# Java EE

**Jean-Marc Farinone**

**Maître de Conférences**
**Conservatoire National des Arts et Métiers**
**CNAM Paris (France)**

# Plan de l'exposé

- Préambule
  - Présentation Java et Java EE
  - Historique des architectures distribuées et programmation réseau en Java
- Les servlets
- Les JavaServer Pages : JSP
- Conclusion : architecture MVC

# Préambule

# Rappel ? : Pourquoi Java

- gratuit, O.O.,
- multi-plateforme car machine virtuelle. Donc ce qui est développé dans cet exposé peut être transporté sur les plateformes (toutes ?) contenant une JVM
- de nombreuses API grandement illustrées par cet exposé (et ce n'est pas tout ;-))
- + Introspection, réflexivité, exécution sécurisée, possibilité de récupérer facilement l'existant (JNI)

# Le but de Java EE

- Donner de nombreuses API (voire framework) pour le développement d'applications pour les entreprises
- Donc pour des services à v(r!)endre
- Donc développement coté serveur
- Java EE = Java Entreprise Edition anciennement J2EE = Java 2 Platform Entreprise Edition
- Depuis janvier 2006, le nom est devenu Java EE : Java Platform, Enterprise Edition
- "Java EE provides web services, component model, management, and communications APIs that make it the industry standard for implementing enterprise class service-oriented architecture (SOA) and web 2.0 applications"
- Site référence :
  `http://www.oracle.com/technetwork/java/javaee/overview/index.html`

# Java EE =

- Technologies Web : Servlet, JSP = JavaServer pages, JSP Standard Pages Library (JSTL)
- Web Services : SOAP, XML
- Technologies composants d'entreprise : EJB, JMS, Java Transaction API (JTA), JavaBeans Activation Framework (JAF), JavaMail
- Site de référence : `http://www.oracle.com/technetwork/java/javaee/tech/index.html`
- On utilise aussi :
  - JDBC + JNDI + RMI + CORBA provenant de Java SE
- Tutorial à `https://docs.oracle.com/javaee/7/tutorial/index.html`

# Introduction

# Historique des architectures distribuées

# Programmation réseau en Java

# Les architectures réparties: Du client-serveur aux N-tiers

- Années 80 :
  - un programme demandeur : le client,
  - un programme qui reçoit la requête et y répond : le serveur
- Mais, le serveur (ou le client) devient trop gros
  - voire parfois, ne remplit plus sa fonction initiale mais d'autres fonction beaucoup plus lourde
    - Exemples : serveur web qui interroge par CGI des BD (années 93 …), client trop lourd qui traite le format d'affichage des résultats
- D'où architecture 3 tiers (client, niveau intermédiaire, le système d'information de l'entreprise = SIE)
- Puis architecture N-tiers

# Rappel

- En Java on peut faire facilement de la programmation :
  - réseau (si, si !) TCP ou UDP ou multicast
  - écrire/lire des objets dans tout endroit où on peut écrire/lire


- Voir à `http://cedric.cnam.fr/~farinone/Java`,
  - cours (en pdf) programmation réseau,
  - cours (en pdf) entrées sorties pour écrire/lire des objets

# Plus précisément : programmation réseau (1/2)

- Pour un serveur TCP, on a la structure de code :

```
ServerSocket listen_socket = new ServerSocket(port);
   Socket client = listen_socket.accept();
   BufferedReader in = new BufferedReader (new
     InputStreamReader(client.getInputStream()));
   PrintStream out = new
     PrintStream(client.getOutputStream());
   // puis des instructions comme
   String uneRequete = in.readLine();
   out.println(uneReponse);
```

- Pour un client TCP, on a la structure de code :

```
Socket socket = new Socket(machineDist, port);
   BufferedReader in =
     new BufferedReader (new InputStreamReader(socket.getInputStream()));
   PrintStream out = new PrintStream(socket.getOutputStream());
   // puis des instructions comme
   out.println(uneRequete);
   String laReponse = in.readLine();
```

# Plus précisément : programmation réseau (2/2)

- En fait les serveurs sont multithreadés et on a plutôt la structure de code :

```
le serveur écoute

lorsqu'un client se connecte, une thread est créée pour
   traiter sa requête

retour de l'écoute
```

# Plus précisément : écriture/lecture d'objets (1/2)

- En Java, les données (attributs, propriétés) des objets peuvent être écrites et relues (donc par exemple sauvegardées) : pas les méthodes ou constructeurs
- On parle de sérialisation
- Si un objet contient une référence sur un autre objet, cet objet inclus est aussi sérialisé et on obtient ainsi un graphe de sérialisation d'objets
- Pour indiquer que les objets d'une classe peuvent être persistents on indique que cette classe implémente l'interface `Serializable`.
  Il y a aucune méthode à implanter dans cette interface qui joue seulement le rôle de marqueur pour dire que les objets peuvent être sérialisés

# Plus précisément : écriture/lecture d'objets (2/2)

- Ecrire un objet :

```
import java.util.*;
   import java.io.*;
   public class EcritObjetPers {
      public static void main(String args[]) {
     Date d = new Date();
     FileOutputStream f;
     ObjectOutputStream s;
     try {
               f = new FileOutputStream ("date.ser");
               s = new ObjectOutputStream(f);
               s.writeObject(d);
               s.close();
     } catch (IOException e) { }
       }
   }
```

- Lire un objet :

```
import java.util.*;
   import java.io.*;
   public class LitObjetPers {
      public static void main(String args[]) {
     Date d = null;
     FileInputStream f;
     ObjectInputStream s;
     try {
            f = new FileInputStream ("date.ser");
            s = new ObjectInputStream(f);
            d = (Date) s.readObject();
            s.close();
     } catch (Exception e) { }
      }
   }
```

# Réseau + écriture/lecture d'objets : une synthèse

- Si au lieu d'écrire dans un fichier `.ser`, on écrit dans un canal de communication entre deux machines,
- Si au lieu de lire dans un fichier `.ser`, on lit depuis ce même canal de communication,
- On va pouvoir transférer des objets sur le réseau …
- … entre deux programmes Java ...
- … tournant sur deux machines distinctes et (éventuellement très !!) distantes ...
- … ayant des systèmes d'exploitation, des architectures éventuellement (très !!) différentes (mais ayant des machines virtuelles Java)

