



# Java pour le développement d'applications Web : Java EE

## JSF : Java Server Faces

Mickaël BARON - 2007 (Rév. Mai 2011)  
<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>

# Licence

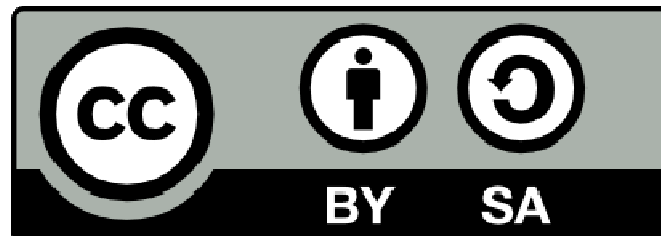
---

## Creative Commons

*Contrat Paternité*

*Partage des Conditions Initiales à l'Identique*

2.0 France



*<http://creativecommons.org/licenses/by-sa/2.0/fr>*

# Déroulement du cours JSF

## ➤ Pédagogie du cours

- Illustration avec de nombreux exemples
- Des bulles d'aide tout au long du cours



**Ceci est une alerte**



**Ceci est une astuce**

## ➤ Pré-requis

- Connaissance de Java
- Connaissance des fondements du WEB (HTML, HTTP)
- Connaissance des technologies « fondatrices » de Java EE (Servlet, JSP, balises personnalisées)

## ➤ Structuration du cours

- Démarrage par les bases de JSF (Bean et Navigation)
- Pour chaque nouveau concept introduit, le présenter en détail
- Terminer par des concepts plus complexes

## ➤ Remerciements

- Developpez.com [*djo.mos* pour ses commentaires]

## Introduction : qu'est-ce-que c'est ...

---

- **Java Server Faces** est un framework de développement d'applications Web en Java permettant de respecter le modèle d'architecture **MVC** et basé sur des composants côté présentation
- Java Server Faces permet
  - une séparation de la couche présentation des autres couches (MVC)
  - un mapping entre l'HTML et l'objet
  - un ensemble de composants riches et réutilisables
  - une liaison simple entre les actions côté client de l'utilisateur (event listener) et le code Java côté serveur
  - Création de nouveaux composants graphiques
  - JSF peut être utilisé pour générer autre chose que du HTML (XUL, XML, WML, ...)

## Introduction : meilleur que Struts, les avantages ...

- Possibilité de créer de nouveaux composants
  - JSF permet de combiner plusieurs composants pour aboutir à un composant plus complexe
- Support différentes technologies d'affichage
  - JSF ne se limite pas à l'HTML (XUL, Flash, ...)
- Accès aux Beans par leurs noms en utilisant les Expressions Language
  - Pour Struts nécessité de mémoriser le nom du formulaire
- Simplification des définitions des contrôleurs et des Beans
- Simplification du fichier de configuration
- L'orientation composants graphiques permet à JSF d'être utilisé de manière plus simple dans les environnements de développement (voir Sun Java Studio Creator 2)

# Introduction : meilleur que Struts ... et les inconvénients

---

- Maturité de Struts par rapport à JSF
  - Struts est largement utilisé en industrie alors que JSF est encore un framework jeune
- Confusion des noms
  - Avec Struts les noms des formulaires (form.jsp) et les contrôleurs (form.do) avaient différentes URL's
  - JSF utilisent les mêmes noms pour la vue et le contrôleur
- La validation dans JSF n'est pas aussi fournie que Struts
  - Possibilité de générer du JavaScript avec Struts pas JSF
  - Possibilité de vérifier la syntaxe d'une carte de crédit, utilisation d'expressions régulières, ...
- Moins de documentation et de ressources
  - Nous allons tenter d'y remédier ...

# Introduction : une spec et plusieurs implémentations

- JSF comme la plupart des technologies proposées par Sun est définie dans une spécification JSR-127 (version 1.1) puis JSR-252 (1.2)
- Il existe donc plusieurs implémentations de JSF
  - Sun Reference : <http://java.sun.com/javaee/javaxserverfaces>
  - Apache MyFaces : <http://myfaces.apache.org>
- L'implémentation proposée par Apache est appelée **MyFaces**
- Apache fournit des fonctionnalités additionnels via le sous projet : **Tomahawk**
  - Composants graphiques
  - Validators plus fournis

La version 1.1 est associé à J2EE 1.4 alors que la version 1.2 de JSF est associée à Java EE 5



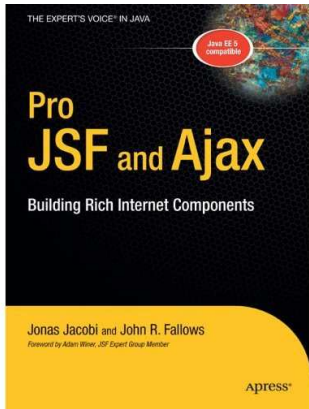
# Introduction : documentations et ressources

---

- Même si JSF n'est pas aussi utilisé et documenté que Struts, de nombreux sites commencent à apparaître pour proposer tutoriels et exemples (peu de ressource en français)
- Tutoriels / Cours
  - Java EE training and tutorials : <http://www.coreservlets.com>
  - Tutoriaux chez Developpez.com : <http://www.developpez.com>
  - JM Doudoux : <http://www.jmdoudoux.fr/accueil.htm>
  - Site de référence : <http://java.sun.com/javaee/javaxserverfaces>
  - Site du projet : <http://javaxserverfaces.dev.java.net>
  - Tutorial Java EE : <http://java.sun.com/javaee/5/docs/tutorial/doc>
  - FAQ : <http://wiki.java.net/bin/view/Projects/JavaServerFacesSpecFaq>
- Exemples et composants
  - Exemples JSF (AJAX, composants) : <http://jsftutorials.net>
  - Communauté JSF : <http://www.jsfcentral.com>

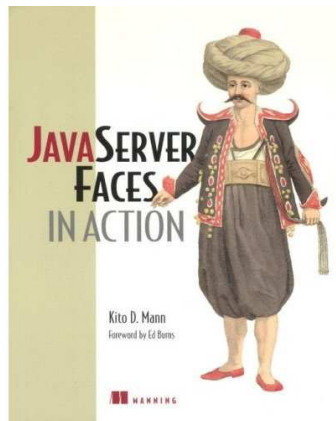


# Bibliothèque ...



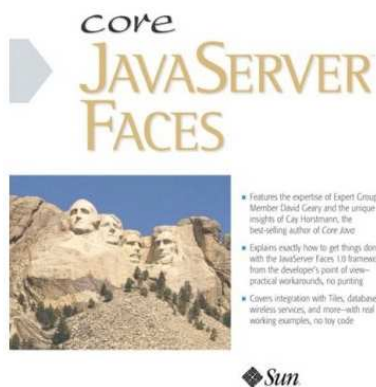
## ➤ Pro JSF & Ajax : Building Rich Internet ...

- Auteur : Jonas Jacobi
- Éditeur : Apress
- Edition : 2006 - 464 pages - ISBN : 1-59059-580-7



## ➤ JavaServer Faces in Action

- Auteur : Kito Mann
- Éditeur : Manning Publications
- Edition : 2005 - 702 pages - ISBN : 1-93239-412-2



## ➤ Core JavaServer Faces

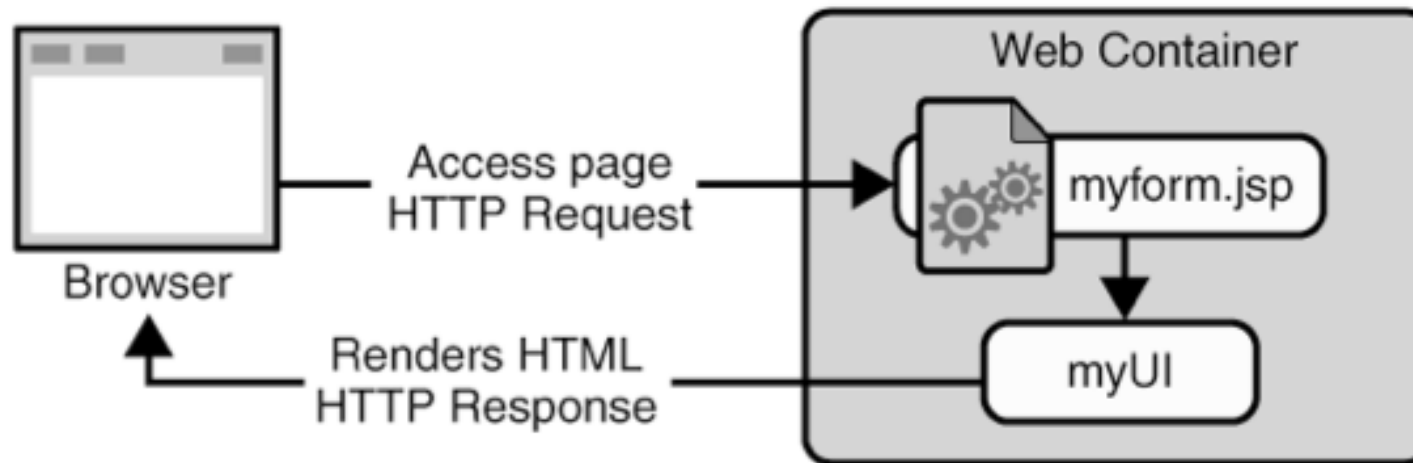
- Auteur : David Geary and Cay Horstmann
- Éditeur : Prentice Hall Ptr
- Edition : 2004 - 658 pages - ISBN : 0-13146-305-5

DAVID GEARY • CAY HORSTMANN

Java™ 2 Platform, Enterprise Edition Series

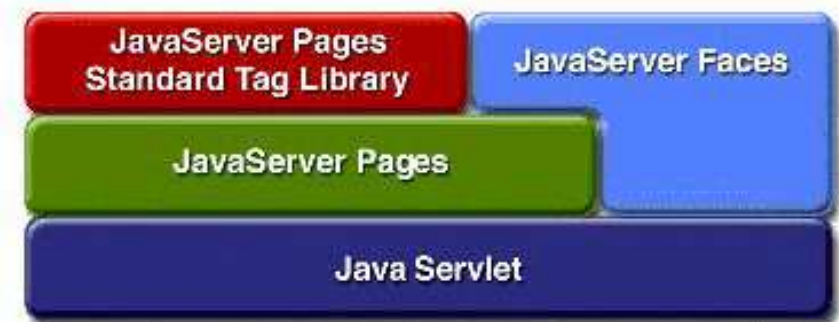
## Introduction : comment ça marche ...

- L'interface utilisateur construite dans la page JSP est générée à l'aide de la technologie JSF (résultat *myUI*)
- Elle fonctionne sur le serveur et le rendu est retourné au client



- JSF s'appuie sur les technologies précédentes

- Génération en Servlet
- Utilisation des composants JSF dans les pages JSP
- Les composants JSF sont exposés aux JSPs grâce aux balises personnalisés



# Introduction : principe pour traiter un formulaire

1. Construire le formulaire dans une page JSP en utilisant les balises JSF
2. Développer un Bean qui effectue un « Mapping » avec les valeurs du formulaire
3. Modifier le formulaire pour spécifier l'action et l'associer au Bean
4. Fournir des *Converters* et des *Validators* pour traiter les données du formulaire
5. Paramétrer le fichier *faces-config.xml* pour déclarer le Bean et les règles de navigation
6. Créer les pages JSP correspondant à chaque condition de retour
7. Protéger les pages JSP utilisées par le contexte JSF de façon à éviter d'y accéder directement

## Plan du cours JSF

---

- **Configuration** : utiliser JSF dans une application Java EE
- **Bean Managé** : stocker, afficher les valeurs d'un formulaire
- **Cycle de vie** : comprendre les rouages d'un composant JSF
- **Navigation** : choisir une vue selon un message
- **Composants CORE et HTML** : gérer et construire la vue
- **FacesContext** : manipuler informations du contexte
- **BackingBean** : manipuler les instances des composants
- **Message** : stocker et afficher des messages dans les JSP
- **Converters** : transformer les valeurs saisies d'un formulaire
- **Validators** : valider les valeurs saisies d'un formulaire
- **Evénements** : gérer les actions utilisateur
- **Comp. Tomahawk** : ajouter de nouveaux composants
- **Création de composants JSF** : étendre la bibliothèque

## Configuration : JSF dans le web.xml

---

- Nécessite la configuration du fichier **web.xml** de façon à ce que JSF soit pris en compte
  - Paramétrer le fonctionnement général de l'application : le contrôleur
  - Identifier la servlet principale : *javax.faces.webapp.FacesServlet*
- Spécifier le nom et le chemin du fichier de configuration
  - Nom du paramètre : *javax.faces.application.CONFIG\_FILES*
  - Exemple : */WEB-INF/faces-config.xml*
- Spécifie où l'état de l'application doit être sauvé
  - Nom du paramètre : *javax.faces.STATE\_SAVING\_METHOD*
  - Valeurs possibles : *client* ou *server*
- Valider ou pas les fichiers XML
  - Nom du paramètre : *com.sun.faces.validateXml*
  - Valeurs possibles : *true* ou *false* (défaut : *false*)

## Configuration : JSF dans le web.xml

- Indique si les objets développés tels que les Beans, les composants, les validators et les converters doivent être créés au démarrage de l'application
  - Nom du paramètre : *com.sun.faces.verifyObjects*
  - Valeurs possibles : *true* ou *false* (défaut : *false*)
- La servlet principale est le point d'entrée d'une application JSF
  - On trouve plusieurs manières de déclencher des ressources JSF
    - Préfixe */faces/*
    - Suffixes *\*.jsf* ou *\*.faces*
  - Exemples (le contexte de l'application est *myAppli*)
    - *http://localhost/myAppli/faces/index.jsp*
    - *http://localhost/myAppli/index.jsf*

**Avec Struts nous utilisons généralement le suffixe « .do » pour atteindre le contrôleur**



# Configuration : JSF dans le web.xml

## ➤ Exemple : paramétrer une application Web de type JSF

Utilisation de *context-param* pour paramétrer le fonctionnement des JSF

```
...
<context-param>
  <param-name>com.sun.faces.validateXml</param-name>
  <param-value>true</param-value>
</context-param>
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.faces</url-pattern>
</servlet-mapping>
...
```

La Servlet qui gère les entrées au contexte JSF

Comment accéder à la Servlet « Faces Servlet »

*web.xml*

## Configuration : accès restreints aux pages JSP

---

- Quand une page JSP utilise des composants JSF elle doit être traitée obligatoirement par la Servlet principale
  - *http://localhost/myAppli/faces/index.jsp* : appel de la page *index.jsp*
- Dans le cas où une page JSP est appelée directement sans passer par la Servlet principale une erreur est générée
  - *http://localhost/myAppli/index.jsp* : **erreur !!!**
- Empêcher donc les clients d'accéder directement aux pages JSP qui exploitent des composants JSF
- Solutions
  - Utiliser la balise *security-constraint* dans le fichier *web.xml*
  - Déclarer chaque page JSP qui doit être protégée (celles qui utilisent des composants JSF)



# Configuration : accès restreints aux pages JSP

## ➤ Exemple : restreindre l'accès aux pages JSP

```
...
<context-param>
  <param-name>com.sun.faces.validateXml</param-name>
  <param-value>>true</param-value>
</context-param>
...
<security-constraint>
  <web-resource-collection>
    <web-resource-name>No-JSF-JSP-Access</web-resource-name>
    <url-pattern>/welcome.jsp</url-pattern>
    <url-pattern>/form1.jsp</url-pattern>
    <url-pattern>/accepted.jsp</url-pattern>
    <url-pattern>/refused.jsp</url-pattern>
    <url-pattern>...</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description>Pas de rôles, donc pas d'accès direct</description>
  </auth-constraint>
</security-constraint>
...
```

Ajouter la balise *security-constraint* dans le fichier web.xml

Les pages JSP qui utilisent des composants JSF

**Protéger efficacement toutes les pages JSP qui utilisent des composants JSF**



## Configuration : le fichier contrôleur « *faces-config.xml* »

---

- Le fichier gérant la logique de l'application web s'appelle par défaut *faces-config.xml*
- Il est placé dans le répertoire WEB-INF au même niveau que *web.xml*
- Il décrit essentiellement six principaux éléments :
  - les Beans managés *<managed-bean>*
  - les règles de navigation *<navigation-rule>*
  - les ressources éventuelles suite à des messages *<message-bundle>*
  - la configuration de la localisation *<resource-bundle>*
  - la configuration des Validators et des Converters *<validator>*  
*<converter>*
  - d'autres éléments liés à des nouveaux composants JSF *<render-kit>*
- Le fichier de configuration est un fichier XML décrit par une DTD. La balise de départ est *<faces-config>*  
(version 1.1)

# Configuration : le fichier contrôleur « faces-config.xml »

- La description de l'ensemble des balises peut être trouvée
  - <http://www.horstmann.com/corejsf/faces-config.html>

```
<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>
    <navigation-rule>
        ...
    </navigation-rule>
    <managed-bean>
        ...
    </managed-bean>
</faces-config> •
```

Description de fonctionnement  
de l'application JSF

**La version 1.2 de JSF utilise un schéma au lieu d'une DTD**



# Configuration : balises personnalisées dans les JSP

- Les composants JSF sont utilisés dans les pages JSP au moyen de balises personnalisées dédiées aux JSF
  - *CORE* : noyau qui gère les vues
  - *HTML* : composants JSF
- Description des balises personnalisées *Core* et *HTML*
  - [http://java.sun.com/javaee/javaxserverfaces/1.2\\_MR1/docs/tlddocs](http://java.sun.com/javaee/javaxserverfaces/1.2_MR1/docs/tlddocs)
  - <http://www.horstmann.com/corejsf/jsf-tags.html>
- Possibilité d'utiliser d'autres balises (Tomahawk, ADF Faces)
- Les composants JSF doivent être encadrés par une vue JSF déclarée par la balise `<core:view>`

**Le détail de CORE et HTML sera donné plus tard** 

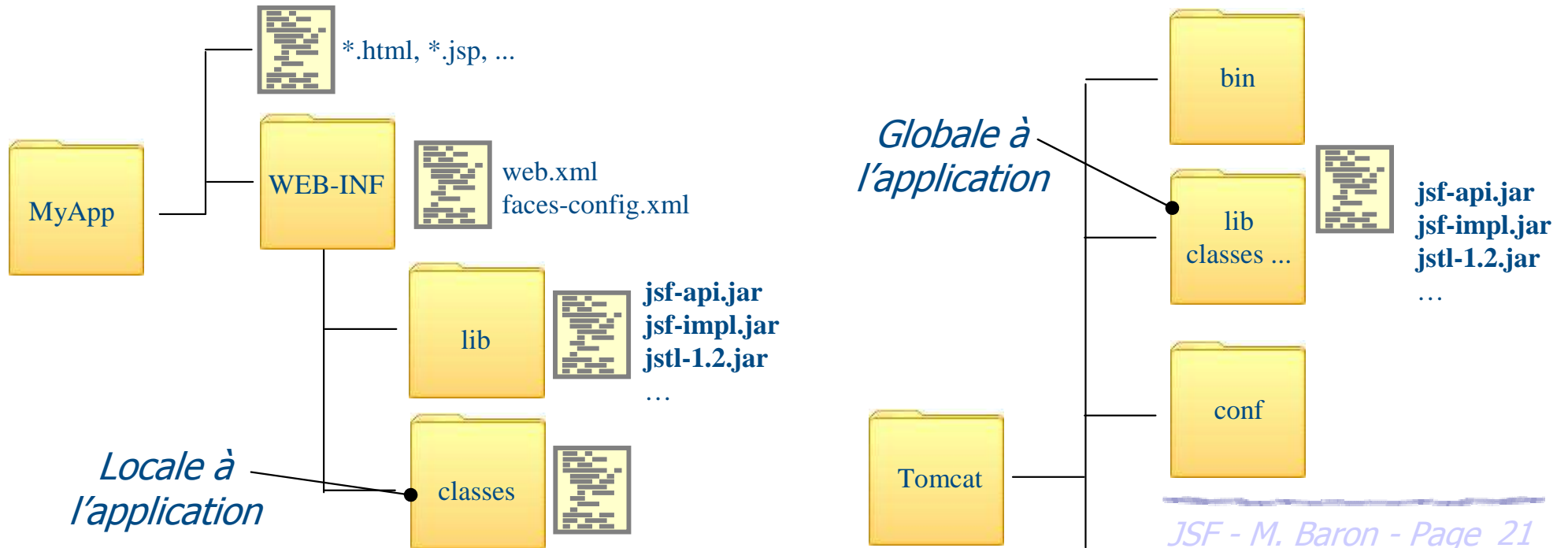
```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
...
<core:view>
    ...
</core:view>
...
```

Utilisation des composants JSF

**Possibilité d'utiliser des balises HTML classiques c'est du JSP!!!** 

# Configuration : JSF et Tomcat

- Pour exploiter et déployer une application WEB avec JSF il faut
  - des librairies JSF (Apache ou Sun) et JSTL
  - configurer le fichier web.xml selon la version de JSF
- Les librairies peuvent être copiées soit en locale à l'appli web sont en globale dans le répertoire Tomcat



# Configuration : JSF et Tomcat

- Selon la version et le serveur d'application, le fichier web.xml doit être complété (Tomcat 6 pour nos tests)
- Pour la version 1.1.5 de JSF (MyFaces)

```
<web-app ...>
  ...
  <listener>
    <listener-class>
      org.apache.myfaces.webapp.StartupServletContextListener
    </listener-class>
  </listener>
</web-app>
```

