AISL / GLG203 – GLG204

Architectures Logicielles Java

Les Enterprise Java Bean (EJB)

EJB : Présentation (1)

- C**omposants Java portables, réutilisables et déployables** qui peuvent être **assemblés** pour créer des **applications**.
- Constituent la **partie centrale** de la plateforme **JEE** en implantant la **logique métier**.
- Réalisent les fonctions attendues en collaborant avec d'autres EJB.
- S'exécutent dans un **conteneur d'EJB** qui va les **gérer** et leur fournir des services tels que **les transactions, la sécurité ou la persistance**. *(Interception, Décoration)*

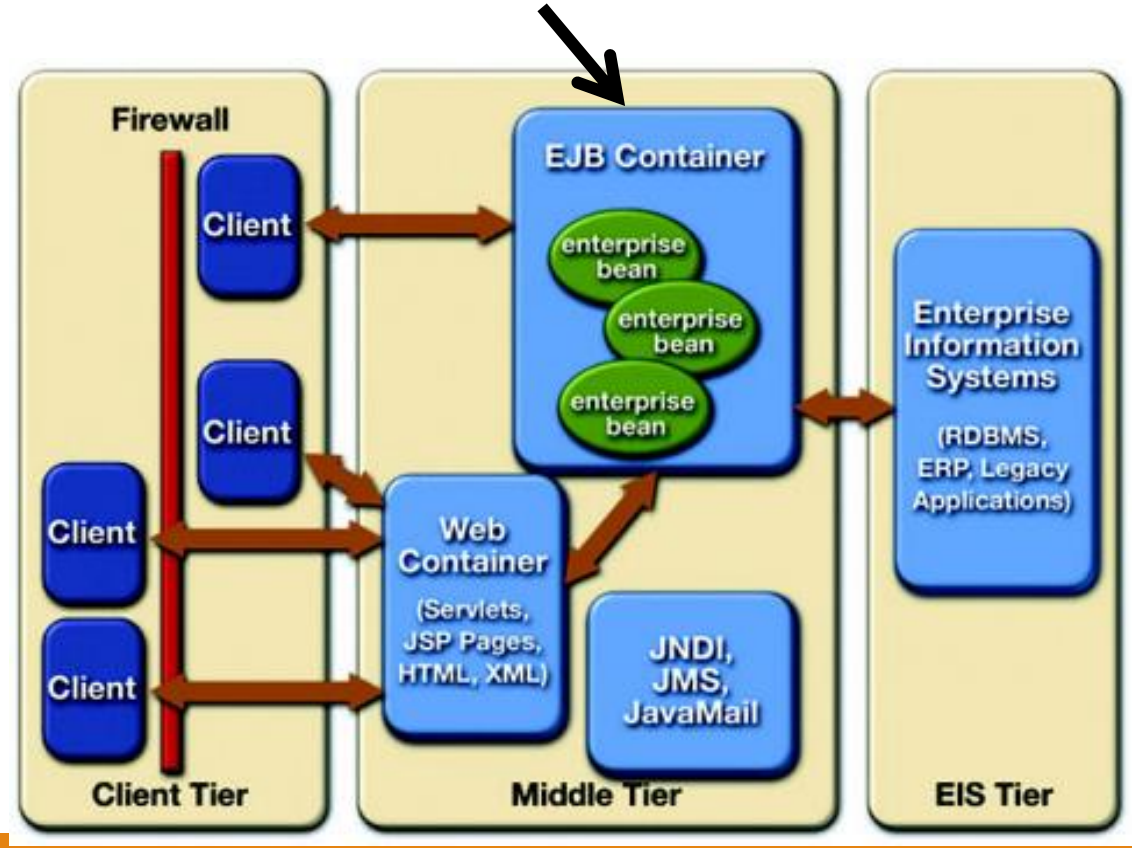
EJB : Présentation (2)

- Un EJB est **fourni par le serveur d'application** :
  - **Jamais de new** : doit pas être instancié dans le code
  - **Demande** de l'EJB **au serveur** en exploitant **JNDI** (Java Naming Directory Interface)

    (ressources nommées dans le code sans connaitre leurs localisations exactes)
  - **Obtention** d'un EJB par **injection de dépendances**
- **Cycle de vie géré** par le **container** (avec Callbacks à base d'annotations)

Les EJB dans l'architecture JEE

- Façades pour accéder aux fonctions métiers et objets métiers



Rôles du conteneur d'EJB (1)

- **Isoler les EJB** accessibles par les clients d'une **implémentation spécifique pour un serveur**
- Gérer le **cycle de vie** des EJB (Activation, Passivation etc.)
- Gestion de Pool d'EJB
- Gestion des invocations dans des **Transactions applicatives**
- Contrôle des **droits d'accès**
- Gérer les **accès concurrents** (verrous)

Rôles du conteneur d'EJB (2)

- Prendre en charge les **appels asynchrones**
- **Répartition de la charge** (distribution sur différentes machines)
- Réaliser l'**Injection de dépendance**
- Permettre les **appels distants**
- Réception et routage des messages **JMS**
- Gère la planification d'appels (**Timer service**)

https://www.careerride.com/Java-services-EJB-container-offers.aspx

EJB : Intérêts

- **Les développeurs se focalisent uniquement sur les fonctions métiers**
- Pas besoin de gérer le non fonctionnel (Transaction, Sécurité, Thread, optimisations par pooling, Mail, etc.)
- **Les EJB sont portables:** peuvent être utilisés (normalement) sur tous les serveurs JEE
- Accès à toutes les **ressources déclarées gérées par le serveurs** (connexion aux Bdd avec JDBC, JMS queues et connexions)

https://stackoverflow.com/questions/12872683/what-is-an-ejb-and-what-does-it-do

EJB : Les limitations

- Pas de champs statiques
- Ne pas créer de Threads
- Ne pas créer de Sockets
- Ne pas manipuler de fichiers à travers le système de fichiers
- Ne pas exécuter de code natif

https://www.oracle.com/technetwork/java/restrictions-142267.html

Utilisation d'un EJB (1)

- **Côté client :**

Client lourd (accès en Remote):

- Localisation du Bean en utilisant annuaire JNDI

- Obtention d'un **Proxy** (cf RMI IIOP)

- Invocation des méthodes de sa « **Remote Interface** »

A partir du Web Container (JSF) ou d'un EJB:

- Le **« Managed Bean »** ou l'**EJB** client Obtient l'EJB cible par **JNDI** ou bien par **Injection**

- Invocation des méthodes de l'interface de l'EJB

Utilisation d'un EJB (2)

- **Côté conteneur :**
  - **Création** ou **activation** de l'EJB (selon étape du cycle de vie).
  - Exécution du code invoqué (Skeletone si appel distant).
  - Assure la **réalisation des services non fonctionnels** :

    Persistance, Transaction, Sécurité, etc.

(nous y reviendrons un peu plus loin)

Les Différents types d'EJB

- **Les Session Bean :** Implémentent des traitements
- **Les Message Driven Bean :** Consomment et traitent les messages asynchrones reçus (JMS)
- **Les Entity Bean :** implémentent les objets métiers
- **Les EJB Timer :** utilisent le Timer Service du serveur JEE pour lancer des traitements planifiés

Les Session Bean (1)

- **Implémentation des traitements** (Eventuellement avec son Interface)
- **Comportent un ensemble de méthodes** chargées de :
  - Mettre à disposition les **« primitives » métiers**
  - Réaliser les **cas d'utilisation** nécessaires à l'Application
- **Les méthodes comportent :**
  - Des calculs, l'application des règles de gestions
  - Des accès aux données du domaine
  - Des utilisations de méthodes fournies par d'autres EJB
  - Des utilisations de services externes

Les Session Bean (2) : Les 3 types

- **Les Bean sans état (Stateless):**

- Pas d'état en dehors des appels
- Possibilité d'appels en parallèles
- Partageable entre plusieurs clients

- **Les Bean avec états (Stateful):**

- Mémorise des états entre les différents appels (requêtes)
- Maintient un état conversationnel

- **Les Bean singleton :**

- Une instance unique pour l'application

Les Session Bean Stateless (1)

- **Les Bean sans état (Stateless):**

- Pas d'état en dehors des appels *(quels sont ces états ?)*
- Possibilité d'appels en parallèles *(pourquoi ?)*
- Partageable entre plusieurs clients, il peut être un WebService

Exemples :

Un Bean avec une méthode pour effectuer un calcul utilisant ses paramètres

Un Bean avec une méthode pour obtenir la liste des Produits d'un catalogue (accès Bdd)

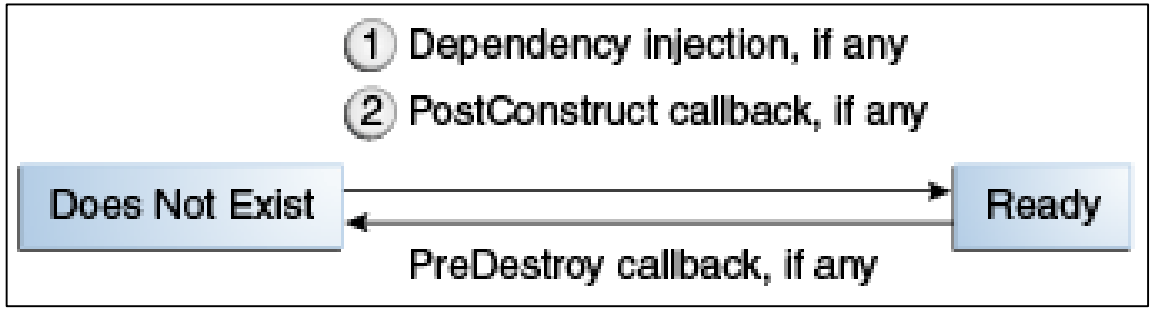
Les Session Bean Stateless (2)

```
@Stateless(mappedName = "ExempleService")
public class ExempleServiceImpl {

 @EJB
 TarifService tarifService;

 // Retourne la liste des produits du catalogue
 public List<Produit> getProduitsDuCatalogue() {
        //...
  }

 // Calcule et retourne le prix pour une quantité de produits
 // En tenant compte des promos du moment
 public Integer calculerPrix(Produit produit, Integer quantite) {
        //...
  }

}
```

Les Session Bean Stateless (3)

- **Le cycle de vie d'un Bean Stateless :**

- Pas de passivation → seulement 2 états
- Le container crée un Pool de Session Bean Stateless avant toute création
- Le container crée une instance du Bean
- Le container réalise l'injection de dépendance
- Si une méthode est annotée @PostConstruct elle est appelée (pour initialisation après injection)
- L'EJB peut être utilisé par un client



- A la fin du cycle de vie si une méthode est annotée @PreDestroy elle est appelée
- Le bean est prêt à être collecté par le ramasse-miette

Les Session Bean Stateful (1)

- **Les Bean avec états (Stateful):**

- Mémorise des états entre les différents appels (requêtes)
- Maintient un état conversationnel
- Associé à **un client unique**

Exemples :

Un Bean avec des méthodes pour gérer un panier sur un site de e-commerce

Un Bean avec des services nécessitant l'utilisateur connectée pour être réalisés correctement

Les Session Bean Stateful (2)

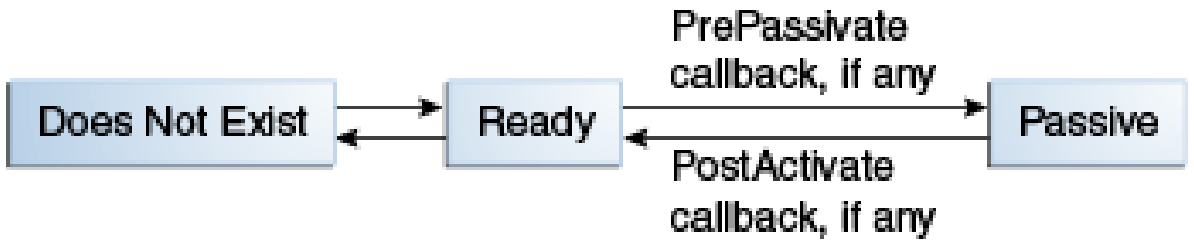
```
@Stateful(mappedName = "Panier")
public class Panier  {

 private Utilisateur utilisateur;

 private List<Produit> produits;


 public void setUtilisateur(Utilisateur utilisateur) {
        //...
 }

 // Ajout d'un article au panier
 public void addProduit (Produit produit) {
        //...
 }

 // Retrait d'un article au panier
 public void removeProduit (Produit produit) {
        //...
 }

}
```

Les Session Bean Stateful (3)

- **Le cycle de vie d'un Bean Stateful :**

- Le client du bean demande une référence, le conteneur crée une instance.
- Le container réalise l'injection de dépendance puis appelle la méthode annotée @PostConstruct
- Le client peut invoquer les méthodes du bean
- Le container peut « Passiver » le Bean (il doit être sérialisable)

1. Create
2. Dependency injection, if any
3. PostConstruct callback, if any
4. Init method, or ejbCreate<METHOD>, if any



Les Session Bean Stateful : Passivation / Activation (4)

La Passivation :

- Pour récupérer des ressources, le container peut **retirer le bean de la mémoire et l'enregistrer en mémoire secondaire** (disque)

- Opération nommée : **« Désactivation »** ou **« Passivation »**

- Si une méthode est annotée **@PrePassivate** elle est invoquée (avant passivation)

Activation :

- Si un bean a été **« Passivé »** lorsque son client invoque une de ses méthodes, son image doit être rechargée en mémoire.

