L'architecture J2EE

5 - 1

Plan

- ***L'architecture J2EE***
- L'architecture Enterprise Java Beans
- Services de base
- Les Sessions Beans
- Les Entity Beans
- Les Message-Driven Beans
- EJB 3.0
- Patrons de conception J2EE

Enterprise Java Beans
5 - 2

Défis du génie logiciel de nos jours

- La réalité d'aujourd'hui requiert le développement d'applications *distribuées* et *performantes*
  - Intégration de systèmes d'information d'une entreprise (qui peut être géographiquement distribuée)
  - Intégration de systèmes d'informations de plusieurs entreprises, à l'occasion de relations d'affaires (B2B)
- Le développement de ces applications est très complexe, et ce, pour plusieurs raisons:
  - Défis techniques dûs intégralement à l'infrastructure de calcul (réseaux de communication, distribution de charge, robustesse ("fault-tolerance")
  - Difficulté de mise en place de certains services (transaction, sécurité)
  - INTER-OPÉRABILITÉ!!!!!

Quatre approches différentes

- Ne rien faire (envoyer des fichiers par e-mail ou des hardcopy)
- Développer vous même *et* l'infrastructure de déploiement et d'exécution *et* la logique d'affaires
- Utiliser l'infrastructure de déploiement et d'exécution d'un fournisseur et vous concentrer sur la logique d'affaire
  - Utiliser la solution propriétaire d'UN fournisseur
  - Se mettre d'accord, entre fournisseurs et clients, sur une *architecture*, permettant à plusieurs fournisseurs de supporter cette architecture, et aux clients de pouvoir choisir sans se pré-occuper de l'inter-opérabilité des applications

Trois grandes tendances

- Au début (88-93):
  - COM/DCOM: solution propriétaire de MICROSOFT
    - COM marche mieux que DCOM/COM+
  - CORBA: standard développé par l'OMG, un organisme volontaire de l'industrie regroupant clients et fournisseurs de technologie
  - Que faire:
    - COM est plus simple à utiliser dans les environnements Windows
    - Une base d'implantation assez importante
    - CORBA est plus ouvert
    - Produits pas mûrs
    - Pas d'inter-opérabilité
    - Il fallait choisir entre les deux

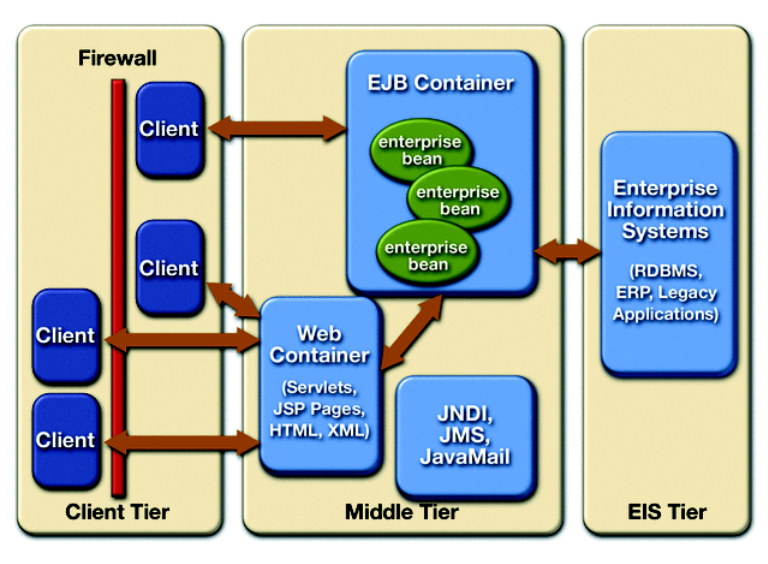
Trois grandes tendances

- Aprés (93 – 96):
  - MICROSOFT et OMG se rendent compte qu'ils doivent offrir des ponts entre leurs technologies pour que les choix de technologie soient moins pénibles et moins critiques
  - Les outils CORBA murissent, mais le développement d'applications CORBA n'en demeure pas moins compliqué
- Avant hier (96/97 – 00)
  - Les technologies Java rentrent dans le marché d'abord par Java RMI:
    - Infrastructure RMI simple (beaucoup plus simple que CORBA)
    - Exige du Java de bout en bout
    - On parle d'inter-opérabilité avec Microsoft
  - L'architecture J2EE, qui offre plus de services que COM et CORBA

Trois grandes tendances

- Hier (00 – 02)
  - Microsoft sort .NET et C# pour concurrencer J2EE et Java
  - Une pénétration assez forte de la technologie J2EE dans le marché nord-américain
  - Un "alphabet soup" de technologies provenant de différents vendeurs
  - Plusieurs couches de technologie bâties par dessus (et des fois à côté) des technologies de base
- Ce matin (02 – aujourd'hui)
  - Beaucoup de technologies de distribution plus légères que J2EE, CORBA ou COM/DCOM, dont les services web
  - Beaucoup d'environnements open-source
  - EJB3

L'architecture J2EE - Principes



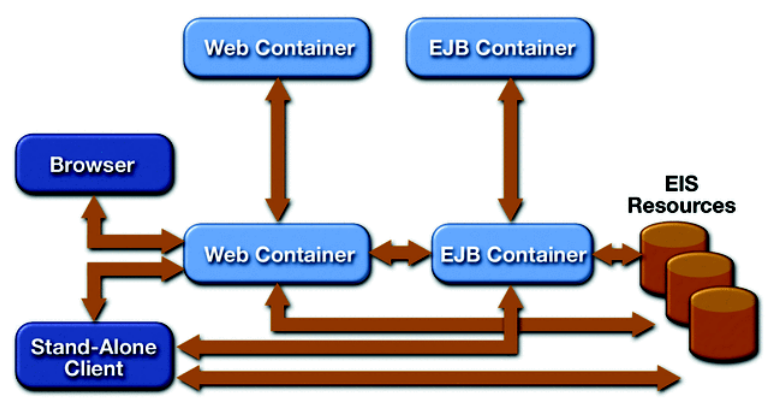
L'architecture J2EE

C'est une architecture trois/quatre tiers:

- Tiers *client*: couche assez mince, contenant la logique d'invocation et d'affichage des données
- Tiers *application*: reflète la logique d'affaires (orientée-objet), riche en fonctionnalités, et offrant plusieurs services "architecturaux"
- Tiers *données*, offrant la persistence et l'accès centralisé aux données

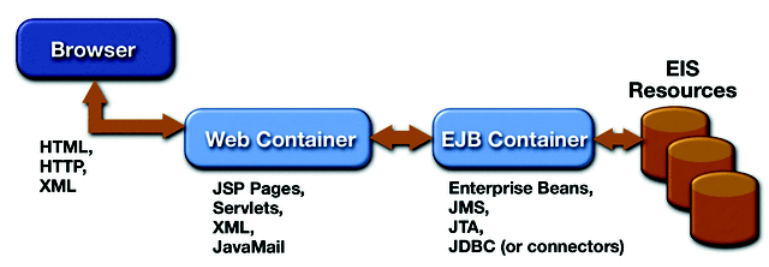
Enterprise Java Beans
5 - 9

Plusieurs scénarios de déploiement



Enterprise Java Beans
5 - 10

Un scénario quatre tiers simple

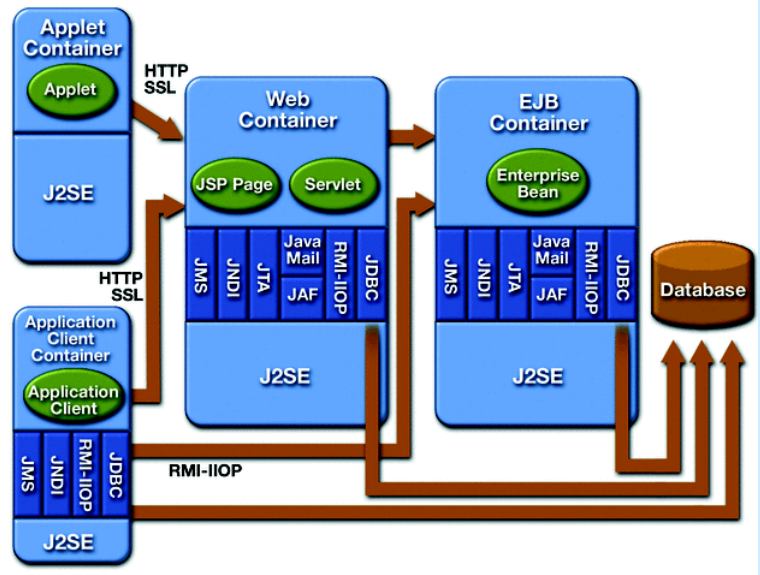


pages HTML contenant applet qui utilise session et entity beans

L'architecture J2EE

- Sun spécifie des composants au niveau de chacun des tiers, et fournit des outils/classes dans chacun de ces tiers
- Pour ce faire, elle utilise beaucoup de technologies