- Pour la version 1.2 de JSF (Sun)

```
<web-app ...>
  ...
  <listener>
    <listener-class>
      com.sun.faces.config.ConfigureListener
    </listener-class>
  </listener>
  <listener>
    <listener-class>
      com.sun.faces.application.WebappLifecycleListener
    </listener-class>
  </listener>
</web-app>
```

*web.xml*

# Le premier exemple « Hello World avec JSF »

## ➤ Exemple : afficher le message « Hello World » avec JSF

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib uri="http://java.sun.com/jsf/core" prefix="core"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="html"%>

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Hello World avec JSF</title>
  </head>
  <body>
    <core:view>
      <h1><html:outputText value="Hello World avec JSF" /></h1><br>
      La même chose avec du HTML : <h1>Hello World avec JSF</h1>
    </core:view>
  </body>
</html>
```



*welcomeJSF.jsp* du projet

***HelloWorld***

# Le premier exemple « Hello World avec JSF »

## ➤ Exemple (suite) : afficher le message « Hello World » ...

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <security-constraint>
    <display-name>Constraint1</display-name>
    <web-resource-collection>
      <web-resource-name>No-JSP-JSF-Page</web-resource-name>
      <url-pattern>/welcomeJSF.jsp</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <description>Pas de rôles, donc pas d'accès direct</description>
    </auth-constraint>
  </security-constraint>
</web-app>
```



# Le premier exemple « Hello World avec JSF »

## ➤ Exemple (suite) : afficher le message « Hello World » ...

Considérons que la page *welcomeJSF.jsp* **n'est pas protégée** et que la ressource est accédée directement (sans passer par la Servlet principale)

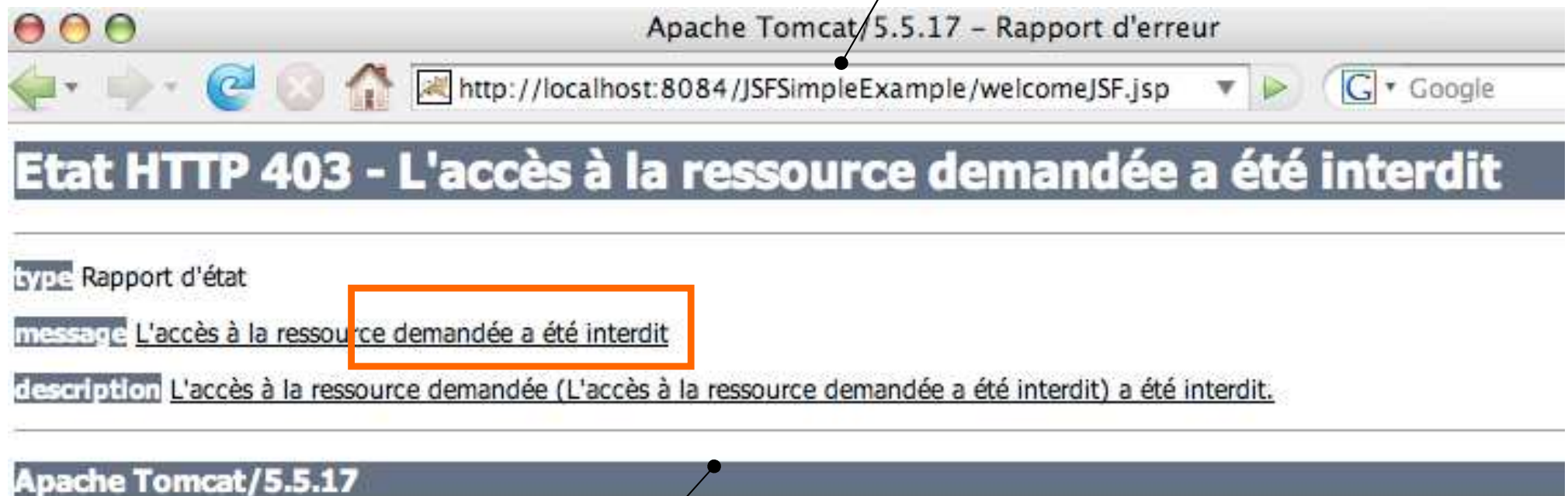


Retour d'un message d'erreur car les éléments JSF ne sont pas traités par la Servlet principale

# Le premier exemple « Hello World avec JSF »

## ➤ Exemple (suite) : afficher le message « Hello World » ...

Considérons que la ressource *welcomeJSF.jsp* **est protégée** et que la ressource est accédée directement (sans passer par la Servlet principale)



Retour d'un message d'erreur car la page est protégée

# Le premier exemple « Hello World avec JSF »

## ➤ Exemple (suite) : afficher le message « Hello World » ...

```
<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

</faces-config>
```

**Dans cet exemple aucune configuration est nécessaire. Nous verrons dans la suite les règles de navigation et la déclaration de Bean managés**



*faces-config.xml*  
du projet  
***HelloWorld***

## Bean Managé : principe

---

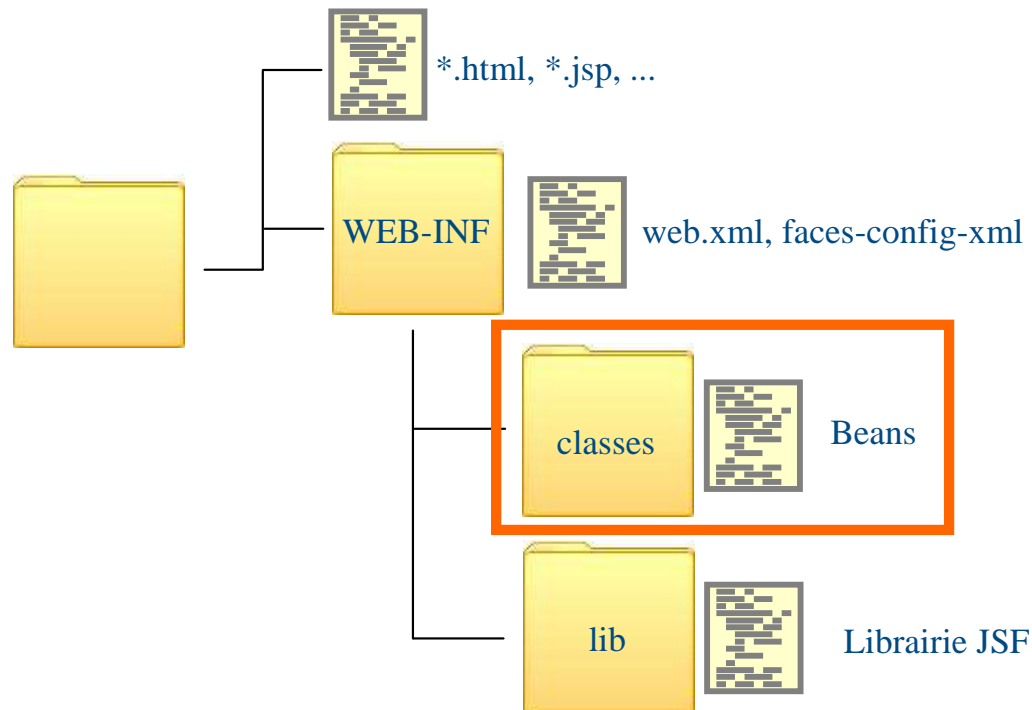
- Rappelons qu'un Bean est une classe Java respectant un ensemble de directives
  - Un constructeur public sans argument
  - Les propriétés d'un Bean sont accessibles au travers de méthodes *getXXX* et *setXXX* portant le nom de la propriété
- L'utilisation des Beans dans JSF permet
  - l'affichage des données provenant de la couche métier
  - le stockage des valeurs d'un formulaire
  - la validation des valeurs
  - l'émission de messages pour la navigation (reçus par *faces-config.xml*)

**Dans la suite nous verrons que les Beans  
peuvent retourner des messages  
exploitables par le contrôleur JSF :  
principe de la Navigation**



# Bean Managé : principe

- Un Bean managé est un Bean dont la vie est gérée par JSF et déclaré dans le fichier de configuration *faces-config.xml*
  - Définir la durée de vie (scope) d'un Bean
  - Initialiser les propriétés d'un Bean
- A la différence des Beans utilisés dans Struts, les Beans managés de JSF n'héritent pas d'une classe particulière
- Les classes sont stockées dans le répertoire WEB-INF/classes



# Bean managé : configuration dans faces-config.xml

- Pour créer un Bean managé il faut le déclarer dans le fichier de configuration de JSF à l'aide de la balise `<managed-bean>`
- Trois éléments essentiels sont à préciser
  - `<managed-bean-name>` définit un nom qui servira d'étiquette quand le Bean sera exploité dans les pages JSP
  - `<managed-bean-class>` déclare le nom de la classe de type *package.class*
  - `<managed-bean-scope>` précise le type de scope utilisé pour le Bean
    - *none, application, session, request*

```

...
<navigation-rule>...</navigation-rule>
<managed-bean>
    <managed-bean-name>MyBean</managed-bean-name>
    <managed-bean-class>mypackage.MyFirstBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
...
    
```

Etudiée dans la partie Navigation

## Bean managé : JSP, Bean et Expression Language (EL)

- Un formulaire JSF doit être construit dans un groupe défini par la balise `<html:form> ... </html:form>`
  - *ACTION* est automatiquement à *SELF* (URL courante)
  - *METHOD* est obligatoirement *POST*
- Utilisation de composants JSF pour saisir des informations
  - `<html:inputText>` pour la balise HTML `<INPUT TYPE="Text">`
  - `<html:inputSecret>` pour la balise `<INPUT TYPE="PASSWORD">`
  - `<html:commandButton>` pour la balise `<INPUT TYPE="SUBMIT">`
- La balise `<html:commandButton>` contient un attribut *action* qui permet d'indiquer un message traité par les règles de navigation définies dans *faces-config.xml*

**Nous verrons dans la partie  
Navigation le traitement de la valeur  
indiquée dans l'attribut *action***





# Bean managé : JSP, Bean et Expression Language (EL)

## ➤ Exemple : formulaire JSP utilisant des composants JSF

```
<%@page contentType="text/html"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="core"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="html"%>
<html>
  <head>
    <title>Premier formulaire avec JSF</title>
  </head>
  <body>
    <core:view>
      <html:form>
        <html:outputText value="Adresse Email " /><html:inputText/><br>
        <html:outputText value="Mot de Passe " /><html:inputSecret/><br>
        <html:commandButton value="Connecter" />
      </html:form>
    </core:view>
  </body>
</html>
```



Premier formulaire avec JSF

Adresse Email

Mot de Passe

Connecter

*beanform1.jsp* du projet  
**ManagedBean**



## Bean managé : JSP, Bean et Expression Language (EL)

- Les **E**xpressions **L**anguages (EL) sont utilisées pour accéder aux éléments du Bean dans les pages JSP
- Un EL permet d'accéder simplement aux Beans des différents scopes de l'application (*page*, *request*, *session* et *application*)
- Forme d'un Expression Language JSF

#{expression}

- Les EL JSF sont différentes des EL JSP qui utilisent la notation *#{expression}*
- Une EL est une expression dont le résultat est évaluée au moment où JSF effectue le rendu de la page

# Bean managé : JSP, Bean et Expression Language (EL)

- L'écriture `#{MyBean.value}` indique à JSF
  - de chercher un objet qui porte le nom de *MyBean* dans son contexte
  - puis invoque la méthode *getValue()* (chercher la propriété *value*)
- Le nom de l'objet est déterminé dans le fichier *faces-config.xml*

```
...  
<navigation-rule>...</navigation-rule>  
<managed-bean>  
    <managed-bean-name>MyBean</managed-bean-name>  
    <managed-bean-class>mypackage.MyFirstBean</managed-bean-class>  
    <managed-bean-scope>session</managed-bean-scope>  
</managed-bean>  
...
```

- Possibilité d'accéder à un objet contenu dans le Bean
  - `#{MyBean.myobject.value}` : propriété *value* de *myobject* contenu dans *MyBean*

# Bean managé : JSP, Bean et Expression Language (EL)

➤ JSF définit un ensemble d'objets implicites utilisables dans les Expressions Languages :

- *param* : éléments définis dans les paramètres de la requête HTTP
- *param-values* : les valeurs des éléments *param*
- *cookies* : éléments définis dans les cookies
- *initParam* : paramètres d'initialisation de l'application
- *requestScope* : éléments défini dans la requête
- *facesContext* : instance de la classe *FacesContext*
- *View* : instance de *UIViewRoot*

Etudiée au niveau de la  
partie *cycle de vie*

Expliquer au niveau des  
composants graphiques

# Bean managé : JSP, Bean et Expression Language (EL)

## ➤ Exemple : objets implicites JSF dans une JSP

```
<html>
  <...>
  <body>
    <core:view>
      <p><html:outputText
        value="#{param.test}" /></p>
      <p><html:outputText
        value="#{cookie.JSESSIONID.value}" /></p>
      <p><html:outputText
        value="#{facesContext.externalContext.requestPathInfo}" /></p>
      <p><html:outputText
        value="#{facesContext.externalContext.requestServletPath}" /></p>
    </core:view>
  </body>
</html>
```

Utilisation des objets impli...

CCD5180840025B229F22EC3B0C235CAC

/converters1.jsp

/faces

*implicitobjects.jsp* du  
projet **ImplicitObjects**

## Bean managé : modifier/afficher la propriété d'un Bean

---

- Un formulaire regroupe un ensemble de composants contenant chacun une valeur (attribut *value*)
- Chaque valeur d'un composant peut être stockée dans une propriété du Bean
- Pour définir une propriété dans un Bean
  - Créer l'attribut correspondant à la propriété
  - Définir des méthodes *set* et *get* pour accéder en lecture et en écriture aux propriétés
- Pour stocker la valeur d'un composant vers le Bean ou afficher la valeur d'une propriété dans un composant : utilisation des EL dans l'attribut *value* du composant JSF
- JSF associe automatiquement la propriété d'un Bean à la valeur d'un composant

# Bean managé : modifier/afficher la propriété d'un Bean

## ➤ Exemple : gestion d'un formulaire « email et mot de passe »

```
package beanPackage;
public class RegistrationBean {

    private String email = "user@host";

    private String password = "";

    public String getEmail() {
        return email;
    }
    public void setEmail(String t) {
        this.email = t;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String t) {
        this.password = t;
    }
}
```

*RegistrationBean.java* du  
projet **ManagedBean**  
possède deux propriétés et  
des modifieurs et accesseurs

# Bean managé : modifier/afficher la propriété d'un Bean

## ➤ Exemple (suite) : gestion d'un formulaire ...

```
...  
<core:view>  
  <html:form>  
    <html:outputText value="Adresse Email " />  
    <html:inputText value="#{registrationbean.email}" /><br>  
  
    <html:outputText value="Mot de Passe " />  
    <html:inputSecret value="#{registrationbean.password}" /><br>  
  
    <html:commandButton value="Connecter" />  
  </html:form>  
</core:view>  
</body>  
</html>
```

EL pour accéder à la  
propriété *email*

EL pour accéder à la  
propriété *password*

*beanform2.jsp* du projet  
**ManagedBean**



Adresse Email :

Mot de Passe :

Valider

Affichage de la valeur  
d'initialisation du Bean

# Bean managé : modifier/afficher la propriété d'un Bean

## ➤ Exemple (suite) : gestion d'un formulaire ...

```
<faces-config>
  <managed-bean>
    <managed-bean-name>
      registrationbean
    </managed-bean-name>
    <managed-bean-class>
      beanpackage.RegistrationBean
    </managed-bean-class>
    <managed-bean-scope>
      request
    </managed-bean-scope>
  </managed-bean>
</faces-config>
```

Identifiant utilisé pour accéder  
au Bean défini par la classe  
*beanpackage.RegistrationBean*

*faces-config.xml* du projet  
**ManagedBean**



# Bean managé : initialisation des propriétés d'un Bean

- Possibilité d'initialiser la propriété d'un Bean dans
  - le Bean lui-même
  - le fichier *faces-config.xml*
- Dans le fichier *faces-config.xml* utilisation des EL
  - Accès aux objets implicites
  - Accès aux autres Beans managés
- L'initialisation d'un Bean est obtenue à l'intérieur de la balise *<managed-bean>* dans la balise *<managed-property>*
- Chaque propriété à initialiser est déclarée par la balise *<property-name>* contenu dans *<managed-property>*
- Une propriété peut être de différents types
  - *List* : définit une propriété de type liste
  - *Map* : définit une propriété de type Map (une clé / valeur)
  - *Value* : définit une propriété sur une valeur

## Bean managé : initialisation des propriétés d'un Bean

- Si la propriété est de type *List*, l'initialisation des valeurs de la propriété est effectuée dans la balise `<list-entries>`
- Dans la balise `<list-entries>` deux informations sont à indiquer
  - `<value-class>` : le type d'objet contenu dans la liste
  - `<value>` : une valeur dans la liste (autant d'élément de la liste)

```
<managed-bean>
...
<managed-property>
  <property-name>cities</property-name>
  <list-entries>
    <value-class>java.lang.String</value-class>
    <value>Poitiers</value>
    <value>Limoges</value>
    <value>Viroflay</value>
  </list-entries>
</managed-property>
</managed-bean>
```

# Bean managé : initialisation des propriétés d'un Bean

- Si la propriété est de type *Map*, l'initialisation des valeurs de la propriété est effectuée dans la balise `<map-entries>`
- Dans la balise `<map-entries>` trois informations sont à donner
  - `<key-class>` : le type de la clé
  - `<value-class>` : le type d'objet contenu dans la *Map*
  - `<map-entry>` : contient une clé et une valeur
- La balise `<map-entry>` contient deux informations
  - `<key>` : la clé
  - `<value>` : la valeur associée à la clé

```
<managed-property>
  <property-name>prices</property-name>
  <map-entries>
    <map-entrie>
      <key>SwimmingPool High Pressure</key>
      <value>250</value>
    </map-entrie>
  </map-entries>
```

# Bean managé : initialisation des propriétés d'un Bean

- Si la propriété est de type Value, l'initialisation de la valeur de la propriété est effectuée dans la balise `<value>`

```
<managed-bean>
  ...
  <managed-property>
    <property-name>email</property-name>
    <value>user@host</value>
  </managed-property>
  <managed-property>
    <property-name>name</property-name>
    <value>your name</value>
  </managed-property>
  <managed-property>
    <property-name>adress</property-name>
    <value>your adress</value>
  </managed-property>
</managed-bean>
```

*faces-config.xml* du  
projet  
**ManagedBean**

# Bean managé : initialisation des propriétés d'un Bean

## ➤ Exemple : initialisation de propriétés dans *faces-config.xml*

```
</managed-bean>
<managed-bean>
  <managed-bean-name>registrationbeanbis</managed-bean-name>
  <managed-bean-class>
    beanPackage.RegistrationBeanBis
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>email</property-name>
    <value>#{registrationbean.email}</value>
  </managed-property>
  <managed-property>
    <property-name>adress</property-name>
    <value>Your Adress</value>
  </managed-property>
</managed-bean>
```

Utilisation d'une EL pour accéder au Bean défini par *registrationbean*

*faces-config.xml* du projet

**ManagedBean**

# Bean managé : initialisation des propriétés d'un Bean

## ➤ Exemple (suite) : initialisation de propriétés ...

```
<html>
  <head>
    <title>Troisième formulaire JSF avec un Bean Managé ...</title>
  </head>
  <body>
    <core:view>
      <html:form>
        <html:outputText value="Adresse Email " />
        <html:inputText value="#{registrationbeanbis.email}" /><br>
        <html:outputText value="Adresse Postale " />
        <html:inputText value="#{registrationbeanbis.adress}" /><br>
        <html:commandButton value="Connecter" />
      </html:form>
    </core:view>
  </body>
</html>
```

*beanform3.jsp* du projet  
**ManagedBean**

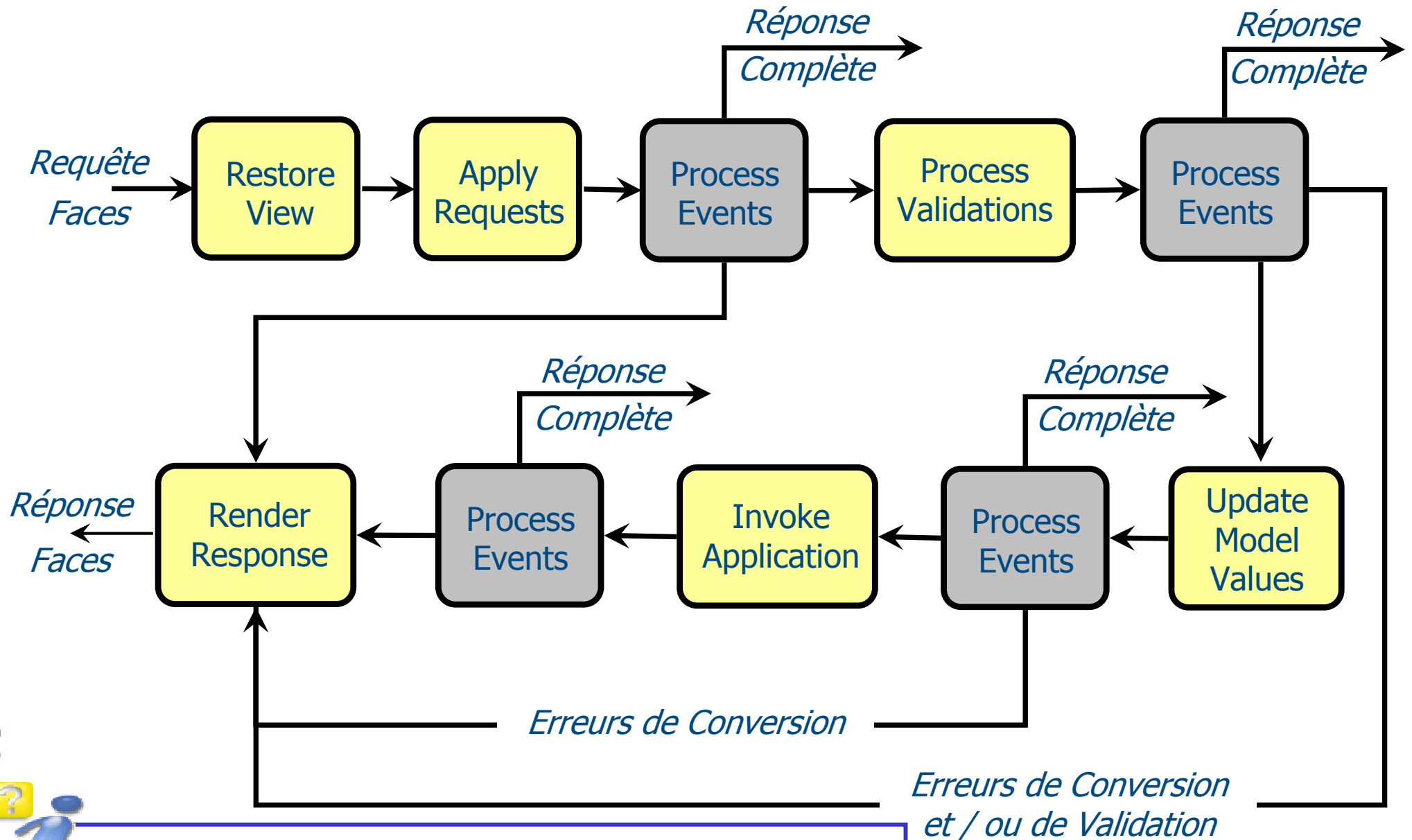
Formulaire JSF avec un Bean...

