

Architectures client/serveur

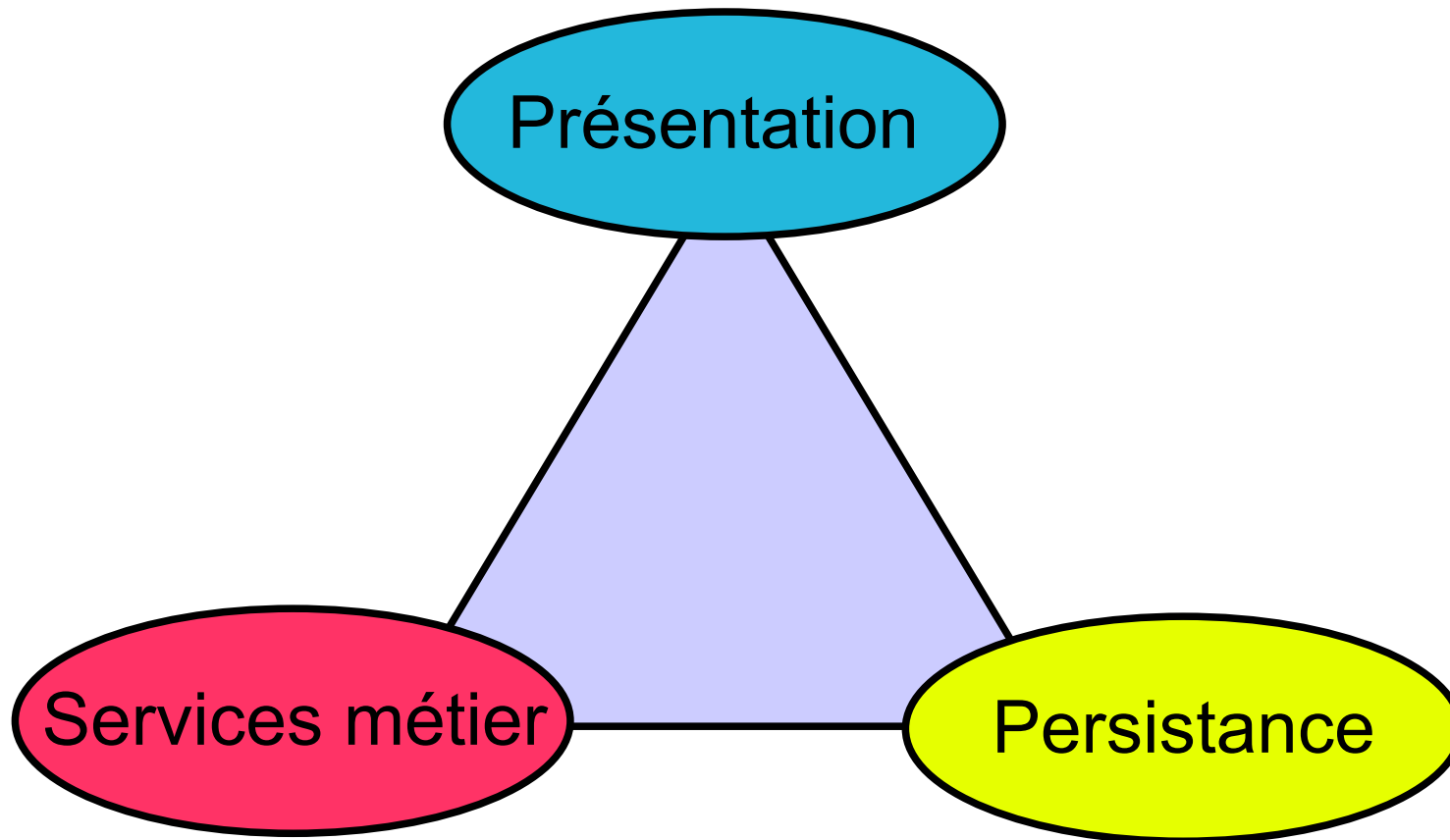
Master Technologies de l'Internet 1^{ère} année

Eric Cariou

*Université de Pau et des Pays de l'Adour
UFR Sciences Pau – Département Informatique*

Eric.Cariou@univ-pau.fr

Tryptique d'une application



Tryptique d'une application

◆ Présentation

- ◆ Interface utilisateur pour interagir avec l'application
 - ◆ Interface classique type GUI (ex : traitement de texte)
 - ◆ Interface Web, plus légère

◆ Persistance

- ◆ Enregistrement sur support physique des données de l'application
 - ◆ Fichiers (binaires, XML, ...)
 - ◆ Base de données
 - ◆ Simple
 - ◆ Avec redondance pour fiabilité
 - ◆ Multiples : fédération de bases de données
 - ◆ ...

Tryptique d'une application

- ◆ Services métier
 - ◆ Intègre la logique métier
 - ◆ Ex: un document est composé de sections, elles mêmes composées de sous-sections ...
 - ◆ Services offerts aux utilisateurs
 - ◆ Ex: créer un document, le modifier, ajouter des sections, l'enregistrer ...
 - ◆ Partie applicative
- ◆ Trois parties
 - ◆ Sont intégrées et coopèrent pour le fonctionnement de l'application
 - ◆ En anglais, on les appelle aussi des « tiers » (étages)
 - ◆ Application « 3 – tiers » quand les 3 parties sont clairement distinctes

Tryptique d'une application

- ◆ Dans un contexte distribué
 - ◆ Les tiers sont / peuvent être exécutés sur des machines différentes
 - ◆ Certains tiers peuvent être sous-découpés
 - ◆ De nombreuses variantes de placement des tiers et de leur distribution
- ◆ Modèle centralisé
 - ◆ Tout est sur la même machine

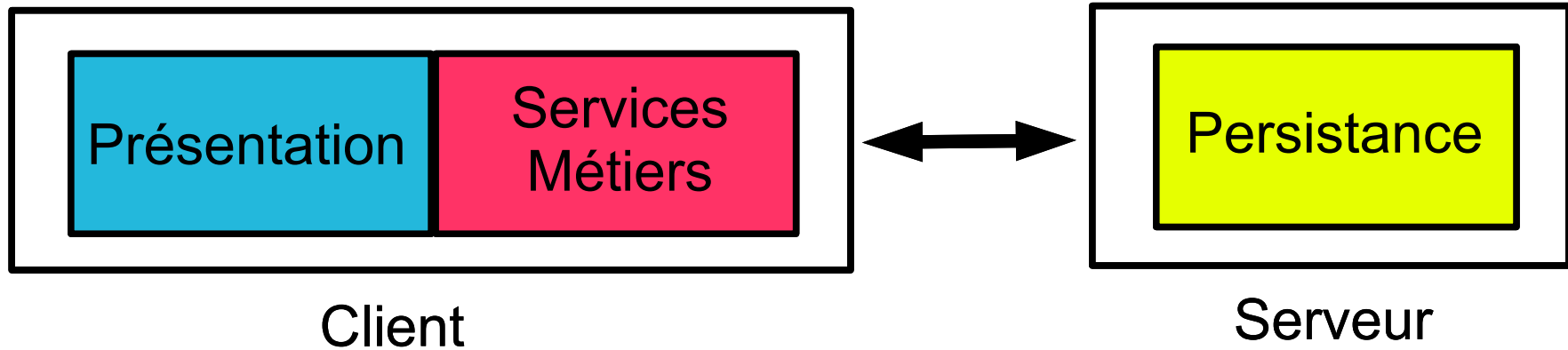


Architecture 2 – tiers

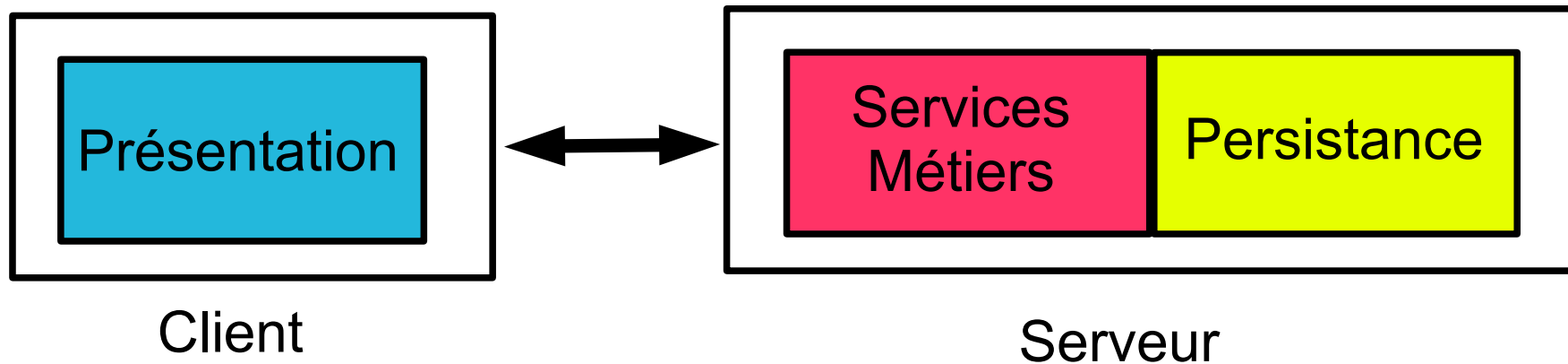
- ◆ Architecture 2 – tiers
 - ◆ Client / serveur de base, avec 2 éléments
 - ◆ Client : présentation, interface utilisateur
 - ◆ Serveur : partie persistance, gestion physique des données
 - ◆ Les services métier / la partie applicative peuvent être
 - ◆ Soit entièrement coté client, intégrés avec la présentation
 - ◆ La partie serveur ne gère que les données
 - ◆ Ex : traitement de texte avec serveur de fichiers distants
 - ◆ Ex : application accédant à une BDD distante
 - ◆ Soit entièrement coté serveur
 - ◆ La partie client ne gère que l'interface utilisateur
 - ◆ L'interface utilisateur peut même être exécutée sur le serveur
 - ◆ Fonctionnement mode terminal / mainframe
 - ◆ L'utilisateur a simplement devant lui écran / clavier / souris pour interagir à distance avec l'application s'exécutant entièrement sur le serveur
 - ◆ Soit découpés entre la partie serveur et la partie client

Architecture 2 – tiers

- ◆ Client : présentation + applicatif

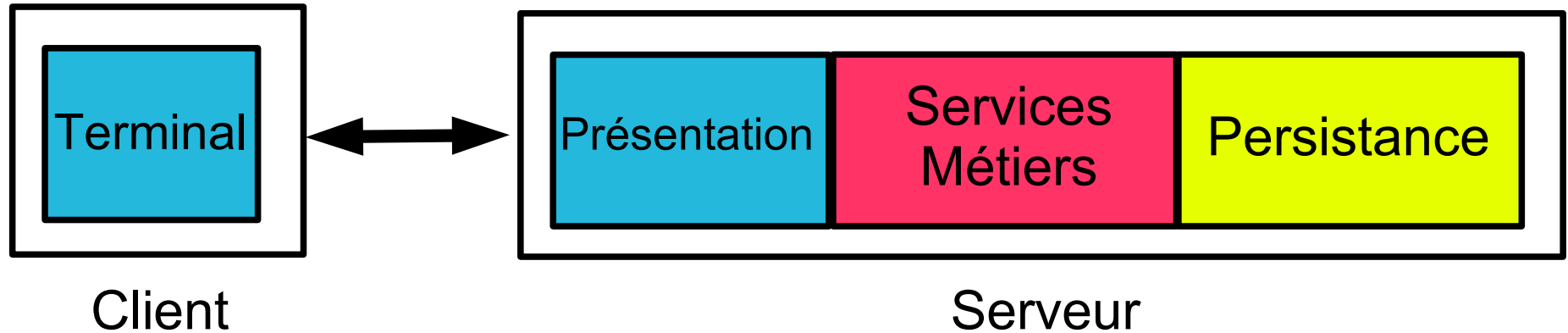


- ◆ Serveur : applicatif + gestion données

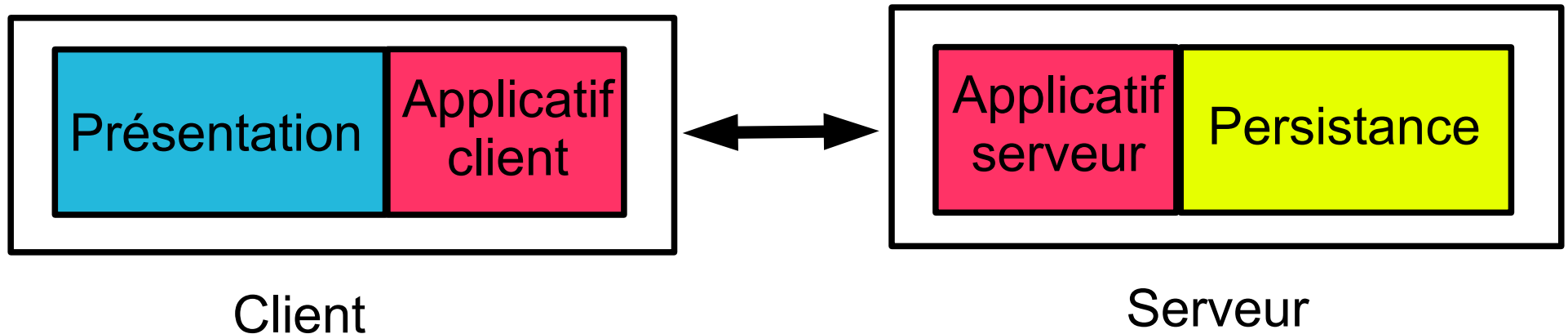


Architecture 2 – tiers

- ◆ Terminal : client intègre un minimum de la partie présentation

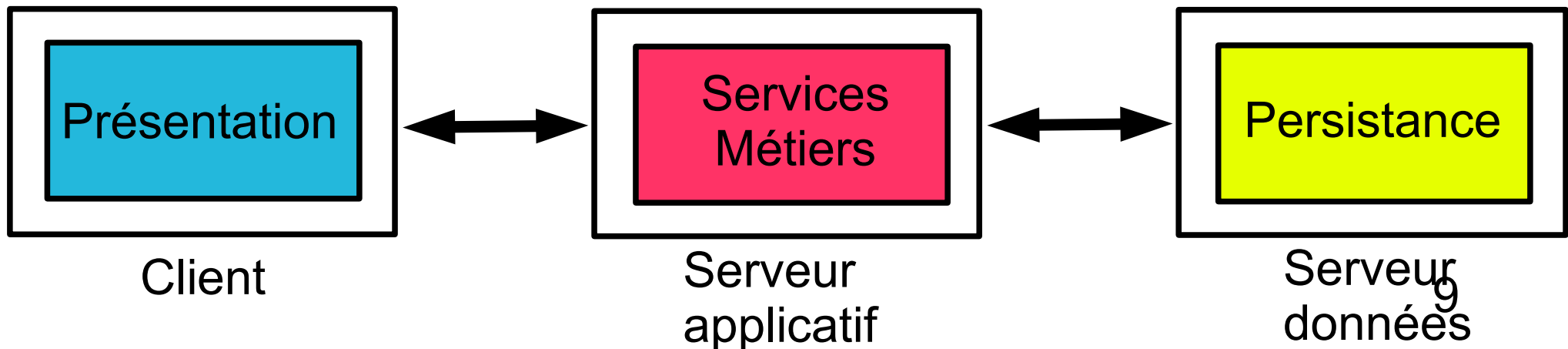


- ◆ Applicatif : découpé entre client et serveur



Architecture 3 – tiers

- ◆ Architecture 3 – tiers
 - ◆ Les 3 principaux tiers s'exécutent chacun sur une machine différente
 - ◆ Présentation
 - ◆ Machine client
 - ◆ Applicatif / métier
 - ◆ Serveur d'applications
 - ◆ Persistance
 - ◆ Serveur de (base de) données



Architecture 3 – tiers sur le web

- ◆ Architecture 3 – tiers
 - ◆ Très développée de nos jours
 - ◆ Avec généralement un fonctionnement au dessus du Web
 - ◆ Couche présentation
 - ◆ Navigateur web sur machine cliente
 - ◆ Client léger
 - ◆ Affichage de contenu HTML
 - ◆ Couche applicative / métier
 - ◆ Serveur d'applications
 - ◆ Serveur HTTP exécutant des composants / éléments logiciels qui génèrent dynamiquement du contenu HTML
 - ◆ Via des requêtes à des BDD distantes
 - ◆ Couche persistance
 - ◆ Serveur(s) de BDD de données

Architecture n – tiers

- ◆ Architecture n – tiers
 - ◆ Rajoute des étages / couches en plus
 - ◆ La couche applicative n'est pas monolithique
 - ◆ Peut s'appuyer et interagir avec d'autres services
 - ◆ Composition horizontale
 - ◆ Service métier utilise d'autres services métiers
 - ◆ Composition verticale
 - ◆ Les services métiers peuvent aussi s'appuyer sur des services techniques
 - ◆ Sécurité,
 - ◆ Transaction
 - ◆ ...
 - ◆ Chaque service correspond à une couche
 - ◆ D'où le terme de N-tiers

Architecture n – tiers

- ◆ Intérêts d'avoir plusieurs services / couches (3 ou plus)
 - ◆ Réutilisation de services existants
 - ◆ Découplage des aspects métiers et technique et des services entre eux : meilleure modularité
 - ◆ Facilite évolution : nouvelle version de service
 - ◆ Facilite passage à l'échelle : évolution de certains services
 - ◆ On recherche en général un couplage faible entre les services
 - ◆ Permet de faire évoluer les services un par un sans modification du reste de l'application
- ◆ Inconvénients
 - ◆ En général, les divers services s'appuient sur des technologies très variées : nécessite de gérer l'hétérogénéité et l'interopérabilité
 - ◆ Utilisation de framework / outils supplémentaires
 - ◆ Les services étant plus découpés et distribués, pose plus de problèmes liés à la distribution

