



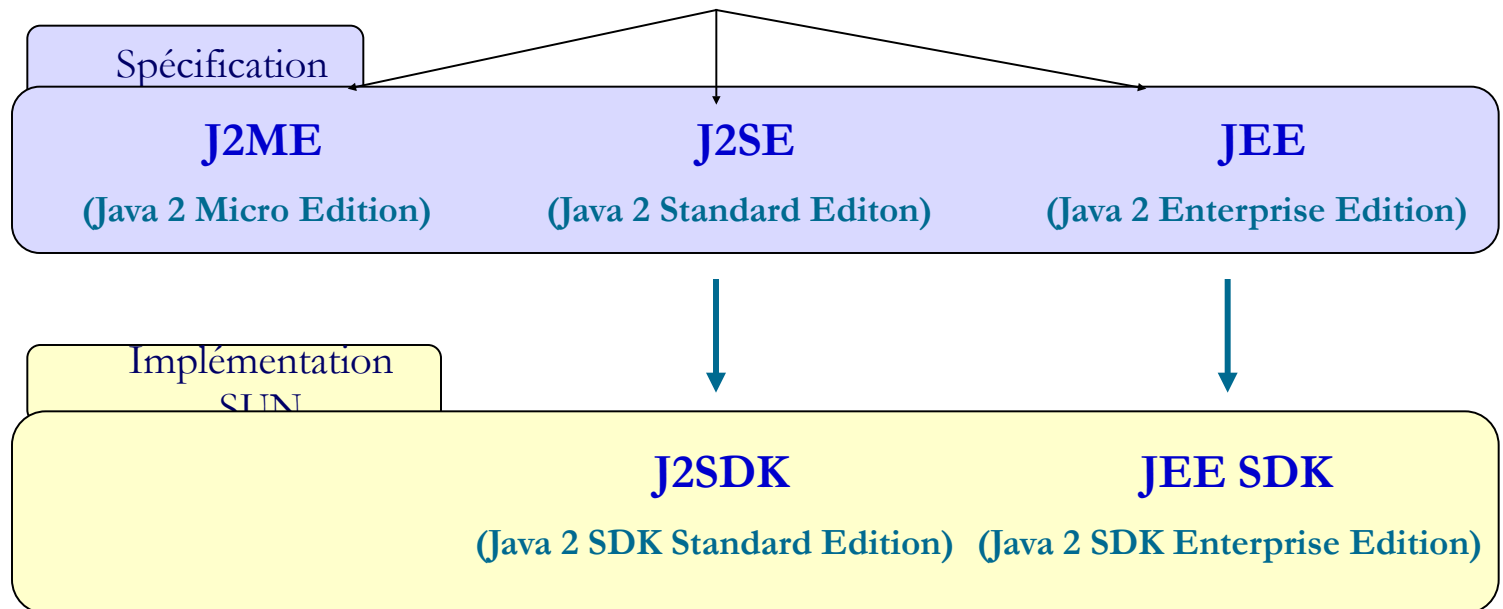
# **JEE**

## **Servlet, JSP, TagLib, JSF, Ejb et JPA**

**© M. Lahmer**

## Historique et Nominations

- JDK 1.0
- JDK 1.1
- Décembre 98 : **Java 2**



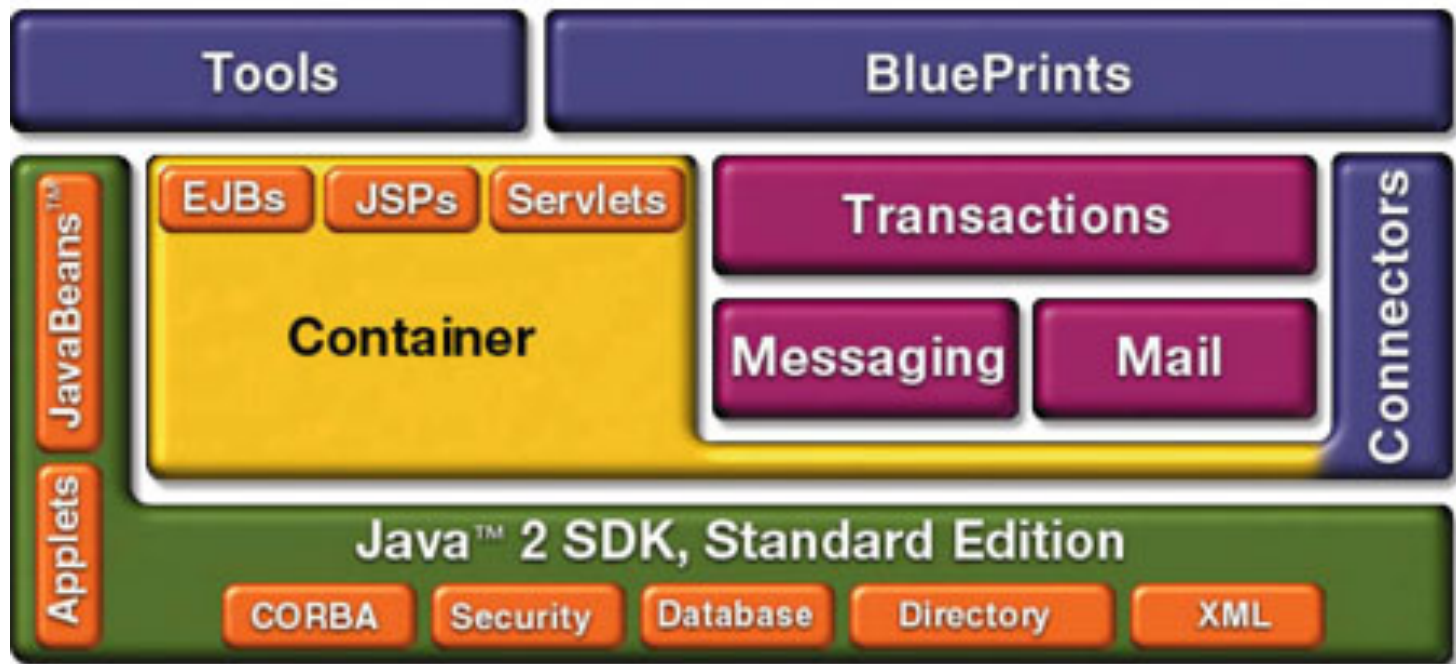
## Objectifs de JEE



Permettre aux développeurs de se concentrer sur le développement métier en les dégageant des services bas niveaux (Transactions, Intégration, Équilibrage de charge, Nommage, Persistance, Sécurité, Clustering...)

JEE

## Vue Globale





# JEE Les APIS

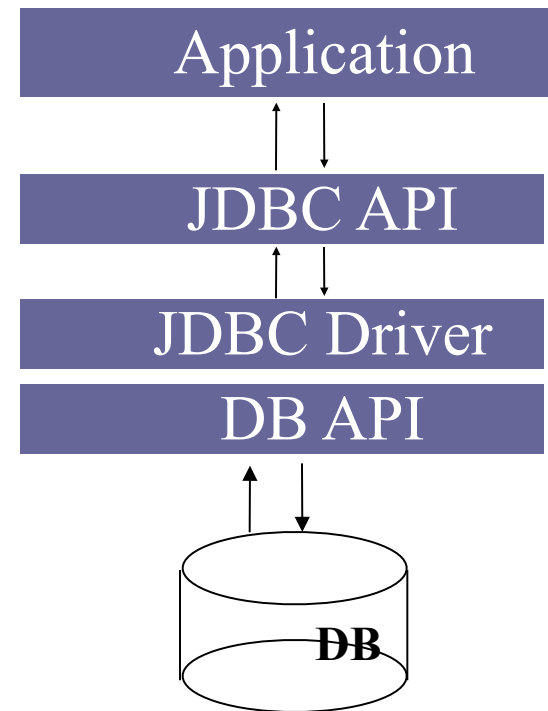
les composants : Servlet, JSP, EJB

les services : JDBC, JTA/JTS, JNDI, JCA, JAAS

la communication : RMI-IIOP, JMS, Java Mail

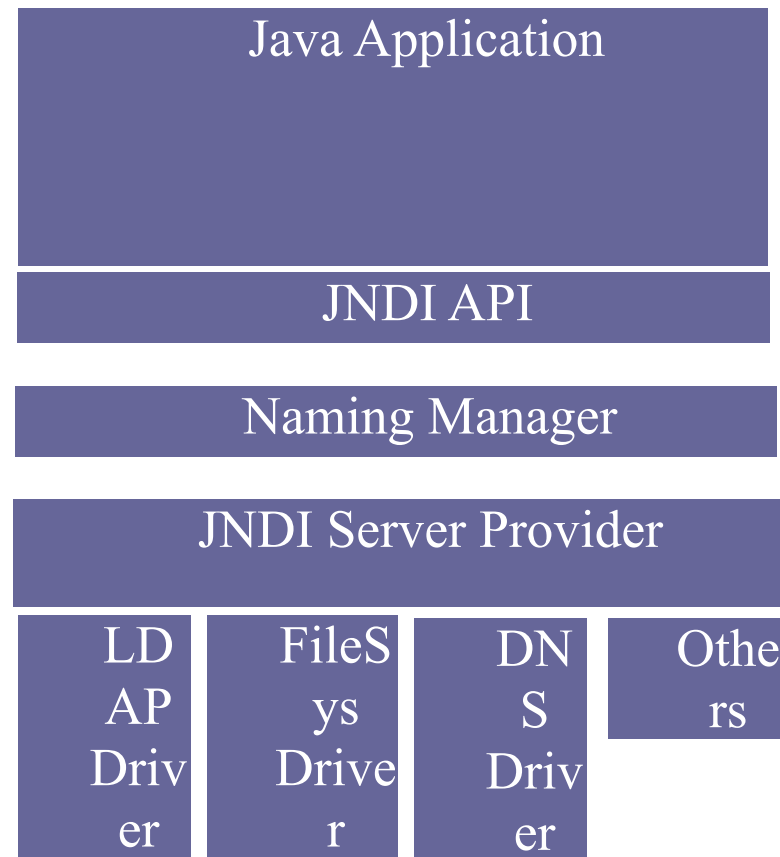
## JDBC (Java DataBase Connectivity)

Interface Java  
pour un accès  
standard à des  
bases de  
données  
hétérogènes.



## JNDI (Java Naming & Directory Interface)

API permettant aux applications d'accéder aux services de naming d'une façon transparente



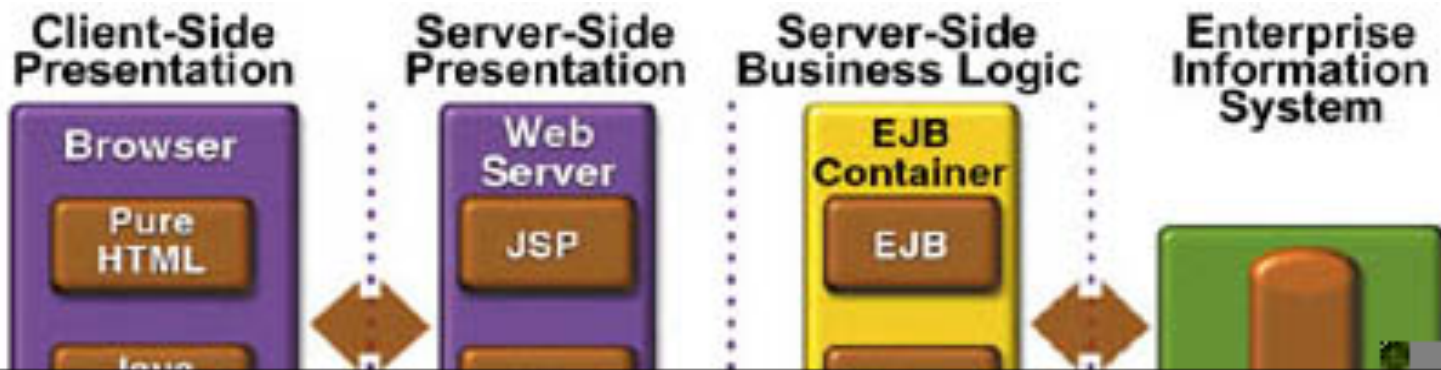
## Java IDL (Java Interface Definition Language)

Utilisée pour intégrer Java et CORBA.

permet de développer des objets Java déployables en tant que ORBs.



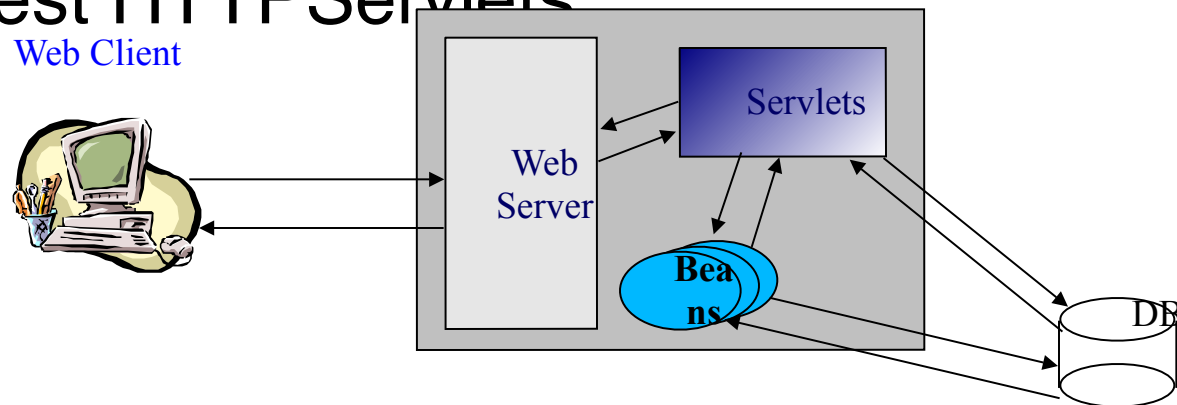
# JEE



## Java Servlets

Classe Java qui fournit un service conversationnel (Request/Response).

Le type de servlets le plus connu est HTTPServlets



## JSP (Java Server Pages)

Type de Servlets éditables.

Convertis en Servlets.

Utilise la notion des tags pour inclure du code Java dans le HTML.

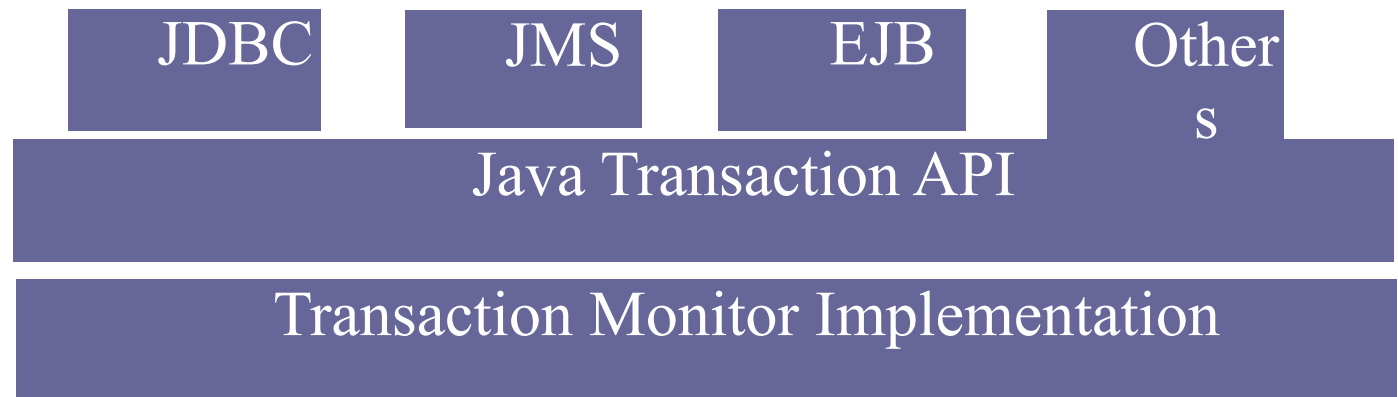
## JTS (Java Transaction Service)

Spécification d'implémentation d'une couche d'adaptation vers le service de transaction objet de CORBA (OTS).

Propage les transactions en utilisant IIOP (Internet Inter-ORB Protocol).

## JTA (Java Transaction Architecture)

API Java standard pour l'accès au moniteur transactionnel.



## JavaMail

API Java pour l'accès aux serveurs de messagerie.

Disponible pour SMTP, IMAP et POP3.

## JAF (JavaBeans Activation Framework)

Intègre un support pour les MIME  
Types pour la plate forme Java.

Utilisé par JavaMail.



**JEE**

## **JMS (Java Message Service)**

API pour accès aux messages  
orientés middleware.



## XML (eXtensible Markup Language)

Langage pour la définition d'autres  
«Markup Language».

Permet de représenter les données  
d'une façon unifier pour différents  
métiers.

## JCA (JEE Connector Architecture)

Spécification pour la  
communication avec les  
systèmes existants tels que SAP,  
CICS/COBOL, Siebel,...