Résumé



Plan

- **L'architecture J2EE**
- ***L'architecture Enterprise Java Beans***
- Services de base
- Les Sessions Beans
- Les Entity Beans
- Les Message-Driven Beans
- EJB 3.0
- Patrons de conception J2EE

Enterprise Java Beans

- "The Enterprise Java Beans ***architecture*** is a ***component architecture*** for the development and deployment of ***object-oriented distributed*** enterprise-level applications. Applications written using the Enterprise Java Beans architecture are ***scalable***, ***transactional***, and ***multi-user secure***. These applications may be written once, and ***deployed*** on any ***server platform*** that supports the Enterprise Java Beans ***specification***"

  © Sun Microsystems's EJB specification, 1998

L'architecture EJB

Cette architecture spécifie

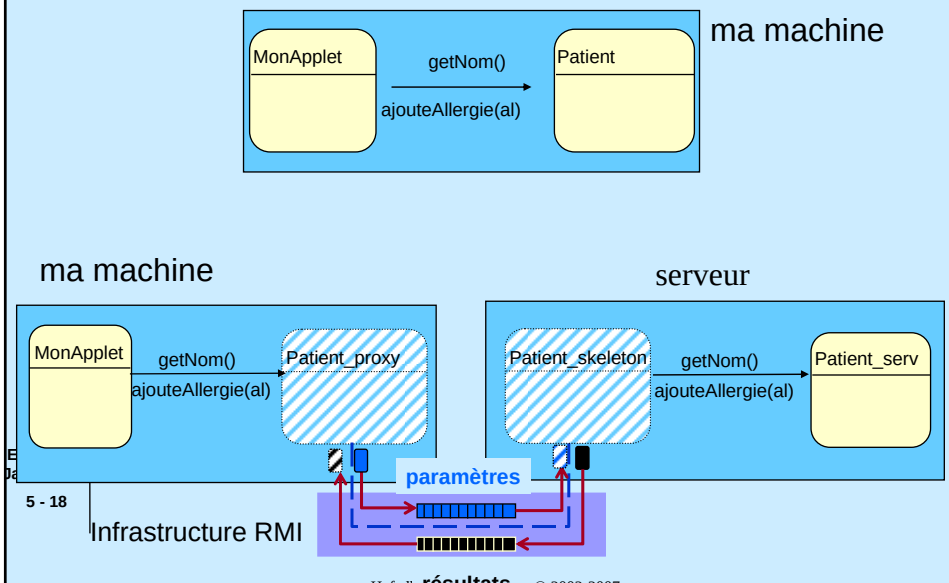
- Les services que doit support un serveur EJB ("EJB-compliant")
- Les interfaces (*contrats*) que les classes ou composants d'affaire doivent supporter pour pouvoir bénéficier de ces services

  =>
- Le *modèle de composants de* EJB

Caractéristiques de serveurs EJB

- Infrastructure d' objects distribués
- Services de concurrence (accès partagé aux mêmes ressources)
- Service de transactions (ACID)
- Distribution de charge
- Gestion (optimale) des ressources
- Sécurité

Distribution d' objets



Distribution d'objets (2)

- "Client" et "serveur" roulent dans deux processus et possiblement machines différentes
- Un "représentant local" (proxy) prend lieu de l'objet distant du côté du client, et se comporte comme son représentant
- Le passage de paramètres à travers le réseau se fait par sérialisation et désérialisation de données (objets)

- L'identité d'un objet n'est plus liée à un espace mémoire local, mais devient identifiable à travers le réseau par d'autres caractéristiques

Infrastructures "RMI" existantes

- ORBs, Java RMI, DCOM (et le TP2) offrent toutes l'invocation de méthodes à distance
- Dans chacune de ces technologies là, les outils de déploiement génèrent des "proxies", du côté client, et des "skelettes", du côté serveur, à partir d'une description "abstraite" (*interface*) de la fonctionnalité de l'objet:
  - CORBA utilise CORBA IDL
  - COM utilise COM IDL
  - Java RMI utilise … les interfaces Java

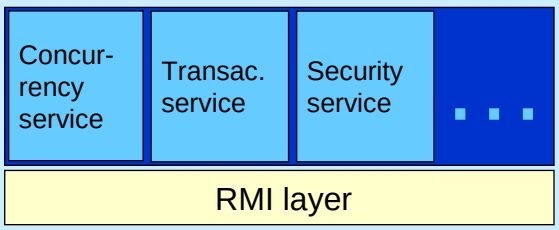
- Le développeur doit implanter la fonctionnalité d'affaires du côté serveur

Infrastructures existantes

- Plusieurs infrastructures offrent (ou dépendent de) un service d'adressage/nommage global:
  - Ce service permet d'attribuer des "noms" à des objets distants, et de les retrouver avec ces noms là
- Peuvent offrir d'autres services qui accompagnent ces services de base:
  - Transactions (distribuées, two-phase commit) (CORBA et probablement DCOM)
  - Lifecycle (COM/DCOM et CORBA)
  - Relations (CORBA)
  - Sécurité (CORBA, COM/DCOM, etc.)

Vue d'ensemble de l'architecture EJB

- La couche RMI est nécessaire mais largement insuffisante
- CORBA offre beaucoup de services, mais les développeurs doivent les invoquer, et règler les problèmes d'intéractions de services.
- La plupart des services EJB sont offerts d'office et configurables au moment du *déploiement*



Variétés de Beans

Il y a trois (quatre) grandes classes de beans:

- ***Entity beans***: Ces bean représentent les entités du domaine d'affaires; ce sont les objets qui vont survivre au delà d'une intéraction client-serveur, et peuvent être partagés par plusieurs usagers (les serveur EJB gèrent l'accès partagé)
- ***Session beans***: elles représentent des processus d'affaires; sont transientes. Typiquement, elles implantent des *use-cases*. Elles sont spécifiques à l'usager/client. Deux sortes
  - ***Stateless beans***: chaque appel de méthode est indépendant de l'appel suivant
  - ***Stateful beans***: les méthodes appelées ont des effets de bord qui sont utilisés pour les appels suivants. Représentent des transactions complexes
- ***Message-driven beans***: une communication asynchrone via des files d'attentes (JMS)

