Composants répartis

Chap #6

Des objets aux objets répartis

- Nous avons abordé dans les cours précédents la notion d'objet réparti. Par rapport aux objets “classiques”, les objets réparti :
  - Collaborent pour le bon fonctionnement d'une l'application bien qu'étants situés sur différentes plate-formes, dans des espaces d'adressage différents.
  - Sont accessible à distance au travers du réseau.
  - Sont accessibles de manière transparente (transparence de localisation): ils disposent d'une souche cliente (stub) et d'une souche serveur (skeleton) invisibles aux yeux de l'utilisateur de l'objet réparti.
- Il existe différents modèles de programmation associés, nous avons étudié :
  - Le modèle RMI en Java.
  - La spécification CORBA.

Des objets aux composants

- Au sens le plus strict du terme, un composant correspond à l'abstraction d'**une brique logicielle réutilisable.**
- Plus précisément, il s'agit d'un module logiciel :
  - **autonome**
  - réutilisable
  - **composable**
  - accessible au travers de ses interfaces (notion d'encapsulation)
- Ces propriétés sont assurés par **le respect d'une spécification** de composant. Quelque-soit cette spécification (JavaBeans, COM, ...), c'est son respect qui confère au module son statut de composant.
- Il s'agit d'un **niveau d'abstraction et d'une granularité plus élevée** que celle des objets, mais les deux notions se complémentent : les composants sont majoritairement implémentés à base d'objets.

Des objets aux composants

- **L'objectif est de permettre la construction de nouvelles applications à partir de plusieurs composants** :
  - Par association et connexion de composants déjà disponibles (notion de COTS).
  - En proposant des outils d'aide à la composition de composants.
- En java, la notion de composant existe sous la forme des **JavaBeans** qui y ajoute les notions suivantes :
  - Composant comme module logiciel autonome : fichier .jar
  - Mécanisme d'introspection du composant : reflection Java
  - Conservation de l'état du composant : sérialisation
  - Composant paramétrable : sérialisation + reflection + événements
- La programmation par composant demande souvent le respect de différentes convention de nommage (des méthodes, des classes, ...)

Des objets répartis aux composants répartis

- On veut disposer des mêmes facilités de répartition en programmation par composants qu'en programmation orientée objets.
  → notion de composants répartis.

- Un aspect central à la notion de composant réparti concerne la disponibilité d'infrastructures d'accueil de ces composants : les conteneurs de composants disponibles au sein de serveurs d'applications.

- Ce sont les serveurs d'applications et leurs conteneurs qui assurent l'infrastructure technique nécessaire à l'exécution des composants et au support de la répartition via un réseau.

- Pourquoi les objets répartis ne suffisent-ils pas ?

Des objets répartis aux composants répartis

- **Raison principale** : des objets répartis trop complexes où les aspects techniques sont à la charge du programmeur.
  Ex. avec RMI :

```
public class ObjetServeur extends UnicastRemoteObjet {
private int x ;

public void setX(int value) throws RemoteException {
   Tx.beginTransaction();
   if (User.getAuthentification()...
   this.x=value ;
   // code JDBC : stockage de X
   ...
   Tx.commitTransaction();
}

public int getX() throws RemoteException {
   ...
}

public ObjetServeur() throws RemoteException {
   ...
}
```

Des objets répartis aux composants répartis

- **Raison principale** : des objets répartis trop complexes où les aspects techniques sont à la charge du programmeur.
  Ex. avec RMI :

```
public class ObjetServeur extends UnicastRemoteObjet {
private int x ;

public void setX(int value) throws RemoteException {
   Tx.beginTransaction();
   if (User.getAuthentification()...
   this.x=value ;
   // code JDBC : stockage de X
   ...
   Tx.commitTransaction();
}

public int getX() throws RemoteException {
   ...
}

public ObjetServeur() throws RemoteException {
   ...
}
```

Gestion des transactions

Gestion de la sécurité

Gestion de la persistance

Gestion du réseau

Des objets répartis aux composants répartis

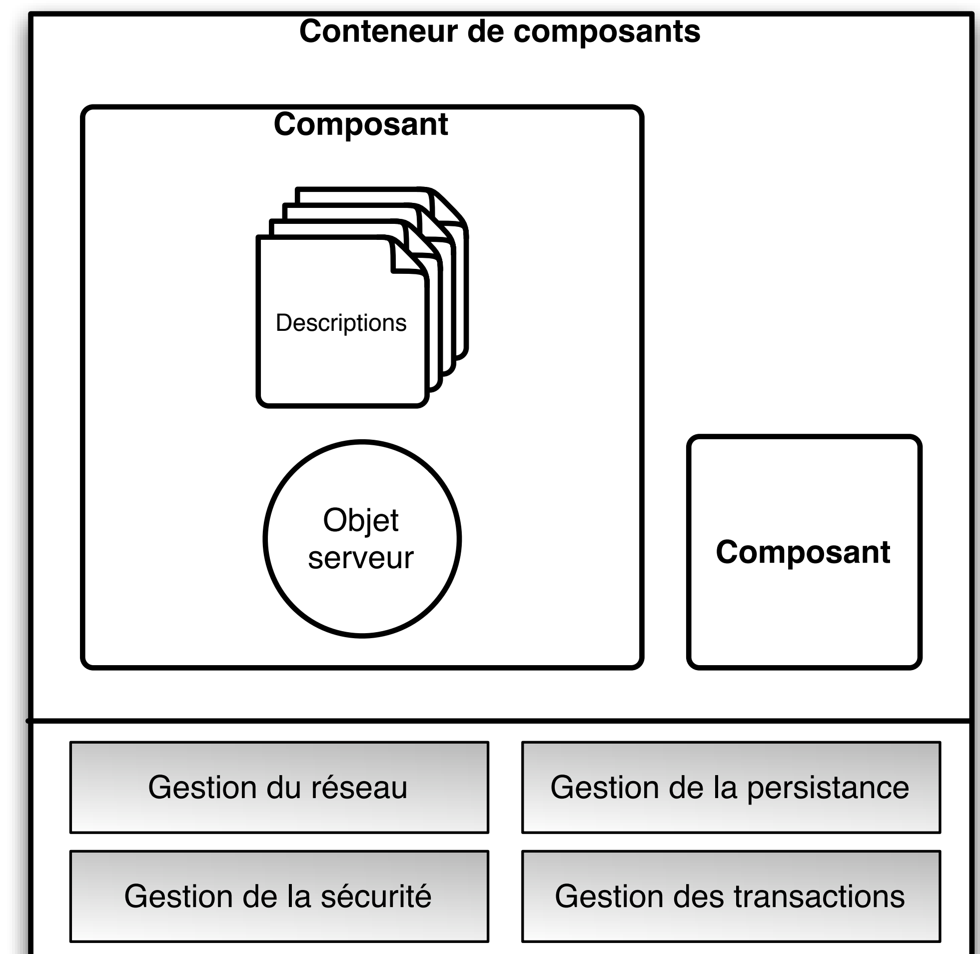
- **Problématique** : la couche fonctionnelle de l'objet réparti est masquée sous un ensemble de services techniques programmées spécifiquement par le développeur de l'objet et donc non-réutilisables.

- **Solution** :

  1. Externalisation des services techniques dans le **conteneur de composants** pour se recentrer sur les aspects fonctionnels (le code métier).

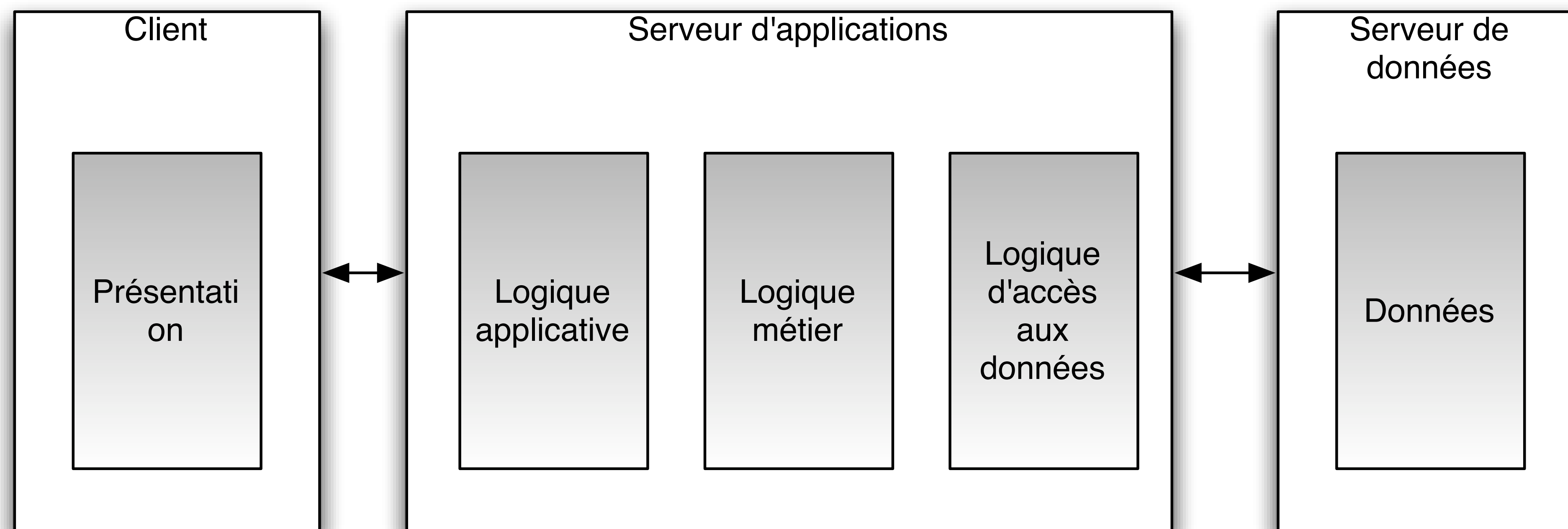
  2. Configuration des services techniques dans des fichiers de description liés à chaque composant.

  3. **Le conteneur gére le cycle de vie des composants** (instanciation, destruction) et leur contexte d'exécution.



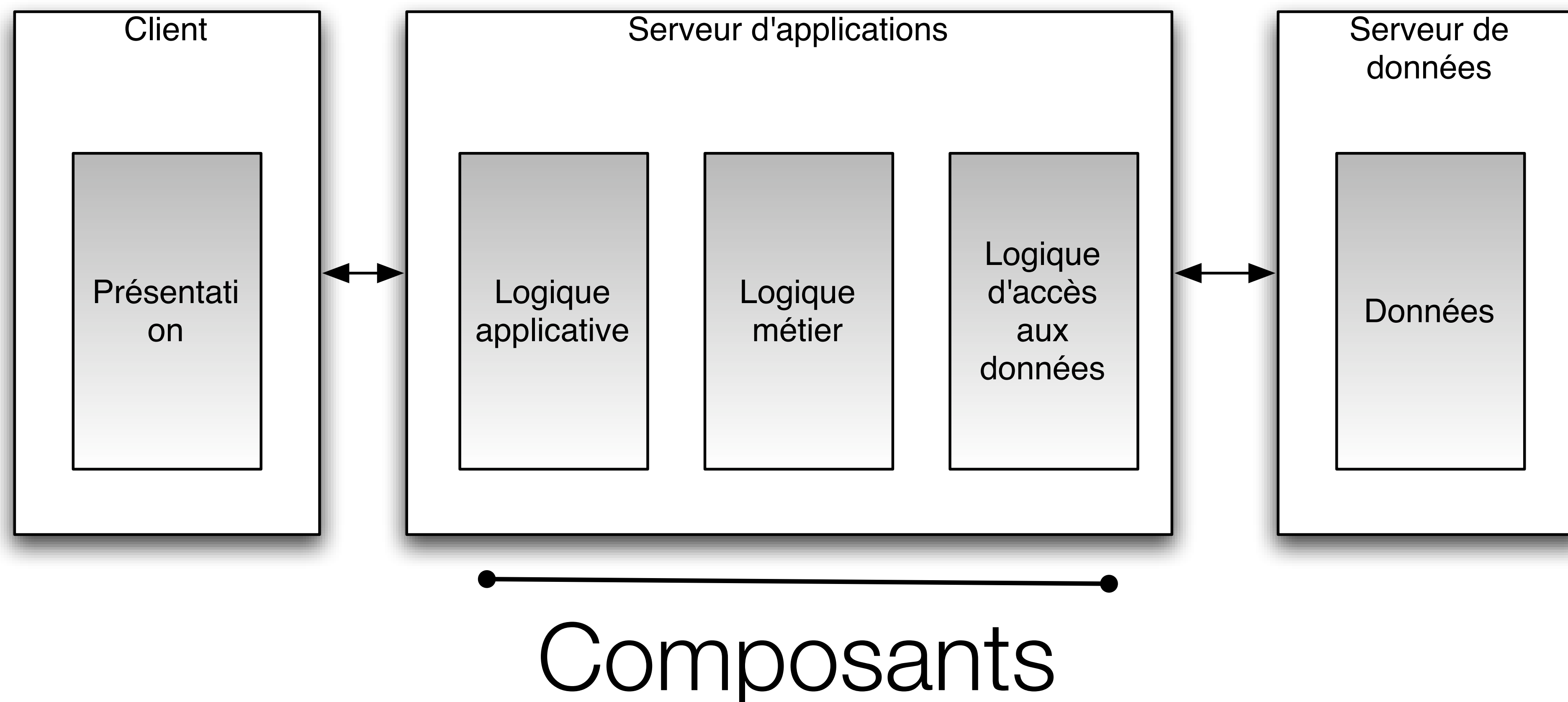
Composants répartis et architecture n-tiers

- Les composants répartis peuvent constituer une solution viable pour le développement de la logique métier dans les architectures 3 et n-tiers.
  - Le programmeur gère le code métier (les composants)
  - Le conteneur de composants gère le reste (persistance, transaction, sécurité, ...) et assure la liaison avec la couche de données (SGBD, ...).