## EJB (Enterprise Java Bean)

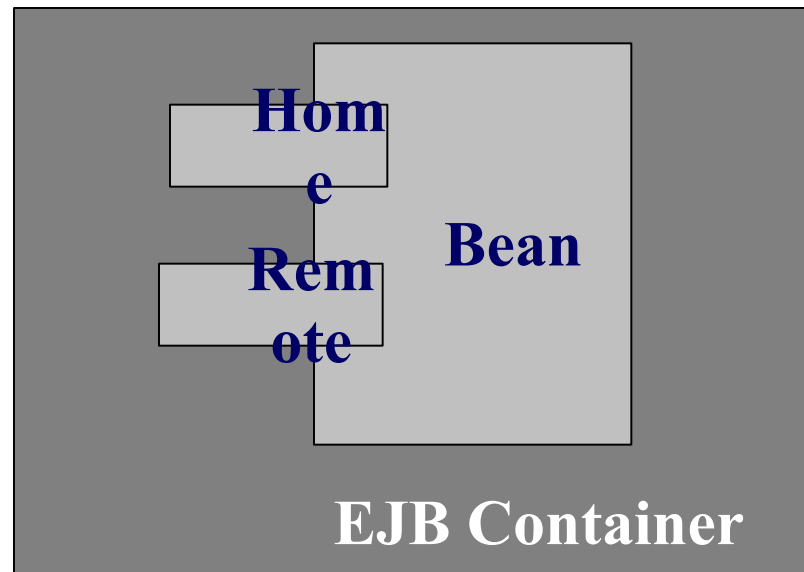
Composants qui fournissent un service métier pour des applications clientes.

Ils sont réutilisables, distribuables et que plusieurs applications peuvent partager.



**JEE**

## **EJB (Enterprise Java Bean)**



## EJB (Enterprise Java Bean)

Trois approches :

- Approche fonctionnelle.
- Approche conversationnelle.
- Approche de persistance.

Équivalent à :

- Stateless session bean
- Stateful session bean
- Entity bean

## **EJB (Enterprise Java Bean) : Stateless Session Bean**

Fournit un « single\_use service ».

Ne maintient pas un état relatif au client.

Ne survit pas aux crashes du serveur.

A une très courte durée de vie (relative à la durée de l'exécution d'une méthode).

Deux instances du même stateless EJB sont identiques.

## **EJB (Enterprise Java Bean) : Stateful Session Bean**

Fournit un mode conversationnel avec l'application cliente.

Maintien l'état de l'EJB par rapport au client.

Ne survie pas aux crashes du serveur.

A une durée relativement courte (la durée d'une session utilisateur).

Chaque instance du même EJB est différente des autres.

## **EJB (Enterprise Java Bean) : Entity Bean**

Représente la persistance des données.

Survie aux crashes du serveur.

Une instance de l'EJB représente une copie des données de la base de données.





# Java et les bases de données :JDBC

---



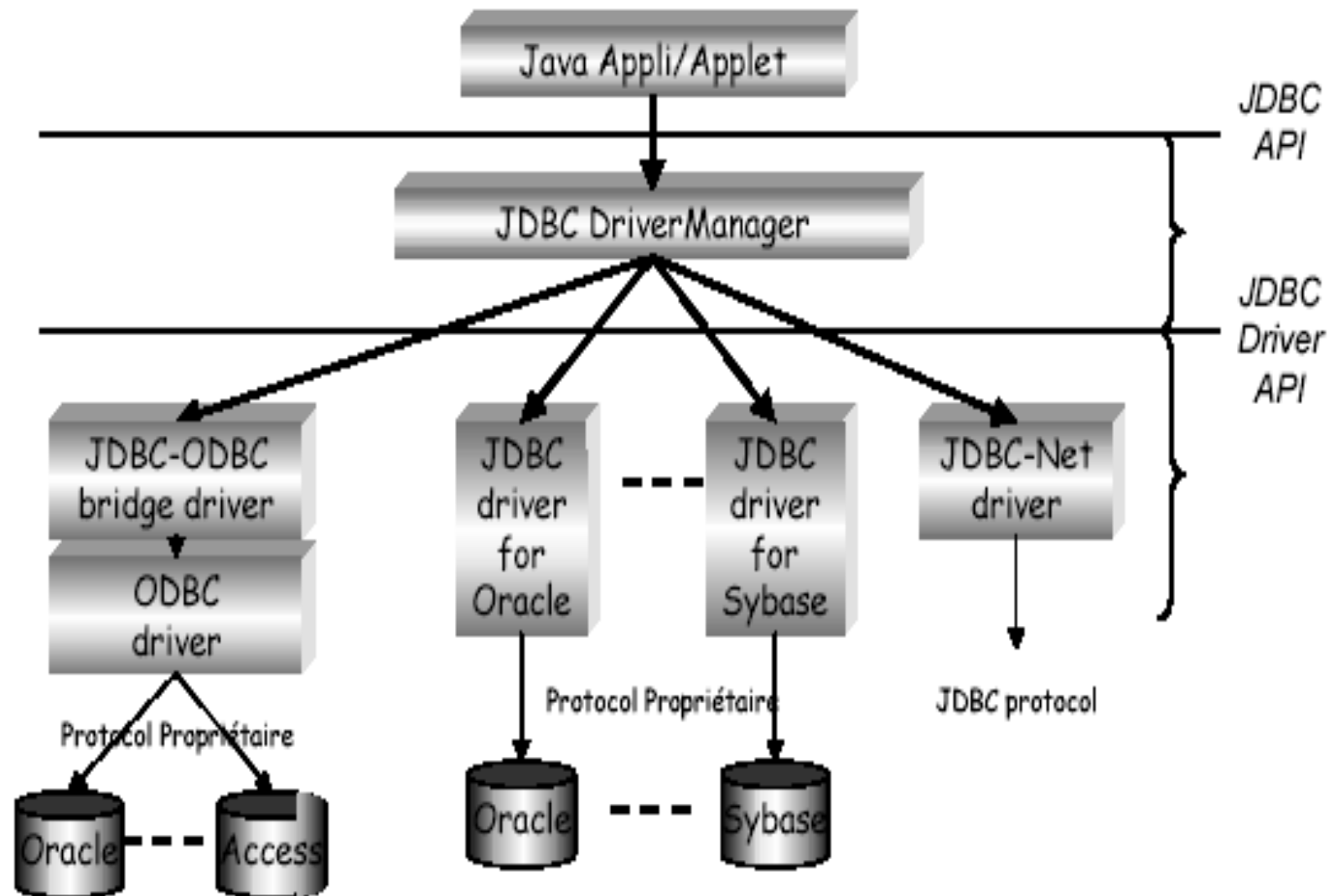
# API JDBC Java Data Base Connectivity

JDBC est une API fournie avec Java permettant l'accès à n'importe quelle base de données à travers un réseau

## Principe de fonctionnement

- Chaque base de données utilise un pilote (*driver*) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBDR.
- Ces drivers dits JDBC (un ensemble de classes et d'interfaces Java) existent pour tous les principaux constructeurs :
  - Oracle, Sybase, Informix, SQLServer, MySQL, MsAccess

# Pilotes JDBC





# Mise en oeuvre

## **Importer le package java.sql**

- 1. Charger le driver JDBC**
- 2. Établir la connexion à la base de données**
- 3. Créer une zone de description de requête**
- 4. Exécuter la requête**
- 5. Traiter les données retournées**
- 6. Fermer les différents espaces**

**utiliser la méthode de chargement à la demande  
forName() de la classe Class**

**Exemples :**

- **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**
- **Class.forName("oracle.jdbc.driver.OracleDriver");**
- **Class.forName(" com.microsoft.jdbc.sqlserver.  
SQLServerDriver ")**

```
try { Class.forName("com.mysql.jdbc.Driver")
```

```
System.out.println("Tout est OK"); }
```

```
catch (Exception e) { System.out.println("Erreur de drivers JDBC"); }
```



# Structure d'une URL

**<protocol>:<subprotocol>:<subname>**

**protocol : jdbc**

**subprotocol : driver utilisé (mysql)**

**subname : //<host>[:<port>][/<databasename>]**

**par exemple `jdbc:mysql://localhost:port (MySQL)/client` ou**

**pour une base de donnée locale `jdbc:odbc:NSD`**

**NSD** : source de données associée à la base de données client exemple Access



# Connexion à la base

Une fois l'URL est créée, on fait appel à la méthode `getConnection()` de `DriverManager` qui renvoie un objet de type `Connection`.

- 3 arguments :

- l'URL de la base de données
- le nom de l'utilisateur de la base
- son mot de passe

- Exemple:

`Connection connect =`

`DriverManager.getConnection(url,user,password);`

Le `DriverManager` essaye tous les drivers qui se sont enregistrés



# Exécution des requêtes

La première étape est de construire un objet *Statement*

**Statement** statement = connect.**createStatement()**;

Quand un objet *Statement* exécute une requête, il retourne un objet *ResultSet* (pour les requête de consultation).

L'exécution de la requête se fait par l'une des trois méthodes suivantes :

- **executeQuery** : pour les requêtes de consultation. Renvoie un *ResultSet* pour récupérer les lignes une par une.
- **executeUpdate** : pour les requêtes de modification des données (update, insert, delete) ou autre requête SQL (create table, ...). Renvoie le nombre de lignes modifiées.
- **execute** : si on connaît pas à l'exécution la nature de l'ordre SQL à exécuter

**ResultSet** rs = statement.**executeQuery**("SELECT \* FROM acc\_acc");





# Manipulation d'un ResultSet

Différentes méthodes vous permettent de récupérer la valeur des champs de l'enregistrement courant

Ces méthodes commencent toutes par `get`, immédiatement suivi du nom du type de données (ex: `getString`)

chaque méthode accepte soit l'indice de la colonne (à partir de 1) soit le nom de la colonne dans la table

`boolean getBoolean(int); boolean getBoolean(String); byte  
getByte(int); byte getByte(String); Date getDate(int); Date  
getDate(String); double getDouble(int); double getDouble(String);  
float getFloat(int); float getFloat(String); int getInt(int); int  
getInt(String); long getLong(int); long getLong(String); short  
getShort(int); short getShort(String); String getString(int); String  
getString(String);`



# Manipulation d'un ResultSet

## Exemple de parcours d'un ResultSet

- `String strQuery = "SELECT * FROM T_Users;"`
- `ResultSet rsUsers = stUsers.executeQuery(strQuery); while(rsUsers.next()) {`
- `System.out.print("Id[" + rsUsers.getInt(1) + "]" + "Pass[" + rsUsers.getString("Password") ); }`
- `rsUsers.close();`

Pour parcourir les enregistrements de n'importe quelle manière

Il faut passer des paramètres à la méthode `createStatement` de l'objet de connexion.

```
st = conn.createStatement(type, mode);
```



# Manipulation d'un ResultSet

**Type ==**

- **ResultSet.TYPE\_FORWARD\_ONLY**
- **ResultSet.TYPE\_SCROLL\_SENSITIVE**
- **ResultSet.TYPE\_SCROLL\_INSENSITIVE**

**Mode ==**

- **ResultSet.CONCUR\_READ\_ONLY**
- **ResultSet.CONCUR\_UPDATABLE**

**Pour le type scroll, on a les possibilités de parcours first, last , next et previous**

**Modification d'un ResultSet**

- **rsUsers.first(); // Se positionne sur le premier enregistrement**
- **rsUsers.updateString("Password", "toto"); // Modifie la valeur du //password**
- **rsUsers.updateRow(); Applique les modifications sur la base**



# Récupération des métadata

JDBC permet de récupérer des informations sur le type de données que l'on vient de récupérer par un SELECT (interface `ResultSetMetaData`), mais aussi sur la base de données elle-même (interface `DatabaseMetaData`)

Les données que l'on peut récupérer avec `DatabaseMetaData` dépendent du SGBD avec lequel on travaille

```
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
    String typeColonne = rsmd.getColumnTypeName(i);
    String nomColonne = rsmd.getColumnName(i);
    System.out.println("Colonne " + i + " de nom " + nomColonne +
        " de type " + typeColonne);
}
```

# Récupération des métadata

## DatabaseMetaData

```
private DatabaseMetaData metaData;  
private java.awt.List listTables = new List(10);  
...  
metaData = conn.getMetaData();  
// Récupérer les tables et les vues  
String[] types = { "TABLE", "VIEW" };  
  
ResultSet rs = metaData.getTables(null, null, "%", types);  
  
String nomTables;  
while (rs.next()) {  
    nomTable = rs.getString(3);  
    listTables.add(nomTable);  
}
```

Schéma

Catalogue



# PreparedStatement

**PreparedStatement** : Requêtes dynamiques pré-compilées plus rapide qu'un **Statement** classique

le SGBD n'analyse (compile) qu'une seule fois la requête pour de nombreuses exécutions d'une même requête SQL avec des paramètres variables

Gérée par le client JDBC.

La méthode `prepareStatement()` throws `SQLException` de l'objet `Connection` crée un `PreparedStatement`

`PreparedStatement ps =`

`con.prepareStatement("UPDATE emp SET sal = ? WHERE nom = ?");`

- les arguments dynamiques sont spécifiés par un "?"
- ils sont ensuite positionnés par les méthodes `setInt()` , `setString()` , `setDate()` , ... de `PreparedStatement` `setNull()` positionne le paramètre à `NULL` (SQL)



# PreparedStatement

```
int [] salaire= {175, 150, 60, 155, 90};  
String [] nom={"Ahmed", "Mohamed", "Salmi", "Haddou", "Alami"};  
for(int i = 0; i < 10; i++) {  
    ps.setFloat(1, salaire[i]);  
    ps.setString(2, nom[i]);  
    int count = ps.executeUpdate(); }  

```

ces méthodes nécessitent 2 arguments :

- le premier (int) indique le numéro relatif de l'argument dans la requête
- le second indique la valeur à positionner



# Commit / Rollback

Les SGBD assurent l'*atomicité* (tout ou rien) même si un crash survient au milieu d'une exécution de transaction.

Par défaut toutes les opérations de mise à jours (update, insert, delete) sont exécutées directement sur la base de données

- `cnx.setAutoCommit(true);`

La gestion de l'atomicité des transactions est assurée par les méthodes de l'objet Connection :

- `cnx.setAutoCommit(false);`
- `Cnx.commit()` // exécuter la transaction
- `Cnx.rollback()` // annuler la transaction

Exemple

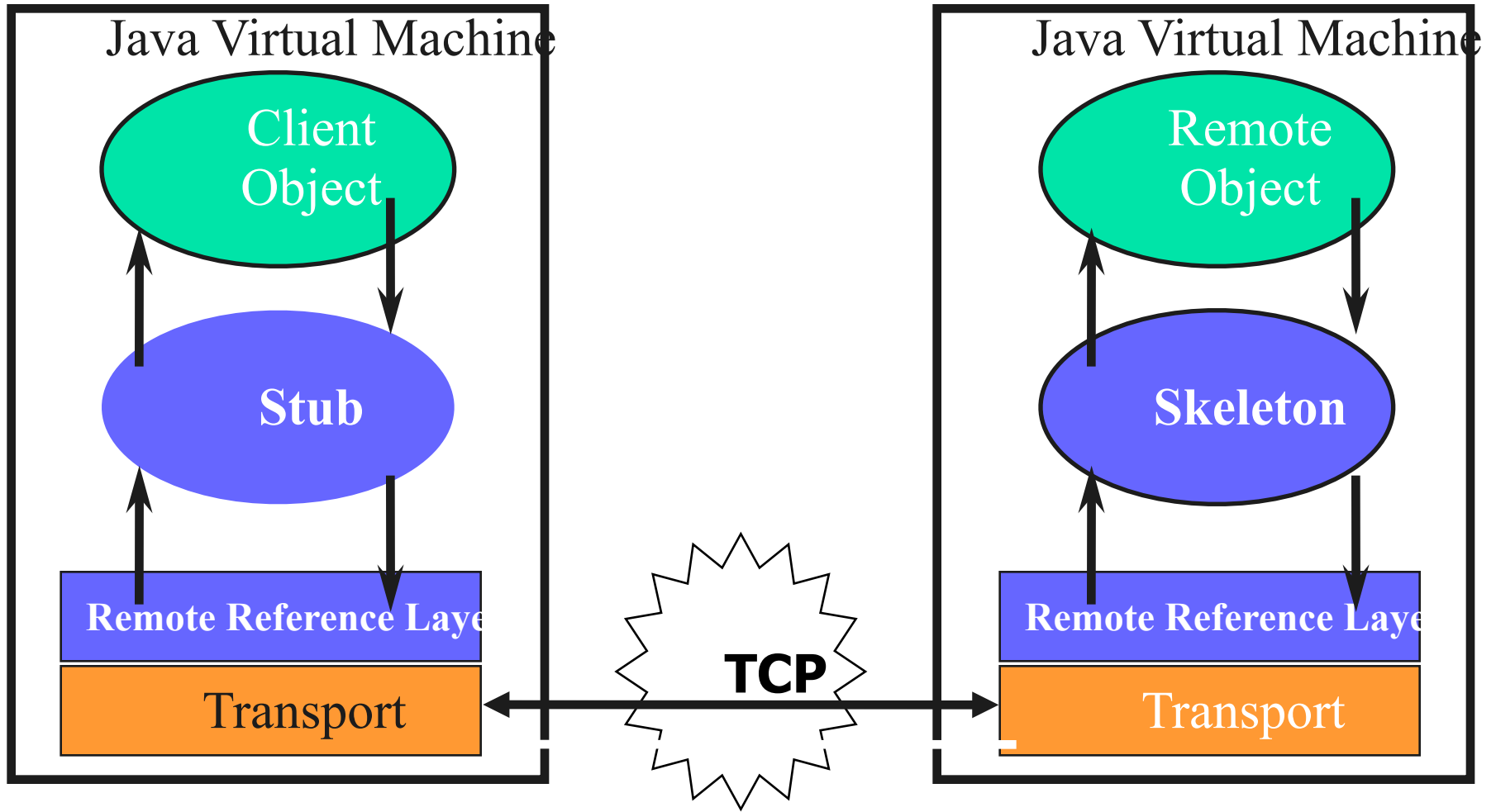




# Commit / Rollback

```
stmt=cnx.createStatement();
cnx.setAutoCommit(false);
try{ for(int i=1;i<=50;i++) {
    ▪ sql="insert into empl (Matricule, Nom, Salaire)" +
      " values('i','','0')";
    ▪ stmt.executeUpdate(sql); }
    ▪ conn.commit(); }
catch(Exception e) {
    ▪ e.printStackTrace();
    ▪ conn.rollback(); }
finally {
    ▪ conn.setAutoCommit(true); }
```

# Architecture RMI





# Client/Serveur Objet : RMI

## Le stub

### Classe sur le site client

- qui reçoit l'appel en mode local
- le transforme en appel distant en envoyant un message.
- reçoit les résultats après l'exécution
- retourne les paramètres résultats comme dans un retour de procédure.