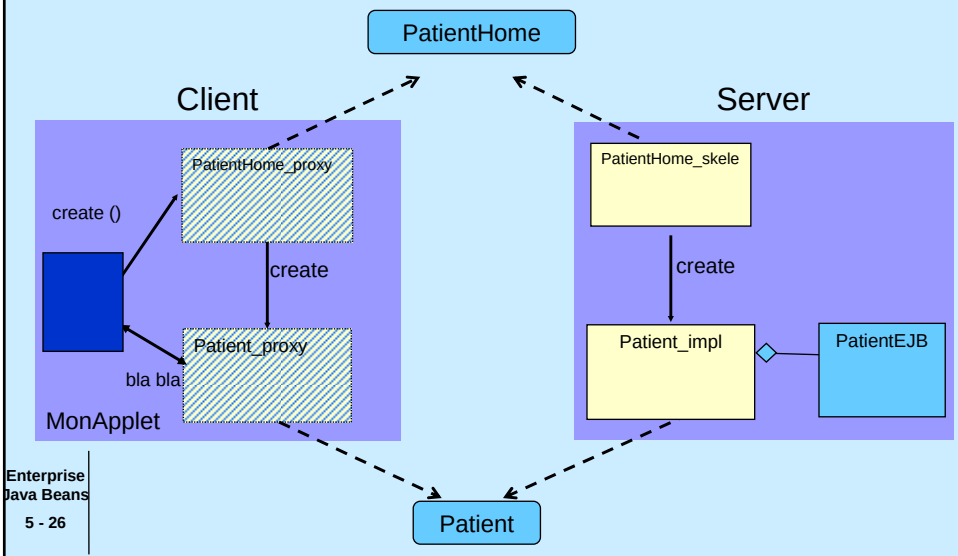
Type de beans

- La différence entre EntityBean, SessionBean, et MessageDrivenBean est importante pour le développeur (différentes conventions, différentes méthodes à implanter) et pour les serveurs
- La différence entre *stateful* SessionBean et *stateless* SessionBean ne fait pas de différence au niveau de la programmation:
  - Elle est spécifiée au moment du déploiement
  - Elle influence la façon dont le serveur traite les beans

Composants d'une EJB (1.x)

- L'interface *remote* (*interface*): décrit la logique d'affaires. Doit être une extension de l'interface *EJBObject*
- L'interface *home* (*interface*): représente l'objet qui nous permet de créér et rechercher les objets de la bean. Cette interface doit être une extension de *EJBHome*
- La classe *bean* (*class*): cette classe doit *implements* l'interface *EntityBean* (*SessionBean*), et doit fournir une implantation pour les méthodes de l'interface *remote* et celles de l'interface *home.*
- Les interfaces *EntityBean* et *SessionBean* incluent des méthodes "callback" qui seront appelées par le serveur pour la gestion du cycle de vie et de la persistence des beans. Les classes "bean" doivent implanter ces méthodes

Vue globale simplifiée



Exemple: une session bean qui donne l'heure

```
package quelleheureestil;
import javax.ejb.EJBObject;
import java.util.Calendar;
import java.rmi.RemoteException;
import javax.ejb.EJBException;

/**
 * Title:        Illustration de la plus simple bean
*/
public interface DonneLHeure extends EJBObject {

  public Calendar quelleHeureEstIl() throws RemoteException;
  public Calendar etEnTerreNeuve() throws RemoteException;
}
```


Home interface

```
package quelleheureestil;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.util.Calendar;
import java.rmi.RemoteException;
import javax.ejb.EJBException;

public interface DonneLHeureHome extends EJBHome {
  public DonneLHeure create() throws RemoteException,
    CreateException;
}
```

L'implantation

```
public class DonneLHeureEJB implements SessionBean {
  public DonneLHeureEJB() {
    System.out.println("Creation d'un donneur de l'heure");
  }

/** implantation de la methode de la remote interface
 */
public Calendar quelleHeureEstIl() throws RemoteException {
    System.out.println("On me demande l'Heure");
    Calendar heure = GregorianCalendar.getInstance();
    return heure;
}
public Calendar etEnTerreNeuve() throws RemoteException {
    System.out.println("On me demande l'Heure en Terre Neuve");
    Calendar heure = GregorianCalendar.getInstance(TimeZone.getTimeZone("GMT-4:30"));
    return heure;
}
public void ejbCreate() throws RemoteException, CreateException { }
public void ejbActivate() throws EJBException, RemoteException {}
public void ejbPassivate() throws EJBException, RemoteException {}
public void ejbRemove() throws EJBException, RemoteException {}
public void setSessionContext(SessionContext parm1) throws EJBException, RemoteException {}
}
```


L'application client

```
public class Principale {
  public Principale() {
  }
  public static void main(String[] args) {
    try {
    // obtiens une instance du service de nommage
    Context instanceDeNamingService = new InitialContext();
    // obtiens reference au serveur du serveur de temps
    Object reference = instanceDeNamingService.lookup("ServeurDuServeurDeTemps");
    // faire un cast sécuritaire à DonneLHeureHome
    DonneLHeureHome serveurDuServeurDetemps =
      (DonneLHeureHome)PortableRemoteObject.narrow(reference,quelleheureestil.DonneLHeureHome
      .class);
    // obtiens un donneur d'heure
    DonneLHeure serveurDeTemps = serveurDuServeurDetemps.create();
    System.out.println("L'heure chez le serveur est " +serveurDeTemps.quelleHeureEstIl());
    }
    catch (Exception exception) {
      exception.printStackTrace();
    }
  }
}
```


Composants d'une EJB (2.x)

- La mécanique de communication et d'invocation est trop lourde lorsque client et serveur vivent dans la même JVM dans le même container
- EJB 2.x offre la possibilité de spécifier *deux modes d'invocation* des EJB
  - Mode remote: celui vu précédemment
  - Mode local: fournir deux interfaces additionnelles, une pour home et une pour les fonctionnalités d'affaires
- =>
- En plus des interfaces *remote* (extension de l'interface *EJBObject*), *home* (extension de *EJBHome*), et de la *bean class,* on doit fournir une interface *local* (extension de *EJBLocalObject*), et une interface *local home* (extension de *EJBLocalHome*)
- La classe *bean* (*class*) doit alors implanter les cinq interfaces
  - Souvent, l'interface *local* contient l'interface remote

Plan

- **L'architecture J2EE**
- **L'architecture Enterprise Java Beans**
- ***Services de base***
- Les Sessions Beans
- Les Entity Beans
- Les Message-Driven Beans
- EJB 3.0
- Patrons de conception J2EE

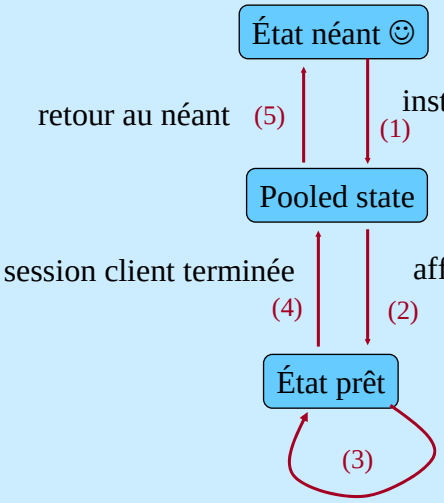
Services de base

- Gestion des ressources
  - Instance pooling
  - Distribution de charge
- Services:
  - Concurrence
  - Transactions
  - Persistence
  - Distribution
  - Naming
  - Security

