The set of managed entity instances within an entity manager at any

given time is called its *persistence context*.

Transaction Review

A transaction is an abstraction that is used to group together a series of operations. Once grouped together, the set of operations is treated as a single unit, and all of the operations must succeed or none of them can succeed. The consequence of only some of the operations being successful would produce an inconsistent view of the data that would be harmful or undesirable to the application. The term used to describe whether the operations succeed together or not at all is called *atomicity* and is arguably the most important of the four basic properties that are used to characterize how transactions behave. Understanding these four properties is fundamental to understanding transactions. The following list summarizes these properties:

• *Atomicity*: Either all the operations in a transaction are successful or none of them is. The success of every individual operation is tied to the success of the entire group.

• *Consistency*: The resulting state at the end of the transaction adheres to a set of rules that define acceptability of the data. The data in the entire system is legal or valid with respect to the rest of the data in the system.

• *Isolation*: Changes made within a transaction are visible only to the transaction that is making the changes. Once a transaction commits the changes, they are atomically visible to other transactions.

• *Durability*: The changes made within a transaction endure beyond the completion of the transaction.

A transaction that meets all these requirements is said to be an ACID transaction (the familiar ACID term being obtained by combining the first letter of each of the four properties). Not all transactions are ACID transactions, and those that are often offer some flexibility in the fulfillment of the ACID properties. For example, the isolation level is a common setting that can be configured to provide either looser or tighter degrees of isolation than what was described earlier. They are typically done for reasons of either increased performance or, on the other side of the spectrum, if an application has more stringent data consistency requirements. The transactions that we discuss in the context of Java EE are normally of the ACID variety.

@Entity  
@Table(name = "USERS")  
public class User implements Serializable {  
  
@Id @GeneratedValue @Column(name = "ID", nullable = false)  
private Long id;  
  
...  
  
@ManyToMany(mappedBy = "friendOf")  
@JoinTable(name = "FRIENDS")  
private Set<User> myfriends = new HashSet<User>();  
  
@ManyToMany()  
@JoinTable(name = "FRIENDS")  
private Set<User> friendOf = new HashSet<User>();  
  
  
  
@ManyToMany(mappedBy = "partOfFamily")  
@JoinTable(name = "FAMILY")  
private Set<User> myfamily = new HashSet<User>();  
  
  
@ManyToMany()  
@JoinTable(name = "FAMILY")  
private Set<User> partOfFamily = new HashSet<User>();

Every relationship needs that one of the entities be the “relationship owner”. Being the relationship owner is nothing more than to have the foreign key in the database table. In the code above it is possible to see that the annotation @JoinColumn has been used. This annotation will indicate to the JPA that the foreign key will be located in the person database table, making the Person entity owner of the relationship.

To transform this relationship in a bidirectional it is necessary just to edit the Cellular entity. Check the class below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import javax.persistence.\*;    @Entity  public class Cellular {        @Id      @GeneratedValue      private int id;        private int number;        @OneToOne(mappedBy="cellular")      private Person person;        // get and set  } |

About the code above:

* The same annotation @OneToOne Person is used in the Cellular entity.
* The parameter “*mappedBy*” was used in the @OneToOne. This parameter indicates that the entity Person is the owner of the relationship; the foreign key must exist inside the person table, and not the Cellular table.

A developer must have in mind that to the JPA works correctly it is a good practice just to leave one side of the relationship as owner. If the annotation @OneToOne found in the Cellular entity was without “mappedBy”, the JPA would handle the Cellular entity as the owner of the relationship too. It is not a good idea to leave either sides of relationship without mappedBy, or both with mappedBy.

Let us start with the ManyToOne side:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import javax.persistence.\*;    @Entity  public class Call {        @Id      @GeneratedValue      private int id;        @ManyToOne      @JoinColumn(name = "cellular\_id")      private Cellular cellular;        private long duration;        // get and set  } |

About the code above:

* The annotation @ManyToOne was used.
* Notice that the annotation @JoinColumn was used to define who the owner of the relationship is.
* The @ManyToOne side will always be the owner of the relationship. There is no way of using the mappedBy attribute inside the @ManyToOne annotation.

To do a bidirectional relationship we need to edit the Cellular entity (created in the last page). Check the code below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import javax.persistence.\*;    @Entity  public class Cellular {        @Id      @GeneratedValue      private int id;        @OneToOne(mappedBy = "cellular")      private Person person;        @OneToMany(mappedBy = "cellular")      private List<Call> calls;        private int number;        // get and set  } |

About the code above:

* The @OneToMany annotation was used. This annotation must be placed in a collection.
* The mappedBy was used to define the Call entity as the relationship owner.

