Intégration d'applications

Java EE - API

Pascal Bleuyard

Pascal.Bleuyard@fr.michelin.com

12 octobre 2011 / ISIMA

Plan du cours

JNDI : Présentation

Accronyme pour Java Naming and Directory Interface.

Interface unique pour utiliser différents services de nommages ou d'annuaires.

API standard pour l'accès à ces services.

Implémentation par pilotes.

Possède un rôle particulier dans les architectures applicatives développées en Java EE : JDBC, EJB, JMS, ...

Centralisation facilite l'administration des données et leur accès.

JNDI : Services de nommage

Un service de nommage permet d'associer un nom à un objet ou à une référence sur un objet.

Plusieurs types de service de nommage :

- Système de fichiers
- DNS (Domain Name System) : nom de domaine internet et adresse IP
- LDAP (Lightweight Directory Access Protocol) : annuaire
- *etc.*

Le nom associé à un objet respecte une convention de nommage particulière à chaque type de service (“/”, “\”, “.”)

JNDI : Annuaires

Service de nommage qui possède en plus une représentation hiérarchique des objets qu'il contient et un mécanisme de recherche.

Possibilité d'associer des attributs à chaque objet

N'est pas une base de données (lecture et modification, ensembliste) !

Exemples :

- Pages blanches : département, ville, nom/prenom
- Pages jaunes : activités, ville, nom

JNDI : Utilisation

Pour définir une connexion, au moins deux éléments :

- Fabrique du contexte racine : assure le dialogue avec le service utilisé en utilisant le protocole adéquat
- Url du service à utiliser

JNDI : Contexte

Binding : Associations nom/objet.
Contexte : Ensemble d'associations nom/objet.

Utilisé lors de l'accès à un élément contenu dans le service.
Deux types de contexte :

- Contexte racine
- Sous contexte (relatif à un contexte racine)

Exemples :

- C :\windows
- test.com

JNDI : Mise en oeuvre

Cinq packages :

- `javax.naming` : Utilisation d'un service nommage
- `javax.naming.directory` : Etend `javax.naming` pour l'utilisation des services de type annuaire
- `javax.naming.event` : Gestion des événements lors d'un accès à un service
- `javax.naming.ldap` : Etend `javax.naming.directory`pour LDAP
- `javax.naming.spi` : Développement de pilotes

JNDI : Interfaces Name

Encapsule un nom en permettant de faire abstraction des conventions de nommage utilisées par le service

Deux implémentations :

- CompositeName (“/”)
- CompoundName (hiérarchie, règles dépendant de l’implémentation)

JNDI : Interface Context et classe IntitialContext

Représente un ensemble de correspondances nom/objet d'un service de nommage.

Propose des méthodes pour interroger et mettre à jour ces correspondances.

La classe javax.Naming.InitialContext`qui implémente l'interface`Context‘ encapsule le point d'entrée dans le service de nommage.

Toutes les opérations sont relatives à ce contexte racine.

Obtention par une **fabrique** (implémentation dépendant du pilote).

JNDI : Utilisation d'un service de nommage

Plusieurs opérations à partir du contexte racine :

- bind : associer un objet avec un nom
- rebind : modifier une association
- unbind : supprimer une association
- lookup : obtenir un objet à partir de son nom
- list : obtenir une liste des associations

Deux versions surchargées :

- Objet de type Name
- Chaine de caractères

JNDI : Obtention d'un objet

```
import javax.naming.*;
...
  public String getValeur() throws NamingException {
    Context context = new InitialContext();
    return (String) context.lookup("/config/monApp");
  }
```

Exemples :

- Configuration d'une application
- Données sensibles

JNDI : Stockage d'un objet

La méthode bind() permet d'associer un objet à un nom.

```
import javax.naming.*;
...
  public void createName() throws NamingException {
    Context context = new InitialContext();
    context.bind("/config/monApp", "valeur");
  }
```

Plusieurs intérêts (annuaire) :

- Objets accessibles par plusieurs applications
- Objets entre plusieurs exécutions d'une même application

Plan du cours

JNDI

JDBC

JPA

EJB

JSP et servlets

JSF et facelets

JDBC : Présentation

Acronyme pour Java DataBase Connectivity.

API standard pour l"accès aux bases de données, indépendamment du fournisseur.

JDBC : Mise en oeuvre

2 packages :

- `java.sql` : coeur
- `javax.sql` : extension Java EE, transactions distribuées, pools de connexions, connexion avec un objet DataSource, **points de sauvegarde**

Nécessite un **pilote** spécifique à la base de données accédée.

JDBC : Types de pilotes

4 types de pilote JDBC :

- Type 1 : Pont JDBC-ODBC
- Type 2 : API native
- Type 3 : Protocole réseau
- Type 4 : Protocol natif

Fichiers .jar dont le chemin doit être ajouté au classpath.

JDBC : Pilote de type 1, Pont JDBC-ODBC

Open Database Connectivity (registre de sources/bases de données).

Fonctionne très bien sous Windows (pilotes ODBC pour la quasi totalité des bases de données)

Possède plusieurs inconvénients :

- Multiplication du nombre de couches :
  - architecture complexe (transparent pour le développeur)
  - détérioration des performances (un peu)
- ODBC et son pilote doivent être installé sur tous les postes clients lors du déploiement
- Partie native (ODBC et son pilote) :
- Application dépendante d'une plateforme (moins portable)

JDBC : Pilote de type 2, API native

Ecrit en Java.

Appelle l'API native de la base de données.

Convertit les instructions JDBC via un pilote natif sur le client.

Nécessite l'utilisation de code natif sur le client.

JDNC : Pilote de type 3, Protocole réseau

Ecrit en Java utilisant un serveur intermédiaire (middleware, connexion par socket).

Utilise un protocole réseau propriétaire spécifique à une base de données.

Serveur dédié reçoit les messages par ce protocole et dialogue directement avec la base de données.

Peut facilement être utilisé par une architecture multicouches mais nécessite que le serveur intermédiaire, en général sur la machine contenant le serveur web.

JDBC : Pilote de type 4, Protocole natif

Ecrit en Java, utilisant le protocole natif de la base de données.

Appelle directement le SGBD par le réseau.

Fourni par l'éditeur de la base de données.