Adresse Email

Adresse Postale

Connecter

# Cycle de vie d'une JSF : généralités



**Détaillé au fur et à mesure : conversion, validation, événements, ...**

## Cycle de vie d'une JSF : généralités

---

- ***Restore View*** : JSF reconstruit l'arborescence des composants qui composent la page.
- ***Apply Requests*** : les valeurs des données sont extraites de la requête
- ***Process Validations*** : procède à la validation des données
- ***Update model values*** : mise à jour du modèle selon les valeurs reçues si validation ou conversion réussie
- ***Invoke Application*** : les événements émis de la page sont traités. Elle permet de déterminer la prochaine page
- ***Render Response*** : création du rendu de la page



## Navigation : configuration de faces-config.xml

---

- Le fichier de *faces-config.xml* joue le rôle de contrôleur, il décide de la ressource qui doit être appelée suite à la réception d'un message
- Les messages sont des simples chaînes de caractères
- Utilisation de la balise *<navigation-rule>* pour paramétrer les règles de navigation
- La balise *<from-view-id>* indique la vue source où est effectuée la demande de redirection. La vue peut être un :
  - Formulaire (action de soumission)
  - Lien hypertext
- Pour chaque valeur de message une page vue de direction est indiquée dans la balise *<navigation-case>*
  - *<from-outcome>* : la valeur du message
  - *<to-view-id>* : la vue de direction

# Navigation : statique ou dynamique

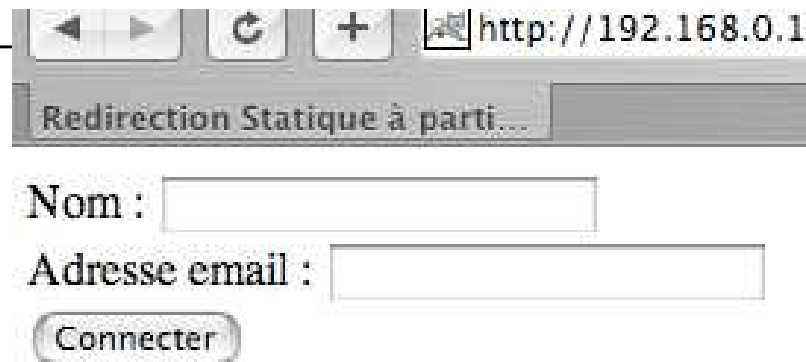
---

- JSF implémente une machine à états pour gérer la navigation entre des pages JSF
- Pour schématiser nous distinguons deux sortes de navigation
  - Navigation statique
  - Navigation dynamique
- **Navigation statique**
  - La valeur de l'outcome est connue au moment de l'écriture de la JSP
- **Navigation dynamique**
  - La valeur de l'outcome est inconnue au moment de l'écriture de la JSP
  - Elle peut être calculée par un Bean Managé ou autre chose ...
- Remarques
  - Si une valeur d'outcome est inconnue la même page JSP est rechargée
  - Si la valeur vaut *null* la page JSP est rechargée

# Navigation statique : exemples

## ➤ Exemple : redirection statique à partir d'un formulaire

```
...
<html>
  <head>
    <title>Redirection Statique à partir d'un formulaire</title>
  </head>
  <body>
    <core:view>
      <html:form>
        <html:outputText value="Nom : " />
        <html:inputText value="#{beancontroller1.name}" /><br>
        <html:outputText value="Adresse email : " />
        <html:inputText value="#{beancontroller1.email}" /><br>
        <html:commandButton value="Connecter" action="register"/>
      </html:form>
    </core:view>
  </body>
</html>
```



Nom :

Adresse email :

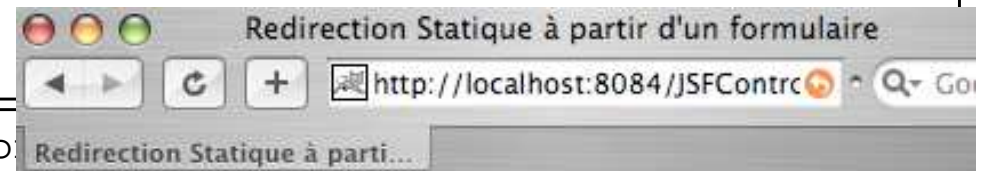
*form1.jsp* du projet  
**Navigation**

# Navigation statique : exemples

## ➤ Exemple (suite) : redirection statique ...

```
...
<faces-config>
  <navigation-rule>
    <from-view-id>/form1.jsp</from-view-id>
    <navigation-case>
      <from-outcome>register</from-outcome>
      <to-view-id>/accepted.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

*faces-config.xml*



## Redirection Réussie

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="core"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="html"%>

<html>
  ...
  <body>
    <core:view>
      <h1><html:outputText value="Redirection Réussie" /></h1>
    </core:view>
  </body>
</html>
```

*acceptedSimple.jsp*  
du projet **Navigation**



**N'oubliez pas de protéger toutes les pages  
JSP : *form1.jsp* et *acceptedSimple.jsp***

# Navigation dynamique : exemples

## ➤ Exemple : redirection dynamique à partir d'un Bean

```
package beanPackage;
public class BeanController2 {

    private String email = "user@host";
    private String name = "";
    ...

    public String loginConnect() {
        if (this.email.isEmpty()) {
            return "Rejected";
        }
        if (this.name.isEmpty()) {
            return "Rejected";
        }
        return "Accepted";
    }
}
```

Cette méthode permet de valider les valeurs et de choisir le message à retourner au contrôleur JSF



**Il s'agit ici d'une technique pour la validation des informations saisies dans un formulaire. Nous approfondirons cet aspect dans la partie *Validators***

*BeanController2.java*  
du projet **Navigation**

# Navigation dynamique : exemples

## ➤ Exemple (suite) : redirection dynamique à partir d'un Bean

```
<faces-config>
  <navigation-rule>
    <from-view-id>form2.jsp</from-view-id>
    <navigation-case>
      <from-outcome>Accepted</from-outcome>
      <to-view-id>/accepted.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>Rejected</from-outcome>
      <to-view-id>/rejected.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  <managed-bean>
    <managed-bean-name>beancontroller2</managed-bean-name>
    <managed-bean-class>
      beanPackage.BeanController2
    </managed-bean-class>
    ...
  </faces-config>
```

*faces-config.xml* du  
projet **Navigation**

# Navigation dynamique : exemples

## ➤ Exemple (suite) : redirection dynamique à partir d'un Bean

```
...
<html>
  <head>
    <title>Redirection Dynamique à partir d'un formulaire</title>
  </head>
  <body>
    <core:view>
      <html:form>
        <html:outputText value="Nom : " />
        <html:inputText value="#{beancontroller2.name}" /><br>
        <html:outputText value="Adresse email : " />
        <html:inputText value="#{beancontroller2.email}" /><br>
        <html:commandButton value="Connecter"
          action="#{beancontroller2.loginConnect}" />
      </html:form>
    </core:view>
  </body>
</html>
```

Accès au Bean identifié par  
*beancontroller2* et à la  
méthode *loginConnect*

*form2.jsp* du projet  
**Navigation**

# Navigation dynamique : exemples

## ➤ Exemple (suite) : redirection dynamique à partir d'un Bean

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="core"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="html"%>

<html>
  <head>
    <title>Résultat du traitement du formulaire</title>
  </head>
  <body>
    <core:view>
      <h1><html:outputText value="Connexion de : " />
        <html:outputText value="#{beancontroller2.name}" /></h1>
      </core:view>
    </body>
  </html>
```

*accepted.jsp* du  
projet  
**Navigation**



Page affichée à la suite de la  
soumission. Lecture de la  
propriété email du *Bean*  
identifié par *beancontroller2*

Connexion de : exemple@toto.fr



# Composants graphiques : présentation

- JSF fournit un ensemble de composants graphiques pour la conception de l'IHM
- Un composant JSF est développé à partir de :
  - classes qui codent le comportement et l'état
  - classes de « rendu » qui traduisent la représentation graphique (HTML, FLASH, XUL, ...)
  - classes qui définissent les balises personnalisées relation entre JSP et classes Java
  - classes qui modélisent la gestion des événements
  - classes qui prennent en compte les *Converters* et les *Validators*
- Pour exploiter les composants JSF dans les pages JSP, des balises personnalisées sont utilisées ...

## Composants graphiques : balises CORE

- Les balises personnalisées décrites dans *CORE* s'occupent d'ajouter des fonctionnalités aux composants JSF
- Pour rappel pour avoir un descriptif de l'ensemble des balises CORE :
  - <http://java.sun.com/javaee/javaserverfaces/1.2/docs/tlddocs>
- La bibliothèque contient un ensemble de balises
  - *facet* : déclare un élément spécifique pour un composant
  - *view* et *subview* : ajoute une vue ou une sous vue
  - *attribute* et *param* : ajoute un attribut ou paramètre à un composant
  - *selectionitem* et *selectionitems* : définit un ou plusieurs éléments
  - *convertDataTime* et *convertNumber* : conversion de données
  - *validator*, *validateDoubleRange*, ... : ajoute des validators
  - *actionListener*, *valueChangeListener* : différents écouteurs
  - *loadBundle* : chargement de bundle (fichier de ressources ...)

## Composants graphiques : balises HTML

- Les balises personnalisées décrites dans HTML s'occupent de la description de l'interface graphique d'une JSF
- La spécification JSF décrit un ensemble fini de composants graphiques. Des bibliothèques tiers (My Faces) proposent des composants complémentaires
- Pour rappel pour avoir un descriptif de l'ensemble des balises HTML :
  - *<http://java.sun.com/javaee/javaserverfaces/1.2/docs/tlddocs>*
- La bibliothèque HTML propose 25 composants qui sont classifiables en quatre catégories
  - Composants de saisies notés I (pour input)
  - Composants de sorties notés O (pour output)
  - Composants de commandes notés C (pour commandes)
  - Composants de regroupement notés R (pour ...)

# Composants graphiques : balises HTML

---

- Famille des *regroupements*
  - *form*
  - *panelGroup*
  - *panelGrid*
- Famille des *saisies*
  - *inputHidden*
  - *inputSecret*
  - *inputText*
  - *inputTextArea*
  - *selectBooleanCheckbox*
  - *selectManyCheckbox*
  - *selectManyListbox*
  - *selectManyMenu*
- *selectOneListbox*
- *selectOneMenu*
- *selectOneRadio*
- Famille des *sorties*
  - *column*
  - *message* et *messages*
  - *dataTable*
  - *outputText*
  - *outputFormat*
  - *outputLink*
  - *graphicImage*
- Famille des *commandes*
  - *commandButton*
  - *commandLink*

# Composants graphiques : balises HTML *saisis*

Ceci est une zone de texte

*inputText*

XXXXXXXXXXXXXXXXXXXX

*inputSecret*

Ceci est une zone de  
texte multi-lignes

*inputTextarea*

☐ Intégrer JSF dans le processus de développement

*selectBooleanCheckbox*

*selectManyCheckbox*

☐ Element 1 ☐ Element 2 ☐ Element 3 ☐ Element 4

Element 2

*selectManyMenu*

Element 1  
Element 2  
Element 3  
Element 4

*selectOneListbox*

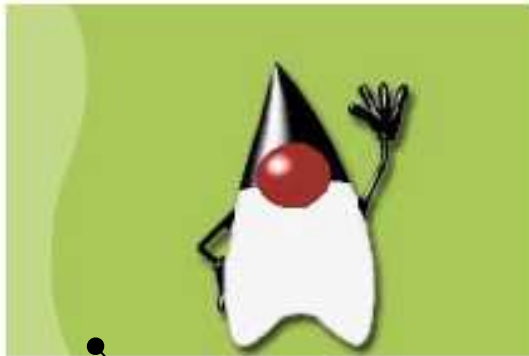
Element 1  
Element 1  
Element 2  
Element 3  
Element 4

*selectOneMenu*

☐ Element 1 ☐ Element 2 ☐ Element 3 ☐ Element 4

*selectOneRadio*

# Composants graphiques : balises HTML *sorties et commandes*



*graphicImage*

Commande Bouton

*commandButton*

Commande Lien

*commandLink*

*outputText*

Du texte affiché avec la balise `outputText`

Google

*outputLink*

Nom	Prenom	Date de naissance	Poids
Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher

*dataTable*

Nom :	<input type="text" value="Baron"/>
Mot de passe :	<input type="password" value="*****"/>

*panelGrid*

# Composants graphiques : détail balises HTML

---

- Une balise contient généralement trois types d'attributs
  - Des attributs basiques (description)
  - Des attributs HTML (description de présentation)
  - Des attributs DHTML (description pour la partie dynamique)
- Selon la nature du composant graphique (saisie, sortie, commande et autre) le nombre d'attributs varie ...
- Nous décrirons rapidement les attributs les plus significatifs. Au fur et à mesure de la présentation des composants graphiques nous les détaillerons via des exemples
- Les transparents qui suivent sont réalisés dans une optique de présentation rapide ...

# Composants graphiques : détail balises HTML

- Les attributs basiques sont utilisés pour ajouter une description spécifique à JSF
- Attributs basiques
  - *id* : identifiant du composant
  - *binding* : association avec un Bean « Backing »
  - *rendered* : *true* composant affiché, *false* composant caché
  - *styleClass* : précise le nom d'une CSS à appliquer au composant
  - *value* : valeur du composant
  - *valueChangeListener* : associer à une méthode pour le changement de valeur
  - *converter* : nom de la classe pour une conversion de données
  - *validator* : nom de la classe pour une validation de données
  - *required* : *true* valeur obligatoire, *false* valeur optionnelle



# Composants graphiques : détail balises HTML

---

- Les attributs HTML sont utilisés pour modifier l'apparence graphique d'un composant JSF
- Attributs HTML (les plus répandus ...)
  - *alt* : texte alternatif au cas où le composant ne s'affiche pas
  - *border* : bordure du composant
  - *disabled* : désactive un composant de saisie ou un bouton
  - *maxlength* : maximum de caractères pour un composant de texte
  - *readonly* : mode lecture uniquement
  - *size* : taille d'un champ de texte
  - *style* : information du style
  - *target* : nom de la frame dans lequel le document est ouvert
  - ... (et beaucoup pleins d'autres ...)

# Composants graphiques : détail balises HTML

- Les attributs DHTML sont utilisés pour ajouter un aspect dynamique pour les composants JSF (appel à du JavaScript)

- Attributs DHTML

- *onblur* : élément perd le focus
- *onClick* : clique souris sur un élément
- *ondblClick* : double clique souris sur un élément
- *onfocus* : élément gagne le *focus*
- *onkeydown*, *onkeyup* : touche clavier enfoncée et relâchée
- *onkeypress* : touche clavier pressée puis relâchée
- *onmousedown*, *onmouseup* : bouton souris enfoncé et relâché
- *onmouseout*, *onmouseover* : curseur souris sort et entre
- *onreset* : formulaire est initialisé
- *onselect* : texte est sélectionné dans un champ de texte
- *onsubmit* : formulaire soumis

**La gestion des événements sera expliquée dans une partie dédiée** ?



## Composants graphiques : *<html:panelGrid>*

---

- La balise *html:panelGrid* est utilisée pour définir l'agencement des composants visuels
- Il ne s'agit donc pas d'un composant visuel mais d'un conteneur dont l'organisation de ces composants enfants est réalisée suivant une grille
- Les composants enfants sont ajoutés dans le corps
- Principaux attributs :
  - *columns* : nombre de colonnes de la grille
  - *bgcolor* : couleur du fond
  - *cellpadding*, *cellspacing* : espacement entre les cellules
  - *border* : border autour de la grille
- Le composant *html:panelGroup* permet de regrouper des composants dans une cellule (une sorte de fusion)

# Composants graphiques : *<html:panelGrid>*

## ➤ Exemple : organiser des composants dans une grille

```
...
<h4>
  <html:panelGrid cellspacing="25" columns="2" >
    <html:outputText value="Nom : " />
    <html:panelGroup>
      <html:inputText size="15" required="true" />
      <html:inputText />
    </html:panelGroup>
    <html:outputText value="Mot de passe : " />
    <html:inputSecret />
  </html:panelGrid>
</h4>
```

Création d'une grille avec deux colonnes

Regroupement de deux composants dans une cellule

Nom :

Mot de passe :

*outputcomponent.jsp*  
du projet  
**GraphicalComponents**

## Composants graphiques : *<html:dataTable>*

---

- Le composant *<html:dataTable>* permet de visualiser des données sur plusieurs colonnes et plusieurs lignes
- La table peut être utilisée quand le nombre d'éléments à afficher est inconnu
- Les données (la partie modèle) peuvent être gérées par des Beans
- Attributs de la balise :
  - *value* : une collection de données (*Array*, *List*, *ResultSet*, ...)
  - *var* : nom donné à la variable à manipuler pour chaque ligne
  - *border*, *bgcolor*, *width* : attributs pour l'affichage
  - *rowClasses*, *headerClass* : attributs pour la gestion des styles (CSS)
- Pour chaque colonne de la table, la valeur à afficher est obtenue par la balise *<html:column>*

# Composants graphiques : *<html:dataTable>*

## ➤ Exemple : la représentation d'une table JSF

```
public class Personne {
    private String name;
    private String firstName;
    private String birthName;
    private String job;

    public Personne(String pN, String pF, String pB, String pJ) {
        name = pN; firstName = pF; birthName = pB; job = pJ;
    }
    public String getName() { return name; }
    public void setName(String pName) { name = pName; }

    public String getFirstName() { return firstName; }
    public void setFirstName(String pFirstName) { firstName = pFirstName; }
}
```

*Personne.java*  
du projet  
**GraphicalComponents**

```
public class DataTableBean {
    private List<Personne> refPersonne;
    public List getPersonne() {
        if (refPersonne == null) {
            refPersonne = new ArrayList<Personne>();
            refPersonne.add(new Personne("Baron", "Mickael", "17081976", "Développeur"));
            refPersonne.add(new Personne("Dupont", "Marcel", "21041956", "Boucher"));
            ...
        }
        return refPersonne;
    }
}
```

*OutputBean.java*  
du projet  
**GraphicalComponents**

# Composants graphiques : *<html:dataTable>*

## ➤ Exemple (suite) : la représentation d'une table JSF

```
<core:view>
  <html:dataTable value="#{outputbean.personne}" var="personne" border="1"
    cellspacing="4" width="60%" >
    <html:column>
      <html:outputText value="#{personne.name}" />
    </html:column>
    <html:column>
      <html:outputText value="#{personne.firstname}" />
    </html:column>
    <html:column>
      <html:outputText value="#{personne.birthdata}" />
    </html:column>
    <html:column>
      <html:outputText value="#{personne.job}" />
    </html:column>
  </html:dataTable>
</core:view>
```

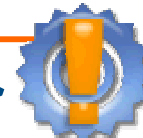
*datatablecomponent1.jsp*  
du projet  
**GraphicalComponents**

Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher
Martin	Alexandre	28011946	Magicien
Fox	Tracy	18021969	Sexologue

## Composants graphiques : *<html:dataTable>*

- La modification des en-tête et pied de page d'une table est obtenue en utilisant la balise *<core:facet>*
- *<core:facet>* s'occupe d'effectuer une relation de filiation entre un composant et un autre
- La filiation consiste à associer le composant *<core:facet>* est un composant défini dans le corps de *<core:facet>*
- Attribut du composant *<core:facet>*
  - *name* : nom de la filiation
- Pour le composant table deux filiations possibles
  - *header* : une filiation entre une colonne et le nom de la colonne
  - *footer* : une filiation entre la table et un nom
  - *caption* : une filiation entre le titre de la table et un nom

**Le composant de filiation à *<core:facet>* doit être absolument un composant JSF**





# Composants graphiques : *<html:dataTable>*

## ► Exemple : table JSF avec des noms de colonnes