Every relationship needs that one of the entities be the “relationship owner”. Being the relationship owner is nothing more than to have the foreign key in the database table. In the first code it is possible to see that the annotation @JoinColumn has been used. This annotation will indicate to the JPA that the foreign key will be located in the call database table, making the Call entity owner of the relationship.

Many-To-Many

Check the Person entity below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.util.List;    import javax.persistence.\*;    @Entity  public class Person {        @Id      @GeneratedValue      private int id;        private String name;        @ManyToMany      @JoinTable(name = "person\_dog", joinColumns = @JoinColumn(name = "person\_id"), inverseJoinColumns = @JoinColumn(name = "dog\_id"))      private List<Dog> dogs;        @OneToOne      @JoinColumn(name = "cellular\_id")      private Cellular cellular;        // get and set  } |

About the code above:

* The @ManyToMany annotation was used.
* The @JoinTable annotation was used to set the relationship table between the entities; “name” will set the table name; “joinColumn” will define the column name in the table of the relationship owner; “inverseJoinColumns” will define the column name in the table of the nonrelationship owner.

The Person entity has unidirectional relationship with Dog. Check how the Dog entity would look like in a bidirectional relationship:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import java.util.List;    import javax.persistence.\*;    @Entity  public class Dog {        @Id      @GeneratedValue      private int id;        private String name;        @ManyToMany(mappedBy="dogs")      private List<Person> persons;        // get and set  } |

It is possible to find the @ManyToMany annotation the mappedBy option, it will define the Person entity as the relationship owner.

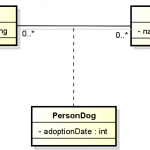
Every relationship needs that one of the entities be the “relationship owner”. For the @ManyToMany annotation the relationship owner is the one with the mappedBy option in the other relation. If the mappedBy option is not found in none of the entities the JPA will define both classes of the ManyToMany relationship as the owner. The mappedBy must point to the attribute name and not to the attribute entity name.

**ManyToMany with extra fields**

Imagine a Person entity that has a ManyToMany relationship with the Dog entity; every time that a dog is adopted by a person the application should register the adoption date. This value must be stored in the relationship and not in the attribute person or dog.

To this kind of situation there is an approach named Associative Class or Associative Entity. With this approach it is possible to store extra data when the ManyToMany relationship is created.

The image below show how this entity can be mapped:

[](http://uaihebert.com/?attachment_id=1642)

To map this extra field, to like the following code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import java.util.List;    import javax.persistence.\*;    @Entity  public class Person {        @Id      @GeneratedValue      private int id;        private String name;        @OneToMany(mappedBy = "person")      private List<PersonDog> dogs;        // get and set  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import java.util.List;    import javax.persistence.\*;    @Entity  public class Dog {        @Id      @GeneratedValue      private int id;        private String name;        @OneToMany(mappedBy = "dog")      private List<PersonDog> persons;        // get and set  } |

The code above is using the @OneToMany relationship with the mappedBy option. Notice that there is not a @ManyToMany relationship between the entities, but there is an entity PersonDog that unite both entities.

Below is the PersonDog code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.util.Date;    import javax.persistence.\*;    @Entity  @IdClass(PersonDogId.class)  public class PersonDog {        @Id      @ManyToOne      @JoinColumn(name="person\_id")      private Person person;        @Id      @ManyToOne      @JoinColumn(name="dog\_id")      private Dog dog;        @Temporal(TemporalType.DATE)      private Date adoptionDate;        // get and set  } |

In the code above it is possible to find the relationships Dog and Person, and an extra attribute to store the adoption date. There is an id class to store the relationship ids PersonDogId:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | import java.io.Serializable;    public class PersonDogId implements Serializable {        private static final long serialVersionUID = 1L;        private int person;      private int dog;        public int getPerson() {          return person;      }        public void setPerson(int person) {          this.person = person;      }        public int getDog() {          return dog;      }        public void setDog(int dog) {          this.dog = dog;      }        @Override      public int hashCode() {          return person + dog;      }        @Override      public boolean equals(Object obj) {          if(obj instanceof PersonDogId){              PersonDogId personDogId = (PersonDogId) obj;              return personDogId.dog == dog && personDogId.person == person;          }            return false;      }  } |

A curiosity is that in the PersonDogId attributes has the same name of Dog and Person attributes found in the PersonDog entity. The name must be the same of the attributes, it is how the JPA works.