

# Formation Java avancé

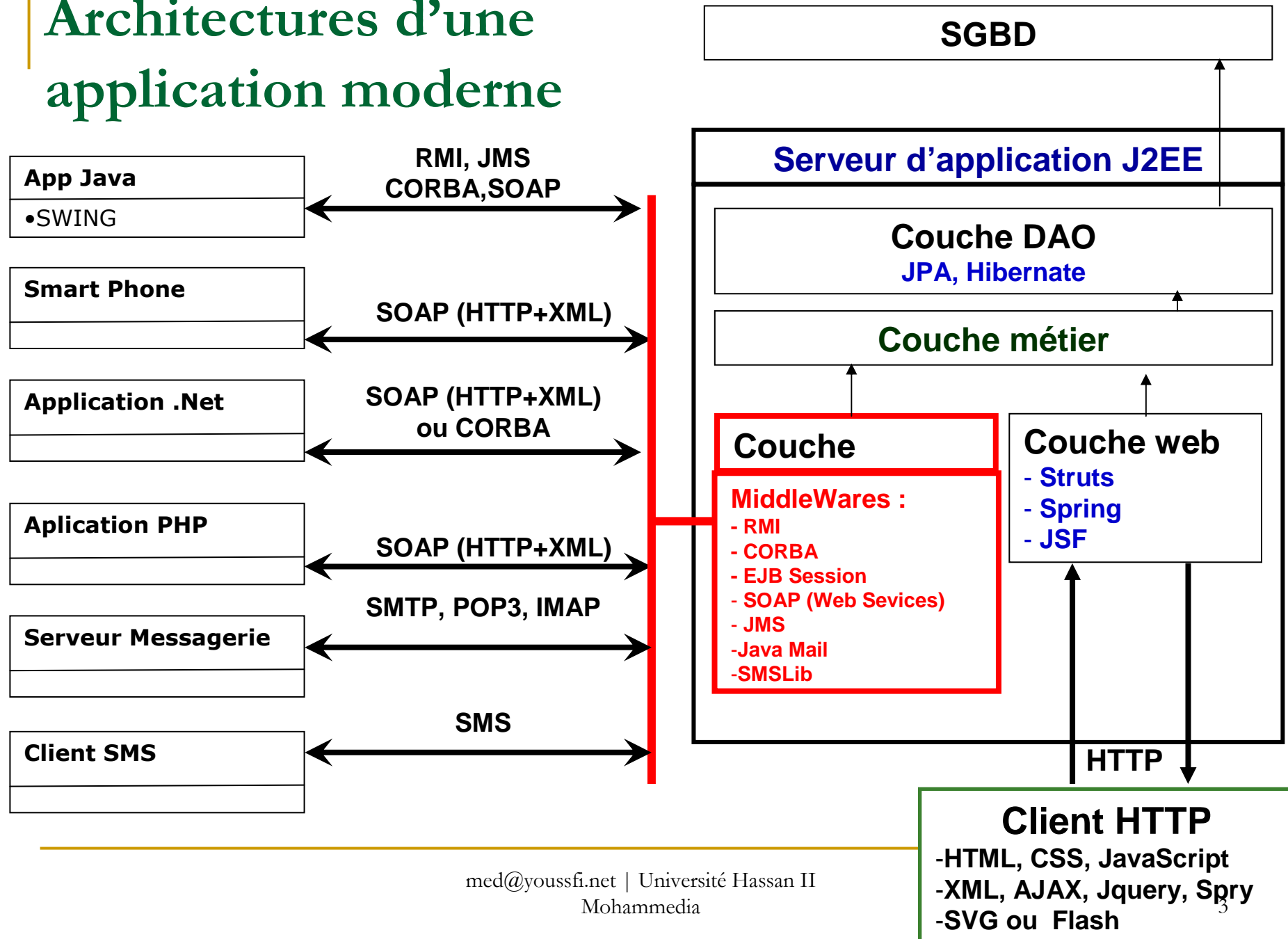
Mohamed Youssfi [ [med@yousfi.net](mailto:med@yousfi.net) ]

ENSET

Université Hassan II Mohammedia

# Rappels

# Architectures d'une application moderne



---

# Rappels :Qualité d'un Logiciel

La qualité d'un logiciel se mesure par rapport à plusieurs critères :

- Répondre aux spécifications fonctionnelles :
  - ❑ Une application est créée pour répondre , tout d'abord, aux besoins fonctionnels des entreprises.
- Les performances:
  - ❑ La rapidité d'exécution et Le temps de réponse
  - ❑ Doit être bâtie sur une architecture robuste.
  - ❑ Eviter le problème de montée en charge
- La maintenance:
  - ❑ Une application doit évoluer dans le temps.
  - ❑ Doit être fermée à la modification et ouverte à l'extension
  - ❑ Une application qui n'évolue pas meurt.
  - ❑ Une application mal conçue est difficile à maintenir, par suite elle finit un jour à la poubelle.

# Qualité d'un Logiciel

- Sécurité
  - Garantir l'intégrité et la sécurité des données
- Portabilité
  - Doit être capable de s'exécuter dans différentes plateformes.
- Capacité de communiquer avec d'autres applications distantes.
- Disponibilité et tolérance aux pannes
- Capacité de fournir le service à différents type de clients :
  - Client lourd : Interfaces graphiques SWING
  - Interface Web : protocole http
  - Client SmartPhone
  - Téléphone : SMS
  - ....
- Design des ses interfaces graphiques
  - Charte graphique et charte de navigation
  - Accès via différentes interfaces (Web, Téléphone, PDA, ,)
- Coût du logiciel

---

# Contenu de la formation

- Accès aux bases de données via JDBC
- Mapping objet relationnel avec Hibernante Framework
- Comment créer une application fermée à la modification et ouverte à l'extension
- Les Threads
- Les sockets
- Programmation Orientée Objets distribués avec RMI

# Accès aux bases de données via JDBC

M.Youssfi

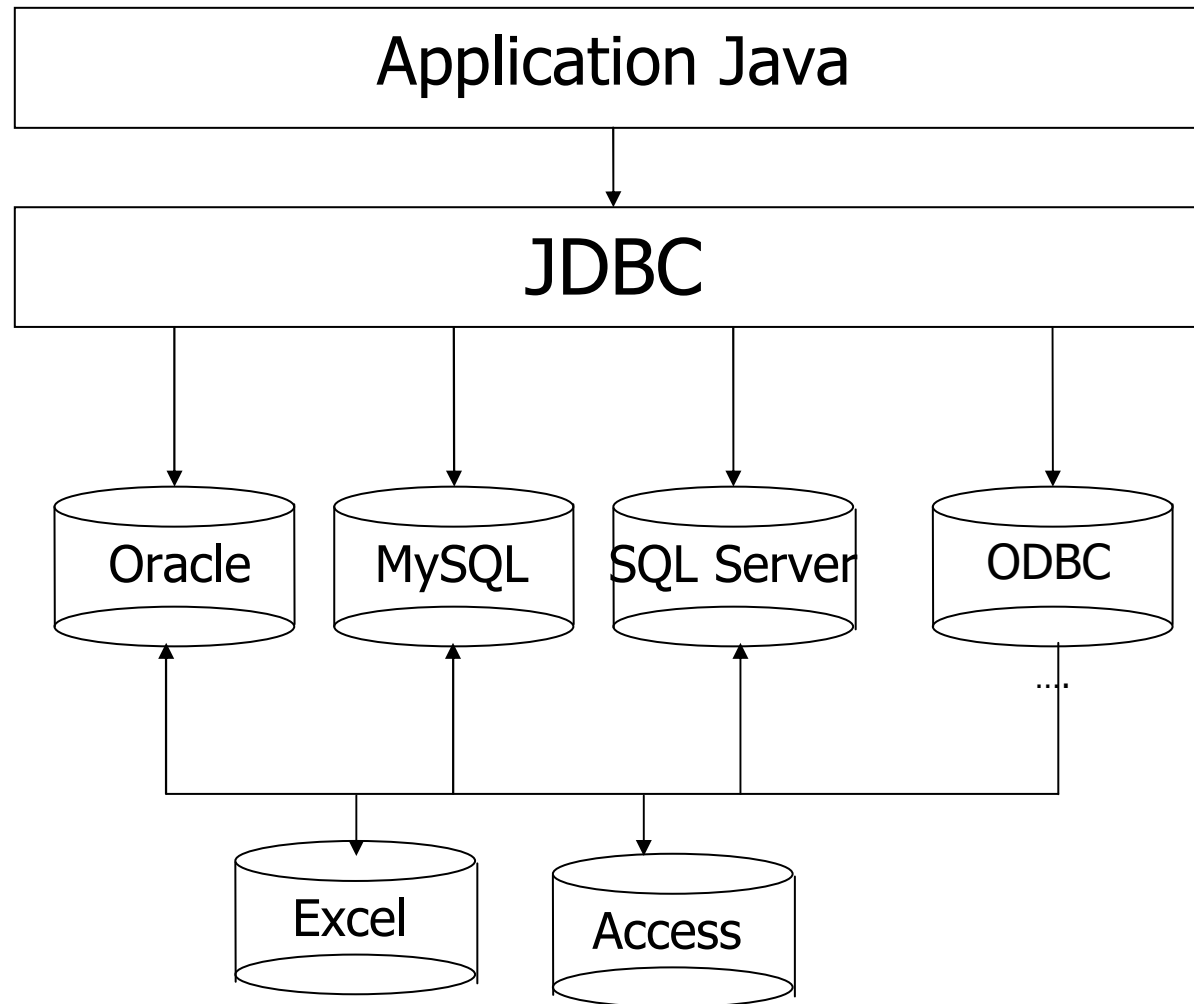
---

# Pilotes JDBC

- Pour qu'une application java puisse communiquer avec un serveur de bases de données, elle a besoin d'utiliser les pilotes JDBC (Java Data Base Connectivity)
- Les Pilotes JDBC est une bibliothèque de classes java qui permet, à une application java, de communiquer avec un SGBD via le réseau en utilisant le protocole TCP/IP
- Chaque SGBD possède ses propres pilotes JDBC.
- Il existe un pilote particulier « JdbcOdbcDriver » qui permet à une application java communiquer avec n'importe quelle source de données via les pilotes ODBC (Open Data Base Connectivity)
- Les pilotes ODBC permettent à une application Windows de communiquer une base de données quelconque (Access, Excel, MySQL, Oracle, SQL SERVER etc...)
- La bibliothèque JDBC a été conçu comme interface pour l'exécution de requêtes SQL. Une application JDBC est isolée des caractéristiques particulières du système de base de données utilisé.



