# JEE-EJB3.0

Reda Bendraou
Reda.Bendraou@lip6.fr
*http://pagesperso-systeme.lip6.fr/Reda.Bendraou/*



# JEE

## Introduction



## Enterprise Applications: Development Challenges

Diversity and heterogeneity of the technologies used in nowadays applications

Different types of clients (Heavy Vs. Light clients)

Management of resources (e.g. databases) and services (transaction, naming… )

**Scalability**
- ✓ Load balancing
- ✓ Replication
- ✓ Clustering

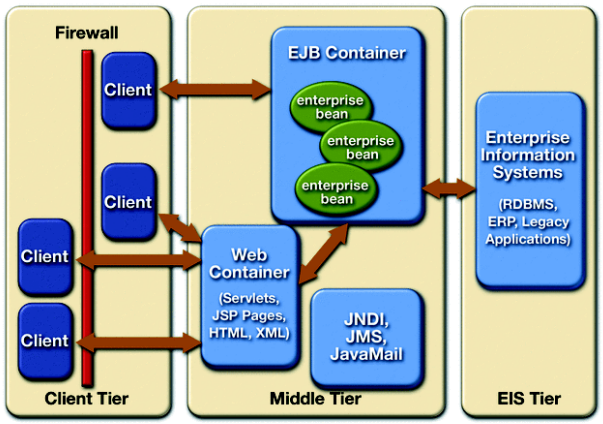
Interoperability with existing I.S. in a secure context



## Current Solutions and Initiatives

- Use of proprietary solutions
  - Combination of different technologies in a non-standard way
  - Often, solutions are not reusable. Not recommended
- Use of well-established solutions and frameworks to deal with the different challenges of EA development
  - **Appeared because of the complexity of previous EJB Versions!!!**
  - One concern, one framework
  - Often, we need to combine different frameworks
  - E.g. Spring, Struts, Hibernate, Tapestry, etc.
- The JEE Standard solution
  - A Built-in solution. All-inclusive services ☺
  - Reusability
  - JSP/Servlet + EJB

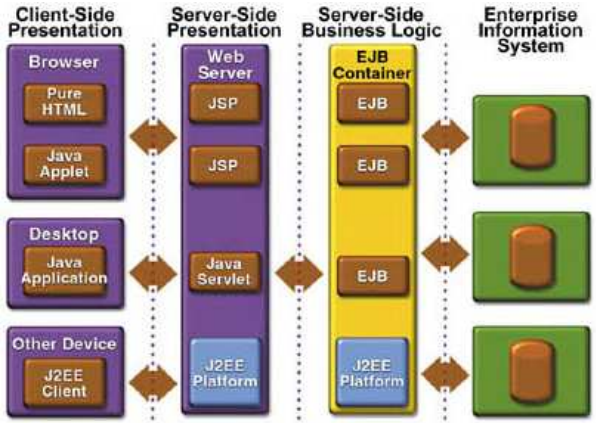
# JEE Architecture




Source : http://java.sun.com/blueprints/guidelines



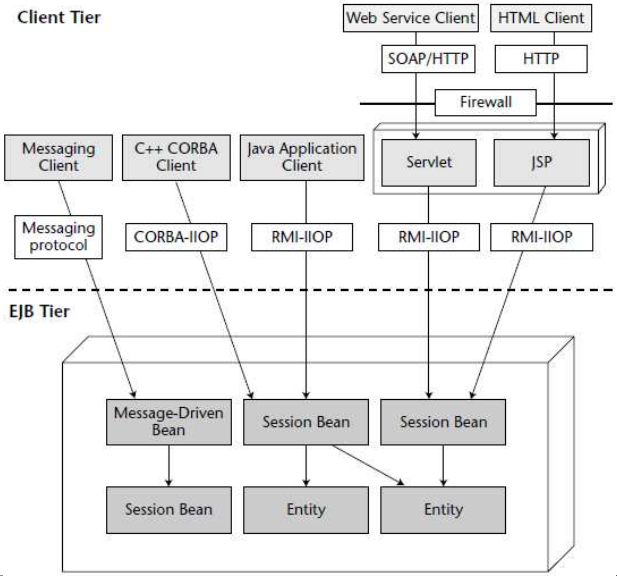
# JEE Architecture : 3 tiers




Source : http://java.sun.com/blueprints/guidelines



# EJB Sub-System: Various Clients and Beans






# JEE Architecture

The EJB architecture is THE Java standard for the design and realization of distributed enterprise applications.

- Comes in form of an API

Aspects addressed by the standard:

- Design
- Deployment
- Life cycle management of application components at runtime

« In programming with the EJB 3.0 API, the developer typically uses the **enterprise bean** class as the primary programming artifact »

# EJB: When?

If the application:

- ✓ - Has to be scalable
- ✓ - Need of a transactional context
- ✓ - Diversity of clients



# Why using Enterprise Java Beans 3?

Encapsulating business logic

- ✓ Business logic separated from control and presentation

Remote access

- ✓ Multiple apps on different servers can access EJBs

Simplicity

- ✓ Relatively easy to use compared to other remote-object systems

Broad vendor support

- ✓ JBoss, Oracle AS, WebLogic, WebSphere, Glassfish, etc.