JDBC : Question

Quel est le type de pilote le plus performant ?

JDBC : Classes principales

4 classes importantes :

- DriverManager (et **DataSource**) : charge et configure le driver de la base de données
- Connection : réalise la connexion et l'authentification à la base de données
- Statement (et PreparedStatement) : contient la requête SQL et la transmet à la base de données
- ResultSet (et **RowSet**) : permet de parcourir les informations retournées par la base de données dans le cas d'une sélection de données

Chacune dépend de l'instanciation d'un objet de la précédente classe.

JDBC : Utilisation

1. Charger le pilote (Class.forName, inutile depuis JDBC 4.0)
2. Ouvrir une connexion (Connection)
3. Créer des instructions SQL (Statement, PreparedStatement, CallableStatement)
4. Exécuter ces instructions
   - interroger (executeQuery)
   - modifier (executeUpdate)
   - autre (execute)
5. Récupérer les résultats (ResultSet)
6. Fermer la connexion (close)

JDBC : Chargement du pilote

```
Class.forName("jdbc.DriverXXX");
```

Charge le pilote et créer une instance de cette classe.

Configurable dans un fichier de déploiement depuis JDBC 4.0.

static forName() de la classe Class peut lever l'exception java.lang.ClassNotFoundException.

JDBC : Connexion

```
String DBurl = "jdbc:odbc:testDB";
Connexion con = DriverManager.getConnection(DBurl);
```

Instancie un objet Connection en lui précisant sous forme d'URL (protocole :sous_protocole :nom) la base à accéder.

getConnection() peut lever une exception de la classe java.sql.SQLException.

```
Connection con = DriverManager.getConnection(url,
    "login", "passwd");
```

JDBC : Accès à la base de données

Une fois la connexion établie, il est possible d'exécuter des requêtes SQL.

Objets utilisés pour obtenir des informations :

- DatabaseMetaData : informations à propos de la base de données (nom des tables, index, version, *etc.*)
- ResultSet : résultat d'une requête et information sur une table, accès enregistrement par enregistrement.
- ResultSetMetaData : informations sur les colonnes (nom et type) d'un ResultSet

JDBC : Exécution de requêtes SQL

Interrogations SQL exécutées avec les méthodes d'un objet Statement obtenu à partir d'un objet Connection.
Résultat renvoyé dans un objet de la classe ResultSet

```
Statement stmt = con.createStatement();
ResultSet résultats = null;
String requete = "SELECT * FROM client";

try {
   Statement stmt = con.createStatement();
   résultats = stmt.executeQuery(requete);
} catch (SQLException e) {
   //traitement de l'exception
}
```

JDBC : Exécution de requêtes SQL

- Interrogation (SELECT) : executeQuery(“requete SQL”)
  - lève une SQLException si la requête ne contient pas SELECT
- Mise à jour : executeUpdate(“requete SQL”)
  - retourne le nombre d’enregistrements mis à jour
  - retourne 0 pour traitement DDL (Data Definition Langage) comme la création d’un table

JDBC : Exécution de requêtes SQL

```
...

//insertion d'un enregistrement dans la table client

requete = "INSERT INTO client VALUES (3,'client3','nom3')";
try {
   Statement stmt = con.createStatement();
   int nbMaj = stmt.executeUpdate(requete);
   affiche("nb mise a jour = "+nbMaj);
} catch (SQLException e) {
   e.printStackTrace();
}

...
```

JDBC : Classe ResultSet

Représente une abstraction d'une table qui se compose de plusieurs enregistrements constitués de colonnes (**curseur**)

- Administration : close (automatique avec Statement), getMetaData
- Navigation :
  - isFirst, isLast, isBeforeFirst, isAfterLast
  - first, last, next, previous, beforeFirst, afterLast
  - getRow, absolute(pos), relative(pos)
- Contenu de la colonne (Int, Float, Date) :
  - getXXX(int)
  - getXXX(String)

JDBC : Utilisation d'un objet PreparedStatement

Encapsule une requête précompilée. Adapté pour une exécution répétée d'une même requête avec des paramètres différents.

Hérite de l'interface Statement.

Obtenu en utilisant la méthode preparedStatement() d'un objet de type Connection.

Attend en paramètre une chaîne de caractères contenant la requête SQL. Paramètres représentés par un caractère " ?".

Ensemble de méthode setXXX(numero, valeur) (type primitif ou String, Date, Object, *etc.*) pour fournir les valeurs de chaque paramètre.

JDBC : Utilisation d'un objet PreparedStatement

Exemple : jdbc2.java

JDBC : Utilisation des transactions

JDBC : Procédures stockées

JDBC : Traitement des erreurs

JDBC : Mise à jour et sélection sur une table

Exemple complet : jdbc1.java

Résultat

```
connexion a la base de données
creation enregistrement
nb mise a jour = 1
creation et execution de la requête
parcours des données retournées
1.0 client 1 prenom 1
2.0 client 2 prenom 2
3.0 client 3 client 4
fin du programme
```

JDBC : Classe DataSource

Fournit une meilleure alternative à la classe DriverManager pour assurer la connexion à une base de données.

Représente une connexion physique à une base de données

Obligatoire pour pouvoir utiliser un pool de connexion et les transactions distribuées.

Doit être enregistré dans un service de nommage une fois créée (instance par JNDI).

Paramètres (taille, nombres minimum et maximum de connexions, *etc.*) configurables par l'API ou le serveur d'applications.

JDBC : Classe DataSource

Exemple :

```
...
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/appDB");
Connection con = ds.getConnection("admin", "mpadmin");
...
```

JDBC : Pools de connexion

Connexion très couteuse en ressources.

Permet de maintenir un ensemble de connexions **réutilisables** vers une base de données.

Limite le nombre de créations et d'améliore les performances.

JDBC : API RowSet

`javax.sql.Rowset` définie des objets pour manipuler les données d'une base.

Nécessaire d'avoir une implémentation (Oracle, tiers, personnells)

2 types d'implémentations :

- Connecté à la base de données durant toute sa durée de vie
- Déconnecté de la base après avoir récupéré des données (modifications reportées à une reconnexion ultérieure)

Gestion des événements par RowSetListener.