# Java et JDBC

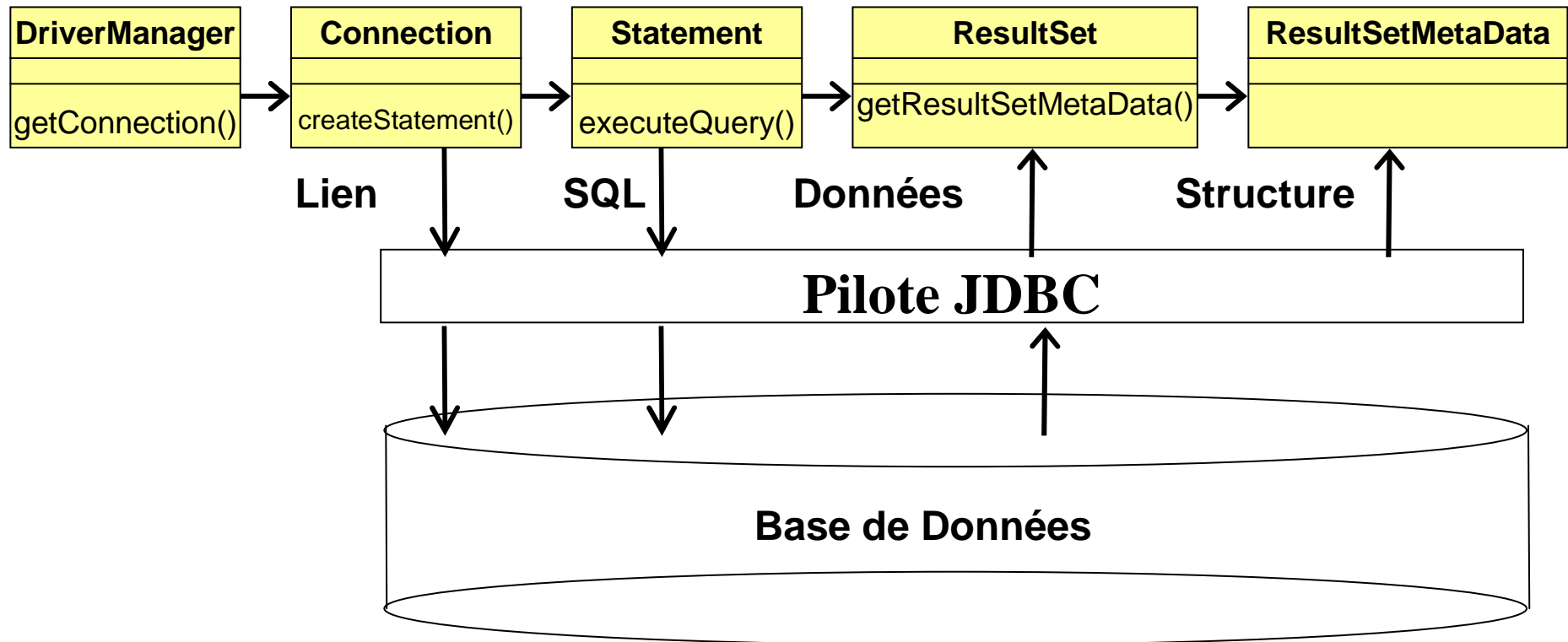


# Créer une application JDBC

Pour créer une application élémentaire de manipulation d'une base de données il faut suivre les étapes suivantes :

- Chargement du Pilote JDBC ;
- Identification de la source de données ;
- Allocation d'un objet **Connection**
- Allocation d'un objet Instruction **Statement** (ou **PreparedStatement**);
- Exécution d'une requête à l'aide de l'objet Statement ;
- Récupération de données à partir de l'objet renvoyé **ResultSet** ;
- Fermeture de l'objet ResultSet ;
- Fermeture de l'objet Statement ;
- Fermeture de l'objet Connection.

# Créer une application JDBC



# Démarche JDBC

## ■ Charger les pilotes JDBC :

- ❑ Utiliser la méthode `forName` de la classe `Class`, en précisant le nom de la classe pilote.

- ❑ Exemples:

- Pour charger le pilote `JdbcOdbcDriver`:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver") ;
```

- Pour charger le pilote jdbc de MySQL:

```
Class.forName("com.mysql.jdbc.Driver") ;
```

# Créer une connexion

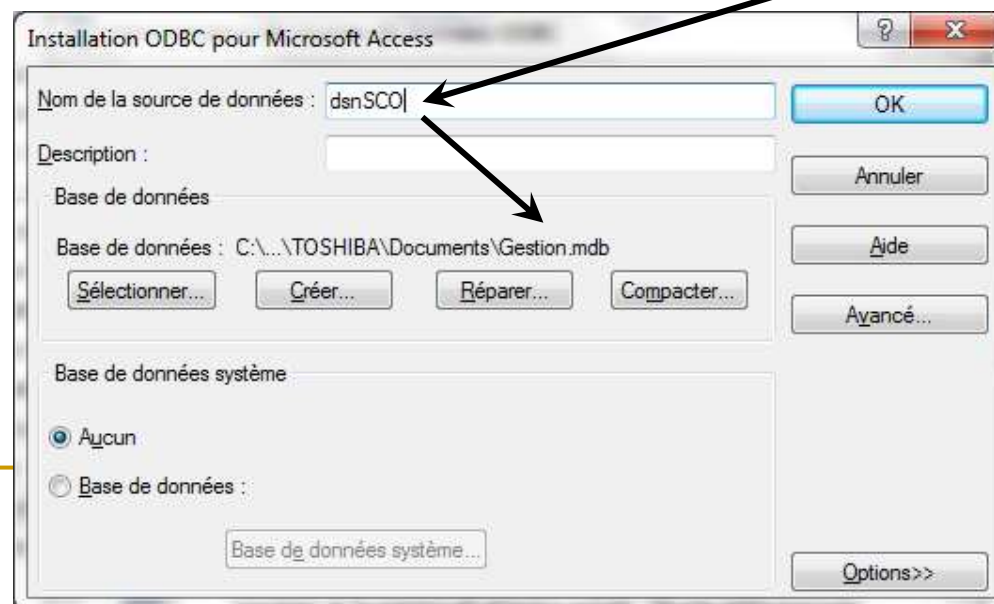
- Pour créer une connexion à une base de données, il faut utiliser la méthode statique `getConnection()` de la classe `DriverManager`. Cette méthode fait appel aux pilotes JDBC pour établir une connexion avec le SGBDR, en utilisant les sockets.

- Pour un pilote `com.mysql.jdbc.Driver` :

`Connection conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/DB", "user", "pass" );`

- Pour un pilote `sun.jdbc.odbc.JdbcOdbcDriver` :

`Connection conn=DriverManager.getConnection("jdbc:odbc:dsnSCO", "user", "pass" );`



## Objets Statement, ResultSet et ResultSetMetaData

- Pour exécuter une requête SQL, on peut créer l'objet Statement en utilisant la méthode `createStatement()` de l'objet Connection.
- Syntaxe de création de l'objet Statement

```
Statement st=conn.createStatement();
```

- Exécution d'une requête SQL avec l'objet Statement :
  - Pour exécuter une requête SQL de type select, on peut utiliser la méthode `executeQuery()` de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

```
ResultSet rs=st.executeQuery("select * from PRODUITS");
```

- Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode `executeUpdate()` de l'objet Statement :

```
st.executeUpdate("insert into PRODUITS (...) values(...)");
```

- Pour récupérer la structure d'une table, il faut créer l'objet `ResultSetMetaData` en utilisant la méthode `getMetaData()` de l'objet `ResultSet`.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

## Objet PreparedStatement

- Pour exécuter une requête SQL, on peut également créer l'objet PreparedStatement en utilisant la méthode `prepareStatement()` de l'objet Connection.
- Syntaxe de création de l'objet PreparedStatement

```
PreparedStatement ps=conn.prepareStatement("select *  
from PRODUITS where NOM_PROD like ? AND PRIX<?");
```

- Définir les valeurs des paramètres de la requête:

```
ps.setString(1, "%" + motCle + "%");
```

```
ps.setString(2, p);
```

- Exécution d'une requête SQL avec l'objet PreparedStatement :

- Pour exécuter une requête SQL de type select, on peut utiliser la méthode `executeQuery()` de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

```
ResultSet rs=ps.executeQuery();
```

- Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode `executeUpdate()` de l'objet Statement :

```
ps.executeUpdate();
```

---

## Récupérer les données d'un ResultSet

- Pour parcourir un ResultSet, on utilise sa méthode `next()` qui permet de passer d'une ligne à l'autre. Si la ligne suivante existe, la méthode `next()` retourne `true`. Si non elle retourne `false`.
- Pour récupérer la valeur d'une colonne de la ligne courante du ResultSet, on peut utiliser les méthodes `getInt(colonne)`, `getString(colonne)`, `getFloat(colonne)`, `getDouble(colonne)`, `getDate(colonne)`, etc... colonne représente le numéro ou le nom de la colonne de la ligne courante.
- **Syntaxe:**