Logique de présentation

- ◆ Les tâches liées à la présentation requièrent
 - ◆ L'interaction avec l'utilisateur
 - ◆ Réalisée par la GUI d'un client lourd : boutons, listes, menus, ...
 - ◆ Réalisée par le navigateur pour un client web
 - ◆ Via interaction plus « basique » : formulaires, liens vers des URLs ...
 - ◆ La logique de présentation
 - ◆ Le traitement des données retournées par les services métiers et leur présentation à destination de l'utilisateur
 - ◆ Réalisée par un client lourd qui généralement fait directement l'appel des services métiers sur le serveur
 - ◆ Pour un client web
 - ◆ Navigateur se contente d'afficher du code HTML qui ne peut pas être statique ici vu que le contenu dépend des données retournées
 - ◆ Doit donc exécuter du code qui récupère les données retournées par les services métiers et génère dynamiquement du code HTML
 - ◆ Logique de présentation peut être exécutée coté client ou coté serveur mais se fait généralement dans un tiers à part coté serveur

Persistence

- ◆ Couche de persistance
 - ◆ Stockage et manipulation des données de l'application
 - ◆ Plusieurs supports physique
 - ◆ Fichiers binaires, textes « de base »
 - ◆ Fichiers XML
 - ◆ Une base de données ou un ensemble de bases de données
 - ◆ Pour ce dernier cas
 - ◆ Nécessité d'envoyer à distance des requêtes (types SQL) et d'en récupérer les résultats
 - ◆ Pour réaliser cela
 - ◆ Soit c'est natif dans le langage utilisé (ex : PHP)
 - ◆ Soit on passe par des frameworks ou des API dédiés

Persistence

- ◆ Quelques standards / outils d'accès à distance à des BDD
 - ◆ RDA (Remote Data Access) de l'ISO
 - ◆ ODBC (Open Data Base Connectivity) de Microsoft
 - ◆ JDBC (Java Data Base Connectivity) de Sun
 - ◆ Framework pour le langage Java
 - ◆ Fonctionnement général
 - ◆ Gestion de requêtes SQL mais avec indépendance du SGBDR utilisé (mySQL, PostgreSQL, Oracle ...)
 - ◆ En général, seule la phase de connexion au SGBDR est spécifique
- ◆ Existe également des frameworks de plus haut niveau
 - ◆ Exemples : Hibernate (Sun, J2EE), NHibernate (libre, pour .Net)
 - ◆ Principe général
 - ◆ On définit de manière abstraite (via XML par exemple) la structure des données à manipuler
 - ◆ L'outil génère un ensemble d'opérations de manipulation de données (en java par exemple) utilisable dans un programme
 - ◆ L'outil fait en interne le lien avec le support physique (SGBDR ...) considéré

Frameworks globaux

- ◆ Une application 3/N – tiers intègre un grand nombre de technologies
 - ◆ Présentation : HTML, GUI ...
 - ◆ Applicatif : objets, composants, scripts exécutables, services ...
 - ◆ BDD : fichiers XML, SGBDR, ...
 - ◆ Protocoles de communication : RPC / RMI, HTTP
- ◆ Pour faciliter l'intégration de ces technologies et le développement d'applications
 - ◆ Editeurs offrent des frameworks globaux
 - ◆ J2EE / Java EE chez Sun
 - ◆ .Net chez Microsoft
 - ◆ Serveur d'application
 - ◆ Serveur permettant d'exécuter les parties applicatives dans le contexte de ces frameworks

Sun J2EE

- ◆ J2EE / Java EE : Java Entreprise Edition
 - ◆ Norme / standard défini par Sun pour le langage Java
 - ◆ Technologies intégrées
 - ◆ Composants logiciels : EJB
 - ◆ Applications orientées Web : JSP, servlet
 - ◆ Communication à distance : Java RMI, IIOP, JMS (Java Message Service : communication par message), Web Services
 - ◆ Gestion données distantes : JDBC, JPA (Java Persistence API)
 - ◆ Gestion d'annuaires (type LDAP) : JNDI
 - ◆ Transactions : JTA
 - ◆ Et bien d'autres ...
- ◆ Existe plusieurs serveurs d'applications J2EE
 - ◆ Versions libres
 - ◆ GlassFish, JBoss, Apache Geronimo, Jonas ...
 - ◆ Versions d'éditeurs
 - ◆ Sun Java System Application Server (basé sur GlassFish), IBM¹⁷ WebSphere, BEA WebLogic, Oracle Container for Java EE, ...

Microsoft .Net

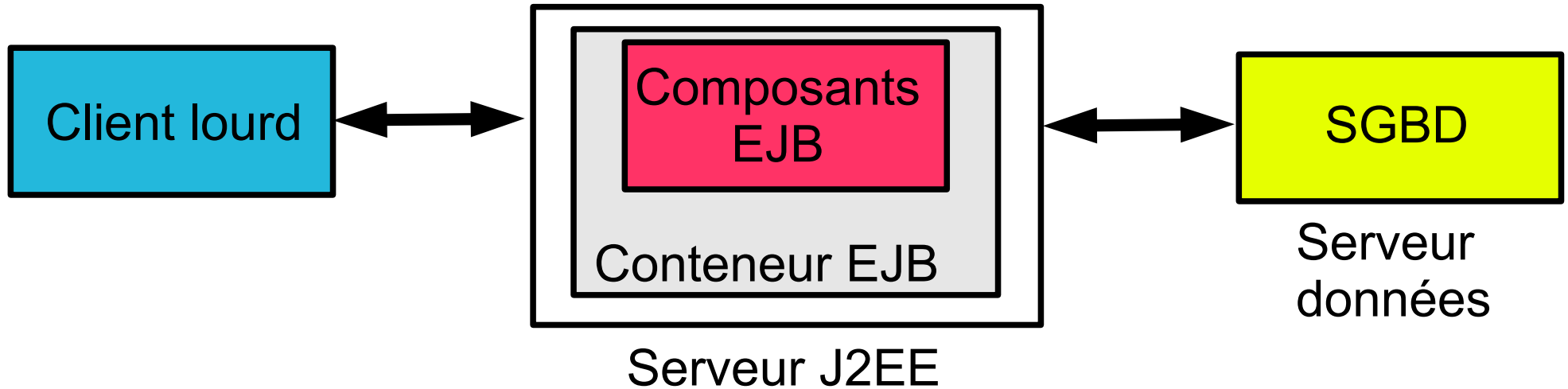
- ◆ Solution Microsoft similaire à J2EE
 - ◆ Particularité : multi-langage
 - ◆ Permet interopérabilité d'éléments écrits en C, Java, C#, J#, Eiffel, VB, ... (plus de 20 langages)
 - ◆ Traduction en code intermédiaire (MSIL) qui sera exécuté par la couche CLR (Common Language Runtime)
 - ◆ Coté Java, c'est le code Java qui était converti en byte code exécuté par la machine virtuelle Java (JVM)
 - ◆ C'est une norme également
 - ◆ La principale mise en oeuvre est bien sûr de Microsoft et pour Windows, mais il existe quelques versions libres (implémentations souvent partielles)
- ◆ Technologies intégrées
 - ◆ Composants logiciels : COM+
 - ◆ Applications orientées Web : ASP .Net,
 - ◆ Communication à distance : .Net remoting, MSMQ, Web services
 - ◆ Accès données : ADO .Net, ODBC
 - ◆ ...

Architecture 3/4 – tiers, contexte J2EE

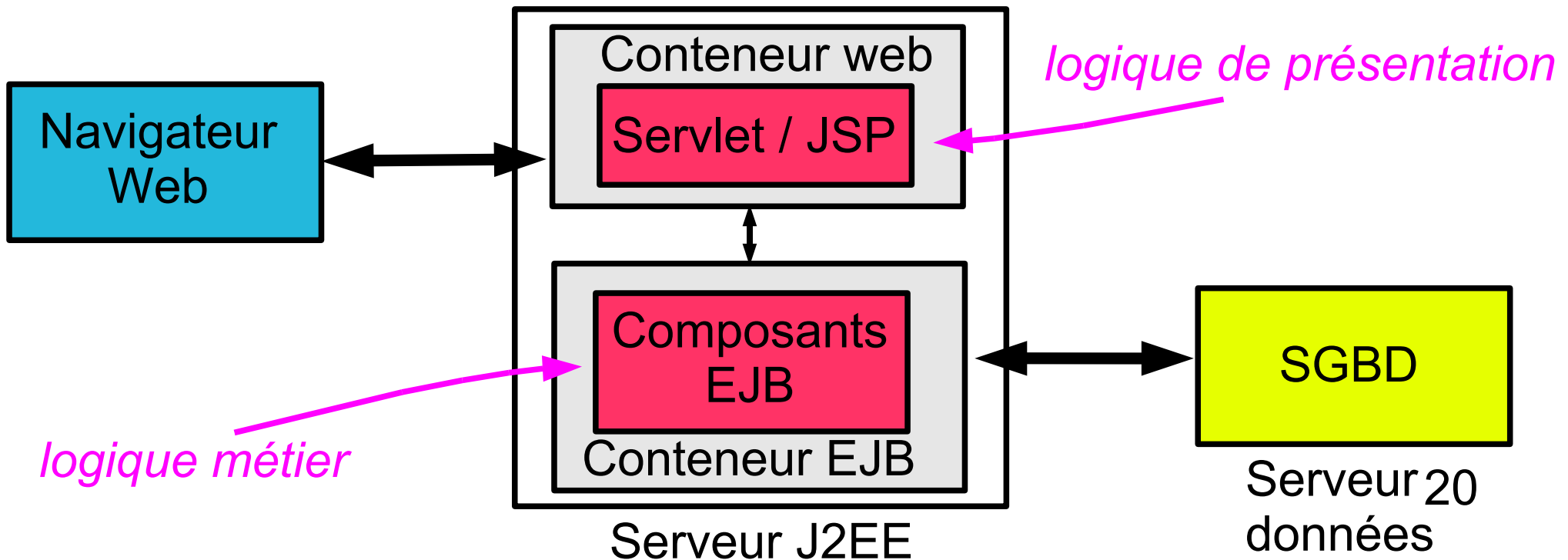
- ◆ Deux architectures générales contexte J2EE
 - ◆ Logique applicative
 - ◆ Réalisée par composants EJB
 - ◆ Communication via Hibernate ou JDBC pour attaquer BDD distante
 - ◆ Présentation
 - ◆ Avec client léger ou client lourd
 - ◆ Client léger : navigateur web
 - ◆ Intérêt : simplifie la présentation (suffit d'un navigateur)
 - ◆ Inconvénient : limite de l'interaction via HTML
 - ◆ Client lourd
 - ◆ Application « standard » coté client, gère la logique de présentation
 - ◆ Intérêt : plus grande possibilité en terme de présentation et d'interaction
 - ◆ Inconvénient : nécessite un développement dédié via des API de widgets
 - ◆ Interaction avec la partie applicative sur le serveur
 - ◆ Via JSP / Servlet pour un client léger
 - ◆ S'occupe de la logique de présentation
 - ◆ Direct si client lourd (via un middleware type RMI)

Architecture 3/4 – tiers, contexte J2EE

◆ Client lourd (3-tiers)



◆ Client léger (4-tiers)



Technologies Web : génération dynamique HTML coté serveur

JSP / Servlet

Technologies Web

◆ Présentation

- ◆ Affichage document HTML + feuilles de styles
- ◆ S'exécute dans un navigateur web : client léger
- ◆ Interaction avec l'utilisateur via des formulaires (listes, entrées texte, boutons ...) et liens

◆ Applicatif

- ◆ But : produire un contenu HTML
 - ◆ Selon les choix de l'utilisateur
 - ◆ Selon les données à afficher
- ◆ Nécessite une génération dynamique des pages HTML
 - ◆ Nécessite d'exécuter du code qui va générer ces pages

Technologies Web

- ◆ Exécution de code : deux possibilités
 - ◆ Exécution coté client
 - ◆ Le client télécharge à partir du serveur du code qui s'exécute dans le navigateur du client
 - ◆ Exemple : applet Java
 - ◆ Exécution d'un programme Java (d'une forme particulière) au sein du navigateur
 - ◆ Problèmes
 - ◆ Téléchargement peut être lourd : chaque client doit télécharger le code
 - ◆ Problème de sécurité car on exécute localement du code venant d'un élément distant
 - ◆ Exécution coté serveur
 - ◆ Du code est exécuté coté serveur pour générer de manière dynamique la page HTML qui sera envoyée au client
 - ◆ On va s'intéresser à ce type d'exécution dans ce cours

Génération pages dynamiques

- ◆ Deux principes généraux de fonctionnement, coté serveur
 - ◆ Langages de scripts
 - ◆ Double type de contenu dans une page HTML
 - ◆ Partie statique : balises HTML standards avec contenu textuel
 - ◆ Partie dynamique : du code écrit dans un langage de script
 - ◆ Ce code génère du code HTML standard
 - ◆ La page entremêle les parties statiques et dynamiques
 - ◆ Quand un navigateur demande le contenu d'une page
 - ◆ La partie dynamique est exécutée et remplacée par le HTML généré
 - ◆ Le navigateur du client reçoit donc uniquement du code HTML
 - ◆ Exécution d'un programme
 - ◆ Un programme complet s'exécute et génère du contenu HTML
 - ◆ La page est entièrement dynamique
 - ◆ Plus précisément : les parties statiques existent mais elles sont intégrées dans le code, pas écrites directement en HTML standard
 - ◆ Le navigateur demande l'exécution d'un programme et récupère le code HTML généré
 - ◆ La demande d'exécution est transparente : utilise URL standard

Génération pages dynamiques

◆ Exemples de technologies

◆ Langages de script

◆ PHP

- ◆ Langage interprété offrant l'avantage d'intégrer nativement les primitives de requêtes SQL sur des BDD

◆ ASP et ASP .Net

- ◆ Solutions Microsoft

◆ JSP (Java Server Page)

- ◆ Solution Sun pour Java

◆ Exécution de programmes

◆ CGI : Common Gateway Interface

- ◆ Historiquement, une des premières solutions

◆ Servlet

- ◆ Solution Sun pour Java : exécution de code Java respectant certaines caractéristiques

Génération pages dynamiques

- ◆ Plateforme d'exécution
 - ◆ Dans tous les cas, il faut pouvoir exécuter du code et communiquer via HTTP avec le navigateur coté client
 - ◆ Deux composants pour génération de pages dynamiques
 - ◆ Serveur HTTP standard
 - ◆ Élément (conteneur) d'exécution des programmes ou des scripts
- ◆ Exemples de serveurs / plateformes
 - ◆ Apache Tomcat
 - ◆ JSP, Servlet
 - ◆ Logiciel libre
 - ◆ Microsoft IIS
 - ◆ ASP, ASP .Net
 - ◆ Propriétaire

Servlet

- ◆ Une servlet est un programme Java particulier
 - ◆ Classe qui hérite de HttpServlet
 - ◆ Equivalent d'une applet Java, mais coté serveur
 - ◆ Classe standard du point de vue de son contenu en terme d'attributs, méthodes ...
 - ◆ Mais exécution et interactions différentes d'un objet Java standard
 - ◆ Pas d'appel de constructeur, pas de main()
 - ◆ A la création de la servlet par le serveur d'exécution
 - ◆ Appel par le serveur d'exécution de la fonction `init(ServletConfig)`
 - ◆ On peut y faire les actions qu'on ferait dans un constructeur
 - ◆ Quand la servlet est détruite
 - ◆ Appel par le serveur d'exécution de `destroy()`
 - ◆ A redéfinir si on veut effectuer certaines actions à la suppression de la servlet

Servlet

- ◆ Navigateur web coté client
 - ◆ Envoie des requêtes HTTP au serveur
 - ◆ GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE
- ◆ Dans une servlet
 - ◆ A chaque requête correspond une méthode `doXXX(...)` héritée de la classe `HttpServlet` et qui sera appelée selon la requête client
 - ◆ On redéfinit ces méthodes pour y placer le code à exécuter par la servlet
 - ◆ En pratique, pas besoin de redéfinir toutes les méthodes (GET et POST minimum)
 - ◆ De manière plus générale (mais moins recommandée), la méthode `service(...)` reçoit toutes les requêtes et peut être redéfinie directement (par défaut elle redispatche aux `doXXX(...)`)
 - ◆ Exemple pour GET
 - ◆ `doGet(HttpServletRequest request, HttpServletResponse response)`
throws `ServletException`, `IOException`
 - ◆ `request` : contient la description de la requête du client
 - ◆ `response` : à utiliser pour envoyer la réponse au client
 - ◆ Toutes les méthodes `doXXX(...)` ont la même signature

Paramètres des doXXX(...)

- ◆ `HttpServletRequest` request
 - ◆ Initialisé par l'appel du client (le navigateur)
 - ◆ Données / informations sur la requête envoyée par le client
 - ◆ On peut y récupérer notamment
 - ◆ Valeurs entrées pour un formulaire
 - ◆ Une session pour gérer un état dédié à chaque client
 - ◆ Cookies du client envoyés avec la requête
 - ◆ Informations sur l'URL utilisée pour l'appel de la servlet
 - ◆ Login de l'utilisateur s'il s'est identifié
 - ◆ Et de manière plus générale, tous les « headers » de l'appel
 - ◆ Identifiants du navigateur, type de requête (GET, PUT ...), type d'encodage supporté par le navigateur, ...

Paramètres des doXXX(...)