# Exécuter des méthodes à distance

- Pour exécuter des méthodes à distance de la forme :

```
…méthode(refObjet1, refObjet2, varDeTypePrimitif, …);
```

- Il suffit de définir une bonne fois pour toute un protocole de lancement (JRMP : Java Remote Method Protocol). Par exemple, d'abord le nom de la méthode puis passer les arguments en les copiant dans le canal

# Conclusion

- On peut avoir très facilement des couches hautes réseau comme des couches de programmation
- et pas seulement des envois de messages
- Java propose une telle architecure : RMI (Remote Method Invocation) :
  - on lance le service de nommage
  - Le serveur RMI est créé et s'enregistre auprès du service de nommage de RMI
  - on lance le client
- Une démonstration ? OK ! (révision Tintinophile)
- Voir aussi `http://docs.oracle.com/javase/tutorial/rmi/TOC.html` (transport dynamique de code !)

# Mais

- Dans l'exemple précédent c'est à l'application à gérer des problèmes comme :
  - la confidentialité, l'intégrité des données, l'authentification des entités bref la sécurité
  - les accès concurrents à des données communes
  - les transactions
  - ...
- Alors que ces problèmes sont connus et bien résolus depuis longtemps (moteurs transactionnels, transferts cryptés, ...)

# La technologie des conteneurs de composants

- Construire des entités qui répondent à certaines (petites ?!) contraintes qui vont leur permettre :
  - de bien s'intégrer à un environnement d'exécution
  - de bien s'intégrer avec d'autres entités de même "type"
- environnement d'exécution =
  - ce qui permet de traiter (exécuter ?) ces entités +
  - tout ce que l'environnement peut amener de services connus (gestion des mises à jour, un contexte commun à toutes ces entités, transaction, sécurité, …)
- entités = composants (souvent des instances ou des classes ou ...)
- environnement d'exécution = conteneurs
- Exemple de conteneurs :
  - conteneur web (pour servlets, JSP)
  - conteneur EJB (EJB)

# Les servlets

# Servlet =

- Une servlet est un programme (plug-in, classe) à ajouter à un serveur (quel qu'il soit)
- Pour l'instant les serveurs acceptant des servlets sont plutôt des serveurs Web
- Contre-exemple : serveur de mail qui détruit les mails contenant des virus

# Motivation et historique

- Nécessité d'avoir des pages HTML dynamiques i.e. pages créées lors de la requête (météo, cours de la bourse, vente aux enchères, etc.)
- Technologie des scripts CGI
- Le serveur Web demande à lancer un programme par le protocole CGI
- Inconvénient : nécessite de créer un process (sauf technique propriétaire)

# Servlets

- La technique des CGI en Java
- MAIS
- Sans créer de processus + toute la puissance de Java (accès aux divers domaines de l'informatique : BD, multimédia, réseau, objets distribués, composants, etc.)
- + indépendance de la plate-forme et du serveur

# Comment ça marche ?

- Le serveur (Web) possède désormais un interpréteur Java : le conteneur de servlets
- => il n'y a pas de processus créé lors de l'exécution de code Java
- Le seul et unique processus qui a été lancé est une JVM
- Cf. les clients Web possèdent un interpréteur Java permettant de lancer des applets
- D'où le nom de servlets

# Conteneurs de servlets (et de JSP)

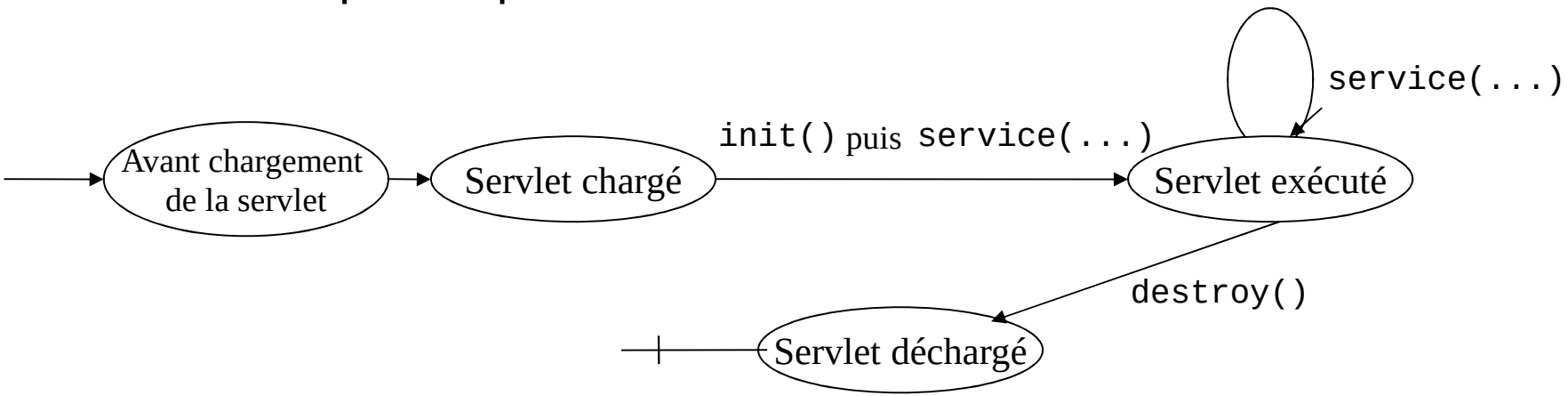
- Certains conteneurs sont des plug-in pour des serveurs Web existants
- Souvent des serveurs Web eux mêmes
- Un (bon) candidat plug-in : tomcat (`tomcat.apache.org`)
- Autres conteneurs de servlets non commerciaux : Jetty (`http://www.eclipse.org/jetty/`), ...
- Conteneurs de servlets commerciaux :
  - IBM WebSphere (`https://www-01.ibm.com/software/fr/websphere/`)
  - BEA WebLogic Server (racheté par Oracle) (`https://docs.oracle.com/cd/E13222_01/wls/docs100/`)
  - Oracle Application Server (`http://www.oracle.com/technetwork/middleware/ias/overview/index.html`)