```

<core:view>
  <html:dataTable value="#{outputbean.personne}" var="personne" border="1"
    cellspacing="4" width="60%" >
    <html:column>
      <core:facet name="header" >
        <html:ouputText value="Nom"
      </core:facet>
      <html:outputText value="#{personne.nom}" />
    </html:column>
    <html:column>
      <core:facet name="header" >
        <html:verbatim>Prénom</verbatim>
      </core:facet>
      <html:outputText value="#{personne.firstname}" />
    </html:column>
    <html:column>
      <core:facet name="header" >
        Date de naissance
      </core:facet>
      <html:outputText value="#{personne.birthdata}" />
    </html:column>
    <html:facet name="footer">
      <html:outputText value="#{outputbean.caption}" />
    </html:facet>
    ...
  </html:dataTable>
</core:view>

```

Nom	Prénom		Emploi
Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher
Martin	Alexandre	28011946	Magicien
Fox	Tracy	18021969	Sexologue
Une Simple Table			

Ne sera jamais affiché car  
*<core:facet>* n'est associé  
à aucun autre composant

*datatablecomponent2.jsp*  
du projet  
**GraphicalComponents**

## Composants graphiques : *<html:dataTable>*

---

- Pour l'instant, seul l'aspect peuplement des informations a été traité, *<html:dataTable>* offre la possibilité de modifier l'apparence d'une table
- Les attributs :
  - *headerClass* : style CSS pour la partie en-tête
  - *footerClass* : style CSS pour le bas de page
  - *rowClasses* : liste de styles CSS pour les lignes
- L'attribut *rowClasses* permet de définir une liste de style CSS qui sera appliquée pour chaque ligne et répétée autant de fois qu'il y a de lignes (définition d'un motif)
- Exemple :
  - Soit la définition suivante : *rowClasses="row1, row2, row2"*
  - Répétition du motif row1, row2, row2, row1, row2, ...

# Composants graphiques : *<html:dataTable>*

## ➤ Exemple : table JSF « maquillé »

```
<head>
  <...>
  <link href="output.css" rel="stylesheet" type="text/css" >
</head>
<body>
<core:view>
  <html:dataTable value="#{outputbean.personne}" var="personne" border="1"
    cellspacing="4" width="60%" rowClasses="row1,row2"
    headerClass="heading" footerClass="footer" >
    ...
  </html:dataTable>
</core:view>
```

*datatablecomponent3.jsp*  
du projet **GraphicalComponents**

```
.heading {
font-family: Arial, Helvetica, sans-serif;
font-weight: bold;
font-size: 20px;
color: black;
background-color:silver;
text-align:center;
}
.row1 {
background-color:#GFGFGF;
}
.row2 {
Background-color:#CECECE;
}
.footer {
background-color:#000009;
Color:white;
}
```

*output.css*  
du projet  
**GraphicalComponents**

Nom	Prenom	Date de naissance	Emploi
Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher
Martin	Alexandre	28011946	Magicien
Fox	Tracy	18021969	Sexologue
Une Simple Table			

# Composants graphiques : *<html:dataTable>*

## ➤ Exemple : table JSF avec des composants JSF de saisie

```
<core:view>
  <html:dataTable value="#{outputbean.personne}" var="personne" border="1"
    cellspacing="4" width="60%" rowClasses="..." >
    <html:column>
      <core:facet name="header" >
        <html:ouputText value="Nom" />
      </core:facet>
      <html:inputText value="#{personne.name}" />
    </html:column>
    <html:column>
      <core:facet name="header" >
        <html:verbatim>Prénom</verbatim>
      </core:facet>
      <html:outputText value="#{personne.firstname}" />
    </html:column>
  </html:dataTable>
</core:view>
```

Nom	Prenom	Date de naissance	Emploi
<input type="text" value="Baron"/>	Mickael	17081976	Ingénieur d'étude et de développement
<input type="text" value="Dupont"/>	Marcel	21041956	Boucher
<input type="text" value="Martin"/>	Alexandre	28011946	Magicien
<input type="text" value="Fox"/>	Tracy	18021969	Sexologue

*datatablecomponent4.jsp*  
du projet  
**GraphicalComponents**

## Composants graphiques : items

---

- Les balises HTML de type sélection (*selectOneRadio*, *selectOneMenu*) permettent de gérer un ensemble d'éléments
- Les éléments peuvent être « peuplés » par :
  - *core:selectItem* : affiche un seul élément
  - *core:selectItems* : affiche plusieurs éléments
- Les balises HTML de sélection s'occupent de gérer la sélection courante (une ou plusieurs) au travers de leur attribut *value*
- Le type de la sélection courante (retourné au serveur) est donné par les éléments de sélection (*int*, *String*, *Object*, ...)
- La balise *html:selectBooleanCheckbox* n'est pas concernée puisqu'elle ne gère qu'un seul élément

## Composants graphiques : `<core:selectItem>`

---

- La balise *core:selectItem* est utilisée pour spécifier un seul élément
- Principaux attributs de la balise *core:selectItem* :
  - *itemDescription* : description (utilisable dans les outils uniquement)
  - *itemDisabled* : active ou pas l'item
  - *value* : valeur qui pointe vers un objet *SelectItem*
  - *itemLabel* : le texte à afficher
  - *itemValue* : la valeur retournée au serveur
- Un objet *SelectItem* est exploitable directement dans un Bean
- Différents constructeurs :
  - *SelectItem(Object value)* : constructeur avec une valeur à retourner et à afficher
  - *SelectItem(Object value, String label)* : constructeur avec une valeur à retourner au serveur et une valeur à afficher

# Composants graphiques : `<core:selectItem>`

## ➤ Exemple : choix d'un élément unique

```
...  
<html:outputText value="Fruit préféré avec SelectItem : " />  
  
<html:selectOneRadio layout="pageDirection"  
    value="#{inputbean.oneRadioValue}" />  
    <core:selectItem value="#{inputbean.bananeItem}" />  
    <core:selectItem value="#{inputbean.orangeItem}" />  
    <core:selectItem itemValue="Clémentine" />  
    <core:selectItem itemValue="Pomme" />  
    <core:selectItem itemValue="Framboise" />  
</html:selectOneRadio>  
...
```

*inputcomponent.jsp*  
du projet  
**GraphicalComponents**

Utilisation d'un binding  
d'un composant  
*SelectItem*

### Fruit préféré avec SelectItem :

- ☐ Banane
- ☒ Orange
- ☐ Clémentine
- ☐ Pomme
- ☐ Framboise

Retourne un objet  
*SelectItem*

```
public class InputBean {  
    private String oneRadioValue;  
    public String getOneRadioValue() {  
        return oneRadioValue;  
    }  
    public void setOneRadioValue(String p) {  
        oneRadioValue = p;  
    }  
    public SelectItem getBananeItem() {  
        return new SelectItem("Banane");  
    }  
    ...  
}
```



## Composants graphiques : `<core:selectItems>`

- L'utilisation de la balise *core:selectItem* peut alourdir l'écriture des différents éléments des composants de sélection
- Cette balise est plutôt adaptée pour un nombre réduits d'éléments
- La balise *core:selectItems* allège donc l'écriture puisqu'une seule occurrence de *core:selectItems* remplace toutes les occurrences de *core:selectItem*
- L'attribut `value` est une valeur binding qui pointe vers une structure de type *SelectItem*
- La structure de type *SelectItem* peut être de type
  - instance unique
  - *map* : les entrées sont les *itemLabels* et *itemValues* de *SelectItem*
  - collection
  - tableau



# Composants graphiques : <core:selectItems>

## ➤ Exemple : choisir de nombreux éléments

Fruit préféré avec SelectItems :

- ☐ Banane
- ☐ Orange
- ☒ Clémentine *html:selectOneRadio*
- ☐ Pomme
- ☐ Franboise

Ajouter des périphériques :

- ☒ Clavier *html:selectManyCheckbox*
- ☐ Souris
- ☒ Ecran
- ☒ Unité centrale
- ☐ Haut parleur

Accessoire préféré :

- Montre
- Collier
- Bague *html:selectOneListbox*
- Bracelet
- Boucles d'oreille
- Chaussures à talon
- Pantalon

Choisir équipements préférés :

- Montre
- Collier
- Bague *html:selectManyListbox*
- Bracelet

Choisir une marque de voiture :

- Renault *html:selectOneMenu*

Choisir plusieurs marques de voiture :

- Renault
- Peugeot
- Citroën *html:selectManyMenu*
- Ford

# Composants graphiques : <core:selectItems>

## ➤ Exemple (suite) : choisir de nombreux éléments

```
<h4><html:outputText value="Fruit préféré avec SelectItems : " />
<html:selectOneRadio value="#{inputbean.oneRadioValue}" />
    <code:selectItems value="#{inputbean.oneRadioItems}" />
</html:selectOneRadio></h4>

<h4><html:outputText value="Ajouter des périphériques : " />
<html:selectManyCheckBox value="#{inputbean.manyCheckBoxValues}" />
    <code:selectItems value="#{inputbean.manyCheckBoxItems}" />
</html:selectManyCheckBox ></h4>

<h4><html:outputText value="Accessoire préféré : " />
<html:selectOneListBox value="#{inputbean.oneListBoxValues}" />
    <code:selectItems value="#{inputbean.manyAndOneListBoxItems}" />
</html:selectOneListBox></h4>

<h4><html:outputText value="Choisir équipements préférés : " />
<html:selectManyListBox value="#{inputbean.manyListBoxValues}" />
    <code:selectItems value="#{inputbean.manyAndOneListBoxItems}" />
</html:selectManyListBox></h4>

<h4><html:outputText value="Choisir une marque de voiture : " />
<html:selectOneMenu value="#{inputbean.oneMenuValue}" />
    <code:selectItems value="#{inputbean.manyAndOneMenuItems}" />
</html:selectOneMenu></h4>

<h4><html:outputText value="Choisir plusieurs marques de voiture : "
/>
<html:selectManyMenu value="#{inputbean.manyMenuValue}" />
    <code:selectItems value="#{inputbean.manyAndOneMenuItems}" />
</html:selectManyMenu></h4>
```

Simplification  
d'écriture avec la  
balise *selectItems*

Principe identique  
quelque soit le  
composant de  
sélection

Une seule valeur  
sélectionnable

Une seule valeur  
sélectionnable

*inputcomponent.jsp*

# Composants graphiques : <core:selectItems>

## ➤ Exemple (suite) : choisir de nombreux éléments

```
public class InputBean {
    private SelectItem[] manyCheckBoxItems = new SelectItem[] {
        new SelectItem("Clavier"),
        new SelectItem("Souris"),
        new SelectItem("Ecran"),
        new SelectItem("Unité Centrale"),
        new SelectItem("Haut Parleur")
    };

    private String[] manyCheckBoxValues;

    public SelectItem[] getManyCheckBoxItems() {
        return manyCheckBoxItems;
    }

    public String[] getManyCheckBoxValues() {
        return this.manyCheckBoxValues;
    }

    public void setManyCheckBoxValues(String[] p) {
        manyCheckBoxValues = p;
    }
}
```

Utilisées pour  
peupler les éléments  
du composant  
*selectManyCheckBox*

Utilisée pour  
afficher la valeur  
sélectionnée  
courante

Utilisée pour  
modifier la valeur  
sélectionnée  
courante

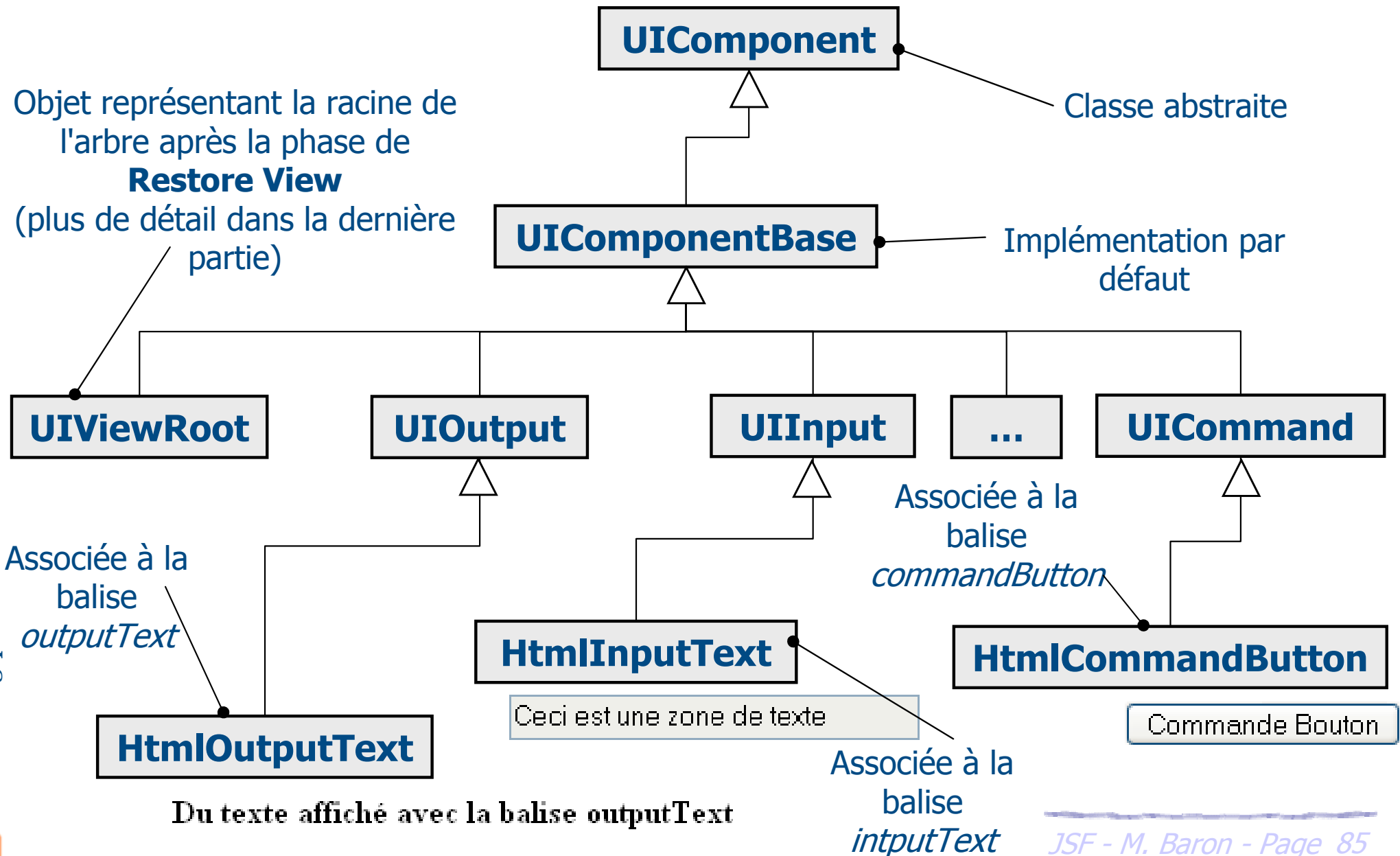
## Composants graphiques : API JSF

---

- Les classes qui modélisent l'état et le comportement d'un composant sont les classes principales où dépendent toutes les autres (rendu, description balises, événements ...)
- Les composants JSF héritent de la classe de plus haut niveau *UIComponent* qui est abstraite
- La classe *UIComponentBase* est la classe utilisée comme implémentation par défaut
- Rappelons qu'un composant JSF peut contenir un ensemble de sous composants (notion de hiérarchie)
- *UIComponentBase* fournit les services suivants
  - API pour la gestion du cycle de vie du composant
  - API pour le rendu du composant
  - ... (complété et détaillé dans la dernière partie)

# Composants graphiques : API JSF

## ➤ Hiérarchie des principaux composants JSF



## FacesContext : principe

---

- *FacesContext* permet de représenter toutes les informations contextuelles associées à la requête et à la réponse
- *FacesContext* est défini par une classe abstraite dont l'instanciation est réalisée (si pas créée) au début du cycle de vie
- Notez qu'il y a autant d'instances de type *FacesContext* qu'il y a de vues JSF
- Quand le processus de traitement est terminé, l'objet *FacesContext* libère les ressources. L'instance de l'objet est conservé (utilisation de la méthode *release()* )
- A chaque nouvelle requête, ré-utilisation des instances de *FacesContext*

## FacesContext : utilisation explicite

---

- *FacesContext* est essentiellement utilisé par les mécanismes internes de JSF. Toutefois il est possible d'en extraire des informations intéressantes
- Un objet *FacesContext* est exploitable dans un Bean ou dans une JSP via son objet implicite associé
- Que peut-on donc faire avec cet objet ?
  - *ExternalContext* : accéder aux éléments de la requête et de la réponse
  - *Message Queue* : stocker des messages (voir partie suivante)
  - *ViewRoot* : accéder à la racine de l'arbre des composants (*UIViewRoot*)
  - Modifier le déroulement du cycle de vie (voir partie événement)
  - Pleins d'autres choses qui sortent du cadre de ce cours ...

# FacesContext : utilisation explicite

## ► Exemple : manipuler l'objet *FacesContext*

```
package beanPackage;
public class FacesContextBean {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String pValue) {
        name = pValue;
    }
    public void apply() {
        ExternalContext context = FacesContext.getCurrentInstance().getExternalContext();
        System.out.println(context.getRequestPathInfo());
        System.out.println(context.getRequestServletPath());

        Iterator myIterator = context.getRequestParameterNames();
        while(myIterator.hasNext()) {
            Object next = myIterator.next();
            System.out.println("Nom du paramètre : " + next);
            Map requestParameterValuesMap = context.getRequestParameterValuesMap();
            Object tabParameter = requestParameterValuesMap.get((String)next);
            if (tabParameter instanceof String[]) {
                System.out.println("Valeur du paramètre : " + ((String[])tabParameter)[0]);
            }
        }
    }
}
```

*FacesContextBean.java*  
du projet  
***FacesContext***



# FacesContext : utilisation explicite

## ➤ Exemple (suite) : manipuler l'objet *FacesContext*

```
...
<body>
<core:view>
  <html:form>
    <html:inputText value="#{facescontextbean.name}" />
    <html:commandButton value="Valider" action="#{facescontextbean.apply}" />
  </html:form>
</core:view>
</body>
```

*facescontext.jsp*  
du projet  
***FacesContext***

Problems Javadoc Declaration Console X Properties Synchronize

Tomcat 6.x [Java Application] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (8 mai 07 22:18:05)

/facescontext.jsp  
/faces

Nom du paramètre : javax.faces.ViewState  
Valeur du paramètre : r00ABXVyABNbTGphdmEubGFuZy5PYmplY3Q7kM5YnxBzKWwCAAB4cAA.

Nom du paramètre : \_idJsp0:\_idcl  
Valeur du paramètre :

Nom du paramètre : \_idJsp0\_SUBMIT  
Valeur du paramètre : 1

Nom du paramètre : \_idJsp0:\_idJsp2  
Valeur du paramètre : Valider

Nom du paramètre : \_idJsp0:\_link\_hidden\_  
Valeur du paramètre :

Nom du paramètre : \_idJsp0:\_idJsp1  
Valeur du paramètre : MickaelBaron

FacesContext et ExternalCo...

MickaelBARON|

Valider

## Backing bean : principe

---

- Rappelons que les « Beans Managés » sont utilisés comme des modèles de vues
- L'accès aux « références » des composants de la vue ne peut se faire directement (assez tordu avec *FacesContext*)
- Il peut être intéressant d'accéder directement à la référence d'un composant graphique pour en modifier son état (à la suite d'un événement ou lors du processus de validation)
- Le framework JSF offre la possibilité d'encapsuler tout ou partie des composants qui composent une vue
- Il s'agit d'un type de Bean appelée « Backing Bean » qui correspond ni plus ni moins à un Bean managé
- Un Backing Bean est donc un bean dont certaines propriétés sont de type *UIComponent*

## Backing bean : principe

- Dans le Bean, il faut déclarer des accesseurs et des modifieurs sur des propriétés de type *UIComponent*

```
package beanPackage;
public class BackingBeanExemple {
    private String nom;
    private UICommand refCommand;

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

    public void setNomAction(UICommand ref) {
        refCommand = ref;
    }
    public UICommand getNomAction() {
        return refCommand;
    }
}
```

Le « Backing Bean »  
se comporte comme  
un Bean Managé

Stocker et relayer la  
référence d'un  
composant JSF

- Au niveau de la page JSP, la liaison entre le « Backing Bean » est la vue se fait par l'attribut *binding*

```
<html:commandButton binding="#{backingbean.nomAction}" />
```

# Backing bean : principe

## ► Exemple : activer/désactiver un bouton par un Bean

```
...  
<body>  
<core:view>  
  <html:form>  
    <html:inputText value="#{backingbean.name}" binding="#{backingbean.composantNom}" />  
    <html:commandButton value="Transformer" binding="#{backingbean.commandButton}"  
      action="#{backingbean.doProcess}" />  
  </html:form>  
</core:view>  
</body>
```

Baron

Une valeur dans un  
composant texte

Transformer

Un bouton activé

*backingbean1.jsp*  
du projet  
**BackingBean**

Un bouton  
désactivé

Nouvelle Valeur

La valeur a changé (sans  
passer par le Bean)

Transformer

# Backing bean : principe

## ► Exemple (suite) : activer/désactiver un bouton par un Bean

