JPA: Java Persistence API

- JavaBeans synchronized with the database
- Other names: Hibernate, EJB3

JPA Object

```
@Entity
public class Wine  implements java.io.Serializable {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id = null;
        private String wineName;

        public Long getId() {  // getters – setters, generated by Eclipse
                return id;
        }
        public void setId(Long id) {
                this.id = id;
        }
        public String getWineName () {
                return wineName;
        }
}
```


Objet JPA with non-persistent attributes

```
@Entity
public class Wine  implements java.io.Serializable {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id = null;
        private String wineName;

        transient private String flag;  // not in the BD, not serialized

        @Transient
        private String caracteristic;                   // not in the BD

        //  . . . getters / setters . . .
}

// the transients are only used in JSF, not in the BD
```


Imports into a JPA

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

Created by clicking on the error marks

```
@Entity
public class Wine implements java.io.Serializable
{
  . . .
}
```

Introduce an object into the database

```
javax.persistence.EntityManager em = ejb3_utility.Manager.open();
javax.persistence.EntityTransaction tx = em.getTransaction();
      tx.begin();
        Wine wine = new Wine(nameW, grape, year);
        em.persist(wine);
        // the modifications done here are also introduced into the DB
      tx.commit();     // or tx.rollback();
ejb3_utility.Manager.close();

// persist adds automatically an id in the field @Id of the object
```


Create a Project Containing JPAs

See http://ltiwww.epfl.ch/~petitpie/InternetProgramming/JPA_Aux/JPA.html

Project with objects JPA

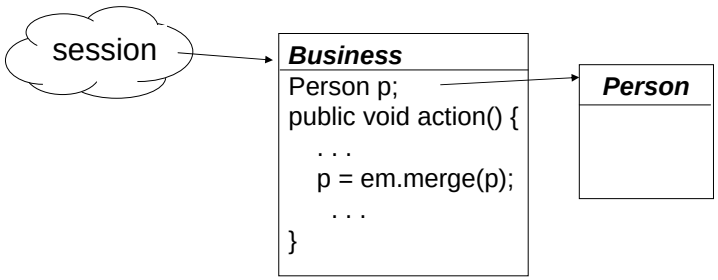
- Create a *Dynamic Web Project*
- Add the list of *.jar* in *WebContent/WEB-INF/lib*
- Add *ejb3_utility/Manager.java* in *src*
- Add *META-INF/persistence.xml* in the directory *src* and introduce the name ***of every JPA*** in this file

Persistence.xml (part)

Located in src/META-INF

```
<persistence-unit name="NameDeManager">
<class>db.Wine</class>
<properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.username" value="username"/>
      <property name="hibernate.connection.password" value="password"/>
      <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/test"/>
      <property name="javax.persistence.transactionType" value="RESOURCE_LOCAL"/>
  </properties>
 </persistence-unit>
</persistence>
```

Indirection in the session
(p remains in the session)



Some explanations for the previous slide

The bean *business* contains the actions as well as the Javabeans that keep the data.

This statement accesses an action from a JSP page: #{business.action}

This one accesses an attribute of *Person*: # {business.p.name}

Getters-setters are required for ***p*** like for the person attributes

In order to access a JavaBeans from an action, one just need to write (referring to the previous slide):

p.name

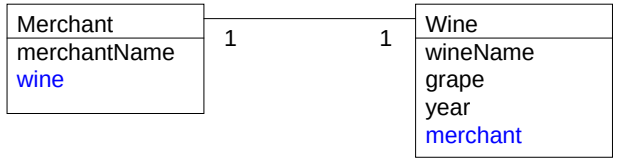
or

getP().getName()

The JavaBeans could also be stored directly in the session, but it would be more difficult to access them.

Relationships between JPAs

Relationships between JPA 1:1



In ***Merchant***
(*one must choose a side for mappedBy*)

```
@OneToOne(mappedBy="merchant")
private Wine wine;
```

```
@OneToOne
private Merchant merchant;
```

In ***Wine***

// Getters – setters for the two fields

Access to the elements of the relationship

In Java code: merchant.getWine().getYear)

wine.getMerchant().getMerchantName()