Scalability

- ✓ Virtually all Java EE app servers support clustering, load balancing, and failover



# Why using Enterprise Java Beans 3?

EJB Servers provide a built-in solution that discharge/unburden the developer of the task of explicitly coding essential services such as:

- Network connections between the clients and the EJBs
- Naming services (JNDI)
- Transactions
- Persistence and the management of DB pool of connections
- Distribution
- Security
- Management of component's life cycle

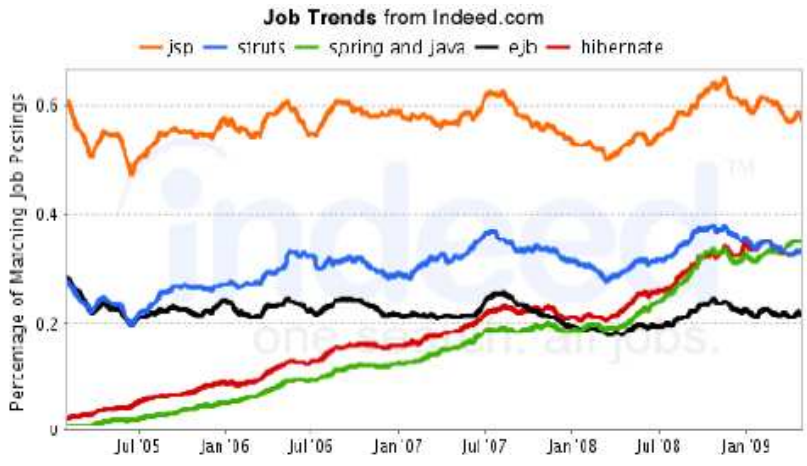
EJB Version 3 are much more readable and easier to implement than previous versions (see next slide)



# Disadvantages of EJB

- **Complexity**
  - ✓ Although EJB3 might be simpler than other remote object systems, remote-object frameworks are much more complex than local-object approaches.
  - ✓ Spring is easier and more powerful for local access
- **Requires Java EE server**
  - ✓ Can't run on Tomcat, Jetty, Resin, JRun, Resin
  - ✓ Java EE servers are usually much harder to configure, dramatically slower to start/restart during development and testing, and usually cost money
- **Requires latest Java EE version**
  - ✓ Must upgrade to latest server releases
- **Bad reputation due to earlier releases**
  - ✓ EJB2 was so complex that EJB has bad rap to this day (see next slide)

# Industry Usage of JEE Frameworks and Standards






# Why the industry was disappointed by EJBs 1.x & 2.x

EJB 1.x and 2.x

- Too complicated, heavy, constraining
- Difficulty to use some basic OO Concepts (inheritance, polymorphism, …)
- limited support of objet/relational mapping
- Lot of XML files, hard to write/maintain/understand

**From EJB 2 to EJB 3**

- Use of Java annotations and genericity in order to simplify Beans writing
- Important reduction of XML files (see next slide)



# Simplicity of EJB3 (wrt. EJB2.x)

| Application Name | Item Measured | J2EE 1.4 Platform | Java EE 5 Platform | Improvement |
|---|---|---|---|---|
| AdventureBuilder | Number of classes | 67 | 43 | 36% fewer classes |
| | Lines of code | 3,284 | 2,777 | 15% fewer lines of code |
| RosterApp | Number of classes | 17 | 7 | 59% fewer classes |
| | Lines of code | 987 | 716 | 27% fewer lines of code |
| | Number of XML files | 9 | 2 | 78% fewer XML files |
| | Lines of XML code | 792 | 26 | 97% fewer lines of XML code |



# Notion of implicit Middleware

**Principle**

- Writing your business object without worrying about distribution, transactions, persistence aspects
  - Some aspects can be customized using configuration files (jndi.properties, persistence.xml, etc.)

**Purpose**

- More readable code/components
- Your component => (almost) POJO
- Reusable solutions => Portable from one server into another

# Enterprise Java Beans

« server-side component that encapsulates the business logic of an application »

## The Triad of Beans

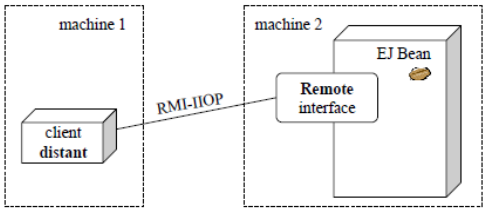
- **Session Beans** : *performs a task for a client*
- **Entity Beans** : *represents a business entity object that exists in persistent storage*
- **Message-Driven Beans :** *listening processing messages asynchronously*



# Accessing the Beans

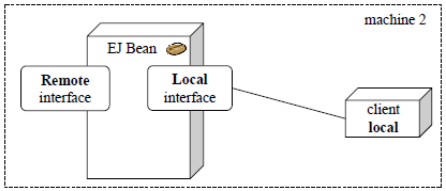
## Each EJ Bean may provide remote interfaces

- Client runs in a different JVM
- Transparent Distribution




## Each EJ Bean may provide local interfaces

- Web components or beans in the same JVM






# Local Vs. Distant Interfaces

## Beans strongly or loosely coupled

- ✓ Strong coupling : two interdependent beans => local

## Type of the client

- ✓ Clients can be :
  - Applications located in a client machine (heavy client) => distant
  - Web Components (jsp, servlet) => distant / local
  - Other beans => distant / local

## Clustering and load balancing

- ✓ Beans on the server side located in different machines => distant



# Session Beans

# Session Beans: Definitions

**Definition:** Session Beans are reusable components that contain logic for business processes

A session bean can perform:
- Banking transaction, stock trades, complex calculation, a workflow, etc.