- Opération nommée : **« Activation »**

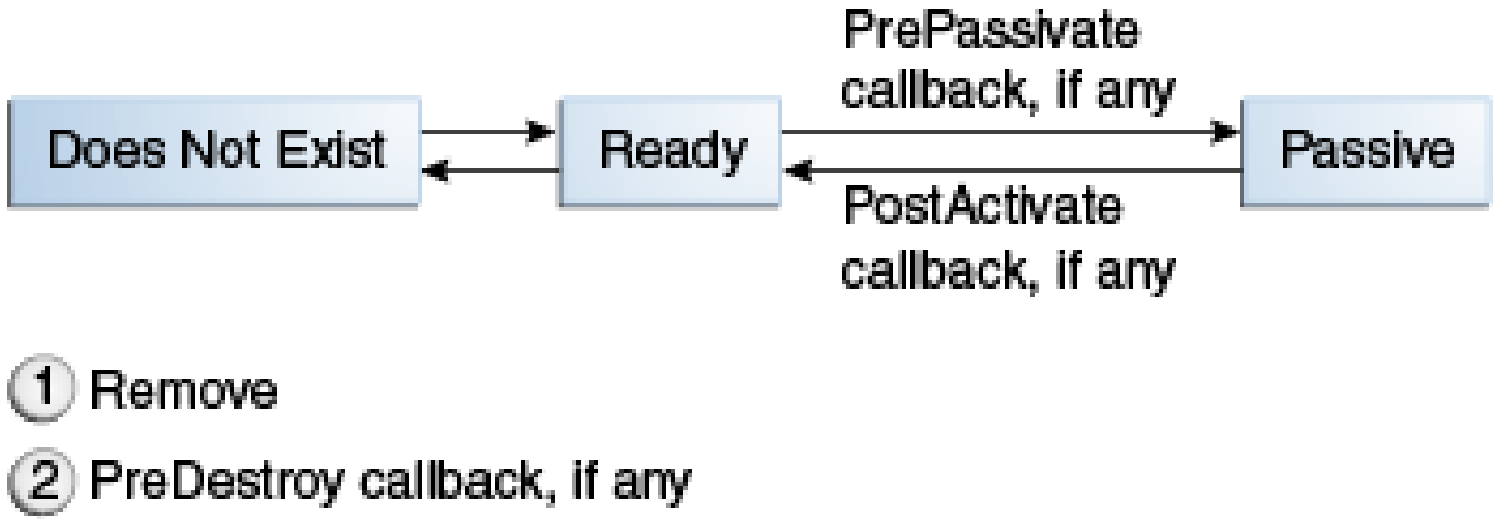
- Si une méthode est annotée **@PostActivate** elle est invoquée (après activation)

Les Session Bean Stateful (5)

- **Fin du cycle :**

- Pour « libérer » le bean, le client invoque une méthode annotée @**Remove.**

- Le container va **détruire le bean**, si une méthode annotée @**PreDestroy** existe elle est invoquée avant cette destruction.



Les Session Bean Stateful (6)

- **Exemple du panier (1 par client) :**

```
@Remote // Interface du service
public interface Panier {
     /**
      * Ajoute une quantité d'article au panier
      * @param article
      * @param quantite
      */
     public void ajouterArticle(ArticleDto article, int quantite);
     /**
      * Supprime tous les articles du type passé en paramètre
      * @param article
      */
     public void retirerArticle(ArticleDto article);
     /**
      * Validation du panier
      */
     public void validerPanier();
}
```

Les Session Bean Stateful (7)

```
@Stateful
public class PanierImpl implements Panier {
	/**
	 * Contenu du panier
	 */
	private Map<ArticleDto, Integer> contenu;
	@PostConstruct
	private void init() {
		contenu = new HashMap<>();
	}
	public void ajouterArticle(ArticleDto article, int quantite) {
		contenu.put(article, quantite);
	}
	public void retirerArticle(ArticleDto article) {
		contenu.remove(article);
	}
	@Remove
	public void validerPanier() {
		// TODO Validation du panier
	}
}
```

Les Session Bean Singleton (1)

- **Les Bean singleton:**

- Une instance unique pour l'application (vit durant toute l'application)
- Partagé par plusieurs clients *(impact ?)*

Exemples :

Un Bean pour obtenir des infos de configurations de niveau application

Un Bean pour « Poster » et « Relever » des messages (Communication entre bean)

Un compteur de click sur une page web

Les Session Bean Singleton (2)

```
/**
 * CounterBean un singleton session bean pour compter
 * le nombre de click d'une page web (tuto JEE 8 Oracle)
 */
@Singleton     // Déclaration en tant que singleton
@Startup       // Directive de chargement au déploiement
@DependsOn("AutreSingletonBean") // Enchainement des initialisations: après AutreSingletonBean
public class CounterBean {
    private int hits;

    @PostConstruct
    protected void init() {
       hits = 1;
    }

    // Increment and return the number of hits
    public int getHits() {
       return hits++;
    }
}
```

Les Session Bean Singleton (2)

- Impact du partage par plusieurs clients :

- Les méthodes seront sûrement appelées simultanément par plusieurs threads
- Nécessité de gérer la concurrence pour préserver les ressources accédés
- Gestion de la concurrence possible en utilisant les annotations **@Lock:**
- Application sur les méthodes pour lesquelles il faut gérer la concurrence :

`@Lock(LockType.READ)` : **Exécution en parallèle autorisée**

`@Lock(LockType.WRITE)` : **Exécution en parallèle non autorisée (Verrouillage)**
Quand une méthode d'écriture est en cours d'exécution, elle interdit l'appel de toute autre méthode sur le Bean (lecture ou écriture)

Les Session Bean Singleton (3)

- **Gestion des accès concurrents :**

- Besoin de synchroniser l'accès à une ressource

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER) // Contrôle concurrence assurée par le container (par défaut)
public class CounterBean {
  private String state;

  @Lock(LockType.READ)
  public String getState() {
    return state;
  }

  @Lock(LockType.WRITE)
  public void setState(String newState) {
    state = newState;
  }
}
```

`@ConcurrencyManagement(BEAN)` : Le développeur doit gérer la concurrence d'accès avec les possibilités Java

Les Session Bean Singleton (4)

- **Cycle de vie :**

- Identique au Stateless : **Pas de Passivation**

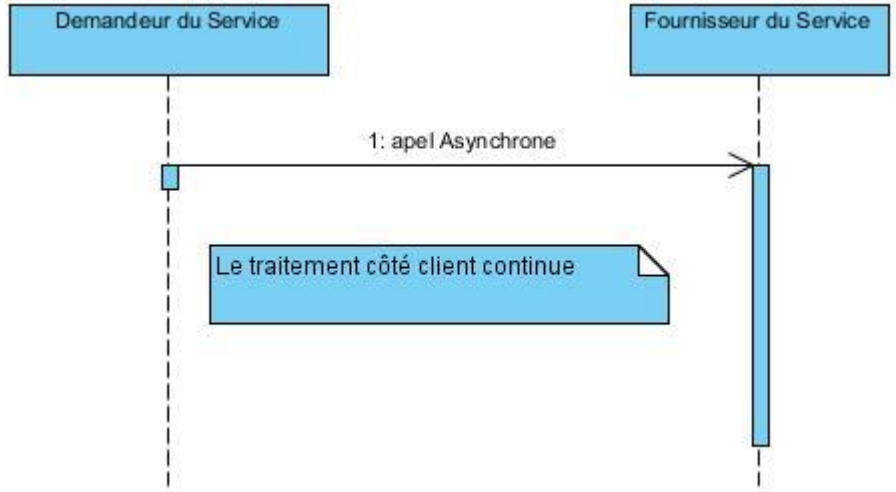
- En fin de cycle, Le container va détruire le bean, si une méthode annotée **@PreDestroy** existe elle est invoquée avant cette destruction.

Traitement asynchrone (1)

- **Appel asynchrone d'une méthode d'un session bean :**

- Lancement de la méthode en tâche de fond et rend la main à l'appelant.
- Utile pour les traitements longs (pour éviter les blocages)
- Le container crée un Thread dédié à cet appel asynchrone

« Comment récupérer le résultat de l'appel ? »



Traitement asynchrone (2)

- **Appel asynchrone d'une ou de toutes les méthodes d'un session bean :**

- Marquage de la méthode à exécuter avec `@Asynchronous`

- Si la classe est annotée avec `@Asynchronous` toutes les méthodes sontà appeler en asynchrones.

```
@Stateless
public class ExempleServiceImpl {
  //...
 @Asynchronous // pas de retour
  public void initialiserPrix() {
    //...
 }
 // Calcule et retourne le prix pour une quantité de produits (Utilisation d'une Future)
 @Asynchronous
 public Future<Integer> calculerPrix(Produit produit, Integer quantite) {
    //...
 }

}
```

Traitement asynchrone (3)

- **Appel asynchrone :**

- Le contexte de sécurité est transmis: Possibilité de vérifier les droits d'exécution des méthodes appelées.

- Le contexte transactionnel n'est pas transmis: les directives pour la gestion de transaction seront invalides

REQUIRES → création

MANDATORY → TransactionRequiredException

- Possiblité de stopper la tâche si elle retourne une future:

Invocation de la méthode **cancel()** sur la future (si tâche non encore lancée)

Invocation de la méthode **cancel(true)** et lecture flag de stop à partir du **SessionContext** (exemple sur slide suivant)

Traitement asynchrone (4)

```
@Stateless
public class ExempleServiceImpl {
 @Ressource SessionContext sc;

  @Asynchronous
  public Future<Integer> calculerPrix(Produit produit, Integer quantite){
      if (sc.wasCancelCalled()) {
            return new AsyncResult<Integer>(-1);
      }