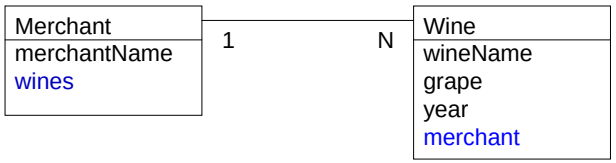
In a JSP: #{merchant.wine.year}

#{wine.merchant.merchantName}

Next slide: the collection must be introduced within a h:DataTable

Of course all these attributes must have getters-setters

Relationships between JPA 1:N



In ***Merchant***

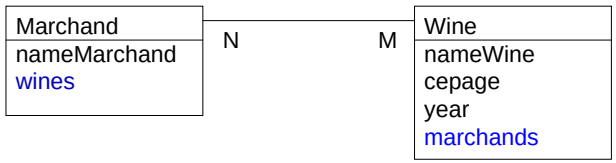
```
@OneToMany(mappedBy="merchant")
private Collection<Wine> wines;
```

```
@ManyToOne
private Merchant merchant;
```

In ***Wine***

// Getters – setters for the collection for the field merchant

Relationships between JPA N:M



In ***Merchant***

```
@ManyToMany
private Collection<Wine> wines;
```

```
@ManyToMany(mappedBy="wines")
private Collection marchands;
```

In ***Wine***

// Getters – setters for the two collections

Relationships between JPA N:M, cascade

In ***Merchant***

```
@ManyToMany (cascade = CascadeType.ALL)
private Collection<Wine> wines = new ArrayList<Wine>();
```

```
@ManyToMany(mappedBy="wines")
private Collection<Merchant> merchants = new ArrayList<Merchant>();
```

In ***Wine***

// Persistence and remove cascades

Insertion of JPAs in the relationships

Relation in the database and in the objects

It is the responsability of the developer to introduce the relationships in both directions between the objets.

In order for the relationship to be forwarded to the database, one must perform this operation within a transaction handled by the manager. Namely, the main data must be attached (see following) and the added object not free.

Moreover, one must introduce at least the element on the side that does not have the indication *mappedBy.*

Relationships entre JPA 1:1

```
@OneToOne(mappedBy="merchant")
private Wine wine;
```

```
@OneToOne
private Merchant merchant;
```

```
tx.begin();
merchant= em.merge(merchant);
em.persist(v);
merchant.setWine(v);
tx.commit();
```

```
tx.begin();
wine = em.merge(wine);
em.persist(m);
wine.setMerchant (m);
tx.commit();

// necessary and sufficient to
// introduce the relation into the DB
```


Relationships between JPA 1:N

```
@OneToMany(mappedBy="merchant")
private Collection<Wine> wines;
```

```
@ManyToOne
private Marchand marchand;
```

```
tx.begin();
merchant = em.merge(merchant);
em.persist(v);
merchant.wines.add(v);
tx.commit();
```

```
tx.begin();
wine = em.merge(wine);
em.persist(m);
wine.setMerchant(m);
tx.commit();

// necessary and sufficient to
// introduce the relation into the DB
```

Relationships entre JPA N:M

```
@ManyToMany
private Collection<Wine> wines;
```

```
@ManyToMany(mappedBy="wines")
private Collection merchants;
```

```
tx.begin();
merchant = em.merge(merchant);
em.persist(v);
merchant.wines.add(v);
tx.commit();
```

```
tx.begin();
wine = em.merge(wine);
em.persist(m);
wine.merchants.add(m);
tx.commit();
```

// the statements of one side are sufficient
// to enter the relationship into the DB

Walk through the elements of a relationship x:N

```
public class Merchant {
      @OneToMany(mappedBy="merchant")
      private Collection<Wine> wines;
      public void setWines(…) { … }
      public Collection<Wine> getWines(…) { … }
}

//  . . . transaction . . .
merchant = em.merge(merchant);
for (Wine v: merchant.getWines()) {   // relationship
      System.out.println(v.getName());
}
```

// The wines are automatically "merged" when the
// collection is walked trhough

The *merge* operation

The merge operation

As has been explained, *persist* introduces the data into the database. This statement must be executed within an transaction and the transaction must be closed between calls from two different pages.

Thus, when a *persist* must be performed for a new page, the transaction must be reopened.

This is done by the *merge* statement.

The object remains in the session. It contains the ID that refers to the record in the database where its data have been stored.