```
package beanPackage;
public class BackingBean {
    private String nom = "Baron";
    private HtmlInputText composantNom;
    private HtmlCommandButton commandButton;

    public String getName() {return name; }
    public void setName(String pName) {this.name = pName; }

    public void setComposantNom(HtmlInputText pCommand) {composantNom = pCommand; }
    public HtmlInputText getComposantNom() { return composantNom; }

    public void setcommandButton(HtmlCommandButton pCB) {this.commandButton = pCB; }
    public HtmlCommandButton getCommandButton() { return this.commandButton; }

    public void doProcess() {
        if (commandButton != null) {
            this.commandButton.setDisabled(true);
        }
        if (composantNom != null) {
            composantNom.setValue("Nouvelle Valeur");
        }
    }
}
```

*BackingBean.java*  
du projet  
**BackingBean**

## Message : manipulation et affichage

---

- Pour l'instant avec un Bean managé il est possible de
  - Stocker et afficher les valeurs d'un formulaire
  - Générer les actions pour l'automate de navigation
- L'API JSF fournit l'objet ***FacesMessage*** pour empiler des messages qui pourront être afficher dans une page JSP
- Un objet *FacesMessage* est caractérisé par :
  - **Sévérité** : *SEVERITY\_INFO*, *SEVERITY\_WARN*, *SEVERITY\_ERROR* et *SEVERITY\_FATAL*
  - Résumé : texte rapide décrivant le message
  - Détail : texte détaillé décrivant le message
- L'API fournit des modifieurs et accesseurs
  - *setSeverity(FacesMessage.Severity)* / *getSeverity()*
  - *setSummary(String)* / *getSummary()*
  - *setDetail(String)* / *getDetail()*

## Message : manipulation et affichage

- Un objet *FacesMessage* est donc construit dans une classe Java (un Bean par exemple)
- Pour exploiter des messages dans une page JSP il faut passer par le contexte courant JSF (*FacesContext*)
  - *addMessage(String id, FacesMessage message)* : ajoute un message à un composant (défini par *id*)
  - Si *id* est *null* le message n'est pas associé à un composant
- Exemple de construction et transmission d'un *FacesMessage* :

```
// Déclaration d'un Message
FacesMessage myFacesMessage = new FacesMessage();
myFacesMessage.setSeverity(FacesMessage.SEVERITY_INFO);
myFacesMessage.setSummary("Un petit résumé");
myFacesMessage.setDetail("Mon message qui ne sert à rien");

// Transmission d'un Message
FacesContext myFacesContext = FacesContext.getCurrentInstance();
myFacesContext.addMessage(null, myFacesMessage);
```

## Message : manipulation et affichage

---

- La bibliothèque HTML propose deux balises personnalisées pour afficher les messages
  - `<html:messages>` : affiche tous les messages
  - `<html:message>` : affiche les messages associés à un *id* de composant
- Les balises contiennent les attributs suivants :
  - *for* : indique l'id du composant (uniquement pour message)
  - *showDetail* : booléen qui précise si le message est détaillé
  - *showSummary* : booléen qui précise si le message est résumé
  - *tooltip* : booléen qui précise si une bulle d'aide est affichée
  - *layout* : précise la manière d'afficher les messages. Valeurs possibles *table* ou *list* (par défaut)



# Message : manipulation et affichage

## ➤ Exemple : afficher les messages en un seul bloc

```
public class BeanMessages {
    private String name;
    public String getName() { return name; }
    public void setName(String p) { this.name = p; }

    public void validAction() {
        FacesMessage facesMessage = new FacesMessage();
        facesMessage.setSeverity(FacesMessage.SEVERITY_INFO);
        facesMessage.setSummary("Validation");
        if (name == null || name.equals("")) {
            facesMessage.setDetail("(nom inconnu)");
        } else {
            facesMessage.setDetail("(nom connu : " + name + ")");
        }

        FacesContext facesContext = FacesContext.getCurrentInstance();
        facesContext.addMessage(null, facesMessage);
        facesContext.addMessage(null, new FacesMessage(
            FacesMessage.SEVERITY_INFO, "Validation 2", "Une seconde ligne"));
    }
    ...
}
```

*BeanMessages* du  
projet **Messages**

# Message : manipulation et affichage

## ➤ Exemple (suite) : afficher les messages en un seul bloc

```
<%@taglib prefix="core" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="html" uri="http://java.sun.com/jsf/html"%>

<html>
  ...
  <body>
    <core:view>
      <html:form>
        <html:messages showDetail="true" layout="table" showSummary="true"/><br>
        <html:outputText value="Nom : " />
        <html:inputText value="#{beanmessages.name}" /><br>
        <html:commandButton value="Valider" action="#{beanmessages.validAction}" />
        <html:commandButton value="Annuler" action="#{beanmessages.cancelAction}" />
      </html:form>
    </core:view>
  </body>
</html>
```

Affichage sous la  
forme d'un tableau

Formulaire pour la gestion ...

Validation (nom connu :Baron Mickael)

Validation 2 Une ligne pour rien

Nom :

Valider

Annuler

*form1.jsp* du projet  
**Messages**

# Message : manipulation et affichage

## ➤ Exemple : afficher les message en plusieurs blocs

```
<%@taglib prefix="core" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="html" uri="http://java.sun.com/jsf/html"%>

<html>
  ...
  <body>
    <core:view>
      <html:form id="logonForm">
        <html:outputText value="Nom :"/>
        <html:inputText id="myIdName" value="#{beanmessage.name}"/><br>
        <html:message for="myIdName" tooltip="true" showDetail="true"
          showSummary="true"/><br>

        <html:outputText value="Email :"/>
        <html:inputText id="myIdEmail" value="#{beanmessage.email}"/><br>
        <html:message for="myIdEmail" tooltip="true" showDetail="true"
          showSummary="true"/><br>

        <html:commandButton value="Valider" action="#{beanmessage.validAction}"/>
        <html:commandButton value="Annuler" action="#{beanmessage.cancelAction}"/>
      </html:form>
    </core:view>
  </body>
</html>
```

**Nécessaire pour identifier les sous composants**

**Identifiant du composant**  
*inputText*

**form2.jsp du projet Messages**

**Précise que le message est associé à l'id myname**

**Les identifiants sont utilisés pour « localiser » les messages**

# Message : manipulation et affichage

## ➤ Exemple (suite) : afficher les message en plusieurs blocs

```
public class BeanMessage {  
    private String name;  
    private String email;  
  
    public String getName() { return name; }  
    public void setName(String p) { this.name = p; }  
  
    public String getEmail() { return email; }  
    public void setEmail(String p) { this.email = p; }  
  
    public void validAction() {  
        FacesContext facesContext = FacesContext.getCurrentInstance();  
        facesContext.addMessage("logonForm:myIdName", new  
            FacesMessage(FacesMessage.SEVERITY_INFO, "Validation Name", "Nom  
            saisi " + this.name));  
        facesContext.addMessage("logonForm:myIdEmail", new  
            FacesMessage(FacesMessage.SEVERITY_INFO, "Validation Email",  
            "Email saisi " + this.email));  
    }  
    ...  
}
```

*myIdName* et  
*logonForm* sont des  
identifiants de  
composants

Précise que *myIdName*  
appartient à *logonForm*

*BeanMessage.java* du  
projet **Messages**

Formulaire pour la gestion ...

Nom :

Validation Name Nom saisi Baron

Email :

Validation Email Email saisi toto@free.fr

## Message : Internationalisation

- *faces-config.xml* permet de configurer les localisations acceptées pour l'application WEB dans la balise *<application>*
- Dans *<information>* deux informations sont à renseigner
  - *<message-bundle>* : la ressource contenant les messages localisés
  - *<local-config>* : indique la locale par défaut et les locales autorisées
- Balise *<local-config>* deux informations sont à renseigner
  - *<default-locale>* : la localisation par défaut
  - *<supported-locale>* : la localisation supportée par l'application

```
...  
<application>  
  <message-bundle>resources.ApplicationMessage</message-bundle>  
  <locale-config>  
    <default-locale>en</default-locale>  
    <supported-locale>fr</supported-locale>  
    <supported-locale>sp</supported-locale>  
  </locale-config>  
</application>
```

*faces-config.xml*

## Message : Internationalisation

- Dans la version 1.2 possibilité de déclarer des variables qui pointent sur des fichiers properties *<resource-bundle>*
- Dans *<resource-bundle>* deux informations sont à indiquer
  - *<base-name>* : nom de la ressource contenant les messages localisés
  - *<var>* : alias permettant d'accéder à la ressource *<base-name>*

```
<application>
  <resource-bundle>
    <base-name>resources.ApplicationMessage</base-name>
    <var>bundle</var>
  </resource-bundle>
</application>
```

- Dans la version 1.1 le chargement des ressources est effectué dans la page JSP à l'aide de la balise *<core:loadbundle>*

```
<core:loadBundle basename="resources.ApplicationMessage" var="bundle" />
```

- Les ressources sont exploitées par l'intermédiaire des alias (attribut *var*) en utilisant les EL

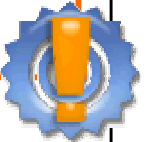
# Message : Internationalisation

## ➤ Exemple : formulaire utilisant une ressource de localisation

```
<%@page contentType="text/html"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="core"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="html"%>

<html>
  <head>
    <title>Formulaire JSF avec un Bean Managé</title>
  </head>
  <body>
    <core:view>
      <core:loadBundle basename="CoursesMessages" var="bundle" />
      <html:form>
        <html:outputText value="#{bundle.EMAIL_ADRESS_MESSAGE}" />
        <html:inputText value="#{registrationbean.email}" /><br>
        <html:outputText value="#{bundle.PASSWORD_MESSAGE}" />
        <html:inputSecret value="#{registrationbean.password}" /><br>
        <html:commandButton value="#{bundle.LOGIN_MESSAGE}" />
      </html:form>
    </core:view>
  </body>
</html>
```

**Solution pour  
la version 1.1  
de JSF**



*CoursesMessages\_fr.xml*

EMAIL\_ADRESS\_MESSAGE = Adresse Email  
PASSWORD\_MESSAGE = Mot de Passe  
LOGIN\_MESSAGE = Connecter

*managedbeanviewbundle.jsp*

# Message : Internationalisation

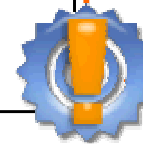
- Exemple (bis) : formulaire qui utilise une ressource pour la localisation version JSF 1.2

```
<html>
  ...
  <body>
    <core:view>
      <html:form>
        <html:outputText value="#{bundle.EMAIL_ADRESS_MESSAGE}"/>
        <html:inputText value="#{registrationbean.email}"/><br>
        <html:outputText value="#{bundle.PASSWORD_MESSAGE}" />
        <html:inputSecret value="#{registrationbean.password}"/><br>
        <html:commandButton value="#{bundle.LOGIN_MESSAGE}" />
      </html:form>
    </core:view>
  </body>
</html>
```

```
<application>
  <resource-bundle>
    <base-name>resources.ApplicationMessage</base-name>
    <var>bundle</var>
  </resource-bundle>
</application>
```

*faces-config.xml*

**Solution pour  
la version 1.2  
de JSF**





# Conversion de données : principe

---

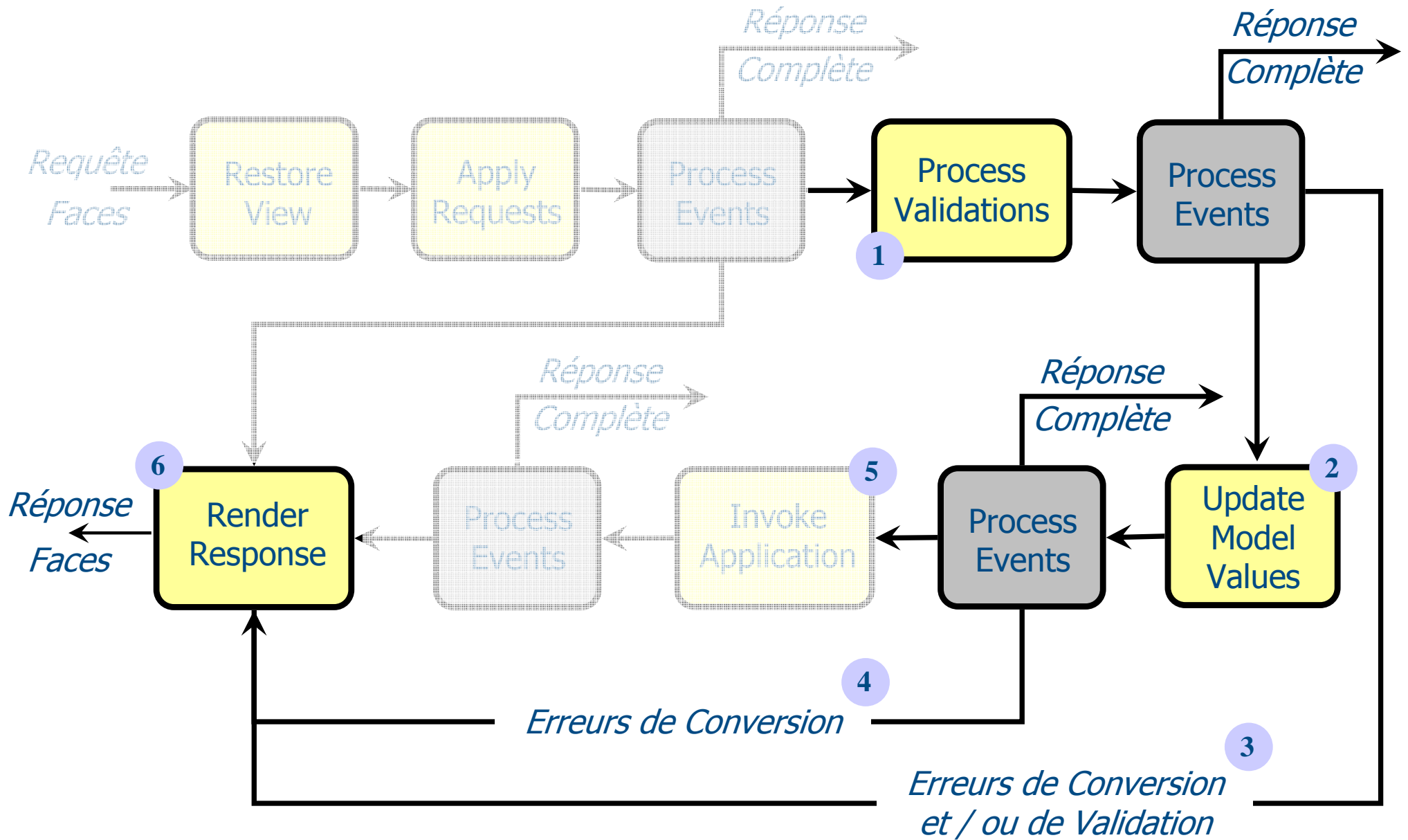
- Conversion des données du modèle et de la vue
  - les paramètres d'une requête sont de type chaînes de caractères
  - les Beans sont définis par des attributs représentés par des types de données (*int*, *long*, *Object*, ...)
  - la conversion de données s'assurent alors qu'une chaîne de caractères peut être traduit en un type de donnée précis
- Respect du formatage en entrée et en sortie
  - Formatage en sortie : affichage d'une date
    - Afficher la date suivant un format compréhensible par l'utilisateur (fonction de la localisation et du type d'utilisateur)
  - Conformité en entrée : saisie d'une date
    - Associé à un composant de type saisie
    - Assurer que l'utilisateur a saisi une chaîne de texte respectant le format de l'entrée

## Conversion de données : principe

---

- Les convertisseurs de données sont appelés **Converters**
- Tous les composants **affichant une valeur** et utilisés lors de **la saisie** fournissent une conversion de données
- Les converters sont utilisables dans une JSP via une balise
- Pour utiliser un converter :
  - soit utiliser les standards fournis par la bibliothèque JSF dans une JSP
  - soit le définir programmatiquement (dans un Bean) et l'utiliser dans une JSP
- Converters standards de la bibliothèque JSF :
  - *core:convertDateTime* : pour la conversion de date et d'heure
  - *core:convertNumber* : pour la conversion de nombre

# Conversion de données : cycle de vie



## Conversion de données : cycle de vie

---

- Dans le cas d'un composant de la famille de type *UIInput* (par exemple un composant *html:inputText*)
  - 1 L'étape **Process Validations** réceptionne les valeurs des paramètres (*submittedValues*) et applique les conversions
  - 2 Si la conversion a réussi, le traitement continu
    - L'étape **Update Model Values** modifie le modèle associé (un Bean par exemple) puis conversion des valeurs du modèle
  - 5 Si la conversion a réussi, continuer processus ...
  - 4 Si la conversion a échoué, génération de messages d'erreurs et direction étape **Render Response** 6
  - 3 Si la conversion a échoué, le traitement est abandonné (pas de changement de page) et des messages d'erreur sont générés et direction étape **Render Response** 6

## Conversion de données : cycle de vie

- Dans le cas d'un composant de la famille de type *UIOutput* (par exemple composant *html:outputText*)
  - 1 L'étape **Process Validations** et l'étape **Update Model Values** ne traitent pas de paramètres car il n'y en a pas ...
  - 2 Puis conversion des attributs du modèle présents dans le Bean
    - 5 Si la conversion a réussi, continuer processus ...
    - 4 Si la conversion a échoué, génération de messages d'erreur et direction étape **Render Response** 6

**La seconde conversion consiste à s'assurer de la cohérence des données du modèle et du format de conversion**



## Conversion de données : cycle de vie (via l'API)

---

- Tous composants qui implémentent l'interface *ValueHolder* peuvent exploiter des Converters (*UIOutput* et *UIInput*)
- L'implémentation de l'interface *ValueHolder* doit fournir des propriétés pour :
  - Effectuer la conversion via un objet *Converter*
  - Stocker la valeur convertie par l'objet *Converter*
  - Stocker la valeur propre du composant utilisée pour modifier le modèle de la vue et fournir une donnée à la réponse

**Ensemble de méthodes  
(accesseurs et modifieurs) pour la  
manipulation de ces propriétés**



# Conversion de données : cycle de vie (via l'API)

- Un objet *Converter* fournit deux méthodes :
  - *Object getAsObject(...)* throws *ConverterException* appelée lors de l'étape Process Validation
  - *String getAsString(...)* throws *ConverterException* appelée après l'étape Update Model Values
- Dans le cas des objets *UIInput* les valeurs des paramètres sont stockées dans l'attribut *submittedValue* (interface *EditableValueHolder*)

**Nous décrirons la classe *Converter* au moment de la création de notre propre *converter***



Par son intermédiaire la phase **Process Validations** peut récupérer les valeurs qui étaient issues des paramètres

# Conversion de données : cycle de vie (via l'API)

---

- Processus de conversion par l'API
  - *submittedValue* est convertie par *getAsObject(...)* et stockée dans *localValue* (si exception de *getAsObject(...)* arrêt du processus)
  - Si *localValue*  $\neq$  null, sa valeur est transmise à *value*
    - *value* transmet la valeur à une propriété du modèle de la vue (via un modifieur du Bean)
    - *value* récupère la valeur d'une propriété du modèle de la vue (via un accesseur du Bean)
  - Si *localValue*  $==$  null, *value* récupère la valeur d'une propriété du modèle
  - *value* est convertie par *getString(...)* et envoyée à la réponse (si exception de *getString(...)* arrêt du processus)



## Conversion de données : tag `convertNumber`

---

- La balise `core:convertNumber` permet la conversion de chaînes de caractères en valeur numérique
- À utiliser dans le corps d'une balise graphique JSF
- Différents attributs disponibles
  - *type* : type de valeur (*number*, *currency* ou *percent*)
  - *pattern* : motif de formatage
  - *(max/min)FractionDigits* : (int) nombre maxi/mini sur la décimale
  - *(max/min)IntegerDigits* : (int) nombre maxi/mini sur la partie entière
  - *integerOnly* : (booléen) seule la partie entière est prise en compte
  - *groupingUsed* : (booléen) précise si les caractères de regroupement sont utilisés (exemple : « , », « ; », « : », ... )
  - *locale* : définit la localisation de la conversion
  - *currencyCode* : code de la monnaie utilisée pour la conversion
  - *currencySymbol* : spécifie le caractère (exclusif avec *currencyCode*)

# Conversion de données : tag convertNumber

## ➤ Exemple : utilisation d'un converter en « sortie »

```
<html>
...
<body>
<core:view>
  <html:form>
    <html:outputText value="#{beanconverter1.price}">
      <core:convertNumber type="currency" currencyCode="EUR"
        minFractionDigits="4" />
    </html:outputText>
  </html:form>
</core:view>
</body>
</html>
```

*converters1.jsp* du  
projet **Converters**

Utilisation d'un Converter e...

**50,5000 EUR**

```
public class BeanConverter1 {
    private double price;

    public BeanConverter1() {
        this.price = 50.50d;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double pPrice) {
        this.price = pPrice;
    }
}
```

*BeanConverter1* du  
projet **Converters**

# Conversion de données : tag convertNumber

## ► Exemple (suite) : utilisation d'un converter en « entrée »

```
<html>
  ...
  <body>
    <core:view>
      <html:form>
        Prix :
        <html:inputText value="#{beanconverter1.price}" >
          <core:convertNumber minFractionDigits="2" />
        </html:inputText><br>
        <html:commandButton value="Envoyer" />
      </html:form>
    </core:view>
  </body>
</html>
```

Utilisation d'un Converter e...

**Prix :**

Envoyer

Avant la conversion, au moment de la saisie ...

Après la conversion

Utilisation d'un Converter e...

**Prix :**

Envoyer

*converters2.jsp* du projet **Converters**

## Conversion de données : tag `convertDateTime`

- La balise `core:convertDateTime` permet la conversion de chaînes de caractères en date ou heure
- À utiliser également dans le corps d'une balise graphique JSF