Les beans

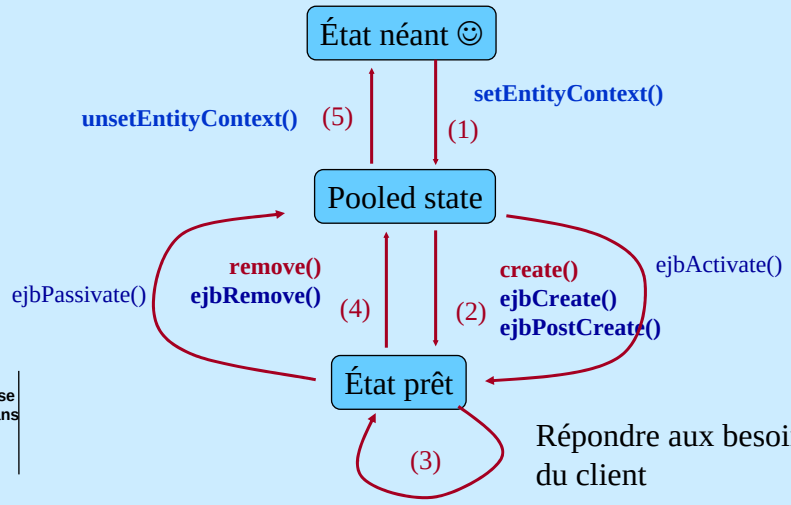
- Une application serveur peut désservir plus milliers de clients en même temps
- La création de ces beans prend pas mal de place, mais surtout, pas mal de temps (allocation dynamique de mémoire)
- On peut optimiser l'utilisation des beans de deux façons:
  - Ne pas détruire les objets quand ils ne sont plus utilisés. On les "vide", mais on attend une nouvelle requête pour un autre bean pour pouvoir les réutiliser pour d'autres programmes clients
  - Dans certains cas, on peut utiliser le même objet pour répondre à plusieurs clients (*stateless session bean*)

État d'un bean (entity)



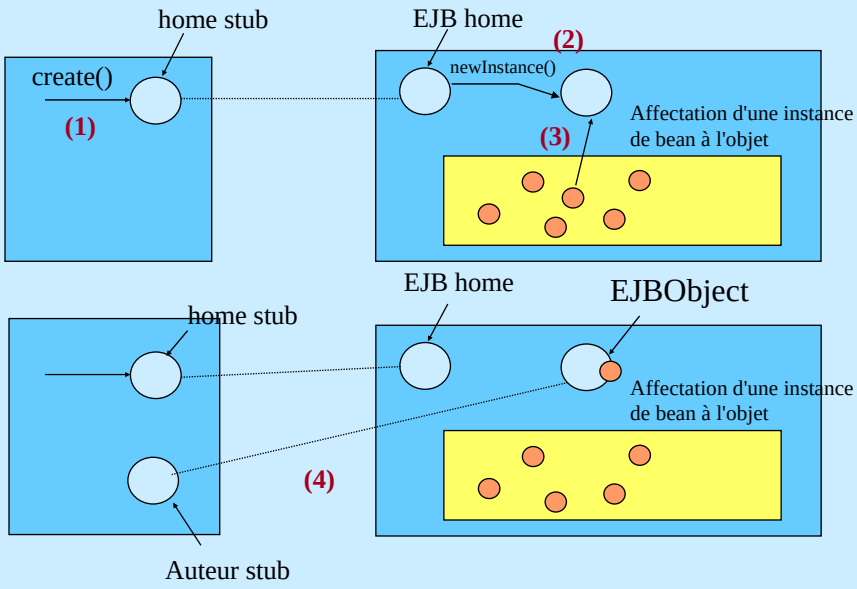
Enterprise Java Beans
5 - 35

État d'un bean (entity)



Enterprise Java Beans
5 - 36

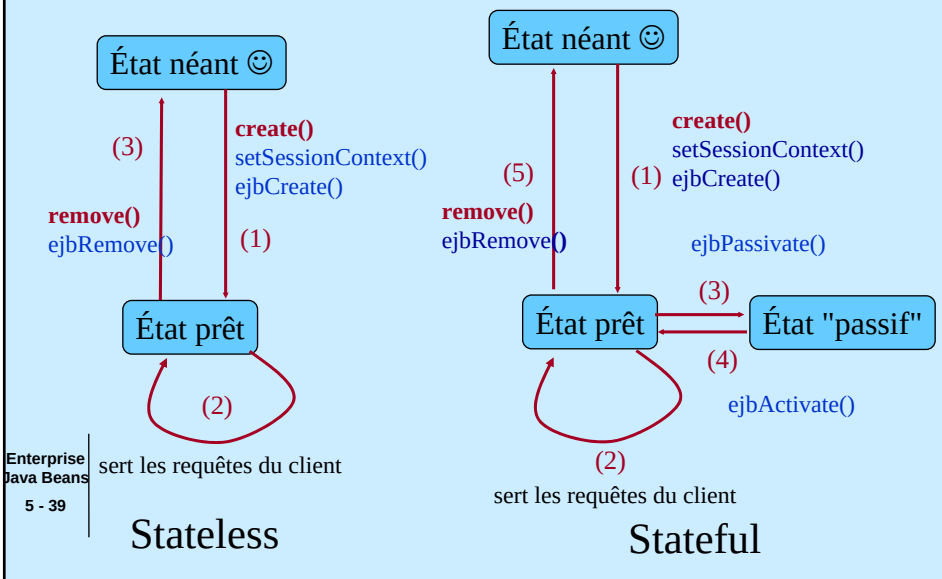
Cycle de vie d'un bean



Instance swapping

- Les session bean qui sont *stateless* peuvent être passées entre clients pendant les pauses entre requêtes d'un client
- Les session beans qui sont *stateful* sont sérialisées en mémoire secondaire lorsqu'elles ne sont pas utilisées:
  - ejbActivate(): que faire lors d'un chargement de mémoire secondaire
  - ejbPassivate(): que faire lorsqu'on s'apprête à sortir le session bean de la mémoire

Cycle de vie des session beans



Gestion des stateful session beans

- L'activation et la passivation des beans est laissée à la discrétion du "container"
- Le but est de conserver les ressources
- L'une des techniques utilisées est celle de la bean "Least Recently Used", i.e. dont l'utilisation remonte au plus longtemps
- Si tout n'est pas sérialisée, il faut veiller à reconstituer l'état de la bean (dans ejbActivate())

Distribution de charge

- L'architecture J2EE ne spécifie pas de stratégie particulière de distribution de charge, mais
- Les implantations doivent permettre la distribution de charge entre plusieurs serveurs, de façon transparente par rapport au programme client

Services

- Services de base
  - Concurrence
  - Transactions
  - Persistence
  - Distribution
  - Naming
  - Security
- Ces services sont des add-ons, qu'il faut invoquer explicitement dans CORBA
- Ils sont requis (et transparents) dans EJB

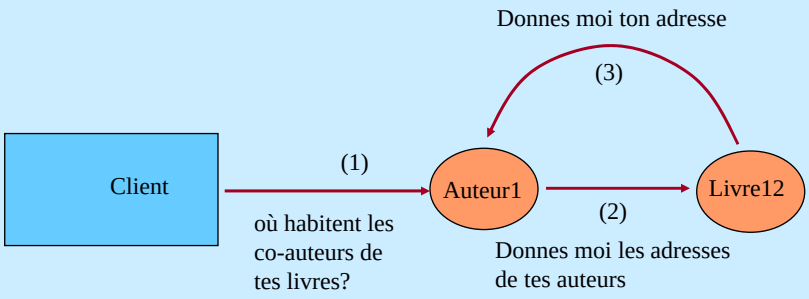
Concurrence

- Les session beans ne supportent pas la concurrence:
  - Les stateful beans sont attribuées une par programme client, et donc, pas besoin de les partager
  - Les stateless beans n'ont pas de données à faire partager entre plusieurs programmes clients, et donc, il n'y a pas lieu de les partager: on peut juste en créer d'autres pour répondre aux requêtes des clients
- Les entity beans: doivent être partagées entre programmes clients
- EJB interdit l'accès concurrent aux entity beans: chaque programme client verrouille l'entity bean qu'il manipule, durant sa manipulation (invocation de méthode)

Concurrence des entity beans

- Les containers implementent la synchronisation automatiquement: toutes les méthodes d'affaires des beans sont considérées comme "synchronized«
  - sont exécutées à l'intérieur de blocs synchronized sur l'objet)
- On ne peut pas créer de thread dans les beans
  - les containers doivent maintenir le contrôle total sur la concurrence dans le système).