- ◆ HttpServletResponse response
 - ◆ A initialiser et utiliser par la servlet pour générer le résultat à envoyer au navigateur client
 - ◆ Définir le type MIME des données envoyées
 - ◆ Généralement du code HTML
 - ◆ `response.setContentType("text/html;charset=UTF-8");`
 - ◆ Mais peut utiliser n'importe quel type MIME : image/gif, audio/mp3, application/pdf, ...
 - ◆ Récupérer le flux de sortie pour envoyer les données
 - ◆ `PrintWriter getWriter()`
 - ◆ Flux texte, typiquement pour du HTML
 - ◆ `ServletOutputStream getOutputStream`
 - ◆ Flux binaire pour des images, vidéos ...
 - ◆ Ajouter/envoyer des données coté client
 - ◆ Créé un nouveau cookie (ou modifier un existant)
 - ◆ Ajouter un header aux données envoyées

Caractéristiques d'une servlet

- ◆ Unicité d'une servlet
 - ◆ Une servlet n'est créée qu'en une seule instance
 - ◆ Si plusieurs clients accèdent à l'URL d'une même servlet
 - ◆ Appelle les méthodes sur cette instance unique
 - ◆ Une servlet possède des attributs donc un état
 - ◆ Cet état est permanent et conservé (tant que la servlet existe)
 - ◆ Il est accédé / modifié par tous les clients : état global
- ◆ Etat associé à un client
 - ◆ Temporaire : mode session
 - ◆ Pour chaque client, une session est créée
 - ◆ On peut lui associer des données via des couples « clé (chaîne) / objet »
 - ◆ La session a une durée de vie configurable
 - ◆ Exemple typique d'utilisation d'une session
 - ◆ Panier d'un utilisateur sur un site de vente en ligne : conserve les produits choisis par l'utilisateur pendant son parcours sur le site
 - ◆ Permanent : cookies du navigateur

Exemple : servlet hello world

- ◆ Exemple basique de servlet
 - ◆ Génère une page qui affiche « hello word »
 - ◆ Gère et affiche un compteur qui est incrémenté à chaque accès à la servlet
- ◆ Note sur le code présenté
 - ◆ Le code qui suit a été créé sous Netbeans qui génère un squelette de code
 - ◆ Ce squelette contient une méthode principale processRequest() qui contiendra le code principal de la servlet
 - ◆ Les méthodes héritées doGet() et doPost() appellent directement cette méthode processRequest() avec les mêmes paramètres
- ◆ Une fois déployée sur le serveur, on accède à une servlet via une URL HTTP
 - ◆ La notre est contenue dans le fichier HelloWorld.java
 - ◆ URL sera de la forme : `http://www.xxxxxx.fr/XXX/HelloWorld`
 - ◆ Note : peut choisir une URL d'accès à la servlet différente du nom de la servlet Java

Exemple : servlet hello world

◆ Code de la servlet HelloWorld.java

```
public class HelloWorld extends HttpServlet {

    // compteur géré par la servlet
    protected int compteur = 0;

    protected void processRequest(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

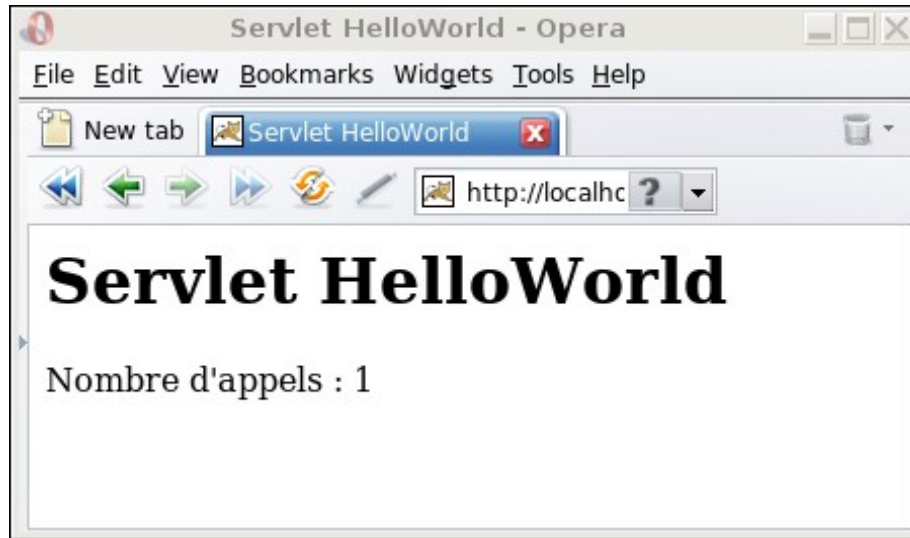
        // out : flux de sortie mode texte qui contiendra le HTML
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {

            // on écrit ligne par ligne le code HTML à afficher chez le client
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet HelloWorld</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HelloWorld</h1>");
            // on insère ici la valeur du compteur (en l'incrémentant au passage)
            out.println("<p>Nombre d'appels : "+(++compteur)+"</p>");
            out.println("</body>");
            out.println("</html>");

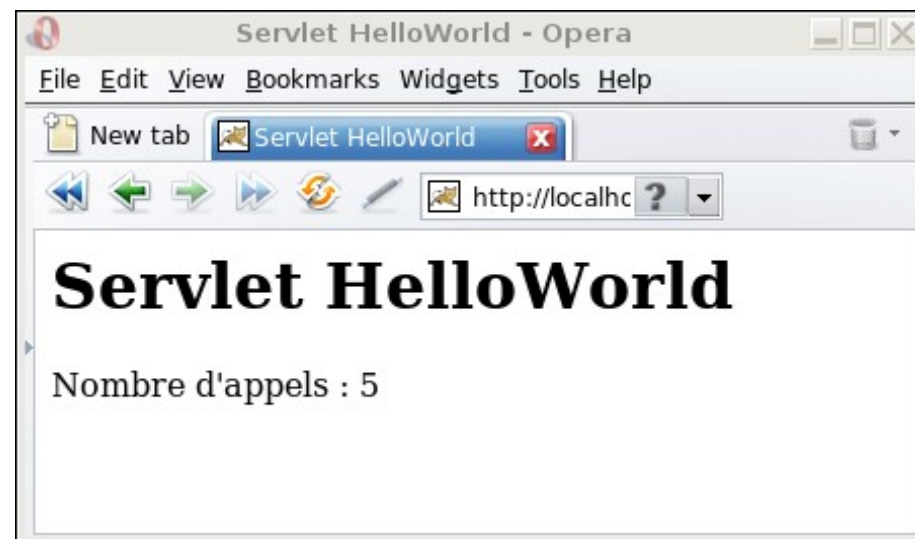
        } finally {
            out.close();
        }
    }
}
```

Exemple : servlet hello world

◆ Exemple d'exécution



Après 4 chargements
de la servlet
⇒ compteur passe à 5



Exemple : servlet hello world

- ◆ Code précédent
 - ◆ Le compteur est global : commun à tous les clients
- ◆ Variante pour gérer un compteur local à chaque client
 - ◆ Passer par les données qu'on peut associer à une session
 - ◆ Modification de processRequest()

```
try {
    // récupère la session associée au client
    HttpSession session = request.getSession(true);

    // récupère le compteur associé à la session
    Integer compteur = (Integer)session.getAttribute("compteur");
    // si valeur null : attribut « compteur » n'existe pas, ça signifie
    // que c'est la première exécution par le client, il faut créer le compteur
    if (compteur==null) {
        compteur = new Integer(1);
        session.setAttribute("compteur", compteur);
        // configure pour qu'une session dure 10 secondes max
        session.setMaxInactiveInterval(10);
    }
    (...)
    out.println("<p>Nombre d'appels : "+compteur+"</p>");
    (...)
    // incrémente le compteur
    session.setAttribute("compteur", new Integer(compteur.intValue()+1));
}
```

Exemple : passage de paramètres

◆ Autre exemple

- ◆ Manipulation de rectangles avec 3 opérations
 - ◆ Création d'un rectangle
 - ◆ Décalage du rectangle courant (non présenté en détail pour gagner de la place)
 - ◆ Calcul de la surface du rectangle courant

◆ Implémentation

- ◆ Une page HTML contient un formulaire pour entrer les paramètres et appeler la servlet avec ces paramètres
 - ◆ Note : dans l'exemple, c'est une page JSP mais c'est du HTML standard, aucun code Java n'est inclus dans la page
- ◆ Servlet
 - ◆ Une servlet gère le rectangle et exécute les 3 opérations en fonction des paramètres venant du formulaire
 - ◆ On récupère ces paramètres via :
`request.getParameter(nomParam)`

Exemple : servlet rectangle

◆ Code de la page HTML

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
</head>
<body>
  <h1>Manipulation de rectangle avec une servlet</h1>

  <form action="RectServlet" method="post">
  <p> <table border="0">
    <tr> <td>X1 ou X décalage</td> <td><input name="x1"> </td> </tr>
    <tr> <td>Y1 ou Y décalage</td> <td><input name="y1"> </td> </tr>
    <tr> <td align="right">X2</td> <td><input name="x2"> </td> </tr>
    <tr> <td align="right">Y2</td> <td><input name="y2"> </td> </tr>
  </table></p>

  <p><input type="radio" name="action" value="creer" checked="checked" />
  Créer Rectangle<br />
  <input type="radio" name="action" value="decaler" />Décaler Rectangle<br />
  <input type="radio" name="action" value="surface" />Calcul Surface</p>

  <p><input type="submit" value="Exécuter">
  <input type="reset" value="Remise à zéro"></p>
  </form>
</body>
</html>
```

Exemple : servlet rectangle

- ◆ Liens entre page HTML et la servlet
 - ◆ Le formulaire de la page HTML contient
 - ◆ Une action associée qui est l'appel de la servlet RectServlet
 - ◆ 4 champs d'entrée de noms x1, y1, x2, y2
 - ◆ Une liste de boutons radio dont selon le choix, on aura le paramètre action égal à « créer », « decaler » ou « surface »
 - ◆ Dans le code de la servlet RectServlet.java
 - ◆ On récupère ces 5 paramètres dans request
 - ◆ Notamment le paramètre action qui précisera quelle opération on veut appeler
 - ◆ La servlet conservera le dernier rectangle créé (avec un premier rectangle défini par défaut) et les fonctions de décalage et de surface seront appelées sur ce rectangle
 - ◆ Pour simplifier, on met les fonctions métiers (manipulation de rectangles) directement dans le code de la classe de la Servlet

Exemple : servlet rectangle

◆ Code de la servlet ServRectangle.java

```
public class RectServlet extends HttpServlet {

    // rectange manipulé par la servlet
    protected Rectangle rectangle;

    public int calculSurface(Rectangle rect) {
        return ( (rect.x2 - rect.x1) * (rect.y2 - rect.y1));
    }

    public Rectangle decalerRectangle(Rectangle rect, int x, y) {
        return new Rectangle(rect.x1 + x, rect.y1 + y,
                             rect.x2 + x, rect.y2 + y);
    }

    public void init(ServletConfig config)
        throws ServletException {
        // initialisation par défaut du rectangle
        rectangle = new Rectangle(10, 20, 30, 40);
    }
}
```

Exemple : servlet rectangle

◆ Code de la servlet ServRectangle.java (suite)

```
protected void processRequest(  
    HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
  
    response.setContentType("text/html;charset=UTF-8");  
    PrintWriter out = response.getWriter();  
    try {  
        // entête du document  
        out.println("<html>");  
        out.println("<head>");  
        out.println("<title>Test de Servlet : RectServlet</title>");  
        out.println("</head>");  
        out.println("<body>");  
        out.println("<h1>Servlet RectServlet</h1>");  
  
        // affichage du rectangle courant  
        out.println("<p>Etat courant de mon rectangle:"+rectangle+"</p>");  
  
        // récupère l'identifiant de l'action à réaliser  
        String action = request.getParameter("action");  
  
        // calcul de surface du rectangle courant  
        if (action.equals("surface"))  
            int surface = calculSurface(rectangle);  
            out.println("<p>Surface : "+surface+"</p>");  
    }  
}
```


Exemple : servlet rectangle

◆ Code de la servlet ServRectangle.java (fin)

```
// création d'un rectangle
if (action.equals("creer")) {
    int x1 = Integer.parseInt(request.getParameter("x1"));
    int y1 = Integer.parseInt(request.getParameter("y1"));
    int x2 = Integer.parseInt(request.getParameter("x2"));
    int y2 = Integer.parseInt(request.getParameter("y2"));

    rectangle = new Rectangle(x1, y1, x2, y2);

    out.println("<p>Rectangle créé : "+ rectangle + "</p>");
}

catch (Exception e) {
    // en cas de problème, affiche un message
    out.println("<p><b>Erreur !!</b><br />");
    out.println(e.toString()+"</p>");
}

// fin du document : lien pour retour en arrière
out.println("<p><a href=\""+request.getContextPath()+"\">
                                                    Retour</a></p>");

out.println("</body>");
out.println("</html>");
out.close();
}
```

Exemple : servlet rectangle

- ◆ Affichage de la page HTML
- ◆ On remplit les 4 champs et sélectionne « créer »

JSP Page - Opera

File Edit View Bookmarks Widgets Tools Help

New tab JSP Page

http://localhost:8084/Rectangles/index.jsp

Manipulation de rectangle avec une servlet

X1 ou X décalage

Y1 ou Y décalage

X2

Y2

Créer Rectangle
 Décaler Rectangle
 Calcul Surface

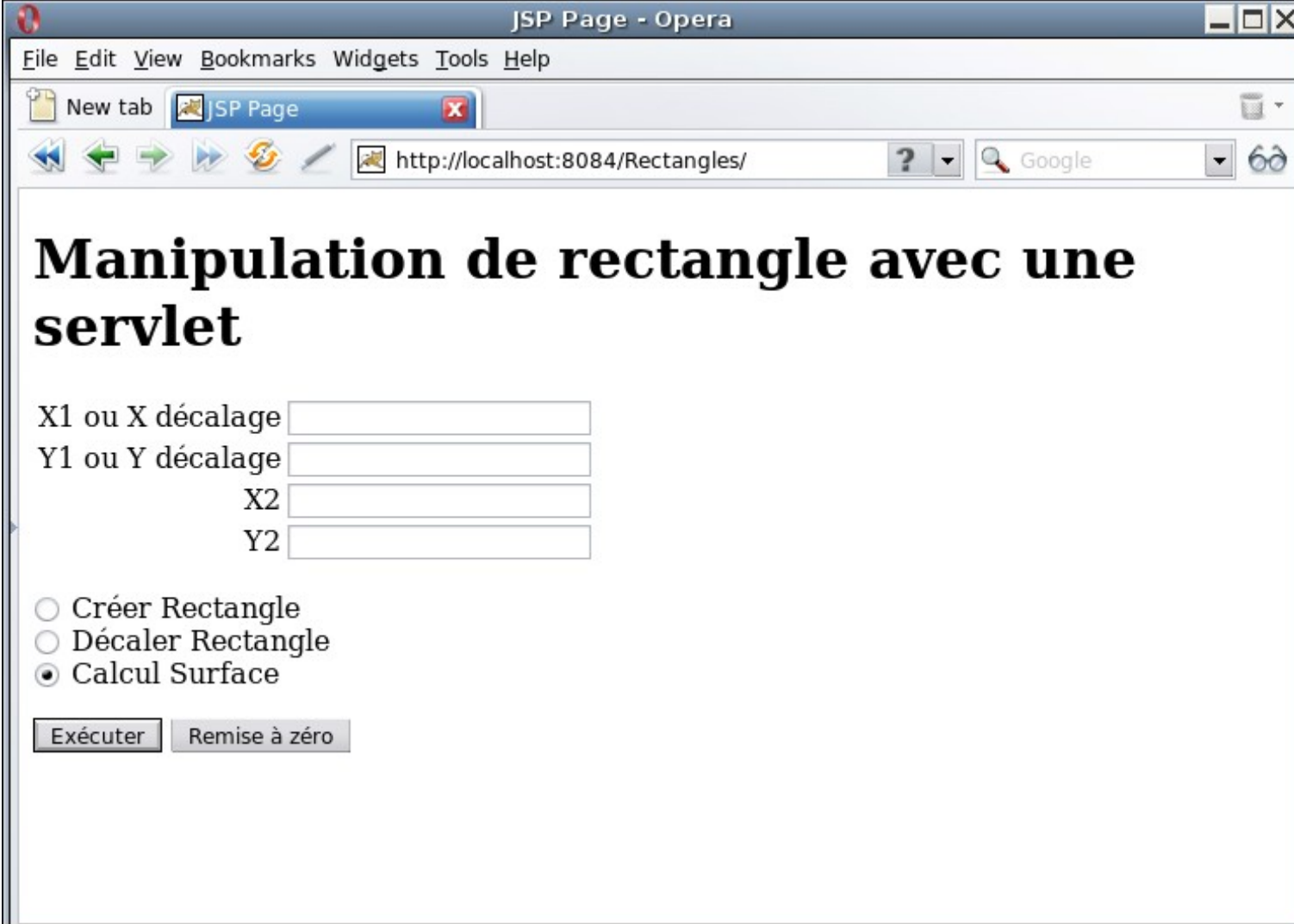
Exemple : servlet rectangle

- ◆ Après clic sur « exécuter »
- ◆ Se retrouve « sur » la servlet qui crée alors un rectangle



Exemple : servlet rectangle

- ◆ Sélection de retour pour revenir à la page HTML puis choix de « calcul surface »



JSP Page - Opera

File Edit View Bookmarks Widgets Tools Help

New tab JSP Page

http://localhost:8084/Rectangles/

Manipulation de rectangle avec une servlet

X1 ou X décalage

Y1 ou Y décalage

X2

Y2

Créer Rectangle

Décaler Rectangle

Calcul Surface

Exécuter Remise à zéro

Exemple : servlet rectangle

- ◆ Après clic sur exécuter
- ◆ Exécute la servlet avec l'opération de calcul surface
- ◆ On voit qu'elle s'applique sur le rectangle qu'on vient de créer et qui a donc été conservé



Java Server Page : JSP

- ◆ Pages mixtes contenant à la fois
 - ◆ Du code HTML statique
 - ◆ Du code Java qui est exécuté dynamiquement et génère du code HTML
 - ◆ Les parties statiques et Java sont entremêlées pour former une page JSP
 - ◆ Le navigateur client récupère une page HTML standard contenant
 - ◆ Les parties statiques
 - ◆ Les parties en Java remplacées par le code HTML qu'elles ont générées
 - ◆ En pratique
 - ◆ Le serveur d'exécution de la JSP la traduit d'abord en une servlet complète équivalente et c'est cette servlet qui est exécutée

JSP : insertion de code Java

- ◆ Attributs implicites (non déclarés) utilisables dans le code Java
 - ◆ Similaires à ce qu'on aurait dans une servlet
 - ◆ out : flux de sortie texte dans lequel on écrit le code HTML généré par les parties de code Java
 - ◆ request : paramètres de l'appel de la JSP / Servlet
 - ◆ response : pour renvoyer la réponse au client
 - ◆ session : la session spécifique au client courant
 - ◆ config : configuration de la JSP
 - ◆ Spécifique au fonctionnement des JSP
 - ◆ page : l'instance de la JSP / Servlet (équivalent du this)
 - ◆ exception : l'exception qui a été levée en cas de problème
 - ◆ application : données communes à toutes les JSP du serveur
 - ◆ pageContext : contexte de la page (gestion d'attributs ...)

JSP : insertion de code Java

- ◆ Balises spéciales pour insérer du code Java
 - ◆ Trois balises `<% ... %>`, `<%! ... %>` et `<%= ... %>`
 - ◆ Peut insérer plusieurs de ces balises dans une même page JSP
- ◆ Point important
 - ◆ JSP compilée en Servlet donc unicité de la JSP également
- ◆ `<%! ... %>`
 - ◆ Déclaration d'attributs et de méthodes
 - ◆ Ils seront globaux à tous les appels de la JSP pour tous les clients
- ◆ `<% ... %>`
 - ◆ Scriptlet : suite d'instructions Java qui sera exécutée à chaque appel de la page JSP
 - ◆ Si on y déclare des attributs, ils sont locaux à chaque appel
- ◆ `<%= expr %>`
 - ◆ Evaluate le `expr` et affiche le résultat dans le code HTML
 - ◆ Equivalent de `<% out.print(expr) %>`

JSP : insertion de code Java

- ◆ Balise spéciale `<%@ page %>`
 - ◆ Informations générales sur la JSP
 - ◆ A utiliser si les valeurs par défaut de la page sont à modifier
 - ◆ Import de classes Java
 - ◆ `<%@ page import = java.util.Vector %>`
 - ◆ Type MIME de la page
 - ◆ Par défaut du text/html
 - ◆ `<%@ page contentType = "image/png" %>`
 - ◆ Gestion des erreurs
 - ◆ Exécuter une JSP en cas d'erreur rencontrée
 - ◆ `<%@ page errorPage="erreur.jsp" %>`
 - ◆ Préciser qu'une JSP est ou pas une page d'erreur
 - ◆ `<%@ page isErrorPage="true" %>`

Exemple : JSP hello world

- ◆ Exemple basique de page JSP
 - ◆ Code statique HTML : affiche « Hello World ! »
 - ◆ Code dynamique et code Java
 - ◆ Gestion d'un compteur local et d'un compteur global d'appel
 - ◆ Affichage de ces 2 compteurs (via mélange code statique et dynamique)
- ◆ Implémentation
 - ◆ Trois insertions de code Java pour gérer les compteurs
 1. `<% ... %>` : déclaration du compteur local
 2. `<%! ... %>` : déclaration du compteur global et d'une méthode d'incrémement de ce compteur
 3. `<% ... %>` : incrémement des 2 compteurs
 - ◆ Puis affiche via des `<%= ... %>` les valeurs des compteurs

Exemple : JSP hello world

◆ Code de la page JSP