```
<html:outputText value="#{packageBean.facturedate}">  
  <core:convertDateTime type="date" dateStyle="long"/>  
</html:outputText>
```

- Différents attributs disponibles
  - *type* : type de valeur (*date*, *time* ou *both*)
  - *dateStyle* : style possible de la date (*short*, *medium*, *long*, *full*, *both*)
  - *timeStyle* : style possible de l'heure (*short*, *medium*, *long*, *full*, *both*)
  - *Pattern* : motif utilisé pour une *Date*
  - *Locale* : choix de la localisation

# Conversion de données : tag convertDateTime

## ➤ Exemple : utilisation d'un converter date en « entrée »

```
...
<body>
<core:view>
  <html:form>
    <html:outputText value="Nom : " />
    <html:inputText value="#{beanconverter2.name}" /><br>
    <html:outputText value="Nombre de tickets : " />
    <html:inputText value="#{beanconverter2.tickets}" >
      <core:convertNumber type="number" integerOnly="true" />
    </html:inputText><br>
    <html:outputText value="Date : " />
    <html:inputText value="#{beanconverter2.date}">
      <core:convertDateTime type="date" dateStyle="medium" />
    </html:inputText>
    <html:commandButton value="Valider" />
  </html:form>
</core:view>
</body>
</html>
```

*converters3.jsp* du  
projet **Converters**

**Nom :**

**Nombre de tickets :**

**Date :**

# Conversion de données : tag convertDateTime

## ➤ Exemple (suite) : utilisation d'un converter *Date* en « entrée »

```
public class BeanConverter2 {  
    private long tickets;  
    private String name;  
    private Date date;  
  
    public BeanConverter2() {  
        tickets = 50;  
        name = "Mon Nom";  
        date = new Date();  
    }  
  
    public long getTickets() { return tickets; }  
    public void setTickets(long pTickets) { tickets = pTickets; }  
  
    public String getName() { return name; }  
    public void setName(String pName) { name = pName; }  
  
    public Date getDate() { return date; }  
    public void setDate(Date pDate) { date = pDate; }  
}
```

*BeanConverter2.java*  
du projet **Converters**

## Conversion de données : messages

- Dans le cas où des erreurs de conversion sont provoquées, des messages sont ajoutés au *FacesContext*
- En utilisant les balises *html:message* et *html:messages* possibilité d'afficher un message précis ou l'ensemble
- Généralement à chaque composant est associé un id, ce qui permettra de localiser précisément l'origine de l'erreur
- Exemple :

*Converters2.jsp* du projet **Converters**

```
...
<html:outputText value="Prix : " />
<html:inputText id="priceId" value="#{beanconverter1.price}" >
    <core:convertNumber minFractionDigits="2" />
</html:inputText><html:message for="priceId" /><br>
<html:commandButton value="Envoyer" />
...
```

# Conversion de données : messages

## ➤ Exemple : convertir date en « entrée » avec messages

```
...
<core:view>
  <html:form>
    <html:outputText value="Nom : " />
    <html:inputText value="#{beanconverter2.name}" /><br>
    <html:outputText value="Nombre de tickets : " />
    <html:inputText id="ticketsId" value="#{beanconverter2.tickets}" >
      <core:convertNumber type="number" integerOnly="true" />
    </html:inputText><html:message for="ticketsId" /><br>
    <html:outputText value="Date : " />
    <html:inputText id="dateId" value="#{beanconverter2.date}">
      <core:convertDateTime type="date" dateStyle="medium" />
    </html:inputText><html:message for="dateId" /><br>
    <html:outputText value="Liste des messages : " /><br>
    <html:messages />
  </html:form>
</core:view>
```

*converters3.jsp* du  
projet **Converters**

Utilisation d'un Converter e...

Nom :

Nombre de tickets :  "ticketsId": La donnée n'est pas un nombre valide.

< / Date :  "dateId": Conversion en Date/Heure impossible.

- Problème pour le champs "Nombre de tickets"
- Problème pour le champs "Date"

Valider



## Conversion de données : messages personnalisés

---

- Les messages sont déclarés par des clés stockées dans un fichier ressource dont le nom dépend de la localisation
- Les fichiers ressources (fichier au format properties) sont stockés dans le répertoire *javax.faces* de la librairie implémentation de JSF (*myfaces-impl-1.x.x.jar*)
- Pour modifier un message pour un convertir donné
  - Créer un fichier properties dans le répertoire source
  - Déclarer le fichier properties dans le faces-config.xml
  - Copier les clés à redéfinir
  - Modifier la valeur du texte dans les clés

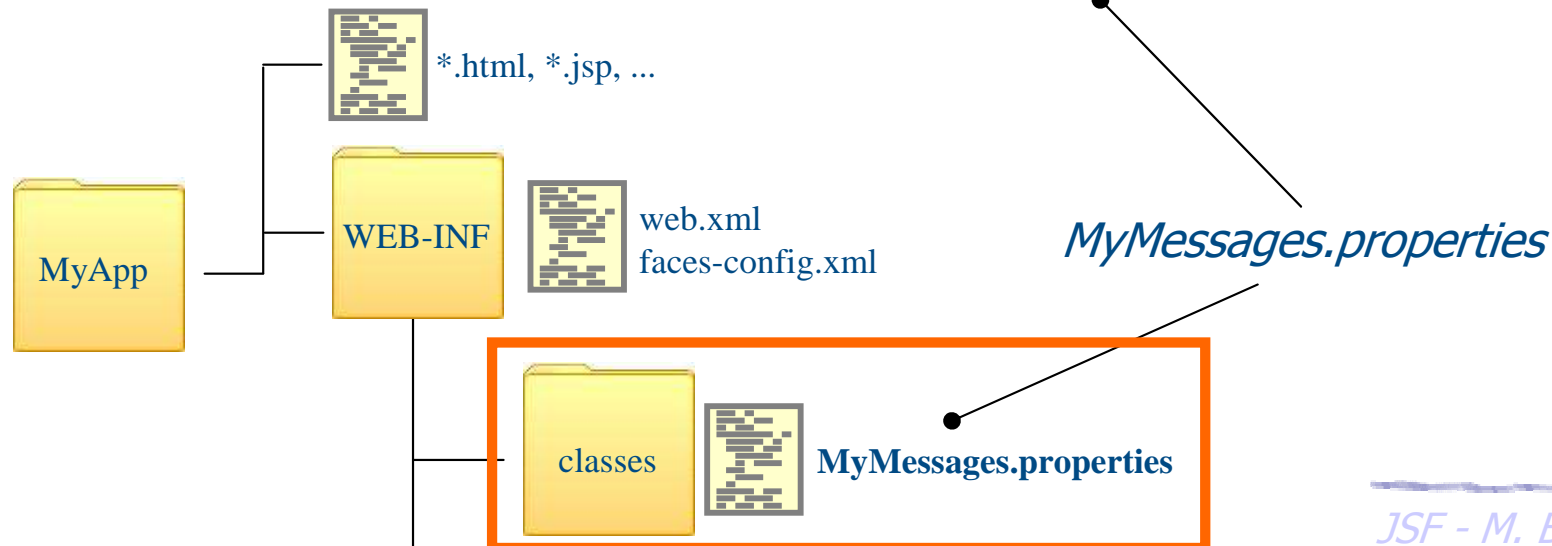
# Conversion de données : messages personnalisés

## ➤ Exemple : modification des messages d'erreurs de conversion

```
...  
<application>  
    <message-bundle>beanPackage.MyMessages</message-bundle>  
</application>  
</faces-config>
```

*faces-config.xml* du  
projet **Converters**

```
...  
javax.faces.convert.NumberConverter.CONVERSION = Problème pour le champs  
    "Nombre de tickets"  
javax.faces.convert.NumberConverter.CONVERSION_detail = "{0}" : La donnée  
    n'est pas un nombre valide  
...
```



## Conversion de données : attribut *immediate*

- Rappelons que toute requête JSF passe obligatoirement par un traitement défini par le cycle de vie JSF
- La conversion de données est obligatoirement traitée par le cycle de vie JSF même si par exemple une requête donnée n'est pas de valider un formulaire (annuler le traitement)
- Si une requête est d'annuler la saisie du formulaire (redirection vers une autre page), le processus effectuera toujours la conversion de données de tous les composants d'une vue JSF
- Il peut donc être utile de passer outre le processus de conversion : utilisation de l'attribut *immediate* pour une balise de composant qui doit soumettre le formulaire
- La présence de l'attribut *immediate* (valeur à *true*) modifie le déroulement du cycle de vie d'une requête JSF

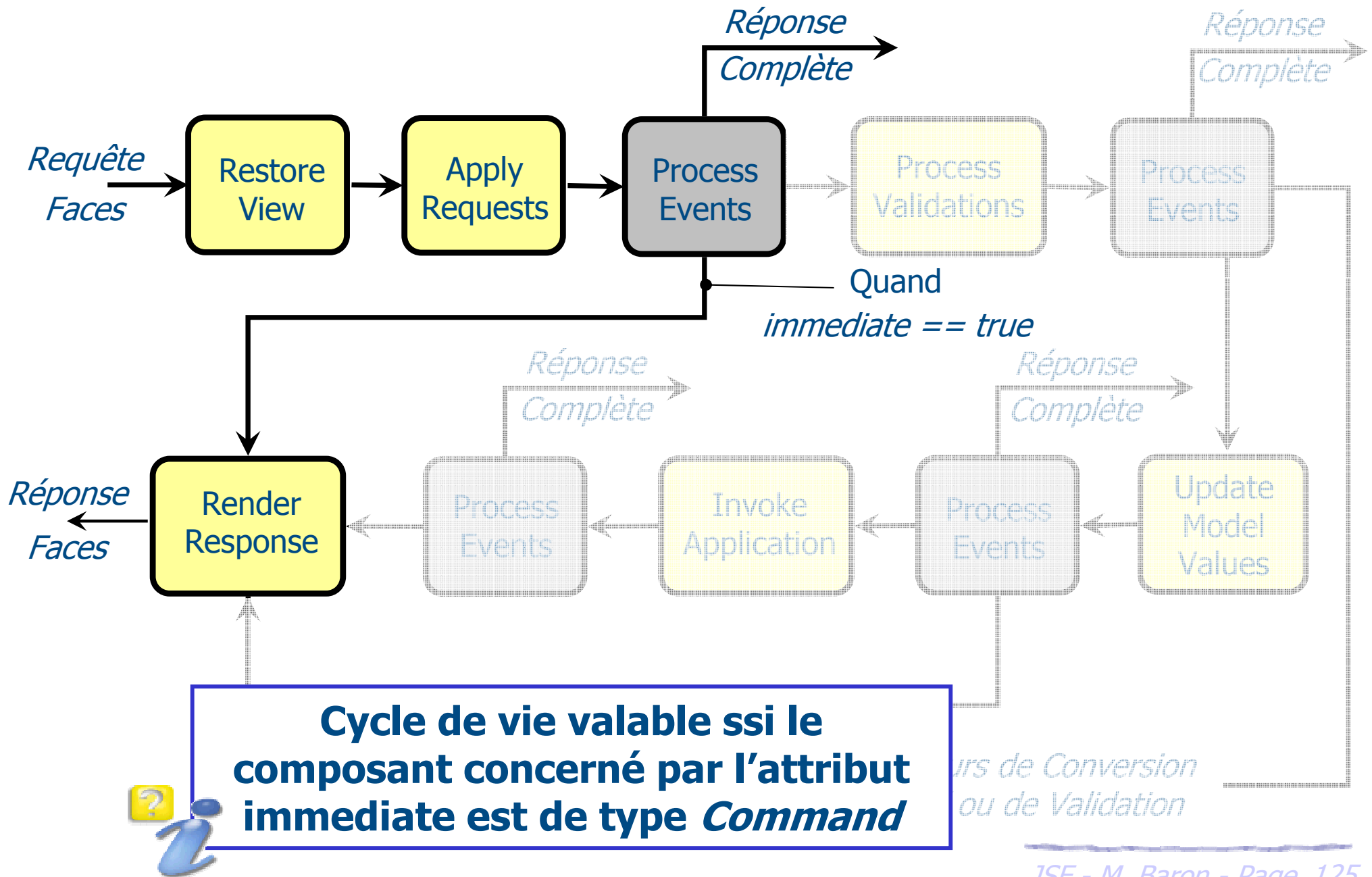
## Conversion de données : attribut *immediate*

- Le composant passé à *immediate* sera géré en premier dans le cycle de vie
- Le fonctionnement donné ci-dessous ne concerne que les composants de type *Command* (*CommandButton* par exemple)
  - Après la phase **Apply Request Values**, le traitement de la requête va être redirigé directement vers **Render Response**
  - De cette manière les phases **Process Validations**, **Update Model Values**, **Invoke Application** ne sont pas exécutées
- Il n'y a donc pas de conversion, ni de modification dans le modèle
- Les différentes saisies de l'utilisateur ne sont jamais traitées et il n'y a donc pas d'erreurs qui pourraient bloquer l'utilisateur

**Une explication détaillée de l'effet de bord de *immediate* (composant *Input* et *Command*) est donnée dans la partie événement**



# Conversion de données : attribut *immediate*



# Conversion de données : attribut *immediate*

## ➤ Exemple : passer outre le processus de conversion

Nom :

Nombre de tickets :

Date :

Liste des messages :

Valider

Annuler (sans immediate)

Annuler (avec immediate)

Les problèmes de conversion ne sont pas pris en compte

### Exemples des Converters

[Utilisation d'un Converter en sortie \(currency\)](#)

[Utilisation d'un Converter en entrée \(currency\)](#)

[Utilisation d'un Converter en entrée/sortie \(currency et date\)](#)

[Utilisation d'un Converter en entrée/sortie \(currency et date\) avec possibil](#)

Nom :

Nombre de tickets :

Date :  "dateId": Conversion en D

Liste des messages :

- Problème pour le champs "Date"

Les problèmes de conversion sont pris en compte. Obligation de saisir correctement les données pour annuler

Valider

Annuler (sans immediate)

Annuler (avec immediate)

# Conversion de données : attribut *immediate*

## ➤ Exemple (suite) : passer outre le processus de conversion

```
...
<core:view>
  <html:form>
    ...
    <html:outputText value="Date : " />
    <html:inputText id="dateId" value="#{beanconverter2.date}">
      <core:convertDateTime type="date" dateStyle="medium" />
    </html:inputText><html:message for="dateId" /><br>
    <html:outputText value="Liste des messages : " /><br>
    <html:messages />
    <html:commandButton value="Valider" />
    <html:commandButton value="Annuler (sans immediate)" action="CancelAction" />
    <html:commandButton value="Annuler (avec immediate)" action="CancelAction"
      immediate="true" />
    </html:form>
  </core:view>
```

*convertersImmadiate.jsp*  
du projet **Converters**

*faces-config.xml* du  
projet **Converters**

```
<faces-config>
  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>CancelAction</from-outcome>
      <to-view-id>/index.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  ...
</faces-config>
```

# Conversion de données : conversion personnalisée

---

- Pour bientôt ...
  - Réalisation de son propre *Converter*



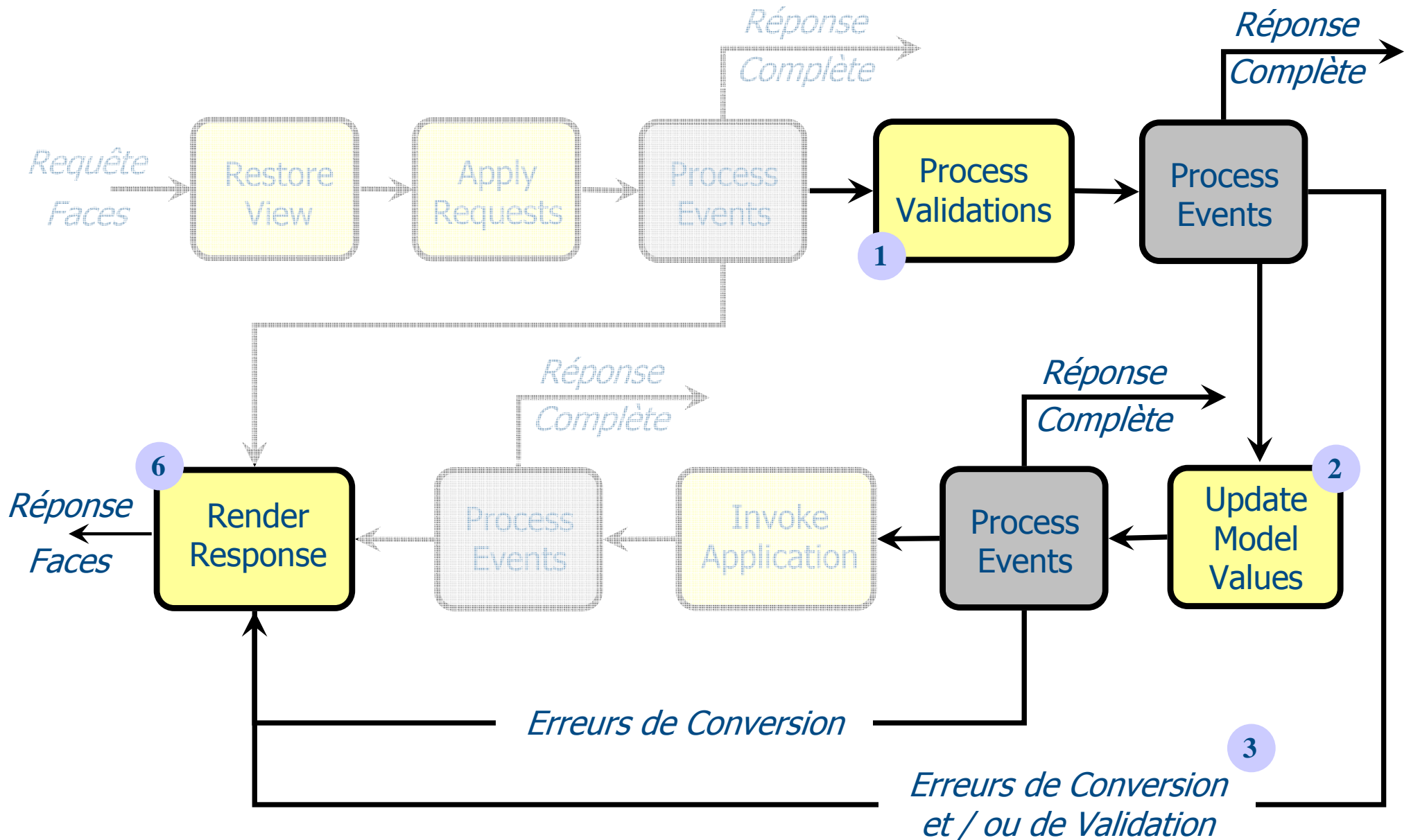


# Validation de données : principe

---

- L'objectif de la validation de données est de s'assurer que les informations qui ont été converties sont valides
- Les objets utilisés pour la validation sont appelés **Validators**
- Tous les composants utilisés pour la saisie de données peuvent exploiter des Validators
- Pourquoi utiliser des Validators (vérification de surface et de fond) ?
  - vérifier la présence de données
  - vérifier le contenu d'une données
  - vérifier la plage de valeurs (entier compris entre 20 et 50)
  - vérifier le format d'une donnée
  - ...

# Validation de données : cycle de vie



# Validation de données : cycle de vie

- Composant de la famille de type *UIInput* (déroulement proche du processus de conversion)
  - 1 L'étape **Process Validations** réceptionne les valeurs des paramètres (*submittedValues*), applique les conversions et validations
  - 2 Si la conversion et/ou la validation ont réussi, le traitement continu
    - Voir partie conversion pour détail de la suite ...
  - Si la conversion et/ou la validation ont échoué, le traitement est abandonné (pas de changement de page) et des messages d'erreur sont générés et direction étape **Render Response** 6



**De nombreux détails ont été repris de la partie Converters, pour les aspects manquants (message, immediate ) s'y référer**

## Validation de données : cycle de vie (via l'API)

---

- Les composants implémentant l'interface *EditableValueHolder* peuvent exploiter des Validators (*UIInput*)
  - *EditableValueHolder* est une sous interface de *ValueHolder*
  - Un objet *EditableValueHolder* fourni les propriétés pour ajouter un objet *Validator* et stocker l'état de validation
  - Un objet *Validator* fournit une méthode
    - *validate(FacesContext ct, UIComponent component, Object value)* throws *ValidatorException* : en charge d'effectuer la validation
- Note** : s'il existe une erreur de validation, lancer une exception de type *ValidatorException*

# Validation de données : principe d'utilisation

---

- Pour utiliser un Validator ...
  - soit utiliser les standards fournis par la bibliothèque JSF
  - soit définir programmatiquement (dans un Bean)
- Utiliser les standards fournis
  - *core:validateDoubleRange* : valide les données de type *Double*
  - *core:validateLongRange* : valide les données de type *Long*
  - *core:validateLength* : valide la longueur de type *Integer*
- Dans tous les cas, ces balises fournissent trois attributs
  - *maximum* : valeur maximale
  - *minimum* : valeur minimale
  - *binding* : évalue une instance du validator en question

# Validation de données : validators standards

## ➤ Exemple : un formulaire, des champs et des validators ...