JDBC : API RowSet

Java 5 fourni en standard une implémentation de référence des interfaces (nom de l'interface suivi de Impl) :

- JDBCRowSet : manipulation des données en mode connecté
- CachedRowSet : manipulation des données d'une source en mode déconnecté (stockées dans l'objet)
- WebRowSet : lecture et l'écriture des données au format XML (hérite de CachedRowSet)
- FilteredRowSet : filtres (hérite de WebRowSet)
- JoinRowSet : jointures (hérite de WebRowSet) avec des objets implémentant l'interface Joinable

JDBC : Bonnes pratiques

- Fermer les ressources inutilisées dès que possible (Connection, Statement, ResultSet) : pool de connexions
- Limiter le nombre de données retournées par une requête SQL uniquement à celles utiles : PreparedStatement, CallableStatment, **BatchUpdates**, **cache**
- Toujours assurer un traitement des warnings et des exceptions

Plan du cours

Plan du cours

EJB - Qu’est-ce que c’est ?

EJB (Entreprise Java Bean) :

- Composant serveur
- Implémente la logique métier
- Accessible par les clients
- S’exécute dans un conteneur d’EJB

EJB - Principaux avantages

- Simplification du développement
  - *Utilise les services (système) fournis par le serveur*
- **Réduction des coûts**
- Architecture multi-couches
  - *Client pour la présentation, bean pour la logique métier*
- Composants réutilisables et portables

EJB - Conteneur

- Environnement “contrôlé” pour l’exécution des composants
- Services fournis par le conteneur
  - Accès distant aux Beans
  - Cycle de vie des Beans
  - Injection de dépendances
  - Intercepteurs
  - Transactions
  - Sécurité
  - Monté en charge (bassin de connexions)
  - *etc.*

EJB - Conteneur

2 types :

- Autonome
  - *Apache OpenEJB*
- Intégré à un serveur d'application JEE
  - *GlassFish*, *JBoss*, *WebSphere*, *etc.*

EJB - Technologie

- Déjà. . . 10 ans d'existence !
  - EJB 1.0 (1998, proposition)
  - EJB 1.1 (1999)
  - EJB 2.0 (2002)
  - EJB 2.1 (2003)
  - EJB 3.0 (2006)
  - EJB 3.1 (2009)
- Objectif initial
  - "Fournir un environnement contrôlé pour aider au développement d'applications d'entreprise"

EJB - Evolution des spécifications

- EJB 1.0 (proposition)
  - Session Beans (stateless/stateful)
  - Entity Beans
  - Interface Remote uniquement
- EJB 1.1 (J2EE 1.2)
  - Descripteur de déploiement XML
- EJB 2.0 (J2EE 1.3)
  - Message-Driven Beans
  - Entity 2.x reposant sur EJB QL
  - Interface Local (pour les appels dans la même JVM)
- EJB 2.1 (J2EE 1.4)
  - EJB Timer Service
  - EJB Web Service Endpoints (JAX-RPC)
  - Amélioration du langage EJB QL

EJB - Evolution des spécifications

- EJB 3.0 (Java EE 5) : **Etape majeure !**
  - POJO et POJI (plus d'interface Home)
  - Annotations (descripteur optionnel)
  - JPA pour les Entity Beans
  - Injection de dépendances
  - Configuration par exeptions
- EJB 3.1 (Java EE 6)
  - Interfaces locales optionnelles
  - Singleton Beans
  - EJB Lite
  - Simplification du paquetage
  - Invocation d'un EJB hors du conteneur

EJB - EJB 2

Puissant. . .

Mais développement des composants lourd et complexe :

- Plusieurs interfaces à mettre en oeuvre
  - Interfaces EJBHome, EJBObject, EntrepriseBean
  - Interfaces JNDI
- Descripteur de déploiement XML

API conçue pour les conteneurs (services), pas pour les développeurs !

EJB - EJB 3

- Objectif principal : **simplification** !
- 4 grands principes :
  - Simplifier le déploiement
    - *Descripteur XML optionnel*
  - Simplifier les interactions entre EJB et serveurs d'applications
    - *Moins d'interfaces à mettre en oeuvre*
  - Configuration par exceptions
    - *Règles par défaut*
  - Gestion de la persistance
    - *POJO (Plain Old Java Object) au lieu des Entity Beans*

EJB - Annotations (rappels)

- Annotations
  - Introduites dans Java EE 5
  - Contrôlent le comportement et le déploiement d'une application
  - Spécifient les classes Beans (`@Stateless`, `@Stateful`, *etc.*)
  - Alternatives aux descripteurs de déploiement
- Descripteurs de déploiement
  - Alternatives aux annotations **ou...**
  - Surcharge des annotations

EJB - Types de composants

- Session
  - Fournit des services au client (session interactive)
  - Implémente la logique métier
  - Non persistant
- Message
  - Permet la communication par messages asynchrones
  - Se comporte comme un Listener
- Singleton
  - Une seule instance dans un conteneur pour une application
- *Entity* : **obsolète**
  - Remplacé par les entités JPA
  - Géré par le pilote de persistance (EclipseLink), pas par le conteneur d'EJB

EJB - Session Beans

- Composant Java s'exécutant dans un contenur d'EJB
- Gère la logique métier de différents types d'applications

Exemple : Gestion d'un panier dans une application de commerce électronique

EJB - Gestion de l'état d'un Bean Session

Etat : valeurs des variables d'instance d'un Bean

2 modes de gestion :

- Sans état : `@Stateless`
- Avec état : `@Stateful`

EJB - Gestion de l'état d'un Bean Session

- Stateless
  - Etat non conservé entre 2 appels
  - Point d'entrée unique pour la couche métier
  - Partagé (par plusieurs clients), supporte donc la montée en charge
  - Instances du même Bean identiques
- Stateful
  - Variables d'instance représentent l'état d'un client unique (**en conversation**)
  - Etat conservé pendant la durée de la session Bean-client
  - **Non persistant !**

EJB - Stateless vs Stateful

- Stateful Session Beans appropriés quand :
  - L'état du Bean ne contient pas de données pour un client spécifique
  - Le Bean réalise une tâche générique pour tous les clients (convertisseur, mail, *etc.*)
- Stateful Session Beans appropriés quand :
  - L'état du Bean représente une interaction entre le Bean et un client spécifique
  - Le Bean doit conserver des informations sur le client entre les invocations de méthodes
  - Le Bean offre une vue simplifiée au client et sert de médiateur avec un autre composant
  - Le Bean gère le *workflow* entre plusieurs EJB