```
while(rs.next()){  
    System.out.println(rs.getInt(1));  
    System.out.println(rs.getString("NOM_PROD"));  
    System.out.println(rs.getDouble("PRIX"));  
}
```



## Exploitation de l'objet ResultSetMetaData

- L'objet ResultSetMetaData est très utilisé quand on ne connaît pas la structure d'un ResultSet. Avec L'objet ResultSetMetaData, on peut connaître le nombre de colonnes du ResultSet, le nom, le type et la taille de chaque colonne.
- Pour afficher, par exemple, le nom, le type et la taille de toutes les colonnes d'un ResultSet rs, on peut écrire le code suivant:

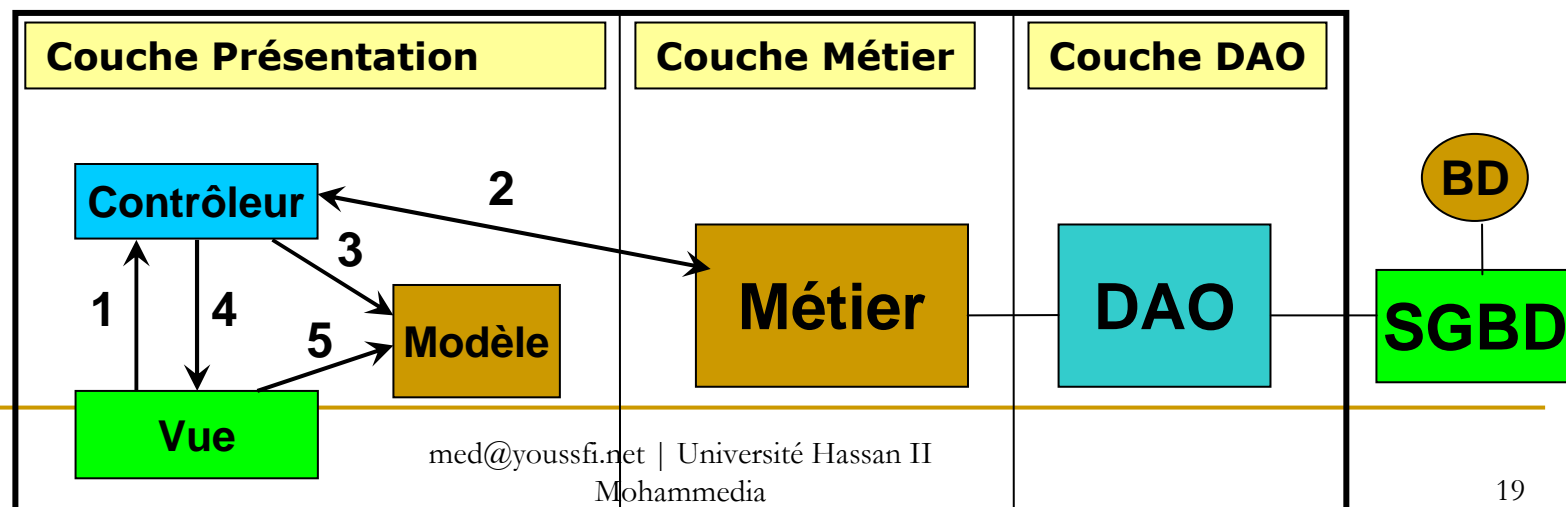
```
ResultSetMetaData rsmd=rs.getMetaData();
// Parcourir toutes les colonnes
for(int i=0;i<rsmd.getColumnCount();i++){
    // afficher le nom de la colonne numéro i
    System.out.println(rsmd.getColumnName(i));
    // afficher le type de la colonne numéro i
    System.out.println(rsmd.getColumnTypeName(i));
    // afficher la taille de la colonne numéro i
    System.out.println(rsmd.getColumnDisplaySize(i));
}
// Afficher tous les enregistrements du ResultSet rs
while (rs.next()){
    for(int i=0;i<rsmd.getColumnCount();i++){
        System.out.println(rs.getString(i));
    }
}
```

# Mapping objet relationnel

- Dans la pratique, on cherche toujours à séparer la logique de métier de la logique de présentation.
- On peut dire qu'on peut diviser une application en 3 couches:
  - La couche d'accès aux données: DAO
    - Partie de l'application qui permet d'accéder aux données de l'application. Ces données sont souvent stockées dans des bases de données relationnelles.
  - La couche Métier:
    - Regroupe l'ensemble des traitements que l'application doit effectuer.
  - La couche présentation:
    - S'occupe de la saisie des données et de l'affichage des résultats;

# Architecture d'une application

- Une application se compose de plusieurs couches:
  - ❑ La couche DAO qui s'occupe de l'accès aux bases de données.
  - ❑ La couche métier qui s'occupe des traitements.
  - ❑ La couche présentation qui s'occupe de la saisie, le contrôle et l'affichage des résultats. Généralement la couche présentation respecte le pattern MVC qui fonctionne comme suit:
    1. La vue permet de saisir les données, envoie ces données au contrôleur
    2. Le contrôleur récupère les données saisies. Après la validation de ces données, il fait appel à la couche métier pour exécuter des traitements.
    3. Le contrôleur stocke le résultat de le modèle.
    4. Le contrôleur fait appel à la vue pour afficher les résultats.
    5. La vue récupère les résultats à partir du modèle et les affiche.

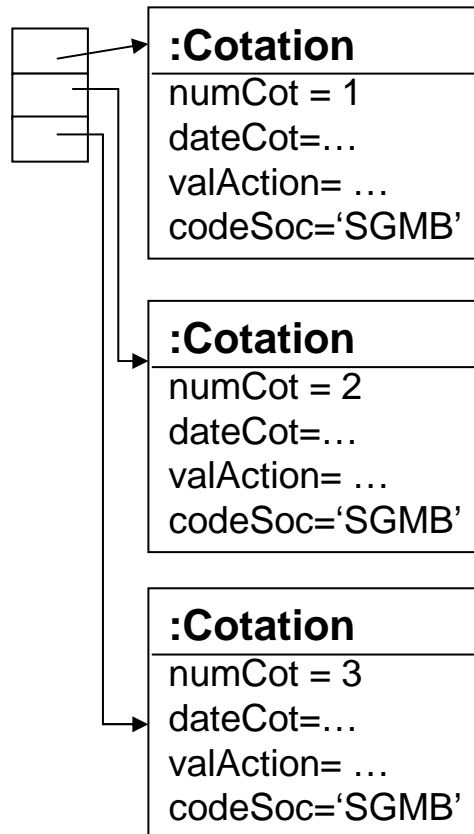


# Mapping objet relationnel

- D'une manière générale les applications sont orientée objet :
  - Manipulation des objet et des classes
  - Utilisation de l'héritage et de l'encapsulation
  - Utilisation du polymorphisme
- D'autres part les données persistantes sont souvent stockées dans des bases de données relationnelles.
- Le mapping Objet relationnel consiste à faire correspondre un enregistrement d'une table de la base de données à un objet d'une classe correspondante.
- Dans ce cas on parle d'une classe persistante.
- Une classe persistante est une classe dont l'état de ses objets sont stockés dans une unité de sauvegarde (Base de données, Fichier, etc..)

# Couche Métier : Mapping objet relationnel

cots



```
public List<Cotation> getCotations(String codeSoc){
    List<Cotation> cotations=new ArrayList<Cotation>();
    try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn=DriverManager.getConnection
            ("jdbc:mysql://localhost:3306/bourse_ws","root","");
        PreparedStatement ps=conn.prepareStatement
            ("select * from cotations where CODE_SOCIETE=?");
        ps.setString(1, codeSoc);
        ResultSet rs=ps.executeQuery();
        while(rs.next()){
            Cotation cot=new Cotation();
            cot.setNumCotation(rs.getLong("NUM_COTATION"));
            cot.setDateCotation(rs.getDate("DATE_COTATION"));
            cot.setCodeSociete(rs.getString("CODE_SOCIETE"));
            cot.setValAction(rs.getDouble("VAL_ACTION"));
            cotations.add(cot);
        }
    } catch (Exception e) { e.printStackTrace();}
    return(cotations);
}
```

NUM_COTATION	DATE_COTATION	VAL_ACTION	CODE_SOCIETE
1	2008-08-30 15:57:50	2093.17199826538	SGMB
2	2008-08-30 15:57:52	258.769396752267	SGMB
3	2008-08-30 15:57:52	1050.71222698514	SGMB

# Application

- On considère une base de données qui contient une table ETUDIANTS qui permet de stocker les étudiants d'une école. La structure de cette table est la suivante :

Champ	Type	Interclassement	Attributs	Null	Défaut	Extra
ID_ET	int(11)			Non	Aucun	auto_increment
NOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
PRENOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
EMAIL	varchar(25)	latin1_swedish_ci		Non	Aucun	
VILLE	varchar(200)	latin1_swedish_ci		Non	Aucun	

ID_ET	NOM	PRENOM	EMAIL	VILLE
1	A	PA	A@YAHOO.FR	casa
2	B	PB	B@YAHOO.FR	rabat
3	C	PC	C@YAHOO.FR	casa
4	BBCAAC	BBCAAC	ab@yahoo.fr	casa

- Nous souhaitons créer une application java qui permet de saisir au clavier un motclé et d'afficher tous les étudiants dont le nom contient ce mot clé.
- Dans cette application devons séparer la couche métier de la couche présentation.

# Application

- Pour cela, la couche métier est représentée par un modèle qui se compose de deux classes :
  - La classe Etudiant.java : c'est une classe persistante c'est-à-dire que chaque objet de cette classe correspond à un enregistrement de la table ETUDIANTS. Elle se compose des :
    - champs privés idEtudiant, nom, prenom, email et ville,
    - d'un constructeur par défaut,
    - des getters et setters.

Ce genre de classe c'est ce qu'on appelle un java bean.

- La classe Scholarite.java :
  - c'est une classe non persistante dont laquelle, on implémente les différentes méthodes métiers.
  - Dans cette classe, on fait le mapping objet relationnel qui consiste à convertir un enregistrement d'une table en objet correspondant.
  - Dans notre cas, une seule méthode nommée getEtudiants(String mc) qui permet de retourner une Liste qui contient tous les objets Etudiant dont le nom contient le mot clé « mc ».

---

# Application

## Travail à faire :

### Couche données :

- ❑ Créer la base de données « SCOLARITE » de type MSAccess ou MySQL
- ❑ Pour la base de données Access, créer une source de données système nommée « dsnScolarite », associée à cette base de données.
- ❑ Saisir quelques enregistrements de test

### Couche métier. ( package metier) :

- ❑ Créer la classe persistante Etudiant.java
- ❑ Créer la classe des business méthodes Scolarite.java

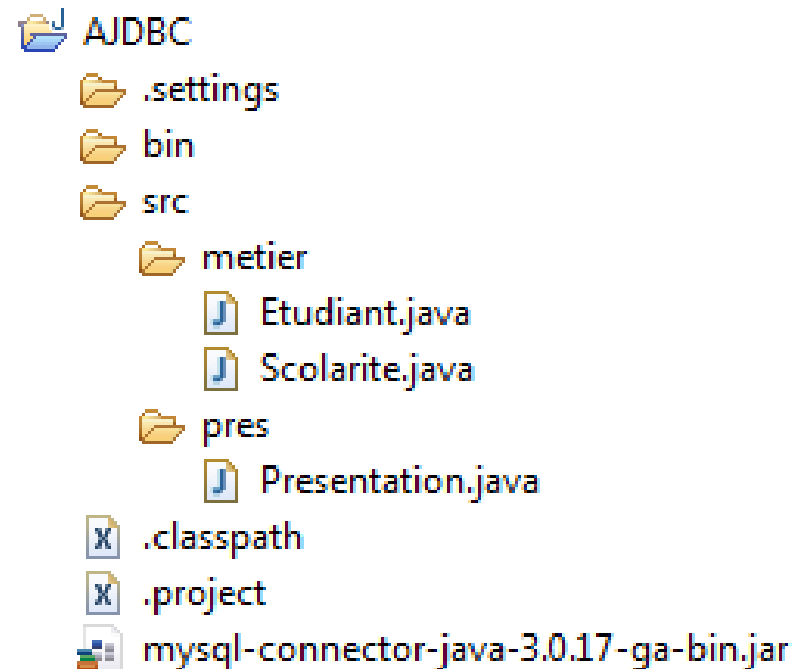
### Couche présentation (package pres):

- ❑ Créer une application de test qui permet de saisir au clavier le mot clé et qui affiche les étudiants dont le nom contient ce mot clé.



## Couche métier : la classe persistante Etudiant.java