## Le skeleton

### Classe sur le site serveur

- qui reçoit l'appel sous forme de message,
- fait réaliser l'exécution sur le site serveur par la procédure serveur (choix de la procédure)
- retransmet les résultats par message.

**Le stub et le skeleton assure la [dé]marshalisation**



## Mode opératoire coté serveur

1. L'objet serveur s'enregistre auprès du **Naming** de sa JVM (méthode bind ou *rebind*)
2. L'objet skeleton est créé, celui-ci crée le port de communication et maintient une référence vers l'objet serveur
3. Le Naming enregistre l'objet serveur, et le port de communication utilisé auprès du serveur de noms
4. L'objet serveur est prêt à répondre à des requêtes



## Mode opératoire coté client

1. L'objet client fait appel au **Naming** pour localiser l'objet serveur (méthode **lookup**)
2. Naming récupère les "références" vers l'objet serveur
3. Crée l'objet Stub et rend sa référence au client
4. Le client effectue l'appel au serveur par appel à l'objet Stub



# RMI : Mise en œuvre (1)

## Définition de l'interface de l'objet distant

- Les arguments locaux et les résultats d'une invocation distante sont toujours passés par copie et non par référence
- leurs classes doivent implémentées **java.io.Serializable** (sérialisation d'objets)

```
import java.rmi.*;
public interface addRemote extends Remote
{
    //Service distant
    public int add(int x, int y) throws RemoteException;
    // ...
}
```



# RMI : Mise en œuvre (2)

## Création de l'objet distant

```
import java.rmi.*;
import java.rmi.server.*;

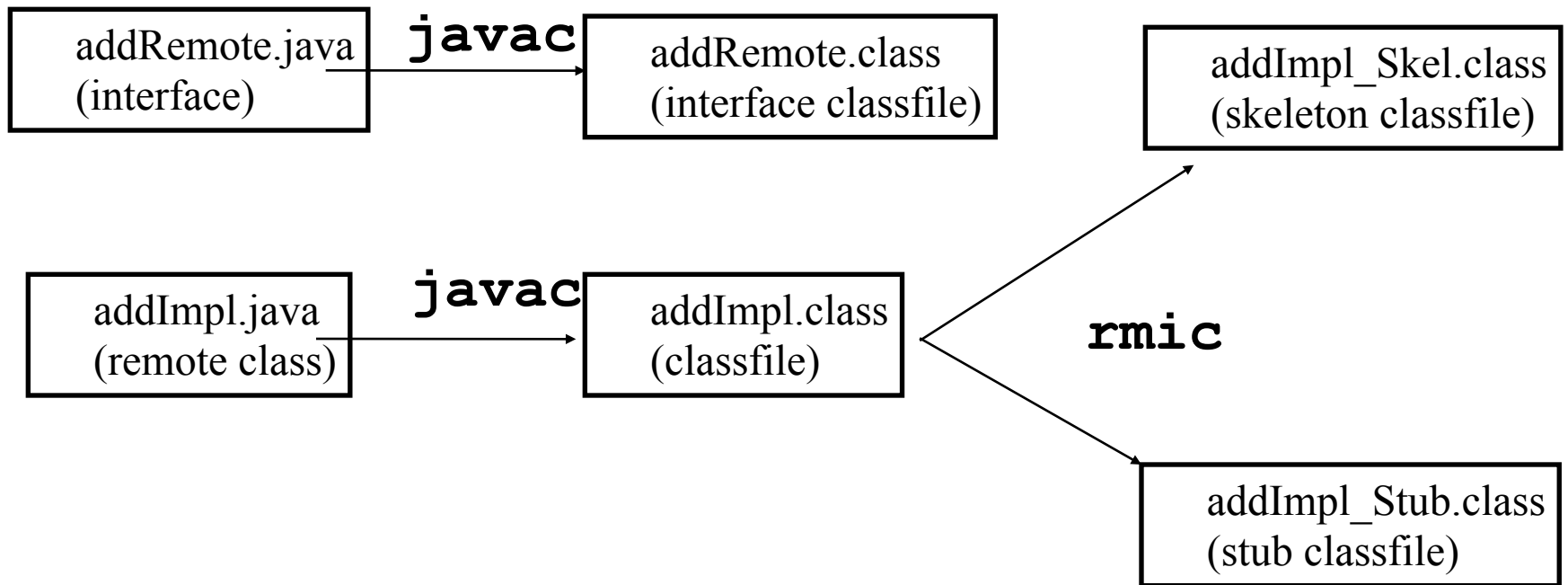
public class addImpl extends UnicastRemoteObject implements addRemote
{

    public addImpl() throws RemoteException { }

    public int add(int x,int y) throws RemoteException
    {
        return x + y;
    }
}
```

# RMI : Mise en œuvre (3)

Génération du Stub et Skeleton ancien JDK 1.2



Génération du Stub et Skeleton est dynamique à partir JDK 1.5





# RMI : Mise en œuvre (4)

## Le Serveur méthode (**classique**)

- Lancer rmiregistry en mode externe
- ```
try {  
    addImpl add = new addImpl();  
    Naming.rebind("key", add);  
    System.out.println("Serveur en attente :");  
} catch (Exception re) { re.printStackTrace();}
```

## Le serveur méthode 2 (**préférée**)

- Le rmiregistry est lancé à partir du code

```
try {  
    int port=2020;  
    Registry registry = LocateRegistry.createRegistry(port);  
    addImpl add = new addImpl();  
    Naming.rebind("key", add);  
}
```



# RMI : Mise en œuvre (5)

## Le Client

```
import java.rmi.*;
public class client {
    public static void main(String args[]) {
        int port=2020;
        String URL = "rmi://localhost:"+port+"/key";
        try {
            // Récupération d'un stub sur l'objet serveur.
            addRemote obj = (addRemote)Naming.lookup(url);

            // Appel du service (ici add).
            System.out.println(obj.add(10,20));
        } catch (Exception e) {e.printStackTrace(); }    }}
```



# RMI : sécurité

## Implémentation d'une politique de sécurité type firewall (fichier de règles)

- gestion fine des droits suivant les autorisations à accorder suivant les opérations
- gestion au niveau du serveur et du client
- Exemple (security.policy)
- ```
grant { permission java.net.SocketPermission "*:1024-65535","connect";  
        permission java.net.SocketPermission "*:80", "connect"; };
```
- `permission java.security.AllPermission;` est le default
- Au lancement du serveur on spécifie l'emplacement du fichier policy
- `-Djava.security.policy=security.policy`

Utilisation du tunneling avec SSL en adaptions la couche transport de RMI par l'utilisation de `SSLClientSocketFactory`, `SSLServerSocketFactory`



# Les Servlet et Beans



# Structure d'une application Web en JEE

## WebContent/

- \*.html, \*.png, \*.jsp,...,\*.war et \*.ear

## src/

- \*.java (bean, servlet, ..)

## WEB-INF/lib/

- \*.jar ( drivers JDBC, jsf, ... )

## WEB-INF/web.xml

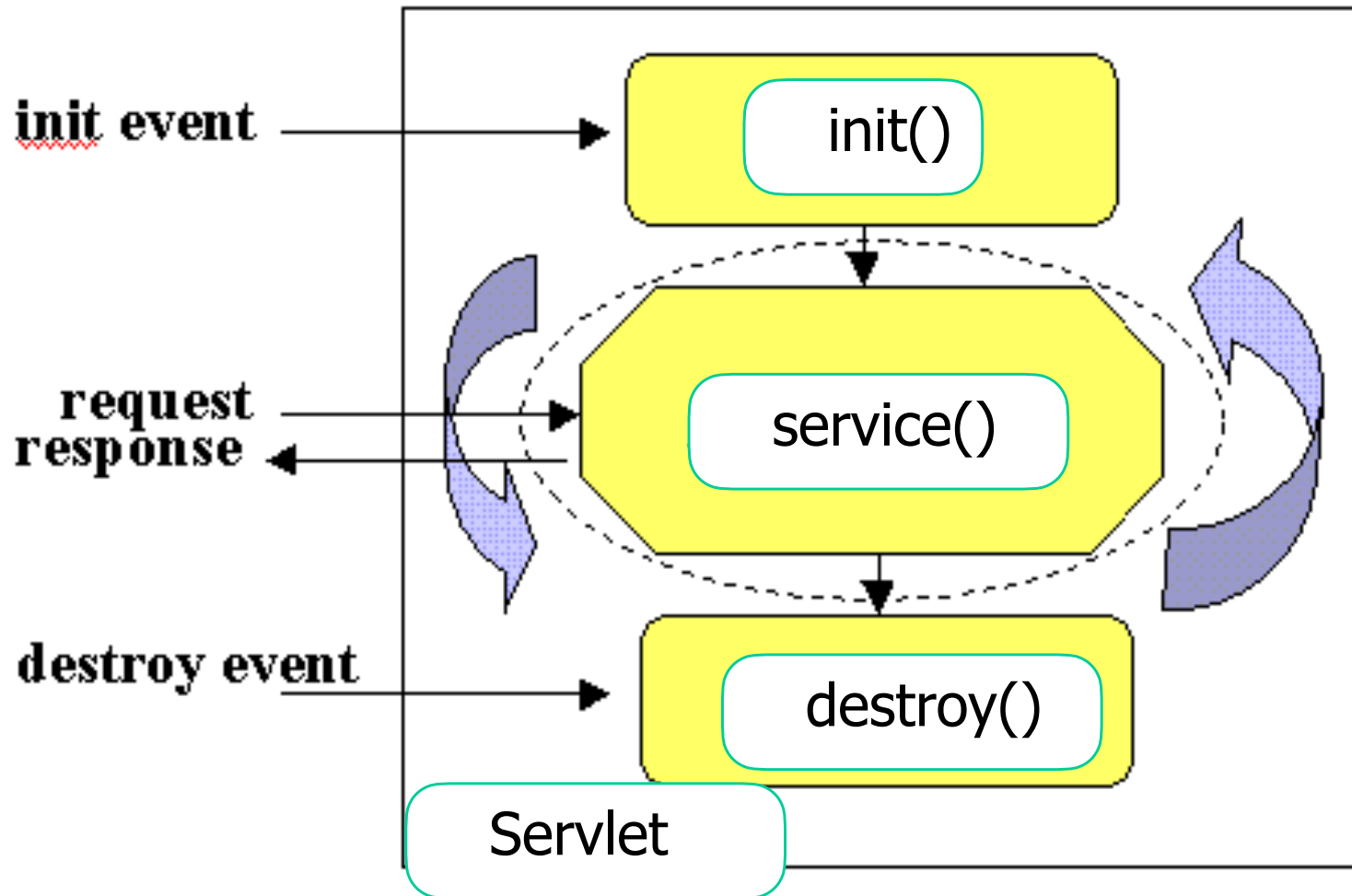
- Fichier de déploiement, Paramétrage des servlets, types MIME additionnels.

## WEB-INF/tlds/

- \*.tld décrivant les TagLibs

# Cycle de vie d'une servlet

Une servlet est une classe Java qui implémente les primitives du protocole HTTP





# Une servlet simple

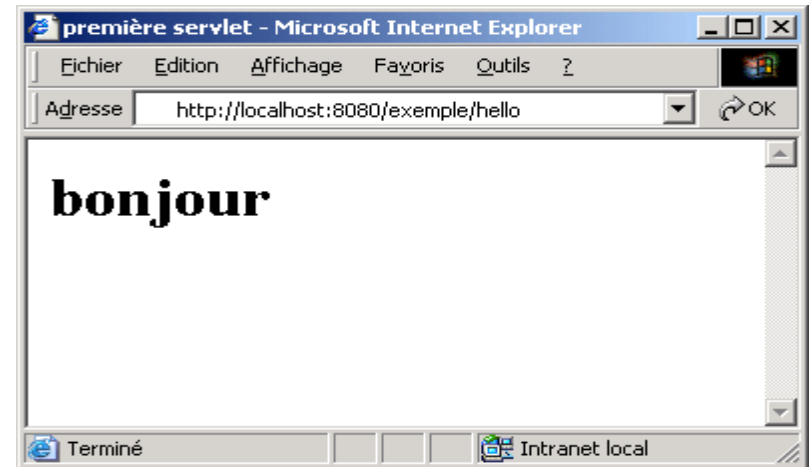
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorldExample extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>première servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>bonjour</h1>");
        out.println("</body>");
        out.println("</html>");    }}
```

# Déploiement d'une servlet

Une application web doit être paramétrée dans le cas des servlets, par un fichier **web.xml**:

```
<web-app >
  <servlet>
    <servlet-name>helloervlet
  </servlet-name>
    <servlet-class>
test.servlet.HelloWorldExample
  </servlet-class>
</servlet>
  <servlet-mapping>
    <servlet-name>helloervlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```







# HttpServletRequest/HttpServletResponse

## HttpServletRequest

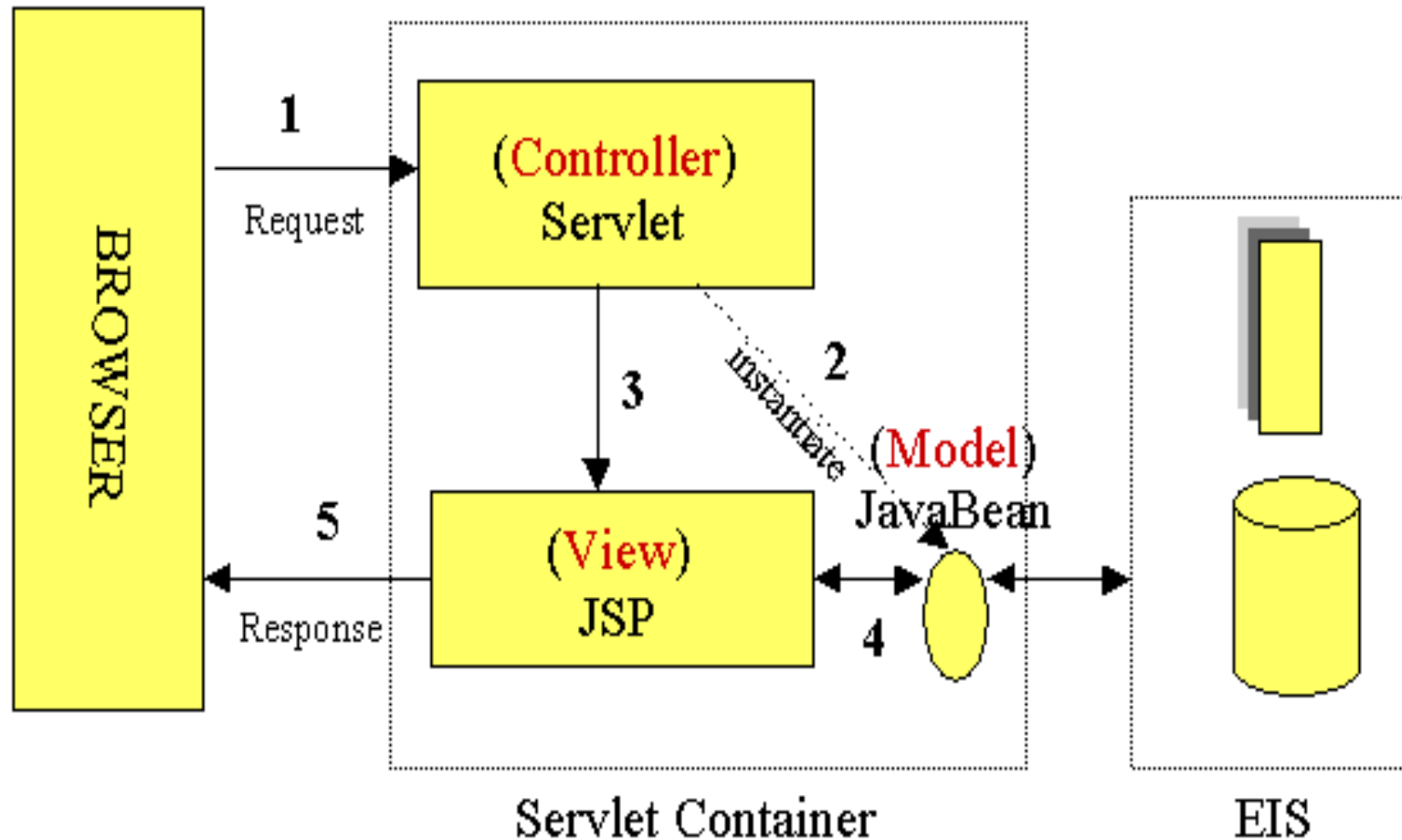
`getParameter(paramName), getParameteres(),getRequestDispatcher()  
getAttribute(attName), getAttributes(),setAttribute(name,value),  
getServletContext(), getCookies(), getSession(), getContentLength,  
getContentType`

## HttpServletResponse

`addCookie(Cookie cookie), sendError(int sc),sendRedirect(String url)  
sendError(int sc,String msg), getWriter(), setCharacterEncoding,  
setContentLength, setContentType, setLocale`

# Modèle MVC

## MVC Design Pattern





# HTML/Servlet

Deux principaux tags peuvent être utilisés pour mettre en œuvre un formulaire HTML **Form** et **Input**

Le tag **<Form>...</Form>** permet de délimiter l'espace d'un formulaire

Deux paramètres sont importants pour la suite du cours

**Action** : indique quelle est la ressource chargée de traiter les données

**Method** : indique la méthode de soumission des données du formulaire au serveur

**POST** : les données sont stockées dans le corps de la requête HTTP. C'est la méthode que nous préférons utiliser

**GET** : les données sont concaténées à la suite de l'URL. Cette dernière étant passée dans l'en-tête de la requête HTTP



## HTML/Servlet (2)

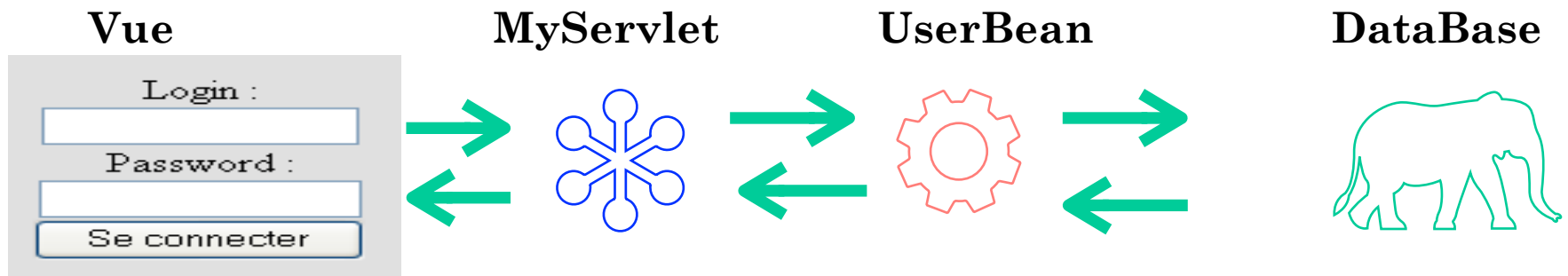
**<INPUT>** Permet d'introduire un champ de saisie (zone de texte, zone de mot de passe, cases à cocher, ...).

