotation. This column has to be part of the table of the to-many side of the association.

```
@Entity
public class PurchaseOrder {

    @OneToMany
    @JoinColumn(name = "fk_order")
    private Set<Item> items = new HashSet<Item>();

    ...
}
```

Hibernate now uses the foreign key column instead of an association table to map the relationship. But it still has to perform an additional SQL UPDATE statement to set the foreign key because the Item entity doesn't map the foreign key column.

Avoid the mapping of huge to-many associations

Hibernate loads all associated entities when it initializes an association. That can take several seconds or even minutes when Hibernate has to fetch several thousand entities.

So, better use an unidirectional many-to-one association. You can't use the to-many mapping anyways, and it removes the risk that someone triggers the initialization by accident.

If you need to join the associated entities in a JPQL query, you can either use the mapped many-to-one association or a Hibernate-specific JOIN clause that doesn't require a mapped relationship.

Think twice before using CascadeType.Remove

Cascade remove is another feature that works well on small to-many associations. But it's very inefficient when it needs to remove a huge number of entities.

Hibernate needs to execute proper lifecycle transitions for all entities. So, Hibernate needs to select all associated *Item* entities and remove them one by one.

You can use a JPQL query to remove all *Item* entities with 1 statement. But please be aware that Hibernate will not call any *EntityListener*s for these entities, and it also doesn't remove them from any caches.

You can update the caches programmatically. The following code snippet shows an example that removes all entities from the first level cache before it calls a JPQL query to remove all *Item* entities associated to a given *Order* entity.

```
em.flush();
em.clear();


Query q = em.createQuery("DELETE Item i WHERE
i.order.id = :orderId");
q.setParameter("orderId", orderId);
q.executeUpdate();


order = em.find(PurchaseOrder.class, orderId);
```

Use orphanRemoval when modeling parent-child associations

The orphanRemoval feature can make it very comfortable to remove a child entity. You can use it for parent-child relationships in which a child entity can't exist without its parent entity.

When you set the *orphanRemoval* attribute of the *@OneToMany* annotation to *true* and the *cascade* attribute to *CascadeType.ALL*, Hibernate automatically removes a child entity when you remove it from the association.

```
@Entity
public class PurchaseOrder {


    @OneToMany(mappedBy = "order",
                        orphanRemoval = true)
    private List<Item> items = new ArrayList<Item>();
}
```

Implement helper methods to update bi-directional associations

When you add an entity to or remove it from a bi-directional association, you need to perform the operation on both ends.

That is an error-prone task. You should, therefore, provide helper methods that implement this logic.

```
@Entity
public class PurchaseOrder {

    ...

    public void addItem(Item item) {
        this.items.add(item);
        item.setOrder(this);
    }
}
```

Define FetchType.LAZY for @ManyToOne association

The JPA specification defines FetchType.EAGER as the default for to-one relationships. It tells Hibernate to initialize the association, when it loads the entity.

That becomes a problem when you load multiple entities. Hibernate then needs to perform an additional query for each of the selected entities. That is often called a n+1 select issue.

You can avoid that by setting the *FetchType* on the *@ManyToOne* annotation to *LAZY*.

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "fk_order")
    private PurchaseOrder order;

    ...
}
```

And if you need the to-one association in your use case, you can use a *JOIN FETCH* clause or one of the other options to initialize lazy relationships.

```
List<Item> items = em.createQuery(
                "SELECT i FROM Item i JOIN FETCH i.order",
                Item.class)
        .getResultList();
```