```
package metier;  
  
public class Etudiant {  
    private Long idEtudiant;  
    private String nom, prenom, email;  
    // Getters et Stters  
}
```



# Couche métier : la classe Scholarite.java

```
package metier;
import java.sql.*; import java.util.*;
public class Scholarite {
    public List<Etudiant> getEtudiantParMC(String mc){
        List<Etudiant> etds=new ArrayList<Etudiant>();
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection conn=
                DriverManager.getConnection("jdbc:mysql://localhost:3306/DB_SCO","root","");
            PreparedStatement ps=conn.prepareStatement("select * from ETUDIANTS where NOM
                like ?");
            ps.setString(1,"%"+mc+"%");
            ResultSet rs=ps.executeQuery();
            while(rs.next()){
                Etudiant et=new Etudiant();
                et.setIdEtudiant(rs.getLong("ID_ET"));et.setNom(rs.getString("NOM"));
                et.setPrenom(rs.getString("PRENOM"));et.setEmail(rs.getString("EMAIL"));
                etds.add(et);
            }
        } catch (Exception e) {    e.printStackTrace(); }
        return etds;
    }
}
```

# Couche Présentation : Applications Simple

```
package pres;
import java.util.List;import java.util.Scanner;
import metier.Etudiant;import metier.Scolarite;
public class Presentation {
    public static void main(String[] args) {

        Scanner clavier=new Scanner(System.in);

        System.out.print("Mot Clé:");

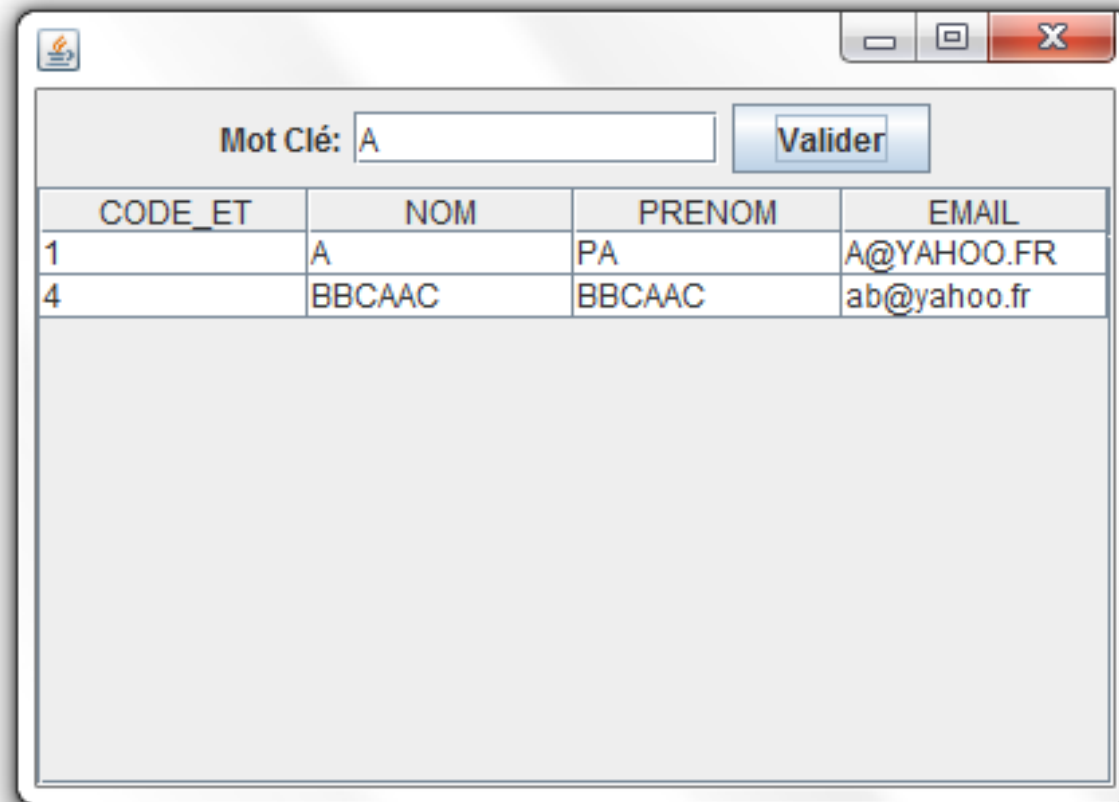
        String mc=clavier.next();

        Scolarite metier=new Scolarite();

        List<Etudiant> etds=metier.getEtudiantParMC(mc);

        for(Etudiant et:etds)
            System.out.println(et.getNom()+"\t"+et.getEmail());
    }
}
```

# Couche Présentation : Application SWING

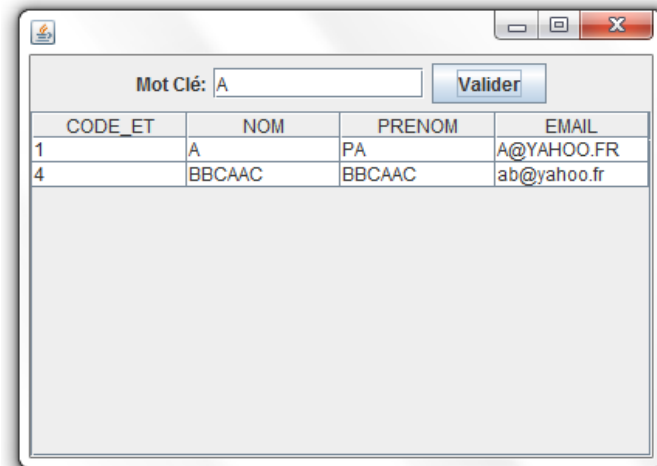


The image shows a Java Swing window with a title bar containing a small icon and standard window controls (minimize, maximize, close). The window contains a login form with a label 'Mot Clé:' followed by a text input field containing the letter 'A' and a 'Valider' button. Below the form is a table with four columns: 'CODE\_ET', 'NOM', 'PRENOM', and 'EMAIL'. The table has two data rows. The first row contains the values '1', 'A', 'PA', and 'A@YAHOO.FR'. The second row contains the values '4', 'BBCAAC', 'BBCAAC', and 'ab@yahoo.fr'. The table is followed by a large, empty rectangular area.

CODE_ET	NOM	PRENOM	EMAIL
1	A	PA	A@YAHOO.FR
4	BBCAAC	BBCAAC	ab@yahoo.fr

# Le modèle de données pour JTable

```
package pres;
import java.util.List;import java.util.Vector;
import javax.swing.table.AbstractTableModel;
import metier.Etudiant;
public class EtudiantModele extends AbstractTableModel{
    private String[] tabelColumnNames=new
    String[]{"CODE_ET", "NOM", "PRENOM", "EMAIL"};
    private Vector<String[]> tableValues=new Vector<String[]>();
@Override
public int getRowCount() {
    return tableValues.size();
}
@Override
public int getColumnCount() {
    return tabelColumnNames.length;
}
@Override
public Object getValueAt(int rowIndex, int columnIndex) {
    return tableValues.get(rowIndex)[columnIndex];
}
```

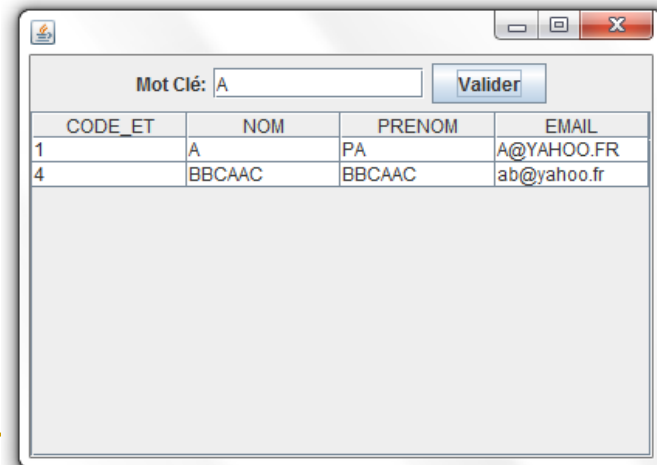


# Le modèle de données pour JTable (Suite)

```
@Override
public String getColumnName(int column) {
    return tabelColumnNames[column];
}

public void setData(List<Etudiant> etudiants){
    tableValues=new Vector<String[]>();
    for(Etudiant et:etudiants){
        tableValues.add(new String[]{
            String.valueOf(et.getIdEtudiant()),et.getNom(),et.getPrenom
            (),et.getEmail()});
    }

    fireTableChanged(null);
}
}
```



The screenshot shows a Java Swing window with a title bar. Inside, there is a text field labeled 'Mot Clé:' containing the letter 'A', followed by a 'Valider' button. Below this is a JTable with four columns: 'CODE\_ET', 'NOM', 'PRENOM', and 'EMAIL'. The table contains two rows of data.

CODE_ET	NOM	PRENOM	EMAIL
1	A	PA	A@YAHOO.FR
4	BBCAAC	BBCAAC	ab@yahoo.fr

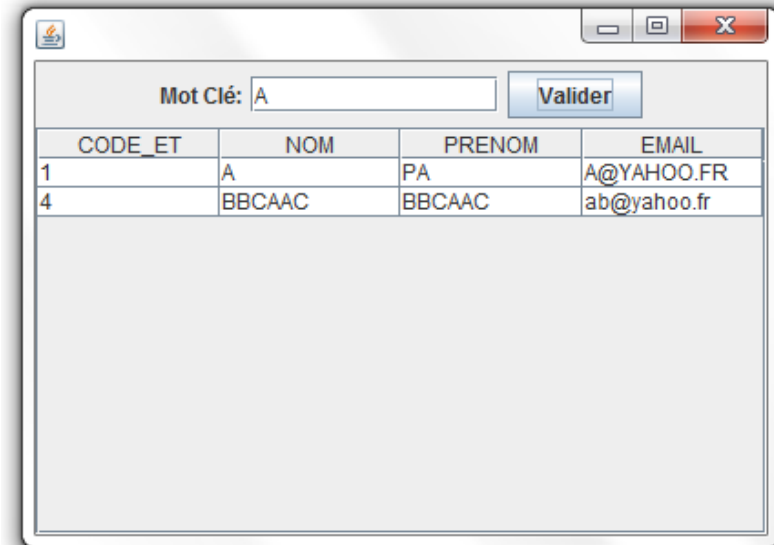
# L'application SWING