```
...
<html:form>
  <p>
    <html:outputText value="Veuillez saisir votre numéro de compte (10
      chiffres) : " />
    <html:inputText id="compteId" value="#{validatorbean.compte}" >
      <core:validateLength maximum="10" minimum="10" />
    </html:inputText><html:message for="compteId" />
  </p>
  <p>
    <html:outputText value="Veuillez saisir votre taille (comprise entre 1.2 et
      2.0) : " />
    <html:inputText id="tailleId" value="#{validatorbean.taille}" >
      <core:validateDoubleRange maximum="2.80" minimum="1.20"/>
    </html:inputText><html:message for="tailleId" /><br>
    <html:commandButton value="Envoyer" />
  </p>
</html:form>
...
```

Veuillez saisir votre numéro de compte (10 chiffres) :

Veuillez saisir votre taille (comprise entre 1.2 et 2.0) :

Envoyer

"tailleId": La donnée n'est pas comprise entre 1,2 et 1,8.

*standardvalidator.jsp* du  
projet **Validators**

# Validation de données : validators personnalisés

---

- Nous avons vu que JSF proposait en standard un ensemble restreint de Validators
- Il peut être intéressant de pouvoir développer ses propres Validators
- JSF offre la possibilité de définir programmatiquement les validators souhaités en utilisant des classes Java
- Deux approches de mise en œuvre sont à distinguer :
  - Utilisation de la balise `<core:validator>`
  - Utilisation de l'attribut `validator`
- Rappelons que la mise en place de validators ne concerne que les composants de type *UIInput*

# Validation de données : validators personnalisés

---

- L'utilisation d'une balise pour lier un validator à un composant passe par l'implémentation de l'interface *Validator*
- Un objet *Validator* fournit une méthode
  - *validate(FacesContext ct, UIComponent cp, Object value) throws ValidatorException* : en charge d'effectuer la validation
    - *ct* : permet d'accéder au contexte de l'application JSF
    - *cp* : référence sur le composant dont la donnée est à valider
    - *value* : la valeur de la donnée
- L'exception permet d'indiquer si la validation ne s'est pas correctement passée
- Si exception déclenchée, au développeur à prendre en charge les messages à retourner via l'objet *FacesContext*



# Validation de données : validators personnalisés

- Pour utiliser une classe de type *Validator*, il faut la déclarer dans le fichier de configuration *faces-config.xml*
- Il faut identifier dans la balise *validator* la classe utilisée pour implémenter le Validator

```
<validator>  
  <validator-id>myValidatorId</validator-id>  
  <validator-class>beanPackage.MyValidator</validator-class>  
</validator>
```

- Pour utiliser le validator dans une page JSP, il faut utiliser la balise *<validator>* dans le corps d'un composant JSF

```
<html:inputText value="#{...}" ... >  
  <core:validator validatorId="myValidatorId" />  
</html:inputText>
```

# Validation de données : validators personnalisés

## ➤ Exemple : utilisation de la balise `<core:validator>`

```
public class PersoValidator implements Validator {  
    public void validate(FacesContext fc, UIComponent comp, Object ref)  
        throws ValidatorException {  
        String myValue = null;  
  
        if (fc == null || comp == null || ref == null) {  
            throw new NullPointerException();  
        }  
  
        myValue = ref.toString();  
  
        final String magicNumber = "123456789";  
        if (!myValue.equals(magicNumber)) {  
            throw new ValidatorException(new FacesMessage(  
                FacesMessage.SEVERITY_ERROR,  
                "Problème de validation", "numéro magique erroné"));  
        }  
    }  
}
```

*PersoValidator.java* du  
projet **Validators**

```
...  
    <validator>  
        <validator-id>myValidatorId</validator-id>  
        <validator-class>beanPackage.PersoValidator</validator-class>  
    </validator>  
</faces-config>
```

*faces-config.xml* du  
projet **Validators**

# Validation de données : validators personnalisés

## ➤ Exemple (suite) : utilisation de la balise `<core:validator>`

```
<core:view>
  <h4><html:outputText value="Utilisation d'un validator personnalisé :
    balise validator" /></h4>
  <html:form>
    <p>
      <html:outputText value="Veuillez saisir votre numéro magique (10
        chiffres) : " />
      <html:inputText id="compteId" value="#{persobean.magicnumber}" >
        <core:validator validatorId="persoValidatorId"/>
      </html:inputText><html:message for="compteId" showDetail="true"
        showSummary="true" />
    </p>
    <p>
      <html:commandButton value="Envoyer" />
    </p>
  </html:form>
</core:view>
```

*persotagvalidator.jsp* du  
projet **Validators**

### Utilisation d'un validator personnalisé : balise validator

Veuillez saisir votre numéro magique (10 chiffres) :  Problème de validation numéro magique erroné

Envoyer

# Validation de données : validators personnalisés

- L'utilisation de l'attribut *validator* du composant *UIInput* à valider s'appuie sur l'utilisation d'un Bean
- Il faut donc fournir à l'attribut *validator* une expression qui désigne la méthode de validation

```
<html:inputText value="#{...}" validator="#{myVal.validate}" ... >
    ...
</html:inputText>
```

- Le nom de la méthode n'est pas imposé, toutefois elle devra respecter la signature de la méthode *validate()* de l'interface *Validator*
- L'inconvénient de cette approche est qu'elle est fortement liée à l'application et il devient difficile de réutiliser le *validator*

# Validation de données : validators personnalisés

## ► Exemple : utilisation de l'attribut *validator*

```
public class PersoBeanValidator {
    private String magicNumber;

    public String getMagicnumber() {
        return magicNumber;
    }
    public void setMagicnumber(String magicNumber) {
        this.magicNumber = magicNumber;
    }

    public void validatePerso(FacesContext fc, UIComponent comp, Object ref)
        throws ValidatorException {
        String myValue = null;

        if (fc == null || comp == null || ref == null) {
            throw new NullPointerException();
        }

        myValue = ref.toString();
        final String magicNumber = "123456789";
        if (!myValue.equals(magicNumber)) {
            throw new ValidatorException(new FacesMessage(
                FacesMessage.SEVERITY_ERROR, "Problème de validation",
                "numéro magique erroné"));
        }
    }
}
```

*PersoBeanValidator.java*  
du projet **Validators**

# Validation de données : validators personnalisés

## ➤ Exemple (suite) : utilisation de l'attribut *validator*

```
<core:view>
  <h4><html:outputText value="Utilisation d'un validator personnalisé :
    attribut validator" /></h4>
  <html:form>
    <p><html:outputText value="Veuillez saisir votre numéro magique (10
      chiffres) : " />
    <html:inputText id="compteId" value="#{persobeanvalidator.magicnumber}"
      validator="#{persobeanvalidator.validatePerso}" />
    <html:message for="compteId" showDetail="true" showSummary="true"/>
  </p>
  <p><html:commandButton value="Envoyer" /></p>
</html:form>
</core:view>
```

*persoattributvalidator.jsp*  
du projet **Validators**

Utilisation d'un validator personnalisé : attribut validator

Veuillez saisir votre numéro magique (10 chiffres) :  Problème de validation numéro magique erroné

Envoyer

# Conversion et Validation de données : écriture balises

---

- Pour bientôt ...
  - Ecriture de ses propres balises pour la conversion et la validation



# La gestion des événements : généralités

---

- JSF fournit un mécanisme d'événements se rapprochant des modèles de gestion d'événements des API Swing et SWT
- Certains composants JSF peuvent émettre des événements
  - C'est une source
- Des objets peuvent recevoir des événements
  - Ce sont des écouteurs (listeners en anglais)
- Le mécanisme d'événements est fondé sur la notion d'abonnement entre une source et un écouteur
- L'API JSF fournit deux types d'événements
  - Changement de valeur (*ValueChangeEvent*) : émis lors du changement de la valeur d'un composant de type *UIInput*
  - Lié à une action (*ActionEvent*) : émis lors d'une action sur un composant de type *UICommand*



 [keulkeul.blogspot.com](http://keulkeul.blogspot.com)



# La gestion des événements : cycle de vie

- Les écouteurs sont notifiés à partir du moment où
  - la **valeur** du composant **change**
  - le **formulaire** encapsulant le composant doit être soumis
- La notification intervient à différent endroit du cycle de vie
- Pour les listeners de type *ValueChangeListener* la notification se produit après la phase de **Process Validations**
- Pour les listeners de type *ActionListener* la notification se produit après la phase de **Invoke Application**
- Le traitement d'un écouteur peut affecter le cycle de vie
  - Laisser le cycle de vie se dérouler normalement
  - Passer les différentes phases jusqu'à celle de **Render Response** via un appel à *FacesContext.renderResponse()*
  - Passer toutes les phases via *FacesContext.responseComplete()*

# La gestion des événements : généralités

- Deux approches sont exploitables pour abonner un écouteur à un composant JSF (type *UIInput* ou *UICommand*)
  - utiliser un attribut (*valueChangeListener* ou *actionListener*) de la balise du composant JSF

```
<html:inputText onChange="submit()" value="#{databean.name}"
    valueChangeListener="#{databean.inputChangement}" />
```

L'attribut *valueChangeListener* précise une méthode appelée si le listener est déclenché

- utiliser une balise de CORE (*core:valueChangeListener* ou *core:actionListener*) dans le corps du composant JSF

```
<html:inputText onChange="submit()" value="#{databean.name}" >
    <core:valueChangeListener type="beanPackage.InputTextChangeListener" />
</html>
```

Cette classe contient une méthode qui est appelée si le listener est déclenché

## La gestion des événements : généralités

---

- Dans les deux cas, il faut fournir les méthodes qui seront appelées lors du déclenchement des écouteurs
- Dans le cas de l'attribut une méthode doit être fournie dans le Bean (ici *inputChangement*) avec en paramètre le type d'événement à traiter (*ValueChangeEvent* ou *ActionEvent*)
- Dans le cas de la balise une classe implémentant l'interface *ValueChangeListener* ou *ActionListener* doit être fournie
- A noter qu'il est possible qu'un Bean (au sens qui stocke des valeurs du paramètres) peut implémenter les interfaces des listeners ...
- Exemples pour chacun des écouteurs ...

# La gestion des événements : changement de valeur

---

- Les événements associés au changement de valeur sont déclenchés quand une valeur d'un composant est modifié (si la valeur est identique pas de déclenchement)
- L'événement transmis est de type *ValueChangeEvent*
  - *UIComponent* *getComponent()* : renvoie la source
  - *Object* *getNewValue()* : retourne la nouvelle valeur
  - *Object* *getOldValue()* : retourne l'ancienne valeur
  - ...
- Préciser un écouteur de type *ValueChangeListener*
  - par l'attribut *valueChangeListener* (sa valeur est une méthode)
  - par la balise *valueChangeListener* (sa valeur est une classe)
- Implémentation de l'interface *ValueChangeListener*
  - *void processValueChange(ValueChangeEvent)* : méthode qui traite le changement de valeur

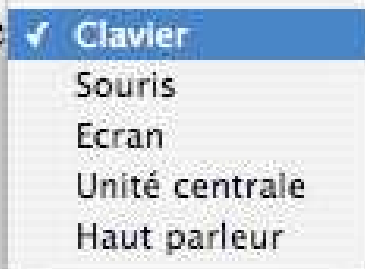
# La gestion des événements : changement de valeur

## ➤ Exemple : abonnement via un « attribut »

```
<html:form>
<p>
  <html:outputText value="Veuillez saisir votre nom : " />
  <html:inputText onChange="submit()" value="#{databean.name}"
    valueChangeListener="#{databean.inputChangement}" />
</p>
<p>
  <html:outputText value="Choisissez votre périphérique : " />
  <html:selectOneMenu onChange="submit()" value="#{databean.oneMenuValue}"
    valueChangeListener="#{databean.selectChangement}">
    <core:selectItems value="#{databean.manyCheckBoxItems}" />
  </html:selectOneMenu>
</p>
</html:form>
```

Attribut valueChangeList...

Veuillez saisir votre nom :

Choisissez votre périphérique : 

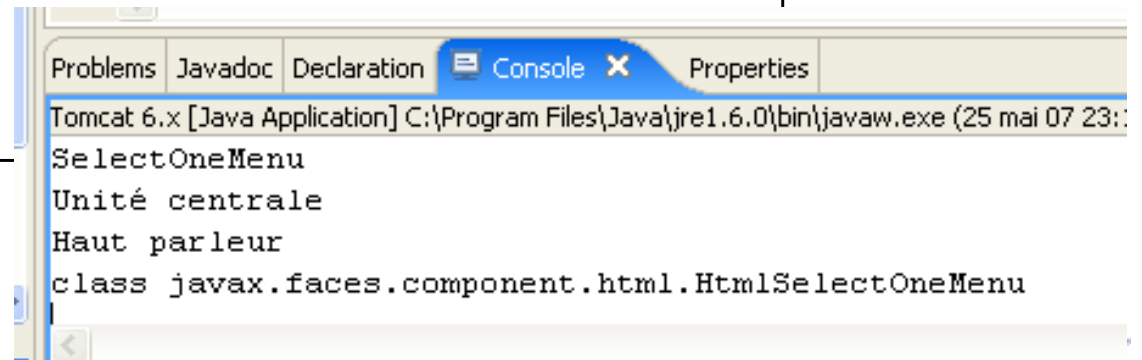
*valueChangeListenerAttribut.jsp*  
du projet **Event**

# La gestion des événements : changement de valeur

## ➤ Exemple (suite) : abonnement via un « attribut »

```
public class DataBean {  
    private String menuValue;  
  
    private String name= "";  
    ...  
  
    public void inputChangement(ValueChangeEvent event) {  
        System.out.println("InputText");  
        System.out.println(event.getOldValue());  
        System.out.println(event.getNewValue());  
        System.out.println(event.getComponent().getClass());  
    }  
    public void selectChangement(ValueChangeEvent event) {  
        System.out.println("SelectOneMenu");  
        System.out.println(event.getOldValue());  
        System.out.println(event.getNewValue());  
        System.out.println(event.getComponent().getClass());  
    }  
}
```

*DataBean.java*  
du projet **Event**



# La gestion des événements : changement de valeur

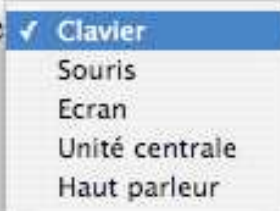
## ► Exemple : abonnement via une « balise »

```
<html:form>
<p>
  <html:outputText value="Veuillez saisir votre nom : " />
  <html:inputText onChange="submit()" value="#{databean.name}" >
    <core:valueChangeListener type="beanPackage.InputTextChangeListener"/>
  </html:inputText>
</p>

<p>
  <html:outputText value="Choisissez votre périphérique : " />
  <html:selectOneMenu onChange="submit()" value="#{databean.oneMenuValue}" >
    <core:valueChangeListener type="beanPackage.SelectOneMenuListener" />
    <core:selectItems value="#{databean.manyCheckBoxItems}" />
  </html:selectOneMenu>
</p>
</html:form>
```

Attribut valueChangeList...

Veuillez saisir votre nom :

Choisissez votre périphérique : 

*valueChangeListenerTag.jsp*  
du projet **Event**



# La gestion des événements : changement de valeur

## ➤ Exemple (suite) : abonnement via une « balise »

```
public class InputTextChangeListener implements ValueChangeListener {  
    public void processValueChange(ValueChangeEvent event) throws  
        AbortProcessingException {  
        System.out.println("InputText");  
        System.out.println(event.getOldValue());  
        System.out.println(event.getNewValue());  
        System.out.println(event.getComponent().getClass());  
    }  
}
```

*SelectOneMenuListener.java*  
du projet **Event**

*InputTextChangeListener.java*  
du projet **Event**

```
public class SelectOneMenuListener implements ValueChangeListener {  
    public void processValueChange(ValueChangeEvent event) throws  
        AbortProcessingException {  
        System.out.println("SelectOneMenu");  
    }  
}
```

# La gestion des événements : changement de valeur

## ➤ Exemple : abonnement via une « balise » et un « attribut »

```
<html:form>
<p>
  <html:outputText value="Veuillez saisir votre nom : " />
  <html:inputText onChange="submit()" value="#{mixdatabean.name}"
    valueChangeListener="#{mixdatabean.inputChangement}" />
</p>
<p>
  <html:outputText value="Choisissez votre périphérique : " />
  <html:selectOneMenu onChange="submit()" value="#{mixdatabean.oneMenuValue}" >
    <core:valueChangeListener type="beanPackage.DataBeanBis" />
    <core:selectItems value="#{mixdatabean.manyCheckBoxItems}" />
  </html:selectOneMenu>
</p>
</html:form>
```

*valueChangeListenerTagBis.jsp* du  
projet **Event**

```
public class DataBeanBis implements ValueChangeListener {
    ...
    public void inputChangement(ValueChangeEvent event) {
        System.out.println("InputText");
        System.out.println(event.getOldValue());
        System.out.println(event.getNewValue());
        System.out.println(event.getComponent().getClass());
    }
    public void processValueChange(ValueChangeEvent event) throws
        AbortProcessingException {
        System.out.println("SelectOneMenu");
    }
}
```

*DataBeanBis.java*  
du projet **Event**

## La gestion des événements : actions

---

- Les événements associés aux actions sont déclenchés quand des cliques sur des boutons ou des liens sont réalisés
- L'événement transmis est de type *ActionEvent*
  - *UIComponent GetComponent()* : renvoie la source
  - ...
- Préciser un écouteur de type *ActionListener*
  - par l'attribut *actionListener* (sa valeur est une méthode)
  - par la balise *actionListener* (sa valeur est une classe)
- Implémentation de l'interface *ActionListener*
  - *void processAction(ActionEvent)* : méthode qui traite l'action

# La gestion des événements : actions

## ➤ Exemple : quatre saisons (abonnement via attribut)

```
<html:form>
  <h1><html:outputText value="Quatre Saisons ... " /></h1>
  <html:panelGrid columns="2" >

    <html:commandButton disabled="false" value="Hivers" id="Hivers"
      binding="#{saisonbean.hivers}" actionListener="#{saisonbean.traiterHivers}" />

    <html:commandButton disabled="true" value="Printemps" id="Printemps"
      binding="#{saisonbean.printemps}" actionListener="#{saisonbean.traiterPrintemps}" />

    <html:commandButton disabled="true" value="Eté" id="Ete"
      binding="#{saisonbean.ete}" actionListener="#{saisonbean.traiterEteAutomne}" />

    <html:commandButton disabled="true" value="Automne" id="Automne"
      binding="#{saisonbean.automne}" actionListener="#{saisonbean.traiterEteAutomne}" />

  </html:panelGrid>
</html:form>
```

*actionListenerAttribut.jsp*  
du projet **Event**

Quatre Saisons ...



Quatre saisons : attribut "a...

Quatre Saisons ...



# La gestion des événements : actions

## ➤ Exemple (suite) : quatre saisons (abonnement via attribut)

```
public class SaisonBean {  
    public void traiterHivers(ActionEvent e) {  
        hivers.setDisabled(true);  
        printemps.setDisabled(false);  
    }  
    public void traiterPrintemps(ActionEvent e) {  
        printemps.setDisabled(true);  
        ete.setDisabled(false);  
    }  
    public void traiterEteAutomne(ActionEvent e) {  
        String id = e.getComponent().getId();  
        if (id.equals("Ete")) {  
            automne.setDisabled(false);  
            ete.setDisabled(true);  
        }  
  
        if (id.equals("Automne")) {  
            automne.setDisabled(true);  
            hivers.setDisabled(false);  
        }  
    }  
    public HtmlCommandButton getAutomne() { return automne; }  
    public void setAutomne(HtmlCommandButton automne) { this.automne = automne; }  
    ...  
}
```

*SaisonBean.java*  
du projet **Event**

# La gestion des événements : actions

## ➤ Exemple : des liens (abonnement par balise)

```
<html:form>
  <h1><html:outputText value="Des liens ..." /></h1>
  <html:commandLink value="Confirmer" id="confirmer">
    <core:actionListener type="beanPackage.ActionBean" />
  </html:commandLink>
  <html:commandLink value="Annuler" id="annuler">
    <core:actionListener type="beanPackage.ActionBean" />
  </html:commandLink>
</html:form>
```

*actionListenerTag.jsp* du projet

**Event**

Des liens : balise "actionLis...

**Des liens ...**

[Confirmer](#) [Annuler](#)