      //...
      return new AsyncResult<Integer>(prixCalcule);
  }

}
```

Les Différents type d'accès aux Bean (1)

- **JEE et les Bean a pour but de permettre la création d'applications réparties :**

- Plusieurs tiers physiques (plusieurs machine)
- Plusieurs tiers logiques (serveur JEE, plusieurs JVM)

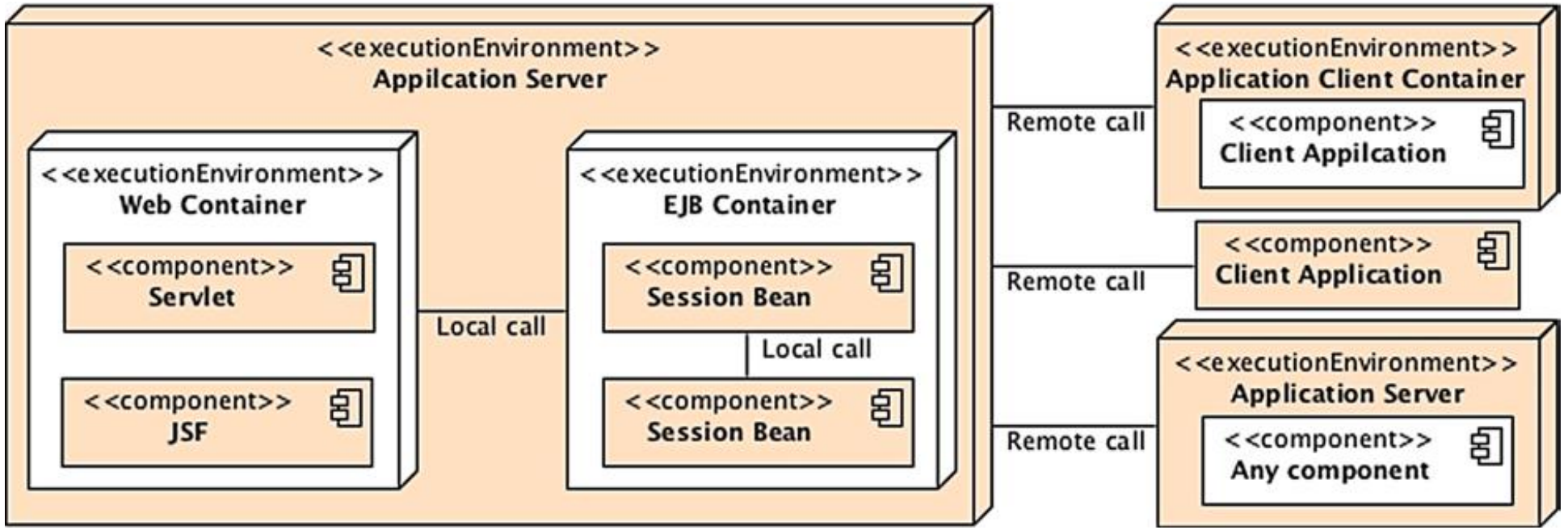
Répartition des composants ➔ Appels distants

- Plusieurs Types de clients pour les services :

- **Les client locaux :** s'exécutent dans la même JVM (et le même ordinateur évidement)
- **Les clients distants :** s'exécutent dans une JVM différente (même ordinateur ou différent)

Les Différents type d'accès aux Bean (2)

- **Différents appels de service dans une application JEE :**



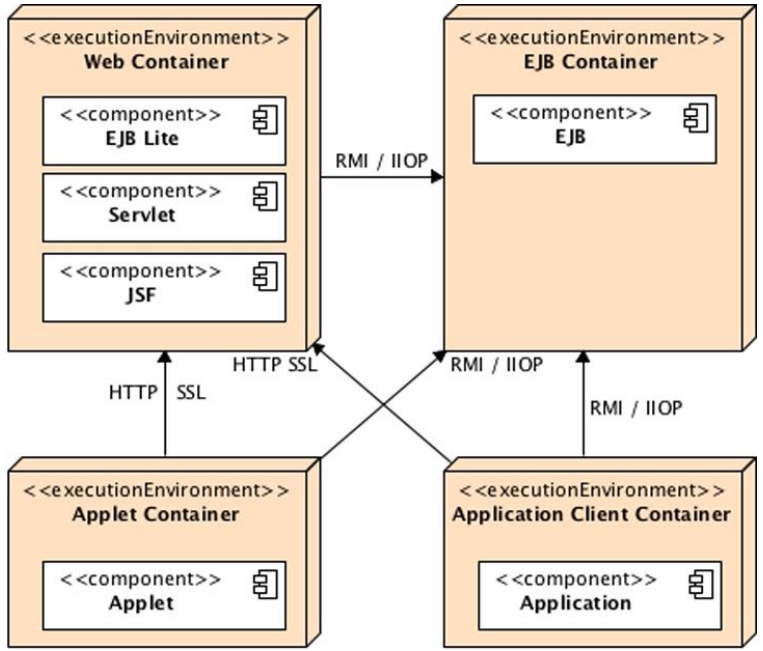
(Extrait du livre « Beginning JEE 7 » de Antonio Goncalves)

Bean accessible à distance (1)

- **Réalisation des interactions distantes en utilisant RMI-IIOP :**

RMI : Remote Method Invocation de JSE (protocole JRMP)

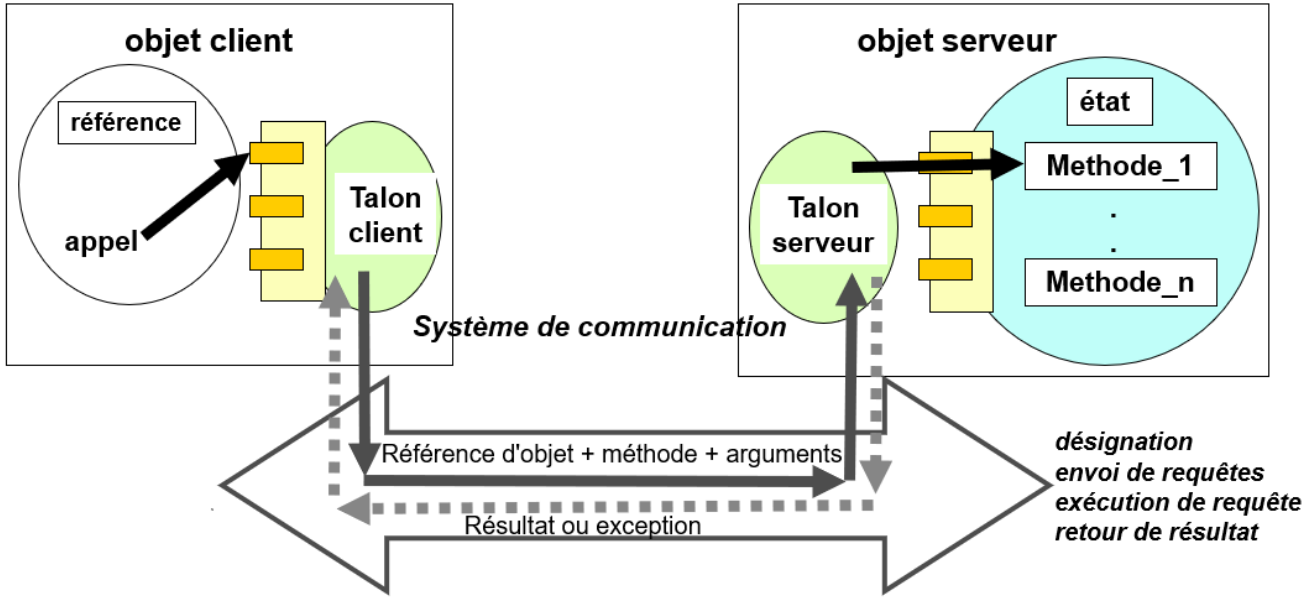
RMI-IIOP : extension de RMI pour obtenir interopérabilité CORBA (IDL)

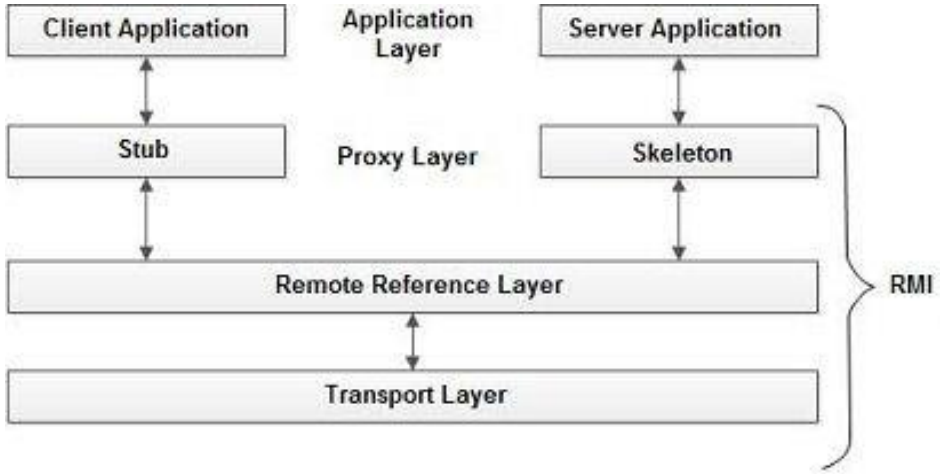


Bean accessible à distance (2)

- **RMI :**

- **Souches: Proxy** côté client et **Skeleton** côté serveur
- Utilisation d'un **réseau** de communication → **Sérialisation**
- **Le Client exploite référence distante du service** (dont il ne connait que l'interface)





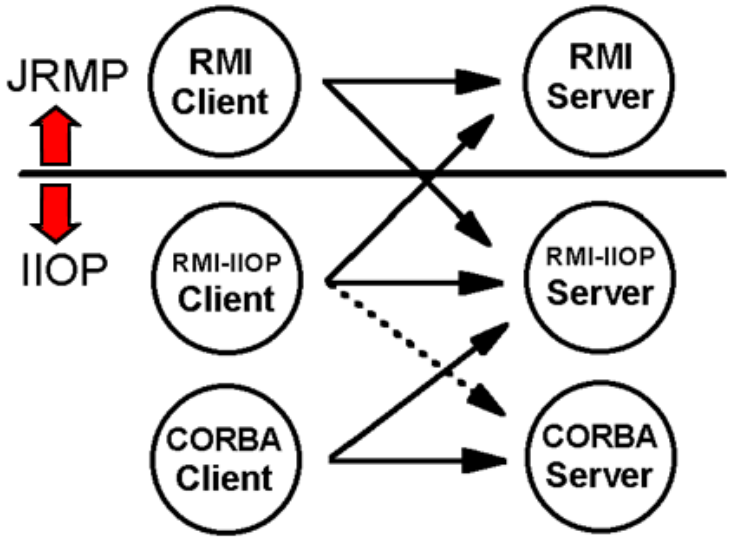
Bean accessible à distance (3)

- **IIOP :**

  - GIOP: (General Inter ORB Protocol) est un standard de communications entre ORB : **Format** pour les références, **CDR** (représentation des données), **Messages** (request, reply)

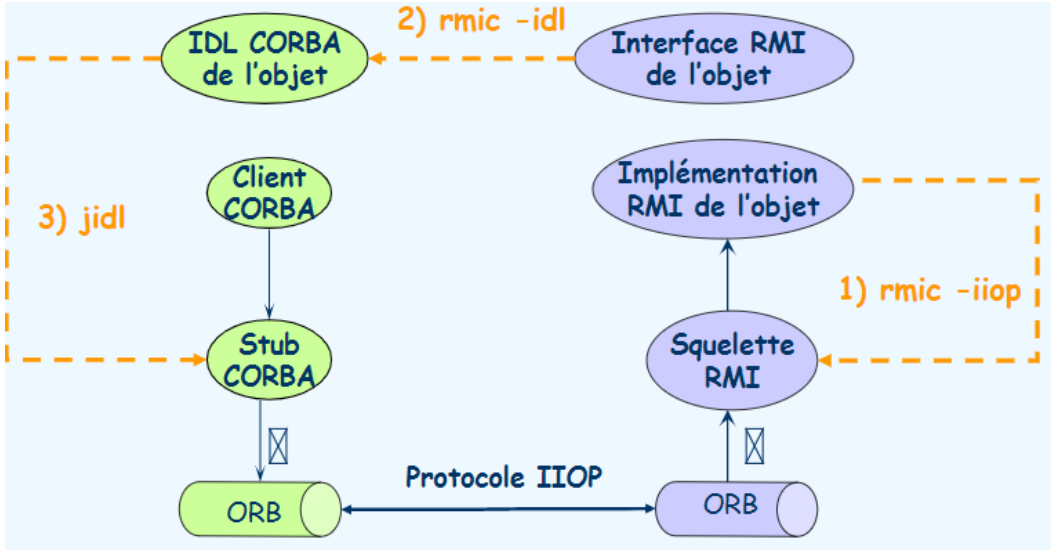
  - **IIOP :** implémentation de GIOP

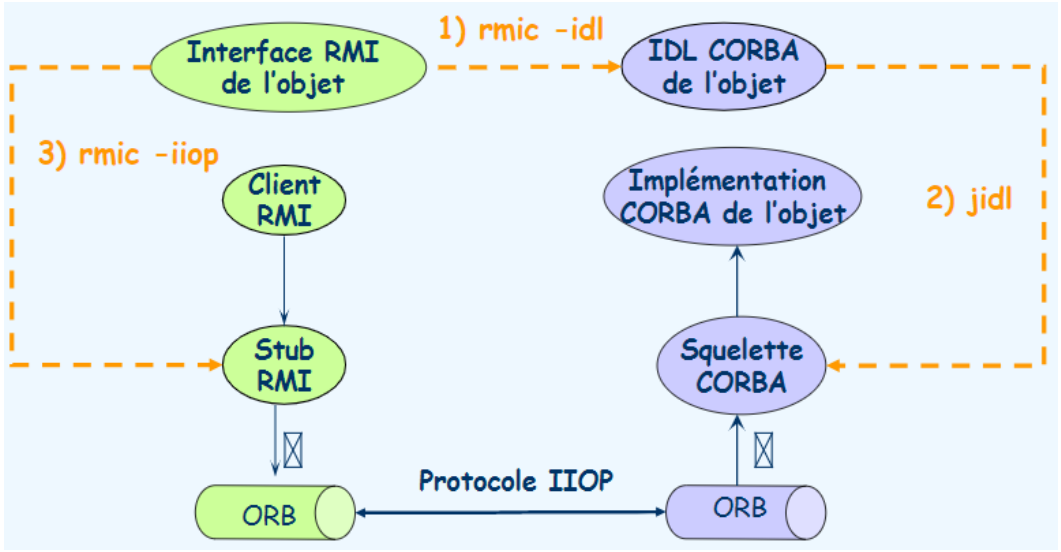
- **IIOP et JRMP sont incompatibles, on voudrait pourtant obtenir l'interopérabilité :**



Bean accessible à distance (4)

- **IIOP :**
- **Génération de souche pour assurer compatibilité IDL ←→ RMI**
- **Pont pour IIOP vers RMI et vice-versa**





Bean accessible à distance (5)

- **Concepts clés pour l'implémentation d'un Session Bean distant :**

- **Arguments + Valeurs de retour doivent être Sérialisable**
- **Interface de service indispensable et annotée @Remote**
- **Implémentation de l'interface par le Session bean @Stateless ou @Stateful**
- **Les bean s'exécutant sur la même JVM peuvent l'utiliser (mais en mode distant)**