Les beans ne peuvent pas être ré-entrantes



Pour le container, l'appel (3) est (1) paraissent comme venant de deux clients différents

Enterprise Java Beans
5 - 45

Transactions

- Les container gèrent les transactions automatiquement
- Durant le déploiement, le développeur spécifie les "attributs transactionnels" de chaque méthode
- Dans l'un des modes (required), le container exécute les méthodes dans des transactions (si la méthode échoue, les changements partiels effectués à la BD sont rétractés)

Enterprise Java Beans
5 - 46

Persistence

- Les entity bean sont, par définition, persistentes
- La persistence des entity beans peuvent être accomplie de deux façons:
  - Gérée automatiquement par le "container" (*container managed persistence*)
  - Gérée "à bras" par le développeur (*bean-managed persistence*)
- Pour la "container-managed persistence", chaque vendeur choisit un(e) (ensemble de) technologie(s) de persistence adaptée
- Pour la "bean-managed persistence", le développeur peut faire comme il veut (y compris lire les données d'un fichier)

Persistence

- Deux méthodes ("callback") figurent dans l'interface d'une entity bean pour synchronizer la base de données et les entity bean du container:
  - ejbLoad() est appelée pour initialiser une entity bean connue par sa clé primaire
  - ejbSave() est appelée pour sauvegarder l'état de l'objet dans la BD
- Pour du container-managed, ces méthodes sont générées
- Pour du bean-managed, le développeur doit les coder
- D'autres méthodes seront nécessaires pour les fonctions d'accès aux objets associés

Container-managed persistence

Deux types de technologies:

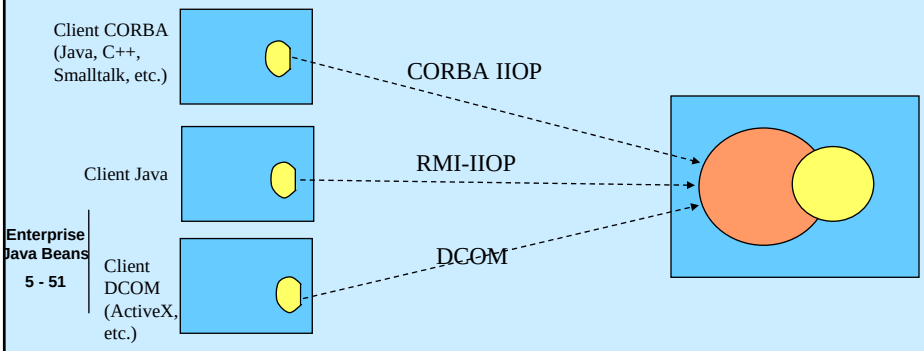
- Bases de données relationnelles:
  - Le développeur spécifie une correspondence entre les classes (entity bean) et les tables d'une base de données relationnelle
  - On fait (trés bien) la correspondence entre attributs et navigation simple (relations un à plusieurs)
  - On fait depuis EJB 2.0 le mapping pour les relations plusieurs à plusieurs
- Bases de données orientées objet
  - Il n'y a quasiment *rien à faire*.

Bean-managed persistence

- La sauvegarde des données est faite grâce à du code que le développeur devra écrire
- Du fait sur mesure, permettant une certaine optimisation (*lazy initialization*)
- De l'ouvrage…

Distribution

- Première version de EJB (1.0) spécifique à Java RMI
- Depuis EJB 1.1, il est en principe, possible d'interfacer un client CORBA avec un serveur EJB, grace au protocole RMI-IIOP, et un client ActiveX grace à DCOM



Nommage (naming)

- Java Naming and Directory Interface: API pour enregistrement et dépistage de clients
- JNDI peut se plugger sur plusieurs services de nommage

```
// obtiens une instance du service de nommage
   Context instanceDeNamingService = new InitialContext();
   // obtiens reference au serveur du serveur de temps
   Object reference = instanceDeNamingService.lookup("ServeurDuServeurDeTemps");
   // faire un cast sécuritaire à DonneLHeureHome
   DonneLHeureHome serveurDuServeurDetemps =
    (DonneLHeureHome)PortableRemoteObject.narrow(reference,quelleheureestil.Donne
    LHeureHome.class);
   // obtiens un donneur d'heure
   DonneLHeure serveurDeTemps = serveurDuServeurDetemps.create();
```

Sécurité

Les serveurs EJB supportent trois types de sécurité:

- Identification: identifier les usagers du système par code usager et mots de passe
- Authorizations: restreindre l'accès aux méthodes des EJBs à certains usagers ou certains groupes d'usagers (*roles*)
- Communication confidentielle: les données transmises sont sécurisées

Authorisations: modèle à base de rôles

- On définit des rôles
- On définit les permissions pour les méthodes par rapport aux rôles
- On associe des usagers ou groupes d'usager à des rôles (via le deploytool)
- On s'identifie au moment de démarrer l'application

Identification de l'usager

```
…
properties.put(Context.SECURITY_PRINCIPAL,nomUsager);
properties.put(Context.SECURITY_CREDENTIALS,motDePasse);
…
Context nommage = new InitialContext(properties);
…
Object reference = nommage.lookup("Ma Bean");
….
```


On peut savoir qui appelle une méthode

```
public abstract interface EJBContext {
   // Methods
   EJBHome getEJBHome();
   EJBLocalHome getEJBLocalHome();
   Properties getEnvironment();
   Identity getCallerIdentity();
   Principal getCallerPrincipal();
   boolean isCallerInRole(Identity p0);
   boolean isCallerInRole(String p0);
   UserTransaction getUserTransaction() throws IllegalStateException;
   void setRollbackOnly() throws IllegalStateException;
   boolean getRollbackOnly() throws IllegalStateException;
}
```

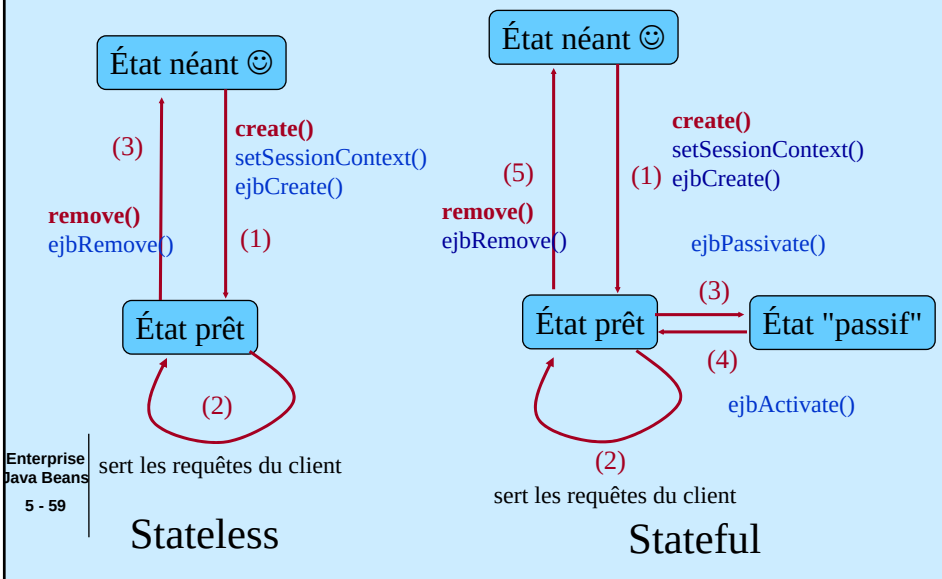

Plan

- **L'architecture J2EE**
- **L'architecture Enterprise Java Beans**
- **Services de base**
- ***Les Sessions Bean***
  - Stateless
  - Stateful
- Les Entity Beans
- Les Message-Driven Beans
- EJB 3.0
- Patrons de conception J2EE