# Tomcat

- Développé par la communauté qui implémente les spécifications des servlets et JSP
- Téléchargeable (en version d'utilisation élémentaire) gratuitement à partir de `http://tomcat.apache.org/`
- Plug-in de Apache, Microsoft IIS, …
- Est aussi un mini-serveur Web
- Dernière version "stable " : 8.0.x
- Version de développement en cours : 9.0.x

# Les états d'une servlet

- Cf. les états d'une applet. Le passage d'un état à un autre est automatique fait par le conteneur de servlets




- Chargement = chargement de la classe par le conteneur de servlets (i.e. la JVM) et instanciation = construction de l'objet servlet
- Le conteneur de servlets lance la méthode `service(…)` sur cet instance (sauf indications contraires !!) à chaque requête

# Construire une servlet pour serveur Web

- Cf. construction des applets
- On construit une classe qui hérite de la classe `javax.servlet.http.HttpServlet`
- On spécialise les méthodes qui nous intéressent
- On dépose cette classe "au bon endroit" du serveur Web

# `service(...)` dans `HttpServlet`

- `service(...)` de `HttpServlet` redirige la requête suivant son type (méthode) HTTP
- Méthode HTTP GET => `doGet(…)`
- Méthode HTTP POST => `doPost(…)`

# Une servlet : code complet

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

public class MaPremiereServlet extends HttpServlet {

     public void doGet(HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {

          // Etape 1.  Spécifier le type MIME du contenu de la réponse
          response.setContentType("text/html");

          // Etape 2.  Récupère le PrintWriter pour envoyer des données au client
          PrintWriter out = response.getWriter();

          // Step 3.  Envoyer l'information au client
          out.println("<html>");
          out.println("<head><title>Bonjour</title></head>");
          out.println("<body>");
          out.println("<h1> Bonjour à tous les étudiants de la valeur GDAPIA</h1>");
          out.println("Il est : " + new java.util.Date());
          out.println("</body></html>");
     }
     public void doPost(HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {
     doGet(request, response);
     }
}
```

# Démonstration

- On lance le serveur Web
- La servlet est rangée sous `REP_INSTAL_TOMCAT\webapps\examples\WEB-INF\classes`
- correspond à l'URL : `http://localhost:8080/examples/servlet/MaPremiereServlet`


- Euh ! Avec un peu de bidouille dans le `REP_INSTAL_TOMCAT\webapps\examples\WEB-INF\web.xml`

# Les "bidouilles" du `web.xml`

- Le fichier `web.xml` du répertoire `WEB-INF` de l'application web configure l'application web
- L'élément `<servlet>` associe un nom à une classe Java servlet
- L'élément `<servlet-mapping>` associe à ce nom l'URL relative qui permet d'atteindre cette servlet
- Une partie du `web.xml` (= `REP_TOMCAT\webapps\examples\WEB-INF\web.xml`) :

```
<servlet>
     <servlet-name>nomQuelconque</servlet-name>
     <servlet-class>MaPremiereServlet</servlet-class>
</servlet>

<servlet-mapping>
     <servlet-name>nomQuelconque</servlet-name>
     <url-pattern>/servlet/MaPremiereServlet</url-pattern>
</servlet-mapping>
```

# Quelques précisions

- Pour compiler on peut utiliser le script (qu'il faut évidemment adapter !) :

```
set TOMCAT_HOME=C:\Applications\Tomcat
set OLD_CLASSPATH=%CLASSPATH%
set CLASSPATH=%TOMCAT_HOME%\lib\servlet-api.jar;%CLASSPATH%
javac MaPremiereServlet.java
copy MaPremiereServlet.class %TOMCAT_HOME%\webapps\examples\WEB-INF\classes
set CLASSPATH=%OLD_CLASSPATH%
```

# Architecture d'une application web (1/3)

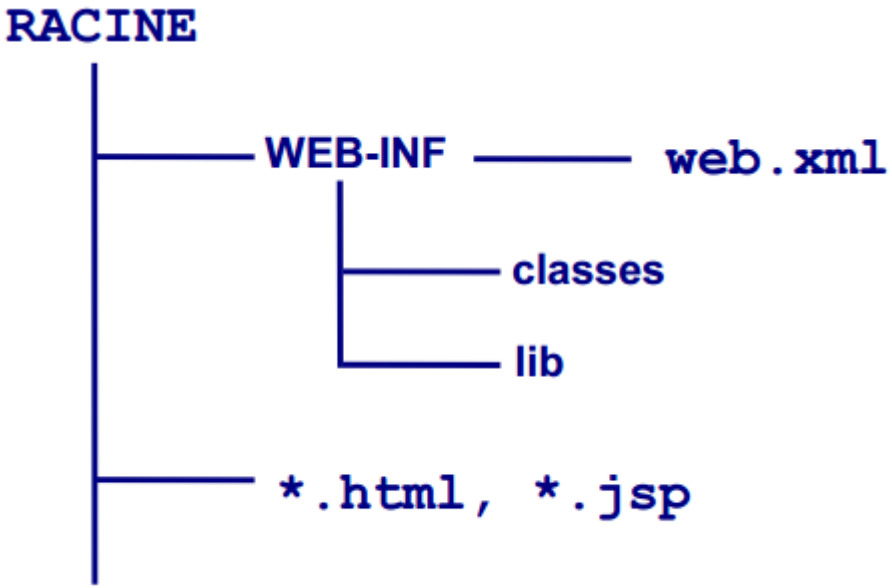
- Une bonne référence est : `http://tomcat.apache.org/tomcat-8.0-doc/appdev/deployment.html`
- Depuis la version 2.2 des spécifications des servlets, les ressources doivent être rangées comme indiqué ci dessous et diapos suivantes
- Un site web est construit dans un arbre commençant au noeud `RACINE` (eh oui !) : c'est le "document root of your application"
- Ce noeud est associé à un "context path" pour les URL
- Exemple : si le "context path" de votre application est `/catalog`, le fichier `index.html` du répertoire `RACINE` est accessible par une URL se terminant par `/catalog/index.html`

# Architecture d'une application web (2/3)

- Les fichiers `.html` et `.jsp` doivent être rangés à partir de `RACINE`
- `/WEB-INF/web.xml` : le fichier descripteur de déploiement de votre application web
- `/WEB-INF/classes/` : le répertoire racine de rangement des `.class` (servlets compilés, etc.). Si les classes sont dans des packages, la hiérarchie des packages doit être respectée à partir de `/WEB-INF/classes/`. Par exemple la classe `cnam.ihm.MaServlet` doit être mise dans `/WEB-INF/classes/cnam/ihm/MaServlet.class`
- Rappel : c'est automatiquement fait si on compile avec `javac -d ...`

# Architecture d'une application web (3/3)

- `/WEB-INF/lib` : le répertoire contenant les `.jar` nécessaires à votre application web (driver JDBC, etc.)




- On peut mettre tout cela dans un fichier compressé : un `.war`

# Déployer ailleurs que sous tomcat (1/3)

- Mettre l'application web (= le site web dynamique) sous l'arborescence de tomcat n'est pas très raisonnable
- Pourquoi ?
- Donc en général, on déploie son site dans un certain endroit (personnel) et on indique à tomcat que les ressources repérées par des URL commençant par un nom précis doivent être cherchées sous une certaine arborescence

# Déployer ailleurs que sous tomcat (2/3)

- Créer un fichier xml sous *REP_INSTALL_TOMCAT*`\conf\Catalina\localhost` appelé `MonSiteWeb.xml` et contenant :