EJB - Interface vs implémentation

- Interface (métier)
  - POJI
  - Déclaration des méthodes métier
  - Contrat entre un client et un EJB
    - *Vue d'un Bean pour le client*
    - *Accès uniquement aux méthodes déclarées*
  - Utilisés par le conteneur pour générés *stubs* (substituts) et *proxies* (**facades**)
  - Peut être générée par le conteneur lors du déploiement (non recommandé)
- Session Beans
  - POJO avec annotations
  - Implémente une ou plusieurs interfaces
  - Invisible au client
    - *Aspects du Bean (implémentation, déploiement, etc.) masqués*
    - *Interagit uniquement avec le conteneur*

EJB - Interface vs implémentation

```
@Stateless
public class HelloWorldBean {
  public String saluer(String nom)
  {
    return "Bonjour "+nom;
  }
}
```

EJB - Mode d'accès

- Local (par défaut)
  - S'exécute sur la même JVM que l'EJB
  - Peut-être un composant Web ou un autre EJB
  - Localisation de l'EJB non tranparente pour un client local
- *Remote* (Distant)
  - Peut s'exécuter sur une machine et une JVM différentes de l'EJB
  - Peut-être un composant Web, une application client ou un autre EJB
  - Localisation de l'EJB tranparente pour un client distant
- Service Web (non présenté ici)

Annotations : `@Remote`, `@Local`, `@WebService`

EJB - Mode d'accès

Annotations @Local et @Remote peuvent être utilisées directement sur l'EJB.

Préférable de définir une interface par mode d'accès et d'utiliser l'annotation adéquate sur chacune des interfaces.

```
import javax.ejb.Local;

@Local
public interface CalculLocal {
  public long additionner(int valeur1, int valeur2);
}
```

EJB - Mode d'accès

```
import javax.ejb.Remote;

@Remote
public interface CalculRemote {
  public long additionner(int valeur1, int valeur2);
}
```

EJB - Mode d'accès

```
import javax.ejb.*;

@Stateless
public class CalculBean implements CalculLocal,
                                   CalculRemote {
  public long additionner(int valeur1, int valeur2) {
    return valeur1 + valeur2;
  }
}
```

EJB - Conventions de nommage

Pas de règles imposées. Exemple :

- Nom du bean : CalculEJB
- Nom de la classe métier : CalculBean
- Interface locale : CalculLocal
- Interface distante : CalculRemote

EJB - Annotations Stateless et Stateful

- `@javax.ejb.Stateless` ou `@javax.ejb.Stateful`
- Attributs optionnels
  - name
  - mappedName
    - *nom JNDI*
    - *non portable (dépend du produit)*
    - *utile pour les Beans distants uniquement*
  - description

EJB - Injection de dépendances

- Fonctions pour “injecter” différents types de ressourcesd dans un Session Bean
  - Autres Beans, sources de données, files de messages, *etc.*
- Exploités par le conteneur EJB
  - Après qu’un EJB est instancié à l’intérieur du conteneur Java EE. . .
  - Mais avant qu’il soit fourni au client
- Spécification
  - Annotations : `@EJB`, `@Ressource`
  - Descripteurs de déploiement XML

EJB - Injection de dépendances

```
/* annotation of instance variables */
@Resource
DataSource myDb;

/* annotation of the setter method */
@Resource
public void setMyDb(DataSource myDb) {
  this.myDb = myDb;
}
```

EJB - Cycle de vie des Beans

Différents états (timeout) :

- Stateless
  - Inexistant (pas encore instancié)
  - Prêt (en conversation, disponible en mémoire et prêt à exécuter des méthodes)
- Stateful
  - Inexistant
  - Prêt
  - **Passivé** (déplacé de la mémoire vers le disque)

EJB - Cycle de vie des Beans

Différentes opérations :

- Création
- Suppression
- Activation
- *Passivation* (sérialisation)

EJB - Cycle de vie des Beans

Différentes annotations (**callbacks**, contrôle fin) :

- @PostConstruct : invoqué après que l'instance soit créée et que les dépendances soient injectées
- @PostActivate : invoqué après que l'instance de l'EJB ne soit déssérialisée du disque
- @Remove : invoqué avant que l'EJB ne soit retiré du conteneur
- @PreDestroy : invoqué avant que l'instance de l'EJB ne soit supprimée
- @PrePassivate : invoqué avant de l'instance de l'EJB ne soit sérialisée sur disque

Exemples : Ouverture et fermeture de la connexion à une BDD

EJB - Cycle de vie des Beans

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.*;
...

@Stateless
@Remote(CustomerDAO.class)
public class CustomerDAOBean implements CustomerDAO {
...
  @PostConstruct
  public void makeConnection() {
    ...
  }
}
```

EJB - Cycle de vie des Beans

- Création/destruction dépendant du serveur
- Réalisés en général par le conteneur
  - Création quand :
    - *L'application est déployée*
    - *Le serveur est démarré*
    - *A la première requête*
  - Destruction quand :
    - *L'application n'est plus déployée*
    - *Le serveur est arrêté*
- En général, un serveur gère un bassin d'EJB Stateless

EJB - Question

Représenter le cycle de vie des Beans Session Stateless et Stateful.

EJB - Intercepteurs

- Permet d'intercepter l'invocation de méthodes métier
- Peut être défini pour les Beans Session et Message
- Exemples d'utilisations
  - Logs supplémentaires
  - Analyse des performances
- Implémentation
  - Annotation `@AroundInvoke` pour une méthode particulière
  - Classe `@Interceptor`
    - *méthodes invoquées avant l'appel d'une méthode métier*

EJB - Intercepteurs

Exemple : Interceptor.java et TestInterceptor.java

```
...
Creation du bean
: com.jmd.test.domaine.ejb.PersonneFacade@1697e2a
...
Invocation de la
methode : public java.util.List
com.jmd.test.domaine.ejb.PersonneFacade.findAll()
...
```

EJB - Intercepteurs

3 niveaux :

- Par défaut
  - Tous les appels de tous les Beans de l'unité de déploiement
  - Définition uniquement par le descripteur de déploiement XML
- Classe
  - Tous les appels de tous les Beans d'une classe spécifique
  - Annotations ou XML
- Méthode
  - Tous les appels d'une méthode spécifique
  - Annotations ou XML

EJB - Intercepteurs

Niveau Classe