```
◆ <html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP HelloWorld</title>
  </head>

  <body>
    <h1>Hello World!</h1>

    <%
      int nbLocal = 0;
    %>

    <%!
      int nbGlobal = 0;
      void incNbGlobal() { nbGlobal++; }
    %>

    <%
      nbLocal++;
      incNbGlobal();
    %>
    <p>Compteur local : <%= nbLocal %> <br />
    Compteur global : <%= nbGlobal %></p>
  </body>
</html>
```

Légende couleur :

- Code exécuté localement
- Code déclaré globalement
- Evaluation de la valeur des attributs
- Code HTML statique

Exemple : JSP hello world

◆ Exemple d'exécution



Après 4 chargements de la servlet
⇒ compteur global passe à 5
mais le local est systématiquement à 1



Exemple : JSP hello world

- ◆ Commentaires sur le code présenté
 - ◆ Dans dernière balise `<% ... %>`
 - ◆ Aurait pu remplacer « `incGlobalNb();` » par directement « `globalNb++;` »
 - ◆ Les attributs et méthodes déclarées en global sont tous accessibles directement
 - ◆ Ne peut par contre pas déclarer une méthode pour incrémenter le compteur local
 - ◆ `void incNbLocal { nbLocal++; }`
 - ◆ Ne peut pas déclarer de méthode dans une balise `<%! ... %>`
 - ◆ Ne contient qu'une suite d'instructions à exécuter
 - ◆ Ne peut pas déclarer cette méthode dans la balise `<% ... %>`
 - ◆ L'attribut `nbLocal` est local et n'est pas connu dans le code global

Exemple : JSP rectangle

- ◆ Même exemple de gestion des rectangles
- ◆ On a besoin d'une seule page JSP au total
- ◆ Elle fera l'affichage du formulaire avec HTML standard et contiendra le code Java des opérations (en vert)
- ◆ Elle s'auto-appellera pour l'exécution de l'opération choisie
- ◆ Code la page rectangle.jsp

```
<head>
  <meta http-equiv="Content-Type"
                                     content="text/html; charset=UTF-8">
  <title>Rectangle</title>
</head>
<body>
```

```
<%@page import="rect.*" %>
```

la classe Rectangle est définie
dans un package rect

```
<h1>Manipulation de rectangles avec du JSP</h1>
```

Exemple : JSP rectangle

◆ Code la page rectangle.jsp (suite)

```
<%!  
    Rectangle rectangle = new Rectangle(10, 20, 30, 40);  
  
    public int calculSurface(Rectangle rect) {  
        return ( (rect.x2 - rect.x1) * (rect.y2 - rect.y1));  
    }  
  
    public Rectangle decalerRectangle(Rectangle rect, int x, int y){  
        return new Rectangle(rect.x1 + x, rect.y1 + y,  
                               rect.x2 + x, rect.y2 + y);  
    }  
%>  
  
<form action="rectangle.jsp" method="post">  
<p> <table border="0">  
    <tr> <td>X1 ou X décalage</td> <td><input name="x1"> </td> </tr>  
    <tr> <td>Y1 ou Y décalage</td> <td><input name="y1"> </td> </tr>  
    <tr> <td align="right">X2</td> <td><input name="x2"> </td> </tr>  
    <tr> <td align="right">Y2</td> <td><input name="y2"> </td> </tr>  
</table></p>  
  
<p><input type="radio" name="action" value="creer" checked="checked" />  
Créer Rectangle<br />  
<input type="radio" name="action" value="decaler" />Décaler Rectangle<br />  
<input type="radio" name="action" value="surface" />Calcul Surface</p>
```

Exemple : JSP rectangle

◆ Code la page rectangle.jsp (suite)

```
<p><input type="submit" value="Exécuter">  
<input type="reset" value="Remise à zéro"></p>  
</form>
```

```
<p>Rectangle courant : <%= rectangle.toString() %> </p>
```

```
<%  
try {  
    String action = request.getParameter("action");  
  
    // cas du premier affichage de la page : pas de paramètre  
    // il n'y a pas d'actions à exécuter  
    if (action == null) return;  
  
    // création d'un rectangle  
    if (action.equals("creer")) {  
        int x1 = Integer.parseInt(request.getParameter("x1"));  
        int y1 = Integer.parseInt(request.getParameter("y1"));  
        int x2 = Integer.parseInt(request.getParameter("x2"));  
        int y2 = Integer.parseInt(request.getParameter("y2"));  
  
        rectangle = new Rectangle(x1, y1, x2, y2);  
        out.println("<p>Rectangle créé : "+rectangle + "</p>");  
    }  
}
```


Exemple : JSP rectangle

◆ Code la page rectangle.jsp (suite)

```
// calcul de surface du rectangle courant
if (action.equals("surface")) {
    int surface = calculSurface(rectangle);
    out.println("<p>Surface : "+surface+"</p>");
}

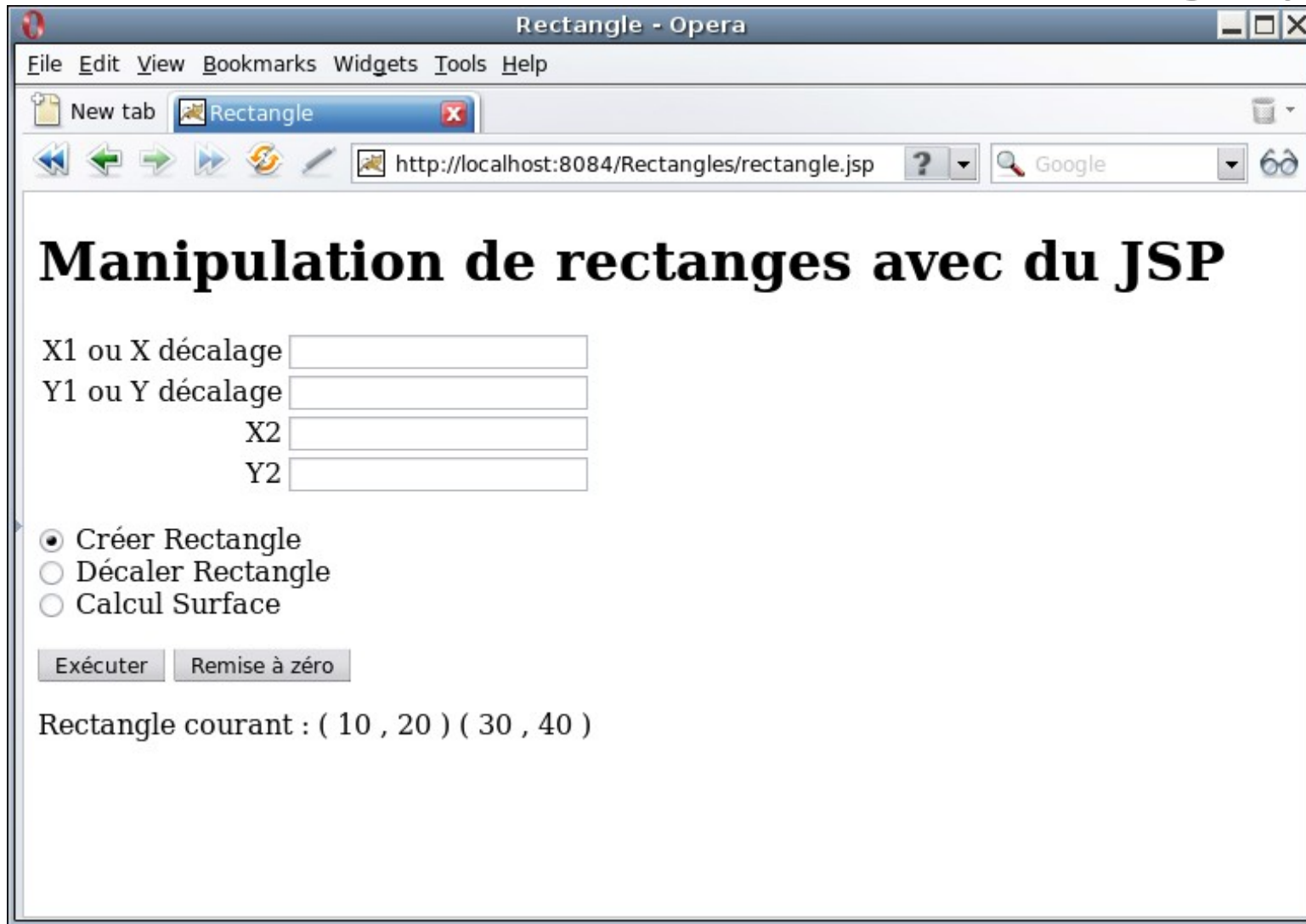
} // try
catch (Exception e) {
    out.println("<p><b>Erreur !!</b><br />");
    out.println(e.toString()+"</p>");
}
%>
</body>
</html>
```

◆ On utilise ici les attributs implicites

- ◆ request : pour récupérer les paramètres passés avec `getParameter()`
- ◆ out : flux de sortie pour générer du code HTML
- ◆ Deux façons de générer du code HTML en dynamique
 - ◆ Utilisation du out comme avec une servlet
 - ◆ Utilisation d'une balise `<%= ... %>` dans du code HTML

Exemple : JSP rectangle

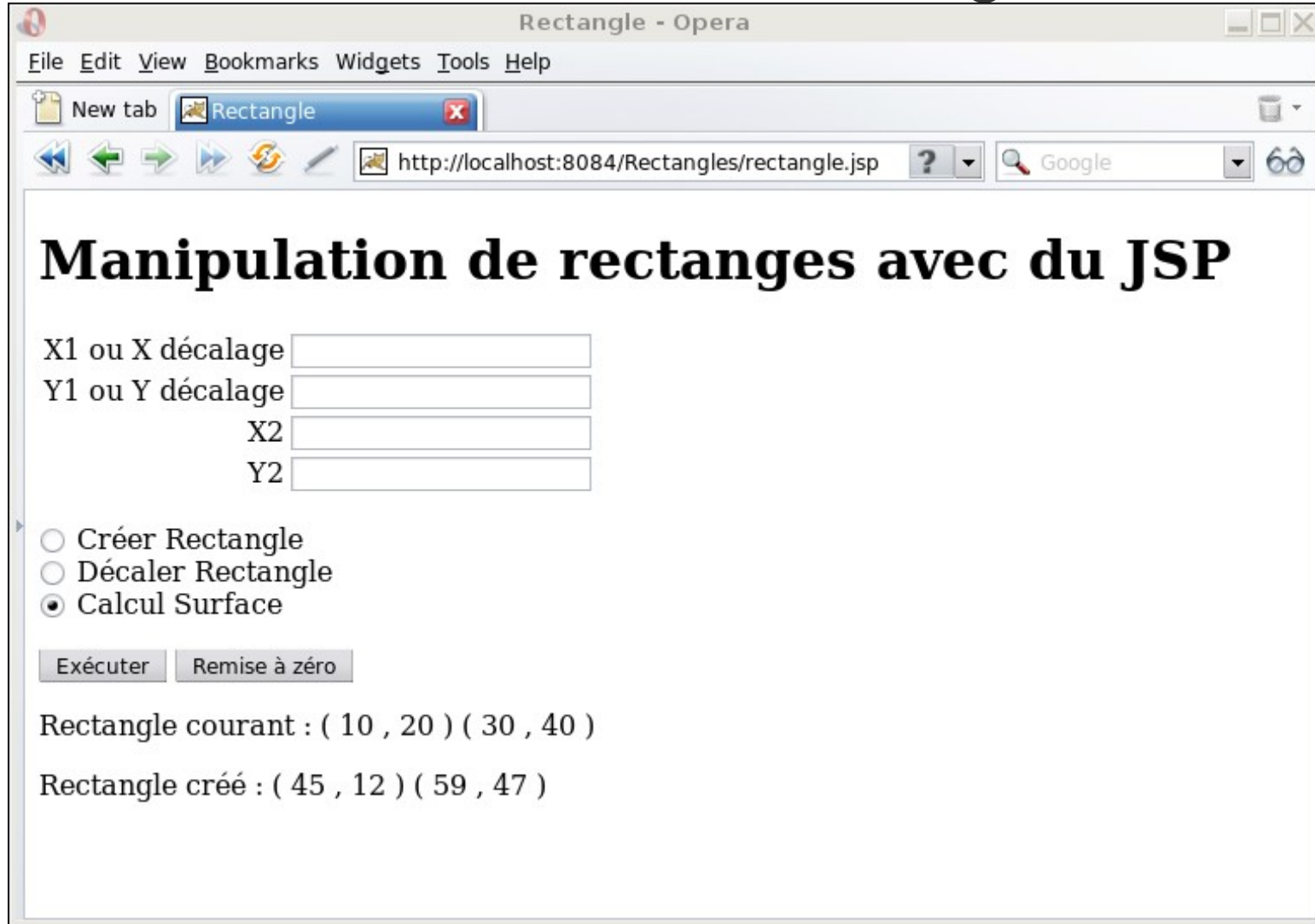
- ◆ Affichage par défaut de la page rectangle.jsp



- ◆ On va remplir les 4 champs avec valeurs 45, 12, 59, 47 et lancer l'exécution de « créer rectangle »

Exemple : JSP rectangle

◆ Résultat de la création du rectangle



The screenshot shows a web browser window titled "Rectangle - Opera". The address bar displays "http://localhost:8084/Rectangles/rectangle.jsp". The page content includes a title "Manipulation de rectangles avec du JSP", four input fields for coordinates (X1, Y1, X2, Y2), three radio buttons for actions ("Créer Rectangle", "Décaler Rectangle", "Calcul Surface"), two buttons ("Exécuter", "Remise à zéro"), and two lines of text showing the current and created rectangle coordinates.

Rectangle - Opera

File Edit View Bookmarks Widgets Tools Help

New tab Rectangle

http://localhost:8084/Rectangles/rectangle.jsp Google

Manipulation de rectangles avec du JSP

X1 ou X décalage

Y1 ou Y décalage

X2

Y2

Créer Rectangle

Décaler Rectangle

Calcul Surface

Exécuter Remise à zéro

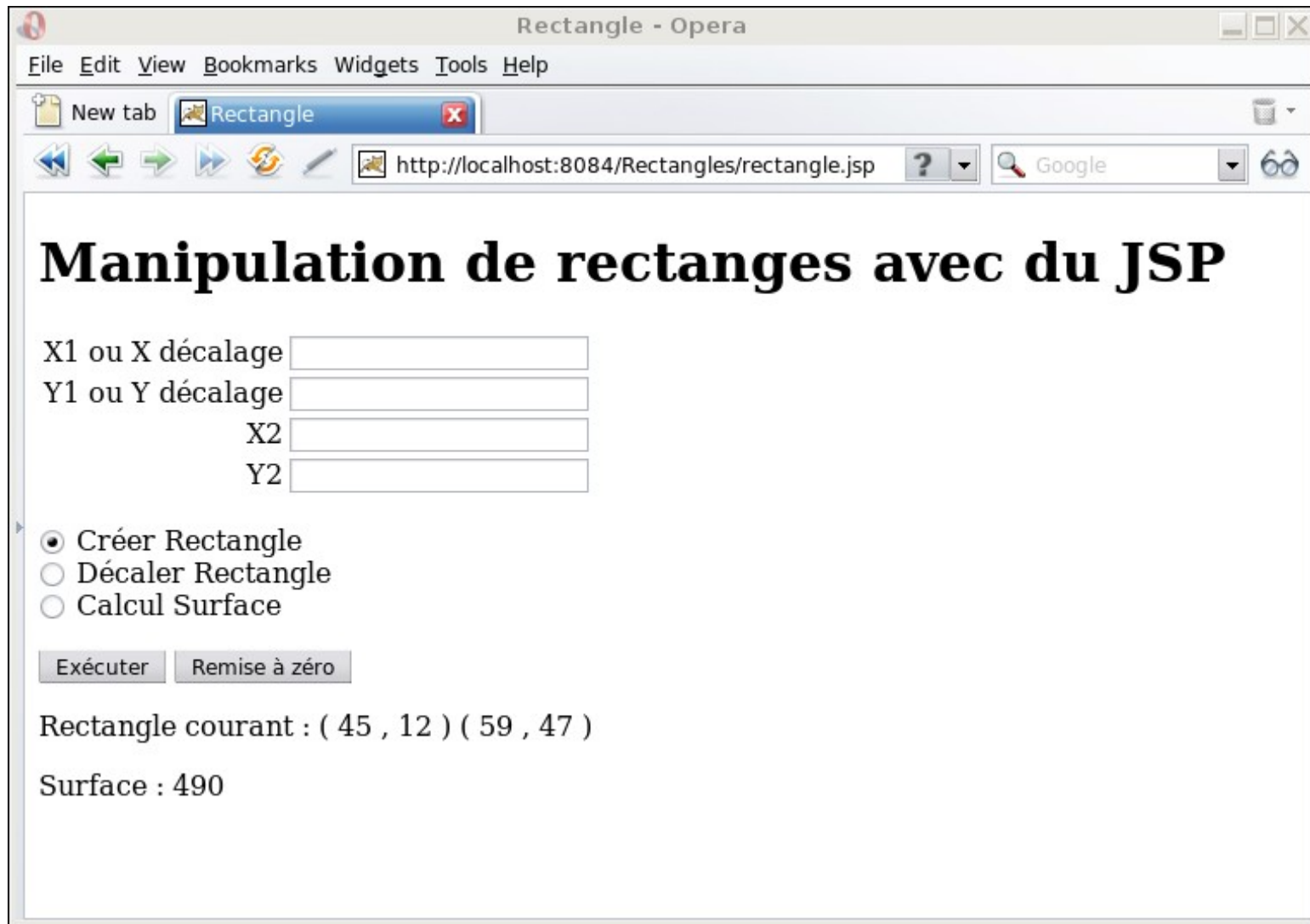
Rectangle courant : (10 , 20) (30 , 40)

Rectangle créé : (45 , 12) (59 , 47)

- ◆ On exécute maintenant à partir de cette page « calcul surface »

Exemple : JSP rectangle

◆ Résultat du calcul de surface



The screenshot shows a web browser window titled "Rectangle - Opera". The address bar displays "http://localhost:8084/Rectangles/rectangle.jsp". The page content includes a title "Manipulation de rectangles avec du JSP", input fields for "X1 ou X décalage", "Y1 ou Y décalage", "X2", and "Y2", radio buttons for "Créer Rectangle", "Décaler Rectangle", and "Calcul Surface", and buttons for "Exécuter" and "Remise à zéro". The output shows "Rectangle courant : (45 , 12) (59 , 47)" and "Surface : 490".

Rectangle - Opera

File Edit View Bookmarks Widgets Tools Help

New tab Rectangle

http://localhost:8084/Rectangles/rectangle.jsp

Manipulation de rectangles avec du JSP

X1 ou X décalage

Y1 ou Y décalage

X2

Y2

Créer Rectangle
 Décaler Rectangle
 Calcul Surface

Exécuter Remise à zéro

Rectangle courant : (45 , 12) (59 , 47)

Surface : 490

Exécution concurrente

- ◆ Modèle par défaut d'exécution
 - ◆ Servlet / page JSP est unique
 - ◆ Serveur HTTP peut avoir plusieurs requêtes d'accès à la servlet/JSP par plusieurs clients à la fois
 - ◆ Exécution multi-threadée par le serveur HTTP
 - ◆ Plusieurs threads peuvent donc exécuter en même temps du code de la même servlet/JSP
- ◆ S'assurer qu'un seul thread à la fois exécute une servlet/JSP
 - ◆ Servlet
 - ◆ Implémentation de l'interface (vide) `SingleThreadModel`
 - ◆ `public class MaServlet extends HttpServlet implements SingleThreadModel`
 - ◆ JSP
 - ◆ Passe par les paramètres généraux de la page : `isThreadSafe`
 - ◆ `<%@ page isThreadSafe="true" %>`
 - ◆ Pour une gestion plus fine et plus précise
 - ◆ Marquer des méthodes ou des parties de code de la servlet/JSP avec `synchronized`

Communication inter-éléments

- ◆ Chaque servlet / JSP possède un état global ou un état local pour chaque client
- ◆ Peut aussi vouloir avoir un état global à toutes les pages JSP ou les servlets gérées par le serveur HTTP
- ◆ Attribut implicite application dans JSP
- ◆ Permet de définir des attributs comme pour une session, mais communs à toutes les pages JSP gérées par le serveur HTTP
- ◆ Exemple : code Java pour gérer un compteur commun à toutes les pages JSP

◆ <%