```
package pres;
import java.awt.*;import java.awt.event.*;import javax.swing.*;
import metier.*;
public class EtudiantFrame extends JFrame {
    private JScrollPane jsp;

    private JLabel jLMC=new JLabel("Mot Clé:");
    private JTextField jTFMC=new JTextField(12);
    private JButton jbValider=new JButton("Valider");

    private JTable jTableEtudiants;
    private JPanel jpN=new JPanel();

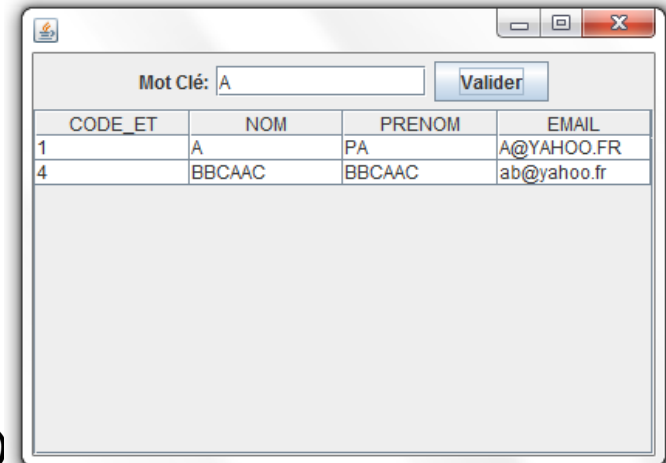
    private Sclarite metier
=new Sclarite();
    private EtudiantModele model;
```



# L'application SWING (Suite)

```
public EtudiantFrame() {
    this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    this.setLayout(new BorderLayout());
    jTFMC.setToolTipText("Mot Clé:");
    jpN.setLayout(new FlowLayout());
    jpN.add(jLMC); jpN.add(jTFMC);
    jpN.add(jBValider);
    this.add(jpN, BorderLayout.NORTH);
    model=new EtudiantModele();
    jTableEtudiants=new.JTable(model);
    jsp=new JScrollPane(jTableEtudiants);
    this.add(jsp, BorderLayout.CENTER);
    this.setBounds(10,10,500,500);
    this.setVisible(true);
    jBValider.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String mc=jTFMC.getText();
        model.setData(metier.getEtudiantParMC(mc));
    }
    });
}

public static void main(String[] args) {new EtudiantFrame();}}
```



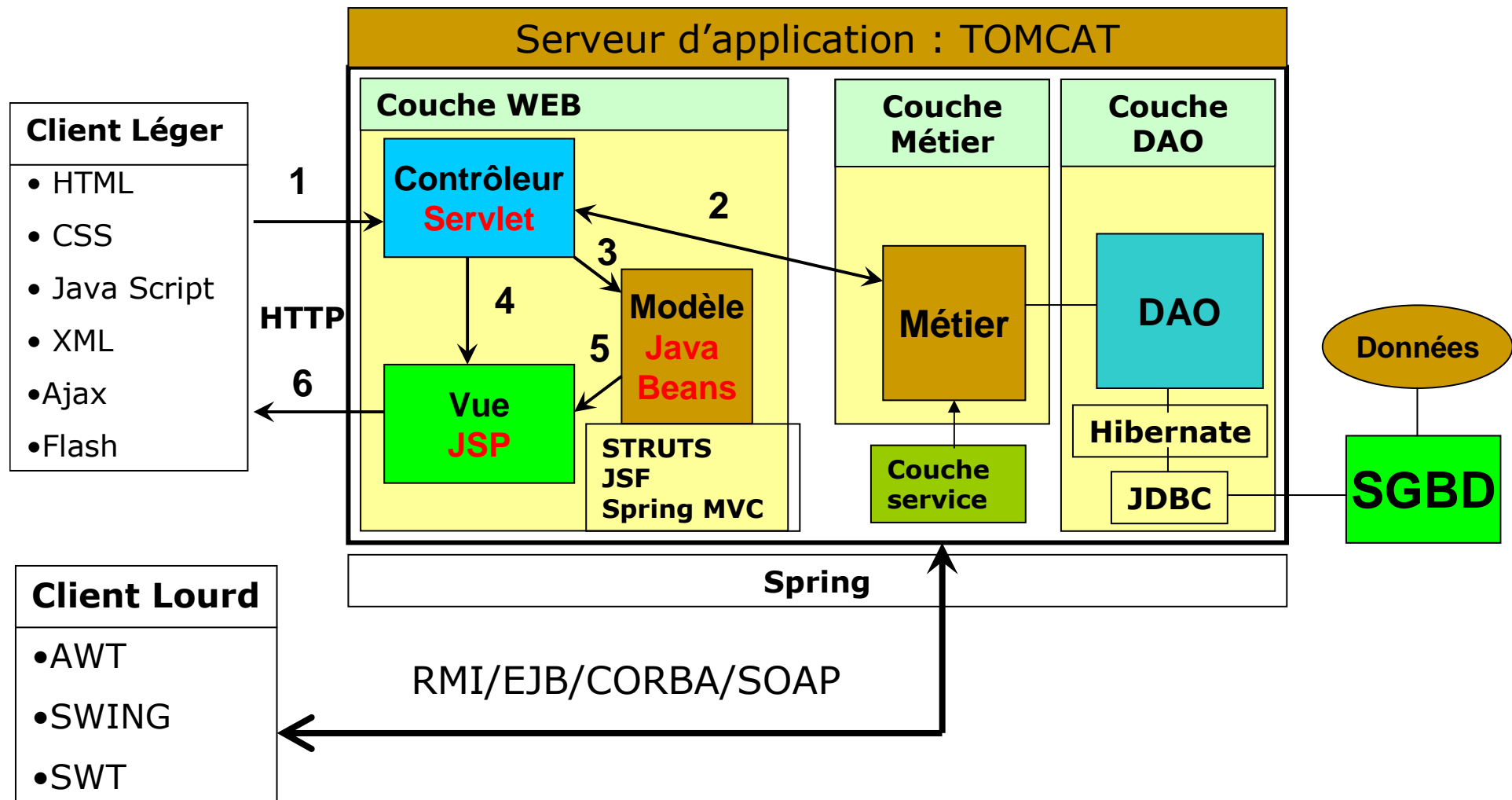


---

# Hibernate Framework

M.Youssfi

# Architecture J2EE



---

# Introduction

- Travailler dans les deux univers que sont l'orienté objet et la base de données relationnelle peut être lourd et consommateur en temps dans le monde de l'entreprise d'aujourd'hui.
- Hibernate est un outil de mapping objet/relationnel pour le monde Java.
- Le terme mapping objet/relationnel (ORM) décrit la technique consistant à faire le lien entre la représentation objet des données et sa représentation relationnelle basée sur un schéma SQL.

---

# Introduction

- Hibernate s'occupe du transfert des objets Java dans les tables de la base de données
- En plus, il permet de requêter les données et propose des moyens de les récupérer.
- Il peut donc réduire de manière significative le temps de développement qui aurait été autrement perdu dans une manipulation manuelle des données via SQL et JDBC

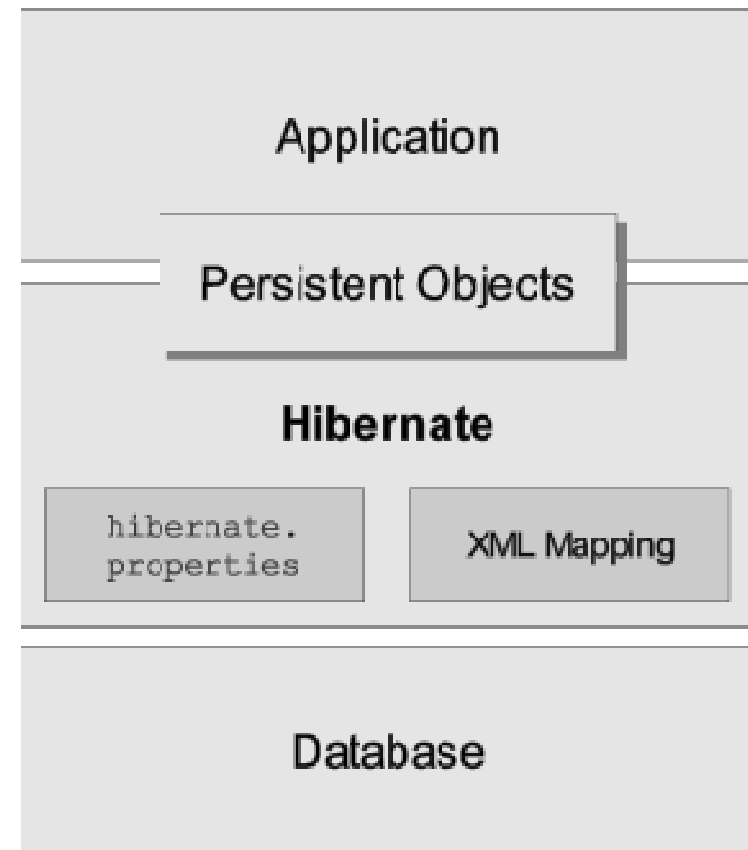
---

# But de Hibernate

- ❑ Le but d'Hibernate est de libérer le développeur de 95 pourcent des tâches de programmation liées à la persistance des données communes.
- ❑ Hibernate assure la portabilité de votre application si vous changez de SGBD.
- ❑ Hibernate propose au développeur des méthodes d'accès aux bases de données plus efficaces ce qui devrait rassurer les développeurs.

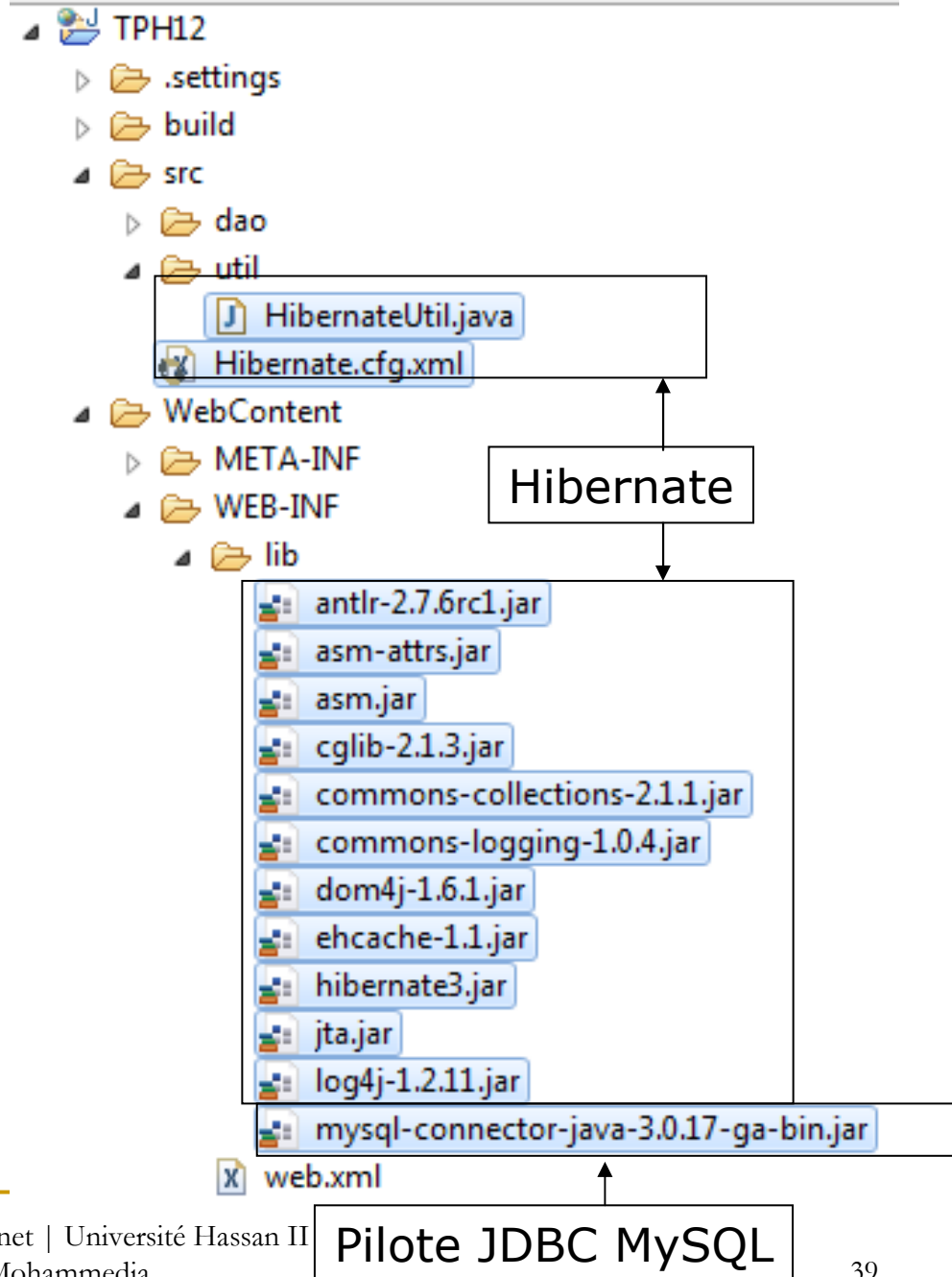
# Première approche de l'architecture d'Hibernate

- Hibernate permet d'assurer la persistance des objets de l'application dans un entrepôt de données.
- Cet entrepôt de données est dans la majorité des cas une base de données relationnelle, mais il peut être un fichier XML.
- Le mapping des objets est effectuée par Hibernate en se basant sur des fichiers de configuration en format texte ou souvent XML.



# Première Application

- Préparer le projet:
  - Au premier lieu, nous avons besoin de télécharger Hibernate et de copier des bichiers .jar dans le répertoire lib de votre projet:
  - Pour travailler avec une base de données MySQL, nous aurons besoin également du connecteur JDBC de MySQL



# Exemple d'application

- Supposant que nous avons besoin de gérer la participation des personnes à des réunions.
- Une personne participe à plusieurs réunions et chaque réunion concerne plusieurs personnes.
- On va s'intéresser tout d'abord au cas d'une classe et par la suite nous reviendrons sur le mapping des associations entre les classes.

C Reunion	
+	idReunion: Long
+	dateReunion: Date
+	titreReunion: String
●	getIdReunion(): Long
●	setIdReunion(idReunion: Long)
●	getDateReunion(): Date
●	setDateReunion(dateReunion: Date)
●	getTitreReunion(): String
●	setTitreReunion(titreReunion: String)



# Classe Reunion