```
@Stateless
@Interceptors ({TracingInterceptor.class})
public class EmailSystemBean {
   ...
}
```

EJB - Intercepteurs

Niveau méthode

```
@Interceptors({AccountsConfirmInterceptor.class})
public void sendBookingConfirmationMessage(long orderId) {
   ...
}
```

EJB - Récupération d'une référence à un Bean

JNDI lookup

```
public void runTest() throws Exception {
  InitialContext ctx = new InitialContext();
  SimpleBean bean = (SimpleBean)
    ctx.lookup("ejb/SimpleBeanJNDI");
  String result = bean.sayHello("TOTO");
  System.out.println(result);
}
```

EJB - Récupération d'une référence à un Bean

Injection de dépendances

```
@EJB(mappedName="ejb/SimpleBeanJNDI")
SimpleBean simpleBean;
```

EJB - Cycle de vie des Beans vu du client

- Obtenir une référence sur l'interface métier d'un bean (par lookup JNDI ou injection de dépendances)
- Utiliser le Bean
  - Invocation de méthodes métier sur l'interface
  - Passage de la référence comme paramètre
- Détruire le Bean
  - Stateless : transparente pour le client (géré par le conteneur)
  - Stateful : invocation d'une méthode `@Remove`

EJB - Difficultés

- Création des fichiers
  - Interfaces métier
  - Classe EJB
  - Classe Helper (autres classes requises par la classe EJB, par exemple les exceptions)
- Déploiement
  - Création du fichier de déploiement
  - Paquetage des EJB dans un fichier JAR
  - Paquetage de l'application dans un fichier EAR (optionnel)
  - Déploiement du Bean sur un serveur d'applications

EJB - Déploiement

EJB - Transactions

EJB - Messages

EJB - Singleton

Plan du cours

Rappel

Java EE permet le développement d'applications distribuées :

- applications d'entreprise (EJB)
- applications web

Technologies web Java EE

- Servlets
- JSP + JSTL
- MVC “Model 1” : servlets + JSP
- MVC “Model 2” : un seule servlet + JSP
- Java Server Faces : facelets

Servlets

- Technique Java pour générer des pages web dynamiquement en fonction des actions du client
  - Requête, traitements, résultat
  - Similaire aux *CGI*, à PHP…
  - S'exécute sur le serveur Web
- Avantages :
  - Tout est géré par la JVM : **portabilité**.
  - Utilisation de processus légers (threads) : **concurrence**.
  - Langage compilé (plus efficace, source "caché").

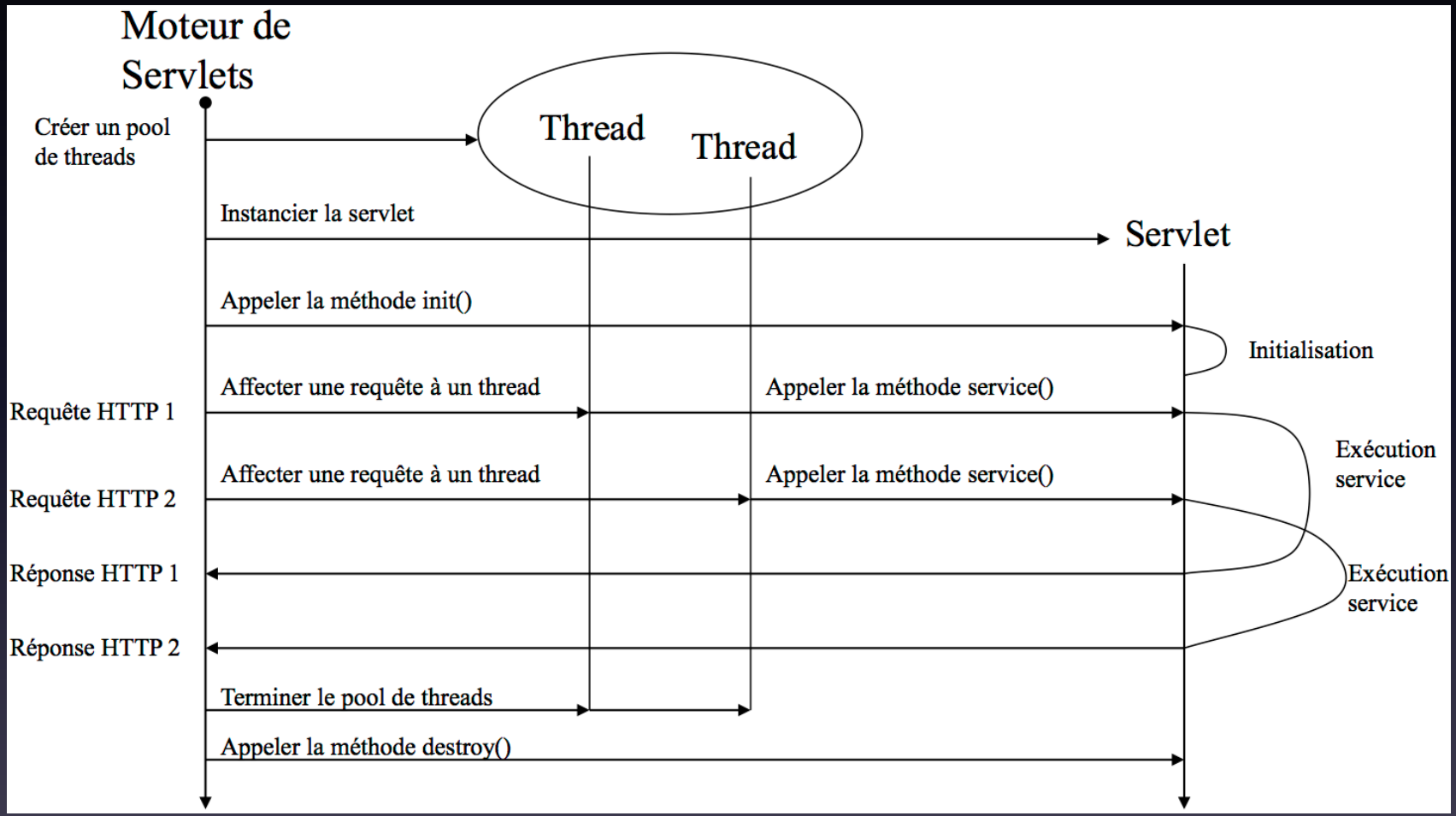
Servlets

- Exécution sur le serveur web : **conteneur web** + **serveur HTTP**
- Pas d'interface graphique
  - Juste de la génération de pages HTML, le navigateur fait le reste
- Privilèges de sécurité différents :
  - On fait confiance au serveur, pas au client
  - Possibilité d'établir des connexions (RMI, . . .)
  - Possibilité de faire des appels systèmes/BD/. . .
  - Possibilité de manipuler des ressources locales (fichiers, . . .)

Développement

- En pratique, on doit implémenter l'interface javax.servlet.Servlet
  - Soit directement : init(), service(), destroy(), getServletConfig(), getServletInfo()
  - Soit en dérivant d'une classe qui l'implémente
    - **HttpServlet**
    - GenericServlet : n'importe quel protocole requête/réponse
- Nécessite un serveur web :
  - Implémentation dépendant du serveur
  - Chargement d'une servlet (.class) : au lancement ou à la première invocation
  - Gestion de son cycle de vie : initialisée **une seule fois**
  - Passage des requêtes et des réponses

Gestion des servlets



Exécution d'une servlet HTTP

Il suffit de saisir une URL qui désigne la servlet dans un navigateur. Le serveur :

1. Reçoit la requête HTTP
2. Instancie la servlet si nécessaire
3. Créé un objet qui représente la requête http et objet qui contiendra la réponse et les envoie à la servlet
    - La servlet créé dynamiquement la réponse sous forme de page html transmise dans l'objet réponse (requête, ressources)
4. Récupère l'objet réponse et envoie la page html au client

Protocole HTTP

Modèle client/serveur :

- navigateur web : envoie une requête au serveur
- serveur web : attend en permanence les requêtes sur un port particulier (80 par défaut)
- connexion uniquement durant l'échange de la requête et de la réponse