Composants répartis et architecture n-tiers

- Les composants répartis peuvent constituer une solution viable pour le développement de la logique métier dans les architectures 3 et n-tiers.
  - Le programmeur gère le code métier (les composants)
  - Le conteneur de composants gère le reste (persistance, transaction, sécurité, ...) et assure la liaison avec la couche de données (SGBD, ...).



Composants répartis

Java EE-EJB

Chap #6

Introduction

- Java EE (“Enterprise Edition”) correspond à la spécification par Sun d’une architecture logicielle **à base de composants** autour du langage Java pour la construction d’**applications d’entreprises coté serveur**.
- Java EE définie 2 grands types de composants :
  - **Les composants web** : ils gèrent la présentation des services des applications via le web.
  - **Les composants métier EJB** (“Enterprise [Java] Beans”) : il gèrent les traitements métiers et la modélisation objet des données.
- Dans ce cours, nous nous intéressons à la notion de composant réparti (vue comme une évolution de le notion d’objet réparti) :
  - Les composants web ne sont pas des composants répartis au sens propre du terme car ils ne sont accessibles qu’au travers d’un navigateur Web **sans réelle notion de transparence de localisation**.
  - Par le suite, nous détaillerons donc les composants définis par la spécification EJB (pierre angulaire de la spécification Java EE).

Objectifs

- Les objectifs initiaux de la spécification Java EE lorsqu'elle à été mise au point par Sun en 1998 étaient les suivants (elle ne portait pas alors le même nom):
  - Fournir aux applications d'entreprise une architecture normalisée de composants logiciels répartis.
  - Etre multi-plate-forme (pas de recompilation Java sur chaque plate-forme cible).
  - Prendre en compte 3 aspects techniques majeurs dans le conteneur : persistance (couplage avec SGBD), sécurité et transactions.
  - Support de la répartition réseau grâce au protocole RMI-IIOP (cf. cours #4.1)
  - Etre adapté aux architectures 3-tiers.

Avantages, Inconvénients

- Les avantages de l'utilisation d'une architecture de type Java EE pour le développement de grandes applications répartis sont multiples :
  - Simplification du développement car les services techniques sont gérés par l'infrastructure du serveur d'application.
  - Séparation claire entre le niveau Présentation (sur le client) et le niveau Traitement (dans les composants, sur le serveur)
    → **Permet de développer des clients plus légers, caractéristique très importante dans les systèmes et réseaux mobiles.**
  - Réutilisabilité des composants : il est possible de composer une toute nouvelle application en réutilisant des briques fonctionnelles de base (composants) déjà implémentés.
- Mais il y a aussi des inconvénients, surtout lors de développements plus restreints :
  - **La lourdeur de la mise en place d'une architecture Java EE.**
  - La toute relative vitesse d'exécution de l'application ainsi réalisée.

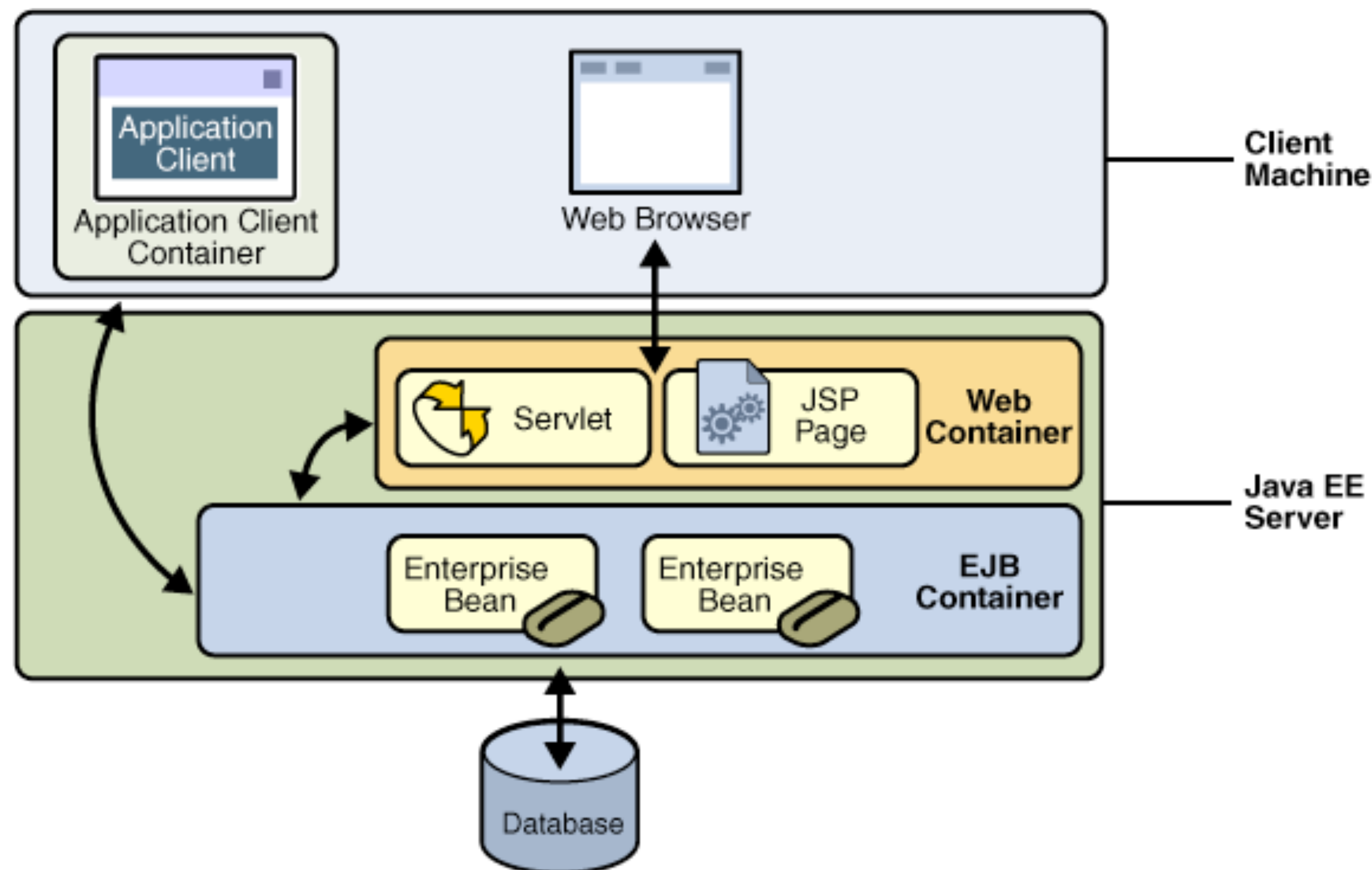
Java EE

- JDK 1.4 (2002) → J2SE, J2EE (inclus EJB v2.1)
- JDK 1.5 (2006) → Java SE, **Java EE** (inclus EJB v3)
- Depuis Java EE 5 on est passé à la spécification de composants métiers EJB3 qui apporte de nombreux changements lors de la conception des “beans” par rapport aux versions précédentes :
  - **Utilisation des annotations** (@local, @Entity, ...).
  - Utilisation des génériques (typage fort des collections).
  - Elimination de la plupart des descripteurs de déploiement.
  - → Simplification générale du processus de développement des applications d’entreprise.
- Mais compatibilité ascendante avec EJB < 3 (bean v3 client d’un bean < v3)

Java EE

Conteneurs

- La spécification Java EE définie un conteneur pour chaque type de composants:
  - Un conteneur pour tous les composants web (“Web Container”)
  - Un conteneur pour tous les composants EJB (“EJB Container”)



Java EE
EJB3

- La spécification 3 des EJB a été élaborée en vue de simplifier la conception d'EJB des cotés serveur et client.
  - **Coté client** : simplification de l'accès aux objets distants.
  - **Coté serveur**:
    - Simplification de la définition des interfaces, suppression d'un bon nombre de points requis dans la version 2.1 (plus besoin d'hériter une interface ou classe pour déclarer un bean).
    - Simplification pour la création de la classe du bean.
    - Simplification des APIs pour l'accès à l'environnement d'un bean.
    - Simplification de la gestion de la couche de persistance des beans.
    - Introduction des annotation pour remplacer les descripteurs XML de déploiement des beans.
- Nota: Le modèle EJB n'est pas lié au modèle JavaBeans vu précédemment.

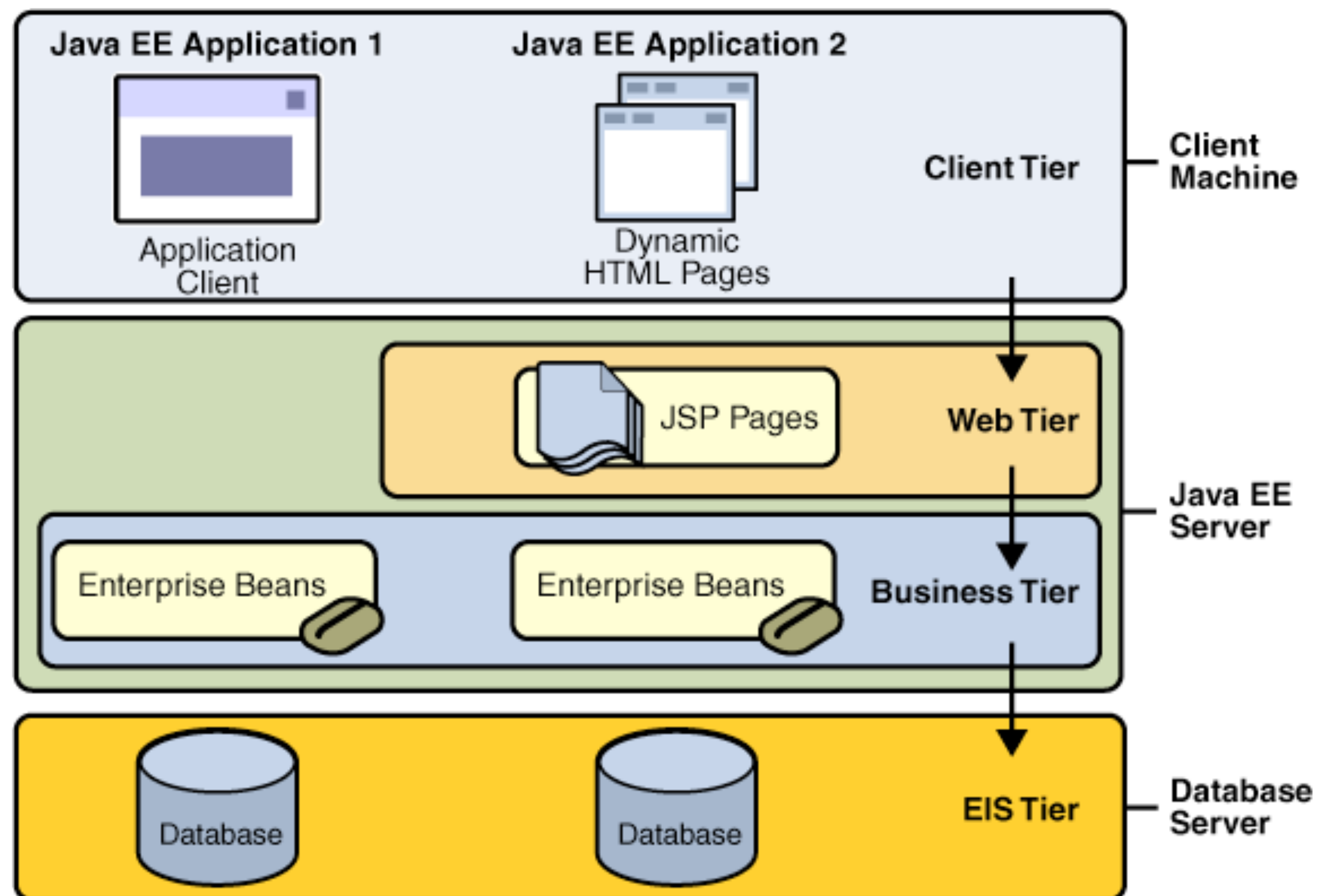
Java EE
... réparti ?