**Pour connaître la nature du composant de saisie on utilise les paramètres**

- **Name** permet de nommer le composant afin de retrouver la donnée associée dans la requête HTTP.
- **Type** : permet de définir la nature du composant de saisie. Parmi les types les plus courants, on retrouve : text, password, checkbox, radio, hidden, submit, button, reset et file.
- **Value** : permet de spécifier la valeur initiale pour ce composant

## HTML/Servlet (3)

```
<FORM Action="Myservlet" method="post">  
Login :<INPUT Type="text" Name="login"/> <br/>  
Password :<INPUT Type="password" Name="password"/><br/>  
<INPUT Type="submit" Value="Se connecter"/>  
</FORM>
```





# Gestion des erreurs

- ◆ Redéfinir une erreur HTTP

```
<error-page>  
    <error-code> 404 </error-code>  
    <Location> /ErrorHandler </location>  
</error-page>
```

- ◆ Redirection d'une erreur spécifique

```
<error-page>  
    <Exception-type>  
        javax.servlet.ServletException  
    </Exception-type>  
    <Location> /ErrorHandler </Location>  
</error-page>
```

- ◆ Un seul gestionnaire d'erreurs pour toutes les exceptions

```
<Error-page>  
    <Exception-type> java.lang.Throwable </Exception-type>  
    <Location> / ErrorHandler </ location>  
</Error-page>
```

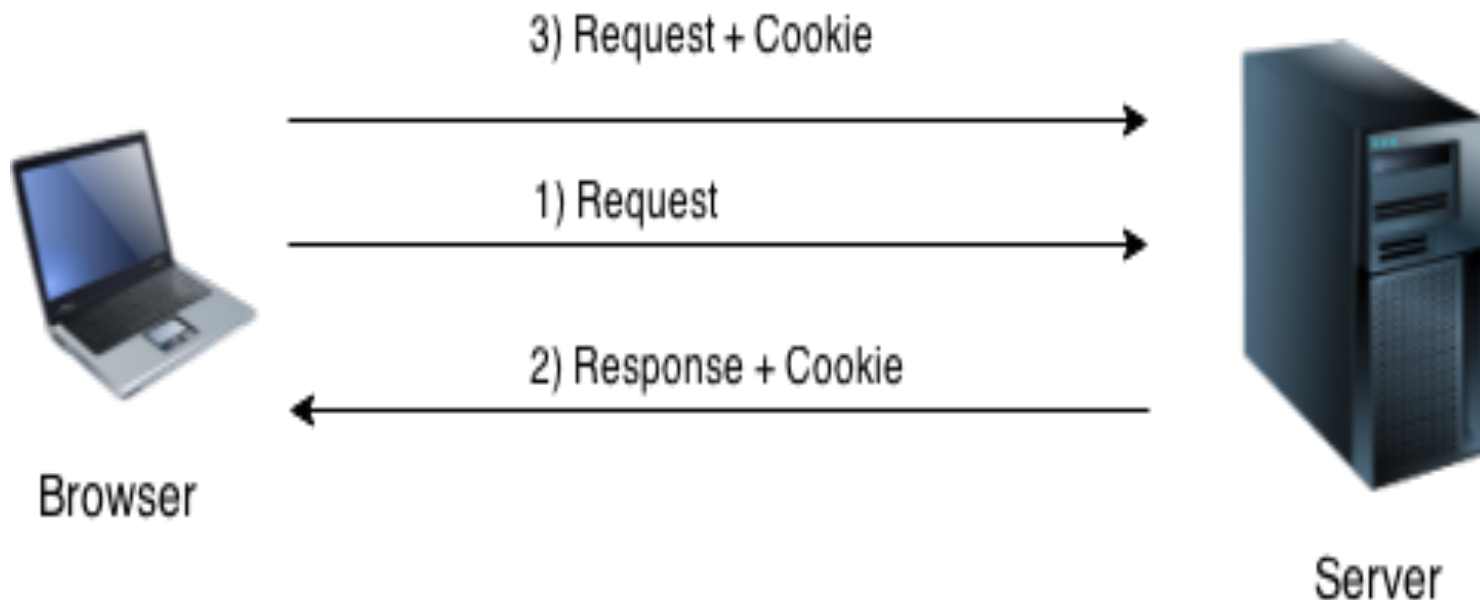
# Cookies

Les cookies sont des objets stockés chez le client pour maintenir son état.

Il existe deux types de cookies :

persistantes (indépendantes de la session client)

non persistantes (dépendent de la session client)





# Cookies

---

C'est la classe **javax.servlet.http.Cookie** deux constructeurs sont disponibles :

- **Cookie(String name, String value)**
- **Cookie()**

Pour ajouter une cookie chez le client on utilise la méthode **addCookie(Cookie ck)** de l'objet response

Pour récupérer les cookies de chez le client on utilise la méthode public **Cookie[] getCookies()** de l'objet request

```
Cookie ck=new Cookie("user", "lahmer");  
response.addCookie(ck);  
Cookie ck[]=request.getCookies();  
for(int i=0;i<ck.length;i++){  
    out.print("<br>" +ck[i].getName()+" "+ck[i].getValue());}
```





# DataSource et JNDI

## Fichier context.xml de Tomcat

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

### <Context>

```
<Resource  
name="jdbc/dbname"  
driverClassName="Driver"  
url="jdbc:mysql:dbname"  
type="javax.sql.DataSource"  
username="user"  
password="password"  
auth="Container"  
maxActive="8" />
```

### </Context>

## Fichier web.xml

```
<resource-ref>  
<res-ref-name>jdbc/dbname  
</res-ref-name>  
<res-type>  
javax.sql.DataSource</res-type>  
<res-auth>Container</res-auth>  
</resource-ref>
```

```
Context ctx = new InitialContext();  
DataSource ds = (DataSource)ctx.lookup(  
    "java:comp/env/jdbc/dbname");  
Connection conn = ds.getConnection();
```



# JSP (Java Server Page)

Combinaison entre java et html pour la production de site web dynamique

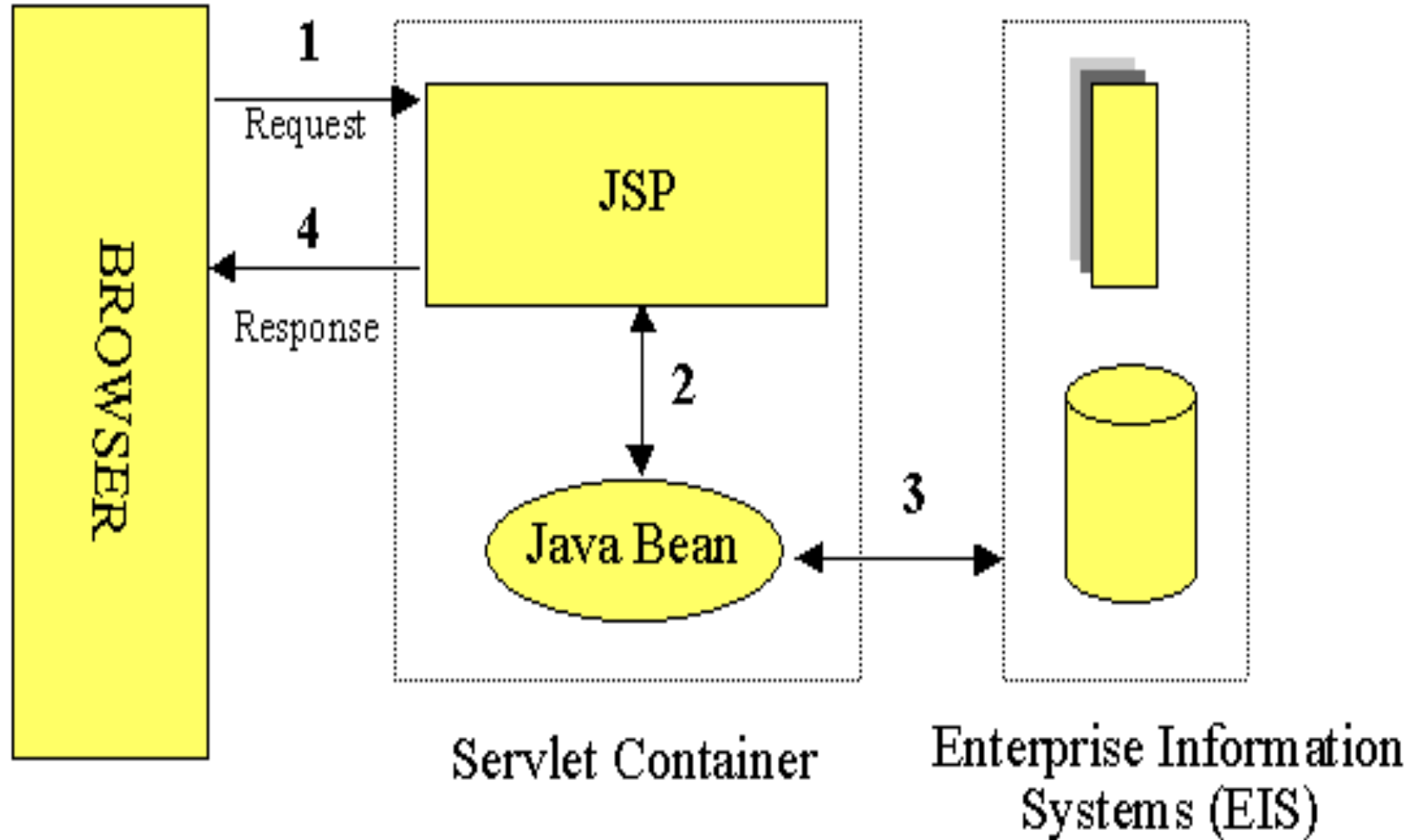
Est une solution sun pour concurrencer ASP et PHP

Le code JSP peut être intégrer dans du HTML, DHTML, XML et WML

Le code JSP est compilé en servlet puis exécuté par un moteur de Servlet type Tomcat de jakarta

La compilation se fait lors du premier appel de la page JSP (plus de temps pour l'affichage)

# Déploiement de JSP





# Éléments de code JSP

**Directives de la forme** `<%@ ... %>` elles sont placées au début de la page : import et des propriétés supplémentaires

## Scriptlets

- **Expressions de la forme** `<%= expr %>` pour afficher chez le client la valeur de expr
- **Scriptlets de la forme** `<% code %>` pour la déclaration de variables de fonctions (syntaxe java).
- **Déclarations de la forme** `<%! code %>` code statique
- **Du commentaires** `<%-- ... --%>`

## Actions standards

- **Exemple:** `<jsp:useBean> ... </jsp:useBean>`

**Des variables comme** `request`, `response`, `out` et d'autres



# Directives

Elles ont la forme :

```
<%@ name attribute1="...", attribute2="..." ... %>
```

page

**Specifie les propriétés page**

include

**Inclure un fichier**

taglib

**Spécifier un tag personnalisé**



# Directives: exemples

## Importer des packages

- **<%@ page import="java.util.\*,java.sql.\*" %>** Ou
- **<%@ page import="java.util.\*" %>**  
**<%@ page import="java.sql.\*" %>**

**<%@ page contentType="text/plain" %>** type (MIME) le contenu de la page retournée.

**<%@ page session="false" %>** précise que la page ne participe pas à une session. La variable session n'est plus accessible. Le défaut est true ;

**<%@ errorPage="Erreur.html"%>** précise que la page à retourner en cas d'exception est Erreur.html (URL relatif) ;

**<%@ isErrorPage="true" %>** précise que la page est une page d'erreur associée une autre JSP.



# Directives supplémentaires

**<%@ extends="A" %>** précise que la serviette générée doit étendre la classe A plutôt que de celle définie par défaut dans le moteur de JSP.

**<%@ info="Information" %>** définit le message retourné par la méthode `getServletInfo()` ;

## Inclure un fichier

- **<%@ include file="header.html" %>**
- On peut spécifier le chemin par rapport à la page jsp

## Redirection

- **<jsp:forward page="page.jsp" />** redirige la requête vers une autre ressource :
- .jsp, servlet, ...



# Expressions de script

Pour les expression de la forme `<%= expr %>`, `expr` est évalué puis affiché sous forme de chaîne de caractère dans la sortie standard out.

Dans une servlet il peut être équivalent à :

```
PrintWriter out = response.getWriter();  
...  
out.print(expr);
```

## Exemples

```
<%= "Salut" %>
```

```
Le nom est <%= request.getParameter("name") %>
```

```
La date est <%= new java.util.Date() %>
```





# Exemple de Scriptlet

## Valider la saisie d'un champ

```
<% String name = request.getParameter("name");  
    if (name == null)  
    { %>  
        <h3>Remplir le nom SVP</h3>  
<% }  
    else  
    { %>  
        <h3>Bien Venu <%= name %></h3>  
<% } %>
```

Il y a trois scriptlet et une expression



# Déclaration

**Pour une déclaration** `<%! declarations %>` . Les variables déclarées sont placées à l'extérieure de la méthode `_jspService()`. Typiquement se sont des variables ou des méthodes d'instance.