Protocole HTTP

Requête composée de trois parties :

- commande
- section en-tête
- corps

Exemple : ‘GET /index.html HTTP/1.0

Commandes : **GET**, **POST**, **HEAD**, OPTIONS, PUT, DELETE, TRACE et CONNECT

Protocole HTTP

Réponse composée des trois mêmes parties :

- statuts
- en-tête normalisé
- corps dépendant de la requête

Codes importants :

- 200 : traitement correct de la requête
- 204 : traitement correct mais aucun contenu (page courante affichée)
- 404 : la ressource demande n'est pas trouvée
- 500 : erreur interne du serveur

Classe HttpServlet

- Encapsule le protocole HTTP
- Hérite de GenericServlet
- Implémente l'interface Servlet
- Création de pages HTML dynamiques et/ou traitements (bdd, image, ...)
- Méthodes à redéfinir en fonction du type de requête (doGet(), doPost(), ...)
- Requête encapsulée dans un objet qui implémente l'interface HttpServletRequest (données, informations sur le client) : getParameter()
- Réponse encapsulée dans un objet qui implémente l'interface HttpServletResponse : getWriter()

Classe HttpServlet

- doHead() : pour les requêtes http de type HEAD
- doPut() : pour les requêtes http de type PUT
- doDelete() : pour les requêtes http de type DELETE
- doOptions() : pour les requêtes http de type OPTIONS
- doTrace() : pour les requêtes http de type TRACE
- init(), destroy() et getServletInfo() : interface Servlet

Servlets et sessions - Problématique

- HTTP = protocole non connecté
  - Chaque requête est indépendante des précédentes.
  - Différent de Telnet, Ftp, ...
- Besoin de stocker des informations entre des requêtes successives
  - Equivalent à être en mode connecté

Solutions avec les servlets

- Utilisation des cookies
- Réécriture d'URL
  - ajouter des informations d'identification dans l'URL
  - Problème : taille, caractères autorisés, sécurité
- Utilisation des champs de formulaire “hidden”
  - Mêmes problèmes
- Utilisation de HttpSession ou de ServletContext

Sauvegarde d'information

- Deux moyens de sauvegarder de l'information :
  - Session et contexte d'une servlet
- Session = session HTTP
  - HTTP est sans état, on maintient une session en faisant passer des informations d'un manière ou d'une autre (cookie, URL rewriting, . . .)
- Contexte = configuration de la servlet
  - Via l'objet ServletContext.
    - Ou indirectement avec ServletConfig (cf fonction init).
  - Méthodes pour obtenir la configuration, échanger des données. . .

ServletContext

- Obtenu par la méthode getServletContext()
- On peut y placer des attributs
  - void setAttribute(String name, Object value)
    - Associe un objet quelconque à une chaîne de caractère.
  - Enumeration getAttributeNames()
    - Liste les noms d'attributs disponibles dans le contexte.
  - Object getAttribute(String name)
    - Obtenir un attribut.
  - void removeAttribute(String name)
    - Supprimer un attribut.
- Ou avoir des informations générales
  - String getServerInfo() /int getMajorVersion() /. . .

L’objet session

- Simple avec l’API des servlets
  - objet HttpSession
- Principe :
  - Un objet “session” peut être associé à chaque requête
  - Il va servir de “container” pour des informations persistantes
  - Durée de vie limitée et réglable

Une application Web

Une application web = un espace virtuel

- Contient html, images, servlets, jsp. . .
- Un fichier web.xml :
  - Fichier de description du déploiement.
  - Décrit ce qui est disponible et comment y accéder :
    - Nom de la servlet.
    - Nom de la classe associée.
    - URL correspondante.
    - Paramètres d'initialisation : ServletConfig
    - Paramètres dontextuels : communs à l'application
    - Listeners
  - À placer dans le répertoire /WEB-INF

Fichier web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet>
        <servlet-name>Hello</servlet-name>
        <servlet-class>Hello</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Hello</servlet-name>
        <url-pattern>/Hello.html</url-pattern>
    </servlet-mapping>
</web-app>
```