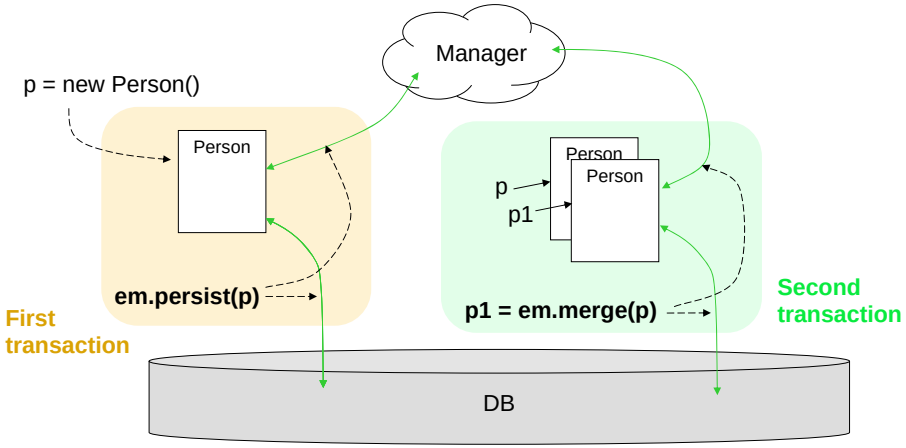
In order to resynchronize the object with the database, one executes:

```
p1 = em.merge(p)          // within a transaction
```

The new object p1 is now linked to the transaction and to the database.

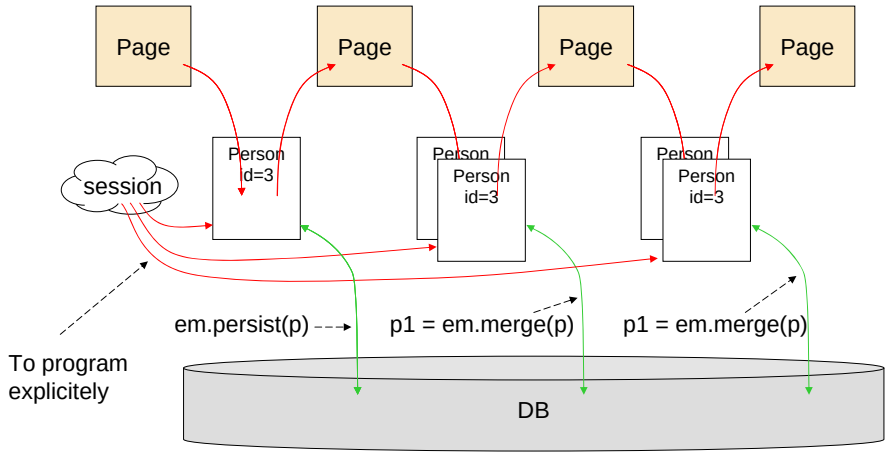
The previous object must be abandoned, and, if needed, replaced in the session by p1.

Operation *merge*



// the modifications of p1 are transmitted into the DB

Operation *merge*

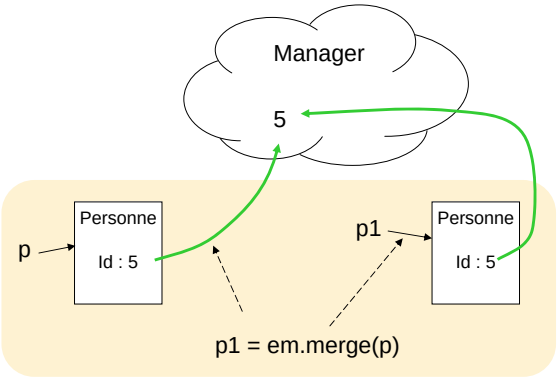


Object already bound to the transaction

If the object is bound to the transaction by a previous operation (a previous merge, a find, its existence in a relationship), the merge function returns the object already in the transaction, but it forwards the content of this object to the object already in the transaction.

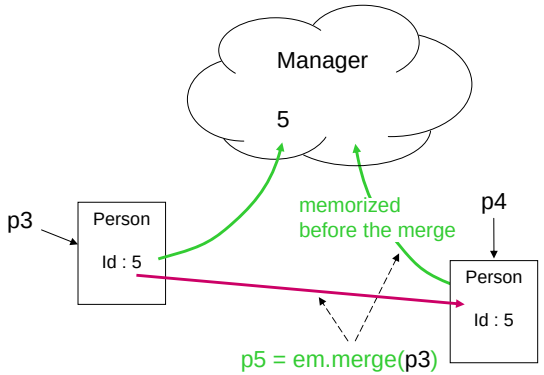
In that case, the object in the argument of the merge must also be abandonned, and the new object may have to be rebound to the session.