Session Bean Types

**1. Stateless session bean**
- ✓ without a state
- ✓ Information is not persistent between two successive calls
- ✓ 2 instances of a given *bean* are equivalent

**2. Stateful session bean**
- ✓ Has a state (in memory)
- ✓ Similar to Servlet/JSP session
- ✓ The same instance along a client's session
- ✓ 1 instance per client
- ✓ Heavy!!! Be carful!



# Stateless Session Beans: Coding

1 interface (or 2 : Local + Remote) + 1 class

The Interface

• annotations @javax.ejb.Local or @javax.ejb.Remote

```
import javax.ejb.Remote;

@Remote
public interface CalculatriceItf {
      public double add(double v1,double v2);
      public double sub(double v1,double v2);
      public double mul(double v1,double v2);
      public double div(double v1,double v2);
}
```



# Stateless Session Beans: Coding

Class (that implements the interface)
- ✓ annotation @javax.ejb.Stateless or @javax.ejb.stateful

```
import javax.ejb.Stateless;

@Stateless
public class CalculatriceBean implements CalculatriceItf {
      public double add(double v1,double v2) {return v1+v2;}
      public double sub(double v1,double v2) {return v1-v2;}
      public double mul(double v1,double v2) {return v1*v2;}
      public double div(double v1,double v2) {return v1/v2;}
}
```

- ✓ possibility to name *beans* : @Stateless(mappedName="**foobar**")
- ✓ **Recommended!!** (to make your application server-independent)
- ✓ Default: the bean class name



# Remote Client's Code for Stateless Beans

Clients find the bean via JNDI
- ✓ Client Java code doesn't even know the machine on which the bean resides

Clients use the bean like a normal POJO
- ✓ But arguments and return values are sent across network
- ✓ So, custom classes should be Serializable

Core code

```
InitialContext context = new InitialContext();
InterfaceName bean =(InterfaceName)context.lookup("JNDI-Name");
```

jndi.properties
- ✓ Text file in classpath; gives remote URL and other info

# Remote Client's Code for Stateless Beans

Code of a distant client

```
public class Client {
public static void main(String args[]) throws Exception {

        javax.naming.Context ic = new javax.naming.InitialContext();
        CalculatriceItf bean = (CalculatriceItf) ic.lookup("foobar");
        double res = bean.add(3,6);
}
}
```



# Note on Name in context.lookup

**Issue**

- ✓ You pass a name to context.lookup. If you just use @Remote with no mappedName, default name is different for JBoss than for Glassfish (or other servers).
- ✓ **In JBoss, the default JNDI name would be "NumberServiceBean/remote"**
- ✓ In Glassfish, the default JNDI name would be **"coreservlets.bean.NumberService"**

Solution: use mappedName

- ✓ I use @Stateless(mappedName="foobar") instead of just @Stateless
- ✓ So, I can use the same name (foobar) regardless of which server the EJB project is deployed to



# Local Client's Code for Stateless Beans

- Could be a servlet or a JSP
- Located in the same server as the bean
- **Use of Dependency Injection mechanism**
  - ✓ Property (variable) typed by the Interface
  - ✓ annotated with **@EJB** eventually @EJB(name="foobar")

```
public class ClientServlet extends HttpServlet {
@EJB(name="foobar")
private CalculatriceItf myBean;

public void service( HttpServletRequest req, HttpServletResponse resp ) {
resp.setContentType("text/html");
PrintWriter out = resp.getWriter();
double result = myBean.add(12,4.75);
out.println("<html><body>"+result+"</body></html>");
}
}
```



# Local Client's Code for Stateless Beans

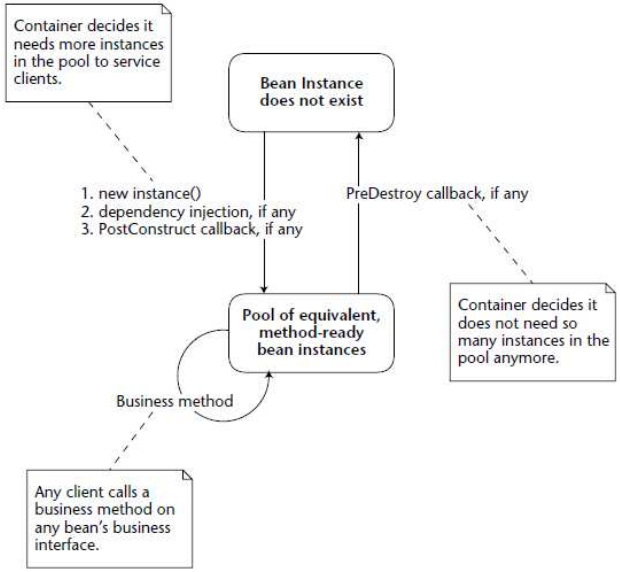
**Restrictions**

– Before instance variables, not local variables

– Both classes must be part of same EAR on same server

- ✓ In Eclipse, all classes in a single EJB project satisfy this
- ✓ If you use an EJB project (EJBs) and Dynamic Web projects (classes that use the EJBs), you must choose "Add project to an EAR" and specify the same one

# Stateless Session Beans: Lifecycle






# Stateful Session Beans: Definition

## POJOs

- ✓ Instance of the Bean relates to a specific client (in memory while he/she is connected)
- ✓ Expires in case of inactivity (similar to session in Servlet/Jsp)
- ✓ Ordinary Java classes; no special interfaces or parent classes.
- ✓ E.g. e-commerce applications with shopping cart.