```java
/**
 * Interface du service distant
 */
@Remote
public interface DemoRemote {
   public void faireLeTruc();
}
```

```java
/**
 * Implémentation du service distant
 */
@Stateless
public class DemoRemoteImpl implements DemoRemote {
    @Override
    public void faireLeTruc() {
        System.out.println("Je suis " + this.getClass()
        + " et je fais le truc distant.");
    }
}
```

Bean accessible en Local uniquement

(1)

- **Accès à un Session Bean s'exécutant sur le même JVM :**
  - Interface de service annotée @Local
  - Implémentation de l'interface par le Session bean @Stateless ou @Stateful

```
/**
 * Interface du service local
 */
@Local
public interface DemoLocal {
    public void faireLeMachinEnLocal();
}
```

```
/**
 * Implémentation du service Local
 */
@Stateless
public class DemoLocalImpl implements DemoLocal {

    @Override
    public void faireLeMachinEnLocal() {
        System.out.println("Je suis " + this.getClass()
        + " et je fais le machin local.");
    }
}
```

Bean accessible en Local uniquement

(2)

- **Depuis les EJB 3.1 @LocalBean pour les « No Interface View »:**

- **Pas besoin d'interface**
- **Implémentation du Session bean @Stateless ou @Stateful annoté @LocalBean**

```
@LocalBean
@Stateless
public class DemoNoInterfaceView {
    public void faireLeService() {
        System.out.println("Je suis " + this.getClass()
        + " et je fais le service.");
    }
}
```

Par convention :

- Un EJB sans annotation @local ou @remote est considéré comme local.

- Quand on a une interface locale ou remote, seules les méthodes de cette interface sont des méthodes EJB (transactionnelles, contrôlées ...). **Les autres méthodes sont « normales »** (non adaptées)

- Quand on n'a pas d'interface déclarée, l'EJB est local et son interface locale correspond à l'ensemble des méthodes publiques.

Les Session Bean : un résumé

- Stateless != Stateful
- Le Singleton partagé: problèmes d'accès concurrents
- Cycle de vie géré par le Container
- Session bean Stateless pour implémenter les « Services »

Utilisation des Session Bean (1)

- **Les utilisateurs des Session Bean** (clients de ses fonctionnalités) **sont :**
  - Les **Applications Clientes** (APPClient JEE) utilisent des Bean distants
  - D'autres **Session Bean** utilisent d'autres Bean locaux ou distants
  - Les **Managed Bean** (JSF) utilisent des Session Bean locaux ou distants

- **Pour utiliser un Session Bean :**
  - Il faut **Obtenir une référence** (Locale ou distante) **sur une instance** du Bean
  - Les Session Bean **ne doivent pas être instanciés** dans le code utilisateur
  - Le **container est chargé d'instancier** et de gérer le cycle de vie des Bean

Utilisation des Session Bean (2)

- **En JEE deux manières principales d'obtenir la référence:**

  - **Utilisation de l'annuaire JNDI** (en tant que Service Locator)

    Le client demande à l'annuaire de lui fournir l'EJB

  Utilisation « Correspondance entre nom symbolique et composant »

  - **L'Injection de dépendances effectuée par le container d'EJB**

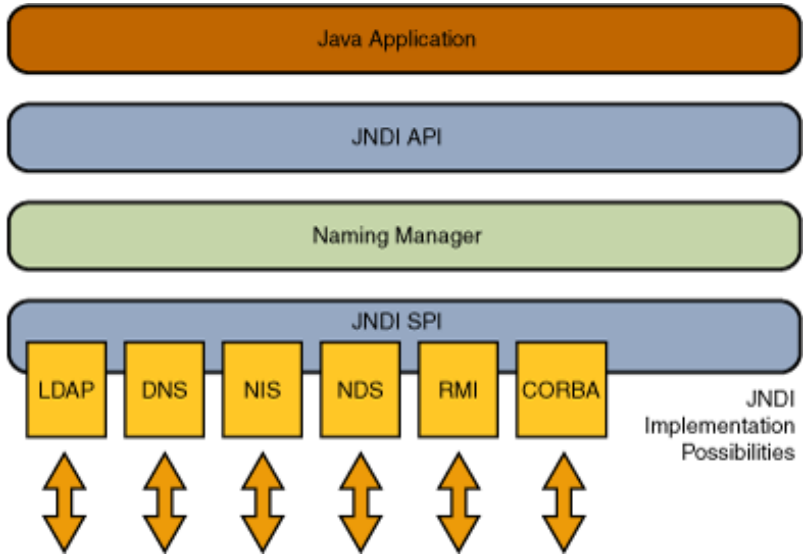
    Alimentation des références à des EJB par le container

JNDI (1)

- **JNDI : Java Naming and Directory Interface**

Fournit un accès unifié à des services d'annuaires (LDAP, DNS, UDDI …)

- JNDI est inclus dans Java SE
- S'utilise avec les API JNDI et les "Fournisseur de services JNDI"
- Le JDK est livré avec les fournisseurs de services :
  - Lightweight Directory Access Protocol (LDAP)
  - Common Object Request Broker Architecture (CORBA)
  - Common Object Services (COS) name service
  - Java Remote Method Invocation (RMI) Registry
  - Domain Name Service (DNS)



JNDI : les primitives (1)

Opérations JNDI :

- Les operations JNDI s'effectuent sur un **`Context`** :

Le Context est l'interface JNDI representant un annuaire

```
// Configuration des propriétés pour la création du Context
Properties props = new Properties();
props.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.enterprise.naming.SerialInitContextFactory");
// Obtention du Context
InitialContext ctx = new InitialContext(props);
```

- **`bind() ou rebind()`** : primitives pour ajouter un objet dans un annuaire. L' ajout se fait en associant un objet à un nom symbolique.

```
// Enregistrement d'un objet de type fruit (une orange) associé au nom « favorite »
ctx.bind("favorite", new Fruit("orange"));
```

JNDI : les primitives (2)

- **`unbind()`** : primitives pour retirer un objet d'un annuaire. Le retrait se fait en donnant le nom symbolique associé à l'objet à retirer.

```
// Retrait du fruit favori (l'orange) de l'annuaire
ctx.unbind("favorite");
```

- **`rename()`** : primitives pour changer le nom symbolique d'un objet d'un annuaire.

```
// Renommage de nom du fruit favori (l'orange)
ctx.rename("favorite", "favoriteFruit");
```

- **`lookup()`** : primitives pour obtenir un objet dans un annuaire. La requête se fait en donnant le nom symbolique de l'objet demandé.

```
// Demande du fruit favoris à l'annuaire
Fruit favoriteFruit = (Fruit) ctx.lookup("favoriteFruit");
```

Cas d'utilisation: obtention Factory JMS, Connexions JDBC, **et les Bean**

Obtention Bean par JNDI (1)

- L' API JNDI est normalisée depuis son apparition
- Par contre les noms symboliques d'enregistrement des Bean ne l'était pas :

Conséquence ?

- Depuis EJB 3.1 :
  - Le nom des Bean a été spécifié → « Nom Portable »
  - Lorsque un Session Bean et son Interface sont déployés dans le container, il est automatiquement ajouté à JNDI associé a son nom portable.

Obtention de la portabilité des applications JEE

Obtention Bean par JNDI (2)

- Le nom portable JNDI d'un Bean est de la forme suivante :

java:<scope>[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]

- **<scope>:** Espace de nom standardisé correspondant à des portées différentes du point de vue d'une application JEE :

  java:global: permet d'accéder à des EJB éloignés (Remote) déployés dans un contexte global

  java:module: permet d'accéder à des EJB locaux déployés dans le même module

  java:app: permet à une application d'accéder à des EJB locaux déployés par elle-même (EAR multi-module)

- **<app-name>: nom de l'application ayant déployé le bean**
- **<module-name>: nom du module auquel appartient le bean**
- **<bean-name>: nom de la classe du bean**
- **<fully-qualified-interface-name>: nom de l'interface préfixé avec le package de l'interface**

Obtention Bean par JNDI: java:global (3)

- **Le namespace JNDI java:global :**

java:global[/application name]/module name/enterprise bean name[/interface name]

Pour accéder à des EJB éloignés (Remote) déployés dans un contexte global

- **application name :** Le nom de l'application est nécessaire seulement si l'application est packagé en EAR
- **module name :** Le nom du module dans lequel est implémenté l'EJB
- **interface name** : nécessaire seulement si l'EJB implémente plusieurs interfaces

Obtention Bean par JNDI: java:module

(4)

- **Le namespace JNDI java:module :**

java:module/enterprise bean name/[interface name]

Pour accéder à des EJB locaux déployés dans le même module.

- **module name :** Le nom du module dans lequel est implémenté l'EJB
- **interface name** : nécessaire seulement si l'EJB implémente plusieurs interfaces

Obtention Bean par JNDI: java:app (4)

- **Le namespace JNDI java:app:**

java:app[/module name]/enterprise bean name[/interface name]

Pour accéder à des EJB locaux déployés dans la même application (EAR).

- **module name :** Le nom du module dans lequel est implémenté l'EJB
- **interface name** : nécessaire seulement si l'EJB implémente plusieurs interfaces

URI équivalentes:

java:module/MonBean est equivalent à ***java:global/monApplication/MonBean***

Obtention Bean par JNDI (5)

- **Les noms portables JNDI des Bean apparaissent dans la trace du serveur au déploiement :**