## Déclaration d'une variable

```
<%! private int count = 0; %>
...
The count is <%= count++ %>.
```

## Déclaration d'une méthode

```
<%!
private int toInt(String s){
    return Integer.parseInt(s);
%>
```

# Exemple de page JSP

```
<html>
<head><title>JSP Test</title></head>
<body>
<center>

<h1>JSP Test</h1>
Time: <%= new java.util.Date() %>

</center>
</body>
</html>
```

**<%= ... %>** est  
équivalente à  
**<% out.print(...); %>**

**JSP Test**

Time: Wed Mar 05 10:37:07 EST 2003



# Persistence des variables

```
<html>
<body><center>
<%!int compteur=0;%>
<%int t=0;%>
<h1>Variable persistante :<%=compteur++%> </h1>
<hr>
<h2>Variable non persistante :<%=t++%> </h2>
</center>
</body>
</html>
```

La valeur de la variable compteur s'incrémente à chaque appel de la page, alors que le contenu de t reste invariant

Le contenu d'une variable persistante est réinitialisé une fois on change le contenu de la page ou on redémarre Tomcat



# Traitement des paramètres d'une page HTML

```
<html>
<head><title>Test Jsp</title></head>
<body>
<h1>JSP Processing form with GET</h1>
<form action="Form.jsp" method="GET">
Login: <input type="text" name="login"><br />
Password: <input type="text" name="passwd">
<p><input type="submit" name="button"
      value="Valider"></p>
</form>
</body>
</html>
```



# Le fichier Form.jsp

```
<html>
<head>
  <title>Appel de JSP</title>
</head>
<body>
<center><h1>resultats de la page JSP</h1>
Bien Venu <%= request.getParameter("login") %>
<%= request.getParameter("passwd") %>
</center></body>
</html>
```



# Les Objets implicites

**request** de type **HttpServletRequest** ;

**response** de type **HttpServletResponse** ;

**out** de type **JspWriter**, sous-type de **PrintWriter** ;

**session** de type **HttpSession** ;

**application** de type **ServletContext** ;

**config** de type **ServletConfig** ;

**pageContext** de type **PageContext** (permet de partager des données entre pages) ;

**page** équivalent à `this`

**exception** définie dans une page jsp d'erreur pour le traitement des erreurs



# Synchronisation

**Par défaut la méthode service d'une page jsp est multithread.**

**Donc il est indispensable d'appliquer un mécanisme de synchronisation sur les objets partagés**

▪ **<%**

```
synchronized (application) {
```

```
SharedObject foo = (SharedObject)
```

```
application.getAttribute("sharedObject");
```

```
foo.update(someValue);
```

```
application.setAttribute("sharedObject",foo);} 
```

**%>**





# Gestion des erreurs

**Par défaut une page jsp est une page qui ne peut pas traiter les erreurs et donc utiliser l'objet exception**

**Pour créer une page d'erreur il faut utiliser le tag**

- **`<%@ page isErrorPage="true" %>`**

**L'affichage de l'erreur rencontrée peut se faire à l'aide de l'objet exception**

- **`<%= exception.toString() %>`**



# Gestion des erreurs

L'appel de la page d'erreur se fait dans une page qui peut générer des erreurs

- `<%@ page errorPage="errorpage.jsp" %>`

```
<!-- source.jsp -->

<%@ page
errorPage="error.jsp" %>

<html>

<body><%
if (..) else {
throw new
Exception("Erreur!");}%>

</body> </html>
```

```
<!-- error.jsp -->

<%@ page
isErrorPage="true"%>

<html>

<body><font color=red>

<%=
exception.toString() %>

</font>

</html> </body>
```



# Composante JavaBean

**Un bean en java est une classe**

- **Avec un constructeur par défaut**
- **Des méthodes de la forme getYYY et setYYY ou YYY est le nom de la propriété de la classe**
- **ne doit pas avoir d'attribut public**
- **Si la propriété est de type booléen la méthode getProp() est remplacée par isProp().**

**Une instance d'un bean peut être construite à partir de JSP par**

- **`<jsp:useBean id = "instance" class = "beanClass" />`**
- **Si l'instance existe déjà alors cette instruction n'a aucun effet**



# Bean : Hello World

---

Package Test

```
public class hello{  
    private String message; // propriété  
  
    public hello()  
    { message = "Hello World"; }  
  
    public String getMessage()  
    { return message; }  
  
    public void setMessage(String m)  
    { message = (m == null) ? "Hello World" : m;  
    }  
}
```



# Communication avec le bean

## Création du bean avec la propriété par défaut

```
<jsp:useBean id="h" class="test.hello" />
```

## Création du bean et modification de la propriété

```
<jsp:useBean id="h" class="test.hello" >  
<jsp:setProperty name="h" property="message"  
    value="Hello JSP World" />
```

## La lecture et l'affichage de la propriété

```
<h1>La valeur du bean est</h1>
```

```
<p><jsp:getProperty name="h" property="message" /></p>
```



# Communication avec le bean

Pour associer tous les paramètres aux propriétés de même nom

- **<jsp:setProperty** name="nom" property="\*" />

Ces associations entraînent une conversion implicite des chaînes de caractères valeurs des paramètres dans le type des propriétés

Par défaut, une instance du JavaBean est créée au moment de la requête

Il est également possible de partager une même instance entre les requêtes en ajoutant l'attribut scope à la balise <jsp:useBean ... />

- **<jsp:useBean** id="nom" class="paquetage.Class" scope="application"> <jsp:useBean />

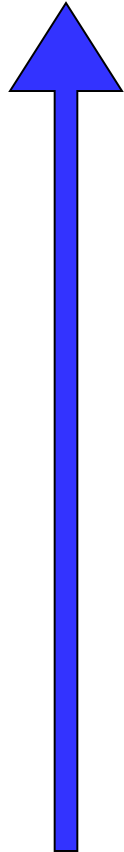
Par défaut scope=page sinon on peut utiliser request, session ou application



# Porté d'un bean : scope

---

plus visible



Moins visible

application

Session

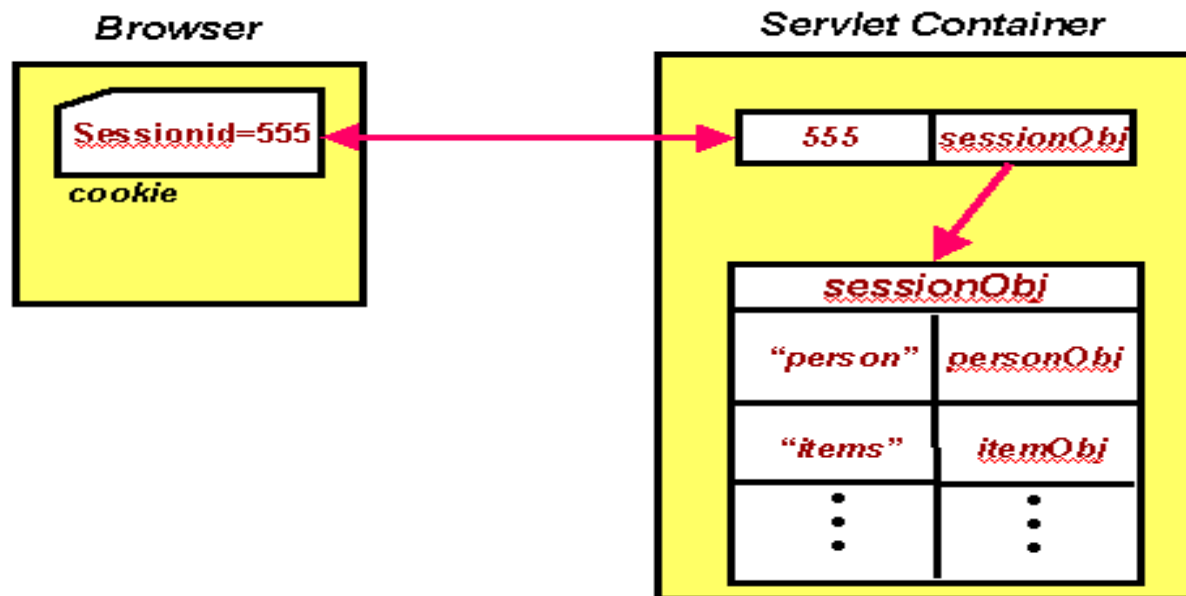
Request

Page

# Interaction entre JSP et Servlets

Par défaut toutes les pages JSP et servlets participent à une session HTTP  
L'objet session est une bonne place pour stocker les beans et les objets partagés

Chaque session est identifiée par un ID et stockée dans le navigateur comme une cookie







# Interaction entre JSP et Servlets

Une page JSP peut ne pas participer dans une session si : `<%@ page session="false" %>`

Pour stocker un objet dans une session on utilise la méthode `putValue` ou `setAttribute`

```
<% Foo foo = new Foo(); session.setAttribute("foo", foo); %>
```

La récupération de l'objet `foo` par une autre page JSP

```
<% Foo myFoo = (Foo) session.getAttribute("foo"); %>
```

A partir d'une servlet :

```
HttpSession session = request.getSession();
```

```
Foo myFoo = (Foo) session.getAttribute("foo");
```

## Pour rediriger un traitement d'une requête vers une autre page jsp

- `<jsp:forward page="unePage.jsp" />`

On peut également faire une redirection vers une servlet ou une page html statique

Il est possible d'intégrer des paramètres dans le tag `<jsp:forward>`

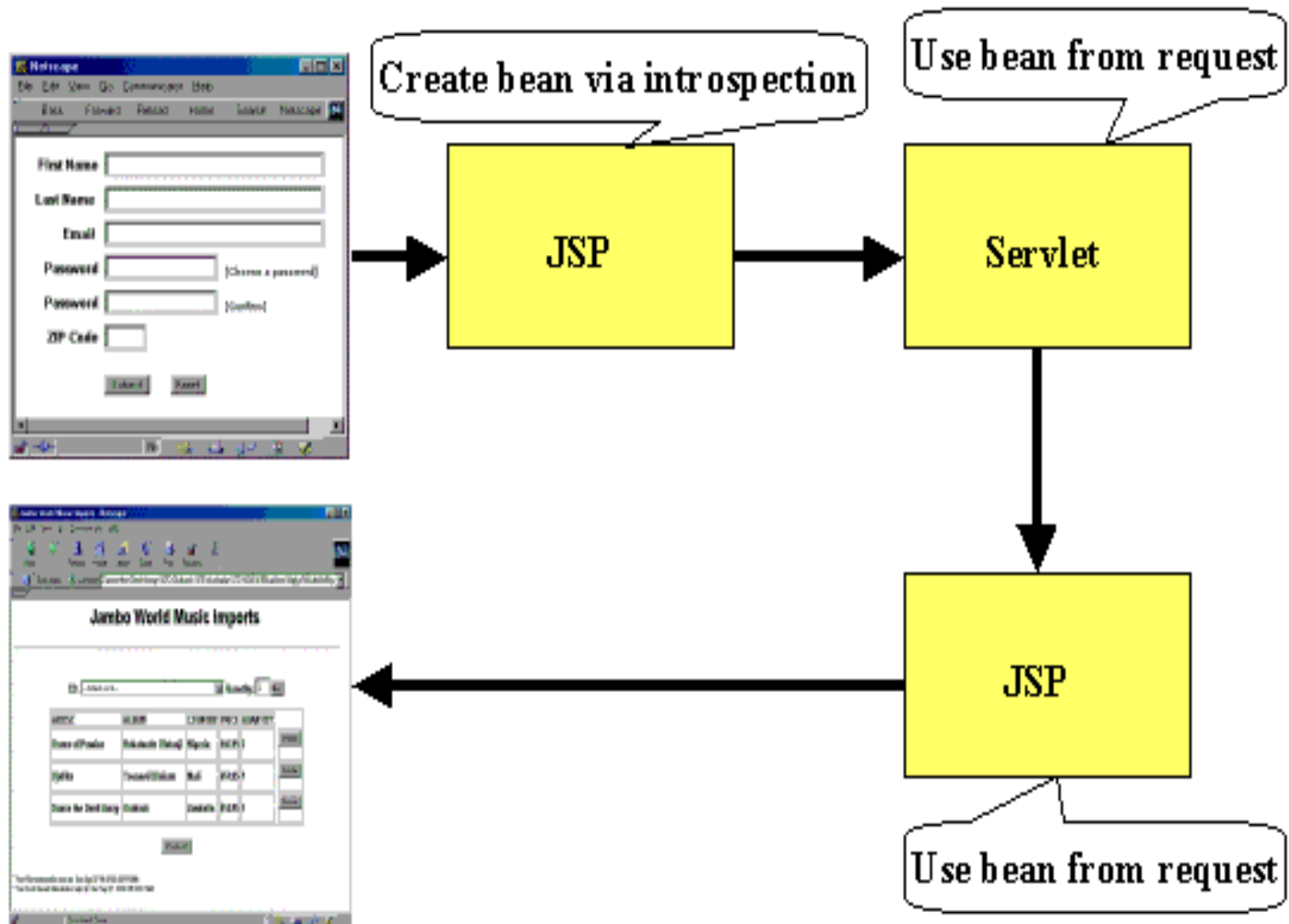
```
<jsp:forward page="unePage" >
```

```
<jsp:param name="nom1" value="valeur1"/>
```

```
<jsp:param name="nom2" value="valeur2"/>
```

```
</jsp:forward>
```

# Interaction JSP Servlets : la redirection





# Interaction JSP Servlets : la redirection

```
<jsp:useBean id="fBean" class="FormBean"
scope="request" />
<jsp:setProperty name="fBean" property="*" />
<jsp:forward page="/servlet/JSP2Servlet" />
```

```
public void doPost (HttpServletRequest request,
    HttpServletResponse response) {
    try {
        FormBean f = (FormBean) request.getAttribute ("fBean");
        // traiter le f
        getServletConfig().getServletContext().getRequestDispatcher("/jsp/
            Bean2.jsp").forward(request, response); }
    catch (Exception ex) { ... }}
```



# Tag Library

## Avantages

- Étendre les balises **JSP** standards
- Balises spécifiques à un cas d'usage
- Réduire l'utilisation des scriptlets
- Améliorer la lisibilité de la page JSP
- Libérer les concepteurs de pages du code Java
- Il existe des tags librairies prédéfinies "jstl.jar", mais on peut créer un tag personnalisé

## Mise en oeuvre

- Importer `javax.servlet.tagext.Tag`
- Étendre `TagSupport` ou `BodyTag`
- Fichier **file.tld** descripteur du tag
- Page **JSP** utilisant la nouvelle balise



# Tag Librairie standard

---

Depuis la version *JSP 1.2*

Spécification développée par le groupe d'experts *JSR 52*

Collection de *Tag Bibliothèques personnalisées* qui implémentent la plus part des *fonctions communes aux pages web*:

- Itérations et conditions (core)
- Formatage des données (format)
- Manipulation de XML (xml)
- Accès au bases de données (sql)

Utilisation du langage EL (Expression Language)



# EL : Expression Language

---

Un identificateur dans EL fait référence à une variable retourné par l'appel de **pageContext.findAttribute(identificateur)** et qui est dans la portée (scope): **page**, **request**, **session** ou **application**.

**`${ var } = pageContext.getAttribute("var")`**

Objets implicites:

- **pageScope**, **requestScope**, **sessionScope**, **applicationScope**

Opérateurs relationnels (**==** **!=** **<** **>** **<=** **>=**), arithmétiques (**+** **-** **\*** **/** **%**) et logiques (**&&** **||** **!**)

L'opérateur **[ ]** pour accéder au objets de type **Map**, **Array** et **List**

Ex: **param["p1"]** equiv **param.get("p1")**



# JSTL : librairies de bases

---

Chaque librairie est désignée par une **URI**

Librairie	URI	Préfixe
core	<a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a>	c
Format	<a href="http://java.sun.com/jsp/jstl/fmt">http://java.sun.com/jsp/jstl/fmt</a>	fmt
XML	<a href="http://java.sun.com/jsp/jstl/xml">http://java.sun.com/jsp/jstl/xml</a>	x
SQL	<a href="http://java.sun.com/jsp/jstl/sql">http://java.sun.com/jsp/jstl/sql</a>	sql
Fonctions	<a href="http://java.sun.com/jsp/jstl/functions">http://java.sun.com/jsp/jstl/functions</a>	fn





# JSTL : Librairie Core

Gère les actions de base d'une application web :

- L'affichage de variable
- La création/modification/suppression de variable de scope
- La gestion des exceptions
- Les itérations