```
Integer compteur = (Integer)application.getAttribute("compteur");  
if (compteur==null) {  
    // on crée le compteur qui n'existait pas  
    compteur = new Integer(0);  
    application.setAttribute("compteur", compteur);  
    out.println("<p>Création du compteur ...</p>");  
}
```

```
application.setAttribute("compteur", ++compteur);  
out.println("<p>Compteur global : "+compteur+"</p>");
```

%>

Communication inter-éléments

- ◆ Etat commun global au serveur HTTP
- ◆ Autre technique : définir une classe accessible (dans même package) par toutes les servlets / JSP
 - ◆ Y définir des attributs ou méthodes statiques
 - ◆ Exemple pour le compteur
 - ◆

```
public class Compteur {  
  
    protected static int compteur = 0;  
  
    public synchronized static int nextValue() {  
        return ++compteur;  
    }  
}
```
 - ◆ On accède par exemple au compteur global dans une servlet ou une page JSP
 - ◆

```
out.println("<p>Nombre d'appels : "+Compteur.nextValue()+"</p>");
```

Relation entre servlets/JSP

- ◆ Dans une servlet ou une page JSP, possibilité
 - ◆ D'appeler une autre servlet ou page JSP pour déléguer le traitement de la requête ou inclure un résultat supplémentaire dans celui de la servlet/page JSP courante
 - ◆ Servlet
 - ◆ Récupère le « request dispatcher »
 - ◆ `RequestDispatcher rd = request.getRequestDispatcher("maServlet");`
 - ◆ Fait suivre le traitement de la requête ou inclus un traitement réalisé par la servlet « maServlet »
 - ◆ `rd.forward(request, response);`
`rd.include(request, response);`
 - ◆ Page JSP
 - ◆ Inclusion du résultat d'une autre page
 - ◆ `<jsp:include page="maPage.jsp"></jsp:include>`
 - ◆ Faire suivre le traitement de la requête à une autre page JSP
 - ◆ `<jsp:forward page="maPage.jsp"></jsp:forward>`
 - ◆ Dans les 2 cas, peut passer des paramètres à la page appelée via inclusion
 - ◆ `<jsp:param name="compteur " value="12" />`

Relation entre éléments

- ◆ En cas d'erreur dans une page JSP, peut aussi exécuter une page JSP particulière
- ◆ Dans la page, insérer une redirection en cas d'erreur
 - ◆ `<%@ page errorPage="erreur.jsp" %>`
 - ◆ En cas d'exception levée par une instruction Java, la page erreur.jsp est alors exécutée
- ◆ Dans erreur.jsp
 - ◆ Préciser que l'on est une page d'erreur
 - ◆ `<%@ page isErrorPage="true" %>`
 - ◆ Peut alors récupérer l'exception levée et par exemple l'afficher
 - ◆ `<p>Exception levée : <%= exception %> </p>`
 - ◆ L'attribut implicite exception est une exception Java classique, on peut appeler dessus des méthodes comme getMessage(), getCause(), printStackTrace(), ...

Retour sur l'exemple des rectangles

- ◆ Dans une JSP, une part importante du code est dédiée à simplement afficher le contenu d'objets
- ◆ Même avec des `<%= ... %>` ça peut être relativement lourd à réaliser
- ◆ Exemple
 - ◆ Reprenons notre manipulation de rectangles
 - ◆ La servlet RectServlet gère le rectangle courant et exécute les actions associés
 - ◆ Une page HTML affiche un formulaire pour rentrer les différents champs
 - ◆ Si l'utilisateur veut modifier une des 4 valeurs de coordonnée (y1 par exemple) du rectangle courant, il doit réentrer les 4 valeurs
 - ◆ On pourrait remplir par défaut les 4 champs avec le contenu du rectangle courant

Retour sur l'exemple des rectangles

- ◆ Modifie RectServlet.java pour ajouter le rectangle en tant qu'attribut de la session
- ◆

```
protected void processRequest(...) {
```

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

```
    // premier appel de la servlet, on crée l'attribut rectangle
    if (session.getAttribute("rectangle") == null)
        session.setAttribute("rectangle", rectangle);
```

```
    if (action.equals("creer")) {
        // récupère les paramètres x1, x2, y1, y2 dans la requête
        (...)
        rectangle = new Rectangle(x1, y1, x2, y2);
        out.println("<p>Rectangle créé : "+rectangle + "</p>");
        // met à jour l'attribut de la session avec le nouveau rectangle
        session.setAttribute("rectangle", rectangle);
    }
    (...)
}
```

Retour sur l'exemple des rectangles

- ◆ Transforme la page HTML associée en page JSP pour pouvoir récupérer le rectangle associé à la session
- ◆

```
<%@page import = "rect.Rectangle" %>
<% Rectangle rect =(Rectangle) session.getAttribute("rectangle"); %>

<h1>Manipulation de rectangle avec une servlet</h1>
<form action="RectServlet" method="post">
<p> <table border="0">
  <tr>
    <td>X1</td>
    <td><input name="x1" value="<%=rect.getX1() %>"></td>
  </tr>
  (...)

```
- ◆ On récupère l'attribut « rectangle » associé à la session
- ◆ On s'en sert pour mettre une valeur par défaut dans les champs d'entrée des 4 valeurs de coordonnées

Retour sur l'exemple des rectangles

- ◆ Le code présenté semble pertinent ... mais ne marche pas
 - ◆ Au premier chargement de la JSP, aucun appel n'a encore été fait sur la Servlet : pas d'attribut « rectangle » associé à la session
 - ◆ L'attribut rect vaut donc null et `<%= rect.getX1() %>` lève une exception
 - ◆ Il faut gérer explicitement le cas où l'attribut n'est pas encore défini
 - ◆

```
<%@page import = "rect.Rectangle" %>
<% Rectangle rect = (Rectangle) session.getAttribute("rectangle");
String x1 = "", x2 = "", y1 = "", y2 = "";
if (rect != null) {
    x1 = new Integer(rect.getX1()).toString();
    (...)
} %>
<h1>Manipulation de rectangle avec une servlet</h1>
<form action="RectServlet" method="post">
<p> <table border="0">
    <tr>
        <td>X1</td>
        <td><input name="x1" value="<%= x1 %>"></td>
    </tr>
</table>
    (...)

```

Retour sur l'exemple précédent

- ◆ Autre solution pour récupérer le rectangle courant
 - ◆ Expression UEL (Unified Expression Language)
 - ◆ Simplifie l'accès aux attributs (dans le sens des attributs associés à une session)
 - ◆ La page JSP devient tout simplement

```
<h1>Manipulation de rectangle avec une servlet</h1>
<form action="RectServlet" method="post">
<p> <table border="0">
  <tr>
    <td>X1</td>
    <td><input name="x1" value=" ${sessionScope.rectangle.x1}"></td>
  </tr>
</table>
(...)

```
 - ◆ Le champ d'entrée X1 est initialisé par le contenu de la propriété x1 de l'attribut rectangle associé à la session
 - ◆ Si l'attribut rectangle n'existe pas, ne génère pas d'erreur, retourne simplement une chaîne vide
 - ◆ Plus besoin de vérifier explicitement que l'attribut existe

UEL & JSTL

- ◆ Unified Expression Language (UEL)
 - ◆ Langage d'expression permettant simplement dans une page JSP
 - ◆ De récupérer des attributs associés à des contextes
 - ◆ Attributs de la session, de l'application, du header ou de la requête HTTP ...
 - ◆ D'écrire des expressions de calcul ou de test
 - ◆ Addition, multiplication, modulo ...
 - ◆ Comparaisons de valeurs, opérateurs logiques (OU, ET ...)
- ◆ Java Server Pages Standard Tag Library (JSTL)
 - ◆ Ensembles de balises offrant des fonctionnalités pour manipuler des fichiers XML, accès à des BDDs, l'internationalisation ...
 - ◆ Ensemble « core » permet notamment de manipuler simplement des ensembles de données, faire des tests et d'associer des balises HTML (ou autres) à ces traitements ...
- ◆ En combinant UEL et JSTL
 - ◆ Facilite grandement dans une page JSP l'affichage de données retournés par la logique métier

UEL : attributs, propriétés

◆ Attribut

- ◆ Instance d'une classe Java respectant quelques contraintes structurelles : Java Beans
 - ◆ Définit des propriétés
 - ◆ Un attribut (de la classe) avec un getter et un setter associé
 - ◆ Définit un constructeur sans paramètre
- ◆ On peut ensuite accéder aux propriétés des attributs via une notation pointée à partir d'un certain contexte
 - ◆ Exemple : `sessionScope.rectangle.x1`
 - ◆ Contexte : `sessionScope`
 - ◆ Attribut : `rectangle`
 - ◆ Propriété : `x1`
 - ◆ Revient à exécuter de manière réflexive
 - ◆ `((Rectangle)sessionScope.getAttribute("rectangle")).getX1()`
 - ◆ On voit bien l'obligation d'avoir un getter/setter associé à chaque propriété pour pouvoir y accéder
- ◆ Si l'attribut ou une de ses propriétés n'est pas définie (valeur null) génère une chaîne vide

UEL : contextes et objets implicites

- ◆ Objets implicites, dont certains similaires aux objets implicites JSP
 - ◆ pageContext, param, paramValues, header, headerValues, cookie, initParam
 - ◆ Contextes (« scope »), à partir desquels on peut associer des attributs
 - ◆ pageScope : la page JSP
 - ◆ requestScope : la requête HTTP
 - ◆ sessionScope : la session associée avec le navigateur client
 - ◆ applicationScope : contexte global à toutes les pages JSP du serveur

UEL : tableaux et map

- ◆ Accède aux données d'un tableau (array, list ...) ou d'une map via une notation indexée
 - ◆ att[index]
 - ◆ Avec index qui est soit la clé pour une map, soit l'index pour un tableau
 - ◆ Peut aussi être le contenu d'une variable
 - ◆ Pour un tableau, peut utiliser un index sous forme d'entier ou de chaîne
- ◆ Exemple : une servlet exécute le code suivant

```
◆ Rectangle rect1 = new Rectangle(10,10,20,20);  
  Rectangle rect2 = new Rectangle(30, 30, 50, 50);
```

```
HashMap<String, Rectangle> mapRect = new HashMap<String, Rectangle>();  
Rectangle[] arrayRect = new Rectangle[2];  
int index = 1;
```

```
mapRect.put("premier",rect1);  
mapRect.put("second", rect2);  
arrayRect[0] = rect1;  
arrayRect[1] = rect2;
```

```
session.setAttribute("mapRect", mapRect);  
session.setAttribute("arrayRect", arrayRect);  
session.setAttribute("index", index);
```

UEL : tableaux et map

- ◆ La servlet forward la requête à la page JSP
 - ◆