```
Portable JNDI names for EJB TracabiliteServiceImpl:
[java:global/com.sysord.sgp.web/TracabiliteServiceImpl!com.sysord.sgp.core.tracabilite.service.TracabiliteService,
 java:global/com.sysord.sgp.web/TracabiliteServiceImpl]]]
```

Le Session Bean « TracabiliteServiceImpl » :

- Est un Bean Local
- Est Stateless
- Réalise l'interface TracabiliteService

L’Obtention des Bean par JNDI (6)

En résumé :

- Pour un bean dans une autre application, utiliser le nom « global »

- Pour une application construite à partir de plusieurs war, jars (EAR): pour accéder à des beans dans d’autres modules d’une même application utiliser le « nom d’application »

- Pour des beans dans le même module, utiliser le « nom de module »

Obtention Bean par Injection de dépendances (1)

- **Injection de dépendances :** mécanisme qui permet d'implémenter le principe d'IOC (inversion de contrôle). Instanciation et alimentation des dépendances d'après une configuration.

- Injection réalisée par le **container** d'EJB.

- **Types de dépendances injectées :**
  - **Autres EJB** nécessaires à la réalisation de fonctionnalités (délégation)
  - **Référence de Web Service** dont l'EJB veut être client.
  - **Ressources :**
    - Ressources particulières: Persistence Context …
    - Autres Ressources : Connexion Bdd, Session Context, Queue JMS etc.

Obtention Bean par Injection de dépendance (2)

- **Définition et configuration des injections :** solution moderne basée sur l'annotations des références à alimenter
- **Injections réalisée par le Container :** instanciation ou réutilisation instance d'un pool
- **Un composant fourni ou injecté par le container** l'est avec toutes ses dépendances
- **2 Solutions pour utiliser l'injection de dépendances :**

- En **EJB 3.0** : **@EJB et @Ressource**
- Depuis **EJB 3.1** : **@Inject** et le **CDI** (Context Dependency Injection)

Injection de dépendances EJB 3.0 (1)

- **L'annotation @EJB (javax.ejb.EJB) :**

Solution la plus simple pour obtenir la référence d'un EJB.

- Les EJB et leurs interfaces sont définis de manière standard

  @Local, @Remote, @Stateless, @Stateful

- Les dépendances à injecter sont **annotées** avec **@EJB**

- Les dépendances (propriétés) sont à **typer avec l'interface** de l'EJB à injecter.

- L'implémentation à injecter est sélectionnée en fonction de l'interface. (Si une seule implémentation facile, et si plusieurs ?)

Exemple Injection de dépendances EJB 3.0 (1)

- **Le service de calcul de tarif**

```
@Remote
public interface CalculateurTarif {
/**
 * Calcule le tarif d'un article pour un quantité donnée
 * @param quantite
 * @param articleDto
 */
public Double calculerTarif(Double quantite, ArticleDto articleDto);
}
```

```
// Implémentation du CalculateurTarif
@Stateless
public class CalculateurTarifImpl implements CalculateurTarif {
     @Override
     public Double calculerTarif(Double quantite, ArticleDto articleDto) {
          return articleDto.getPrix() * quantite;
     }
}
```

Exemple Injection de dépendances EJB 3.0 (2)

- **Le service de commande**

```
@Local
public interface CommandeService {
     /** Crée une commande à partir d'un ensemble d'article pour un client */
     public void creerCommande(ClientDto  client, Collection<ArticleDto> articles);
}
```

```
// Implémentation du service de commande
@Stateless
public class CommandeServiceImpl implements CommandeService {
     /** Injection de l'EJB calculateur de tarif */
     @EJB CalculateurTarif calculateurDuTarif;

     public void creerCommande(ClientDto  client, Collection<ArticleDto> articles) {
          double prixTotal = 0;
          for(ArticleDto article : articles) {
               prixTotal += calculateurDuTarif.calculerTarif(1d, article);
          }
          System.out.println("Commande créé pour " + client + " Montant total:" + prixTotal);
     }
}
```

Exemple Injection de dépendances EJB 3.0 (3)

- **En résumé :**
  - Le service **`CalculateurTarif`** est défini en tant que EJB Remote (Interface)
  - Une implémentation particulière nommée **`CalculateurTarifImpl`** a été réalisée
  - L'implémentation **`CommandeServiceImpl`** a besoin du service **`CalculateurTarif`** pour réaliser l'opération **`creerCommande`**
  - Le service de commande déclare la dépendance vers **`CalculateurTarif`** sous la forme de sa propriété **`calculateurDuTarif`** annotée avec **`@EJB`**
  - Lorsque le container fournira le **`CommandeService`** à un client, ce sera une instance de **`CommandeServiceImpl`** dont la propriétés **`calculateurDuTarif`** sera alimentée avec une instance de **`CalculateurTarifImpl.`**

Injection de dépendances EJB 3.0 (2)

- **L'annotation @Resource (javax.annotation.Resource) :**

- Elle sert à **marquer les besoins en ressource** d'un EJB

Ressources : Connexion JDBC, Connexion JMS, Contexte de Session, etc.

- Les **ressources ont été enregistrées** dans l'annuaire **JNDI** sur le serveur.
- Le **nom** sous lequel a été enregistrée une ressource est utilisé pour demander son injection.

Exemple d'injection d'une dataSource JDBC:

```
@Resource(name = "jdbc/MaConnexion")
javax.sql.DataSource dataSource;
```

Injection de dépendances EJB 3.0 (3)

- **D'autres annotations spécifiques pour obtenir des ressources particulières :**

Exemple d'injection d'un contexte de persistence (JPA) :

```
@PersistenceContext (unitName="monUniteDePersistance")
protected EntityManager entityManager;
```

- L'**EntityManager** à injecter est spécifié par le nom du « Contexte de persistence » (nom dans la configuration persistence.xml)
- Une fois l'« Entity Manager » acquis il permet d'accéder aux fonctionnalités permettant de réaliser la persistance des objets métiers.

Injection de dépendances EJB 3.0 (4)

- Les annotations de marquage pour l'injection peuvent être appliquées sur différents éléments de la classe :

Type d'élément annoté	Type d'injection
Attribut / Propriété	**Injection « directe »**
Méthode	**Injection par « Setter »**
Constructeur	**Injection par « Constructeur »**

Injection de dépendances EJB 3.1 : CDI (1)

- **Apports de CDI (Context Dependency Injection) :**

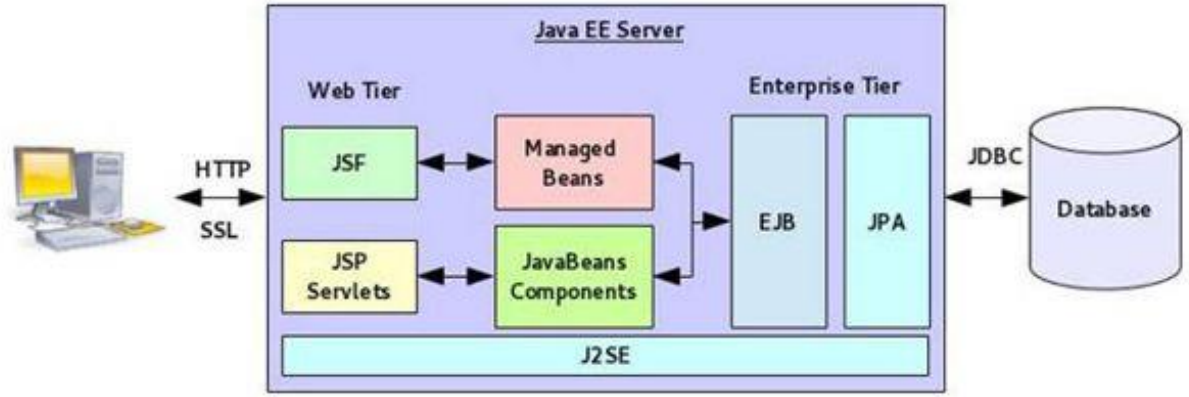
  - Injection avec une **notion de « Scope »** portée

  - Possibilité d'injecter des Pojo

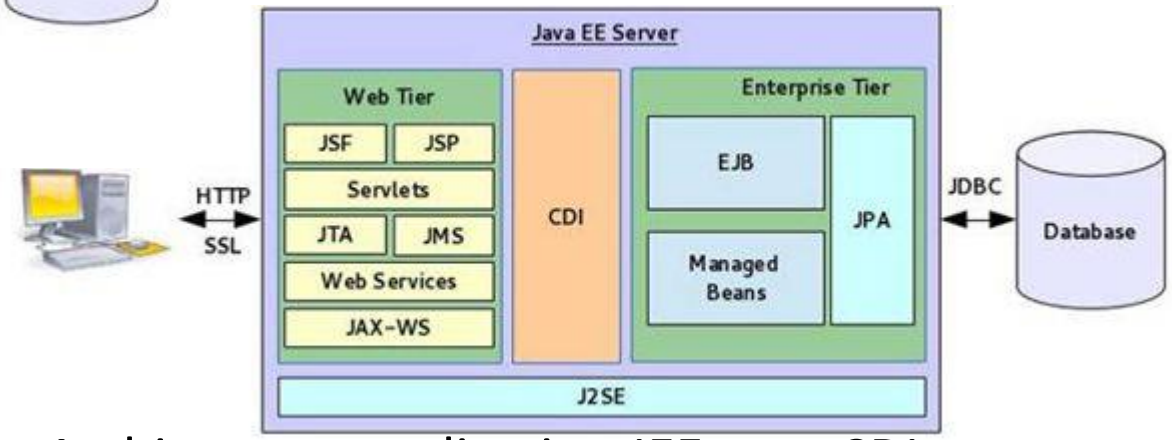
  - Accès uniforme aux EJB

Injection de dépendances EJB 3.1 : CDI (2)

- **Apports de CDI :**



Architecture application JEE sans CDI



Architecture application JEE avec CDI

Injection de dépendances EJB 3.1 : CDI (3)

- Marquage des dépendances à injecter par : **@Inject**

  - Injection avec une **notion de « Scope »** portée

  - **Possibilité d'injecter même des Pojo**

  - Accès uniforme aux EJB

Exemple Injection de dépendances EJB 3.1

- **Le service de commande**

```
// Implémentation du service de commande
@Stateless
public class CommandeServiceImpl implements CommandeService {


    /** Injection de l'EJB calculateur de tarif */
    @Inject CalculateurTarif calculateurDuTarif;

    public void creerCommande(ClientDto  client, Collection<ArticleDto> articles) {
        double prixTotal = 0;
        for(ArticleDto article : articles) {
            prixTotal += calculateurDuTarif.calculerTarif(1d, article);
        }
        System.out.println("Commande créé pour " + client + " Montant total:" + prixTotal);
    }
}
```

L’Injection : Sélection de l’instance

- **S’il existe une seule implémentation du type à injecter :**
La sélection est implicite
- **S’il n’existe pas d’implémentation du type à injecter :**
Une exception est levée
- **S’il existe plusieurs implémentations du type à injecter (indétermination) :**
Une exception est levée
- **La dépendance est fournie (injectée) par le container :**
C’est une implémentation du type demandé
- **S’il existe déjà une instance du type qui est compatible avec la Portée (Scope) :**
L’instance existante est fournie
- **Sinon une nouvelle instance est créée et fournie**

CDI : La portée (Scope)

- La Portée définit la **durée de vie d'un instance**
- **Tout objet injecté a une portée**
- **La portée d'un objet est définie** par une annotation sur sa classe

```
// Implémentation du CalculateurTarif
@Stateless
@ApplicationScoped // Portée Application
public class CalculateurTarifImpl implements CalculateurTarif {
     @Override
     public Double calculerTarif(Double quantite, ArticleDto articleDto) {
          return articleDto.getPrix() * quantite;
     }
}
```

CDI : Les portées

- Annotations du package **javax.enterprise.context** à ne pas confondre avec celles de **javax.faces.bean**
- `@ApplicationScoped:` portée application, objet créé à la première injection et supprimé à la fin de l'application (arrêt du serveur)
- `@RequestScoped:` portée requête (http) , durée de vie de l'objet limité au traitement de la requête.
- `@SessionScoped:` portée session, durée de vie de l'instances liée à la durée de connexion d'un utilisateur.
- `@ConversationScoped:` portée conversation, durée de vie de l'instances définie par l'intervalle *Conversation.begin() et Conversation.end()*.
- `@ViewScoped:` portée vue JSF, durée de vie de l'instances liée à la durée de vie de la vue (tant que l'utilisateur ne change pas de page).
- `@Dependent:` aucune portée, une nouvelle instance est créée à chaque injection. C'est la portée par défaut si aucune annotation n'est spécifiée.

Injection avec sélection de l'implémentation (1)

- **2 implémentations du service de calcul de tarif** (problème induit ?)