```
public class ActionBean implements ActionListener {
  public void processAction(ActionEvent event) throws AbortProcessingException {
    String id = event.getComponent().getId();

    if (id.equals("confirmer")) {
      System.out.println("Action Confirmation");
    }

    if (id.equals("annuler")) {
      System.out.println("Action Annulation");
    }
  }
}
```

*ActionBean.java* du projet

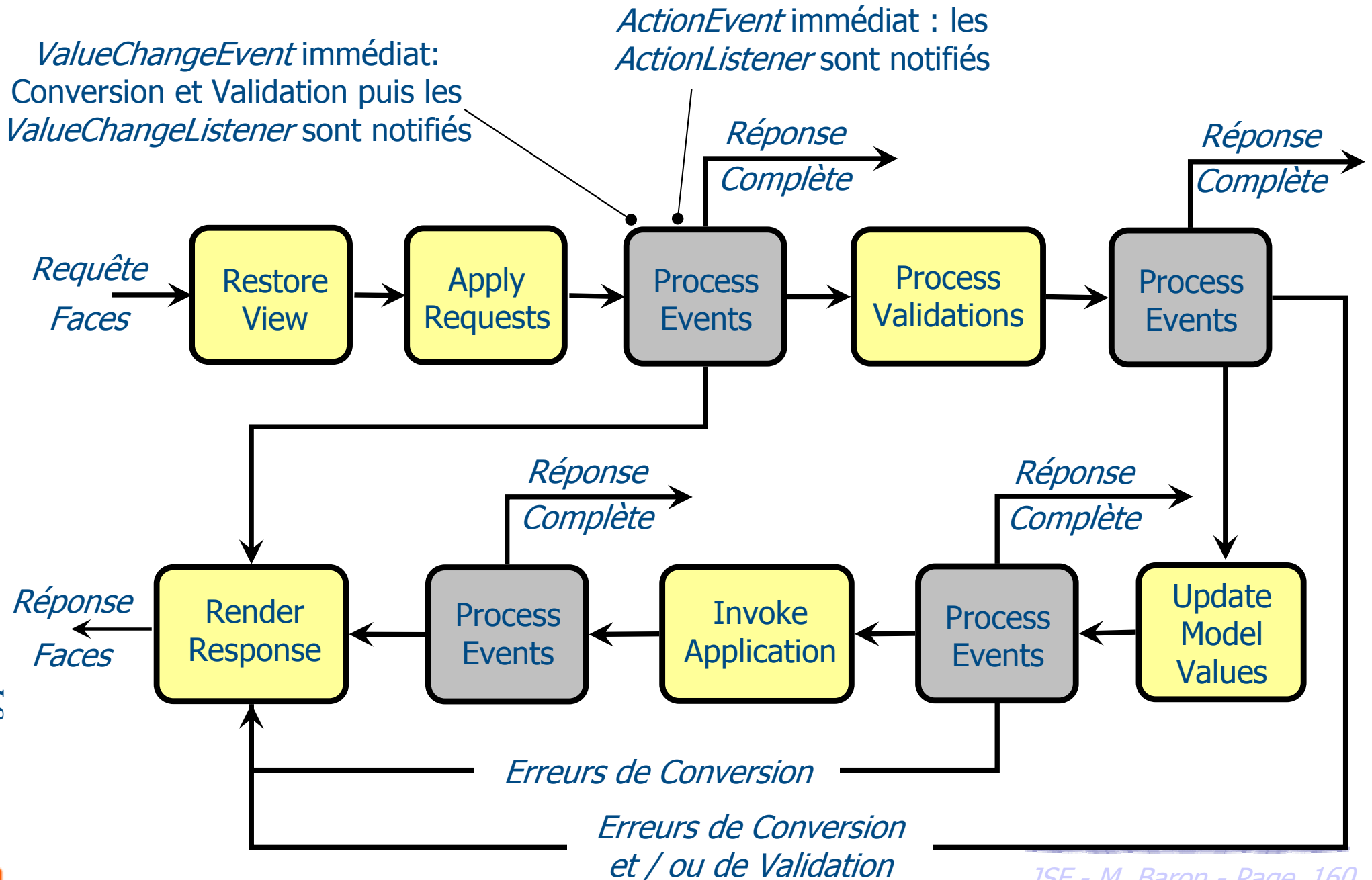
**Event**

## La gestion des événements : immediate

---

- Rappelons la position de déclenchement des événements dans le cycle de vie JSF
  - *ValueChangeEvent* après la phase de **Process Validations**
  - *ActionEvent* après la phase **Invoke Application**
- Toutefois, il serait intéressant de pouvoir déclencher ces événements juste après la phase **Apply Request**
- Pourquoi, quel est l'intérêt de déclencher des événements immédiatement ?
  - Déclencher les événements d'un composant avant tous les événements des autres composants
  - « Court-circuiter » le processus de cycle de vie pour désactiver ou pas la conversion et la validation des autres composants

# La gestion des événements : immédiate





## La gestion des événements : *immediate*

- Pour déclencher des événements immédiatement, il faut modifier la valeur de l'attribut *immediate* à *true*
- Si un composant de saisie à un déclenchement immédiat
  - la conversion et la validation sont réalisées après la phase **Apply Request Values**
  - les *ValueChangeListener* sont ensuite déclenchés
  - le cycle de vie continue sa progression (phase **Process Validations**)
  - Pour le stopper appel explicite à *renderResponse()* de *FacesContext*
- Si un composant de command à un déclenchement immédiat
  - les *ActionListener* sont déclenchés
  - le cycle de vie est court-circuité car appel implicite à *RenderResponse* (*FacesContext*)
- Il est à noter que le dernier cas a déjà été rencontré dans la partie « Conversion des données »

# La gestion des événements : immediate

## ➤ Exemple : changement de localisation sans validation ...

Téléphone

Pays

Ecran de démarrage

Téléphone

"priceId": La donnée a plus que les 10 caractères maximum autorisés.

Pays

Si validation via le bouton de soumission, le champ Téléphone notifie d'une erreur de saisie

Phone number

Country

Si changement de localisation (composant *SelectOneMenu*) avec l'attribut *immediate* à *true*, la validation n'est pas effectuée

# La gestion des événements : immediate

## ➤ Exemple (suite) : changement de localisation sans validation

```
<core:view>
  <core:loadBundle basename="EventMessages" var="bundle" />
  <html:form>
    <p>
      <html:outputText value="#{bundle.PHONE_NUMBER_MESSAGE}" />
      <html:inputText id="priceId" value="#{selectimmediate.phonenumber}" >
        <core:validateLength minimum="10" maximum="10" />
      </html:inputText><html:message for="priceId" />
    </p>
    <p>
      <html:outputText value="#{bundle.COUNTRY_MESSAGE}" />
      <html:selectOneMenu onchange="submit()" immediate="true"
        valueChangeListener="#{selectimmediate.countryChanged}"
        value="#{selectimmediate.countryValue}" >
        <core:selectItems value="#{selectimmediate.countryNames}" />
      </html:selectOneMenu>
    </p>
    <p>
      <html:commandButton value="#{bundle.VALID_MESSAGE}" />
    </p>
  </html:form>
</core:view>
```

*immediateAttribut.jsp* du projet  
**Event**

# La gestion des événements : immediate

## ➤ Exemple (suite) : changement de localisation sans validation

```
public class DataBeanSelectImmediate {
    private static final String FRENCH_COUNTRY_NAME = "Français";
    private static final String ENGLISH_COUNTRY_NAME = "English";

    private SelectItem[] countryNames = new SelectItem[] {
        new SelectItem(FRENCH_COUNTRY_NAME),
        new SelectItem(ENGLISH_COUNTRY_NAME)
    };

    public void countryChanged(ValueChangeEvent e) {
        System.out.println("DataBeanSelectImmediate.countryChanged()");
        FacesContext current = FacesContext.getCurrentInstance();
        if (FRENCH_COUNTRY_NAME.equals(e.getNewValue())) {
            current.getViewRoot().setLocale(Locale.FRENCH);
        } else {
            current.getViewRoot().setLocale(Locale.ENGLISH);
        }
        current.renderResponse();
    }
}
```

*DataBeanSelectImmediate.java*  
du projet **Event**

*EventMessages\_fr.properties* du projet **Event**

```
PHONE_NUMBER_MESSAGE = Téléphone
COUNTRY_MESSAGE = Pays
VALID_MESSAGE = Valider
```

Court-circuite le cycle de vie par un appel explicite Ne traite pas les autres composants JSF (pas de conversion ni validation)

# La gestion des événements : cycle de vie

---

- Avant et après chaque phase du cycle de vie JSF, des événements de type *PhaseEvent* sont déclenchés
- L'abonnement peut être utile lors de phases de déboguage
- Le traitement des *PhaseEvent* est réalisé par un objet de type *PhaseListener*
- L'abonnement se fait via le fichier de configuration JSF en déclarant le nom de la classe qui implémente *PhaseListener*
- Modification du fichier *faces-config.xml*
  - Ajouter la balise *<lifecycle>* et la sous balise *<phase-listener>*
  - Préciser le nom de la classe de type *PhaseListener*

# La gestion des événements : cycle de vie

---

## ➤ Événement *PhaseEvent*

- *getFacesContext()* : retourne l'instance du *FacesContext* pour la requête en cours de traitement
- *getPhaseId()* : retourne l'ID de la phase du cycle de vie en cours de traitement

## ➤ Ecouteur *PhaseListener*

- *afterPhase(PhaseEvent)* : appelée quand une phase du cycle de vie se termine
- *beforePhase(PhaseEvent)* : appelée avant le démarrage d'une phase du cycle de vie
- *PhaseId getPhaseId()* : retourne un objet *PhaseId* permettant d'identifier le listener à la phase

## ➤ La classe *PhaseId* définit un ensemble de constantes permettant d'identifier les différentes phases du cycle de vie

# La gestion des événements : cycle de vie

## ➤ Exemple : traiter les événements de toutes les phases

```
<faces-config>
    ...
    <lifecycle>
        <phase-listener>beanPackage.LifecycleListener</phase-listener>
    </lifecycle>
</faces-config>
```

*faces-config.xml* du projet **Event**

```
public class LifecycleListener implements PhaseListener {

    public void afterPhase(PhaseEvent phaseEvent) {
        System.out.println("LifecycleListener.afterPhase()" +
            phaseEvent.getPhaseId());
    }
    public void beforePhase(PhaseEvent phaseEvent) {
        System.out.println("LifecycleListener.beforePhase()" +
            phaseEvent.getPhaseId());
    }
    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}
```

*LifecycleListener.java* du projet **Event**

```
LifecycleListener.beforePhase() RESTORE_VIEW(1)
LifecycleListener.afterPhase() RESTORE_VIEW(1)
LifecycleListener.beforePhase() APPLY_REQUEST_VALUES(2)
LifecycleListener.afterPhase() APPLY_REQUEST_VALUES(2)
LifecycleListener.beforePhase() PROCESS_VALIDATIONS(3)
SelectOneMenu
Unité centrale
Haut parleur
class javax.faces.component.html.HtmlSelectOneMenu
LifecycleListener.afterPhase() PROCESS_VALIDATIONS(3)
LifecycleListener.beforePhase() UPDATE_MODEL_VALUES(4)
LifecycleListener.afterPhase() UPDATE_MODEL_VALUES(4)
LifecycleListener.beforePhase() INVOKE_APPLICATION(5)
LifecycleListener.afterPhase() INVOKE_APPLICATION(5)
LifecycleListener.beforePhase() RENDER_RESPONSE(6)
LifecycleListener.afterPhase() RENDER_RESPONSE(6)
```

## Composants additionnels : Apache Tomahawk

- La spécification JSF décrit un ensemble de composants qui ont été implémentés entre autre par Apache MyFaces
- Ces composants sont à l'utilisation très limités au niveau interaction
- Apache a donc fourni la bibliothèque Tomahawk dont l'objectif est d'ajouter des fonctionnalités supplémentaires
  - <http://myfaces.apache.org/tomahawk>
- Cette bibliothèque fournit
  - Des composants graphiques (extension à *html*)
  - Des composants fonctionnels (extension à *core*)
  - Des validators
  - Et pleins d'autres choses
- Adresse d'exemples
  - <http://www.irian.at/myfaces>

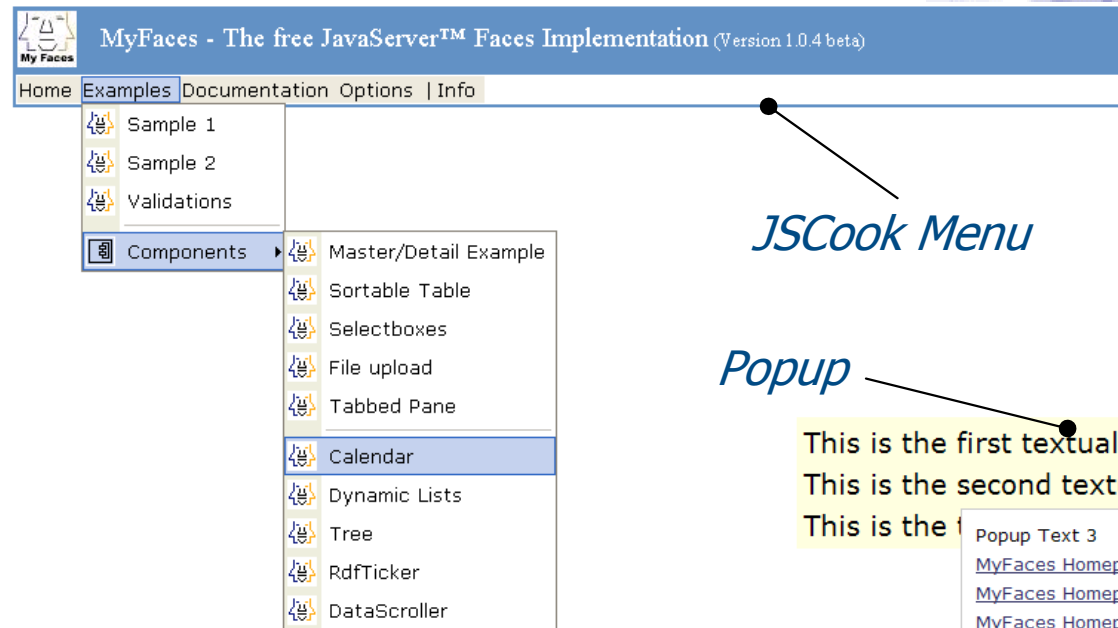
**A voir également « Tabago »  
qui est une autre bibliothèque  
de composants**

**<http://myfaces.apache.org/tobago>**





# Apache Tomahawk : composants graphiques



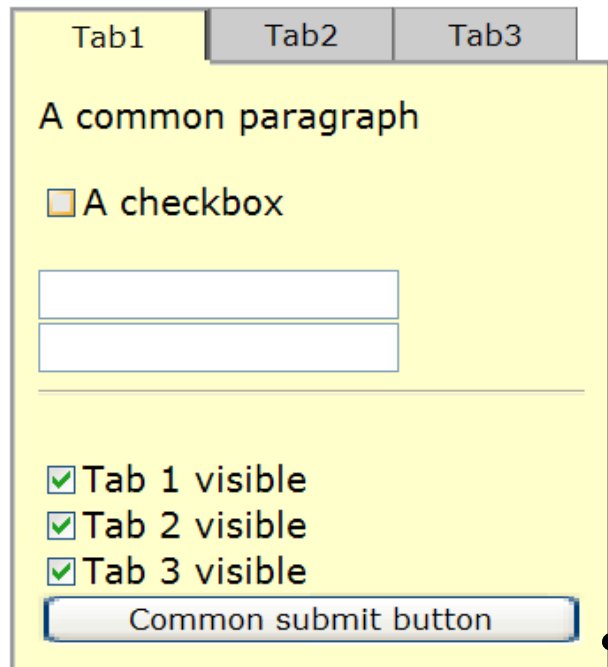
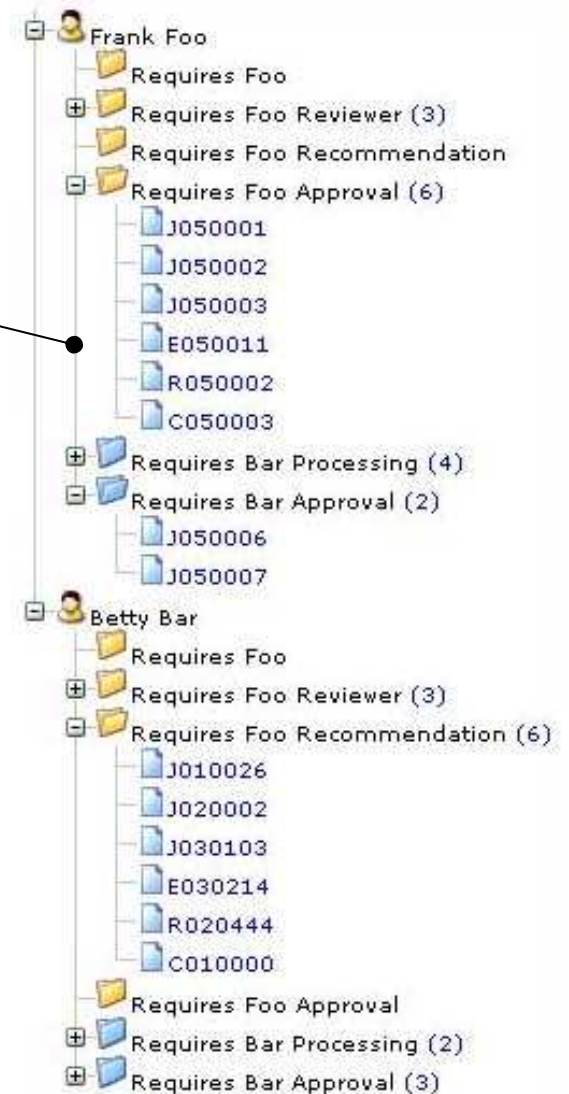
*JSCook Menu*

*Popup*

This is the first textual text situation.  
This is the second textual text situation.  
This is the third textual text situation.

Popup Text 3  
[MyFaces Homepage](#)  
[MyFaces Homepage](#)  
[MyFaces Homepage](#)  
[MyFaces Homepage](#)  
[MyFaces Homepage](#)

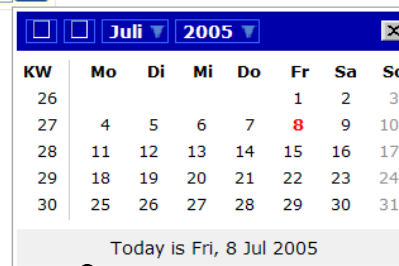
*Tree « 2 »*



*TabbedPane*

08.07.05

Submit



*Calendar*

## Apache Tomahawk : composants graphiques (suite)

Normal B I U x<sub>2</sub> x<sup>2</sup> Undo Redo ...

**Properties**

Title:

Description:

*HtmlEditor*

	Mo
1	Car T
2	Car T
3	Car T
4	Car T
5	Car T








### Extended Data Table

(header table)	
<u>Typ ↑</u>	<u>Farbe</u>
car A	red
car B	blue
car C	green
car D	yellow
car E	orange
(footer col1) (footer col2)	
(footer table)	

	Modell	Farbe
1	Car Type 1	blue
2	Car Type 2	blue
3	Car Type 3	blue
4	Car Type 4	blue
5	Car Type 5	blue
6	Car Type 6	blue
7	Car Type 7	blue
8	Car Type 8	blue
9	Car Type 9	blue
10	Car Type 10	blue



## Data Scroller

Header 1	Header 2	Header 3
9001	  XY	XY 9001
9001	  A	A 9001
9001	  B	B 9001
9001	 C	C 9001
Footer		

## Tree Table

# Apache Tomahawk : composants graphiques (suite)

Product Information
» Search Products
» <b>Search Accessory</b>
» Advanced Search
Online Shop
Corporate Information
Contact

*Panel Navigation 2*

*Schedule*

*Panel Navigation*

Startseite
<b>Beispiele</b>
Beispiel 1
Beispiel 2
<b>Komponenten</b>
Master/Detail Beispiel
<b>Sortierbare Liste</b>
Selectboxen
Datei hochladen
Karteireiter
Kalender
Dokumentation
Einstellungen
Info

2006-03-27	2006-03-28	2006-03-29	2006-03-30	2006-03-31 11:00-14:00: Long lunch	2006-04-01
					2006-04-02
2006-04-03	2006-04-04	2006-04-05	2006-04-06	2006-04-07	2006-04-08
					2006-04-09
2006-04-10	2006-04-11	2006-04-12	2006-04-13	2006-04-14	2006-04-15
					2006-04-16
2006-04-17	2006-04-18	2006-04-19	2006-04-20	2006-04-21	2006-04-22
					2006-04-23
2006-04-24	2006-04-25 14:00-14:45: Test MyFaces schedule component 14:25-16:25: Show schedule component to boss	2006-04-26 9:00-17:00: Thoroughly test schedule component	2006-04-27	2006-04-28	2006-04-29
					2006-04-30

# Apache Tomahawk : Validators

- Tomahawk fournit également des validators complémentaires à ceux fournis par la spécification JSF
- *validateCreditCard*
  - Pour valider un numéro de carte de crédit
  - Permet de choisir la marque (parmi amex, visa, mastercard, discover)
- *validateUrl*
  - Pour valider une URL
- *validateEmail*
  - Pour valider une adresse électronique
- *validateRegExpr*
  - Pour valider des expressions régulières

```
<html:inputText id="regExprValue" value="#{validateForm.regExpr}" required="true">  
  <t:validateRegExpr pattern='\d{5}' />  
</html:inputText>
```

# Apache Tomahawk : installation

- Apache Tomahawk n'est ni plus ni moins qu'une librairie de balise
- Se rendre à l'adresse : *<http://myfaces.apache.org/download>*
- Télécharger l'archive *tomahawk-x.x.x-bin.zip*
- Décompresser dans le répertoire WEB-INF/lib de l'application web
- Importer les balises personnalisées de Tomahawk

```
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>
```

- Utiliser les balises de votre choix ...

```
<html:inputText id="regExprValue" value="#{validateForm.regExpr}" required="true">  
  <t:validateRegExpr pattern='\d{5}' />  
</html:inputText>
```

# Composant personnalisé : principe

---

- A approfondir ...
  - Généralité sur la conception de composants personnalisés
  - Balises personnalisés, code du composant, rendu
  - Exemple : thermomètre étendu ...



- Premières impressions d'un novice ...
  - Une impression que c'est pas aussi « souple » que Struts
  - La disponibilité de la spécification de JSF et du tutorial sont vraiment utiles (surtout pour comprendre le fonctionnement du cycle de vie)
  - Mes coups de coeur : l'approche par composants et la présence d'un cycle de vie qui « tient la route »
  - Les outils, gratuits, sous Eclipse de bonne qualité sont peu nombreux (utilisation du plugin Amateras : <http://amateras.sourceforge.jp>)
- Les choses non étudiés, prochainement ...
  - Terminer : « *créer son converter* », « *créer ses propres balises pour la conversion et la validation* » puis « *composant personnalisé* »
  - Comparatif des outils pour gérer du JSF avec « plaisir » ;-)
  - « Propriétier » les sources des exemples et les diffuser
  - Les Facelets (système de *template* pour JSF)