JSP

- Objectif : simplifier l'écriture de servlets :
  - Servlets = "HTML inside Java"
  - JSP = "Java inside HTML"
- Avertissements :
  - JSP = servlet
  - On est vite limité sans servlet
    - Sauf en utilisant des frameworks qui reposent dessus

Servlet – Hello World

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Hello extends HttpServlet {
protected void doGet(HttpServletRequest req,
                     HttpServletResponse res)
throws ServletException, IOException {
    PrintWriter out = res.getWriter();
    out.println("<html>");
    out.println("<body>");
    out.println("Hello World");
    out.println("</body>");
    out.println("</html>");
} }
```

JSP – Hello World

```
<html>
<body>
    <% out.println("Hello World"); %>
</body>
</html>
```

JSP – Hello World

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <display-name>test</display-name>
    <description>test</description>
</web-app>
```

Conversion JSP en Servlet

- Le code JSP est :
  - Converti en une servlet Java
    - Héritage de la classe javax.servlet.jsp.HttpJspBase
    - Implémente la méthode _jspService
  - Compilé de manière classique :
    - Le premier accès à la page jsp est donc potentiellement plus lent.
- Le code source de la servlet intermédiaire n'est pas conservée en général
  - Dépend du moteur de servlet/JSP
  - Sous Glassfish par défaut elle n'est pas conservé

Conversion JSP en Servlet

- Pour voir la source sous Glassfish :
  - Modifier le fichier web.xml
  - Aller chercher le fichier :
    - En général dans . . . \domain1\generated\jsp\
    - Ou ailleurs

Conversion JSP en Servlet

```
<servlet>
<servlet-name>jsp</servlet-name>
<servlet-class>org.apache.jasper.servlet.JspServlet
</servlet-class>
<init-param>
    <param-name>keepgenerated</param-name>
    <param-value>true</param-value>
</init-param>
<load-on-startup>3</load-on-startup>
</servlet>
```

Conversion JSP en Servlet

Exemple : ServletGenerated.java

Conversion JSP en Servlet

- 3 méthodes essentielles :
  - jspInit() appelée après le premier chargement
  - _jspService() appelée à chaque appel à la JSP
  - jspDestroy() appelée lors du déchargement
- Équivalent aux méthodes de Servlet :
  - init()
  - service()
  - destroy()

Les objets implicites

- Définis dans la fonction _jspService et donc utilisables dans du code jsp.
- Il suffit de regarder la servlet générée :
  - request et response : cf servlets.
  - out : utilisé pour envoyer la réponse au client.
  - session : objet commun aux pages gérant les sessions.
  - application : contient le contexte de la servlet.
  - config : permet d'accéder à la configuration de la servlet.
  - page : this.
  - context : informations sur l'environnement du serveur.

La classe PageContext

- Permet de manipuler une variable quelque soit le contexte
  - Object getAttribute(String name,int scope)
  - void setAttribute(String name, Object o, int scope)
  - void removeAttribute(String name,int scope)
  - Scopes possibles : PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE, APPLICATION_SCOPE
- Recherche dans tous les contextes :
  - Object getAttribute(String name)
  - void setAttribute(String name, Object o)
  - void removeAttribute(String name)

JSP ou Servlets ?

- Servlets = “HTML inside Java”
- JSP = “Java inside HTML”
- Le code JSP est compilé :
  - Plus rapide à l’exécution qu’un langage interprété (Php ou autre)
    - Lié aussi à la qualité du code et au serveur.
  - La compilation peut prendre du temps lors de la première requête sur la ressource.
  - Les requêtes ultérieures utilisent le fichier compilé si le jsp n’a pas été modifié.

Les balises JSP

- Commentaires : <%–
- Scriptlet : <%
  - Insertion du code Java : out.println(“Hello”) ;
- Expression : <%=
  - Afficher une valeur
- Déclaration : <% !
  - Déclarer des variables ou fonctions
- Directive : <%@
  - Modifier la traduction du code JSP

Commentaires JSP

- <%– ... –%>
  - Pas inclus dans la réponse envoyée au navigateur
  - A utiliser pour commenter le JSP, pas le HTML

Commentaires JSP

```
<html>
<body>
<!-- commentaire visible en HTML -->
<%-- commentaire JSP invisible en HTML --%>
</body>
</html>
```

Scriplets JSP

- <jsp :scriplet ... /> ou <% ... %>
  - Inclusion de code Java dans la JSP
  - Permet de faire du code plus ou moins lisible

Scriplets JSP

```
<html>
<body>
<%
for(int i=0;i<5;i++) {
%>
Hello<br>
<%
}
%>
</body>
</html>
```

Scriplets JSP

```
<html>
<body>
<%
for(int i=0;i<5;i++) {
  out.println("Hello<br>");
}
%>
</body>
</html>
```

Expressions JSP

- <jsp :expression ... /> ou <%= ... %>
  - Affichage d'une valeur ou expression
  - Pas de ; à la fin d'une expression

Expressions JSP

```
<%@ page import="java.util.*" %>
<html>
<body>
<%
for (Enumeration e = request.getParameterNames () ;
     e.hasMoreElements() ;) {
  String name= (String) e.nextElement();
%>
<%= name %> : <%= request.getParameter(name) %>
<%
}
%>
</body>
</html>
```

Déclarations JSP

- <jsp :declaration ... /> ou <%! ... %>
  - Déclaration de variables d'instance ou de méthodes.
  - Le code est inséré tel quel au début de la classe générée.