```
// Implémentation du CalculateurTarif
@Stateless
public class CalculateurTarifImpl implements CalculateurTarif {
	@Override
	public Double calculerTarif(Double quantite, ArticleDto articleDto) {
		return articleDto.getPrix() * quantite;
	}
}
```

```
// Implémentation du CalculateurTarif avec frais de port forfaitaire (1,5€)
@Stateless
public class CalculateurTarifImplV2 implements CalculateurTarif {
	@Override
	public Double calculerTarif(Double quantite, ArticleDto articleDto) {
		return articleDto.getPrix() * quantite + 1.5;
	}
}
```

Injection avec sélection de l'implémentation (2)

- **2 implémentations du service de calcul de tarif :**

Sélection de l'implémentation à injecter dans **`CommandeServiceImpl`** ?

Plusieurs solutions :

~~(1) Marquage de l'implémentation à utiliser par l'annotation `@Default`~~

(2) Marquage de l'implémentation et de la dépendance à utiliser avec un **« Qualifier »**

(3) Utilisation de l'annotation `@Any`

(4) Utilisation de l'annotation `@Named`

Sélection de l'implémentation avec @Default

- Marquage de l'implémentation à utiliser par l'annotation @Default

```
// Implémentation du CalculateurTarif
@Stateless
public class CalculateurTarifImpl implements CalculateurTarif {
	@Override
	public Double calculerTarif(Double quantite, ArticleDto articleDto) {
		return articleDto.getPrix() * quantite;
	}
}
```

```
// Implémentation du CalculateurTarif avec frais de port forfaitaire (1,5€)
@Default
@Stateless
public class CalculateurTarifImplV2 implements CalculateurTarif {
	@Override
	public Double calculerTarif(Double quantite, ArticleDto articleDto) {
		return articleDto.getPrix() * quantite + 1.5;
	}
}
```

NE FONCTIONNE PAS CAR TOUT SESSION BEAN COMPORTE L'ANNOTATION @Default

Sélection de l'implémentation avec un `@Qualifier` (1)

- `@Qualifier` est une annotation pour annotation (**javax.inject.Qualifier**)

Création d'une annotation `@PortFacture` utilisable en tant que Qualifier

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD,
ElementType.PARAMETER, ElementType.TYPE})
public @interface PortFacture {
 // . . .
}
```

Sélection de l'implémentation avec un `@Qualifier` (2)

- Marquage de l'implémentation et de la dépendance avec le `@Qualifier`

```
// Implémentation du CalculateurTarif avec frais de port forfaitaire (1,5€)
@PortFacture
@Stateless
public class CalculateurTarifImplV2 implements CalculateurTarif {
     @Override
     public Double calculerTarif(Double quantite, ArticleDto articleDto) {
          return articleDto.getPrix() * quantite + 1.5;
     }
}
```

Intérêts?

```
// Implémentation du service de commande
@Stateless
public class CommandeServiceImpl implements CommandeService {
     @Inject @PortFacture CalculateurTarif calculateurDuTarif;

     public void creerCommande(ClientDto  client, Collection<ArticleDto> articles) {
          // . . .
     }
}
```

Sélection de l'implémentation avec @Any

- Marquage de la dépendance avec `@Any` pour obtenir toutes les implémentations et choisir manuellement celle que l'on veut utiliser.

```
// Implémentation du service de commande
@Stateless
public class CommandeServiceImpl implements CommandeService {
     @Inject @Any javax.enterprise.inject.Instance<CalculateurTarif> calculateursDuTarifs;

     public void creerCommande(ClientDto  client, Collection<ArticleDto> articles) {
          // . . .
     }
}
```

(javax.enterprise.inject.Instance est une interface qui hérite de **java.util.Iterable)**

Intérêts?

Alias nom de l'implémentation avec @Named

- Marquage de la dépendance avec `@Named` pour définir un nom auquel il peut être fait référence dans les EL (expression Language) de JSF.

```
// Implémentation du service de commande
@Stateless
public class CommandeServiceImpl implements CommandeService {
     @Inject @Named(«CalcTar») CalculateurTarif calculateursDuTarifs;

     public void creerCommande(ClientDto client, Collection<ArticleDto> articles) {
          // . . .
     }
}
```

Un texte dont la valeur est définie en EL en JSF:

<h:outputText value=« #{CalcTar. calculerTarif(2.0, article)}» />

Intérêts?

Sélection de l'implémentation avec @Alternative (1)

- Marquage de l'implémentation de la dépendance avec @Alternative.

```
// Implémentation spécifique du service de calcul
@Stateless
@Alternative
public class CalculTarifAlternatif implements CalculateurTarif {
      // ...
}
```

- Marquage de la dépendance simplement avec @Inject.

```
// Implémentation du service de commande
@Stateless
public class CommandeServiceImpl implements CommandeService {
      @Inject CalculateurTarif calculateursDuTarifs;
      public void creerCommande(ClientDto  client, Collection<ArticleDto> articles) {
            // . . .
      }
}
```

Sélection de l'implémentation avec `@Alternative` (2)

- Déclaration dans bean.xml des Alternatives à sélectionner

```
<beans ... >
       <alternatives>
              <class>tarif.service.impl.CalculTarifAlternatif</class>
       </alternatives>
</beans>
```

- Durant l'exécution les implémentation du **`CalculateurTarif`** injectées seront des **`CalculTarifAlternatif`**. En commentant la ligne class dans le bean.xml les règles standard de sélection s'appliquent.

Intérêt de @Alternative ?

Production de l'implémentation avec `@Produces` (1)

- Déclaration d'une méthode marquée `@Produces` en mesure de fournir l'implémentation d'une dépendance.

```
@Stateless
public class ItemProducer {
     @Produces
     public Item creerItem() {
          return new Item();
     }
}
```

- La demande d'injection d'un **`Item`** marquée par **`@Inject`** provoquera l'invocation de la méthode **`creerItem()`**.
- Il est possible d'utiliser des **Qualifier** pour **sélectionner les Producteurs** à utiliser

Production de l'implémentation avec @Produces (2)

- Pour spécifier le traitement à réaliser à la fin de l'utilisation d'un objet « Produit » on utilise une « **Disposer Method** »

```
@Stateless
public class ItemProducer {
     @Produces
     public Item creerItem() {
          return new Item();
     }

     public void libererItem(@Disposes Item item) {
          item.libererRessources();
     }
}
```

- Si la production a utilisé un **Qualifier** pour **sélectionner le Producteur** à utiliser Alors le `@Disposes` doit être complété par ce même **Qualifier**. Le « Disposer » est appelé par le container quand une instance créée par @Produces est détruite.

https://docs.oracle.com/javaee/6/api/javax/enterprise/inject/Disposes.html

Injection de beans prédéfinis (Ressource / Contexte)

- JEE permet d'injecter des ressources et beans prédéfinis

```
@Resource  UserTransaction transaction;
@Resource  Principal principal;
@Resource  Validator validator;
@Resource  ValidatorFactory factory;
@Inject  HttpServletRequest req;
@Inject  HttpSession session;
@Inject  ServletContext context;
```

https://javaee.github.io/tutorial/cdi-adv004.html#CJGHGDBA

CDI : Infos Complémentaires

https://javaee.github.io/tutorial/cdi-basicexamples002.html

Design Patterns avec les EJB

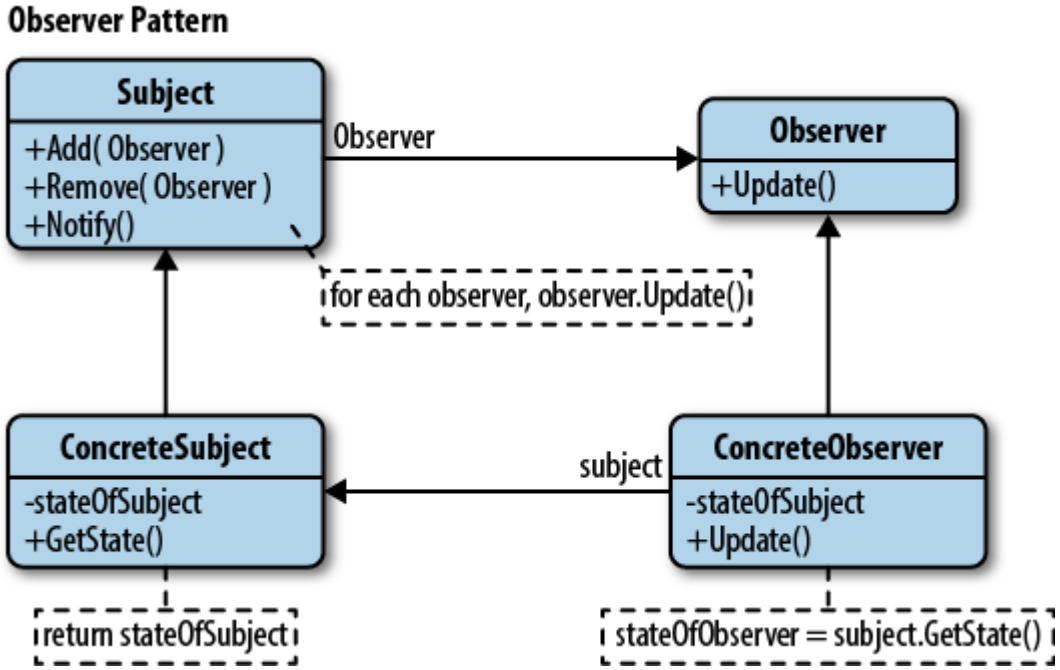
- Communication entre EJB avec **Publish / Subscribe** ,

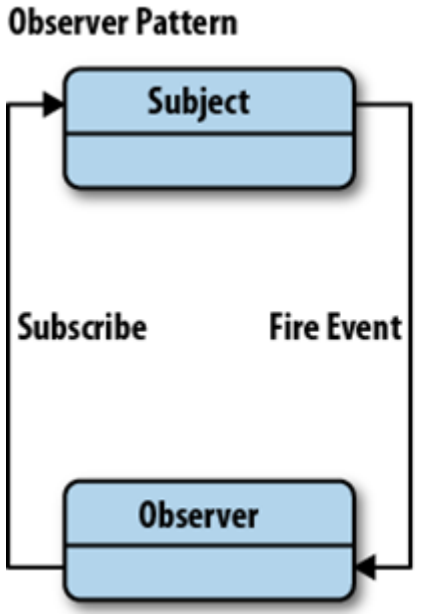
Observator

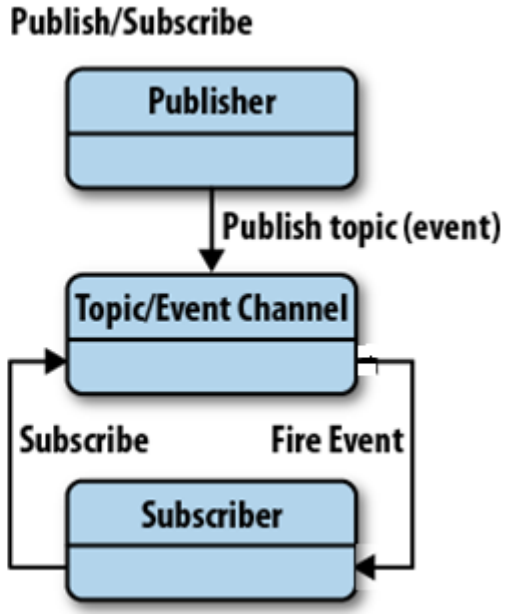
- **Interceptor** : pour intervenir sur le cycle de vie « orienté aspect »
- **Decorator :** pour étendre dynamiquement le comportement d'un objet. (Alternative à l'héritage)

Pattern Observateur et Publish/Subscribe (1)

- **Observateur <==> Publish/Subscribe:**







Communication par Event avec Pattern Observateur (2)

- **Evénement :** « signal » destiné à être créé et envoyé par un composant pour notifier d'autres composants d'une situation qui vient de se produire.