- Dans le cadre de Java EE la répartition se situe à 2 niveaux différents :
  - **On peut parler d'architecture (4-tiers) répartie** car la logique d'une application Java EE est divisée en "composants"
    - dont l'hébergement est réparti sur différentes machines (client, serveur J2EE, serveur de données),
    - en fonction de leurs rôles parmi les 4 tiers logiques spécifiés par Java EE (client, web, métier, système d'information).
  - **On peut parler de composants répartis**, car les EJB sont accessibles à distance par des client, et ceci de manière transparente pour les clients (tout comme un objet réparti RMI).

    Les EJB, bien que manipulés coté clients, sont hébergés dans le conteneur d'EJB du coté du serveur d'applications.

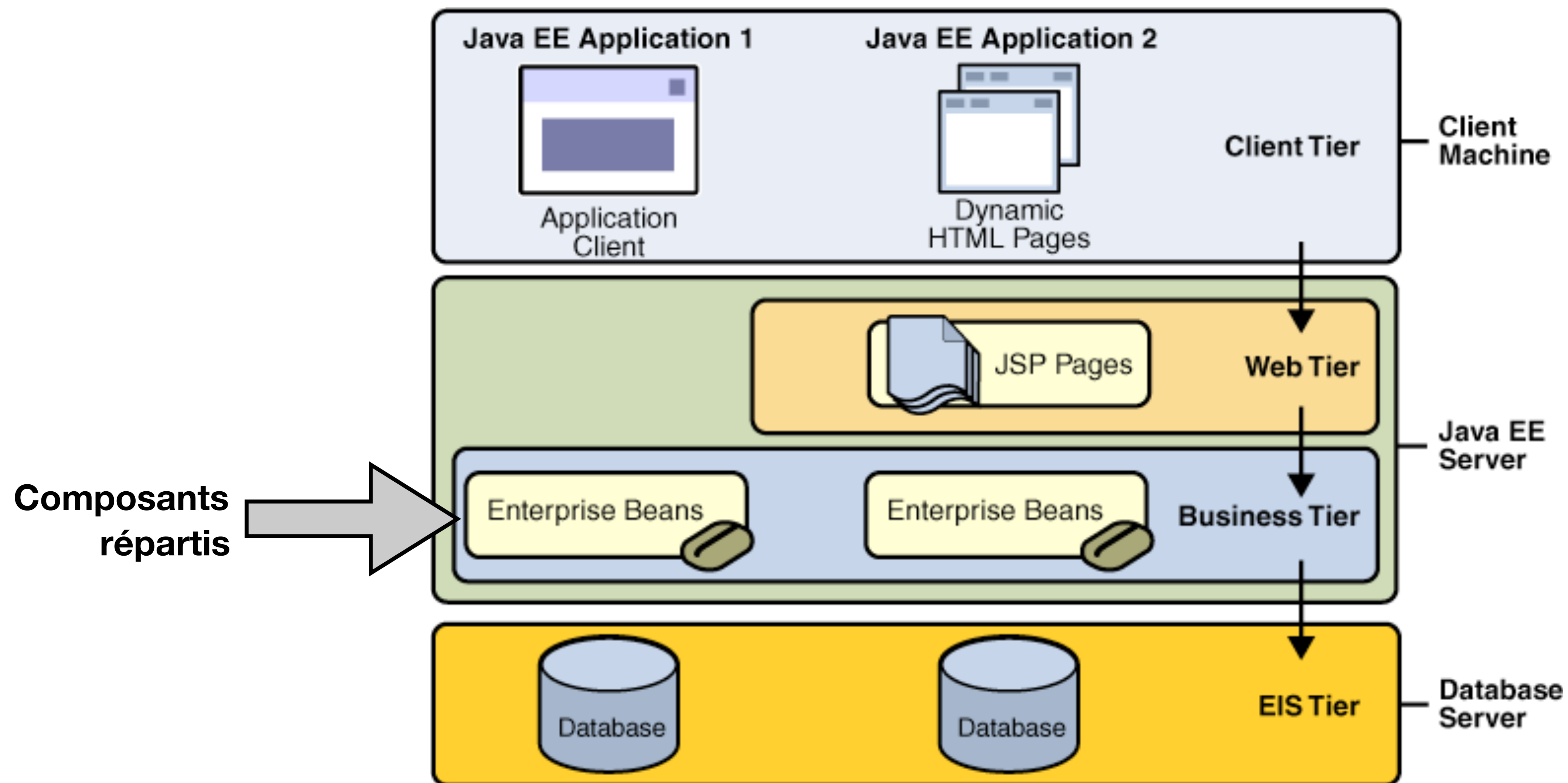
Java EE

Architecture 4-tiers répartie



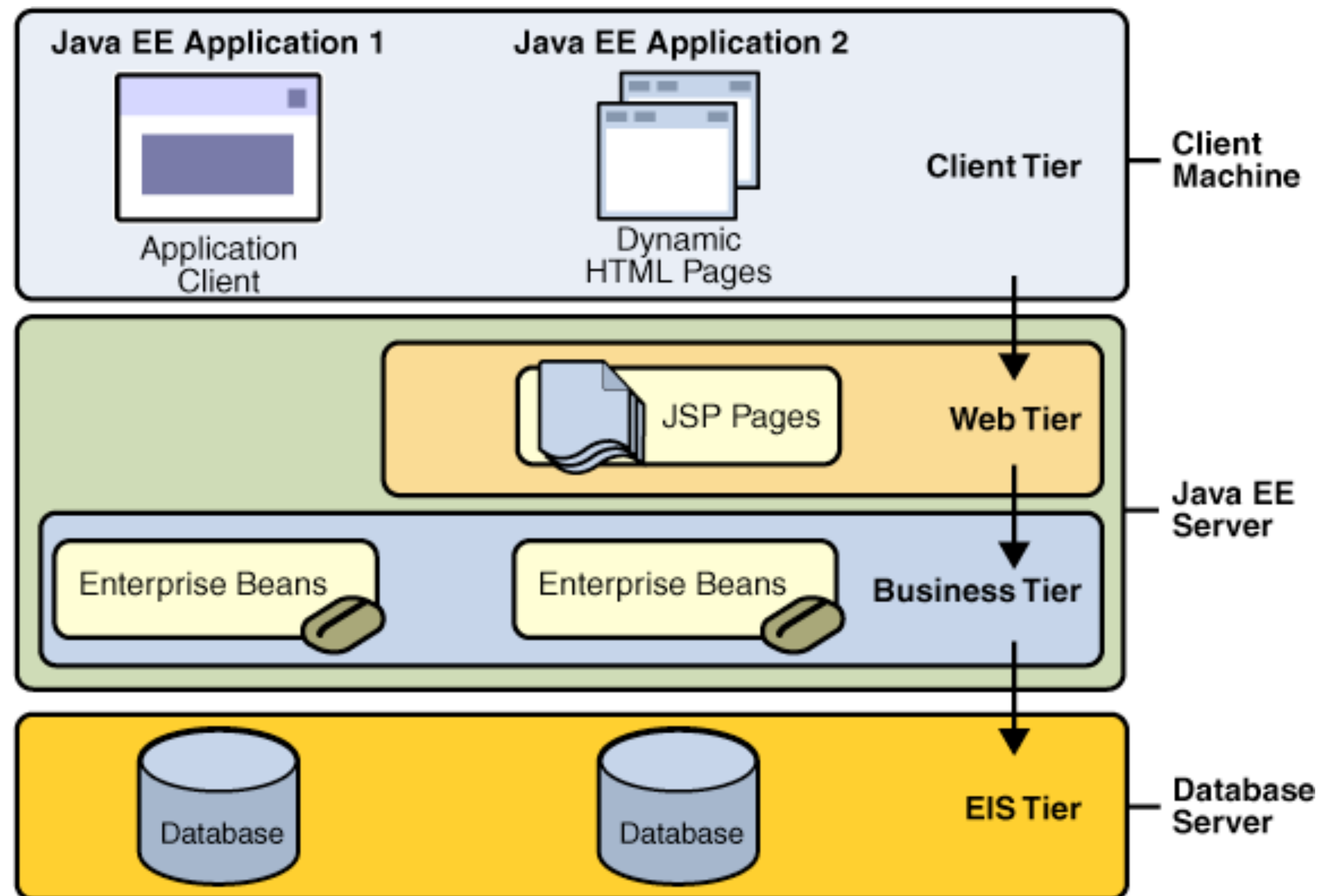
Java EE

Architecture 4-tiers répartie



Java EE
Client-Serveur

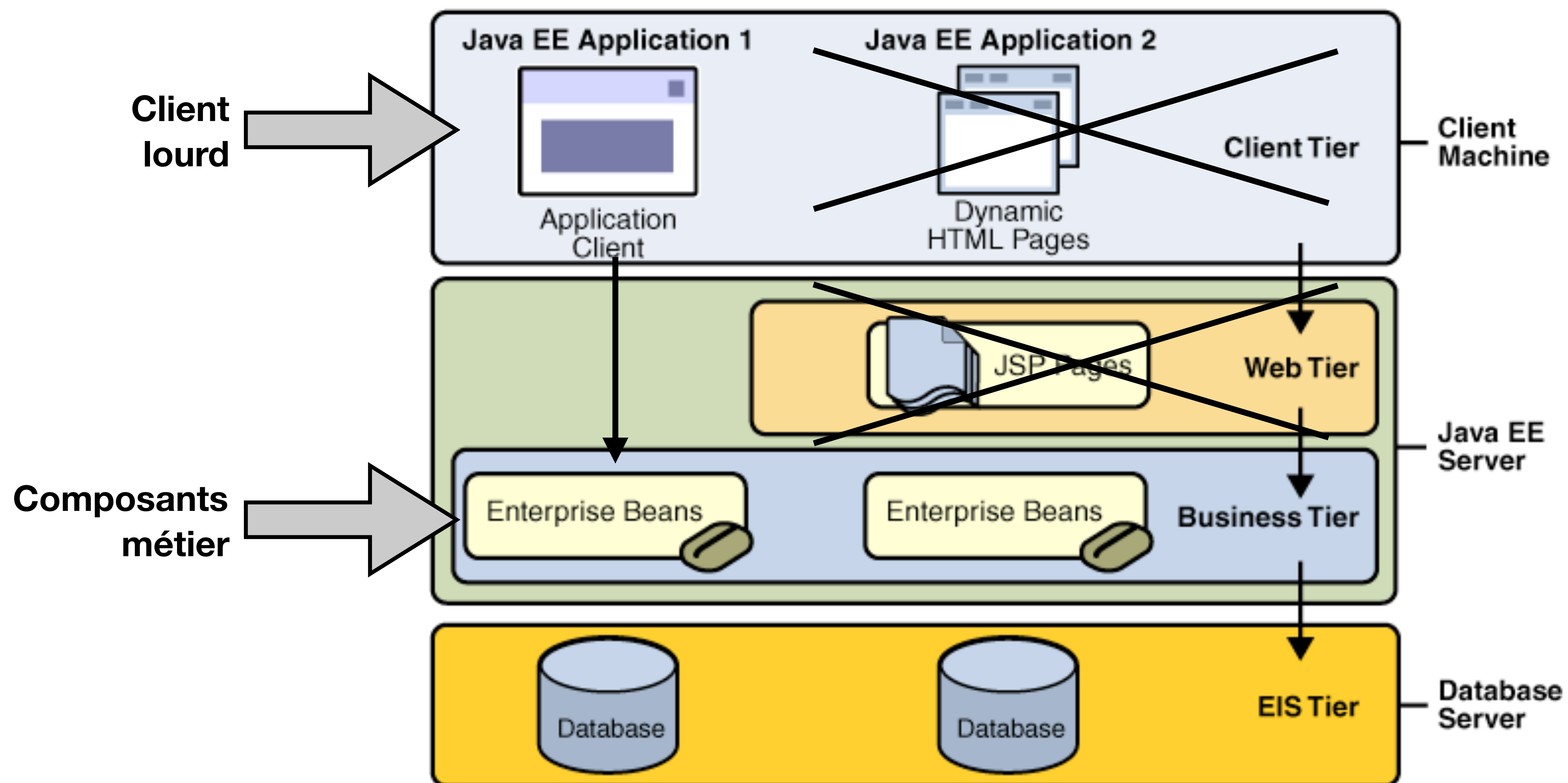
- Dans ce cours, nous nous intéressons à la liaison “directe” entre les clients lourds (Application Client) et les composants métier (“Enterprise Beans”) sans passer par la couche Web. On ne traitera pas JSP/Servlet.