```
<Context docBase="cheminMenantAuRepertoireDeLAppliWeb" reloadable="true"/>
```

- Et c'est tout ! Les modifications faites sous *cheminMenantAuRepertoireDeLAppliWeb* sont immédiatement prises en compte (`reloadable="true"`)
- "Set to true if you want Catalina to monitor classes in /WEB-INF/classes/ and /WEB-INF/lib for changes, and automatically reload the web application if a change is detected. This feature is very useful during application development, but it requires significant runtime overhead and is not recommended for use on deployed production applications. That's why the default setting for this attribute is false."
- source : `https://tomcat.apache.org/tomcat-8.0-doc/config/context.html`

# Déployer ailleurs que sous tomcat (3/3)

- Le fichier `nomFic.xml` étant sauvegardé sous *`REP_INSTALL_TOMCAT`*`\conf\Catalina\localhost`, l'application web sera accédée une URL commençant par `http://`*`nomMachine`*`:`*`numPort`*`/MonSiteWeb`
- Remarque : avec cette technique, il n'y a plus besoin de créer (et recréer à chaque modification) un `.war`
- C'est peut être dommage !

# Et si on ne veut pas de `web.xml` ? (1/4)

- C'est dommage !
- On utilise les annotations. Il vaut mieux utiliser les `.war` (même si on peut s'en passer merci AH !)
- Il faut écrire une servlet qui vérifie les spécifications 3.0. Par exemple :

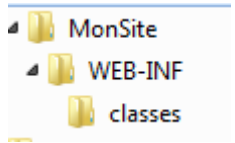
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

//@WebServlet(name="mytest",
//  urlPatterns={"/maServlet"})

// On peut aussi écrire
@WebServlet("/maServlet")
public class MaPremiereServlet extends HttpServlet {
// la suite comme la servlet précédente
```

# Et si on ne veut pas de `web.xml` ? (2/4)

- Il suffit de se préparer un site (= un répertoire) en local contenant le répertoire `WEB-INF` contenant le répertoire `classes`

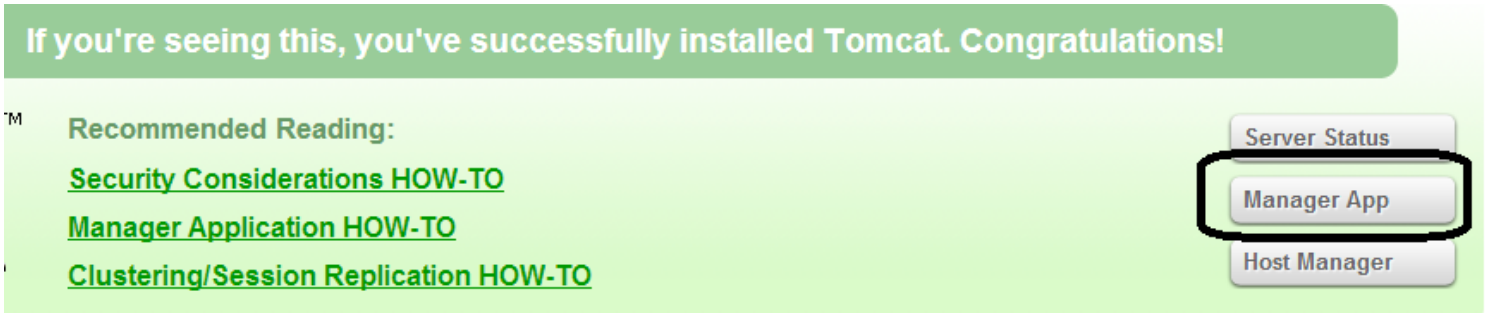



- On compile et construit un `.war`. Par exemple grâce au script (à adapter) :

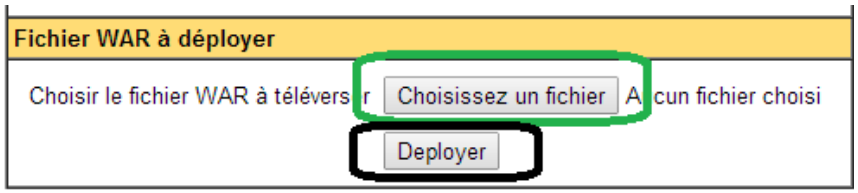
```
set TOMCAT_HOME=C:\Applications\Tomcat-7.0.34
set OLD_CLASSPATH=CLASSPATH
set CLASSPATH=%TOMCAT_HOME%\lib\servlet-api.jar;%CLASSPATH%
javac MaPremiereServlet.java
copy MaPremiereServlet.class MonSite\WEB-INF\classes
cd MonSite
jar cvf ..\monWar.war *
cd ..
set CLASSPATH=OLD_CLASSPATH
```