```
package dao;

import java.util.Date; import java.io.*;
public class Reunion implements Serializable {
    private Long idReunion;
    private String titre;
    private Date dateReunion;

    public Reunion() {
    }
    public Reunion(String titre, Date dateReunion) {
        this.titre = titre;
        this.dateReunion = dateReunion;
    }
    // Getters et Setters

}
```

# Fichier de mapping de la classe Reunion

- La structure basique d'un fichier de mapping ressemble à celà:

```
<?xml version="1.0"?>
```

```
<!DOCTYPE hibernate-mapping PUBLIC      "-  
    //Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
```

```
    [Contenu]
```

```
</hibernate-mapping>
```

---

## Contenu du fichier de mapping de la classe Reunion: Reunion.hbm.xml

```
<hibernate-mapping package="dao">
  <class name="Reunion" table="REUNIONS">
    <id name="idReunion" column="ID_REUNION">
      <generator class="native"></generator>
    </id>
    <property name="titre" column="TITRE"></property>
    <property name="dateReunion"
      column="DATE_REUNION"></property>
  </class>
</hibernate-mapping>
```

# Version Annotations JPA

```
package dao;
import java.util.Date; import javax.persistence.*; java.io.*;
@Entity
@Table(name="REUNIONS")
public class Reunion implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID_REUNION")
    private Long idReunion;
    @Column(name="TITRE")
    private String titre;
    private Date dateReunion;

    public Reunion() {
    }
    public Reunion(String titre, Date dateReunion) {
        this.titre = titre; this.dateReunion = dateReunion;
    }
    // Getters et Setters
}
```

# Classe DaoImpl

```
package dao;

import java.util.List; import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import util.HibernateUtil;

public class DaoImpl {
    private SessionFactory sessionFactory;
    public MetierImpl() {
sessionFactory=HibernateUtil.getSessionFactory();
    }
    // Méthode de mapping objet relationnel
}
```

# Méthodes de Gestion de la persistance

```
public void addReunion(Reunion r){
    Session session=sessionFactory.getCurrentSession();
    session.beginTransaction();
    session.save(r);
    session.getTransaction().commit();
}

public List<Reunion> getReunionsParMotCle(String mc){
    Session session=sessionFactory.getCurrentSession();
    session.beginTransaction();
    Query req=session.createQuery("select r from Reunion r
        where r.titre like :x");
    req.setParameter("x", "%"+mc+"%");
    return req.list();
}
```

# Méthodes de Gestion de la persistance

```
public Reunion getReunion(Long idR){
    Session session=sessionFactory.getCurrentSession();
    session.beginTransaction();
    Reunion r=(Reunion) session.get(Reunion.class, idR);
    return r;
}

public void deleteReunion(Long idR){
    Session session=sessionFactory.getCurrentSession();
    session.beginTransaction();
    Reunion r=(Reunion) session.get(Reunion.class, idR);
    session.delete(r);
    session.getTransaction().commit();
}
```

# Nom et emplacement du fichier de mapping de la classe Reunion

- Pour respecter les règles de nommage, le fichier est nommé : Reunion.hbm.xml
- Il devrait être placé au même endroit de la classe Reunion:
  - +lib
    - <Hibernate et bibliothèques tierces>
  - +src
    - +dao
      - Reunion.java
      - DaoImpl.java
      - Reunion.hbm.xml
      - Test.java
    - +util
      - HibernateUtil.java
    - Hibernate.cfg.xml



---

# Configurer Hibernate

- Nous avons maintenant une classe persistante et son fichier de mapping. Il est temps de configurer Hibernate.
- Avant ça, nous avons besoin d'une base de données.
- Lancer MySQL et créer une base de données Nommée **GEST\_REUNIONS** .
- La structure de cette base de données sera créée automatiquement par Hibernate lors de l'exécution de l'application

## Configurer Hibernate : Hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC      "-
//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="connection.url">
      jdbc:mysql://localhost:3306/GEST_REUNIONS
    </property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
```

## Configurer Hibernate : Hibernate.cfg.xml (Suite)

```
<property name="connection.pool_size">1</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">create</property>
<property name="dialect">
    org.hibernate.dialect.MySQLDialect
</property>
<property name="cache.provider_class">
    org.hibernate.cache.NoCacheProvider
</property>
<mapping resource="gest/Reunion.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

# Emplacement du fichier Hibernate.cfg.xml

- Ce fichier doit être placé dans la racine du classpath.
- C'est-à-dire dans le répertoire src
  - +lib
    - <Hibernate et bibliothèques tierces>
  - +src
    - **Hibernate.cfg.xml**
    - +dao
      - Reunion.java
      - Reunion.hbm.xml

# Application Test

- Il est temps de charger et de stocker quelques objets Reunion, mais d'abord nous devons compléter la configuration avec du code d'infrastructure.
- Nous devons démarrer Hibernate. Ce démarrage inclut la construction d'un objet SessionFactory global et le stocker quelque part facile d'accès dans le code de l'application.
- Une SessionFactory peut ouvrir des nouvelles Sessions.
- Une Session représente une unité de travail la SessionFactory est un objet global instancié une seule fois.
- Nous créerons une classe d'aide HibernateUtil qui s'occupe du démarrage et rend la gestion des Sessions plus facile. Voici son implémentation :

# Classe Utilitaire HibernateUtil.java

```
package util;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class HibernateUtil {
    public static final SessionFactory sessionFactory;
    static {
        try {
            // Création de la SessionFactory à partir de hibernate.cfg.xml
            sessionFactory = new
                Configuration().configure("Hibernate.cfg.xml").buildSessionFactory();
        } catch (Exception ex) {
            e.printStackTrace();
        }
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

---

# Configurer le système de logs

- Nous avons finalement besoin de configurer le système de "logs".
- Cette opération n'est pas obligatoire. Mais elle permet au développeur d'afficher sur une sortie les messages fournis par Hibernate lors de son exécution.
- Ce qui permet de rendre transparente la couche logicielle de Hibernate.
- Hibernate utilise commons-logging et vous laisse le choix entre log4j et le système de logs du JDK
- La plupart des développeurs préfèrent log4j
- copiez **log4j.properties** de la distribution d'Hibernate (il est dans le répertoire etc/) dans votre répertoire src.
- Ce fichier contient une configuration du système de logs qui peut être personnalisée par l'utilisateur.
- L'infrastructure est complète et nous sommes prêts à effectuer un travail réel avec Hibernate.