Session beans: rappel

- ***Session beans***: elles représentent des processus d'affaires; sont transientes. Typiquement, elles implantent des *use-cases*. Elles sont spécifiques à l'usager/client. Deux sortes
  - ***Stateless beans***: chaque appel de méthode est indépendant de l'appel suivant
  - ***Stateful beans***: les méthodes appelées ont des effets de bord qui sont utilisés pour les appels suivants. Représentent des transactions complexes

Cycle de vie des session beans



Enterprise Java Beans
5 - 59

Pour implanter une session bean

Il faut définir trois entités:

- *remote interface*: une interface qui représente les fonctionnalités d'affaire de la "bean"
- *home interface*: une interface qui représente un "utilitaire" pour créer, ou chercher des instances de la *home* interface
- *Classe EJB*: classe qui implante la fonctionnalité de la session bean

Enterprise Java Beans
5 - 60

L'interface "remote"

- Doit être une extension de EJBObject
- Chaque méthode doit lancer:
  - RemoteException ou EJBException pour représenter les exceptions d'infrastructure (lien de communication rompue, connexion à des ressources non-disponibles, etc.)
  - Les exceptions spécifiques au domaine d'application, au besoin
- Les arguments et résultats de retour doivent être acceptables pour RMI-IIOP
  - Ils doivent être sérializables
  - Peuvent représenter des références à des objets "distants"

L'interface home

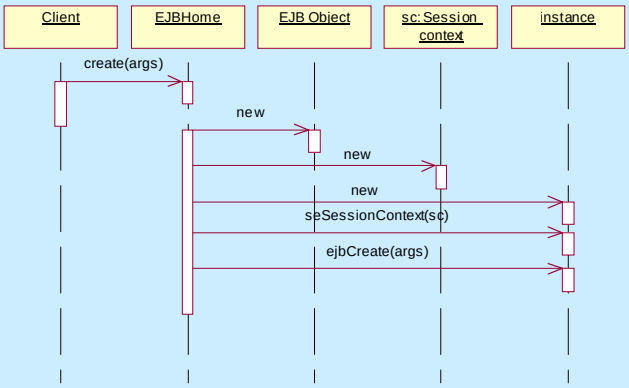
- Cette interface doit hériter de javax.ejb.EJBHome
- Cette interface contient des méthodes pour:
  - Créer des beans (entity beans ou session beans)
  - Appeler d'autre méthodes utilitaires (non appelable directement sur un objet du domaine)
- Doit inclure au moins une méthode "create" (peu importe le nombre d'arguments)
- Peut inclure des méthodes "remove"
- Les mêmes restrictions pour les types de paramètres et de retour des méthodes

La classe "bean"

Cette classe doit:

- Implanter l'interface SessionBean
- Implanter les méthodes incluses dans la *remote* interface
  - Même signature que dans la remote interface
- Implanter les méthodes incluses dans la *home interface*
  - Nom de méthode de la home interface préfixé par ejb. Exemple: si dans home interface la méthode s'appelle "create()", dans la classe Bean, il doit y avoir une méthode correspondente avec le nom ejbCreate, et qui lance les mêmes exceptions
- Doit avoir un constructeur public sans arguments

Création d'une session bean (stateful)



Création d'une session bean (stateless)

- On crée le EJB object suite à la demande du create() du client
- La "création" de la bean instance, et son affectation au EJB object est reportée jusqu'au moment où on a besoin de la bean (invocation d'une méthode)

Interface session bean

- **public void setSessionContext**(SessionContext ctx) **throws** EJBException, RemoteException
  - La bean doit emmagasiner le "contexte" dans une variable d'instance. Ce contexte permettra plus tard de savoir qui a appelé (pour des questions de securité)
- **public void ejbActivate**() **throws** EJBException, RemoteException
  - Le container appelle cette méthode à la fin de la réactivation de la bean (stateful)
- **public void ejbPassivate**() **throws** EJBException, RemoteException
  - Le container informe la bean qu'elle va se faire passiver
- **public void ejbRemove**() **throws** EJBException, RemoteException
  - Le container informe la bean qu'elle va se faire effacer

Interface Session Bean

- Très souvent, on peut laisser ejbPassivate() et ejbActivate() vides
- Dans certains cas, on les utilise pour:
  - reconstruire certaines données non-sérialisées
  - relâcher ou réacquérir les ressources qu'on utilise
- Typiquement, ejbRemove() et ejbPassivate() vont se ressembler
- ejbRemove() sera souvent appelée automatiquement par le container, mais à l'occasion par le client (via remove())

Plan

- **L'architecture J2EE**
- **L'architecture Enterprise Java Beans**
- **Services de base**
- **Les Sessions Bean**
- ***Les Entity Bean***
  - JDBC
  - Persistence gérée par la bean (*bean-managed*)
  - Persistence gérée par le serveur (*container-managed*)
- Les Message-Driven Beans
- EJB 3.0
- Patrons de conception J2EE

Pour implanter une entity bean

Il faut définir:

- *remote interface*: une interface qui représente les fonctionnalités d'affaires de la "bean"
- *home interface*: une interface qui représente un "utilitaire" pour créer, ou chercher des instances de la *home* interface
- *Classe EJB*: classe qui implante la fonctionnalité de la entity bean
- *Classe clé primaire* (optionnelle): une classe qui représente la clé primaire des entity bean. Cette classe peut être String, si on est capable d'identifier uniquement les instances de la bean avec juste une valeur

Remote et home interface

- Mêmes restrictions que pour la session bean au niveau des signatures des méthodes
- La home interface *doit* inclure la signature

```
<interface remote> findByPrimaryKey(<type de clé>
  arg)throws RemoteException, FinderException;
```

- N'est pas obligé d'avoir des méthodes "create(…)"
- La home interface peut inclure d'autres méthodes de fouilles d'entity bean. Les méthodes "finder" doivent:
  - Commencer par le préfix "find"
  - Soit retourner une instance de la bean (interface remote)
  - Soit retourner une *collection* d'instances de la bean
  - Doivent lancer *au moins* les exceptions `RemoteException` et `FinderException`

La bean class

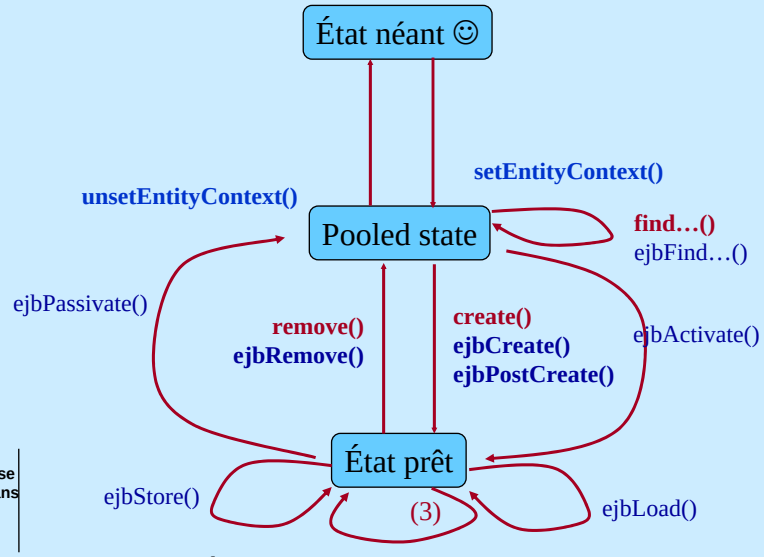
Cette classe doit:

- Implanter l'interface EntityBean
- Implanter les méthodes incluses dans la *remote* interface
  - Même signature que dans la remote interface
- Implanter les méthodes incluses dans la *home interface*
  - Nom de méthode de la home interface préfixé par ejb. Exemple: si dans home interface la méthode s'appelle "create()", dans la classe Bean, il doit y avoir une méthode correspondente avec le nom ejbCreate, et qui lance les mêmes exceptions
- Doit avoir un constructeur public sans arguments
- L'implantation des méthodes "finder" doit retourner une clé primaire ou une collection de clés primaires …