## Local or remote access

- ✓ Can be accessed either on local app server or remote app server

## Annotations

**@Stateful : declaring a Stateful *bean***

**@Remove: defines the methods that ends the session**

- ✓ The Session expires when the method annotated with @Remove is executed



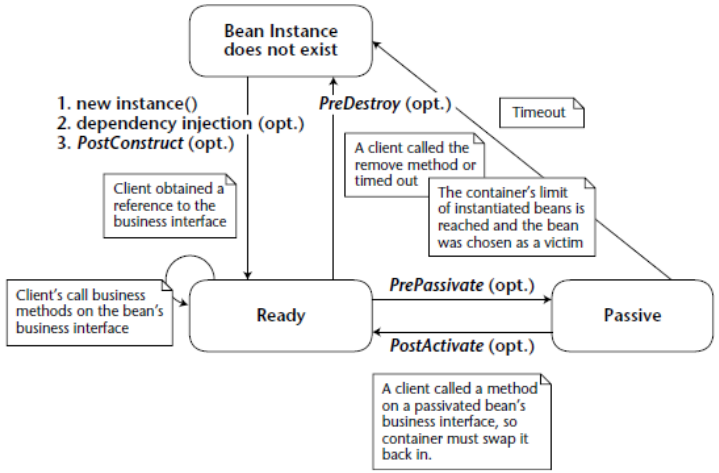
# Stateful Session Beans: Coding

For the interfaces and Client coding=>same as Stateless beans

```
@Stateful
public class CartBean implements CartItf {
      private List items = new ArrayList();
      private List quantities = new ArrayList();
      public void addItem( int ref, int qte ) { ... }
      public void removeItem( int ref ) { ... }
@Remove
public void confirmOrder() { ... }
}
```



# Stateful Session Beans: Lifecycle



# Callback Methods

Management of Bean's lifecycle

A Callback method is :

- ✓ Decorated with an annotation
- ✓ Of type `void`, and without arguments

Example :

```
@Stateful public class ShoppingCartBean implements ShoppingCart
{
       private float total;
       private Vector productCodes;
       public int someShoppingMethod(){...};
       ...
       @PreDestroy void endShoppingCart() {...};
}
```



# Callback Annotations

```
@PostConstruct
public void initialise() { … at Bean's initialization … }

@PreDestroy
public void detruit() { … destruction of Bean … }

@PrePassivate //only for Stateful beans
public void avantSwap() { … to do before Bean is swapped … }

@PostActivate //only for Stateful beans
public void apresSwap() { … to do after Bean is activated … }
```



# Session Beans…

Compilation : javac

Add javaee.jar in the classpath

Packaging : in a .jar file

/test/HelloWorld.class
/test/HelloWorldBean.class
/test/Test.class
/test/TestBean.class

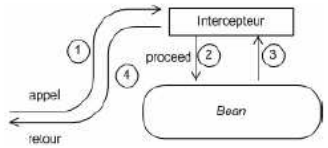
Deployment : in the server's deploy folder



# Advanced Concepts: Interceptors

Executing a behavior before/after bean's methods

- ✓ AOP inspiration (cf. AspectJ, Spring AOP, …)
- ✓ **@Interceptors** : methods to be intercepted
- ✓ **@AroundInvoke** : interception methods




**Syntax**

**Object <methodname>( InvocationContext ctx ) throws Exception javax.interceptor.InvocationContext**

- ✓ Allow obtaining information over the intercepted methods
- ✓ Provide a ***proceed()*** method to pursue the execution of the intercepted method

# Advanced Concepts: Interceptors

```
public class EnchereBean {
    @Interceptors(MyInterceptor.class)
    public void ajouterEnchere( Bid bid ) { ... } }


public class MyInterceptor {
@AroundInvoke
public Object trace( InvocationContext ic ) throws Exception {
// ... code before...

java.lang.reflect.Method m = ic.getMethod();
Object bean = ic.getTarget();
Object[] params = ic.getParameters();
// eventually modification of the parameters with ic.setParameters(...)
Object ret = ic.proceed(); // calling the bean (optional)

// ... code after ...
return ret; } }
```



# Session Beans: Summary

**Stateless session beans**

- ✓ Interface: mark with @Remote
- ✓ Class: mark with @Stateless(mappedName="blah")

**Stateful session beans**

- ✓ Mark class with @Stateful instead of @Stateless
- ✓ Mark a method with @Remove

**Session bean clients**

```
InitialContext context = new InitialContext();
InterfaceType var = (InterfaceType)context.lookup("blah");
var.someMethod(args);
```

- ✓ For stateful beans, call specially marked method when done
- ✓ Need jndi.properties specific to server type

**Local access to beans**

- ✓ @EJB private InterfaceType var;



# Entity Beans



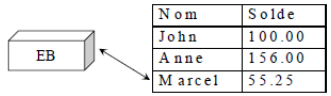
# Entity Beans

Entity Bean = A tuple in a database (RDB)

Entity Bean : POJO

(Plain Old Java Object)