# Et si on ne veut pas de `web.xml` ? (3/4)

- Il faut un conteneur de servlet qui suit les spécifications servlet 3.0. Tomcat 7.0.x convient
- Il faut déployer le `.war` dans tomcat 7. Pour cela on utilise le "Gestionnaire d''applications WEB Tomcat" accessible par le bouton "Manager App" de la page d'accueil de tomcat

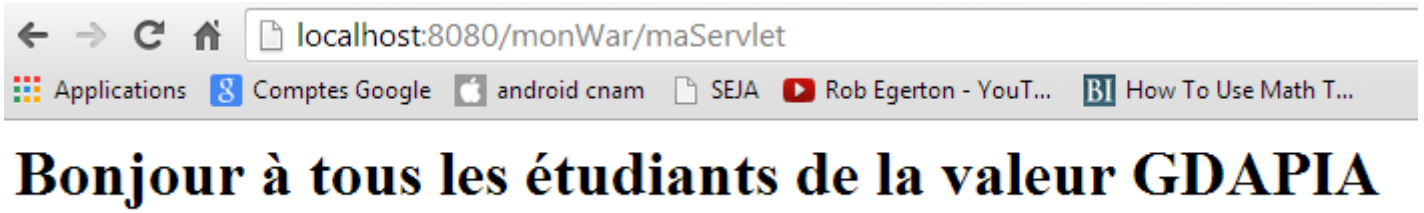



- On arrive à une nouvelle page qui permet de déployer le `.war` dans tomcat :



# Et si on ne veut pas de `web.xml` ? (4/4)

- Il suffit alors d'accéder à la servlet par l'URL `http://localhost:8080/`*`nomDuWar/urlPatternIndiqueDansLAnnotation`*
- Par exemple `http://localhost:8080/monWar/maServlet`, si on a contruit un `monWar.war`




- Compléments : l'annotation `@WebServlet` peut avoir plusieurs attributs permetant de configurer la servlet (lors de son lancement, ...). Voir à `http://blog.caucho.com/2009/10/06/servlet-30-tutorial-weblistener-webservlet-webfilter-and-webinitparam/`

# Envoi des paramètres au programme distant

- Il faut donner un nom aux composants graphiques. Ce nom est utilisé coté serveur dans notre cas (mais aussi coté client en JavaScript)
- Le serveur web reçoit une ligne contituée de `NameOfComponent=value` séparé par `&`
- On a aussi des transformations (caractère blanc, accent, ...)
- Par exemple

```
<INPUT TYPE="CHECKBOX" NAME="choice1" VALUE="choice 1 is selected">the choice 1<BR>
<INPUT TYPE="CHECKBOX" NAME="choice2">second choice<BR>
<INPUT TYPE="CHECKBOX" NAME="choice3">and the third one<BR>
<P>
Give your text : <INPUT TYPE="TEXT" NAME="wtext" VALUE="initial text"><BR>
Give your password : <INPUT TYPE="PASSWORD" NAME="wpass" VALUE="initial password"><BR>
<P>
<INPUT TYPE="SUBMIT" NAME="Send" VALUE="click here to send">
```

peut envoyer

```
choice1=choice+1+is+selected&choice2=on&wtext=my+text&
wpass=my+passwd&Send=click+here+to+send
```

# Récupérer, coté serveur, les données de la requête

- Dans une servlet, on récupère ces données par la méthode `String getParameter(String nomComposantFormulaire)` lancée sur `request`
- Plus précisément si on a, dans la page HTML :

```
Login : <input type="text" name="login" />
```

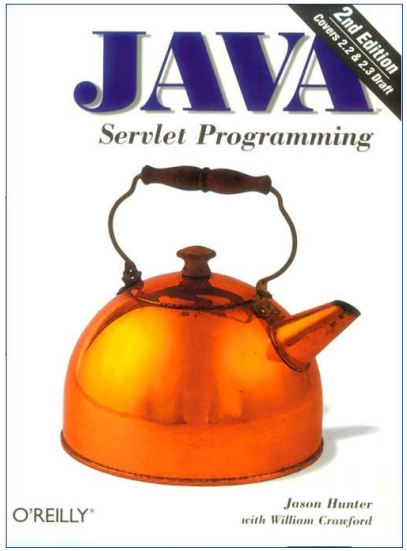
on doit avoir, dans les méthodes `doXXX( )` de la servlet (ou dans la JSP) :

**respecter la casse (majuscules et minuscules significatives !)**

```
String nomLogin = request.getParameter("login");
```

# Bibliographie

- Page de départ de la technologie servlets : `http://www.oracle.com/technetwork/java/javaee/servlet/index.html`
- Java servlets, Jason Hunter, ed O'Reilly traduit en français



# JavaServer Pages (JSP)

# ssi : la technique server side include

- Une page ssi (`shtml`) (= page HTML avec certaines portions qui sont du code à exécuter) est demandée par un client web
- Le serveur Web passe la main au programme adéquat qui traite la partie de la page le concernant
- Ce programme génère la partie dynamique
- La page HTML créée dans son ensemble est retournée au serveur puis au client Web

# JavaServer Pages

- = JSP = la technique des ssi en Java
- = une page HTML contenant du code Java
- => meilleure division des tâches :
  - présentation générale par les graphistes
  - coté dynamique par des programmeurs (Java)

# Comment ça marche ?

- Concrêtement :
  - toute la page JSP est convertie en une servlet (par le conteneur JSP)
  - cette servlet est traitée par le conteneur de servlets intégré au serveur Web (technologie des servlets) et retourne la page HTML construite

# JSP vs. Servlets

- Servlet = du code Java contenant de l'HTML
- JSP = une page HTML contenant du code Java
- Concrètement avec les JSP :
  - les parties statiques de la page HTML sont écrites en HTML
  - les parties dynamiques de la page HTML sont écrites en Java