Identité des objets

- Les objets sont identifiés uniquement par:
  - leur base (leur *home*)
  - leur clé primaire
- Pour vérifier l'identité de deux objects, on utilise la méthode isIdentical(EJBObject unObject)
- Il ne faut pas utiliser "==" pour comparer des entity (ou session bean) du coté client

Cycle de vie d'une entity bean



Répondre aux besoins du client

Cycle de vie

- La bean instance est créée par le container avec BeanClass.newInstance(), suivi de setEntityContext(EntityContext ctx)
  - Le contexte contient de l'information sur le container et sur l'objet qui appelle
- Dans l'état pooled, l'instance de bean n'est pas attribuée à un EJBObject
  - Dans ce temps là, elle peut exécuter n'importe quelle méthode finder (méthodes de la home interface).
  - L'instance ne va pas vers un ready state à la fin d'une finder method (même pas findByPrimaryKey(…))

Cycle de vie (suite)

- C'est le container qui sélectionne une bean instance pour l'attribuer à un objet, et donc, la faire passer vers l'état prêt.
- Deux transitions possibles:
  - Par le biais de ejbCreate(…) et ejbPostCreate(…) (en réponse à un appel sur le home de la part du client)
  - Par le biais d'un ejbActivate(), qui est appelée automatiquement par le container quand il a besoin d'une bean instance pour répondre aux besoins d'un client

Cycle de vie (suite)

- A l'état prêt, la bean instance est associée à un objet, et le container peut être amené à invoquer ejbLoad et ejbStore plusieurs fois pour synchroniser l'état de la bean instance
- A l'état prêt, tout ce que l'on sait c'est que le contexte de la bean instance peut nous donner accès à la clé primaire.
- La bean instance revient à l'état pooled de deux façons:
  - le container peut appeler ejbPassivate() (gestion de ressources)
  - le client appelle ejbRemove()
- unsetEntityContext() enlèves une bean instance de l'état pooled (la prépare pour la garbage collection)

L'interface EntityBean (suite)

- public void setEntityContext(EntityContext ctx) throws EJBException, RemoteException
  - Il faut sauvegarder le contexte dans une variable d'instance pour pouvoir y référer dans d'autres méthodes (e.g. pour savoir qui appelle). On peut réserver des ressources ici
- public void unsetEntityContext() throws EJBException, RemoteException
  - Le dernier soupir de la bean instance. On y lâche les ressources acquises dans setEntityContext(…)

L'interface EntityBean (suite)

- public <cle> ejbCreate(<args>) throws CreateException, RemoteException, <d'autres exceptions>
  - La bean instance existe, mais n'est pas attribuée à un objet encore
  - Cette méthode valide les arguments (y compris les possibilités de conflict de clé primaire), initialise la bean instance, et insère l'objet dans la BD
  - Retourne la clé primaire
- public void ejbPostCreate(<args>) throws CreateException, RemoteException, <d'autres exceptions>
  - La bean instance existe, mais et est attribuée à un objet
  - Si on demandait la clé primaire de l'objet (à partir du contexte), on la trouverait
  - Sera très souvent vide
  - Peut servir pour notifier une autre bean de la création de celle ci, en appelant getEJBObject() sur le contexte pour avoir la référence à l'objet courant pour "se passer" comme argument

L'interface EntityBean (suite)

- public void ejbActivate() throws EJBException, RemoteException
  - Cette méthode est appelée quand le container sélectionne la entity bean instance pour l'attribuer à un objet
  - La clé primaire est disponible
  - On peut l'utiliser pour acquérir de nouvelles ressources
  - Ce n'est pas ici qu'on synchronise la bean instance avec la BD (ejbLoad(…) s'en occupe, et elle sera appelée par le container)
- public void ejbPassivate() throws EJBException, RemoteException
  - Cette méthode est appelée quand le container décide de dissocier la entity bean instance d'un objet
  - On peut l'utiliser pour relâcher les ressources détenues
  - Ce n'est pas ici qu'on synchronize la bean instance avec la BD (ejbStore(…) s'en occupe, et elle sera appelée par le container)

L'interface EntityBean (fin)

- public void ejbLoad() throws EJBException, RemoteException
  - appelée par le container pour synchroniser l'état de la bean instance avec la BD
  - la clé primaire est disponible
  - la clé primaire est la seule garantie de l'identité de l'objet
- public void ejbStore() throws EJBException, RemoteException
  - appelée par le container pour sauvegarder l'état de la bean instance dans la BD

Les finders

- Les finders sont exécutés par des bean instance qui sont dans l'état pooled.
- On ne peut pas supposer que l'instance qui exécute un findByPrimaryKey(…) sera la même que le container choisira d'attribuer à l'objet en question
  - Pas la peine de rien initialiser dans cette méthode
- La home interface retourne des objets (remote interface) ou des collection d'objet), alors que dans la bean entity class, on retourne la clé primaire (ou une collection de clés primaires)
- Il faut laisser le choix au container de choisir les bean instance qu'il veut!
- Considérez les comme des méthodes statiques!

Plan

- **L'architecture J2EE**
- **L'architecture Enterprise Java Beans**
- **Services de base**
- **Les Sessions Bean**
- **Les Entity Bean**
- ***Les Message-Driven Beans***
- EJB 3.0
- Patrons de conception J2EE

Message-driven beans

- Utilisation
- JMS
- MDB

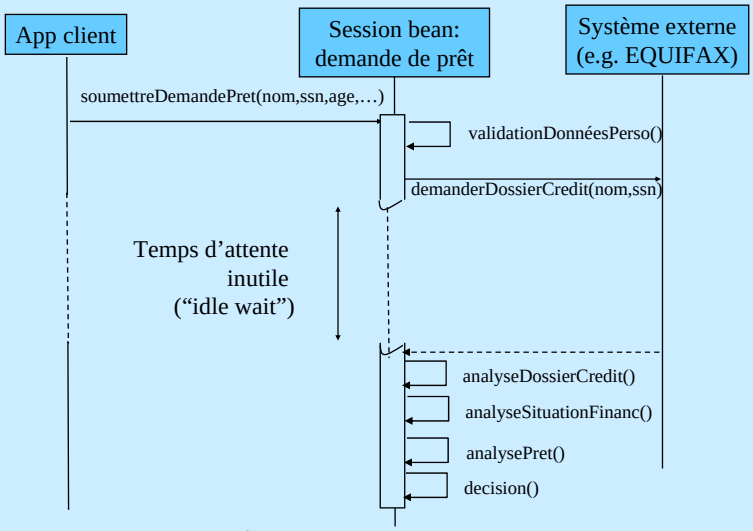
Scénario (1/5)

- Considérez l'exemple d'une application de demande de prêts. Les étapes de traitement sont:
  1. Validation de l'information personnelle
     - Vérifier que: 1) le nom est une chaine de caractères, 2) la date de naissance est valide et raisonnable, etc.
  2. Analyse du dossier de crédit
     - Obtenir rapport de crédit d'une agence de crédit externe
     - L'analyser pour une recommandation positive, négative, ou neutre
  3. Analyse des informations financières
     - Revenus, avoirs, dettes
  4. Analyse des données du prêt-même
     - Montant, raison, terme de remboursement
  5. Décision
     - Approbation vs rejet
     - Termes du prêt