- ✓ POJO property = a column in a table




Entity Bean API : JPA (Java Persistence API)

Inspired from Hibernate, TopLink…

Managing persistency in a transparent way

Advantage: using objects instead of SQL requests

# Entity Beans

annotation **@Entity** => a class corresponding to the *entity bean* (EB)

each EB class => table

- Default: the name of the class=name of the table
- Except if annotation **@Table(name="...")**



# Entity Beans

2 (exclusive) modes for the definition of table's columns

- ✓ *property-based access* : annotate *getter methods*
- ✓ *field-based access* : annotate attributes

Default: column name= *field/property* class

Except if annotation **@Column (name="...")**

annotation **@Id** : defines a primary key



# Entity Beans: Example

```
@Entity
@Table(name = "FILMS")
public class Film implements java.io.Serializable {
  private int id;
  private String name;

  @Id @GeneratedValue(strategy = GenerationType.AUTO)
  public int getId() { return id; }
  public void setId(int id) { this.id = id; }

  public String getName() { return name; }
  public void setName(final String name) { this.name = name }
}
```



# Entity Beans

**@Basic** or nothing : indicates that a field is persistent

All fields are persistent

Except if annotated **@Transient**

Primary Key is mandatory : primitive type or composed

**@GeneratedValue(strategy=?)** : indicates how Ids are generated

*Auto, Identity, Sequence, etc.*

# Entity Beans: Composed Primary Key

```
public class ClefEtudiant implements
java.io.Serializable{
        private String nomId;
        private String prenomId;

        public String getNomId(){
                return nomId;
        }
        public void setNomId( String nomId ){
                this.nomId = nomId;
        }
        public String getPrenomId(){
                return prenomId;
        }
        public void setPrenomId( String prenomId ){
                this.prenomId = prenomId;
        }
        public int hashCode(){
                return ...
        }
        public boolean equals(Object otherOb) {
                ...
        }}
```

```
@IdClass(ClefEtudiant.class)
@Entity
public class Etudiant{

        private String nomId;
        @Id
        public String getNomId(){
                return nomId;
        }
        public void setNomId( String nomId ){
                this.nomId = nomId;
        }

        private String prenomId;
        @Id
        public String getPrenomId(){
                return prenomId;
        }
        public void setPrenom( String prenomId ){
                this.prenomId = prenomId;
        }
}
```



# Entity Beans: Two classes in one table

**@Embeddable** & **@Embedded** : fields of two classes into one table

```
@Embeddable
public class Address implements Serializable {
private String rue; private int codePostal;
}
```

```
@Entity
public class User {
private String nom;
@Embedded
private Address adresse;
}
```



# Entity Beans: Associations

**Entity Beans are linked with each other through Associations**

**Association Multiplicities**

- ✓ 1 - 1 (**one-to-one**)
- ✓ 1 - n (**one-to-many**)
- ✓ n - 1 (**many-to-one**)
- ✓ n - n (**many-to-many**)

**Navigability of Associations (Direction)**

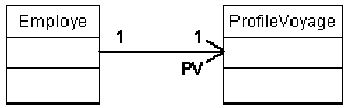
- ✓ **Bi-directional** : **two sides** : an **owner** side and an **inverse** side
- ✓ **Unidirectional** : **one side** : the **owner**



# Unidirectional : OneToOne

```
@Entity
public class Employe {
private ProfilVoyage pv;
@OneToOne
public ProfilVoyage getPv() {
return pv; }
public void setPv(ProfilVoyage profil) {
this.pv = profil; }
...
}
```

```
@Entity
public class ProfilVoyage
{
...
}
```




***Employe*** entity ➔ ***Employe*** table

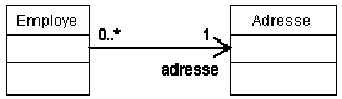
***ProfilVoyage*** entity ➔ ***ProfilVoyage*** table with ***pf_Id*** as **PK**

***Employe*** table owns a foreign key to ***ProfilVoyage***, ***PV***

# Unidirectional : ManyToOne

```
@Entity
public class Employe {
private Adresse ad;
@ManyToOne
public Adresse getAd() {
return ad; }
public void setAd(Adresse a) {this.ad = a ; }
...
}
```

```
@Entity
public class Adresse
{
...
}
```




***Employe*** entity ➔ ***Employe*** table

***Adresse*** entity ➔ ***Adresse*** table with ***Id_ad*** as **PK**

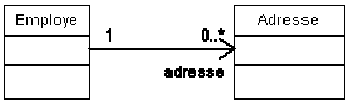
***Employe*** table owns a foreign key to ***Adresse*** , ***ad***



# Unidirectional : OneToMany

```
@Entity
public class Employe {
private Collection<Adresse> adresses;
@OneToMany
public Collection<Adresse> getAdresses() {return adresses; }
public void setAdresses
        (Collection<Adresse> adresses) {this.adresses =
adresses;}
...
}
```

```
@Entity
public class Adresse
{
...
}
```




***Employe*** entity ➔ ***Employe*** table

***Adresse*** entity ➔ ***Adresse*** table with ***Id_ad*** as **PK**

Creation of a join table Employe_Adresse with two columns (i.e. Employe_Pkemploye & Adresse_PKAdresse, each column represents a PK to each table