Functioning of the *merge*



p1 is a copy of *p*
Only the modifications on *p1* will be transmitted to the database at the closure of the transaction

merge of an object already in the transaction



If the id is already referenced, the content of the new object is copied into the referenced object and the referenced object is returned

Same thing in more details

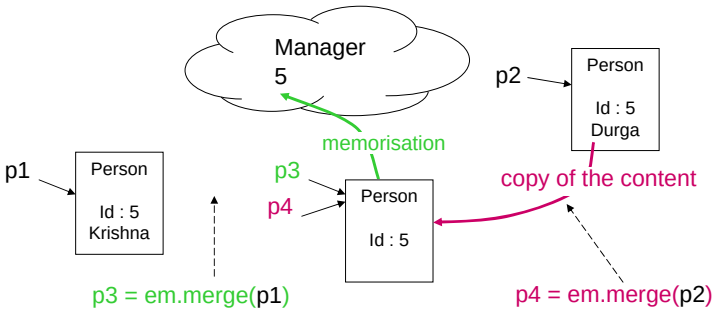
One assumes that *p1* and *p2* have been created by some means, for example by directly introducing 5 in the id.

The first merge introduces *p3* in the transaction

As the id of *p3* is now referenced, the second merge copies the content of the new object into object *p3* and then the reference of this object is returned in *p4*

At the end, *p3* and *p4* point to the same object, which is coherent

merge of an object already in the transaction



Avoiding losing the session objects

The object structure already presented copes nicely with the particularities of the *merge* statement, .

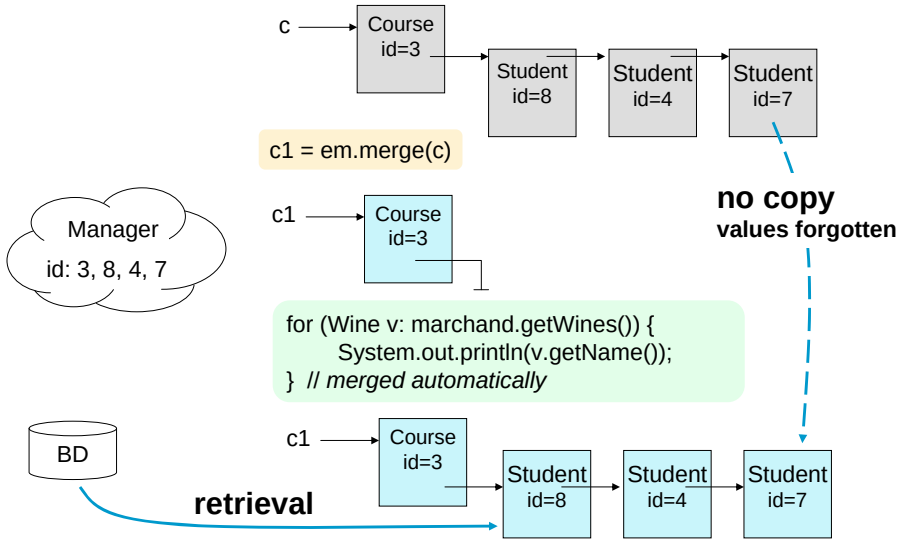
```
class Business {
        Person p;
        public void action() {
            . . .
            p = em.merge(p);     // same variable in and out
            . . .
        }
}
```

As the *business* JavaBean remains in the session, there is no need to reinsert *p* into the session.