Java EE

Client-Serveur

- Dans ce cours, nous nous intéressons à la liaison “directe” entre les clients lourds (Application Client) et les composants métier (“Enterprise Beans”) sans passer par la couche Web. On ne traitera pas JSP/Servlet.



Java EE
Serveur d’applications

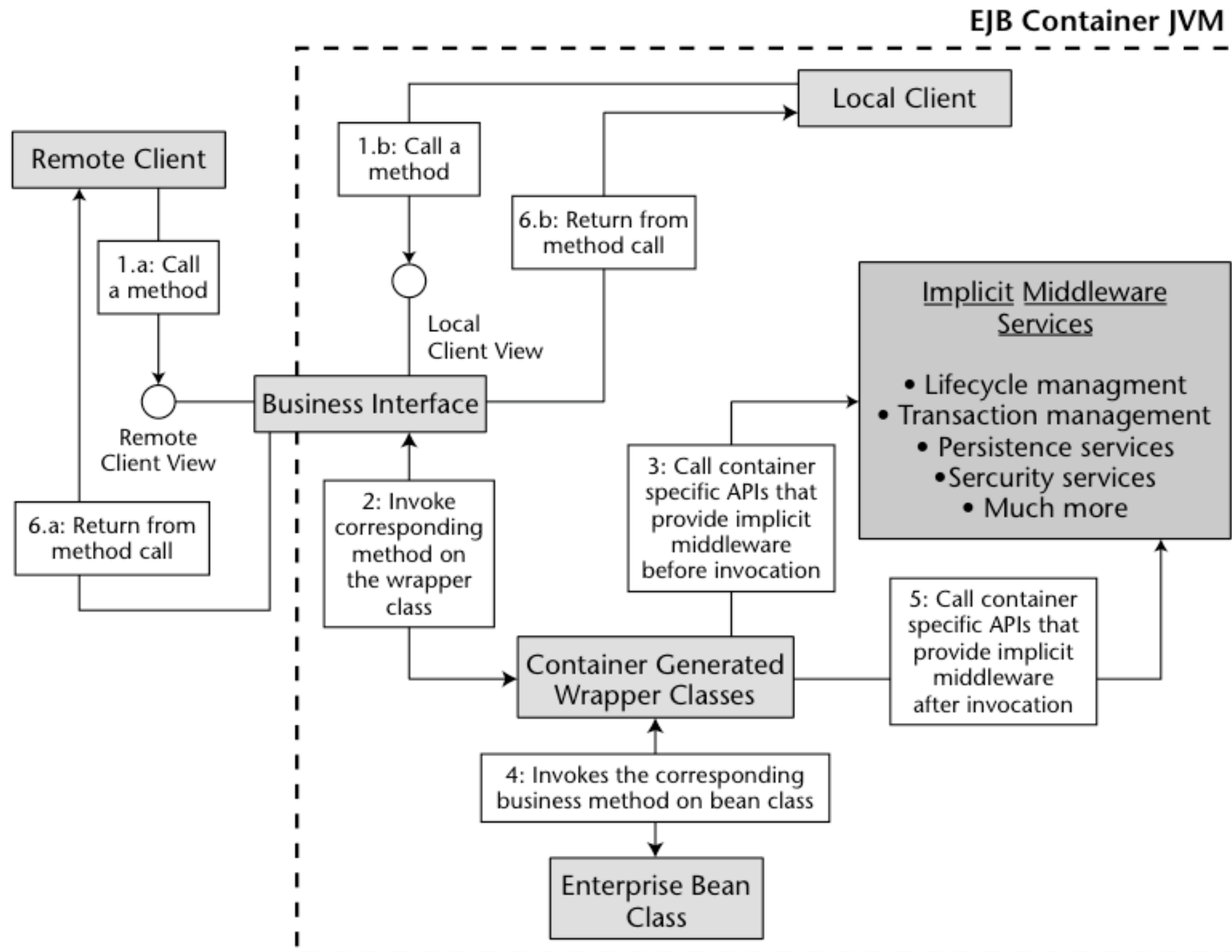
- Une implémentation de la spécification Java EE s’appelle un serveur d’applications.
- C’est le serveur d’applications qui héberge les conteneurs de composants et sous-traite les aspects transactionnels, de sécurité, ...
- A cette date, il existe plusieurs serveurs d’applications certifiés Java EE 5:
  - **Sun Java System Application Server Plateform Edition 9.0**
  - WebLogic Application Server 10.0, de BEA Systems.
  - SAP NetWeaver Application Server, de SAP.
  - JEUS 6, de TmaxSoft
  - Version beta de JBoss pour Java EE 5, mais pas encore certifiée.

Java EE
Notion d'application

- Quelles sont les types d'application servis par un serveur d'application ?
  - Une application complète Java EE est constituée notamment de composants Web, de composants métiers (les EJBs) et de bibliothèques diverses.
- Une application ne peut être déployée que sur un seul serveur.
- Mais elle peut communiquer avec d'autres applications situées ou non sur le même serveur.
- Cette communication peut prendre place entre applications clientes et composants serveurs Java EE ou directement entre des composants.
  - Elle peut avoir lieu en local ("client" et "serveur" sur la même machine),
  - ou à distance (via le réseau).

Java EE

Modèle de programmation



Les Enterprise Beans

- Un Enterprise Bean est un composant métier écrit dans le langage Java, exécuté coté serveur, accessible à distance via RMI-IIOP (pas forcement client Java)
- Certains de ces composants ont la responsabilité d'**encapsuler la logique métier** de l'application. C'est à dire le code qui assure les services offerts par l'application.
  - Par exemple, dans le cadre d'une application de gestion d'inventaires, un enterprise bean pourraient implémenter la logique métier dans deux méthodes appelées `checkInventoryLevel` (vérifier le nombre de produits en stock) et `orderProduct` (commander un produit).
  - En invoquant ces méthodes les clients peuvent accéder aux services fournis par l'application.
- D'autres ont la fonction de r**eprésenter à un niveau métier les données** manipulées par l'application.
  - Par exemple, au lieu d'accéder directement à la base de données la méthode `orderProduct` utilise des composants de type `Product` liées aux données contenues dans la base.
- Avec EJB3 :
  1 Enterprise Bean = 1 classe Java simple [+ interface(s)] + annotations

Les Enterprise Beans

Annotations

- Les annotations Java sont des méta-données appliquées sur votre code.
- Elles permettent d’apporter des informations supplémentaires ou de caractériser une classe, une méthode, une variable ...
- Les annotations sont utilisées pour apporter une information contextuelles.
- Les commentaires dans le code correspondent à une forme basique d’annotations dans la mesure où ils ne sont pas interprétés par le compilateur/interpréteur.
- Autres exemples d’annotations : JavaDoc /** **/, les directives du pré-processeur dans le langage C.
- **Java 1.5 introduit les annotations sous la forme** `@annotation[(données)]`.
  Elles peuvent être appliquées à des classes, des méthodes, des variables, des constructeurs, ...
- **Java EE 5 définit un ensemble d’annotations spécifiques pour les EJB.**

Les Enterprise Beans

Différents types de composants

- **Les composants Session** ("Session Beans", EJB Session) **:**
  - **Modélisent les traitements métiers.**
  - Interface avec les composants web ou applications clientes.
  - Avec ou sans état transactionnel (stateful, stateless).
- **Les composants Entités** ("Entity Beans", beans entités)**:**
  - **Représentent les objets métiers.**
  - Interface avec la base de données.
- **Les composants "Message-Driven"** ("Message-driven Beans")**:**
  - Gestion asynchrone, événementielle.

Session Bean

- Les EJB session modélisent les traitements métier d'une application répartie Java EE.
- Un EJB session est associé à un client unique **à un instant donné**, il ne peut être partagé.
- Sa durée de vie est celle de la session du client: de l'obtention de sa référence jusqu'à la fin de son utilisation.
- Il n'est pas persistant : il disparaît également lorsque le serveur d'application s'arrête de fonctionner.
- On peut comparer les EJB session à des verbes car ils exécutent des actions.
- On peut citer comme exemples d'EJB session : un système de commandes, un service d'autorisation pour cartes de crédit ou un service de gestion de portefeuille d'actions.

Session Bean
Conception

- L'élaboration d'un EJB session s'effectue simplement :

  1. **Créer la classe de l'EJB**, dite "Bean class".

     (Elle doit comporter un constructeur public sans argument)

  2. **Créer les interfaces** de l'EJB, dont au moins l'une est destinée à lui servir d'interface métier (distante ou locale).

     (Le nom des méthodes ne doit pas commencer par "ejb")

  3. **Préciser les annotations** adéquates sur la classe et sur les interfaces.

Session Bean
Interface métier

- Un EJB session est accessible par des clients, il faut donc définir dans une **interface métier** ("Business Interface") les méthodes qui seront visibles pour les clients.
- Un EJB session peut être accédé à distance (via le réseau) ou en local, il peut donc posséder simultanément :
  - **une interface métier distante :**
    - annotation `@Remote(Interface.class)` sur la classe de l'EJB
    - ou alors, annotation `@Remote` directement sur l'interface métier
  - **une interface métier locale :**
    - annotation `@Local(Interface.class)` sur la classe de l'EJB
    - ou alors, annotation `@Local` directement sur l'interface métier

Session Bean
Interface métier

- Si les annotations `@Remote()/@Local()` sont utilisées sur la classe de l'EJB et aucune annotation n'est portée sur les interfaces métiers :
  - L' EJB n'est pas obligé de préciser en plus qu'il implémente ces interfaces (instruction `implements` Java).
  - Mais il est **fortement recommandé** d'indiquer quand même explicitement l'implémentation (pour les vérifications à la compilation).
- Si l'EJB implémente une seule interface, elle est considérée comme interface métier.
  - Si le type de cette interface n'est pas précisé (local ou remote) alors elle est considérée comme locale par défaut.
- Si plus d'une interface implémentée par l'EJB, alors l'interface métier doit être désignée par utilisation d'une annotation.
- Une même interface ne peut être à la fois interface locale et distante du même EJB, **mais elles peuvent hériter d'une interface commune**.

Session Bean
Client

- Le client d'un EJB session peut être **une application**, un composant web ou un autre EJB.
- Le client obtient une référence sur l'interface métier (locale ou distante) de l'EJB, au choix :
  - Par un *lookup* JNDI (comme avec la spécification EJB 2.x)
  - Par une injection de référence sur l'EJB : mécanisme bien plus simple mais **uniquement possible si le client est lui-même un EJB**.
- Il invoque ensuite les méthodes métiers sur l'EJB.
- Lorsqu'il a fini d'utiliser un EJB session stateless, le client ne le signale pas au conteneur.
- Lorsqu'il a fini d'utiliser un EJB session stateful, le client doit le signaler au conteneur en invoquant une méthode annotée `@Remove` : l'instance sera ensuite détruite par le conteneur.

Session Bean
Client

- **Accès via JNDI :**
  - Lorsqu'un EJB est déployé dans le serveur, il est enregistré dans le service de nommage (JNDI) du serveur d'applications.
  - Par défaut, **le nom JNDI de l'EJB est celui de son interface métier**.
  - Si l'EJB déclare deux interfaces métiers, il possède alors un nom JNDI pour chacune des interfaces.
  - Le client de l'EJB va effectuer un *lookup* JNDI en précisant le nom JNDI de l'interface métier de l'EJB.

```
Context ctx = new InitialContext();
HelloRemote hello = (HelloRemote)
     ctx.lookup(“org.chatelp.stateless.HelloRemote”);
```

Session Bean
Client

- **Accès par injection de référence (plus simple):**
  - l'injection de référence d'EJB est un mécanisme par lequel le conteneur trouve des interfaces métier d'EJB, lui permettant ensuite d'initialiser des champs ou méthodes "setter" d'un client situé dans le conteneur.
  - Le type du champ et son nom suffisent généralement pour permettre au conteneur de trouver la référence demandée (sinon paramètres de l'annotation).
  - Utilisation de l'annotation @EJB :

```
package org.chatelp.stateless;

@Stateless
public class HelloBean implements HelloRemote {

   @EJB WorldRemote worldBean; //defined in the same package

   public String hello() {
       System.out.println("SessionBean : hello() method called");
       return "Hello "+worldBean.world();
   }
}
```

Session Bean
Client

- **Accès par injection de référence (plus simple):**
  - l’injection de référence d’EJB est un mécanisme par lequel le conteneur trouve des interfaces métier d’EJB, lui permettant ensuite d’initialiser des champs ou méthodes “setter” d’un client situé dans le conteneur.
  - Le type du champ et son nom suffisent généralement pour permettre au conteneur de trouver la référence demandée (sinon paramètres de l’annotation).
  - Utilisation de l’annotation @EJB :

```
package org.chatelp.stateless;

@Stateless
public class HelloBean implements HelloRemote {

   @EJB WorldRemote worldBean; //defined in the same package

   public String hello() {
       System.out.println("SessionBean : hello() method called");
       return "Hello "+worldBean.world();
   }
}
```