# Unidirectional : ManyToMany

```
@Entity
public class Employe {
private Collection<Adresse> adresses;
@ManyToMany
public Collection<Adresse> getAdresses() {return adresses; }
public void setAdresses
        (Collection<Adresse> adresses) {
this.adresses = adresses; }
...
}
```

```
@Entity
public class Adresse
{
...
}
```

***Employe*** entity ➔ ***Employe*** table

***Adresse*** entity ➔ ***Adresse*** table with ***Id_ad*** as **PK**

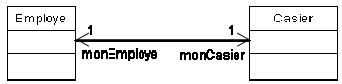
Creation of a join table Employe_Adresse with two columns (i.e. Employe_Pkemploye & Adresse_PKAdresse, each column represents a PK to each table



# Bidirectional : OneToOne/OneToOne

```
@Entity
public class Employe {
private Casier monCasier;
@OneToOne
public Casier getMonCasier()
{ return monCasier; }
public void setMoncasier(Casier c)
{ this.monCasier = c; }
...
}
```

```
@Entity
public class Casier {
private Employe monEmploye;
@OneToOne(mappedBy="monCasier")
public Employe getMonEmploye()
{ return monEmploye; }
public void setMonEmploye(Employe e)
{ this.monEmploye = e; }
...
}
```




***Employe*** entity ➔ ***Employe*** table

***Casier*** entity ➔ ***Casier*** table with ***Id_ad*** as **PK**

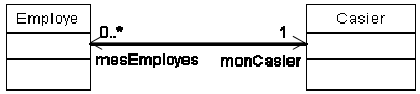
***Employe*** table owns a foreign key to ***Casier*** , ***monCasier***

# Bidirectional : ManyToOne/OneToMany

```
@Entity
public class Employe {
private Casier monCasier;
@ManyToOne
public Casier getMonCasier()
{ return monCasier; }
public void setMoncasier(Casier c)
{ this.monCasier = c; }
...
}
```

```
@Entity
public class Casier {
private Collection<Employe> mesEmployes;
@OneToMany(mappedBy="monCasier")
public Collection<Employe> getMesEmployes()
{ return mesEmployes; }
public void setMesEmployes
        (Collection<Employe> e)
{ this.mesEmployes = e; }
...
}
```

***Employe*** entity ➔ ***Employe*** table




***Casier*** entity ➔ ***Casier*** table with ***Id_ad*** as **PK**

***Employe*** table owns a foreign key to ***Casier*** , ***monCasier***



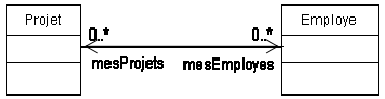
# Bidirectional : ManyToMany/ManyToMany

```
@Entity
public class Projet {
Collection<Employe> mesEmployes;
@ManyToMany
public Collection<Employe> getMesEmployes()
{ return mesEmployes; }
public void setMesEmployes
        (Collection<Employe> e)
{ this.mesEmployes = e; }
...
}
```

```
@Entity
public class Employe {
private Collection<Projet> mesProjets;
@ManyToMany(mappedBy= »mesEmployes")
public Collection<Projet> getMesProjets()
{ return mesProjets; }
public void setMesProjets
        (Collection< Projet > p)
{ this.mesProjets = p; }
...
}
```

***Projet*** entity ➔ ***Projet*** table

***Employe*** entity ➔ ***Employe*** table




Creation of a join table **Projet_Employe** with two columns (i.e. mesProjets_PKProjet & mesEmployes_Pkemploye, each column represents a PK to each table



# Entity Beans: Inheritance

Entities support inheritance and polymorphism

Entities may be concrete or abstract

An Entity can inherit a non-entity class

A non-entity class can inherit an entity class



# Inheriting abstract class

```
@Entity
public abstract class Personne{

    @Id
    protected String numSecuSociale;

}
```

```
@Entity
public class Employe extends Personne{

    protected float salaire;

}
```

# Entity Beans: Inheritance Strategies

:

## One Table by classes hierarchy (Default)

```
@Inheritance(strategy=SINGLE_TABLE)
```

## One Table by concrete class

```
@Inheritance(strategy=TABLE_PER_CLASS)
```

## Join Strategy: a join between the concrete class and the super class tables

- No duplication of the fields, a Join operation to get the info

```
@Inheritance(strategy=JOINED)
```



# Entity Beans: Inheritance Strategies

One Table by classes hierarchy (Default)

- ✓ Implemented in most tooling solutions
- ✓ Good support of polymorphism
- ✓ Columns proper to sub-classes set at null

One Table by concrete class

- ✓ Some issues remain regarding polymorphism

Join Strategy

- ✓ Good support of polymorphism
- ✓ Not always implemented
- ✓ Join operation can be costly



# Inheritance Strategies : One table

A discriminator column is used

Possible Types

- ✓ DiscriminatorType.STRING (Default)
- ✓ DiscriminatorType.CHAR
- ✓ DiscriminatorType.INTEGER.

