Java EE

Cours 5

JavaBeans et Scope

Cours de 2e année ingénieur

JavaBeans

JavaBeans?

Les **JavaBeans** sont des classes Java(POJO) qui suivent certaines conventions:

– Doivent avoir un **constructeur vide** (zéro paramètre)

=> On peut satisfaire cette contrainte soit en définissant explicitement un tel constructeur, soit en ne spécifiant aucun constructeur

– Ne doivent **pas avoir d'attributs publics**

=> Une bonne pratique réutilisable par ailleurs…

– La valeur des attributs doit être manipulée à travers **des accesseurs** (getters et setters)

=> Si une classe possède une méthode getTitle qui retourne un String, on dit que **le bean possède une propriété** String nommée title

=> Les propriétés Boolean utilisent isXXX à la place de getXXX

Réf. sur les beans : `http://docs.oracle.com/javase/tutorial/javabeans/`

Pourquoi faut-il utiliser des accesseurs?

- Dans un bean, on ne peut pas avoir de champs publics
- Donc, il faut remplacer

```
public double speed;
```

Par

```
private double speed;

public double getSpeed() {
   return(speed);
}

public void setSpeed(double newSpeed) {
   speed=newSpeed;
}
```

Pourquoi faut-il faire cela pour **tout** code Java?

Pourquoi faut-il utiliser des accesseurs?

1) On peut imposer des contraintes sur les données

```
public void setSpeed(double newSpeed){
        if(newSpeed < 0){
                SendErrorMessage(...);
                newSpeed = Math.abs(newSpeed);
        }
        speed = newSpeed;
}
```

Pourquoi faut-il utiliser des accesseurs?

2) On peut changer de représentation interne sans changer d'interface

```
// newSpeed and internal representation are in km units
public void setSpeed(double newSpeed){
        speed = newSpeed;
}

// Now using miles units for internal representation (but
// keeping km units for newSpeed)
public void setSpeed(double newSpeed){
        speed = convertKMToMiles(newSpeed);
}
```

Pourquoi faut-il utiliser des accesseurs?

3) On peut rajouter du code annexe

```
public double setSpeed(double newSpeed){
        speed = newSpeed;
        updateSpeedometerDisplay();
}
```

Comment utiliser des beans ?

Une fois la classe créée, les beans sont des objets Java sur lesquels on peut faire ces actions :

- instanciation d'un nouveau bean
- récupération de la valeur d'une propriété du bean
- affectation/modification de la valeur d'une propriété du bean

Pour faire cela dans une Servlet, ça ne pose pas de problème particulier, le bean est traité comme un objet Java standard.

Cependant, dans une JSP, l'utilisation des balise de scriptlets n'est pas très « propre », on utilisera plutôt l'une des manière suivantes :

- **balises JSP (JSP 1.2)** : <jsp:useBean>, <jsp:getProperty>, <jsp:setProperty>
- **utilisation des EL (JSP 2.0)** (si le serveur le supporte) : ${...}

=> Dans ce cours nous verrons la deuxième approche, la première n'étant aujourd'hui utilisée que pour des raisons de compatibilités avec la version du serveur installé

Implémentation du MVC

1. Définir les beans pour représenter les données
2. Utiliser un servlet pour gérer les requêtes
   - Le servlet lit les paramètres de requêtes, vérifie les données manquantes ou malformées, appelle le code métier, etc.
3. Créer les beans
   - Le servlet invoque le code métier (spécifique à l'application) ou accède à une base de données pour obtenir les résultats. Les résultats sont dans les beans définis à l'étape 1
4. Placer le bean dans le bon scope (défini dans la suite du cours)
   - Le servlet appelle **setAttribute** sur la requête, la session, ou le servlet context pour garder une référence sur les beans qui représentent le résultat de la requête

Implémentation du MVC avec RequestDispatcher

5. Transmettre la requête à la JSP (forward)
   - Le servlet détermine quelle JSP est appropriée à la situation et utilise la méthode *forward* du *RequestDispatcher* pour transférer le contrôle à la JSP
6. Extraire les données des beans
   - JSP 1.2: la JSP accède aux beans avec jsp:useBean et un scope correspondant au choix de l'étape 4. La JSP utilise ensuite jsp:getProperty pour afficher les propriétés des beans
   - JSP 2.0: la JSP utilise ${nameFromServlet.property} pour afficher les propriétés des beans

   => La JSP ne crée pas ou ne modifie pas les beans: c'est la vue du MVC!

Exemple MVC : Modèle

```
package model;

public class PersonneBean {
        private String nom;
        private String prenom;

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

- Les Beans doivent être déployés dans le même répertoire que les autres classes Java:

  `WEB-INF/classes/nomPackage`

- Les Beans doivent **toujours** être dans des packages

Exemple MVC : Contrôleur

- En MVC, **les beans sont crées/modifiés par le contrôleur**, en fonction de la requête du client
=> surtout pas par la vue !