Déclaration de la librairie 'core' :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core " prefix="c" %>
```

Affichage d'une variable


- `<c:out/>` : Afficher une expression
- `<c:out value="$expression" default="Inconnu"/>`
- `<!-- Même chose en utilisant le corps du tag : -->`
- `<c:out value="${header['user-agent']}">`
- Inconnu
- `</c:out>`

# JSTL : Librairie Core

## Affichage

- `<c:out value=" expression " />`  `<%= expression %>`
- `<c:out value="$expression" default="Inconnu"/>`

## Affectation

- `<c:set value="value" var=" varName " scope=" session " />`
-  `<%= pageContext.setAttribute("varName",value,PageContext.SESSION_SCOPE) %>`
- **Cas d'un Bean**
- `<c:set target="${session['varName']}" property="name" value="new value"/>`
- `<!-- vous pouvez utiliser le corps du tag : -->`
- `<c:set target="${session['varName']}" property="name">`
- Nouvelle valeur de la propriété "name "
- `</c:set>`



# JSTL : Librairie Core

**<c:remove/>** : Supprimer une variable de scope

- **<!-- Supprime l'attribut "varName" de la session -->**
- **<c:remove var="varName" scope="session"/>**

**<c:catch/>** : Intercepter les exceptions

- **<!-- Ignorer toutes les exceptions d'une partie de la page : -->**
- **<c:catch>**
- **<c:set target="beans" property="prop" value="1"/>**
- **</c:catch>**
- **<!-- Stocker dans le scope page l'exception intercepté : -->**
- **<c:catch var="varName">**
- **<c:set target="beans" property="prop" value="1"/>**
- **</c:cath>**

*Si var est spécifié et qu'aucune exception n'est lancée, alors la variable de page "var" sera supprimée.*



# JSTL : Librairie Core

## Condition

- `<c:if test="${user.visitCount == 1}">`
- `<c:out value="Première visite. Bienvenue!" />`
- `</c:if>`

## Selection

- `<c:choose>`
- `<c:when test="${value==1}"> value vaut 1 (Un) </c:when>`
- `<c:when test="${value==2}"> value vaut 2 (Deux) </c:when>`
- `<c:otherwise>`
- `value vaut ${value} (?)`
- `</c:otherwise>`
- `</c:choose>`

## Itérations

- `<c:forEach var="user" items="sessionScope.members" [begin] [end] [step]>`
- `<c:out value="nom: ${user.name}" />`
- `</c:forEach>`



# JSTL : Librairie SQL

Cette librairie facilite l'accès aux bases de données via le langage SQL au sein d'une page JSP.

Déclaration de la librairie 'SQL' :

- `<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>`

## Connexion

- `<sql:setDataSource var="Db" scope="page"`
- `url="jdbc:mysql://localhost/base"`
- `driver="org.gjt.mm.mysql.Driver"`
- `user="login"`
- `password="password"/>`

## Sélection

- `<sql:query var="Clients" dataSource="${Db}">`
- `SELECT * FROM Clients </sql:query>`
- `<sql:query var="Clients" dataSource="${Db}" sql="SELECT * FROM Clients" />`



# JSTL : Librairie SQL

**<!-- Affichage d'une requête `${Clients}` étant la variable de scope créée par `<sql:query/>` -->**

- `<table>`
- **`<!-- Affichage de l'entête avec le nom des colonnes -->`**
- `<tr>`
- `<c:forEach var="name" items="${Clients.columnNames}">`
- `<th>${name}</th>`
- `</c:forEach>`
- `</tr>`
- **`<!-- Affichage des données -->`**
- `<c:forEach var="ligne" items="${Clients.rows}">`
- `<tr>`
- `<c:forEach var="valeur" items="${ligne}">`
- `<td>${valeur}</td>`
- `</c:forEach>`
- `</tr>`
- `</c:forEach>`
- `</table>`



# JSTL : Librairie SQL

**<!-- Exemple de requête paramétrée avec deux dates : -->**

- **<sql:query var="result" dataSource="\${Db}">**
- **SELECT \* FROM Commande**
- **WHERE date BETWEEN ? AND ?**
- **<sql:dateParam value="\${startDate}" type="date"/>**
- **<sql:dateParam value="\${endDate}" type="date"/>**
- **</sql:query>**

**<!-- Exemple de requête Update paramétrée : -->**

- **<sql:update var="updateCount" dataSource="\${Db}">**
- **UPDATE Client SET Nom=?**
- **<sql:param value="Mohamed Lahmer" WHERE Code=1/> </sql:update>**



# Java Server Face : JSF

Java Server Faces est un framework de développement d'applications Web en Java Java Server Faces permet

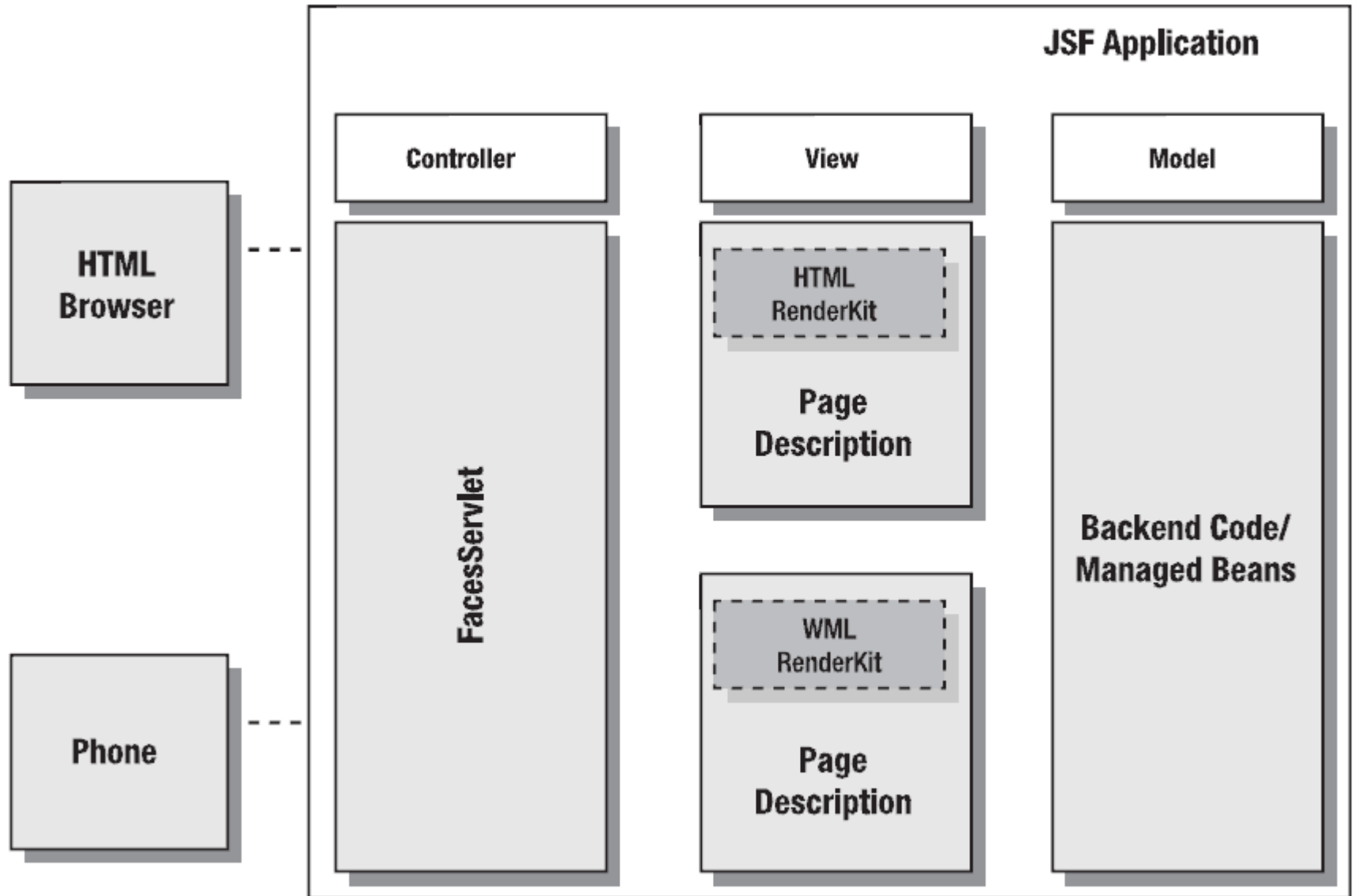
- une séparation de la couche présentation des autres couches (MVC)
- un mapping entre l'HTML et l'objet
- un ensemble de composants riches et réutilisables
- une liaison simple entre les actions côté client de l'utilisateur (eventListener) et le code Java côté serveur

Création de nouveaux composants graphiques

JSF peut être utilisé pour générer autre chose que du HTML (XUL, XML, WML, ...)



# Java Server Face MVC2





# Java Server Face : JSF

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 : [javaserverfaces](#)
- Apache : [myfaces](#)

Apache fournit des fonctionnalités additionnels via le sous

- projet : Tomahawk
- Composants graphiques
- Validators plus fournis

primefaces : <https://www.primefaces.org>

IceFaces : <http://www.icefaces.org>

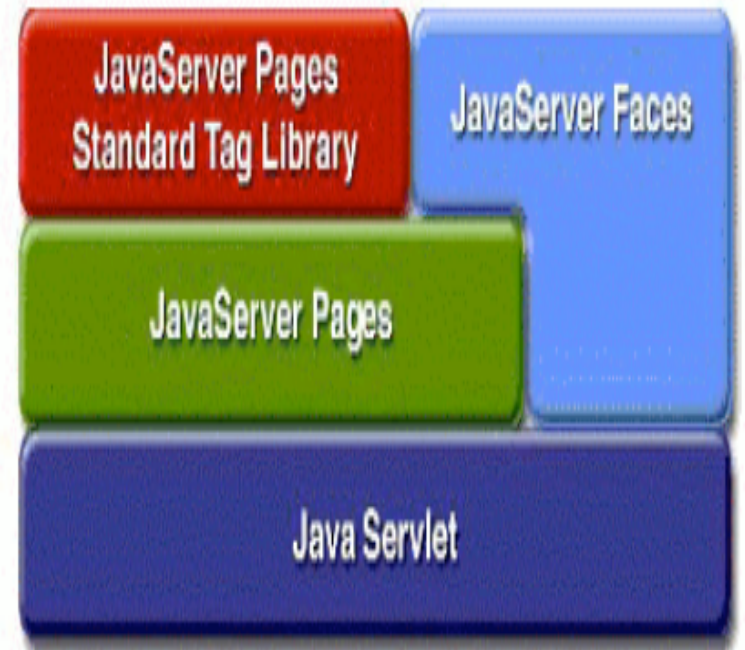
# Java Server Face : principe de fonctionnement

JSF s'appuie sur les technologies précédentes

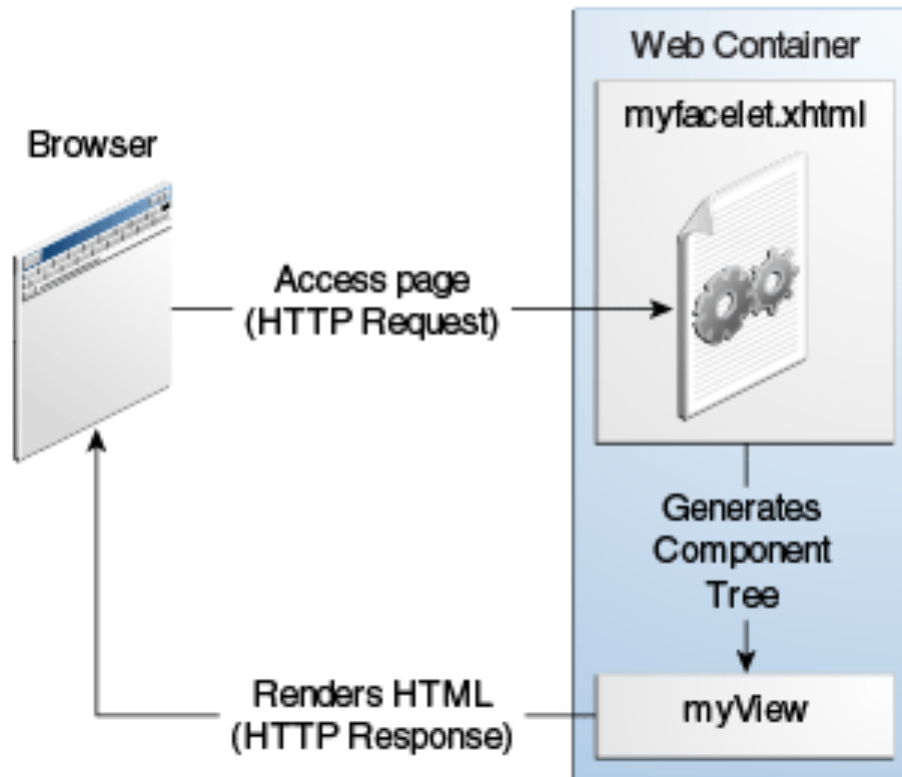
- Génération en Servlet
- Utilisation des composants JSF dans les pages JSP
- Les composants JSF sont exposés aux JSPs grâce aux balises personnalisés

L'interface utilisateur construite dans la page JSP est générée à l'aide de la technologie JSF

Elle fonctionne sur le serveur et le rendu est retourné au client



# Java Server Face : principe de fonctionnement

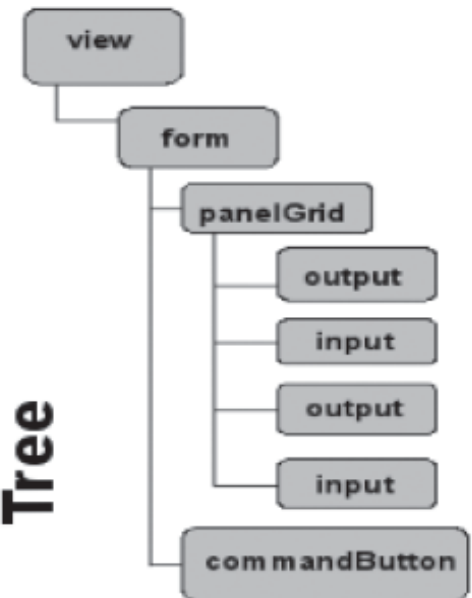


# Java Server Face : principe de fonctionnement

## 1 Development

```
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="#{msg.emailLabel}"/>
    <h:inputText value="#{login.email}"/>
    <h:outputText value="#{msg.passwordLabel}"/>
    <h:inputText value="#{login.password}"/>
  </h:panelGrid>
  <h:commandButton action="#{login.check}"
    value="#{msg.btnLabel}"/>
</h:form>
```

## 2 JSF Component Tree



## 3 Output

HTML	Browser
<pre>&lt;td&gt;Email:&lt;/td&gt; &lt;td&gt;&lt;input type="text" name="j_id2:j_id5" value="" /&gt;&lt;/td&gt; &lt;/tr&gt; &lt;td&gt;Password:&lt;/td&gt; &lt;td&gt;&lt;input type="text" name="j_id2:j_id7" value="" /&gt;&lt;/td&gt; &lt;/tr&gt; &lt;/tbody&gt; &lt;/table&gt; &lt;input type="submit" name="j_id2:j_id8" value="Sign In" /&gt; &lt;/form&gt;</pre>	<p>Email: <input type="text"/></p> <p>Password: <input type="text"/></p> <p><input type="button" value="Sign In"/></p>



# Java Server Face : principe de fonctionnement

Construire le formulaire dans **une page JSP** en utilisant les **balises JSF**

Développer **un Bean** qui effectue un « **Mapping** » avec les valeurs du formulaire

- Modifier le **formulaire** pour spécifier **l'action** et **l'associer au Bean**

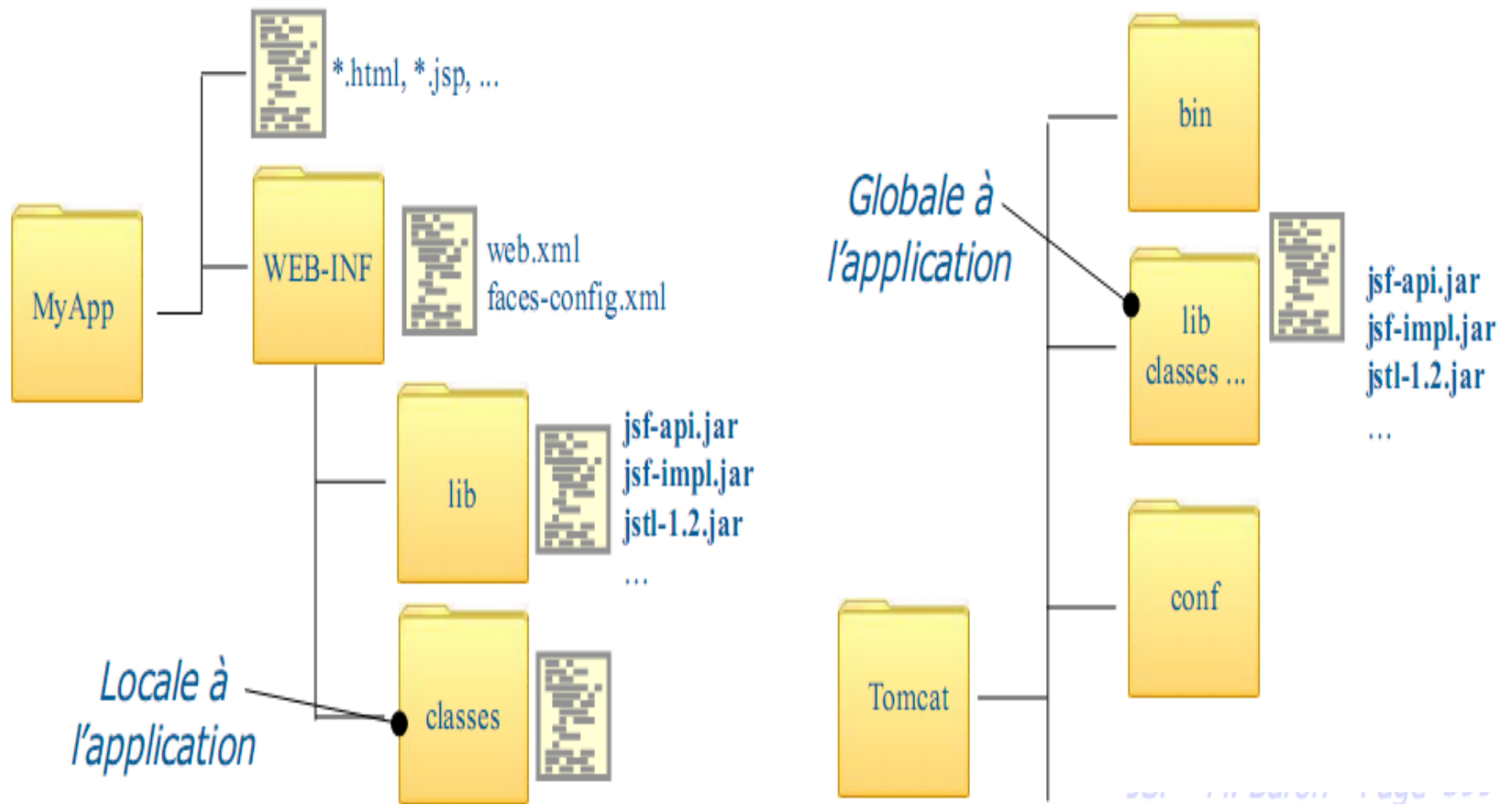
Fournir des **Converters** et des **Validators** pour traiter les données du formulaire

Nécessite la configuration du fichier web.xml de façon à ce que JSF soit pris en compte

Paramétrer le fichier **faces-config.xml** pour **déclarer le Bean** et les **règles de navigation**

- Créer les pages JSP correspondant à chaque condition de **retour**

# Configuration JSF Tomcat





# JSF : web.xml

```
<web-app>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>No-JSF-JSP-Access</web-resource-name>
    <url-pattern>/welcome.jsp</url-pattern>
  </web-resource-collection>
</security-constraint>
<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>*.jsf</url-pattern>
</servlet-mapping>
....
</web-app>
```





# JSF : 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 `<validator>` et des Converters `<converter>`



# JSF : faces-config.xml

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//
EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
  <navigation-rule>
    <from-view-id>/page.jsp</from-view-id>
    <navigation-case>
      <from-outcome> Cas-1 </from-outcome>
      <to-view-id>/page1.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  <managed-bean>
    <managed-bean-name>Nom Bean</managed-bean-name>
    <managed-bean-class>Classe du Bean</managed-bean-class>
    <managed-bean-scope>portée</managed-bean-scope>
  </managed-bean>
</faces-config>
```



# Exemple de page JSF

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<f:view>
    <h1>
        <h:outputText value="Hello World avec JSF" />
    </h1><br>
</f:view>
</body>
</html>
```



# Exemple de page jfs avec mapping

```
package jsf;
public class LoginBean {
    private String nom;
    private String mdp;
    public String getMdp() {
        return mdp;
    }
    public String getNom() {
        return nom;
    }
    public void setMdp(String
string) {
        mdp = string;
    }
    public void setNom(String
string) {
        nom = string;
    } }
}
```