Session Bean

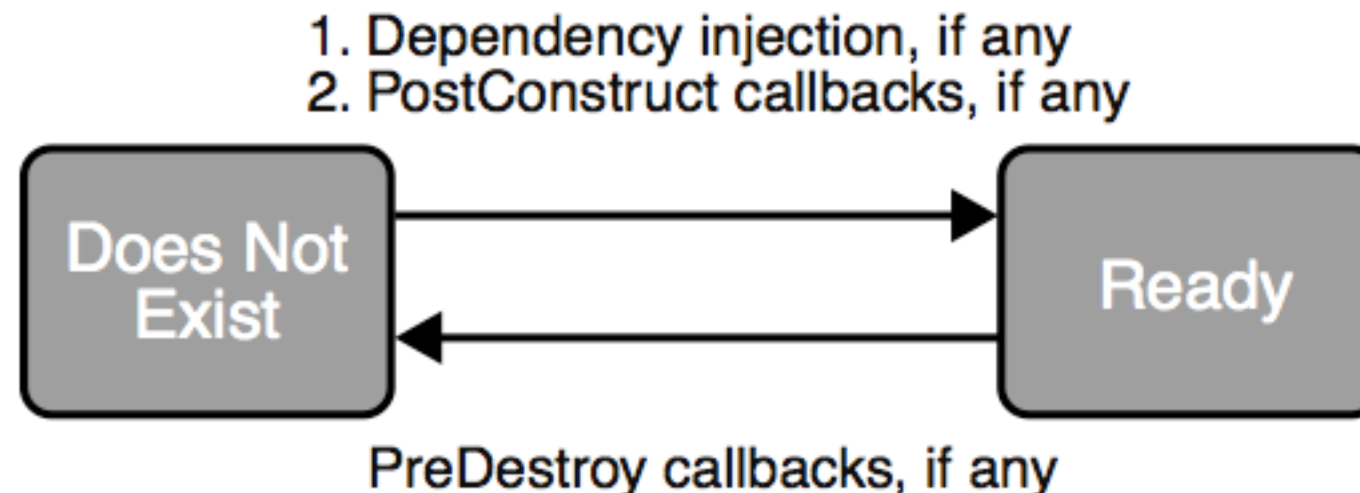
Avec ou sans état

- **EJB session sans état** (“stateless”) :
  - La durée de vie minimale d’un EJB session sans état correspond à l’exécution d’une méthode, il ne peut conserver d’état conversationnel entre deux invocations.
  - Toutes les instances d’un même type de bean sont équivalentes → elles sont interchangeables, le serveur d’application peut retirer un composant de la mémoire entre deux invocations de méthodes.
  - Les attributs d’un EJB stateless doivent représenter des données communes aux utilisateurs, et pas individuelles.
- **EJB session avec état** (“stateful”) :
  - Il conserve un état transactionnel, encapsulé **via ses variables d’instance**, au travers de plusieurs invocations de ses méthodes **par un même client (attention !)**.

Stateless Session Bean

Cycle de vie

- L'état inexistant ("Does Not Exist") signifie que l'EJB n'a pas encore été créé.
- L'état prêt ("Ready") est obtenu après instanciation et injections de dépendances éventuelles.
- Le conteneur adapte le nombre d'instances en fonction du nombre de requêtes clientes.
- Lorsqu'un EJB est sollicité par une requête cliente, le conteneur associe une instance du pool à l'objet local ou remote pour la durée d'exécution de la méthode.



Stateless Session Bean
Conception

- La classe du bean doit être annotée avec `@Stateless`.
- Ce type de bean supporte les callbacks d'événements suivants liés à son cycle de vie:
  - `@PostConstruct` : appelé après toutes les dépendances d’injection effectuées par le conteneur et avant le premier appel d’une méthode métier.
  - `@PreDestroy` : appelé au moment où l’instance du composant est détruite.
- **Nota** : un callback d’événement EJB3 est une méthode Java standard annotée, elle est appelée par le conteneur du composant pour le notifier d’un changement lié à son état.

Stateless Session Bean

Exemple

- Interface métier du bean :

```
package org.chatelp.stateless;

public interface HelloRemote {
   public String hello();
}
```

- Classe du bean

```
package org.chatelp.stateless;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote(HelloRemote.class)
public class HelloBean implements HelloRemote {
   public String hello() {
      System.out.println("method hello() called");
      return "Hello, World!";
   }
}
```

Stateless Session Bean

Exemple

- Client (on n'utilise pas l'injection, ce n'est pas un bean):

```
package org.chatelp.stateless;
import javax.naming.Context;
import javax.naming.InitialContext;

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

        /*
         * Obtain the JNDI initial context.
         * The initial context is a starting point for connecting to a JNDI tree.
         */
        Context ctx = new InitialContext();
        HelloRemote hello = (HelloRemote)
              ctx.lookup("org.chatelp.stateless.HelloRemote");

        /*
         * Call the hello() method on the bean. We then print the result to the screen.
         */
        System.out.println(hello.hello());
    }
}
```

Stateful Session Bean

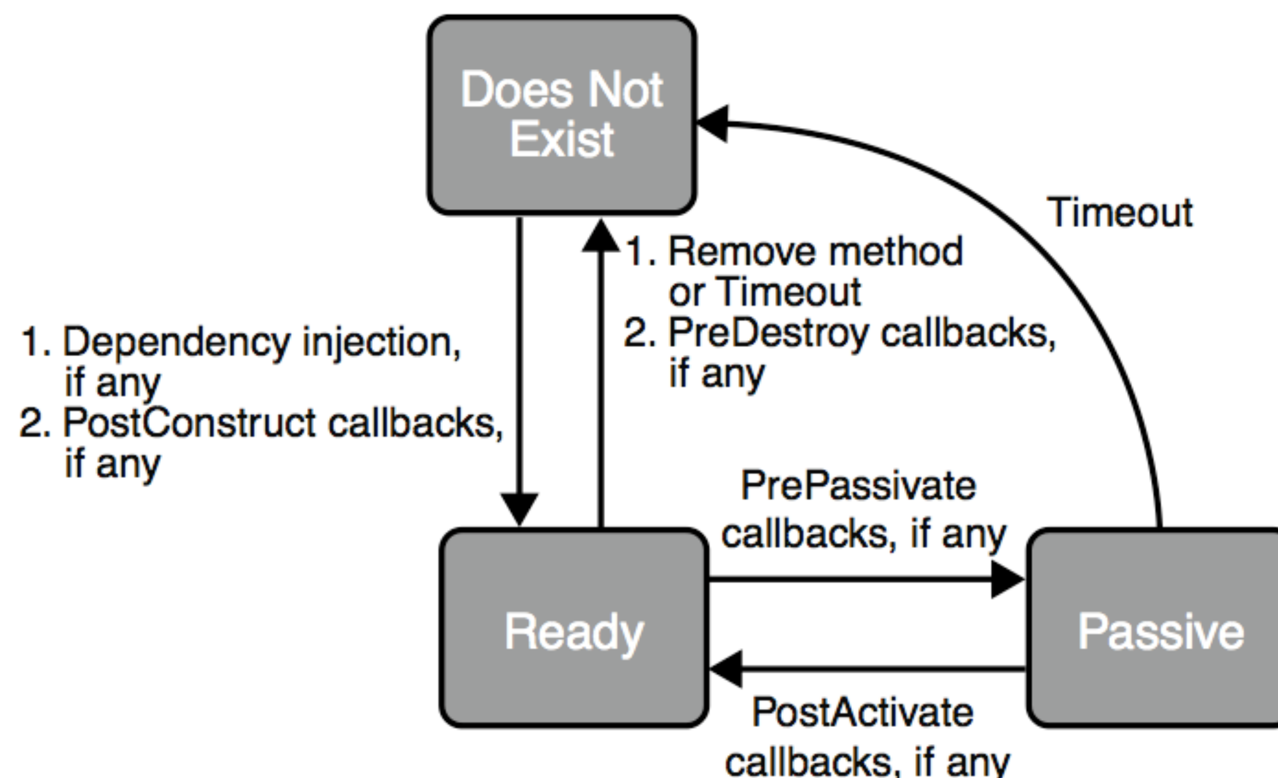
Cycle de vie

- Le conteneur d'EJB cherche toujours à limiter le nombre d'instances d'EJB en mémoire de façon à disposer de ressources suffisantes en toutes circonstances.
- Contrairement aux EJB stateless, les EJB stateful **ne sont pas interchangeables**. Le conteneur exploite alors un pool d'instances de quelques EJBs pour un nombre très supérieur de clients.
- Pour ce faire, il utilise un mécanisme de passivation :
  - Cela consiste à sauvegarder temporairement sur disque **l'état conversationnel** d'un EJB stateful. L'instance dont l'état vient d'être sauvegardé est alors disponible pour un autre client.
  - Quand le premier client sollicite à nouveau l'EJB, le conteneur choisit une instance disponible et y restaure l'état de l'EJB avant la dernière opération de passivation : c'est **l'activation**.

Stateful Session Bean

Cycle de vie

- **Le mécanisme de passivation s'appuie sur la sérialisation Java des instances** : la classe Bean d'un EJB stateful devrait implémenter `java.io.Serializable` (**mais la plupart des conteneurs se débrouillent sans l'implémentation explicite**).
- Si certains champs ne sont pas sérialisables (sockets, ...) alors le developpeur doit fournir du code supplémentaire de façon à restaurer l'état de ces champs (grâces aux callbacks d'événements, cf. transparents suivants).



Stateful Session Bean

Conception

- La classe du bean doit être annotée avec `@Stateful`.
- Ce type de bean supporte les callbacks d'événements suivants liés à son cycle de vie :
  - `@PostConstruct` : appelé après les injections de dépendances et avant le premier appel d’une méthode métier.
  - `@PreDestroy` : appelé au moment où l’instance du composant est détruite.
  - `@PrePassivate` : appelé lorsqu’il y a trop de composants instanciés. Il faut donc stocker l’état du bean.
  - `@PostActivate` : appelé pour restaurer les composants qui auraient été passifiés.
- Le client d’un bean peut appeler une méthode annotée `@Remove` pour enclencher la suppression du bean lorsque cette méthode aura fini sont exécution (normale ou anormale) → appel auto. à `@PreDestroy`.

Stateful Session Bean

Exemple

- Interface métier du bean :

```
package org.chatelp.stateful;
/**
* The business interface - a plain Java interface with only
* business methods.
*/
public interface Count  {
   /**
   * Increments the counter by 1
   */
   public int count();
   /**
   * Sets the counter to val
   * @param val
   */
   public void set(int val);
   /**
   * removes the counter
   */
   public void remove();
}
```

Stateful Session Bean

Exemple

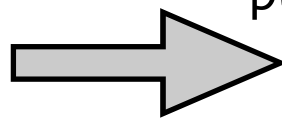
- Classe du bean

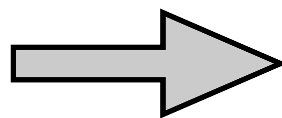
```
package org.chatelp.stateful;
import javax.ejb.*;
@Stateful
@Remote(Count.class)

public class CountBean implements Count {
  private int val; //the current counter is our conversational state.

  public int count() {
    System.out.println(“count() called”);
    return ++val;
  }

  public void set(int val) {
    this.val = val;
    System.out.println(“set() called”);
  }

  @Remove
  public void remove() { System.out.println(“remove() called”); }
}
```

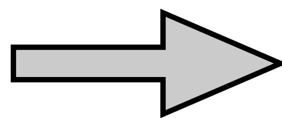
Stateful Session Bean

Exemple

- Classe du bean

Etat du bean 

Modification de l'état 

Modification de l'état 

```
package org.chatelp.stateful;
import javax.ejb.*;
@Stateful
@Remote(Count.class)

public class CountBean implements Count {
  private int val; //the current counter is our conversational state.

  public int count() {
    System.out.println(“count() called”);
    return ++val;
  }

  public void set(int val) {
    this.val = val;
    System.out.println(“set() called”);
  }

  @Remove
  public void remove() { System.out.println(“remove() called”); }
}
```

Stateful Session Bean

Exemple

- Client :

```
package org.chatelp.stateful;
import javax.naming.*;

public class CountClient {
   public static void main(String[] args) {
      try {
         /* Get a reference to the bean */
         Context ctx = new InitialContext();

         Count counter = (Count) ctx.lookup(Count.class.getName());

         counter.set(0);
         System.out.println(counter.count()); // --> 1
         System.out.println(counter.count()); // --> 2
         System.out.println(counter.count()); // --> 3
         System.out.println(counter.count()); // --> 4
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Session Bean
Accès local ou distant

- En accès distant, les arguments et les résultats des méthodes sont transmis par valeur (donc sérialisés).
- En accès local, les arguments et les résultats des méthodes sont transmis de la même façon qu'en Java standard :
  - par valeur pour les types primitifs.
  - par référence pour les objets.
- L'accès local à un EJB n'est possible que si le client se situe dans la même JVM que le conteneur de l'EJB : ce peut être un autre EJB ou un composant web (pas étudié dans le cadre ce cours).

Session Bean
Accès local ou distant

- Le choix entre local ou distant doit tenir compte des considérations suivantes :
  - L'accès distant permet de s'affranchir de la localisation physique de l'EJB.
  - En accès distant les objets transmis aux méthodes de l'EJB et retournés par elles doivent être sérialisables.
  - L'accès distant est penalisant en terme de performance (couche réseau, sérialisation, désérialisation, ...).
  - En accès local les objets sont transmis par référence, ce qui entraine des risques d'altération **concurrente** de ces objets (cf. cours #2.2).
  - L'accès local impose que le client et l'EJB **soient exécutés dans la même machine virtuelle**, ce qui restreint la distribution des composants.

Session Bean

Identité

- Un EJB session (stateful ou stateless) possède une identité unique allouée par le conteneur lors de la création de l'EJB.
- Le client d'un EJB peut déterminer si deux références sur un EJB session désignent ou non le même EJB : méthode `equals` entre les références :
  - Instances distinctes pour chaque référence dans le cas d'un EJB stateful (obligatoire pour maintenir un état conversationel).
  - Instances identiques pour chaque référence dans le cas d'un EJB stateless (puisqu'ils sont tous interchangeables).