  Un événement **peut comporter des propriétés** fournissant des informations complémentaires à propos de la notification

- Les **« Event »** permettent une **communication entre EJB sans aucune dépendance à la compilation.**

- **Définition d'un type d'événement :** Classe simple mais sérialisable

```
public class UnEvenement implements Serializable{
     String information
     Integer distance
     // ...
     public UnEvenement(String info, Integer dist) {...}
}
```

Communication par Event avec Pattern Observateur (3)

- **L'Observateur / Subscriber :** le bean qui est à l'écoute et s'attend à être notifié prévenu lorsque quelque chose de particulier se produit.

- **La Notification de l'observateur est réalisée par l'invocation d'une méthode.**

La méthode prend en argument l'événement qui est annoté par `@Observes.`
Des « Qualifiers » peuvent affiner la condition de notification (type d'écoute)

```
@SessionScoped
public class UnObservateur implements Serializable{
     public void onCas1(@Observes @Cas1 UnEvenement eventCas1) {
          //...
     }
     public void onCas2(@Observes @Cas2 UnEvenement eventCas2) {
          //...
     }
}
```

Communication par Event avec Pattern Observateur (4)

- **Le Publisher :** le bean « publisher » instancie et lève les événements dans le but de notifier les « Subscribers » à cet événement.

- **Pour lever un événement, le « Publisher » se fait injecter un « Médiateur » qui permettra d'envoyer l'événement aux « Subscribers ».**

```
@SessionScoped
public class BeanPublieur {
      @Inject @Cas1
      Event<UnEvenement> publieurCas1;
      @Inject @Cas2
      Event<UnEvenement> publieurCas2;
      public void faireUnTruc(String machin) {
            //...
            // Lève événement pour le cas 1
            publieurCas1.fire(new UnEvenement("Infos_1", 0));
            // Lève événement pour le cas 2
            publieurCas2.fire(new UnEvenement("Infos_2", 22));
      }
}
```

Intérêt de cette construction ?

Communication par Event avec Pattern Observateur (5)

- **Intérêts :**

- Découplage entre Bean collaborant (pas de dépendances à la compilation)

. . .

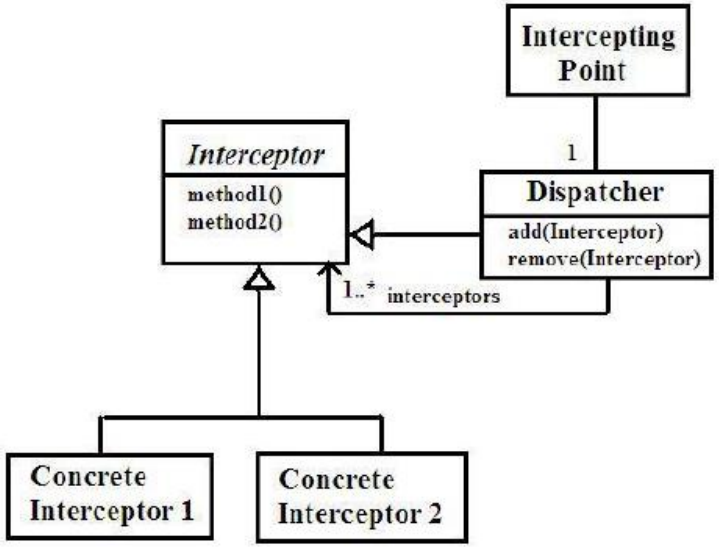
- **Limites :**

- En opposition au Publish / Subscribe classique, les abonnements sont statiques.

. . .

Pattern Interceptor (1)

- Permet de **compléter ou modifier les comportements d'un composant** en intervenant sur l'appel de ses méthodes. (**A**spect **O**riented **P**rogramming)



- Un **Interceptor** en JEE est une classe utilisée pour intervenir au moment de l'appel d'une méthode. (avant, après, à la place)

Interceptor (2)

- **Interceptor Binding :** Annotation pour définir une catégorie d'Interceptor.
  - Réalisé par la création d'une annotation nommant le type d'interceptor.
  - L'annotation doit être annotée avec **`@InterceptorBinding`**

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface CategorieInterceptor {
}
```

- **L'annotation créée permettra de :**
  - Définir **les Classes d'interceptors** de cette catégorie
  - **De marquer les classes** sur lesquelles doivent s'appliquer les interceptors de cette catégorie

Interceptor (3)

- **Interceptor :** défini par une classe comportant des méthodes devant s'exécuter durant le cycle de vie d'un bean marqué pour subir les « Interceptions » de cet interceptor.
  - Annoté avec **`@Interceptor`** pour déclarer la classe comme étant un interceptor
  - Annoté avec l'annotation de Binding créée pour la catégorie d'interceptor

```
@CategorieInterceptor
@Interceptor
public class CategorieInterceptorImpl {
     @AroundInvoque
     public void surInterception(InvocationContext invocationContext) throws Exception {
          //...
          System.out.println("Entering method: "
                 + invocationContext.getMethod().getName() + " in class "
                 + invocationContext.getMethod().getDeclaringClass().getName());
        return invocationContext.proceed();
     }
}
```

https://docs.oracle.com/javaee/7/api/javax/interceptor/InvocationContext.html

Interceptor (4)

- **Exemple d'interceptor :**

```
@CategorieInterceptor
@Interceptor
public class CategorieInterceptorImpl {
     @AroundInvoque
     public void surInterception(InvocationContext invocationContext) throws Exception {
          // Force à null le premier paramètre avant appel
          Object[] parameters = ctx.getParameters();
          parameters[0] = null;
          ctx.setParameters(parameters);
     }
}
```

https://docs.oracle.com/javaee/6/tutorial/doc/gkeci.html

Interceptor (4)

- **Interception des appels d'un Bean :** Annotation des méthodes à intercepter ou de la classe si toutes les méthodes doivent être interceptées. Utilisation annotation avec l'annotation de Binding créée pour la catégorie d'interceptor

```
// Implémentation du service de commande
@CategorieInterceptor
@Stateless
public class CommandeServiceImpl implements CommandeService {

     @Inject CalculateurTarif calculateursDuTarifs;

     public void creerCommande(ClientDto  client, Collection<ArticleDto> articles) {
          // . . .
     }
}
```

Interceptor (5)

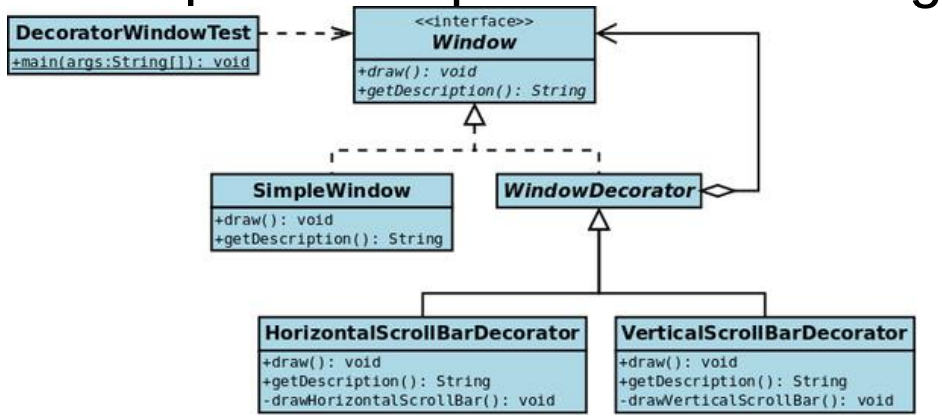
- **Activation des interceptors :**

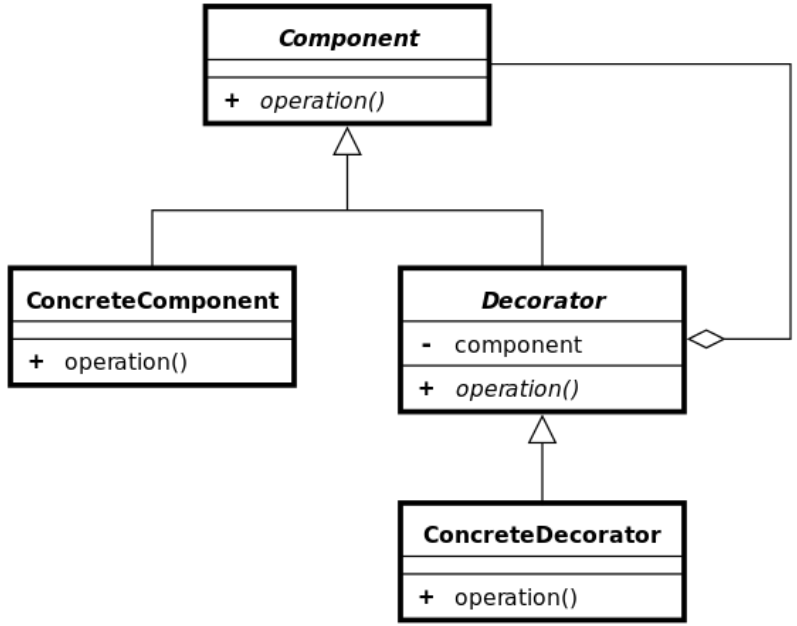
Déclaration dans le fichier bean.xml des Interceptors à activer.

```
<beans ... >
       <interceptors>
              <class>customInterceptors.CategorieInterceptorImpl</class>
       </interceptors >
</beans>
```

Pattern Decorator (1)

- Permet d'étendre dynamiquement les responsabilités (comportements) d'un objet.
  - Alternative à l'héritage.
  - Indispensable pour éviter héritage multiple





- Lorsqu'une hiérarchie d'héritage conduit à une explosion combinatoire:

  L'utilisation d'un Pattern Décorateur devrait résoudre ce problème

Decorator (2)

- **Implementation d'un Decorator :**
  - Annotation de la classe pour la marquer comme décorateur **`@Decorator`**.
  - Injection du décoré par annotation **`@Inject`** et **`@Delegate`**
  - Redéfinition des méthode décorée (possibilité de créer des Decorator abstraits)

```
// Decorateur de CalculateurTarif : Frais préparation de 0.5€ par article
@Decorator
public class CalculateurTarifDecorator implements CalculateurTarif {

	@Inject
	@Delegate
	CalculateurTarif decore;

	@Override
	public Double calculerTarif(Double quantite, ArticleDto articleDto) {
		return decore.calculerTarif(quantite, articleDto) + (quantite * 0.5);
	}
}
```

Quelle différence entre Decorator et Interceptor ?

Decorator (3)

- **Activation des Decorator :**
  - Déclaration dans le fichier bean.xml des décorateurs à activer.
  - Si il existe plusieurs décorateurs actifs: l'ordre de déclaration dans bean.xml détermine le « montage»