```
</faces-config>
<managed-bean>
<managed-bean-name>
login
</managed-bean-name>
<managed-bean-class>
jsf.LoginBean
</managed-bean-class>
<managed-bean-scope>
session
</managed-bean-scope>
</managed-bean>
```




# Exemple de page jfs avec mapping

```
<f:view>
  <h:form>

    <h:panelGrid columns="2">
      <h:outputText value="Nom :"/>
      <h:inputText value="#{login.nom}" />
      <h:outputText value="Mot de passe :"/>
      <h:inputSecret value="#{login.mdp}"/>
      <h:commandButton value="Login"
        action="Enregistrer"/>
    </h:panelGrid>
  </h:form>
</f:view>
```

Navigation





# JSF : Navigation

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 (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.

Pour chaque valeur de message une page vue de direction est indiquée dans la balise `<navigation-case>`

`<from-action>`: l'action dans le managedBean

`<from-outcome>`: la valeur du message

`<to-view-id>` : la vue de direction

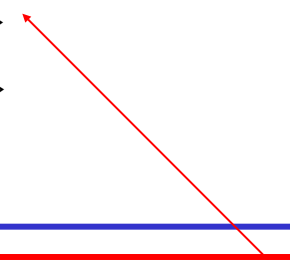


# JSF : Navigation

```
<faces-config>
  <navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-action>#{login.verify}</from-action>
      <from-outcome>Enregistrer</from-outcome>
      <to-view-id>/accueil.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Navigation dynamique conditionnée  
par le résultat d'une méthode du  
Bean

```
public String verify(){
    if(mdp.isEmpty() ||
       nom.isEmpty())
        return "Annuler";
    else return "Enregistrer";
}
```



```
<navigation-case>
    <from-outcome>Annuler</from-outcome>
    <to-view-id>/Rejet.jsp</to-view-id>
</navigation-case>
```



# JSF : TagLib HTML

Tag	Rôle
form	le tag <form> HTML
commandButton	un bouton
commandLink	un lien qui agit comme un bouton
graphicImage	une image
inputHidden	une valeur non affichée
inputSecret	une zone de saisie de texte mono ligne dont la valeur est non lisible
inputText	une zone de saisie de texte mono ligne
inputTextarea	une zone de saisie de texte multi-lignes
outputLink	un lien
outputFormat	du texte affiché avec des valeurs fournies en paramètre
outputText	du texte affiché
panelGrid	un tableau
panelGroup	un panneau permettant de regrouper plusieurs composants





## JSF : TagLib HTML

selectBooleanCheckbox	une case à cocher
selectManyCheckbox	un ensemble de cases à cocher
selectManyListbox	une liste déroulante où plusieurs éléments sont sélectionnables
selectManyMenu	un menu où plusieurs éléments sont sélectionnables
selectOneListbox	une liste déroulante où un seul élément est sélectionnable
selectOneMenu	un menu où un seul élément est sélectionnable
selectOneRadio	un ensemble de boutons radio
dataTable	une grille proposant des fonctionnalités avancées
column	une colonne d'une grille
message	le message d'erreur lié à un composant
messages	les messages d'erreur liés à tous les composants



# Vue dataTable

```
<h:dataTable id="dt1" value="#{tableBean.liste}" var="item" >
  <f:facet name="header">
    <h:outputText value="Exemple de dataTable" />
  </f:facet>
  <h:column>
    <f:facet name="header">
      <h:outputText value="CIN" />
    </f:facet>
    <h:outputText value="#{item.cin}" />
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Nom" />
    </f:facet>
    <h:outputText value="#{item.nom}" />
  </h:column>
</h:dataTable><br> </center></body></html></f:view>
```



# Validator prédéfini

Tag	Function
f:validateDoubleRange	Attributs minimum et maximum
f:validateLongRange	Attributs minimum et maximum
f:validateLength	Attributs minimum et maximum
f:validateRegEx	Une expression régulière
f:validator	Un validateur personnalisé l'attribut binding pointe vers <b>une classe</b> qui implémente Validator
f:validateBean	Un validateur personnalisé l'attribut binding pointe vers <b>une instance</b> de BeanValidator
f:validateRequired	Ne doit pas être nul (même effet que <i>required</i> )



# Gestion des erreurs

- Dans le managedBean

```
FacesMessage facesMessage = new FacesMessage("Votre message");  
FacesContext facesContext = FacesContext.getCurrentInstance();  
facesContext.addMessage(null, facesMessage);
```

- Dans la page jsf

```
<h:messages style = "color:red;margin:8px;" />
```

- ◆ Le cas ou le message est destiné à un champs spécifique

```
facesContext.addMessage("myform:id_Input", facesMessage);  
<h:message for="id_Input" style="color:red" />
```



# Validator personnalisé

```
@FacesValidator("org.estm.EmailValidator")
public class EmailValidator implements Validator{
    private static final String EMAIL_PATTERN = "^[_A-Za-z0-9-]+(\\." +
        "[_A-Za-z0-9-]+)*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)*" +
        "(\\.[A-Za-z]{2,})$";
    private Pattern pattern;
    private Matcher matcher;
    public EmailValidator(){
        pattern = Pattern.compile(EMAIL_PATTERN);
    }
    @Override
    public void validate(FacesContext context, UIComponent component,
        Object value) throws ValidatorException {
        matcher = pattern.matcher(value.toString());
        if(!matcher.matches()){
            FacesMessage msg =
                new FacesMessage("E-mail validation failed.",
                    "Invalid E-mail format.");
            msg.setSeverity(FacesMessage.SEVERITY_ERROR);
            throw new ValidatorException(msg);
        }
    }
}
```



# Converter

**Conversion automatique si on a associé le composant à une propriété d'un managed bean avec un type standard**

**BigDecimalConverter, BooleanConverter, DateTimeConverter, DoubleConverter, EnumConverter, ...**

**<h:inputText converter="javax.faces.convert.IntegerConverter"/>**

**Pour une date (date, time, both):**

**<h:outputText value="#{user.date-naiss}">**

**<f:convertDateTime type="date" dateStyle="full" /> </h:outputText>**

**Attributs pour personnaliser le convertisseur**

**dateStyle, locale, pattern, timeStyle, timeZone, type**

**Manuelle Pour un objet: convertisseur personnalisés**



# Converter personnalis 

Deux m thodes   red finir `getAsObject()` et `getAsString()`

- `getAsObject()` converti la valeur (input) en un Object
- `GetAsString()` converti la valeur (input) en une Cha ne de caract re

@FacesConverter("org.estm.MyConverter")

```
public class MyConverter implements Converter{
```

```
@Override
```

```
public Object getAsObject(FacesContext context, UIComponent component,  
    String value)
```

```
{
```

```
@Override
```

```
    public String getAsString(FacesContext context, UIComponent component,  
        Object value)
```

```
{    //Dans .xhtml on ajoute
```

```
    }    <h:inputText value="#{Bean.age}" />
```

```
    }    <f:converter ConverterId="#{MyCovverter}" />
```

```
    </h:inputText>
```

# Ajax en JSF

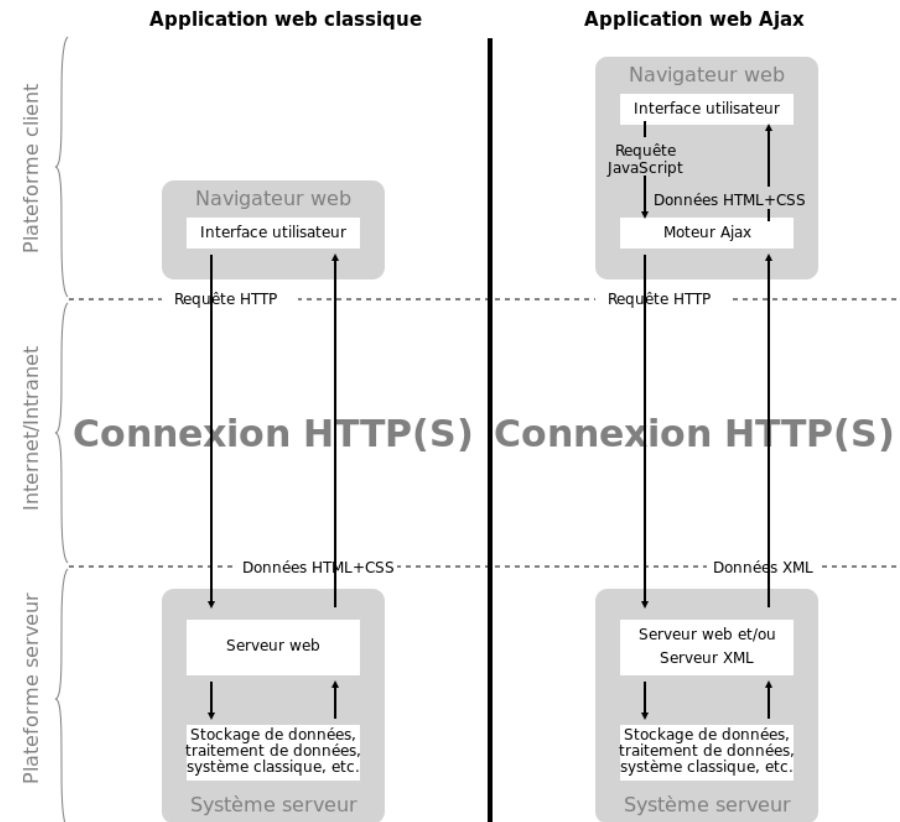
## Asynchronous JavaScript And XML (AJAX)

Ajax combine JavaScript, les CSS, JSON, XML, le DOM et le XMLHttpRequest

Ajax vise à :

- Diminuer les temps de latence ;
- Eviter le rechargement de la page ;

```
1 <h:form id="form_1">
2   <h:panelGrid id="panel_1" rows="2">
3     <!-- content irrelevante -->
4   </h:panelGrid>
5 </h:form>
6
7 <h:form id="form_2">
8   <h:commandButton id="button">
9     <f:ajax render=":form1:panel_1 panel_2" />
10  </h:commandButton>
11  <h:panelGrid id="panel_2">
12    <!-- content irrelevante -->
13  </h:panelGrid>
14 </h:form>
```







# AJAX : demo

**<f:ajax execute="input-component-id" render="output-component-id" />**

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:form>
    <h:inputText id="inputName" value="#{user.login}"></h:inputText>
    <h:commandButton value="valider">
      <f:ajax execute="inputName" render="outputMessage" />
    </h:commandButton>
    <h:outputText id="outputMessage"
      value="#{user.login!=null ? user.welcomeMessage : ' '}" />
    </h:form>
  </html>
```



# JSF : Gestion des messages

- Dans une application professionnelle, la gestion des messages (warning, info, error) est un composant essentiel dans l'interaction avec le client (Internationalisation).
- En JSF, les messages sont des ressources qu'on peut gérer par le resource bundle.
- On met tous les messages qui peuvent être générés par l'application dans un fichier **messages.properties**.
- Pour l'internationalisation, on crée un autre fichier **messages.properties\_en** (contenant la traduction) relatif à la langue utilisée.
- Le format d'une ligne du fichier peut être comme suit :
  - E0001 = "message erreur 1"**
  - I0001 = "message information 1" - {0} et {1}**



# Demo messages

faces-config.xml

```
<application>
  <resource-bundle>
    <base-name>estm.dsic.business.messages</base-name>
    <var>msg</var>
  </resource-bundle>
</application>
*.xhtml
<f:loadBundle basename="estm.dsic.business.messages" var="« msg"/>
<h:outputText value="« #{msg['E0001']}/>
<h:outputText value="« #{msg.E0001}/>
<h:outputFormat value="« #{msg['I0001']}>
  • <f:param value="param0" />
  <f:param value="param1" />
</h:outputFormat>
```



# Compléments JSF

## Logout

```
public void logout() throws IOException {  
    ExternalContext ec =  
        FacesContext.getCurrentInstance().getExternalContext();  
    //HttpSession session = (HttpSession) ec.getSession(false);  
    //session.invalidate();  
    ec.invalidateSession();  
    ec.redirect(ec.getRequestContextPath() + "/home.xhtml");  
}
```

## Annotation supplémentaires

@ManagedBean

@SessionScoped, @ViewScoped, @RequestScoped, ApplicationScoped



# Enterprise JavaBeans

**Enterprise Java Beans :**

- Composants logiciels serveur distribué

**Objectif**

- Standardiser le développement et le déploiement de composants serveurs écrits en Java
- Le développeur ne prend en compte que la logique métier de l'EJB. Le reste est prise en charge par le conteneur.

**Les spécifications EJB**

- Stateless Session Bean et Statefull Session Bean
- Entity Bean
- Message Driven Bean

**L'historique EJB 1.1, 2.0, 2.1, 3.0 et 3.1**

- Possibilité de publier l'ejb en tant que web service(JAX-WS)
- L'utilisation du JPA (Java Persistence API) pour l'ejb type entity



# Services du conteneur d'EJB

## Services internes

- Gestion de la charge du serveur (cycle de vie, accès client, passivation...)
- Service de nommage
- Gestion des accès aux objets métiers

## Services externes

- Gestion du mapping sur BD relationnelle
- Gestion des transactions
- Gestion des échanges de messages
- Des API sur les services (JDBC/JTS/JMS)

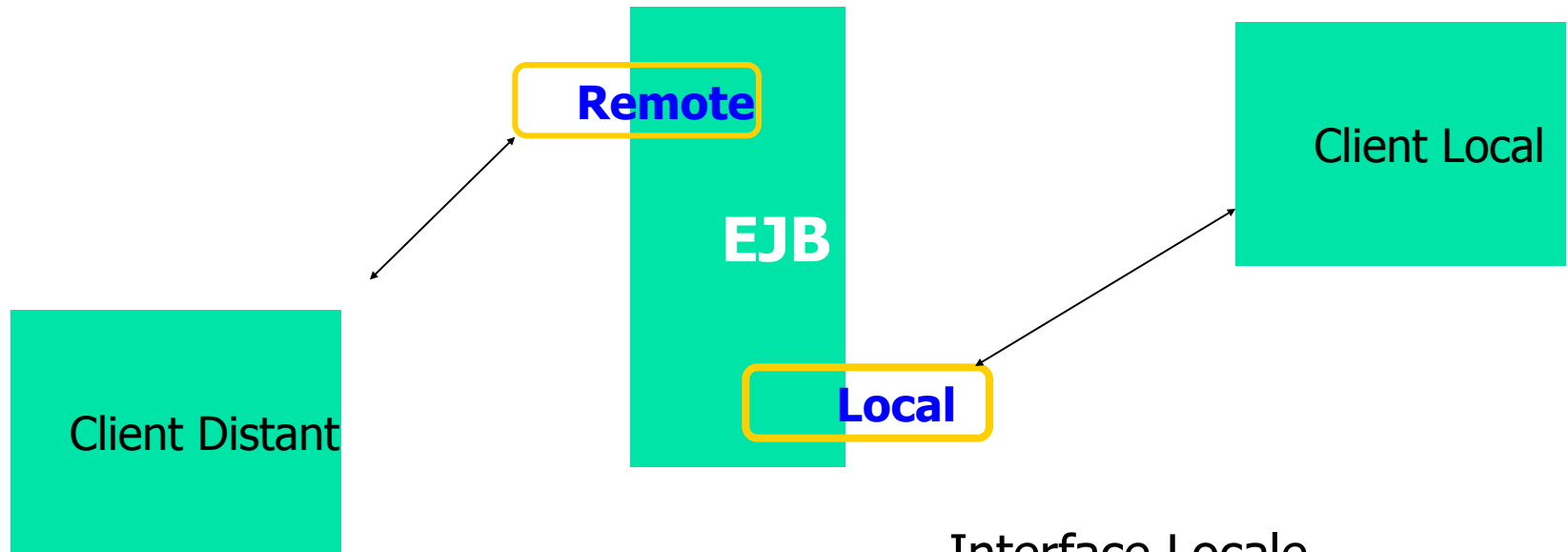


# Types EJB

- **Session** : l'invocation est faite par l'appelle de méthode
  - **stateless** : ne garde pas l'état
  - **stateful** : serialisable avec état
  - **singleton** : une seule instance qui garde un état partagé par tous les clients. l'accès concurrent est géré soit par le conteneur (CMC) ou l'ejb lui même (BMC)
- **Message driver** : l'invocation est faite par l'envoi de message (JMS) appropriés à des traitements orientés événements. ne dispose pas de vue client (Pas d'interface)
- **Entity** : sont persistants et directement lié à une base de données (SQL/NoSQL) via un mapping (JPA)

limité à un seul client

# EJB



Interface distante

Les services offerts par le bean  
à ses clients

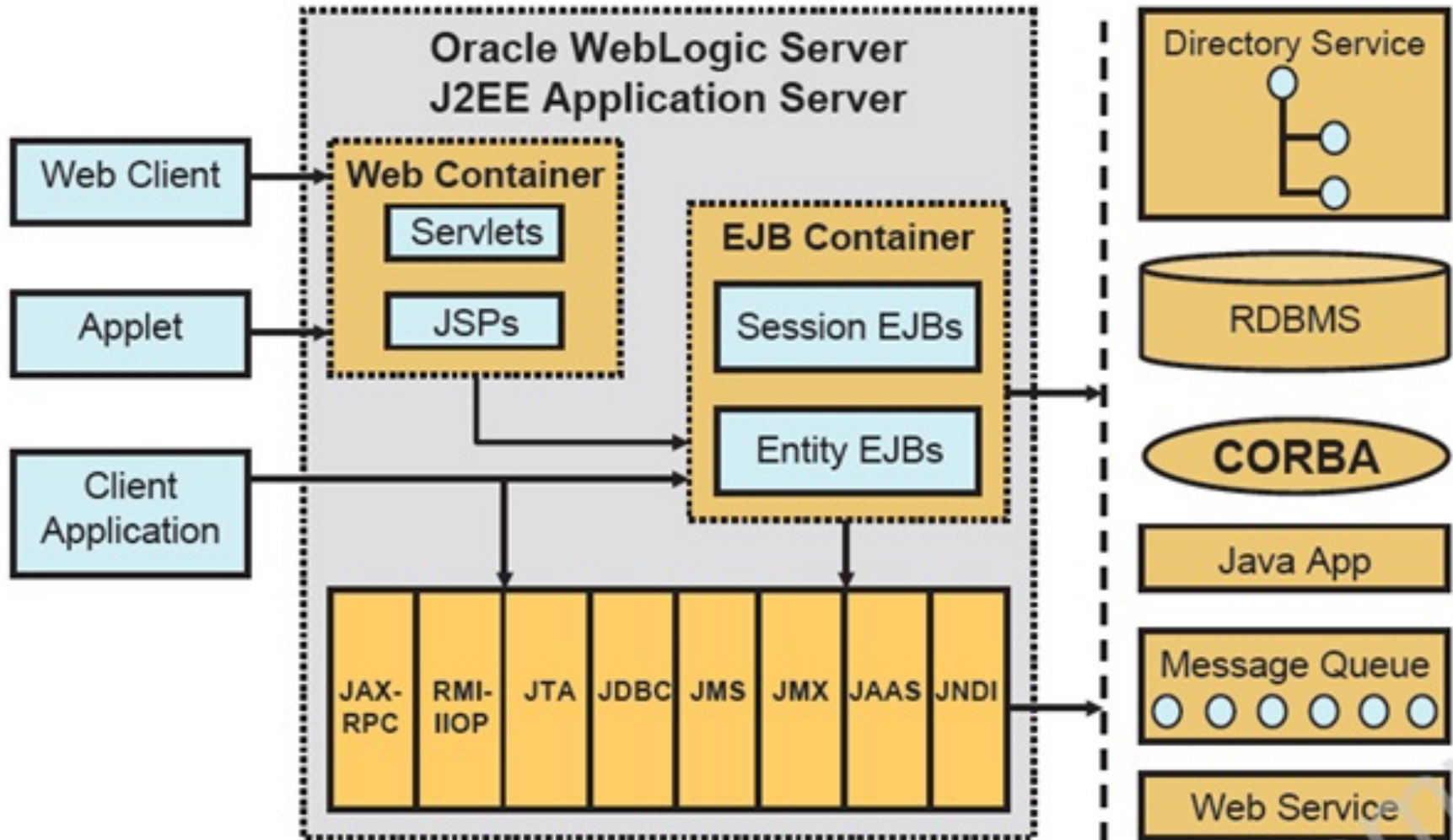
Navigateur, classe Main, Activity

Interface Locale

Les mêmes (ou d'autres  
services) que l'interface distante  
page jsp, servlet, EJB



# Architecture EJB





# EJB : Interface Remote

## Session Bean

Une interface (éventuellement 2 : Local + Remote) + 1 classe

## Interfaces

- Langage annotations intégré depuis jdk1.5
- @javax.ejb.Local ou @javax.ejb.Remote