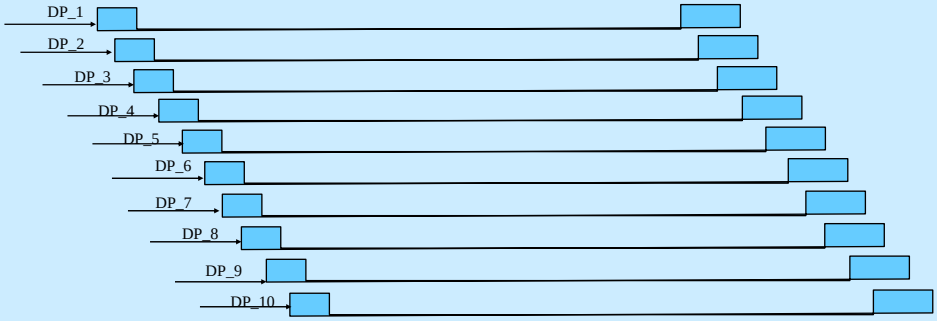
Scénario (2/5)

- Les tâches 1, 3, 4, et 5 sont quasi-instantanées
- La tâche 2 implique l'accès à des ressources externes et peut prendre beaucoup de temps
  - Latence du réseau
  - Système distant incontrollable
- L'analyse du dossier de crédit résulte *souvent* en une recommandation positive

Scénarion (3/5): Solution 1



Scénario (4/5): profils d'activité



Enterprise Java Beans
5 - 87

- Exemple:
  - 300 requêtes (demandes de prêt) à la seconde
  - Une moyenne de 5 secondes pour avoir le dossier de crédit
  - => un minimum de 1500 sessions actives en même temps

Scénario (5/5): solution 2

- Point de vue processus d'affaires: deux phases:
  - Phase 1
    1. Validation de l'information personnelle
    2. Demande du dossier crédit, de façon *asynchrone*
    3. Analyse des informations financières
    4. Analyse des données du prêt-même
    5. Décision *préliminaire*
  - Phase 2: à la *réception* du dossier de crédit
    - Analyse du dossier de crédit
    - Décision *finale*
- Point de vue du traitement
  - Demander le dossier de crédit par un message non-bloquant
  - Entamer la phase 2 lors de la réception d'un message contenant le dossier d'un crédit

Enterprise Java Beans
5 - 88

JMS: Java Messaging Service

- MOM: Message-Oriented Middleware
  - Diverses implantations par des constructeurs différents
- JMS (Java Messaging Service): Une API standard permettant d'invoquer divers MOMs
  - Similaire au rôle joué par JDBC dans le domaine des BD

JMS (1/6)

- Principes
  - Composante A envoie un message à une composante B de façon asynchrone
    - L'envoi du message ne bloque pas A en attendant que B reçoive le message ou le traite
    - B peut être occupé à autre chose au moment de l'envoi du message par A: elle ira chercher le message quand elle se sera libérée
  - La livraison des messages est garantie:
    - Les messages seront reçus dans le bon ordre
      - On utilise des files (queues) de message pour garantir l'ordre de livraison
    - Chaque message sera reçu une fois
      - Les files (queues) sont régulièrement sauvegardées en mémoire secondaire permettant de relancer le système en cas de panne.

JMS (2/6)

- Styles d'envoi de messages
  - Style point à point
    - Les expéditeurs placent les messages sur une file de messages
    - Le(s) récepteur(s) viennent chercher les messages à l'autre bout de la file
      - Chaque message sera consommé *une* fois par l'un des récepteurs
  - Style publish & subscribe:
    - Les messages soumis / envoyés sont associés à des *thèmes* (« topics »)
    - Différents clients peuvent s'abonner à ces thèmes, et ils recevront *tous* chaque message correspondant à leur thème
      - Chaque message sera consommé par un client *juste une fois*, mais le même message pourra être consommé par plusieurs clients

JMS (3/6)

- Concepts de base (modèle de programmation)
  - Clients:
    - Applications Java qui envoient ou reçoivent des messages
  - Messages
    - Patentes contenant des données ou évènements échangés
  - Destinations
    - Des objets java qui représentent le « médium » de communication
      - Dans une communication PAP, une destination correspond à une file d'attente (queue)
      - Dans une communication P&S, une destination correspond à un thème (*topic*)
  - Fournisseurs de services de messagerie (*JMS Provider*)
    - L'implantation *spécifique* du MOM sous-jacent (IBM MQSeries, etc.)

JMS (4/6)

- L'API
  - ConnectionFactory: un objet pour créer une connexion vers un fournisseur de service JMS
    - Le fournisseur est identifié par un nom JNDI

```
ConnectionFactory maFabrique = (ConnectionFactory)namingContext.lookup(nomJNDI);
```

  - Connection: une connexion vers un fournisseur de services JMS

```
Connection maConnexion = maFabrique.createConnection();
```

  - Session: offre un contexte transactionnel pour l'envoi et la réception de message

```
Session maSession =
   maConnexion.createSession(false,Session.AUTO_ACKNOWLEDGE);
```


JMS (5/6)

- L'API (suite)
  - Un objet **Session** permet de créer des **MessageProducer**'s et des **Message**'s

```
MessageProducer prod= session.createProducer(destination);
TextMessage message = session.createTextMessage();
Message.setText(«Bonjour »);
Prod.sendMessage(message);
```

  - Un objet **Session** permet de créer des **MessageConsumer**'s

```
MessageConsumer consommateur= session.createConsumer(destination);
```

  - Deux façons de recevoir les messages:
    - Par invocation explicite à « receive (…) », qui est un appel bloquant

```
Message monMessage = consommateur.receive(TIME_OUT);
```

    - En enregistrant un « message listener » qui implante la méthode « onMessage(Message mess) »

```
monListener = new MessageListener() { public void onMessage(Message mess) {…}};
consommateur.setMessageListener(monListener);
```

JMS (6/6)

- **Message**: composé de:
  - Une entête (« message header »)
    - Comprenant la destination, expiration, priorité, priorité, identificateur, timestamp, type, re-livré, etc.)
  - Des propriétés (facultatives)
    - Des paires <clé, valeur>
  - Un corps (facultatif)
- Types de messages:
  - **TextMessage**: corps est un String
  - **ObjectMessage**: corps est un objet Java serializable
  - **StreamMessage**: corps est un « stream » de valeur Java primities
  - Etc.

Message-Driven Beans

- Un Message-Driven Bean (MDB) est un EJB dont le but est de consommer des messages JMS
- Un MDB est associé à une destination (*topic* ou *queue*)
- Implante l'interface MessageListener
  - Implante la méthode onMessage(Message mess)
- Un MDB est:
  - Stateless
  - Pooled
  - Single-threaded
- La méthode onMessage(Message mess) peut-être marquée comme transactionnelle ou non

Message-Driven Bean

- Un MDB n'implante pas une interface métier:
  - La méthode « onMessage(…) » doit décortiquer le message reçu et trouver le traitement à appliquer
- Un MDB ne *retourne pas* de résultats
  - Pour ce faire, trois possibilités:
    - Utiliser une file de message pour le retour des résultats
      - Une file temporaire
        - Plus légère, mais les messages ne sont pas persistés
      - Une file permanente
        - Plus fiable et tolérant aux fautes, mais d'autres clients peuvent écouter la réponse
    - Utiliser un mécanisme dans JMS Request/Response
      - Est bloquant et non-transactionnel

Plan

- **L'architecture J2EE**
- **L'architecture Enterprise Java Beans**
- **Services de base**
- **Les Sessions Bean**
- **Les Entity Bean**
- **Les Message-Driven Beans**
- **EJB 3.0**
- ***Patrons de conception J2EE***

Considérations de conception

- Les appels distants coûtent chers
- Ne "beanez" pas toutes vos classes:
  - Dans une aggrégation, ne beanez que l'objet principal
  - La sauvegarde en BD s'assurera de tout sauvergarder
  - Ce qui n'est accessible que indirectement peut rester sur le serveur
- Faut –il manipuler des objets ou des identificateurs
- Granularité des méthodes des sessions bean
- Co-localisation des beans