```
<p>  
Map -> "premier" : ${sessionScope.mapRect["premier"].x1}<br />  
Map -> 'second' : ${sessionScope.mapRect['second'].x1}<br />  
Map -> "troisieme" : ${sessionScope.mapRect["troisieme"].x1}<br />  
Array -> 0 : ${sessionScope.arrayRect[0].x1}<br />  
Array -> '1' : ${sessionScope.arrayRect['1'].x1}<br />  
Array -> 2 : ${sessionScope.arrayRect[2].x1}<br />  
Array -> index : ${sessionScope.arrayRect[sessionScope.index].x1}<br />  
</p>
```
- ◆ Affichage résultant
 - ◆ Map -> "premier" : 10
Map -> 'second' : 30
Map -> "troisieme" :
Array -> 0 : 10
Array -> '1' : 30
Array -> 2 :
Array -> index : 30
 - ◆ Si des éléments dans la map n'existent pas ou si on dépasse l'index max du tableau, renvoie une chaîne vide

UEL : opérateurs

- ◆ Opérateurs logiques
 - ◆ && (ou and), || (ou or), ! (ou not)
- ◆ Opérateurs de comparaison
 - ◆ == (ou eq), != (ou ne), > (ou gt) ...
- ◆ Opérateurs de calcul
 - ◆ +, -, *, / (ou div), % (ou mod)
 - ◆ Types de nombre disponibles : entier et réel
- ◆ Divers
 - ◆ empty : renvoie vrai si valeur « vide »
 - ◆ Egale à null, chaîne vide, un tableau vide ou une map vide
 - ◆ ? : choix conditionnel
 - ◆ test ? choixVrai : choixFaux

JSTL : core

- ◆ Pour utiliser du JSTL core dans une page JSP, insérer la balise suivante dans la page
 - ◆ `<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`
 - ◆ Au déploiement, ne pas oublier le lien vers la librairie (.jar) externe implémentant cet ensemble de balises
- ◆ Affichage d'une valeur
 - ◆ `<c:out value="valeur" />`
- ◆ Test d'une valeur
 - ◆ `<c:if test="condition"> ... </c:if>`
- ◆ Choix conditionnels
 - ◆ `<c:choose>`
 - `<c:when test="condition1"> ... </c:when>`
 - `<c:when test="condition2"> ... </c:when>`
 - ...
 - `<c:otherwise> ... </c:otherwise>`
 - `</c:choose>`

JSTL : core

- ◆ Parcours d'une liste / tableau
 - ◆ `<c:forEach items="laListe" var="eltCourant">`
...
`</c:forEach>`
 - ◆ `items` : la liste ou le tableau à parcourir
 - ◆ `var` : variable qui contiendra l'élément courant
- ◆ Parcours d'une map
 - ◆ Comme pour une liste mais sur l'élément courant
 - ◆ Accède à sa clé via : `eltCourant.key`
 - ◆ Accès à sa valeur via : `eltCourant.value`
- ◆ Autres tags existants
 - ◆ `<c:forTokens>`, `<c:set>`, `<c:remove>`, `<c:catch>`,
`<c:import>`, `<c:url>`, `<c:redirect>`, `<c:param>`

JSTL : core

◆ Exemple

- ◆ `<c:if test="\${empty sessionScope.arrayRect[2].x1}">`
Pas d'élément d'index 2 dans le tableau
`

`
`</c:if>`
Surface des rectangles :
`<c:forEach items="\${sessionScope.arrayRect}" var="rect">`
-> `<c:out value="\${(rect.x2 - rect.x1) * (rect.y2 - rect.y1)}"/>`
`</c:forEach>`

◆ Affichage résultant

- ◆ Pas d'élément d'index 2 dans le tableau

Surface des rectangles :

-> 100

-> 400

Servlet vs JSP

◆ Servlet

- ◆ C'est du java standard qui est exécuté
- ◆ Les lignes « `out.println()` » servent à générer le code HTML qui sera affiché
- ◆ Problème : une bonne partie de ce code HTML est statique
 - ◆ Entête, affichage du titre ...
 - ◆ Un peu lourd à générer de la sorte

◆ Page JSP

- ◆ Permet de « mélanger » du code HTML standard avec du Java standard
- ◆ Meilleure découpage des différentes parties qu'avec des servlets
- ◆ Moins lourd à programmer qu'une servlet
 - ◆ Simplification des affichages des données, surtout si combiné avec UEL et JSTL
 - ◆ Mais moins structuré du point de vue du code Java

Servlet et JSP

- ◆ Servlet / JSP
 - ◆ Technologie coté serveur permettant donc la génération dynamique de page HTML
 - ◆ Intérêt est de profiter de la puissance d'un langage Objet
- ◆ En pratique
 - ◆ Le code embarqué dans les Servlet / JSP n'intègre pas de préférence le code et la logique métier
 - ◆ On s'appuie sur d'autres éléments (composants EJB par exemple)
 - ◆ Pour un serveur applicatif Web, découper son rôle en
 - ◆ Logique de présentation
 - ◆ Manière de présenter et de générer les pages aux clients
 - ◆ Logique métier
 - ◆ Appelée par la logique applicative pour générer les pages

***Frameworks de persistance :
introduction à Hibernate***

Problématique

- ◆ Exemple simple d'application
 - ◆ Gérer une liste de personnes
 - ◆ Programmation partie métier / client en Java
 - ◆ Classe qui contient les informations sur une personne
 - ◆

```
public class Personne {  
    String nom;  
    Int age;  
  
    // méthodes d'accès aux attributs age et nom  
}
```
 - ◆ Lors du stockage d'une personne, on lui associera un identifiant unique
 - ◆ Deux techniques de stockages de l'ensemble de personnes
 - ◆ Directement en Java via une HashMap
 - ◆

```
HashMap<Integer, Personne> personnes;
```
 - ◆ Via un SGBD avec accès en SQL aux données
 - ◆ Stockage sous forme de tables SQL

```
personne (int id, varchar(20) nom, int age)
```

Problématique

- ◆ Code pour récupérer une personne via son id
- ◆ Pur Java

```
public Personne getPersonne(int id) throws InvalidIdException {  
  
    // si la personne n'existe pas, lève exception  
    if ( ! (personnes.containsKey(new Integer(id))))  
        throw new InvalidIdException("invalid index value : "+id);  
  
    // sinon, retourne son identifiant  
    return (personnes.get(new Integer(id)));  
}
```

- ◆ Manipule directement la HashMap personnes
- ◆ Code « naturel », programmation Java standard

Problématique

◆ Java avec stockage BDD et accès via JDBC

```
◆ public int getId(Personne p) throws InvalidIdException {
    try {
        Statement requete = connection.createStatement();

        // requête SQL pour rechercher la personne
        ResultSet result = requete.executeQuery("SELECT NOM, AGE
        FROM PERSONNE WHERE ID=\""+id+"\"");

        // regarde si la requête a renvoyé quelque chose : si ça n'est pas
        // le cas, la personne n'existait pas
        if (!result.next())
            throw new InvalidIdException("invalid index value : "+id);

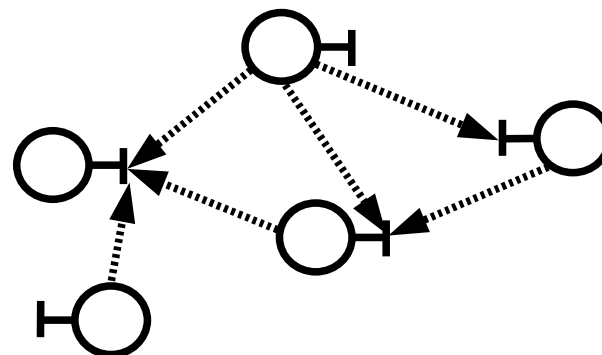
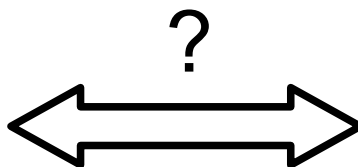
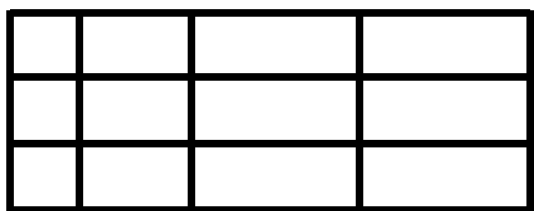
        // sinon, à partir du résultat retourné, instancie une personne
        // avec les valeurs du résultat
        else return new Personne(result.getInt("age"), result.getString("nom"));
    }
    catch(SQLException e) {
        System.err.println(e); }
}
```

Problématique

- ◆ Java avec stockage BDD et accès via JDBC
 - ◆ Nécessite des requêtes SQL
 - ◆ Utilisation d'un framework dédié : JDBC
 - ◆ Sort du format « standard » de représentation Java des données
 - ◆ Une requête de type SELECT retourne un ResultSet
 - ◆ C'est à dire un ensemble de ligne de plusieurs colonnes
 - ◆ On accède aux éléments du ResultSet en naviguant selon les lignes et les colonnes
 - ◆ Peu pratique mais difficile de faire autrement vu ce que retourne de manière native les requêtes SQL
 - ◆ Sauf à passer par des SGBD objet-relationnel

Problématique

- ◆ Java avec stockage BDD et accès via JDBC
 - ◆ Au delà de l'accès à faire à distance
 - ◆ Qui complexifie forcément les choses mais ne peut pas y couper
 - ◆ Représentation des données très différentes
 - ◆ Coté BDD
 - ◆ Structure en table avec langage de requête dédié
 - ◆ Récupère toujours une « sous-table » via le résultat de l'exécution d'une requête de type SELECT
 - ◆ Coté Java
 - ◆ Instances de classe (Personne ici) : objets
 - ◆ Les objets sont associés entre eux et forment un graphe d'objet dans l'application



Problématique

- ◆ Doit être capable de faire la correspondance d'une représentation à une autre
- ◆ Pas simple : d'une représentation objet à une représentation relationnelle
- ◆ Solution
 - ◆ Le faire à la main
 - ◆ Comme pour notre exemple
 - ◆ Utiliser une base de données relationnelle orientée objet
 - ◆ Coté BDD, la structure de stockage intégrera des concepts objets
 - ◆ Passage plus facile du programme au stockage dans la base
 - ◆ Mais toujours besoin de faire des requêtes explicites
 - ◆ Utiliser un framework de persistance

Framework de persistance

◆ Principes

- ◆ S'abstraire des problématiques de stockage des données
 - ◆ Pouvoir manipuler directement des entités métiers indépendamment de leur stockage physique
- ◆ Définition de mappings entre
 - ◆ La structure des classes utilisées dans l'application
 - ◆ Le stockage physique utilisé par le support de persistance
 - ◆ Généralement, un SGBD relationnel
- ◆ Lors de l'exécution de l'application
 - ◆ Création, suppression, manipulation, collections d'objets ... gérés de manière classique, programmation « standard »
 - ◆ Le framework fait le lien entre le graphe d'objet en mémoire dans l'application et le stockage physique
 - ◆ Typiquement : envoi de requête SQL au SGBD pour récupérer des données, les modifier, les ajouter / supprimer

Framework de persistance

- ◆ Intérêt des frameworks de persistance
 - ◆ Plus besoin de se préoccuper du stockage physique et des problèmes de correspondance
 - ◆ Comme on le voit dans notre exemple de l'atelier
 - ◆ Ecriture des requêtes SQL via JDBC est assez lourd
 - ◆ L'implémentation des accès aux données peut représenter un temps important lors du développement d'une application
 - ◆ On gagne ainsi en temps de développement et en indépendance des supports physiques
 - ◆ Suffit de changer les règles de mappings sans modifier le code de l'application
- ◆ Hibernate
 - ◆ Framework de persistance libre pour le monde Java
 - ◆ NHibernate pour .Net
 - ◆ Standard de fait, très utilisé dans l'industrie
 - ◆ Utilisé également par JPA (framework standard de persistance sous Java – Java Persistence API)

Hibernate

- ◆ Principes généraux d'Hibernate
 - ◆ Définition des structures de données / entités manipulées
 - ◆ Relation entre la structure coté BDD et la structure des entités
 - ◆ Via des fichiers XML
 - ◆ Ou des classes Java annotées
 - ◆ Coté Java, l'entité est une classe Java dite persistante
 - ◆ Un POJO : Plain Old Java Objet
 - ◆ Classe Java totalement standard avec définition d'attributs, d'accesseurs, de constructeurs pour gestion données de l'entité
 - ◆ Ne s'appuie donc pas sur des frameworks ou librairies particuliers, contrairement par exemple aux entités d'EJB
 - ◆ Persistante : son contenu est lié avec les données physiques sur la BDD et est sauvegardé dans cette BDD
 - ◆ Définition, en XML, d'un ensemble de fichiers de configuration
 - ◆ Mapping d'une entité vers un support physique
 - ◆ Configuration générale de l'accès au support physique

Hibernate

- ◆ Principes généraux d'Hibernate (fin)
 - ◆ Pour gérer la persistance, on crée une session Hibernate
 - ◆ Via cette session, peut
 - ◆ Récupérer sous forme d'instances de POJO des données stockées par le support physique
 - ◆ Utilisation notamment d'HQL (Hibernate Query Language)
 - ◆ Modifier le contenu de ces POJO, leurs associations, en ajouter, en supprimer, ...
 - ◆ Le framework fera le lien entre les modifications locales et le stockage physique
 - ◆ Fonctionne en mode transactionnel pour les modifications
 1. Création d'une transaction
 2. Modification des objets
 3. Validation de la modification (commit) ou annulation en cas de problème (rollback)

Exemple Hibernate

- ◆ Exemple de gestion de sports et de disciplines
 - ◆ Un sport contient plusieurs disciplines
 - ◆ Tables du SGBD (Oracle ici)
 - ◆ sport (*code_sport*, intitule)
 - ◆ discipline (*code_discipline*, intitule, code_sport)
 - ◆ Ici coté Java, on choisira d'avoir des objets de type Sport et Discipline
 - ◆ Avec les mêmes structures de données que ce qui est stocké dans la base
 - ◆ Mais ça n'est pas une obligation d'avoir exactement la même chose (voir plus loin)

Exemple Hibernate

◆ Contenu des tables pour l'exemple

◆ Table sport

code	intitule
1	athletisme
2	ski
3	natation

◆ Table discipline

code	intitule	code
discipline		sport
1	100 metres	1
2	200 metres	1
3	saut en hauteur	1
4	saut en longueur	1
5	100m 4 nages	3
6	100m papillon	3
7	marathon	1

Hibernate : ex. configuration générale

◆ Configuration générale d'Hibernate

```
◆ <hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.OracleDialect</property>
    <property name="hibernate.connection.driver_class">oracle.jdbc.OracleDriver</property>
    <property name="hibernate.connection.url">jdbc:oracle:thin:@scinfo099:1521:etud10</property>
    <property name="hibernate.connection.username">ecariou</property>
    <property name="hibernate.connection.password">toto</property>
    <property name="hibernate.show_sql">>true</property>
    <mapping class="data.Sport" file="" jar="" package="" resource="data/Sport.hbm.xml"/>
    <mapping class="data.Discipline" file="" jar="" package="" resource="data/Discipline.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

◆ On y trouve, notamment

- ◆ Les paramètres de connexion à la base Oracle (avec le type de driver à utiliser)
 - ◆ On peut voir que l'accès en pratique se fera via JDBC
- ◆ L'association d'un POJO avec son fichier de mapping
 - ◆ Ex: data.Sport (classe Java) dont le mapping est précisé dans Sport.hbm.xml

Hibernate : ex. de fichier de mapping

◆ Fichier Sport.hbm.xml

```
◆ <hibernate-mapping>
  <class name="data.Sport" table="SPORT" schema="ECARIOU">
    <id name="codeSport" type="short">
      <column name="CODE_SPORT" precision="4" scale="0" />
      <generator class="assigned" />
    </id>
    <property name="intitule" type="string">
      <column name="INTITULE" length="20" />
    </property>
    <set name="disciplines" inverse="true">
      <key>
        <column name="CODE_SPORT" precision="4" scale="0" />
      </key>
      <one-to-many class="data.Discipline" />
    </set>
  </class>
</hibernate-mapping>
```

◆ Dans classe data.Sport, attribut codeSport correspond à la colonne CODE_SPORT de la table SPORT

◆ Définition d'un ensemble nommé disciplines dans la classe data.Sport

◆ Jointure avec la table discipline : toutes les disciplines de ce sport

Hibernate : ex. de POJO

- ◆ POJO d'un sport, généré à partir du fichier de mapping

- ◆ `public class Sport implements java.io.Serializable {`

```
private short codeSport;  
private String intitule;  
private Set disciplines = new HashSet(0);
```

```
public Sport() {  
}
```

```
public Sport(short codeSport) {  
    this.codeSport = codeSport;  
}
```

```
public Sport(short codeSport, String intitule,  
             Set disciplines) {  
    this.codeSport = codeSport;  
    this.intitule = intitule;  
    this.disciplines = disciplines;  
}
```

```
public short getCodeSport() {  
    return this.codeSport;  
}
```

```
public void setCodeSport(short codeSport) {  
    this.codeSport = codeSport;  
}
```

```
public String getIntitule() {  
    return this.intitule;  
}
```

```
public void setIntitule(String intitule) {  
    this.intitule = intitule;  
}
```

```
public Set getDisciplines() {  
    return this.disciplines;  
}
```

```
public void setDisciplines(Set disciplines) {  
    this.disciplines = disciplines;  
}
```

- ◆ Définition des attributs, avec getter et setter associés

- ◆ Définition des constructeurs (avec un constructeur sans paramètre)

- ◆ Redéfinir méthodes `equals()` et `hashCode()` pour gestion dans collections

Hibernate : ex. accès aux données

- ◆ Affichage de la liste des sports, avec pour chacun la liste de ses disciplines

- ◆ // création session Hibernate (et donc connexion au serveur Oracle)
Session session = (new
 Configuration().configure().buildSessionFactory()).openSession();

```
// requête en HQL : récupère la liste de tous les sports,  
// sous forme d'une liste Java
```

```
Query query = session.createQuery("from data.Sport");  
List sports = query.list();
```

```
// variables utilisées pour le parcours de la liste des sports
```

```
Set disciplines;
```

```
Iterator itSport, itDisc;
```

```
Sport sport;
```

```
Discipline disc;
```

Hibernate : ex. accès aux données

- ◆ Affichage de la liste des sports, avec pour chacun la liste des disciplines (suite)

```
◆ // parcourt les sports
itSport = sports.iterator();
while (itSport.hasNext()) {
    sport = (Sport) itSport.next();
    System.out.println(sport.getCodeSport() + " -> " + sport.getIntitule());

    // récupère l'ensemble des disciplines du sport
    disciplines = sport.getDisciplines();
    itDisc = disciplines.iterator();

    // parcourt les disciplines
    while (itDisc.hasNext()) {
        disc = (Discipline) itDisc.next();
        System.out.println("    "+disc.getCodeDiscipline() +
                           " -> "+disc.getIntitule());
    }
}
```

Hibernate : ex. accès aux données

◆ Affichage résultant de l'exécution

- ◆ 1 -> athlétisme
 - 1 -> 100 metres
 - 2 -> 200 metres
 - 7 -> marathon
 - 4 -> saut en longueur
 - 3 -> saut en hauteur
- 2 -> ski
- 3 -> natation
 - 6 -> 100m papillon
 - 5 -> 100m 4 nages