```
@EJB Toto toto1;
@EJB Toto toto2;
...
if(toto1.equals(toto2)) {
   //le résultat de la comparaison dépend du type EJB session consideré
}
```

Entity Bean

- En EJB de type entité représente nécessairement des données métiers persistantes de l'application, comme un compte bancaire, un fournisseur, un article.
- Ce type d'EJB sera utilisé par un client pour accéder ou mettre à jour les informations concernant ces données.
- On peut les comparer à des noms auxquels sont appliqués les verbes représentés par les session beans.
- Un EJB entité est partagé par plusieurs clients.
- En pratique, il est associé à une base de données objet ou relationnelle.
- Un EJB entité comporte des attributs associés à des champs d'une ou plusieurs tables d'une base de données relationnelle (mapping Objet/Relationnel).

Entity Bean

- Un EJB entité n'est que la représentation sous la forme d'un objet d'un enregistrement en base de données (→ nécessité d'un nom unique par entité).
- **En EJB 3, un EJB entité est un simple objet Java (POJO) comportant des annotations liées au "mapping O/R".**
- **Dans la suite de ce cours, il sera appelé "bean entité", compte tenu qu'il ne s'agit plus d'un composant EJB au sens propre comme dans la spécification 2.x.**
- La persistance de l'EJB est maintenant assurée par un service appelé EntityManager.
- La plupart du temps, les beans entités sont gérés par des EJB session.

Entity Bean
Conception

- L'élaboration d'un bean entity s'effectue simplement :

  1. **Créer une classe comme pour les Java Beans** (propriétés privées accessibles au besoin par méthodes getX/setX publiques)

     (Elle doit comporter un constructeur public sans argument, et le nom des méthodes ne doit pas commencer par "ejb")

  2. **Préciser les annotations** adéquates sur la classe ou renseigner un descripteur de déploiement.

- Le développeur élabore la classe du bean en précisant son type par une annotation : @javax.ejb.Entity
- Cette annotation indique que les instances de cette classe pourront être prises en charge par un service de persistance EntityManager.

Entity Bean
Conception

- Bien que rigoureusement non obligatoire, il est souhaitable que la classe implémente l'interface java.io.Serializable, cela permet aux instances de transiter sur le réseau entre le conteneur et un client distant.
- Par défaut le bean entité est associé à une table du nom de sa classe.
- Le nom des colonnes est par défaut celui du nom de ses propriétés.
- Un EJB entité est toujours associé à une clé primaire ("primary key"), qui permet de l'identifier de façon unique dans la base de données.
- Cette clé primaire doit être un champ du bean entité.
- La méthode getX d'accès à ce champ doit être annoté avec @java.persistance.Id

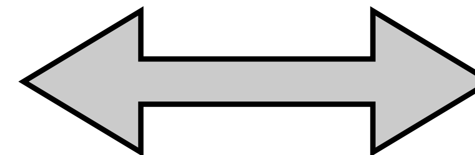
Entity Bean

Exemple 1

- Dans l'exemple suivant :
  - La classe Personne est associée à une table Personne.
  - Chacune des propriétés sont mappées vers des colonnes de mêmes noms dans la table Personne:
    - id
    - nom
    - prenom
  - La **clé primaire** (identifiant unique) est désignée par la propriété id.

Entity Bean
Exemple 1


```
@Entity
public class Personne implements Serializable {
    private int id;
    private String nom;
    private String prenom;

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

    public String getNom() { return nom; }
    public void setNom(String nom) { this.nom = nom; }

    public String getPrenom() { return prenom; }
    public void setPrenom(String prenom) { this.prenom = prenom; }
}
```



Personne		
id *(pk)*	nom	prenom
123	Foo	Bar

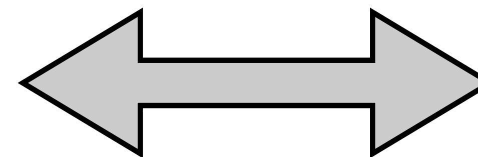
Entity Bean
Exemple 2

- Dans l’exemple suivant :
  - La classe Personne est associée à une table PERSONNEL.
  - Chacune des propriétés sont mappées vers des colonnes de noms différents dans la table PERSONNEL:
    - id → MATRICULE
    - nom → NOM
    - prenom → PRENOM_1
  - La clé primaire est désignée par la propriété id.
- Pour ce faire on paramétrise les annotations utilisées.

Entity Bean

Exemple 2


```
@Entity
@Table(name="PERSONNEL")
public class Personne implements Serializable {
    private int id;
    private String nom;
    private String prenom;

    @Id
    @Column(name="MATRICULE")
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }

    @Column(name="NOM")
    public String getNom() { return nom; }
    public void setNom(String nom) { this.nom = nom; }

    @Column(name="PRENOM_1")
    public String getPrenom() { return prenom; }
    public void setPrenom(String prenom) { this.prenom = prenom; }
}
```



PERSONNEL		
MATRICULE ***(pk)***	NOM	PRENOM_1
123	Foo	Bar

Entity Manager

- **L’instanciation d’une classe annotée avec @Entity n’entraîne pas automatiquement la persistante de l’instance.**
- Il faut faire intervenir l’EntityManager, service du middleware Java EE qui gère la persistance des entités en base relationnelle :
  - recherche, insertion et suppression d’entités
  - synchronisation entre instances et base de données
  - requêtes
  - gestion des caches
  - interaction avec les services de gestion des transactions

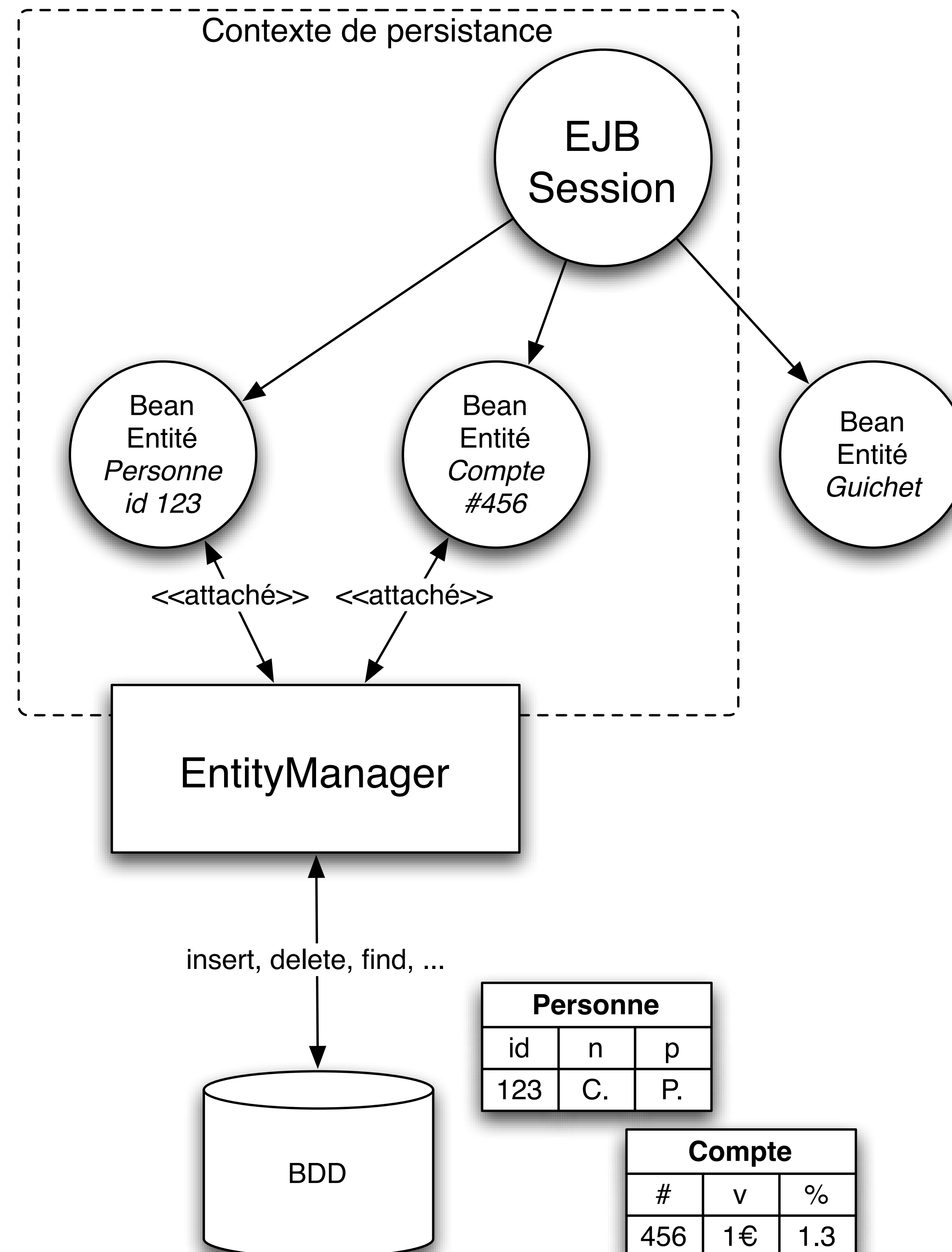
Entity Manager

Lien avec les beans entités

- Au cours de son cycle de vie, un bean entité peut être :
  - attaché à un EntityManager : l'état du bean et sa représentation en base de données sont synchronisées par l'EntityManager.
  - détaché d'un EntityManager : l'état du bean peut évoluer indépendamment de sa représentation en base de données.
- Un EntityManager est lui même associé à un **contexte de persistance** ("persistence context").
- La fermeture du contexte de persistance d'un entity manager entraîne le détachement de tous ses beans entité.

Entity Manager

Lien avec les beans entités



Personne

id	n	p
123	C.	P.

Compte

#	v	%
456	1€	1.3

Entity Manager
Contexte de persistance

- Il existe deux types de contexte de persistance :
  - **Le contexte de persistance de transaction** (“transaction-scoped persistence context”).
    - Les beans entités sont attachés à l’EntityManager le temps d’exécution d’une transaction (**en pratique celui d’une méthode**).
    - Ils deviennent détachés à la fin de la transaction.
    - Contexte habituellement utilisé dans le cadre des EJB session stateless.
  - **Le contexte de persistance étendu** (“extended persistence context”).
    - Les beans entités qui sont attachés à l’EntityManager le restent même après la fin de la transaction (d’un appel de méthode par ex.).
    - Contexte géré uniquement par des applications autonomes ou des EJB session stateful.

Entity Manager
Unité de persistance

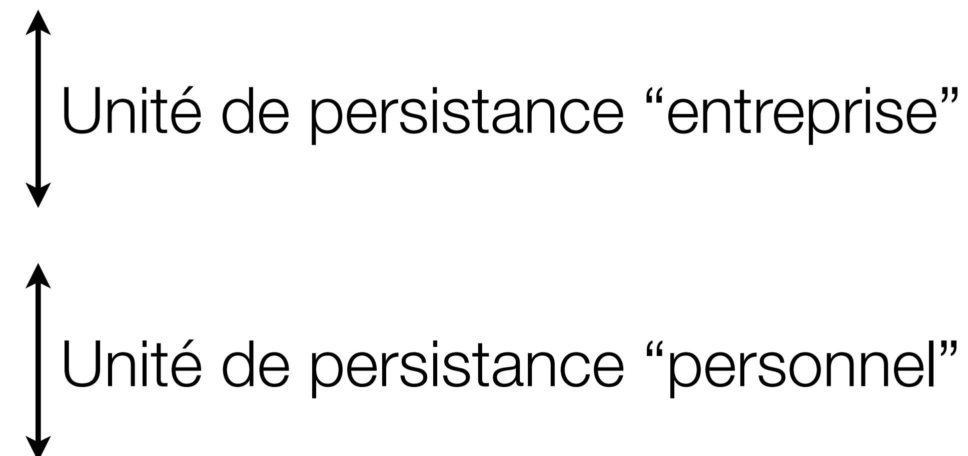
- Une unité de persistance est l'ensemble des beans entités d'une application associés à une même base de données.
- Une unité de persistance est définie dans un fichier persistence.xml.
- Le jeu de classes d'une unité de persistance est par défaut l'ensemble des classes annotés @Entity dans le fichier .jar de déploiement des composants de l'application.
- Le jeu de classes peut aussi être défini explicitement dans le fichier persistence.xml.