```
import javax.ejb.Remote;
```

```
@Remote
```

```
public interface Salam {
```

```
    Public String getSalam(String Lang);}
```

```
import javax.ejb.Stateless;
```

```
@Stateless(mappedName= "salam")
```

```
public class SalamBean implements Salam {
```

```
    public String getSalam(String Lang) {
```

```
        If(Lang.equals("Ar")) return "Salam "
```

```
        If(Lang.equals("Fr")) return "Bonjour "}}}
```



# Appel EJB depuis un client

```
public class Client {  
    public static void main(String args[]) throws Exception  
    {  
        Properties props = new Properties();  
        props.setProperty("org.omg.CORBA.ORBInitialHost", "localhost");  
        props.setProperty("org.omg.CORBA.ORBInitialPort", "3700");  
        Context ic = new InitialContext(props);  
        Salam bean = (Salam) ic.lookup("salam");  
        System.out.println(bean.getSalam("Fr")); } }
```

//CAS JBOSS

```
props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.Naming  
ContextFactory");  
props.put(Context.PROVIDER_URL, "localhost:1099");
```

//CAS TOMEE

```
props.put(java.naming.factory.initial, "org.apache.openejb.client.Rem  
oteInitialContextFactory");  
props.put(Context.PROVIDER_URL, « localhost:4201");
```



# Appel d'un EJB

## Client Local

- typiquement une servlet ou une JSP
- co-localisée sur le même serveur que le bean
- attribut du type de l'interface
- annoté @EJB éventuellement @EJB(name=« salam »)

```
public class ClientServlet extends HttpServlet {  
    @EJB(name="salam")  
    private SalamBean sBean;  
    public void service( HttpServletRequest req,  
        HttpServletResponse resp ) {  
        resp.setContentType("text/html");  
        PrintWriter out = resp.getWriter();  
        out.println("<html><body>" + sBean.getSalam("Ar")  
            + "</body></html>");  
    }  
}
```

# **Java Persistence API**

## **« JPA »**

# JPA Les principes

- ❖ JEE propose une couche d'abstraction par dessus JDBC pour la persistance des données appelée JPA
- ❖ JPA précise les modalités de fonctionnement d'un ORM (Object Relationnel Mapping) Java à base des annotations
- ❖ Plusieurs implémentations JPA :
  - ❖ Eclipse Link
  - ❖ OpenJpa
  - ❖ Hibernate
- ❖ Les entités dans les spécifications de l'API Java Persistence permettent d'encapsuler les données d'une occurrence d'une ou plusieurs tables.
- ❖ Généralement, elle représente une table dans une base de données relationnelle, et chaque instance de l'entité correspond à une ligne dans cette table.

# Exemple entité JPA

```
import java.io.Serializable;
```

```
import javax.persistence.*
```

```
@Entity  
public class Mesure implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    @Column()  
    private double cm;  
    @Column()  
    private double l;  
    //getters  
    //setters
```

# Les Relations JPA

4 types de relations à définir entre les entités de la JPA:

- One to One
- Many to One
- One to Many
- Many to Many

Chacune de ces relations peut être unidirectionnelle ou bidirectionnelle sauf one-to-many et many-to-one qui sont par définition mutuellement bidirectionnelles

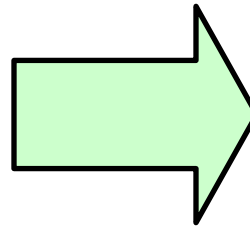


# Relation: Many to One

```
@Entity
@Table(name="EMP")
public class Employee {
    @Id
    private int id;

    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department d;

    // getters & setters
    ...
}
```



```
@Entity
public class Department
{
    @Id
    private int id;

    private String dname;

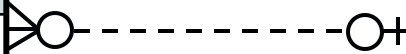
    // getters & setters
    ...
}
```

**EMP**

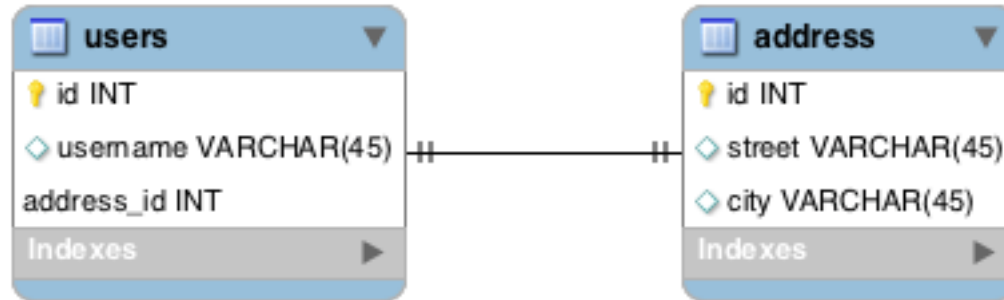
ID	DEPT_ID		
PK	FK		

**DEPARTMENT**

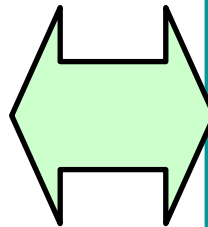
ID	DNAME		
PK			



# Relation: One to One



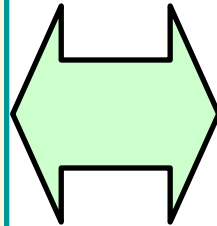
```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy =
 GenerationType.AUTO)
    @Column(name = "id")
    private Long id;
    @OneToOne(cascade =
 CascadeType.ALL)
    @JoinColumn(name = "address_id",
 referencedColumnName = "id")
    private Address address;
    // ... getters and setters
}
```



```
@Entity
@Table(name = "address")
public class Address {
    @Id
    @GeneratedValue(strategy =
 GenerationType.AUTO)
    @Column(name = "id")
    private Long id;
    @OneToOne(mappedBy = "address")
    private User user;
    //... getters and setters
}
```

# Relation: Many to Many

```
@Entity
@Table(name="EMP")
public class Employee {
    @Id
    private int id;
    @JoinTable(name="EMP_PROJ",
        joinColumns=
            @JoinColumn(name="EMP_ID"),
        inverseJoinColumns=
            @JoinColumn(name="PROJ_ID"))
    @ManyToMany
    private Collection<Project> p;
}
```

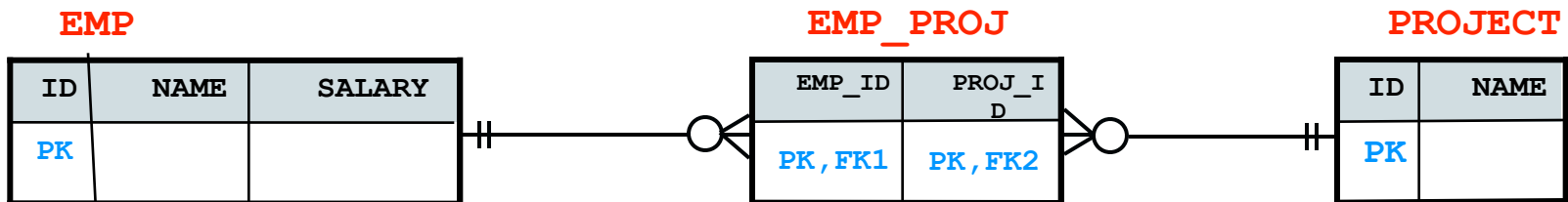


```
@Entity
public class Project {

    @Id
    private int id;

    private String name;

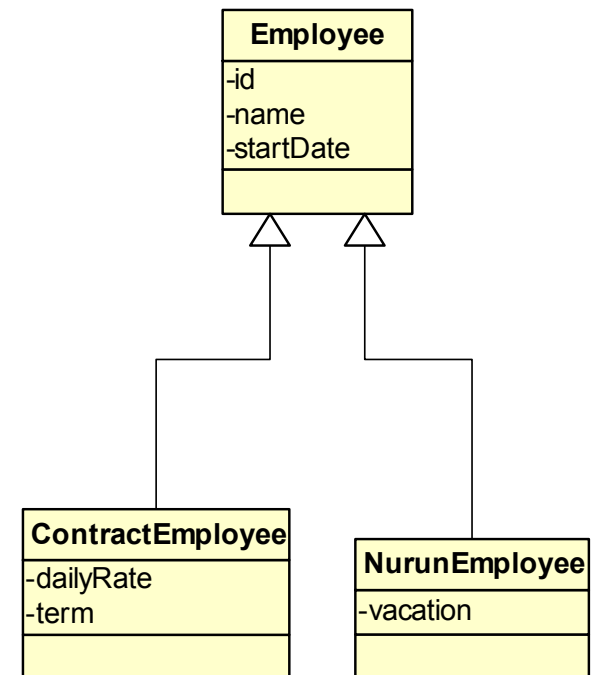
    @ManyToMany(mappedBy="p")
    private Collection<Employee> e;
    // getters & setters
    ...
}
```



# Héritage : single table

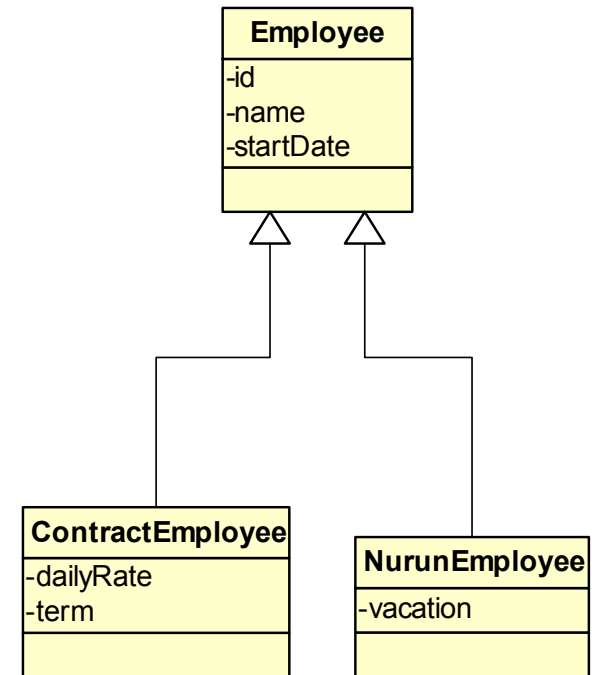
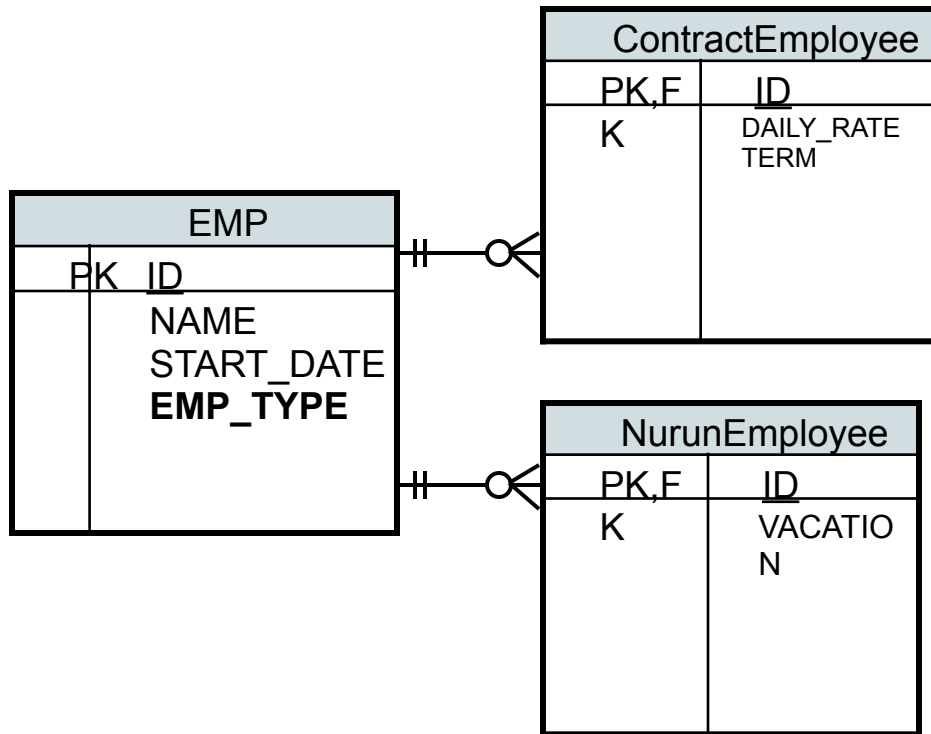
Cette stratégie aplatit la hiérarchie de classes dans une seule table contenant tous les attributs de la hiérarchie. Les spécialisations sont différenciées par un discriminateur.

EMP	
PK <u>ID</u>	
	NAME START_DATE DAILY_RATE TERM VACATION EMP_TYPE



# Héritage: joined table

Cette stratégie imite la hiérarchie de classes dans plusieurs table reliées. Ici aussi un discriminateur est nécessaire.



# Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/
persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://
xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
  <persistence-unit name="ds" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/ds</jta-data-source>
    <class>estm.jee.dsic.entities.User</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://
localhost:3306/GProduits"/>
      <property name="javax.persistence.jdbc.user" value="estm"/>
      <property name="javax.persistence.jdbc.password" value="1234"/>
      <property name="javax.persistence.schema-generation.database.acti
value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```

# Entity Manager: le cœur de la JPA

Les entités, une fois annotées, ne peuvent se persister de par elles-mêmes.

Elles ont besoins d'un engin, qui lui, performera les opérations sur la base de données, en concordance avec les mappings définis dans les annotations.

C'est le rôle de **Entity Manager** qui permet de lire et rechercher des données ainsi les mettre à jour (ajout, modification, suppression).

L'EntityManager est donc au cœur de toutes les actions de persistance.

- Étape 1: mettre la main sur une instance **EntityManager**
- Étape 2: persister nos POJO

# Entity Manager: petit exemple

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory( « UnitName » );
EntityManager em = emf.createEntityManager();
//Insert
user u = new user();
em.persist(u);

//Search, update and remove
Employee u = em.find(user.class, 1);
u.setName('new Name');
em.remove(u);
//Select

Query q = em.createQuery("SELECT u FROM user u");
List<user> userList = q.getResultList();
```



# Queries: NamedQueries

On peut sauvegarder des gabarits de requête dans nos entités.

C'est ce qu'on appelle une *NamedQuery*. Ceci permet :

- ❖ La réutilisation de la requête
- ❖ D'externaliser les requête du code.

```
@Entity
@NamedQuery(name="myQuery", query="Select o from MyPojo o")
public class MyPojo { ... }

public class MyService {
    public void myMethod() {
        ...
        List results = em.createNamedQuery("myQuery").getResultList();
        ...
    }
}
```