# Notre première JSP

- fichier `MaDate.jsp`

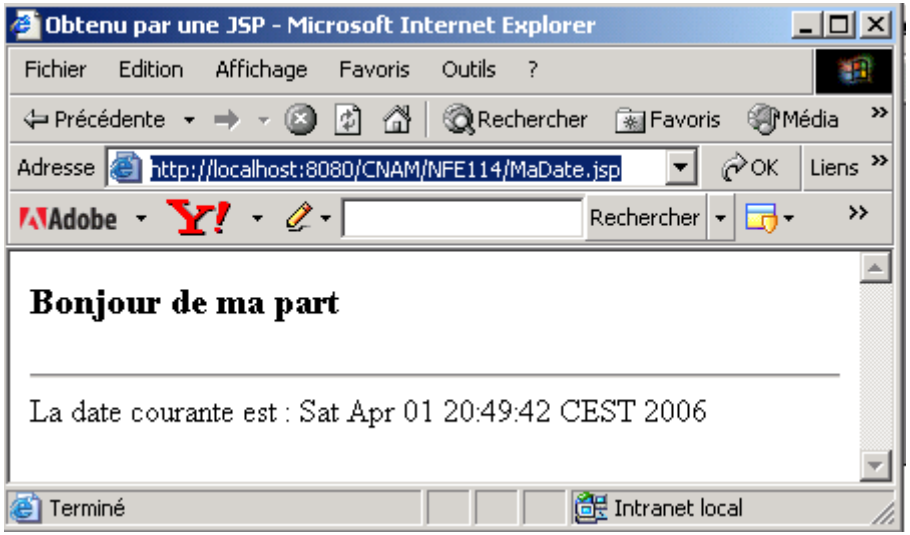
```
<html><head><title>Obtenu par une JSP</title></head>
<body>

<h3>Bonjour de ma part </h3> <hr>
La date courante est :  <%= new java.util.Date() %>
</body>
</html>
```

- Avec Tomcat 8.0 :
  - rangée à `REP_INSTAL_TOMCAT\webapps\GDAPIA`
  - Traitée quand le client demande l'URL de la JSP :
    `http://serveurWeb:<port>/GDAPIA/MaDate.jsp`

# Exécution de JSP

- Une démo:
- Connexion à `http://serveurWeb:<port>/GDAPIA/MaDate.jsp`




- Une autre exécution donne une autre date => dynamicité

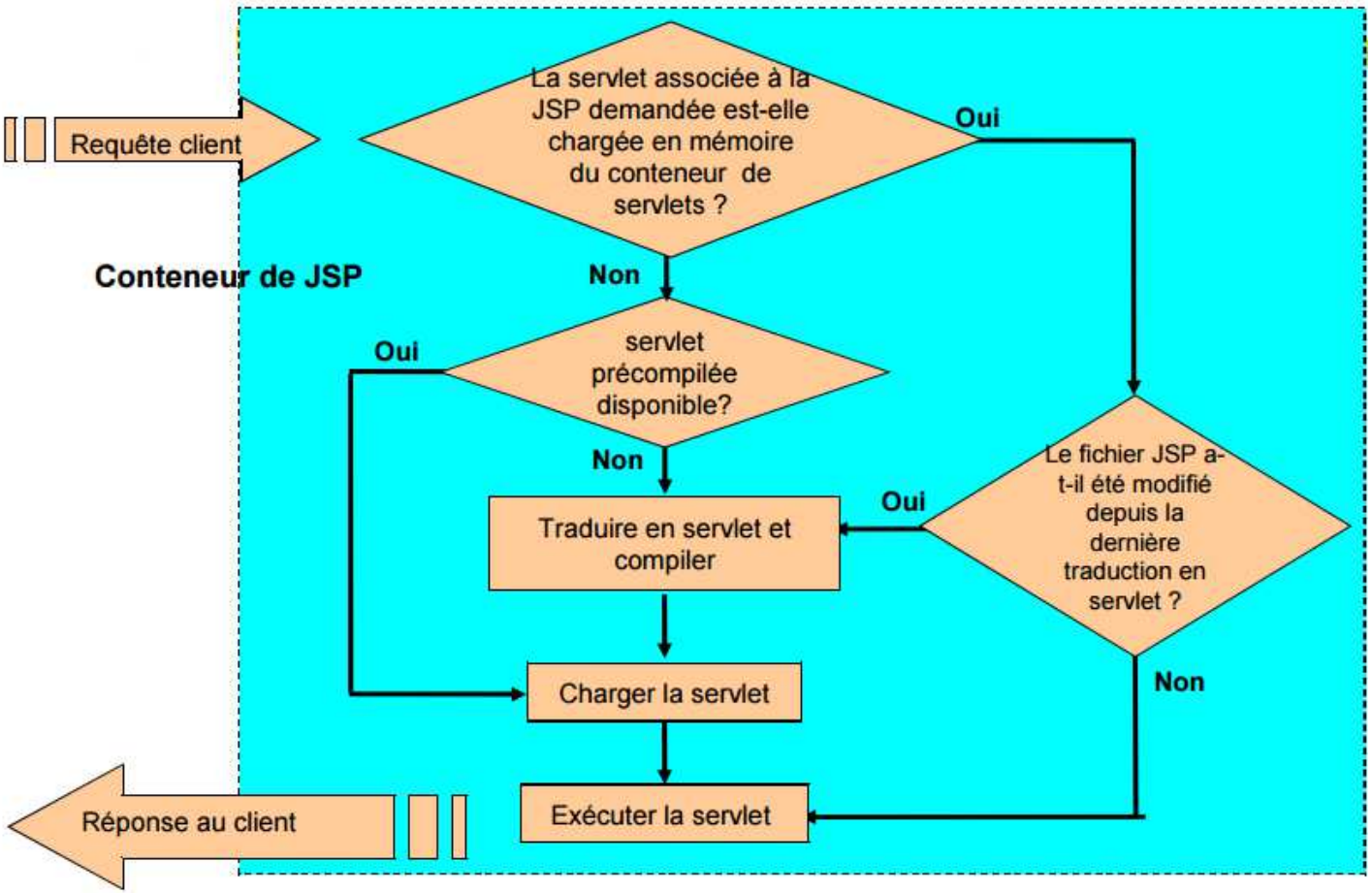
# Que s'est il passé ?