```
import model.PersonneBean;

@WebServlet("/MyServlet")
public class MyServlet extends HttpServlet {
        protected void doGet(...)
        {
                String urlVue = "vue.jsp";                          // URL de la vue à appeler
                String nom = request.getParameter("...");           // Récupération du champs nom
                                                                    //(par exemple via un formulaire)
                String prenom = request.getParameter("...");        // Récupération du champs prenom
                if(nom == null || nom.trim().equals("")) {
                        // Erreur : nom non saisi
                }
                else if(prenom == null || prenom.trim().equals("")) {
                        // Erreur : prenom non saisi
                }
                else // Cas sans erreur : on traite la requête, et crée les beans nécessaires
                {
                        PersonneBean bean = new PersonneBean(); // Instanciation d'un bean
                                                                // de type PersonneBean
                        bean.setNom(nom);               // Affectation de la propriété nom
                        bean.setPrenom(prenom);         // Affectation de la propriété prenom
                        request.setAttribute("myBean", bean);   // Attacher ce bean au
                                                                // scope de requête
                }
                // Forward à la vue:
                request.getRequestDispatcher(urlVue).forward(request, response);
        }
}
```

Exemple MVC : Vue

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%><html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Ma Vue</title>
</head>
<body>
        Salut ${myBean.prenom} ${myBean.nom} !
</body>
</html>
```

- En MVC, les beans sont uniquement consultés par la vue, pour faire son affichage.
- L'instruction ${...} est une EL (Expression Language), nous reviendrons dessus au prochain TP

=> L'EL permet de récupérer l'information sur un bean (s'il a été mis préalablement dans un scope par une servlet avec un *setAttribute*):

```
${nomBean.property}
```

- *nomBean* est celui défini lors du setAttribute fait par la servlet
- *property* est le nom de la propriété que l'on veut accéder (attention aux normes d'écriture Java => respectez la casse)

Les Scopes

Un Scope ?

- Un scope peut être vu comme un **conteneur de beans** stocké du côté du **serveur** (le client ne peut avoir aucune connaissance sur ces beans)
- Il existe 4 types de Scopes, que l'on distingue par la **durée de vie des beans** qui y sont stockés :
  - **Requête :** ce scope est créé à chaque requête du client, les beans qu'il contient sont détruit lorsque le serveur à envoyé la réponse au client
  - **Session :** ce scope est créé automatiquement par le conteneur JEE pour chaque client qui se connecte au serveur. Les beans qu'il contient ne sont visible que par le client détenant la session, et sont détruit lorsque la session du client se termine (géré par un timeout du côté du serveur)
  - **Application :** ce scope est créé automatiquement au lancement du projet JEE sur le serveur. Les beans qu'il contient sont partagés par tous les clients et ne sont détruit que lors de l'arrêt/rechargement du projet JEE sur le serveur.
  - **Page :** ce scope restreint la durée de vie des beans aux requêtes POST effectuées sur une page donnée, dès qu'on en sort, les beans sont détruit.

    => Ce scope n'est quasiment pas utilisé

Partage de données sur requête

- Servlet

```
ValueObject value = new ValueObject(...);
request.setAttribute("key", value);
RequestDispatcher dispatcher =
  request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

- JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="request" />
<jsp:getProperty name="key" property="someProperty" />
```

- JSP 2.0 (utilisation des EL)

```
${key.someProperty}
```

Partage de données sur requête: exemple simple

- Servlet

```
Customer myCustomer=
  new Customer(request.getParameter("customerID"));
request.setAttribute("customer", myCustomer);
RequestDispatcher dispatcher=
   request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

- JSP 1.2

```
<jsp:useBean id="customer" type="somePackage.Customer"
             scope="request" />
<jsp:getProperty name="customer" property="firstName"/>
```

- JSP 2.0

```
${customer.firstName}
```

Partage de données sur session

- Servlet

```
ValueObject value = new ValueObject(...);
HttpSession session=request.getSession();
session.setAttribute("key", value);
RequestDispatcher dispatcher=
  request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

- JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="session" />
<jsp:getProperty name="key" property="someProperty" />
```

- JSP 2.0

```
${key.someProperty}
```

Partage de données sur session: variation

- Redirection vers une page au lieu d'un transfert
  - Utiliser response.sendRedirect à la place de RequestDispatcher.forward
- Différences avec sendRedirect:
  - L'utilisateur voit l'URL de la JSP (l'utilisateur ne voit que l'URL du servlet avec RequestDispatcher.forward)
  - Deux aller-retour pour le client (au lieu d'un avec forward)
- Avantage du sendRedirect
  - L'utilisateur peut accéder à la JSP séparément
    - Possibilité de mettre la JSP en marque-page
- Désavantages du sendRedirect
  - Deux aller-retour, c'est plus coûteux
  - Puisque l'utilisateur peut accéder àla JSP sans passer par le servlet d'abord, les beans qui contiennent les données peuvent ne pas être disponibles (si stockés dans la requête, au deuxième appel ils n'existent plus)
    - Il faut du code en plus pour détecter cette situation

Partage de données sur application (Rare)

- Servlet

```
synchronized(this){
  ValueObjectvalue= new ValueObject(...);
  getServletContext().setAttribute("key", value);
  RequestDispatcher dispatcher=
    request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
  dispatcher.forward(request, response);
}
```

- JSP 1.2

```
<jsp:useBean id="     key" type="somePackage.ValueObject"
                scope="application" />
  <jsp:getProperty name="key" property="someProperty" />
```

- JSP 2.0

```
${key.someProperty}
```

URL relatives dans les JSP

- Problème:
  - Transférer avec un *request dispatcher* est transparent pour le client. L'URL *originale* est la seule URL que le navigateur connaît.
- Pourquoi est-ce important?
  - Que fera un navigateur avec les balises suivantes?

```
<img src="foo.gif" …>
<link rel="stylesheet"
        href="my-styles.css"
        type="text/css">
 <a href="bar.jsp">…</a>
```

  - Le navigateur traîte les adresses relatives comme des adresses relatives à l'URL du *servlet*