```
<?xml version=”1.0” encoding=”UTF-8”?>
<persistence>
   <persistence-unit name=”entreprise”>
   ...
   </persistence-unit>
   ...
   persistence-unit name=”personnel”>
   ...
   </persistence-unit>
</persistence>
```

Entity Manager
Unité de persistance

- Une unité de persistance est l’ensemble des beans entités d’une application associés à une même base de données.
- Une unité de persistance est définie dans un fichier persistence.xml.
- Le jeu de classes d’une unité de persistance est par défaut l’ensemble des classes annotés @Entity dans le fichier .jar de déploiement des composants de l’application.
- Le jeu de classes peut aussi être défini explicitement dans le fichier persistence.xml.

```
<?xml version=”1.0” encoding=”UTF-8”?>
<persistence>
   <persistence-unit name=”entreprise”>
   ...
   </persistence-unit>
   ...
   persistence-unit name=”personnel”>
   ...
   </persistence-unit>
</persistence>
```



Entity Manager
Obtention

- Un EntityManager peut s'obtenir :
  - A partir d'un EntityManagerFactory :

```
@Stateless
public class GestionBeta implements GestionBetaRemote {
   @PersistenceUnit(unitName="beta")
   private EntityManagerFactory factory;
   private EntityManager manager;

   public GestionBeta() { manager = factory.createEntityManager(); }
}
```

  - Recommandé : par injection de dépendance (mécanisme similaire à l'injection de référence) uniquement dans les EJB :

```
@Stateless
public class GestionRemote implements GestionPersonneRemote {
   @PersistenceContext(unitName="entreprise")
   private EntityManager manager;
   ...
}
```

Entity Manager

Obtention

- L'annotation @javax.persistence.PersistenceContext précise :
  - l'unité de persistance utilisée (unitName)
  - le type du contexte de persistance (type) : TRANSACTION (par défaut) ou EXTENDED.
- **Un EJB session stateless ou message-driven ne peut utiliser qu'un contexte de persistance de transactions.**
- **Un EJB session stateful peut aussi utiliser un contexte de persistance étendu :**

```
@Stateful
public class GestionAlpha implements GestionAlphaRemote {
   @PersistenceContext(
         unitName=”alpha”,
         type=PersistenceContextType.EXTENDED)
   private EntityManager manager;
   ...
}
```

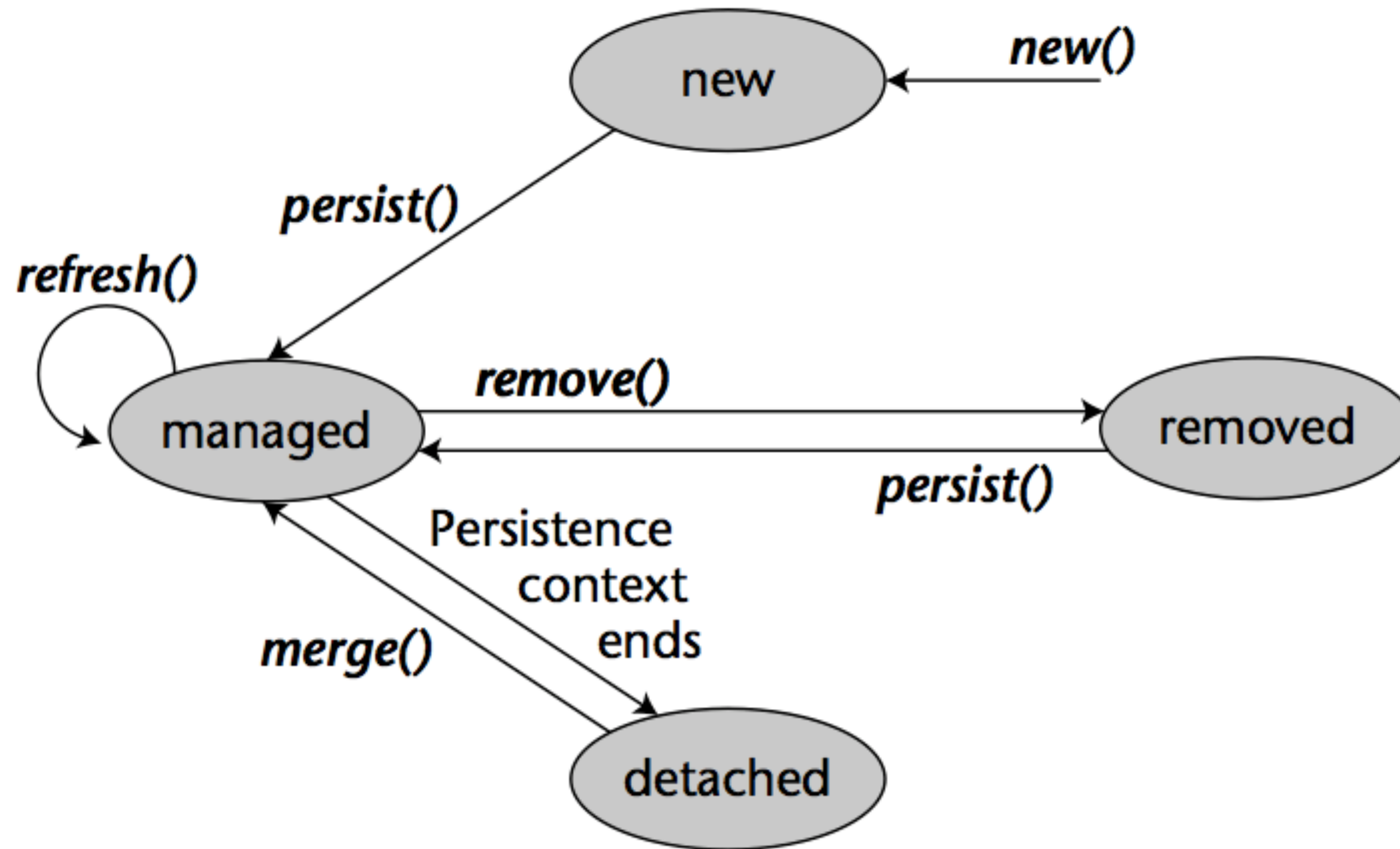
Entity Manager

Cycle de vie des entités

- “New” : l’instance de bean entité à été crée en mémoire mais n’est pas encore associée à un contexte de persistance ou à une base de données.
- “Managed” : le bean entité à une identité persistante dans la BDD et est actuellement associé à un contexte de persistance. Les changements apportés au bean seronts automatiquement synchronisés avec la BDD lorsque les transactions seronts “commitées” ou lorsque demandé explicitement par la méthode flush().
- “Detached” : le bean entité à une identité persistante dans la BDD mais n’est plus associé à un contexte de persistance.
- “Removed” : le bean entité est actuellement associé à un contexte de persistance mais il est marqué pour suppression de la BDD dés que possible.

Entity Manager

Cycle de vie des entités



Entity Manager

Interface

- La méthode qui permet d'attacher un bean entité est :
  `void persist(Object entity);`
  Le bean transmis en paramètre est inséré en base de données (INSERT SQL).
- La méthode qui permet de supprimer un bean entité de la base (DELETE SQL) est :
  `void remove(Object entity);`
- La méthode qui permet de synchroniser un bean entité à partir de sa représentation en base de données (SELECT SQL) est :
  `void refresh(Object entity);`
- La méthode qui permet de mettre à jour la BDD à partir d'un bean entité (UPDATE SQL) est :
  `<T> T merge(T entity);`
  Le bean entité fourni en paramètre reste détaché mais la méthode retourne un bean attaché à l'EntityManager.

Entity Manager

Exécution des requêtes

- Par défaut, lors de l’appel aux méthodes persist, merge, remove, l’EntityManager n’exécute la requête SQL qu’en fin de transaction (fin de méthode) ou si une autre requête sur les mêmes données intervient.
- Le choix du mode d'exécution s’effectue par la méthode :
  `void setFlushMode(FlushModeType flushMode);`
  - Le “flush mode” de l’EntityManager est de type “**AUTO**” par défaut.
  - Il est possible d’utiliser le mode “**COMMIT**” demandant à l’EntityManager de refléter les changements en base de données seulement en fin de transaction.
- Pour anticiper l'exécution par l’EntityManager de la requête SQL il faut faire appel à la méthode :
  `void flush();`

Entity Manager

Interface

- La méthode qui permet de savoir si un bean entité est attaché à un EntityManager est :
  `boolean contains(Object entity);`
- La méthode qui permet de détacher tous les beans entités d'un EntityManager est :
  `void clear();`
- **Nota** : il est prudent d'exécuter la méthode `flush` avant `clear` de façon à mettre à jour la base de données.
- Les méthodes permettant de rechercher un bean entité **à partir de sa clé primaire** sont :

```
<T> T find(Class<T> entityClass, Object primaryKey);
<T> T getReference(Class<T> entityClass, Object primaryKey);
```

- **Nota** : si le bean entité n'est pas trouvé `find` retourne null alors que `getReference` lance une exception `EntityNotFoundException`.

Entity Manager

Interface

- Il existe aussi des méthodes permettant de rechercher un bean entité **à partir de requêtes directes sur la base de données** :
  - Les méthodes suivantes recoivent une requête **EJB-QL** en argument :
    - Query createQuery(String qlString);
    - Query createNamedQuery(String name);
  - Les méthodes suivantes recoivent une requête **SQL** en argument :
    - Query createNativeQuery(String sqlString);
    - Query createNativeQuery(String sqlString, Class resultClass);
    - Query createNativeQuery(String sqlString, String resultSetMapping);
  - Utilisez ensuite la méthode Collection getResultList() sur une instance de Query donnée pour l'exécuter et obtenir le résultat.

Entity Manager

Exemple


```
@Stateless
public class GestionRemote implements GestionPersonneRemote {
   @PersistenceContext(unitName=”entreprise”)
   private EntityManager manager;