- Le conteneur de JSP a construit une servlet (`MaDate_jsp.java` sous l'arborescence `work`)
- Cette phase est parfois appelée la traduction de la JSP
- Puis a compilé et exécuté la servlet

# La servlet construite

```
package org.apache.jsp;
...
public class MaData_jsp extends HttpJspBase {
...
    public void _jspService(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {
  ...
            pageContext = _jspxFactory.getPageContext(...);
            session = pageContext.getSession();
            out = pageContext.getOut();
            // HTML
   // begin [file="C:\\...\\examples\\jsp\\MaDate.jsp";from=(0,0);to=(4,24)]
         out.write("<html><head><title>Obtenu par une JSP</title></head>\r\n
<body>\r\n\r\n<h3>Bonjour de ma part</h3> <hr>\r\n
La date courante est :  ");
     // end
//begin [file="C:\\...\\examples\\jsp\\MaDate.jsp";from=(4,27)to=(4,49)]
                out.print( new java.util.Date() );
            // end
            // HTML
// begin [file="C:\\...\\examples\\jsp\\date.jsp";from=(4,51);to=(6,7)]
                out.write("\r\n</body>\r\n</html>"); // end
            ...
    }
}
```

# Algorithme d'exécution de la JSP



# 3 parties d'une JSP

- scriptlets <% %>
- déclarations <%! %>
- expressions <%= %>

# Scriptlets <% %>

- contient du code Java
- insérer dans `_jspService()` de la servlet, donc peut utiliser `out`, `request`, `response`, etc.

```
<%
    String[] langages = {"Java", "C++", "Smalltalk", "Simula 67"};
    out.println("<h3>Principaux langages orientés objets : </h3>");
    for (int i=0; i < langages.length; i++) {
              out.println("<p>" + langages[i] + "</p>");
    }
%>
```

# Déclarations <%! %>

- Sont des déclarations Java
- Seront insérées comme des membres de la servlet
- Permet de définir des méthodes ou des données membres
- Exemples :

```
<%!
      int random4() {
            return (int)(Math.random() * 4);
      }
%>
```

```
<%!
      int nombreFetiche = 2;
%>
```

# Expressions <%= %>

- En fait expression Java qui renvoie un objet `String` ou un type primitif
- Un raccourci pour `<% out.println(…); %>`
- `<%= XXX %>` ~ `<% out.println(XXX); %>`
- attention au `;`
- est donc converti en `out.println(…)` dans la méthode `_jspService(…)` de la servlet

```
La somme est:  <%= (195 + 9 + 273) %>

Je vous réponds à l'adresse :  <%= request.getParameter("email_address") %>
```

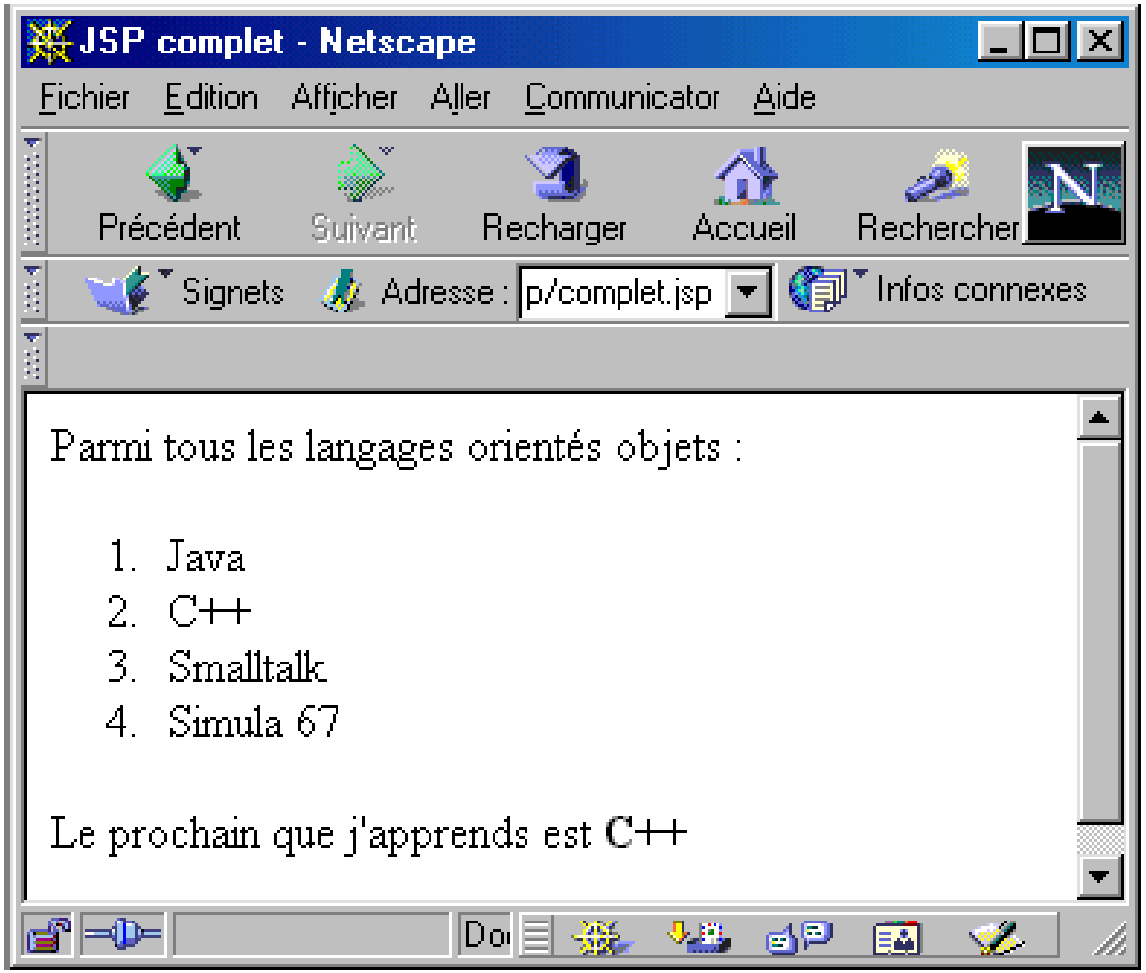
# Objets prédéfinis dans une JSP

- 3 objets peuvent être immédiatement utilisés dans une expression ou une scriptlet d'une JSP :
  - `out` : le canal de sortie
  - `request` (`HttpServletRequest`) : l'objet requête
  - `response` (`HttpServletResponse`) : l'objet réponse
- Il y en a d'autres
- Cf. ces mêmes objets dans une servlet