Déclarations JSP

```
<html>
<body>
<%!
String Chaine = "bonjour";
public String bold(String s) {return "<b>" + s + "</b>";}
%>
<%= bold(Chaine) %>
</body>
</html>
```

Un exemple

```
<html>
<body>
<%-- variable locale --%>
<% int n = 0; %>
Page chargée <%= ++n %> fois

<%-- variable d'instance --%>
<%! int m = 0; %>
Page chargée <%= ++m %> fois
</body>
</html>
```

Directives JSP

- <jsp :directive ... /> ou <%@ ... %>
- include :
  - inclusion d'un fichier dans la JSP au moment de la compilation ou de l'exécution (include/forward)
  - <%@ include file = "test.html" %>
  - <%@ include page = "test.jsp" %>
- page :
  - attributs de la page (type de contenu, librairies java...).
  - <%@ page import="java.util.Date" contentType="application/vnd.ms-excel" isErrorPage="false" %>

Directives JSP – nouveaux tags

- Utilisation d'autres tag du genre <jsp :. . . />
- Possibilité de définir des librairies de tags :
  - Chemin et préfixe utilisé pour une librairie de tag.
    - <%@ taglib prefix="p" uri="WEB-INF/library.tld" %>

Inclusion dynamique

- <jsp :include page= “/test.jsp”/> .
  - Inclusion dynamique, effectuée au moment de la requête.
  - Possibilité de transmettre des infos à la page incluse.
    - <jsp :include page= “/test.jsp”>
<jsp :param name= “maVariable” value= “laValeur” />
</jsp :include>‘
  - Récupération du paramètre :
    - <%out.println(request.getParameter(maVariable)) ;%>

Redirection dynamique

- <jsp :forward page= “/test.jsp”/> .
  - redirection dynamique, effectuée au moment de la requête.
  - Possibilité de transmettre des infos à la page incluse.
    - <jsp :forward page= “/test.jsp”>
      <jsp :param name= “maVariable” value= “laValeur” />
      </jsp :forward>
  - Récupération du paramètre :
    - <% out.println(request.getParameter(maVariable)) ; %>

Directives de page

```
<%@ page language="scriptingLanguage"
extends="className" //étend une classe
import="importList" //packages à importer
session="true|false" // utilisation d'une session ou pas
buffer="none|sizekb"
autoFlush="true|false"
isThreadSafe="true|false"
info="info_text"
errorPage="error_url" // fixe une page en cas d'erreur
isErrorPage="true|false"
contentType="ctinfo"
pageEncoding="peinfo"
isELIgnored="true|false"
>
```

Gestion des erreurs

- Plusieurs types d'erreurs :
  - Erreur au moment de la conversion de la jsp en servlet
    - Aucune solution, mis à part corriger la jsp
  - Erreur au moment de l'exécution (exception)
    - Récupérable facilement via une page d'erreur
- Dans la déclaration d'une page on peut déclarer :
  - Vers quelle page faire suivre les erreurs
  - Si la page courante est une page d'erreur

Gestion des erreurs

```
<%-- fichier index.jsp --%>

<%@ page errorPage="/error.jsp" %>
<html>
<body>

<%
int i=0/0;
%>

</body>
</html>
```

Gestion des erreurs

```
<%-- fichier error.jsp --%>

<%@ page
    isErrorPage="true"
    import="java.io.*"
    contentType="text/plain"
%>

<%
exception.printStackTrace(
  new PrintWriter(out));
%>
```

Création de beans

- Bean = classe annexe utilisée dans le code JSP
  - Objectif : séparer la présentation du modèle
- <jsp :useBean id= “. . . ” class=“. . . ” scope= “. . . ” />
  - Instanciation de la classe et ajout dans le “contexte”
    - id : identifiant du bean.
    - class : classe associée au bean.
    - scope : portée du bean.
  - Si le bean existe déjà dans le contexte choisi alors il est récupéré et utilisable

Utilisation de beans

- jsp :useBean :
  - instanciationdelaclasse/récupérationdel'objet
- jsp :getProperty property="xxx"
  - Obtenirlavaleurd'unattributdel'objet
  - L'objetdoitcontenirungetterassocié :getxxx()
- jsp :setProperty property="xxx" value="valeur"
  - Modifierlavaleurd'unattributdel'objet
  - L'objetdoitcontenirunsetterassocié :setxxx()
- Toute méthode définie dans le bean accessible de manière naturelle
  - <%=monObjet.getPrice()%>

setProperty

- Modification directe :
  - <jsp :setProperty name=“myBean” property=“test” value= “toto” />
  - <jsp :setProperty name= “myBean” property=“test” value=“<%=myBean.getTest()+1%>” />

setProperty

- Modification via des paramètres reçus par GET ou POST :
  - De manière générale, via l'objet request :
    - <jsp :setProperty name="myBean" property="name" value= "<%=request.getParameter("nom")%>" />
  - Plus simplement en connaissant le nom du parametre et celui de l'attribut :
    - <jsp :setProperty name="myBean" property="name" param="nom"/>
  - Encore plus simplement si le parametre et l'attribut sont identiques :
    - <jsp :setProperty name="myBean" property="name"/>
    - "équivalent à" myBean.name = request.getParameter("name")

Utilisation de Beans

Exemple : useBean.jsp

```
package fr.ntw;
public class myBean {
    private String name;
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
}
```

Portée des beans

- Page :
  - Visible pour une page JSP uniquement
  - Similaire à une variable locale dans la méthode doGet
- Request :
  - Comme page mais le bean est utilisable en cas d'include ou de forward
  - Similaire à un request.setAttribute() avant le forward

Portée des beans

- Session :
  - Visible durant toute la vie d'une session (donc pour un utilisateur donné)
    - Utilisable pour un panier par exemple
  - Similaire à un ajout dans la session
- Application :
  - Accessible dans toute l'application (jusqu'à son redémarrage) par toutes les pages et tous les utilisateurs. A tester avec deux navigateurs.
    - Utilisable pour une connexion à une BD par exemple
  - Similaire à un ajout dans le contexte

Portée des objets

Exemple : ObjectScope.jsp

Portée des objets

Premier chargement de la servlet

```
Avant :
page : null
request : null
session : null
application : null

Après :
page : null
request : r
session : s
application : a
```

Portée des objets

Après rechargement dans le même navigateur

```
Avant :
page : null
request : null
session : s
application : a

Après :
page : null
request : r
session : s
application : a
```

Portée des objets

Après rechargement dans un autre navigateur

```
Avant :
page : null
request : null
session : null
application : a

Après :
page : null
request : r
session : s
application : a
```

Gestion de formulaires

- Comme avec une servlet :
  - Utilisation de request.getParameter()
- Utilisation de Bean :
  - Création d'une classe correspondant aux différents champs du formulaire avec les méthodes get et set pour chaque champ.
  - Cf. exemple plus tôt.

Gestion (rapide) de formulaires

Exemple : Form.jsp

Autres solutions

- Vu ici :
  - JSP + Beans
- Autres solutions :
  - Librairies de tag (à définir ou existantes)
  - JSF + facelets
  - …

Plan du cours