◆ Point intéressant

- ◆ A partir d'un objet de classe Sport, on accède directement à sa liste de disciplines via `getDisciplines()` : ensemble d'objets
 - ◆ Les objets sont chargés à partir de la base directement sans besoin de traitement particulier, c'est transparent
 - ◆ Si on était passé par du SQL via JDBC, on aurait du faire une requête explicite du type
 - ◆ `SELECT * FROM DISCIPLINE WHERE CODE_SPORT=XX`

Hibernate : ex. accès aux données

- ◆ A l'exécution, peut demander à tracer les requêtes SQL effectuées, ce qui donne ici
 - ◆ *Hibernate: select sporto_.CODE_SPORT as CODE1_o_, sporto_.INTITULE as INTITULEo_ from ECARIOU.SPORT sporto_*
1 -> athletisme
 - ◆ *Hibernate: select disciplineo_.CODE_SPORT as CODE2_1_, disciplineo_.CODE_DISCIPLINE as CODE1_1_, disciplineo_.CODE_DISCIPLINE as CODE1_1_o_, disciplineo_.CODE_SPORT as CODE2_1_o_, disciplineo_.INTITULE as INTITULE1_o_ from ECARIOU.DISCIPLINE disciplineo_ where disciplineo_.CODE_SPORT=?*
1 -> 100 metres
2 -> 200 metres
7 -> marathon
4 -> saut en longueur
3 -> saut en hauteur
(...)
- ◆ Premier SELECT : pour récupérer la liste des sports
- ◆ Deuxième SELECT : pour récupérer la liste des disciplines du sport courant

Hibernate : ex. accès aux données

◆ Autre exemple : ajout d'une discipline au sport natation

◆ try {

```
// ouvre la session et une transaction
```

```
session = (new Configuration().configure().buildSessionFactory()).openSession();
```

```
trans = session.beginTransaction();
```

```
// récupère l'instance de sport correspondant à natation (requête en HQL)
```

```
query = session.createQuery("select sport from Sport sport where sport.intitule='natation'");
```

```
// ajout du relais 4 x 100 dans le sport natation en rendant l'objet persistant
```

```
disc = new Discipline((short)8, sport, "relais 4 x 100");
```

```
session.persist(disc);
```

```
// commit pour valider physiquement l'ajout
```

```
trans.commit();
```

```
} catch (Exception e) {
```

```
System.err.println("Erreur Hibernate : " + e);
```

```
// problème : on annule la transaction
```

```
trans.rollback();
```

```
}
```

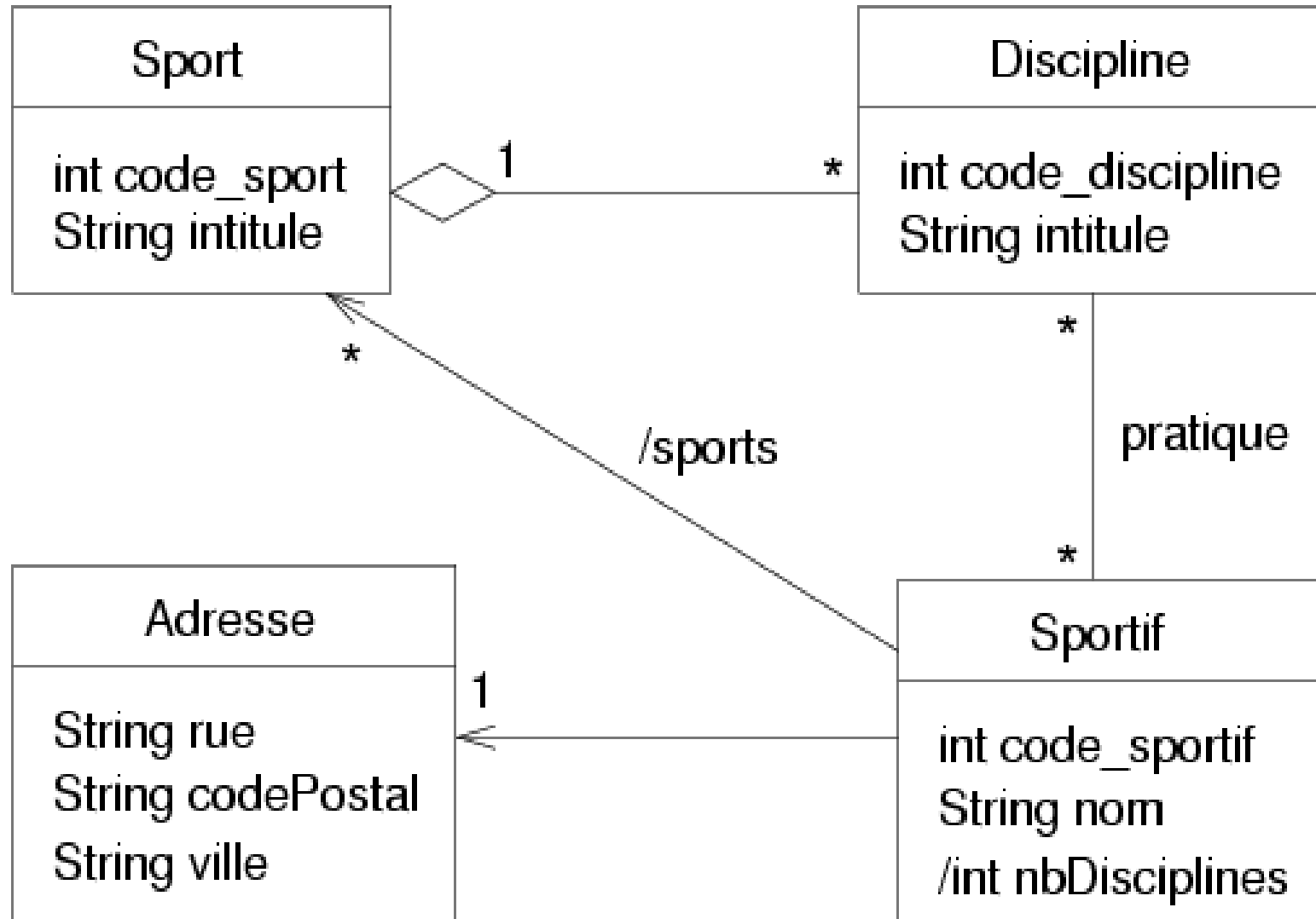
```
session.close();
```

Hibernate : ex. accès aux données

- ◆ Notes sur code d'ajout
 - ◆ On crée l'objet discipline disc avec l'objet sport en paramètre
 - ◆ On passe une référence d'objet, on ne donne pas une clé identifiant un sport comme on le ferait avec SQL
 - ◆ On est bien dans de la manipulation « standard » d'objets
 - ◆ Le persist() sur cet objet permet directement, sans traitement supplémentaire, de le rendre persistant (voir plus loin pour détails)
 - ◆ C'est-à-dire l'enregistrer dans la BDD
- ◆ Si on trace les requêtes SQL, on obtient
 - ◆ `Hibernate: select sporto_.CODE_SPORT as CODE1_o_, sporto_.INTITULE as INTITULEo_ from ECARIOU.SPORT sporto_ where sporto_.INTITULE='natation'`
 - ◆ `Hibernate: insert into ECARIOU.DISCIPLINE (CODE_SPORT, INTITULE, CODE_DISCIPLINE) values (?, ?, ?)`
 - ◆ La première requête est un SELECT qui récupère les données du sport « natation »
 - ◆ La deuxième requête est un INSERT qui ajoute la discipline 103 qu'on vient de créer

Exemple plus détaillé

- ◆ Modélisation métier de l'application de sports avec ajout d'une classe / entité sportif



Exemple plus détaillé

- ◆ Classes Sport et Disciplines
 - ◆ Reprend les mêmes contenus qu'on vient de voir
 - ◆ Une discipline est associée à un sport
- ◆ Ajoute une entité Sportif
 - ◆ Un sportif pratique plusieurs disciplines et une discipline est pratiquée par plusieurs sportifs
 - ◆ Association bidirectionnelle * <-> * pratique entre les 2 classes
 - ◆ Attribut nbDisciplines est dérivé
 - ◆ Le nombre de disciplines se détermine à partir de l'association pratique
 - ◆ Ex. en OCL :
context Sportif def: nbDisciplines : int = pratique -> size()
 - ◆ Un sportif a un nom et une adresse
 - ◆ Association unidirectionnelle dérivée /sports entre Sportif et Sport
 - ◆ Les sports pratiqués par le sportif
 - ◆ Se détermine de l'association pratique en récupérant les sports de ces disciplines : pratique.sport -> asSet()

Exemple plus détaillé

- ◆ Tables associées dans le SGBD
 - ◆ Conserve les tables sport et discipline déjà vues
 - ◆ sport (*code_sport*, intitule)
 - ◆ discipline (*code_discipline*, intitule, *code_sport*)
 - ◆ *code_sport* : clé étrangère venant de sport
 - ◆ Données sur un sportif
 - ◆ Une adresse est spécifique à un sportif : peut inclure les données de l'adresse avec les données du sportif
 - ◆ sportif(*code_sportif*, nom, rue, *code_postal*, ville)
 - ◆ Association entre disciplines et sportifs : nécessite une table d'association
 - ◆ pratique(*code_sportif*, *code_discipline*)
 - ◆ Clé primaire : combinaison de *code_sportif* et *code_discipline*
 - ◆ Clés étrangères : *code_sportif* de sportif et *code_discipline* de discipline

Exemple plus détaillé

- ◆ Point fondamental lors de l'utilisation d'un framework de persistance
 - ◆ Pas de nécessité/peu recommandé d'avoir un mapping 1 vers 1 entre une table de la BDD et une classe coté métier
 - ◆ On modélise d'un coté le métier et de l'autre la BDD en s'attachant à faire les meilleures modélisations selon le domaine
 - ◆ Coté métier : structure logique et facilité de manipulation des données
 - ◆ Coté BDD : optimiser la sauvegarde des données et performance d'accès
 - ◆ On définira ensuite les mappings adéquats
- ◆ Dans notre exemple
 - ◆ Les données de la classe Adresse et de la classe Sportif sont stockées via une seule table et non pas deux tables différentes
 - ◆ La table d'association pratique n'a pas à être représentée par une classe
 - ◆ On aura directement des méthodes dans les classes Sportif et Discipline pour récupérer les éléments correspondants
 - ◆ Le fichier de mapping permettra de directement gérer cette association
 - ◆ L'attribut nbDisciplines est utile/pertinent seulement coté métier

Mapping pour la classe Sportif

◆ Contenu de la classe Sportif : POJO

◆ `public class Sportif implements java.io.Serializable {`

```
private int codeSportif;  
private String nom;  
private Set disciplines = new HashSet(o);  
private int nbDisciplines;  
private Adresse adresse;
```

```
public Sportif() {  
}  
(...)
```

◆ Définition des attributs

- ◆ Avec disciplines qui est une association à cardinalité multiple donc utilise un Set
- ◆ Chaque attribut doit posséder un getter et un setter
- ◆ Obligation d'un constructeur sans paramètre
- ◆ Peut en définir d'autres en plus

Mapping pour la classe Sportif

- ◆ Fichier de mapping pour la classe Sportif
 - ◆ `<hibernate-mapping>`
 - ◆ `<class name="data.Sportif" table="sportif" catalog="sports">`
 - ◆ La classe persistante `data.Sportif` se base sur le contenu de la table `sportif`
 - ◆ `<id name="codeSportif" type="int">`
 - ◆ `<column name="code_sportif" />`
 - ◆ `<generator class="assigned" />`
 - ◆ `</id>`
 - ◆ Définition d'un attribut `codeSportif` correspondant à la colonne `code_sportif` de la table
 - ◆ La balise `<id>` précise qu'on est en train de définir un attribut jouant le rôle particulier d'identification de l'objet
 - ◆ Balise non obligatoire mais fortement recommandée
 - ◆ Par principe, elle sera en général liée à la clé primaire de la table
 - ◆ La balise `<generator>` précise comment l'identifiant est généré
 - ◆ `assigned` : géré à la main par celui qui crée les classes
 - ◆ Peut utiliser d'autres valeurs pour une génération automatique (voir plus loin)

Mapping pour la classe Sportif

- ◆ `<property name="nom" type="string">`
 `<column name="nom" length="20" />`
`</property>`
- ◆ Balise `<property>` définit une propriété au sens des Java Beans, concrètement, un attribut simple (autre qu'une collection, tableau ...) de la classe
 - ◆ Ici l'attribut `nom` mappe la colonne `nom` de la table
- ◆ `<property name="nbDisciplines" type="integer"`
`formula = "(select count(*) from PRATIQUE p where`
`code_sportif = p.code_sportif)" />`
- ◆ Champ `formula` (ou balise `<formula>`) permet de définir une formule (expression SQL) au lieu d'un mapping vers une colonne d'une table
 - ◆ Ici on compte le nombre de disciplines associées au sportif courant via une requête sur la table `pratique`
 - ◆ L'appel du setter associé (`setNbDisciplines`) ne servira à rien ...

Mapping pour la classe Sportif

- ◆

```
<component name="adresse" class="data.Adresse">  
  <property name="rue" type="string">  
    <column name="rue" length="50" />  
  </property>  
  <property name="codePostal" type="string">  
    <column name="code_postal" length="5" />  
  </property>  
  <property name="ville" type="string">  
    <column name="ville" length="15" />  
  </property>  
</component>
```
- ◆ Balise `<component>` définit un élément composé (relation UML de composition) dans la classe courante
- ◆ Ici on définit un attribut adresse de classe Adresse qui contient les attributs rue, codePostal et ville mappant les colonnes équivalentes dans la table sportif (la table de la classe engloblante)
- ◆ La classe Adresse est définie comme un POJO car sera persitante
- ◆

```
public class Adresse implements java.io.Serializable {  
  private String rue;  
  private String codePostal;  
  private String ville;
```

Mapping pour la classe Sportif

- ◆

```
<set name="disciplines" inverse="false" table="pratique">  
  <key>  
    <column name="code_sportif" not-null="true" />  
  </key>  
  <many-to-many entity-name="data.Discipline">  
    <column name="code_discipline" not-null="true" />  
  </many-to-many>  
</set>
```
- ◆ Balise `<set>` définit un ensemble de données
- ◆ Correspond ici au mapping d'une association avec une cardinalité multiple sur une autre table (ici pratique)
- ◆ Balise `<key>` : clé étrangère sur la table avec laquelle on fait la jointure
- ◆ Sur la table pratique, la clé étrangère est `code_sportif`
- ◆ Balise `<many-to-many>` définit une association en `* <-> *`
- ◆ On fait la jointure sur la colonne `code_discipline` de pratique
- ◆ Retournera des instances de la classe `Discipline`
- ◆ L'attribut `disciplines` de la classe `Sportif` correspond donc à l'ensemble des disciplines trouvées dans pratique avec le même `code_sportif` que le sportif courant

Association bidirectionnelle

- ◆ Dans la classe Discipline, on trouvera l'association inverse, de discipline vers sportif
- ◆ Dans la classe Discipline, attribut
 - ◆ private Set sportifs;
- ◆ Dans le mapping de la classe Discipline
 - ◆

```
<set name="sportifs" inverse="true" table="pratique">  
  <key>  
    <column name="code_discipline" not-null="true" />  
  </key>  
  <many-to-many entity-name="data.Sportif">  
    <column name="code_sportif" not-null="true" />  
  </many-to-many>  
</set>
```
 - ◆ Ensemble de sportifs correspondant à la jointure via la table pratique et la clé code_discipline

Association bidirectionnelle

- ◆ Contrainte pour un mapping d'association bidirectionnelle
 - ◆ D'un des deux cotés, on doit avoir un `inverse = true` dans la définition du set (et un `false`, par défaut, de l'autre)
 - ◆ Ici c'est dans le mapping pour la classe `Discipline`
- ◆ Dans les classes, pour assurer la bidirectionnalité, on doit faire un ajout dans les deux ensembles des deux éléments
 - ◆ Classe `Sportif`
 - ◆

```
public void addDiscipline(Discipline disc) {  
    this.disciplines.add(disc);  
    disc.getSportifs().add(this)  
}
```
 - ◆ Classe `Discipline`
 - ◆

```
public void addSportif(Sportif sportif) {  
    this.sportifs.add(sportif);  
    sportif.getDisciplines().add(this);  
}
```

Association et cardinalités

- ◆ Quatre type d'associations, par jointure en précisant la clé étrangère
 - ◆ <one-to-one>
 - ◆ Un vers un
 - ◆ <one-to-many>
 - ◆ Un vers plusieurs : sport vers discipline
 - ◆ <many-to-one>
 - ◆ Plusieurs vers un : discipline vers sport
 - ◆ <many-to-many>
 - ◆ Plusieurs vers plusieurs : entre discipline et sportif
 - ◆ Nécessite une table d'association entre les 2 tables

Mapping pour la classe Sportif

- ◆ Reste à définir l'ensemble (dérivé) des Sports
 - ◆ Consiste à récupérer les sports des disciplines du sportif courant
 - ◆ En SQL, jointure sur les 4 tables pratique, discipline, sportif et sport, en fonction de la valeur *code_monSportif*
 - ◆ **select** distinct(sport.code_sport), sport.intitule
from discipline, pratique, sportif, sport
where sportif.code_sportif = *code_monSportif* **and**
discipline.code_discipline = pratique.code_discipline **and**
pratique.code_sportif = sportif.code_sportif **and**
sport.code_sport = discipline.code_sport
 - ◆ Difficile à exprimer par un mapping Hibernate, on va passer par une requête
 - ◆ Rajoute dans la classe Sportif la méthode suivante
 - ◆ `public Set<Sport> getSports() { ... }`
 - ◆ Pas besoin d'un attribut sports car il ne sera pas persistant

Requête *getSports()*

- ◆ Première (mauvaise) implémentation
 - ◆ Utilisation de Java au maximum
 - ◆

```
public Set<Sport> getSports() {  
    Set<Sport> sports = new HashSet<Sport>();  
    Iterator it = this.getDisciplines().iterator();  
    while (it.hasNext())  
        sports.add(((Discipline)it.next()).getSport());  
    return sports;  
}
```
 - ◆ Parcourt les disciplines du sportif et récupère pour chacun le sport associé
 - ◆ On place ces sports dans un Set (qui par principe ne contiendra pas de doublon)
 - ◆ Très mauvaise idée !
 - ◆ On oblige en effet à charger en mémoire (de la BDD vers le programme Java) toutes les disciplines du sportif alors qu'on n'en a pas besoin
 - ◆ Problème potentiel de performance