   public void registerPersonne(Personne p) {
      manager.persist(p);
   }

   public Personne findPersonne(int id) {
      //id est déclaré comme clé primaire / identifiant du bean entité
      //Personne
      return manager.find(Personn.class, id);
   }
}
```

Entity Bean

Relations entre beans entité

- Les beans entités modélisant des objets métiers, il doivent être capable de refléter des relations qui peuvent exister dans le monde réel.
- La spécification EJB 3.0 permet de modéliser des relations complexes entre les beans entités :
  - **Associations :**
    - Les associations entre objets dépendent du nombre d'objets participants de part et d'autre à l'association : ce que l'on appel la **cardinalité**.
    - Les associations dépendent aussi de la **navigabilité** : c'est à dire de la possibilité ou non pour un objet d'atteindre le(s) autre(s) objet(s) participant à la relation
  - **Héritage :** la spécification EJB3 supporte l'héritage entre classes de beans entités.

Entity Bean

Cardinalité d'une association

- On distingue quatre types de relations en fonction de leur cardinalité :
  - **La relation de un-vers-un** (one-to-one relationship) : il s'agit par exemple de la relation entre un client et une adresse, chaque client ayant une adresse, et à une adresse correspondant un client.
  - **La relation de un-vers-plusieurs** (one-to-many relationship) : il s'agit par exemple de la relation entre une société et ses employés, chaque société ayant (en général) plusieurs employés.
  - **La relation de plusieurs-vers-plusieurs** (many-to-many relationship) : il s'agit par exemple de la relation entre un client et des chambres d'hôtel, une même chambre pouvant être réservée par plusieurs clients (pas simultanément) et un même client pouvant réserver plusieurs chambres.

Entity Bean
Navigabilité d'une association

- La navigabilité indique le sens de navigation de la relation d'association.
- Le fait pour un objet A de pouvoir atteindre un objet B se traduit dans l'objet A par **la présence d'une référence** sur l'objet B.
- Si un objet A peut atteindre un objet B avec lequel il a par conséquent une relation d'association, l'objet B peut-il également atteindre l'objet A ?
- La navigabilité peut être :
  - **Unidirectionnelle** : un seul des objets participant à la relation possède une référence sur un objet participant à la relation.
  - **Bidirectionnelle** : chaque objet participant à la relation possède une référence sur son objet participant à la relation.

Entity Bean

Types associations

- La combinaison de la cardinalité et de la navigabilité conduit à 7 types de relations entre objets :

  1. Relation de un vers un, unidirectionnelle.
  2. Relation de un vers un, bidirectionnelle.
  3. Relation de un vers plusieurs, unidirectionnelle.
  4. Relation de un vers plusieurs, bidirectionnelle.
  5. Relation de plusieurs vers un, unidirectionnelle.
  6. ~~Relation de plusieurs vers un, bidirectionnelle~~ (= #4).
  7. Relation de plusieurs vers plusieurs, unidirectionnelle.
  8. Relation de plusieurs vers plusieurs, bidirectionnelle.

Entity Bean
Associations et BDD

- Les associations entre les EJB entités apparraissent nécessairement dans les schémas de base de données.
- Elles se traduisent généralement par la présence de colonnes supplémentaires dans les tables associées, voire des tables supplémentaires.
- Dans ce cas il faut utiliser l'annotation **@JoinColumn** pour indiquer la colonne permettant d'effectuer la jointure (le lien) entre les tables.
- Les associations sont indiquées par les annotations suivantes sur les méthodes getter des champs d'association : **@OneToOne, @OneToMany, @ManyToOne, @ManyToMany**.
- Ces annotations seront présentes dans les deux classes en relation sur la relation est bidirectionnelle, dans l'une des classes sinon.
- Un attribut "**cascade**" de ces annotations permet d'indiquer à l'EntityManager quelles opérations (**ALL, PERSIST, REMOVE, MERGE, REFRESH**) propager aux instances liées d'un bean entité.

Entity Bean

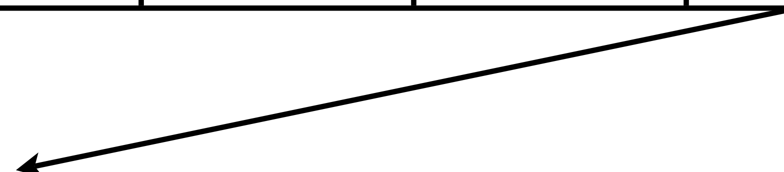
Exemple d’association

- Association un-vers-un unidirectionnelle entre Customer et Address :
  - Dans Customer uniquement :

```java
@OneToOne(cascade={CascadeType.ALL})
public Address getAddress() { return address; }
```

Customer

ID (PK)	firstname	lastname	ADDRESS_ID (FK)
1234	Eva	Luateur	**add_a**
5678	Ray	Cursif	**add_b**

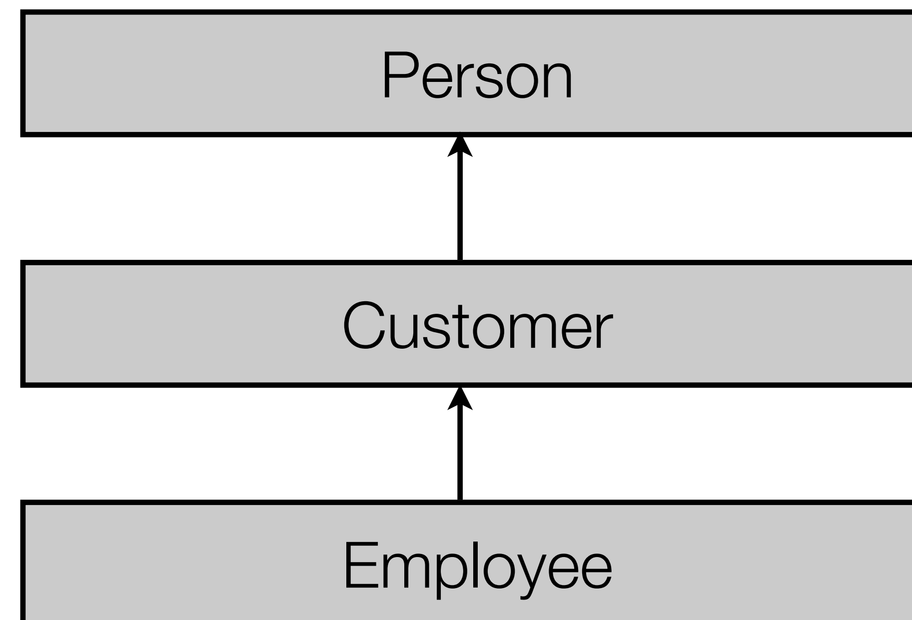


Address

ADDRESS_ID (PK)	number	road	city
add_a	12	Rue St Anne	Paris
add_b	5	Avenue Carnot	Massy

Entity Bean
Héritage

- La spécification EJB 3 supporte l'héritage de bean entités :



- L'héritage peut être mis en oeuvre selon **trois stratégies** différentes dans le modèle relationnel, son choix est à la charge du developpeur (détails hors du champs de ce cours, cf. biblio) :
  - Une table unique par hiérarchie de classe.
  - Une table par classe concrête.
  - Une table par sous-classe.

Message-Driven Bean

- Ce type de composant est similaire aux EJB session dans la mesure où ils exécutent eux aussi des actions.
- La différence vient du fait qu'on ne peut invoquer une méthode d'un EJB MD.
- On ne peut d'adresser à un EJB MD qu'indirectement par l'envoi de messages sur un flux, ces messages seronts ensuite traités par le bean de manière asynchrone :
  - Un EJB MD n'est pas accessible par RMI.
  - **Il ne dispose pas d'interface métier.**
- On peut citer comme exemples : le bean qui reçoit les accusés réception d'un échange d'action, le bean qui reçoit les messages d'autorisation de crédit.
- Pour réaliser sa tâche, un EJB MD peut ensuite s'adresser directement à un autre EJB ou indirectement par l'envoi de message à un autre EJB MD.

Message-Driven Bean

Notion de MOM

- Les MOM (Message Oriented Middleware) sont des serveurs dont le rôle est de veiller au bon acheminement des messages asynchrones entre applications.
- Le principe est de permettre à une application d'émettre un message à destination d'une autre application, et de continuer à fonctionner sans attendre la réponse du destinataire.
- Dans tous les cas, le destinataire doit recevoir tôt ou tard le message qui lui est destiné, même s'il n'est pas disponible à l'instant auquel l'émetteur le lui a envoyé.
- Il existe diverses implémentations de MOM, parmis les plus connues : Tibco Rendezvous, IBM Websphere MQ, BEA Tuxedo/Q, Sun Java System Messaging Server, Microsoft MSMQ, Sonic Software SonicMQ, FioranoMQ.

Message-Driven Bean
JMS

- Un serveur d'application compatible Java EE intégre la notion de MOM au travers de l'API JMS (Java Message Service), package *javax.jms* de J2EE.
- L'API JMS désigne un ensemble de classes et d'interfaces Java permettant de créer, d'envoyer, de recevoir et de lire des messages.
- JMS permet des communications :
  - Asynchrones.
  - Avec un faible degré de couplage entre applications.
  - Fiables.
- L'emploi de JMS est utile partout où l'on souhaite qu'une application fonctionne sans que tous ses composants soient nécessairement opérationnels simultanément, ou qu'un composant envoi des informations à un autre composant sans que le premier nécessite une réponse immédiate pour continuer à fonctionner.

Message-Driven Bean

JMS

- Il existe deux modèles de message (“messaging domains”) :
  - **le modèle PTP (Point To Point) :**
    - Il’appuie sur les concepts de file d’attente (“queue”), d’émetteurs (“senders”) et de destinaires (“receivers”).
    - Chaque message est emis vers une fille d’attente spécifique, les destinataires se chargeant d’extraire les messages des files d’attente qui leur on été attribuées.
    - Il est utilisé lorsque chaque message doit avoir un seul destinataire et lorsque le facteur temps n’a pas d’importance dans la relation émetteur-destinataire.
  - **le modèle Pub/Sub (Publish/Subscribe) :**
    - Cermet à des clients d’émettre des messages vers un sujet (“topic”), les messages étants reçus par les clients abonnés à ce sujet.
    - Chaque message peut donc avoir plusieurs destinataires (“consumers”), un destinataire ne recevant que les messages auxquels il est abonné.

Message-Driven Bean

JMS

- Les messages JMS peuvent être consommés de deux manière différentes :
  - **De manière synchrone :**

    Un récepteur reçoit un message en faisant explicitement appel à la méthode `receive`.

    Cette méthode peut attendre sur place l'arrivée d'un message ou se terminer en time-out si le message attendu n'est pas parvenu dans le temps imparti.
  - **De manière asynchrone :**

    Un recepteur peut enregistrer un délégué de message ("message listener") de sorte que lorsqu'un message est émis, le message soit transmis à la méthode on`Message` du délégué.

Message-Driven Bean

Principe

- Les EJB session ou entité ne peuvent pas recevoir de messages JMS.
- Les EJB MD sont des objets transactionnels distribués spécialement conçus pour le traitement des messages asynchrones JMS de PTP ou Pub/Sub.
- Le conteneur EJB prend à sa charge l’environnement de l’EJB, comme l’aspect transactionnel, la sécurité, **la concurrence** et l’acquittement des messages.
- L’apport le plus intéressant des EJB MD est sans-doute la gestion concurrente des messages par le conteneur. Ceci évite au programmeur le besoin de développer des applications transactionnelles multi-threadées : **il n’y a pas besoin d’écrire de code de synchronisation pour ce genre de bean.**
- Les EJB MD étant sans-état (stateless), le conteneur va multiplier les instances d’EJB MD pour pouvoir traiter simultanément plusieurs centaines de messages JMS provenant d’applications différentes.

Message-Driven Bean

Conception

- L'élaboration d'un EJB s'effectue simplement :

  1. Créer une classe dite "classe Bean" qui implémente l'interface javax.jms.MessageListener. La spécification précise que la classe Bean doit comporter un constructeur public sans argument.

  2. Préciser les annotations adéquates sur la classe (ou bien renseigner le descripteur de déploiement).

     - Utilisation sur la classe de l'annotation @javax.ejb.MessageDriven

     - Cette annotation prend en argument une liste de paramètres de configuration afin de permettre l'utilisation de tout type de fournisseur de messages, pas seulement JMS.

Message-Driven Bean

Conception

- EJB 3 défini un jeu de 5 propriétés adaptées à un fournisseur de message de type JMS :
  - `destinationType` : prend les valeurs `javax.jms.Queue` ou `javax.jms.Topic` .
  - `destination` : indique le nom de la destination.
  - `acknowledge` : prend les valeurs `Auto-acknowledge` ou `Dups-ok-acknowledge`.
  - `messageSelector` : indique une expression de filtrage.
  - `subscriptionDurability` : prend les valeurs `Durable` ou `NonDurable`.

Message-Driven Bean
Exemple


```
@MessageDriven(activationConfig={
   @ActivationConfigProperty(
      propertyName   = “destinationType”,
      propertyValue  = “javax.jms.queue”),
   @ActivationConfigProperty(
      propertyName   = “destination”,
      propertyValue  = “queue/reservationQueue”),
   })
public class ReservationProcessorBean implements javax.jms.MessageListener {
   ...

   public void onMessage(Message message) {
      ...
   }
   ...

}
```

Message-Driven Bean

Exemple


```
@MessageDriven(activationConfig={
   @ActivationConfigProperty(
      propertyName   = “destinationType”,
      propertyValue  = “javax.jms.queue”),
   @ActivationConfigProperty(
      propertyName   = “destination”,
      propertyValue  = “queue/reservationQueue”),
   })
public class ReservationProcessorBean implements javax.jms.MessageListener {
   ...

   public void onMessage(Message message) {
      ...
   }
   ...

}
```

File type JMS

Adresse file d’attente

Notification message

Environnement de TP

- Utilisation de l'IDE **NetBeans** :

  L'intérêt d'utiliser cet environnement de développement en TP provient de son intégration très fine avec le serveur d'application Sun qui permet d'avoir à tout configurer à la main et de se concentrer sur le développement.

- Utilisation du serveur d'application "officiel" de Sun issu du projet GlassFish : **Sun Java System Application Server 9** (SJSAS 9) :
  - GlassFish est un projet open source né de l'ouverture au monde open source du serveur d'application Sun Application Server 8.
  - GlassFish intègre la couche de persistance TopLink d'Oracle.
  - Le serveur est une implémentation **complète** de la spécification Java EE 5.

Bibliographie

- Une référence fondamentale et gratuite !
  Mastering EJB 3.0
  http://www.theserverside.com/news/thread.tss?thread_id=41363

Fin du cours #7