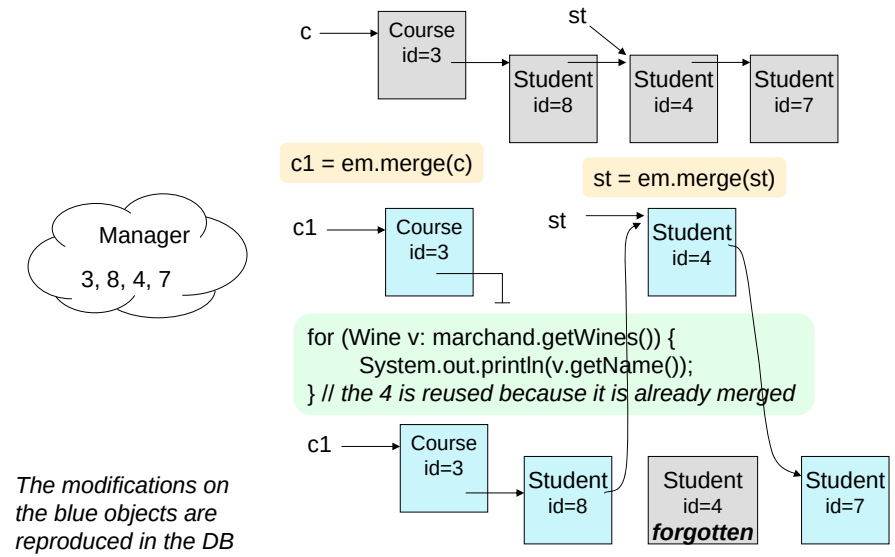
Merge for two different pages

Use of the merge command with relationships

merge of an element of the relationships



merge in the relationships



The modifications on the blue objects are reproduced in the DB

Reconnection to the database

As previously described, the JPA objects are created independently from the database and their synchronization with it is performed explicitely.

An object can thus be **free** (no ID), **attached** (an ID and within a transaction) or **detached** (keeps its ID, but is not attached to a transaction).

An object within the HTTP session can keep its ID from one page to the other. In order to reintegrate a transaction, it must be ***merged***, as explained in the previous pages.

Similarly, an object that would be retrieved by a query obtains an ID, but it is not attached to the transaction.

Attached Elements

```
em = Manager.open();
client = new Client("Hans");
// free
tx.begin();
em.persist(p);
// attached
tx.commit();
Manager.close();
// detached, its id remains initialized
client.setId(0);
// free
client = (Client) result.getSingleResult();
// detached
```


merge()

```
Client client = (Client)result.getSingleResult();
// detached, its  id is initialized
tx.begin();
client2 = em.merge(client);
// attached, the modifications will be registered at the commit
tx.commit();

// em.merge() returns a copy integrated in the transaction
// Thus take care, if client is registered in the session
// HTTP, its copy must be re-registrered in the session !
```


Update

When an ***attached*** object is updated, its modifications are followed by the manager and at the ***commit***, they are transmitted into the dadabase.

The same thing happens with the relationships.

In order to verify that the system reacts according to our plans, one can simply look at the tables with the help of the MySQL monitor

JQL

Adaptation of SQL to the JPAs

Use of JQL

```
Query result = em.createQuery("SELECT v FROM Wine v");
wine = (Wine)result.getSingleResult();          // a single line,
                                                // an exception is thrown if the command
                                                //      defines 0 or more than one entry

wines = (ArrayList<Wine>) result.getResultList(); // several lines
                                                  (0 to n)

// the find is automatically attached to the manager, but there is
// no need of a transaction, if the object is not to be modified
```


Elimination of an element

```
Query result = em.createQuery(
  "SELECT v FROM Wine v WHERE v.wineName="Bordeaux"
);
wine = (Wine)result.getSingleResult();

   . . .
em.remove(wine);


// no need of a transaction
```


JQL with relationships

Relationship 1:1

```
SELECT m FROM Merchant m
    WHERE m.wine.wineName='Epesse‘
```

(*m.wine is an attribute*)

Relationship 1:N, N:N

```
SELECT m FROM Merchant m, IN(m.hisWines) w
    WHERE w.wineName='Epesse'
```

(*m.hisWines is a collection*)

Introduce an element in a relationship

```
Query result = em.createQuery(
     "SELECT c FROM Client c
      WHERE c.name=' "+clientName+" ' ");
Client client = (Client) result.getSingleResult();

tx.begin();
client.getBasket().add(p);
tx.commit(); // or tx.rollback();
```


Lazy loading (1)

When the manager loads a relationship that only needs one attribute (1:1, 1:N), it loads the single element of the relationship directly.

When it must load a list (N:M, N:1), it loads the elements only when they are accessed. In the example below, the elements are not reachable any more, because the transaction is closed before the elements have been touched (one assumes that *client* has a 1:N relationship with *product*):

```
Client client = (Client)result.getSingleResult();
tx.begin();
client2 = em.merge(client);
tx.commit();
for (Product p: client2.listOfProducts) {
   print(p);   // are not reachable any more
}
```


Lazy loading (2)

In order to force the retrieval when the list is in the transaction, one must access the liste, for example by calling function *size()*:

```
Client client = (Client)result.getSingleResult();
tx.begin();
Client client2 = em.merge(client);
client2.size();     // just to enforce the retrieval
tx.commit();
for (Product p: client2.listeProducts) {
   print(p);        // available
}

// Note: one could also specify that the retrieval be made automatically
// (eager loading) in the parameters controlling the JPA (see the JPA doc).
```