# Créer des Réunions

```
public class Test {  
    public static void main(String[] args) {  
        MetierImpl metier = new MetierImpl();  
        // Ajouter 3 Réunions  
        metier.addReunion(new Reunion("R1", new Date()));  
        metier.addReunion(new Reunion("R2", new Date()));  
        metier.addReunion(new Reunion("R3", new Date()));  
        // Afficher les Réunions dont le titre contient R  
        List<Reunion> reunions=metier.getReunionsParMotCle("R");  
        for(Reunion r:reunions)  
            System.out.println(r.getTitre());  
        // Afficher le titre de la réunion dont le id est 2  
        Reunion r=metier.getReunion(2L);  
        System.out.println(et.getTitre());  
    }  
}
```



---

# Mapper les associations

- Nous avons mappé une classe d'une entité persistante vers une table.
- Partons de là et ajoutons quelques associations de classe.
- D'abord nous ajouterons des personnes à notre application, et stockerons une liste de réunions auxquelles ils participent.

# Mapper les associations

- Rappelons le problème : Une Réunion concerne plusieurs personnes et une personne peut participer à plusieurs réunions.



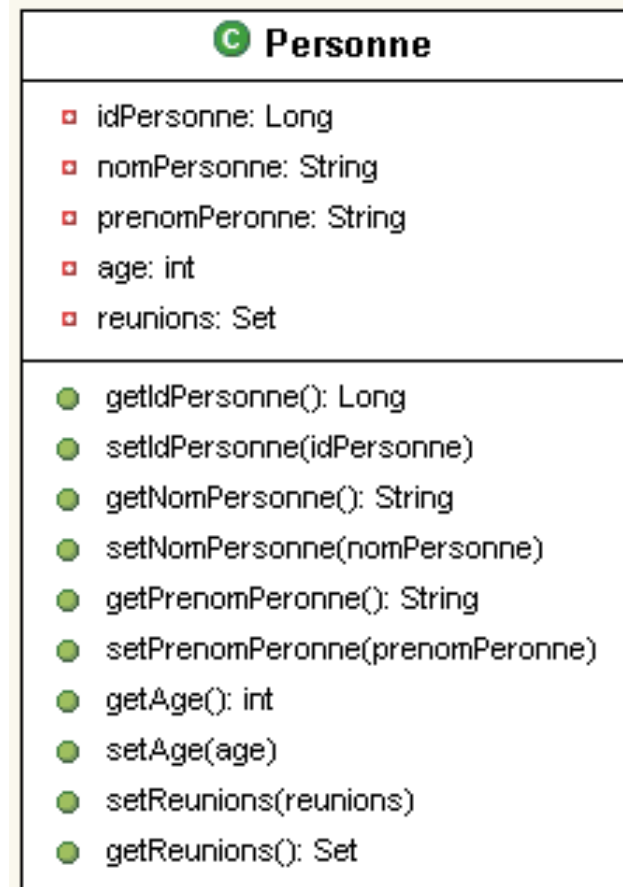
# Association Unidirectionnelle basée sur un Set

- Si on suppose que dans notre problème, nous aurons besoin de savoir pour chaque Personne chargée, quelles sont les différentes réunions auxquelles il a participé.
- Il est clair que notre association peut être modélisée unidirectionnelle.



# Mapper la classe Personne

- Une personne est caractérisée par les attributs privés:
  - Son identifiant
  - Son nom
  - Son prénom
  - et son age.
- Une collection Set de réunions.
- Les autres méthodes sont des accesseurs et les mutateurs.
- Sans oublier le constructeur par défaut.



---

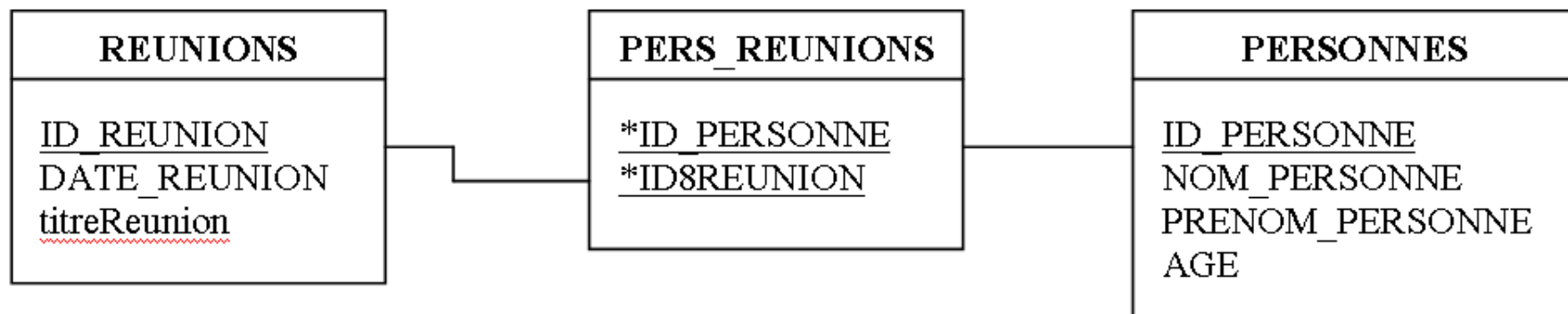
# Personne.java

```
package dao;
import java.util.*; import java.io.*;
public class Personne implements Serializable {
    private Long idPersonne;
    private String nomPersonne;
    private String prenomPersonne;
    private int age;
    private Set<Reunion> reunions=new HashSet<Reunion>();

    // getters and setters
}
```

# Fichier de mapping de la classe Personne : Personne.hbm.xml

- Avant d'écrire notre fichier xml de mapping, il est important de savoir à ce stade, comment serait modéliser notre problème dans une base de données relationnelle.
- Nous avons une association plusieurs à plusieurs non porteuse de propriété.
- Le schéma de la base de données relationnelle serait le suivant :



# Fichier de mapping de la classe Personne : Personne.hbm.xml

```
<hibernate-mapping package="dao">
  <class name="Personne" table="PERSONNES">
    <id name="idPersonne" column="ID_PERSONNE">
      <generator class="native"></generator>
    </id>
    <property name="nom"></property>
    <property name="prenom"></property>
    <property name="age"></property>
    <set name="reunions" table="PERS_REUNIONS">
      <key column="ID_PERSONNE"></key>
      <many-to-many class="Reunion" column="ID_REUNION">
        </many-to-many>
      </set>
    </class>
  </hibernate-mapping>
```

# En utilisant les annotations JPA

```
package dao;
import java.io.*; import java.util.*; import javax.persistence.*;
    @Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
private Long idPersonne;
    @Column(name="NOM_PERSONNE")
private String nomPersonne;
private String prenomPersonne;
private int age;
    @ManyToMany
    @JoinTable(name="PERS_REUNIONS")
private Set<Reunion> reunions=new HashSet<Reunion>();
// getters and setters
}
```



---

# Personne.hbm.xml

- Ce fichier doit être également placé dans le même dossier que la classe `Personne.java` c'est-à-dire `src\gest`.
- Ajouter ce fichier de mapping dans le fichier de configuration de hibernate `Hibernate.cfg.xml` :

```
<mapping resource="gest/Reunion.hbm.xml" />
```

```
<mapping resource="gest/Personne.hbm.xml" />
```

# Méthode pour ajouter une nouvelle Personne

```
public void addReunion(Reunion r){  
    Session session=  
        HibernateUtil.getSessionFactory().getCurrentSession();  
    session.beginTransaction();  
    session.save(r);  
    session.getTransaction().commit();  
}
```

# Méthode pour ajouter Une réunion à une personne

```
public void addReunionToPersonne(Long idPersonne, Long idReunion) {  
    Session session =  
        HibernateUtil.getSessionFactory().getCurrentSession();  
    session.beginTransaction();  
    // Charger une personne  
    Personne p = (Personne) session.get(Personne.class, idPersonne);  
    // Charger une réunion  
    Reunion r = (Reunion) session.get(Reunion.class, idReunion);  
    // Ajouter la réunion r à la collection reunions de la personne p  
    p.getReunions().add(r);  
    session.getTransaction().commit();  
}
```

---

# Méthode pour ajouter Une réunion à une personne

- Après le chargement d'une personne et d'une réunion, modifiez simplement la collection en utilisant les méthodes normales de la collection.
- Ici, nous utilisons la méthode add de la collection SET pour ajouter la réunion à la collection reunions de l'objet personne.
- Comme vous pouvez le voir, il n'y a pas d'appel explicite à update() ou save(), Hibernate détecte automatiquement que la collection a été modifiée et a besoin d'être mise à jour.
- Ceci est appelé *la vérification sale automatique* ("automatic dirty checking"),

# Méthode pour ajouter Une réunion à une personne

- et vous pouvez aussi l'essayer en modifiant le nom ou la propriété date de n'importe lequel de vos objets.
- Tant qu'ils sont dans un état *persistant*, c'est-à-dire, liés à une Session Hibernate particulière (c-à-d qu'ils ont juste été chargés ou sauvegardés dans une unité de travail), Hibernate surveille les changements et exécute le SQL correspondant.
- Le processus de synchronisation de l'état de la mémoire avec la base de données à la fin d'une unité de travail, est appelé *flushing*.
- Dans notre code, l'unité de travail s'achève par un commit (ou rollback) de la transaction avec la base de données - comme défini par notre option thread de configuration pour la classe `CurrentSessionContext`.