# Un exemple complet : complet.jsp

```
<html><head><title>JSP complet</title></head>
<body>
<%! String[] langages = {"Java", "C++", "Smalltalk", "Simula 67"};
    int random4() {
              return (int) (Math.random() * 4);
    }
%>
<p>Parmi tous les langages orientés objets :</p>
<ol>
<%
    for (int i=0; i < langages.length; i++) {
              out.println("<li>" + langages[i] + "</li>");
    }
%>
</ol>
<p>Le prochain que j'apprends est <b><%= langages[random4()] %> </b></p>
</body>
</html>
```

# complet.jsp



# Servlet, JSP : une synthèse ou un exercice ;-)

- Et la portabilité ?
- Et si le client est un macintosh ?

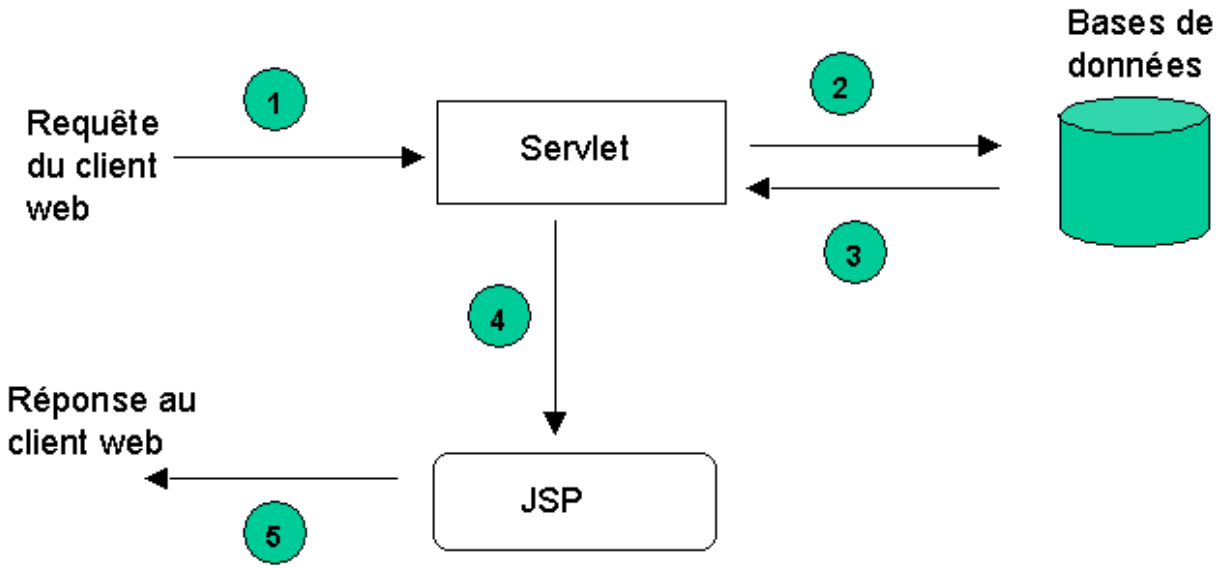
- Et pourquoi cela a marché ? Est ce que c'est parce que Java est portable ?
  - Meuh non, Pfff ! Ce n'est pas cette raison
  - Utilisation du "client universel"

# En fait, servlet vs. JSP ?

- A priori deux outils pour répondre aux mêmes besoins
- D'ailleurs on pourrait tout faire avec, pour chaque interaction une seule JSP à consulter : le modèle 1
- Ou, pour chaque interaction, consulter une page JSP construite à l'aide d'autres pages JSP (ou HTML) : le modèle 3
- En fait complémentaires : le modèle 2 (MVC)

# L'architecture MVC coté serveur : le modèle 2

- modèle = les données accédées par un code Java (JDBC, RMI, EJB, etc.)
- vues = JSP
- contrôleur = servlets



# MVC modèle 2 : un peu de syntaxe (1/2) (dans la servlet)

- Syntaxe dans la servlet pour lancer la JSP :

```
public void doPost(HttpServletRequest request, HttpServletResponse response){
   ServletContext context = getServletContext();  // héritée de GenericServlet
   RequestDispatcher dispatcher =
context.getRequestDispatcher("/maPageMiseEnForme.jsp");
   dispatcher.forward(request, response);
}
```

- La servlet peut passer des valeurs à la JSP appelé grâce à `setAttribute()`

```
public void doPost(HttpServletRequest request, HttpServletResponse response) {
     // appelle les méthodes sur les objets métiers
     ArrayList theList = // un objet à passer
     // ajoute à la requête
     request.setAttribute("nomDelObjet", theList);
    ServletContext context = getServletContext();
    RequestDispatcher dispatcher = context.getRequestDispatcher("/jspAAppeler.jsp");
    dispatcher.forward(request, response);
}
```

# MVC modèle 2 : un peu de syntaxe (2/2) (dans la servlet)

- La JSP extrait les objets de `request` grâce à `getAttribute()`

```
<%  ArrayList theList = (ArrayList)
       request.getAttribute("nomDelObjet");
// maintenant, utiliser l'ArrayList pour l'afficher correctement
// (i.e. en HTML)
%>
```

# Fin