Example

```
@Entity
@DiscriminatorColumn(name="DISCRIMINATEUR_PERSONNE"
discriminatorType=DiscriminatorType.INTEGER)
public class Personne{
        ...
}
```



# Inheritance : *MappedSuperClass*

Entities can inherite non-entity beans

**MappedSuperClasses** are not accessible to the Entity Manager

Not considered as an Entity (no table in the DB)

```
@MappedSuperclass
public class BaseEntity {
        public String baseattribute1;
        public String baseattribute2;.
}
```

```
@Entity
public class Entity extends BaseEntity{
        @Id
        protected int id;
        protected float attribute;
}
```

# Entity Bean : Complements

**Fetch** : option for loading the graph of objects

- ✓ FetchType.**EAGER** : loads all the tree (required if Serializable)
- ✓ FetchType.**LAZY** : only on demand (unusable with Serializable)

Cascade : transitivity of operations over the beans

- ✓ CascadeType.ALL : every operation is propagated
- ✓ CascadeType.MERGE : in case of a merge
- ✓ CascadeType.PERSIST : Film becomes persistent ⇒ List<SalleProg> too
- ✓ CascadeType.REFRESH : loading from the DB
- ✓ CascadeType.REMOVE : delete in cascade



# Entity Beans: Entity Manager

Managing Entities: *Entity Manager*

Ensure synchronization between Java objects and DB tables

In charge of adding/updating/deleting records

Executing requests

**Accessible through dependency injection**

- ✓ type of the attribute javax.persistence.EntityManager
- ✓ annotated by **@PersistenceContext**



# Entity Beans: Entity Manager

Utilization of Entity Beans inside Session Beans

```
@Stateless
public class MyBean implements MyBeanItf {
@PersistenceContext
private EntityManager em;
public void init() {
      Book b1 = new Book("Honore de Balzac","Le Pere Goriot");
      Book b2 = new Book("Honore de Balzac","Les Chouans");
      Book b3 = new Book("Victor Hugo","Les Miserables");
      em.persist(b1);
      em.persist(b2);
      em.persist(b3);
} }
```



# Entity Beans: Entity Manager

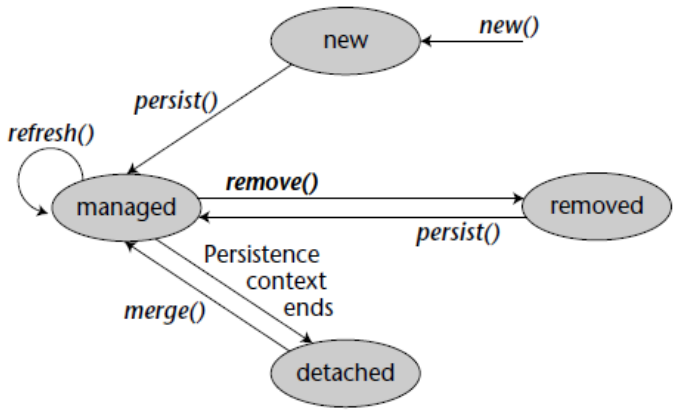
Research by id

Book myBook = em.**find**(Book.class,12);

- returns null if the key does not exist in the table

- IllegalArgumentException
  - ✓ If first parameter is not a EB class
  - ✓ If second parameter is not of the same type as the Id's type

# Entity Beans: Lifecycle






# Entity Beans: Lifecycle

- **new.** The entity instance was created in memory, but is not yet associated with either a persistent identity in the database or a persistence context. This is the state that our Account entity was in right after creation. Changes in the entity state are not synchronized with the database at this stage.
- **managed.** The entity has a persistent identity in the database and is currently associated with a persistence context. Our Account entity was in the managed state after the `persist()` method was called. Changes to the entity will be synchronized with the database when transactions are committed or when synchronization is explicitly triggered using the `flush()` operation.
- **detached.** The entity does have a persistent identity but is not or is no longer associated with the persistence context.
- **removed.** The entity is currently associated with a persistence context but has been scheduled for removal from the database.



# Entity Beans: Lifecycle

Attaching & detaching beans

- ✓ `void` **`merge`**`(Object entity)`

Merge the state of the given entity into the current persistent context

Used with a detached bean

E.g. the Bean is modified at the client and sent back to the server

- ✓ `void` **`persist`**`(Object entity)`: persist and attach the bean

Usable over a new Bean (after a new)

```
Film createFilm(String name) {
  Film film= new Film();
  res.setName(name);
  em.persist(film); //attach the bean + makes it persistent
  return film; // the copy is detached
}
```



# Entity Beans…

Principal methods of Entity Manager

- ✓ `Object` **`find`**`(Class cl, Object key) :`

  Find an EntityBean by its id
- ✓ `boolean` **`contains`**`(Object entity) :`

  True if the entity is attached to the EntityManager
- ✓ `Query` **`createQuery`**`(String qlString) :`

  Creating a query in EJB-QL
- ✓ `Query` **`createNamedQuery`**`(String name) :`

  Creating a named query
- ✓ `Query` **`createNativeQuery`**`(String sqlQuery) :`

  Creating an SQL query
- ✓ `void` **`remove`**`(Object entity) :`

  Remove the entity from the base
- ✓ `void` **`refresh`**`(Object entity) :`

  Recharging the bean from the DB

# Entity Beans…

Examples

Finding a film by its id

```
public void findFilm(int id) {
  return em.find(Film.class, Integer.valueOf(id));
}
```

Removing a film :

```
public void removeFilm(int id) {
  em.remove(findFilm(id));
}
```



# Entity Beans : the Query Language

« EJB-Query Language » : Close to SQL

- ✓ Selection from the Bean's name
- ✓ Parameters indicated by : pram-name (fname in the example)
- ✓ Request in the `Query` object
- ✓ Result from `Query`