Requête `getSports()`

◆ Bonne implémentation

- ◆ Passer par une requête en HQL (Hibernate Query Language)
- ◆ Equivalente à celle qu'on aurait faite en SQL ... mais en plus simple

◆ `public Set<Sport> getSports() {`

```
    Set<Sport> sports = new HashSet<Sport>();
```

```
    // création de la requête
```

```
    Query query = session.createQuery(
```

```
        "select distinct(sport) from data.Sport sport, data.Discipline disc"+
```

```
        "where disc.sport = sport and :leSportif in elements(disc.sportifs)");
```

```
    // le sportif est l'objet courant
```

```
    query.setParameter("leSportif", this);
```

```
    // construit le set à retourner à partir de la liste résultat de la requête
```

```
    Iterator it = query.list().iterator();
```

```
    while (it.hasNext())
```

```
        sports.add(it.next());
```

```
    return sports;
```

```
}
```

- ◆ Note : ne pas oublier de gérer en plus les exceptions

Langage de requête HQL

- ◆ Requête HQL précédente
 - ◆ Se lit assez simplement : récupérer la liste des sports des disciplines qui sont associées au sportif passé en paramètre
 - ◆ Utilisation d'un paramètre nommé pour préciser le sportif
 - ◆ `query.setParameter("nomParam", valeur);`
 - ◆ Avec la valeur pouvant directement être un objet
 - ◆ Comme ici : c'est une instance de la classe `Sportif`
 - ◆ Exécution de la requête
 - ◆ Appel de `list()` sur l'objet requête qui retourne la liste des résultats
 - ◆ On peut ensuite associer un itérateur à cette liste
 - ◆ Attention !
 - ◆ Il existe une méthode `iterate()` appellable directement sur l'objet requête
 - ◆ `Iterator it = query.iterate();`
 - ◆ Cette méthode ne charge pas en mémoire les objets persistants retournés par la requête
 - ◆ S'ils étaient déjà chargés, ça marche, sinon on ne récupère rien
 - ◆ Donc privilégier l'usage de : `Iterator it = query.list().iterate();`

Langage de requête HQL

- ◆ HQL est un langage de requête aussi puissant que SQL et qui est en plus orienté objet
 - ◆ Permet de manipuler directement des objets (instances de POJO)
 - ◆ Définir une « variable » sport correspondant à une instance de la classe data.Sport : ... data.Sport sport ...
 - ◆ Permet de naviguer sur les associations entre objets
 - ◆ Récupérer l'objet sport associé à une discipline : ... disc.sport ...
 - ◆ Mêmes fonctionnalités que SQL, gestion du polymorphisme...
- ◆ Au final, les requêtes HQL sont généralement (bien) plus simples que les requêtes SQL équivalentes
 - ◆ Trace de la requête SQL exécutée
 - ◆ **select** distinct sporto_.code_sport as code1_7_, sporto_.intitule as intitule7_ **from** sports.sport sporto_, sports.discipline discipline1_ **where** discipline1_.code_sport=sporto_.code_sport **and** (? **in** (**select** sportifs2_.code_sportif **from** pratique sportifs2_ **where** discipline1_.code_discipline=sportifs2_.code_discipline))
 - ◆ Deux requêtes SELECT imbriquées sur 4 tables

Langage de requête HQL

- ◆ Si l'on sait qu'une requête ne renvoie qu'un seul résultat
 - ◆ Peut utiliser `uniqueResult()` au lieu de `list()` pour récupérer le résultat
 - ◆

```
query = session.createQuery("select sport from data.Sport sport where  
sport.codeSport = '2' ");  
Sport sport = (Sport)query.uniqueResult();  
if (sport == null) System.out.println("Pas de sport 2");  
else System.out.println("Le sport 2 est : "+sport.getIntitule());
```
- ◆ Si une requête sélectionne plus d'une valeur, on récupère un ensemble de tableaux, un tableau par résultat
 - ◆

```
query = session.createQuery("select sportif.nom, sportif.adresse.ville  
from data.Sportif sportif");  
Iterator it = query.list().iterator();  
System.out.println("Nom des personnes avec leur ville");  
while(it.hasNext()) {  
    Object[] res = (Object[])it.next();  
    System.out.println("nom : " + res[0] + ", ville : " + res[1]);  
}
```

Cycle de vie d'un objet persistant

- ◆ Toutes les fonctions de gestion du cycle de vie d'un objet se font par rapport à une session Hibernate ouverte
- ◆ Toute modification d'état se fait en mode transactionnel
 - ◆ Rendre persistant un objet, modifier le contenu d'un objet persistant, supprimer un objet persistant, ...
 - ◆ Schéma général à appliquer de préférence
 - ◆ En cas d'erreur (exception levée) lors de la transaction, exécuter systématiquement un rollback pour annuler ce qui a pu être modifié
 - ◆

```
session = (new Configuration().configure().buildSessionFactory()).openSession();  
(...)  
try {  
    trans = session.beginTransaction();  
    // faire les modifications voulues  
    (...)  
    // pas d'erreur, on commit  
    trans.commit();  
} catch (Exception e) {  
    // problème : on annule tout  
    if (trans != null) trans.rollback();  
}
```

Cycle de vie d'un objet persistant

- ◆ Plusieurs méthodes pour charger un objet persistant à partir de la BDD
 - ◆ Exécuter des requêtes HQL comme on l'a vu
 - ◆ Utiliser des requêtes natives SQL
 - ◆ `SQLQuery query = session.createQuery(...);`
 - ◆ Récupérer une connexion JDBC puis faire des requêtes SQL
 - ◆ `Connection conn = session.connection();`
 - ◆ Utiliser l'API Criteria
 - ◆ Si on connaît l'identifiant d'un objet : `load()` ou `get()`
 - ◆ Charger le sportif d'identifiant égal à 2
 - ◆ `Sportif sportif = session.load(Sportif.class, 2);`
 - ◆ Différence entre `load()` et `get()`
 - ◆ Si ne trouve pas l'objet (identifiant inexistant), `load()` lève une exception et `get()` renvoie null
 - ◆ Contrairement à `get()`, `load()` ne charge pas forcément tout de suite l'objet en mémoire, l'exception potentielle sera donc levée seulement quand on accèdera aux données de l'objet

Cycle de vie d'un objet persistant

- ◆ Créer un objet et le rendre persistant
 - ◆ Instantier de manière normale un POJO et positionner la valeur de ses attributs
 - ◆ Si l'on ne veut pas gérer soi même la clé primaire associé au POJO dans la BDD
 - ◆ Dans fichier de mapping, balise <id>, balise <generator>, champ class peut prendre différentes valeurs
 - ◆ assigned : à la charge de celui qui instancie l'objet de fixer la clé primaire (valeur par défaut si aucun générateur n'est précisé)
 - ◆ native : laisse le SGBD déterminer la clé primaire (valeur entière seulement) selon ses fonctionnalités
 - ◆ uuid : identifiant sous forme de chaine, en utilisant l'adresse IP de la machine
 - ◆
 - ◆ Ensuite, on rend l'objet persistant via l'appel de `persist()` ou `save()` sur la transaction courante
 - ◆

```
Sport sport = new Sport();  
sport.setIntitule("pêche");  
session.persist(sport);
```

Cycle de vie d'un objet persistant

- ◆ Créer un objet et le rendre persistant (suite)
 - ◆ Différence entre `persist()` et `save()`
 - ◆ `save()` réalise immédiatement la génération de la clé mais l'enregistrement physique se fera plus tard
 - ◆ `persist()` fait les deux actions simultanément et immédiatement
 - ◆ Attention aux dépendances entre objets créés
 - ◆

```
Transaction trans = trans.beginTransaction();  
Sport sport = new Sport("pêche");  
Discipline disc = new Discipline("mouche", sport);  
session.persist(disc);  
trans.commit();
```
 - ◆ Ce code génèrera une exception car le sport associé à la discipline n'a pas été rendu persistant
 - ◆ Il faut donc le faire explicitement
 - ◆ Ou préciser dans les fichiers de mapping que la persistance se fait automatiquement en cascade
 - ◆ Dans une définition d'association (<many-to-one>, ...) ajouter le champ : `cascade = "persist"`

Cycle de vie d'un objet persistant

- ◆ Détacher / rattacher un objet
 - ◆ Tout objet persistant (chargé depuis la BDD, instantié puis rendu persistant) est attaché à la session
 - ◆ L'état de l'objet est lié avec le contenu de la BDD tant que la session existe
 - ◆ Dans certains cas, on peut vouloir détacher un objet
 - ◆ Méthodes `evict()` ou `close()`
 - ◆ Il n'est plus associé à la session mais existe encore en BDD
 - ◆ Ses modifications ne sont plus répercutées sur la BDD
 - ◆ Peut ensuite rattacher l'objet à la session et répercuter ses modifs
 - ◆ Méthodes `merge()`, `lock()`, `update()`, `saveOrUpdate()`
- ◆ Supprimer un objet persistant
 - ◆ Méthode `delete()` : `session.delete(sport);`
 - ◆ Le contenu de l'objet est supprimé de la BDD (mais l'objet Java existe toujours lui, il n'a juste plus aucun lien avec la BDD)
 - ◆ Attention aux dépendances entre objets lors de la suppression
 - ◆ Dans fichier de mapping, utiliser au besoin des champs :
`cascade="delete"` et `cascade="all-delete-orphan"`

Cycle de vie d'un objet persistant

- ◆ Modifier un objet persistant
 - ◆ Modifier son contenu via l'appel des setters et autres accès aux collections
 - ◆ `sportif.getAdresse().setRue("rue Emile Guichenné");`
`sportif.getAdresse().setCodePostal("64000");`
`sportif.getAdresse().setVille("Pau");`
`sportif.addDiscipline(disc);`
 - ◆ La détection de la modification et la mise à jour sur la BDD sont faites automatiquement par Hibernate
 - ◆ En pratique, la méthode `flush(obj)` est appelée pour faire la mise à jour physique
- ◆ Recharger le contenu d'un objet persistant
 - ◆ Méthode `refresh(obj)`
 - ◆ Il n'y a pas de raison d'appeler explicitement cette méthode vu que les modifications sont transmises, sauf si on sait qu'un trigger a été exécuté

Autres fonctionnalités de mapping

◆ Collections

- ◆ On a défini des mappings correspondant à des jointures retournant une collection java de type Set via une balise <set>
- ◆ On peut définir d'autres types de collections, correspondant aux types Java équivalents
 - ◆ bag, list, map, array, collections indexées ...

◆ Balise <join>

- ◆ Permet de mapper des données venant de plusieurs tables dans une seule classe côté Java

◆ Héritage

- ◆ Mappings pour une hiérarchie d'héritage
- ◆ Trois stratégies possibles
 - ◆ Une table unique pour toutes les classes
 - ◆ Une table par classe logique (mapping pour la classe mère + mappings pour les classes filles)
 - ◆ Une table par classe « concrète » (pas de mapping pour la classe mère)

Concurrence & performances

◆ Performance

- ◆ Les performances sont principalement liées au chargement en mémoire des données
- ◆ Comme le graphe d'objets liés est potentiellement grand, on essaye de charger les objets uniquement à la demande
 - ◆ Champ lazy="true" dans les mappings des collections
 - ◆ Par défaut, quand Hibernate charge un objet, il ne charge pas ses collections associées tant qu'on y accède pas
- ◆ Exemple avec code précédent, et sa trace, accédant aux sports et leurs disciplines
 - ◆ Un premier SELECT récupère tous les sports
 - ◆ Pour chaque sport, un autre SELECT est exécuté pour récupérer la liste des disciplines quand on y accède (au moment du sport.getDisciplines())

◆ Gestion de la concurrence

- ◆ Mêmes fonctionnalités que dans les SGBD pour gérer de manière précise les accès concurrents : verrous sur données, ...
- ◆ La gestion détaillée des transactions est importante

Mapping par annotation

- ◆ Nos mappings se basent sur des couples de fichiers
 - ◆ La classe Java (le POJO) et son fichier XML de mapping
- ◆ Peut aussi directement ajouter des annotations dans la classe Java pour préciser les mappings

- ◆ `@Entity`
`@Table(name = "sport")`
`@NamedQueries({@NamedQuery(name = "Sport.findAll", query = "SELECT s FROM Sport s"),`
`@NamedQuery(name = "Sport.findByCodeSport", query = "SELECT s FROM Sport s WHERE`
`s.codeSport = :codeSport"), @NamedQuery(name = "Sport.findByIntitule", query = "SELECT s`
`FROM Sport s WHERE s.intitule = :intitule")})`

```
public class Sport implements Serializable {  
    private static final long serialVersionUID = 1L;
```

```
    @Id  
    @Basic(optional = false)  
    @Column(name = "code_sport")  
    private Integer codeSport;
```

```
    @Column(name = "intitule")  
    private String intitule;
```

```
    @OneToMany(mappedBy = "codeSport")  
    private Collection<Discipline> disciplines;  
    (...)
```

Conclusion sur Hibernate

- ◆ Framework de persistance très puissant (on n'en a vu qu'une introduction détaillée)
 - ◆ Indépendance des supports physiques
 - ◆ Gestion des caractéristiques objet (héritage ...)
 - ◆ Politiques de chargements des objets (à la demande ...)
 - ◆ Langage de requête HQL
 - ◆ Un SQL-like orienté objet
 - ◆ Permet vraiment de se consacrer sur la logique métier
 - ◆ La gestion des données est principalement du Java natif, pas à soucier de comment sont gérées et accédées les données
 - ◆ Au niveau du programme en tout cas, car il faut quand même concevoir la base de donnée et les mappings
 - ◆ Coté Java, on manipule donc directement les entités métiers telles qu'on les a conçues
 - ◆ Même si on doit avoir en tête qu'elles ont une persistance en BDD

Documentation Hibernate

- ◆ Site officiel d'Hibernate
 - ◆ <http://www.hibernate.org>
 - ◆ Avec beaucoup de documentation
- ◆ Traductions françaises du manuel de référence du site d'Hibernate
 - ◆ <http://www.dil.univ-mrs.fr/~massat/docs/hibernate-3.1/reference/fr/html/index.html>
 - ◆ <http://docs.jboss.org/hibernate/core/3.3/reference/fr/html/index.html>
- ◆ Livre
 - ◆ « Hibernate 3.0 – Gestion optimale de la persistance dans les applications Java/J2EE », Anthony Patricio, Eyrolles

Exemple final

- ◆ Combinaison de Hibernate, servlets, pages JSP avec UEL et JSTL pour afficher les sportifs avec leurs disciplines et sports pratiqués
- ◆ Page HTML via un lien envoie une requête à la servlet SportServlet
- ◆ Servlet SportServlet.java (d'URL « /Sports »)
 - ◆ Récupère via une requête HQL l'ensemble des sports
 - ◆ Ajoute cet ensemble à la requête HTTP reçue par la servlet
 - ◆ Puis fait suivre (forward) la requête HTTP à la page afficherSports.jsp
- ◆ Page afficherSports.jsp
 - ◆ Récupère l'ensemble des sports dans la requête HTTP
 - ◆ Via UEL et JSTL, met en page leur affichage
- ◆ La servlet ne génère aucun code HTML, c'est le rôle de la page JSP à qui on a forwardé la requête

Exemple final

◆ Lien dans la page HTML

- ◆ `Afficher la liste des sportifs`

◆ Code de SportServlet.java

- ◆

```
public List<Sportif> getListeSportifs() throws Exception {
    Session session = this.openHibernateSession();
    Query query = session.createQuery("from data.Sportif");
    return query.list();
}
```

```
protected void processRequest( ... request, ... response) {
    String operation = request.getParameter("operation");
    if (operation.equals("listeSportif")) {
        // récupère la liste des sports et l'associe à la requête HTTP
        request.setAttribute("sportifs", getListeSportifs());
        // forwarde la requête à la page JSP
        getServletConfig().getServletContext().getRequestDispatcher(
            "/afficheSportifs.jsp").forward(request,response);
    } (...)
```

Exemple final

◆ Page JSP afficheSportifs.jsp

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

`<h2>Liste des sportifs et de ce qu'ils font</h2>`

`<c:forEach items="${requestScope.sportifs}" var="sp">`

`<h3>${sp.nom}</h3>`

`<p>Adresse : ${sp.adresse.rue} - ${sp.adresse.codePostal} ${sp.adresse.ville}</p>`

`<c:choose>`

`<c:when test="${empty sp.disciplines}">`

`<p><i>Ne pratique aucune discipline sportive</i></p>`

`</c:when>`

`<c:otherwise>`

`<p>Liste des disciplines pratiquées : </p>`

``

`<c:forEach items="${sp.disciplines}" var="disc">`

`${disc.intitule}`

`</c:forEach>`

``

`<p>Liste des sports : </p>`

``

`<c:forEach items="${sp.sports}" var="sport">`

`${sport.intitule}`

`</c:forEach>`

``

`</c:otherwise>`

`</c:choose>`

`</c:forEach>`

Exemple final

The screenshot shows a Mozilla Firefox browser window with the title 'JSP Page - Mozilla Firefox'. The address bar contains the URL 'http://localhost:8084/HibernateSports/Sports?operation=listeSporti'. The page content is as follows:

Liste des sportifs et de ce qu'ils font

Roger
Adresse : Rue de la soif - 35000 Rennes
Liste des disciplines pratiquées :

- super geant
- descente
- 200 metres
- 100 metres
- contre la montre

Liste des sports :

- ski
- athletisme
- cyclisme

Simone
Adresse : Impasse du cimetiere - 64000 Pau
Liste des disciplines pratiquées :

- 100 metres
- contre la montre

Liste des sports :

- cyclisme
- athletisme

Gerard
Terminé