```
<beans ... >
        <decorators>
                <class>tarif.service.CalculateurTarifDecorator</class>
        </decorators>
</beans>
```

- **Utilisation :**
  - L'injection de l'interface décoré provoque l'injection du décorateur.
  - Il existe des possibilités à base de « Qualifier » pour injecter des decorés ou bien des instances non décorés

Les Message-Driven Bean (MDB) (1)

- **Bean récepteur de messages JMS (Java Messaging Service) :**

- **JMS: M**essage **O**riented **M**iddleware de Java
- **Listener** d'un EndPoint JMS, **Producer** de messages
- Consomme messages produits par application Java ou non (protocole JMS)
- Réception et traitement de messages en mode asynchrone
- MDB sont **clients d'autres Sessions Bean** pour traiter les messages

Cas d'utilisation :

EDI: *échange de données informatisées (documents: commandes, factures etc.)*

Les Message-Driven Bean (MDB) (2)

- **Bean récepteur de messages JMS (Java Messaging Service) :**

- Bean **non accessible** par des clients (Autonome)

- Le MDB est **« Stateless »** *(rappel …)*

- Equivalence entre toutes les instances d'un type de MDB **(Pooling possible)**

- Consomme messages produits par application Java ou non (protocole JMS)

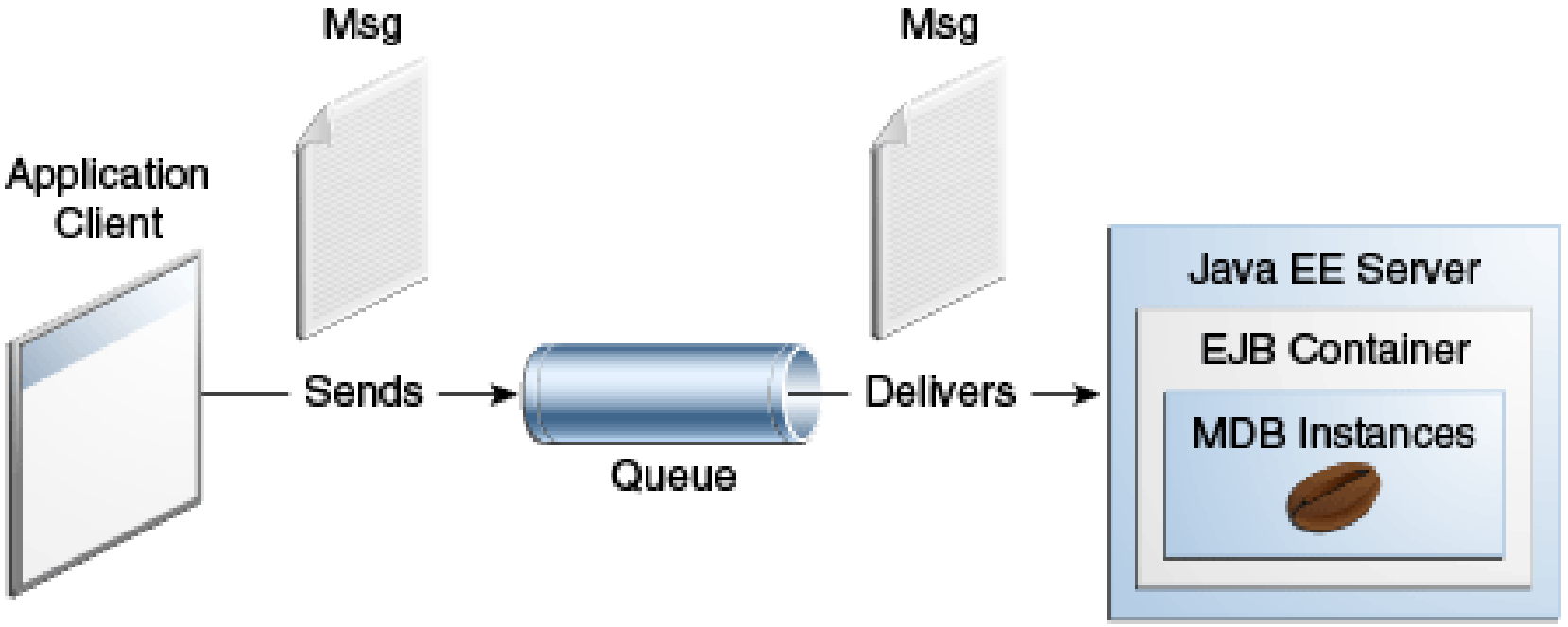
- Réception et traitement de message en mode asynchrone

- Production et transmission de messages

- Un MDB traite les messages de plusieurs clients

Les Message-Driven Bean (MDB) (3)

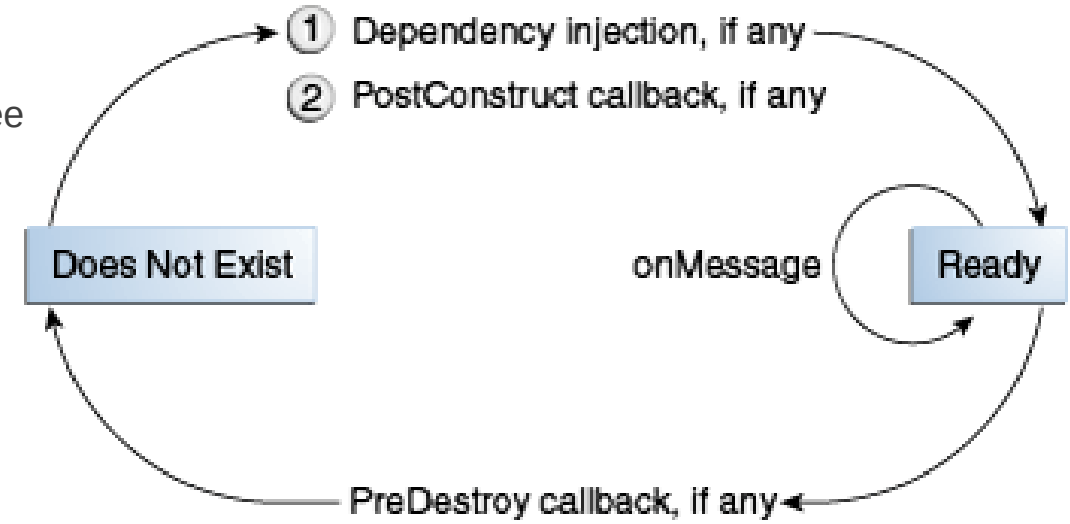
- **Application utilisatrice de JMS avec un MDB:**



Les Message-Driven Bean (MDB) (4)

- **Le cycle de vie d'un Message-Driven Bean :**

- Le container crée un Pool de MDB avant toute création
- Le container crée une instance du MDB
- Le container réalise l'injection de dépendance
- Si une méthode est annotée @PostConstruct elle est appelée
- Le MDB est prêt à recevoir et traiter les messages



- A chaque réception de message la méthode `onMessage(Message msg)` est invoquée
- A la fin du cycle de vie si une méthode est annotée @PreDestroy elle est appelée

A venir: cours plus approfondi sur JMS et les MOM

Les EJB Timer (1)

- EJB utilisateur du **TimerService de l'EJB Container**
- Pour **lancer des traitements sur un évènement temporel**
- Permet la planification de l'exécution d'une méthode :

  - Par programme en utilisant le TimerService et une méthode pour recevoir les notifications

  - Par utilisation de l'annotation avec **@Schedule** de la méthode à planifier

Les EJB Timer (2)

- EJB Timer par programme

```
@Stateless
public class TimerDemo {
    @Resource
    private TimerService timerService;

    @PostConstruct
    private void init() {
        // Création du timer: première expiration au bout
        // d'une seconde puis toutes les 2 secondes
        timerService.createTimer(1000, 2000, "IntervalTimerDemo_Info");

        // Création d'un timer sans "récurence"
        // expiration au bout de 5 secondes
        TimerConfig timerConfig = new TimerConfig();
        timerConfig.setInfo("SingleActionTimerDemo_Info");
        timerService.createSingleActionTimer(5000, timerConfig);

        // Création d'un timer sans "récurence"
        // expiration au bout de 1 minute
        Date expirationDate = new Date(System.currentTimeMillis() + 60000);
        timerService.createSingleActionTimer(expirationDate, new TimerConfig());
    }
```

Les EJB Timer (3)

- EJB Timer par programme: handler pour recevoir la notification d'expiration

```
@Stateless
public class TimerDemo {
    @Resource
    private TimerService timerService;

    @PostConstruct
    private void init() {
        // . . .
    }

    // Handler pour recevoir la notification d'expiration d'un timer
    @Timeout
    public void execute(Timer timer) {
        System.out.println("Timer Service : " + timer.getInfo());
        System.out.println("Current Time : " + new Date());
        System.out.println("Next Timeout : " + timer.getNextTimeout());
        System.out.println("Time Remaining : " + timer.getTimeRemaining());
    }
```

Les EJB Timer (4)

- EJB Timer par « Schedule »

Attribute	Description	Default Value	Allowable Values and Examples
`second`	One or more seconds within a minute	0	0 to 59. For example: `second="30"`.
`minute`	One or more minutes within an hour	0	0 to 59. For example: `minute="15"`.
`hour`	One or more hours within a day	0	0 to 23. For example: `hour="13"`.
`dayOfWeek`	One or more days within a week	*	0 to 7 (both 0 and 7 refer to Sunday). For example: `dayOfWeek="3"`. `Sun, Mon, Tue, Wed, Thu, Fri, Sat`. For example: `dayOfWeek="Mon"`.
`dayOfMonth`	One or more days within a month	*	1 to 31. For example: `dayOfMonth="15"`. −7 to −1 (a negative number means the *n*th day or days before the end of the month). For example: `dayOfMonth="-3"`. `Last`. For example: `dayOfMonth="Last"`. `[1st, 2nd, 3rd, 4th, 5th, Last] [Sun, Mon, Tue, Wed, Thu, Fri, Sat]`. For example: `dayOfMonth="2nd Fri"`.
`month`	One or more months within a year	*	1 to 12. For example: `month="7"`. `Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec`. For example: `month="July"`.
`year`	A particular calendar year	*	A four–digit calendar year. For example: `year="2011"`.

Les EJB Timer (5)

- Planification par des expression de « Schedule »

```
second="*/10", minute="*", hour="8-23", dayOfWeek="Mon-Fri", dayOfMonth="*", month="*",
year="*", info="MyTimer“
```

- **Le joker ‘*’:** toutes les valeurs acceptables

  Exemple: *minute="*" pour toutes les minutes*

- **Liste de valeurs :** plusieurs valeurs séparées par une virgule

  Exemple: *hour=“1,6,20" pour 1h, 6h et 20h*

- **Plage de valeurs :** plage de valeur exprimé avec le tiret ‘-’

  Exemple: *hour=“8-23" pour toutes les heures entre 8h et 23h* *(possibilité de mixer Listes et plages)*

- **Intervalles de valeurs :** plage de valeur exprimé avec le tiret ‘-’

  Exemple: *minute=“*/10" pour toutes les 10 minutes*

https://docs.oracle.com/javaee/6/tutorial/doc/bnboy.html#giqmx

Les EJB Timer (6)

- EJB Timer par « Schedule »

```
@Stateless
public class TimerDemoSchedule {
     // . . .
      @PostConstruct
      private void init() {
        TimerConfig timerConfig = new TimerConfig();
         timerConfig.setInfo("CalendarProgTimerDemo_Info");
         ScheduleExpression schedule = new ScheduleExpression();
         schedule.hour("*").minute("*").second("13,34,57");
         timerService.createCalendarTimer(schedule, timerConfig);
      }

      @Timeout
      public void execute(Timer timer) {
            // . . .
      }

      @Schedule(second="*/10", minute="*", hour="8-23", dayOfWeek="Mon-Fri", dayOfMonth="*", month="*", year="*", info="MyTimer")
      private void scheduledTimeout(final Timer t) {
         System.out.println("@Schedule called at: " + new java.util.Date());
      }
}
```

Les Entity Bean (1)

- **Les Bean Entités :**

- Représentent des objets métiers persistants
- Liens vers d'autres entités
- La persistance est réalisée dans des BDD relationnelles (MOR)
- Représentent les données manipulées par les session bean
- Peuvent être manipulées dans des transactions (JTA)
- Une entité survit aux pannes et aux redémarrages du serveur
- Gérées par un outil ORM, implémentation de la spécification JPA

A venir: cours plus approfondi sur JPA et la persistance

Exemples :

Un « Produit », « Un Catalogue de produit»

Conclusion sur les EJB

- Plusieurs types (Session, Entity, Message Driven, Timer)
- Destinés à différents types de clients : Standalone, Web, autres EJB
- Clients Distants ou Locaux
- Fournis par JNDI ou mécanismes d’injections
- Quelques restrictions