Example :

```
public Film findFilmByName(String name) {
  Query q = em.createQuery(
     "select f from Film where f.name = :fname");
  q.setParameter("fname", name);
  List<Film> res = q.getResultList();
  return res.size() == 0 ? null : res.get(0);
}
```

getSingleResult() in case of a unique result

• NonUniqueResultException in case of no unique result



# Entity Beans : Named Queries

A named query attached to the EB

```
@Entity
@NamedQuery(name="allbooks",query="select OBJECT(b) from Book b")
public class Book { ... }
Query q = em.createNamedQuery("allbooks");
List<Book> list = (List<Book>) q.getResultList();
```



# Entity Beans : Named Queries

Multiple Queries

```
@Entity
@Table(name = "FILMS")
@NamedQueries({
   @NamedQuery(name = "findAllFilms",
          query = "select f from Film f"),
   @NamedQuery(name = "findFilmByName",
      query = "select f from Film f WHERE f.name = :fname")
    })
public class Film implements java.io.Serializable { …

Used in a SessionBean :
void findFilmByName() {
  Query q = em.createNamedQuery("findFilmByName"); …
```

## Entity Beans : Native Queries

Query createNativeQuery(String **sqlString**)

- ✓ Create an instance of Query for executing a native SQL statement, e.g., for update or delete.

Parameters:

sqlString - a native SQL query string

Returns: the new query instance

Throws:

IllegalStateException - if this EntityManager has been closed.



## Entity Beans : Lifecycle- Interceptors

Interception of state changes

Around the creation (em.persist) :

- ✓ @PrePersist
- ✓ @PostPersist

At loading time from DB (em.find, Query.getResultList)

- ✓ @PostLoad

Around updates (modification of a filed, em.merge)

- ✓ @PreUpdate
- ✓ @PostUpdate

Around a remove action (em.remove)

- ✓ @PreRemove
- ✓ @PostRemove



## Deployment Descriptor: Persistence.xml

Specifies advanced mapping concepts, the data source (here the JBoss default DB)

In the **META-INF** folder in the EJB Project

```
<persistence>
    <persistence-unit name="IntroEJB3">
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
        <property name="hibernate.hbm2ddl.auto" value="update"/>
      </properties>
    </persistence-unit>
  </persistence>
```



## Message-Driven Beans

# Message-Driven Beans

Message-Driven Bean : interaction by messages

MOM : Message Oriented Middleware

Two modes of communication

- ✓ N vers 1 : Queue
- ✓ N vers M : Topic

Message Driven Bean : JMS-based specification

JMS : Java Message Service



# Message-Driven Beans

Principles of Message Driven Beans

- ✓ Consume asynchronous messages
- ✓ Stateless (all instances of the same MDB are equivalent)
- ✓ Handle client messages
- ✓ 1 business method (**onMessage**)
  - Fixed Parameters
  - No return value
  - No exception

When to use a MDB

- ✓ Avoid Blocking calls
- ✓ When you have clients (producers) et servers (consumers)



# Message-Driven Beans

JMS (java.sun.com/jms)

Queue : Thread of discussion (one consumer)

Topic : Topic of discussion (diffusion)

ConnectionFactory : Factory of connections towards queue/topic

Connection : connection towards queue/topic

Session :

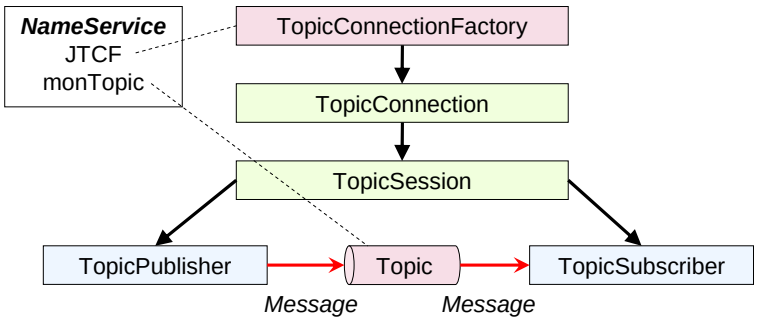
- ✓ Creation of an sender and of a receiver
- ✓ Can be transactional



# Message-Driven Beans : Architecture

Architecture of Java Message Service

(for queue, remplace Topic by Queue, Publisher by Sender, Subscriber by Receiver)



# Message-Driven Beans : Coding

Example of  Message-Driven Bean

```
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(
      propertyName = "destination",
      propertyValue = "topic_rigolo"),
   @ActivationConfigProperty(
      propertyName = "destinationType",
      propertyValue = "javax.jms.Topic")})

public class Mdb implements MessageListener {
  public void onMessage(Message inMessage) {
    System.out.println(((TextMessage)msg).getText());
  }
}
```



# Message-Driven Beans : Coding

Example of a sander

```
@Resource(name="rigolo", mappedName="topic_rigolo")
Topic topic;
@Resource(name="factory", mappedName="JTCF")
TopicConnectionFactory factory;
TopicSession   session;
TopicPublisher sender;

public void publish(String value) {
  TopicConnection tc = factory.createTopicConnection();
  session = tc.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
  sender = session.createPublisher(topic);
  TextMessage msg = session.createTextMessage();
  msg.setText("MDB: " + value);
  sender.publish(msg);    }
```