## Méthode qui retourne une personne chargée en utilisant son identifiant

```
public Personne getPersonne(Long idPersonne) {  
    Session session =  
        HibernateUtil.getSessionFactory().getCurrentSession();  
    session.beginTransaction();  
    Personne p = (Personne) session.get(Personne.class, idPersonne);  
    return p;  
}
```

**Ou encore en utilisant l'objet Criterea:**

```
public Personne getPersonneV2(Long id){  
    Session session=  
        HibernateUtil.getSessionFactory().getCurrentSession();  
    session.beginTransaction();  
    Criteria crit=session.createCriteria(Personne.class);  
    crit.add(Expression.eq("idPersonne", id));  
    Object o=crit.uniqueResult();  
    return (Personne)o;  
}
```

# Tester les méthodes

```
package dao;
import java.util.Date;import java.util.Iterator;
public class Test {
public static void main(String[] args) {
    DaoImpl dao=new DaoImpl();
    dao.addReunion(new Reunion(new Date(),"RT1"));
    dao.addReunion(new Reunion(new Date(),"RR2"));
    dao.addReunion(new Reunion(new Date(),"RT3"));
    dao.addPersonne(new Personne("P1","N1"));
    dao.addPersonne(new Personne("P2","N2"));
    dao.addPersonne(new Personne("P3","N3"));
    dao.addPersonneToReunion(new Long(1), new Long(1));
    dao.addPersonneToReunion(new Long(1), new Long(2));
    dao.addPersonneToReunion(new Long(2), new Long(1));
}
```

# Tester les méthodes

```
//Iterator<Reunion> it=dao.getAllReunions().iterator();
Iterator<Reunion> it=dao.getAllReunionsParMotCle("RT").iterator();
while(it.hasNext()){
    Reunion r=(Reunion)it.next();
    System.out.println(r.getTitre());
}
System.out.println("Consulter la personne 1");
Personne p=dao.getPersonne(new Long(1));
System.out.println("Nom="+p.getNom());
System.out.println("Prénom="+p.getPrenom());
System.out.println("Reunions");
Iterator<Reunion> it2=p.getReunions().iterator();
while(it2.hasNext()){
    Reunion r=it2.next();
    System.out.println(r.getTitre());
}
}
```



# Association bidirectionnelle

- Dans l'exemple précédent, nous avons mappé l'association entre la classe Personne et Reunion dans un seul sens.
- Ce qui nous permet d'ajouter des réunions à une personne, et en chargeant une personne, on peut connaître toutes les réunions auxquelles a participé cette personne.
- Si nous souhaitons également créer une réunion et lui ajouter des personne,
- et si nous voulions qu'une fois qu'on charge une réunion, on veut connaître toutes les personnes qui participent à cette réunion, Il faudrait également mapper l'association entre les deux classe dans l'autre sens.

# Association bidirectionnelle



- Dans ce cas l'association se traduit par :
  - ❑ Création d'une collection dans la classe **Personne** qui permet de stocker les réunions. Cette collection a été déjà créée dans le cas précédent. Nous avons appelé cette collection **reunions** et elle est de type **Set**.
  - ❑ Création d'une collection dans la classe **Reunion** qui permet d'associer des personnes à une réunion. Cette collection est de type **Set** et sera appelée **personnes**.

---

# Association bidirectionnelle

- Nous allons faire fonctionner l'association entre une personne et une réunion à partir des deux côtés en Java.
- Bien sûr, le schéma de la base de données ne change pas, nous avons toujours une pluralité many-to-many.
- Une base de données relationnelle est plus flexible qu'un langage de programmation orienté objet, donc elle n'a pas besoin de direction de navigation

# Association bidirectionnelle

- D'abord, ajouter une collection de participants à la classe Reunion :

```
private Set personnes = new HashSet();  
public Set getPersonnes() {  
    return personnes;  
}  
public void setPersonnes(Set personnes) {  
    this.personnes = personnes;  
}
```

## Association bidirectionnelle

- Maintenant mapper ce côté de l'association aussi, dans Reunion.hbm.xml.

```
<set name="personnes"  
    table="PERS_REUNIONS" inverse="true">  
    <key column="ID_REUNION" />  
    <many-to-many column="ID_PERSONNE"  
        class="gest.Personne" />  
</set>
```

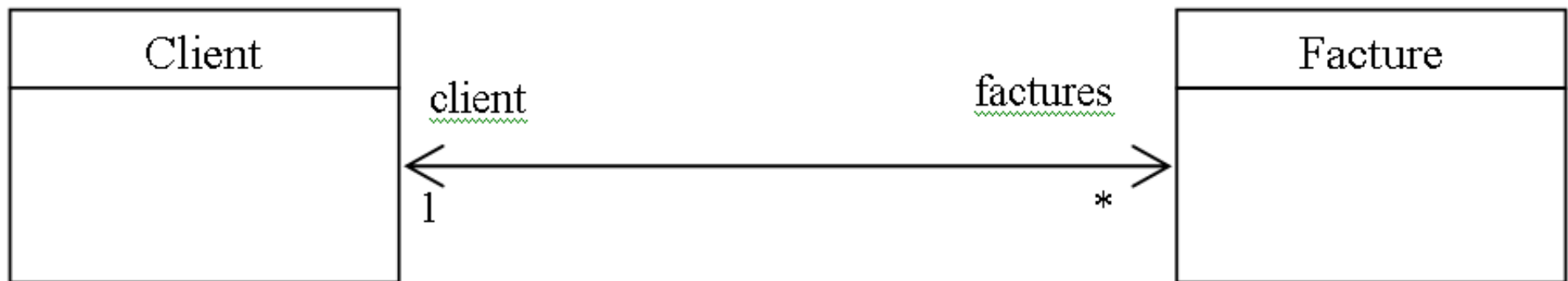
# Travailler avec des liens bidirectionnels

- Beaucoup de développeurs programment de manière défensive et créant des méthodes de gestion de lien pour affecter correctement les deux côtés, par exemple dans `Personne` :

```
protected Set getReunions() {  
    return reunions;  
}  
protected void setReunions(Set reunions) {  
    this.reunions = reunions;  
}  
public void addReunion(Reunion r) {  
    reunions.add(r);  
    r.getPersonnes().add(this);  
}  
public void removeReunion(Reunion r) {  
    reunions.remove(r);    r.getPersonnes().remove(this);  
}
```

# Association bidirectionnelle de type one-to-many et many-to-one

- Maintenant, nous allons nous intéresser à un type d'association très fréquente qui est un à plusieurs dans les deux sens.
- Prenons l'exemple d'un client qui possède plusieurs factures et chaque facture appartient à un client.



---

## Travailler avec des liens bidirectionnels

- Cette association se traduit par :
  - La création d'une collection d'objets Facture dans la classe Client
  - La création d'une référence (handle) à un objet Client dans la classe Facture.
- Cette manière de modéliser notre problème nous permet à un moment donné de connaître le client d'une facture donnée et les factures d'un client donné.

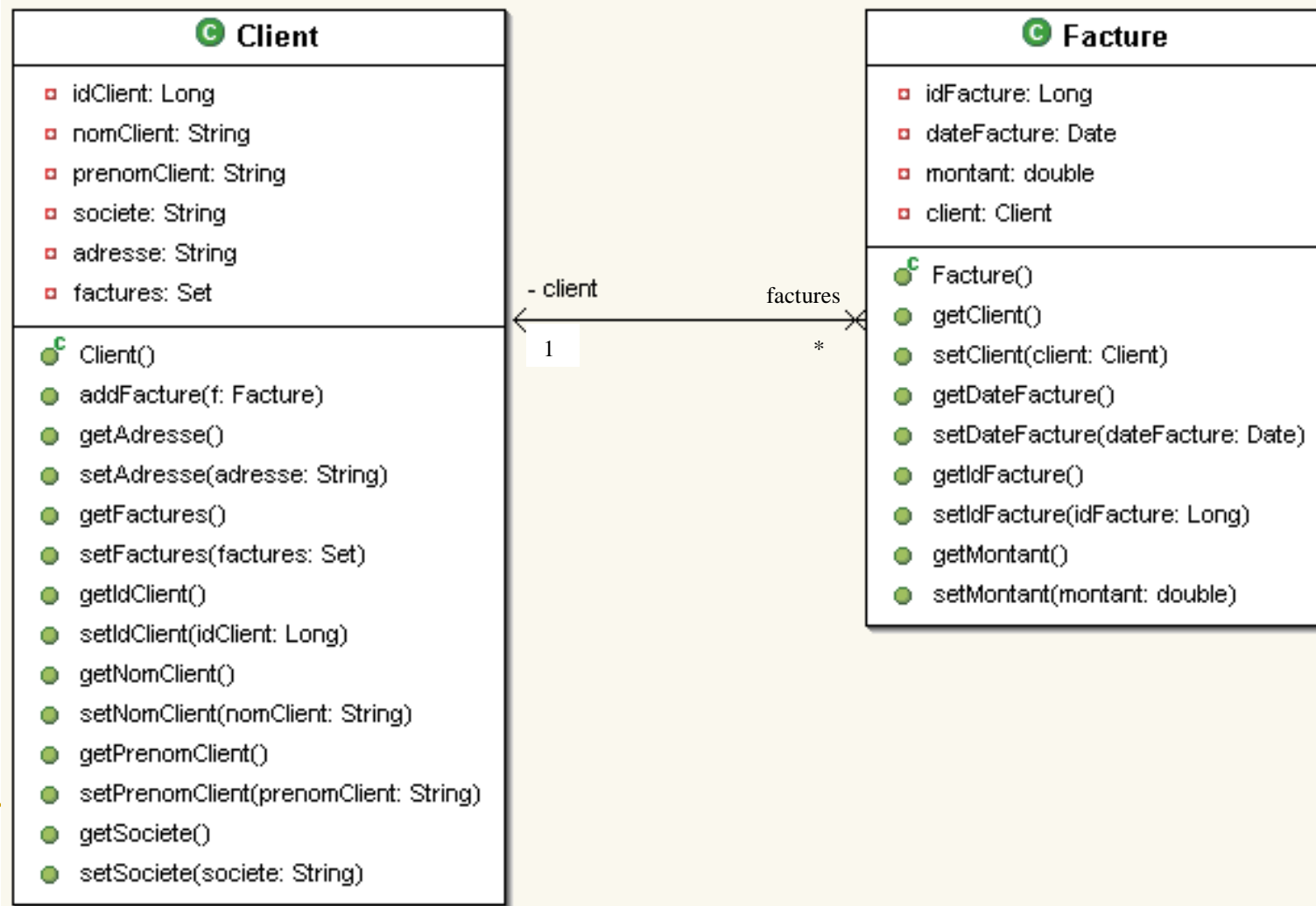


# Travailler avec des liens bidirectionnels

- Du côté de la base de données, ce genre d'association se traduit par :
  - ❑ la création de deux tables CLIENTS et FACTURES
  - ❑ avec d'une clé étrangère, qui représente le client, dans la table FACTURES.



# Diagramme de classes



# Client.java

```
package dao;
import java.util.*; import java.io.*;
public class Client implements Serializable {
    private Long idClient; private String nomClient;
    private String prenomClient; private String societe;
    private String adresse;
    private Set<Facture> factures=new HashSet<Facture>();

    // Constructeurs
    // Getters et Setters

    public void addFacture(Facture f){
        factures.add(f);
        f.setClient(this);
    }
}
```

# Facture.java

```
package dao;

import java.util.Date; import java.io.*;
public class Facture implements Serializable {
    private Long idFacture;
    private Date dateFacture;
    private double montant;
    private Client client;
    // Constructeurs
    // Getters et Setters
}
```

# Mapper la classe Client

```
<hibernate-mapping package="dao">
  <class name="Client" table="clients" >
    <id name="idClient" column="ID_CLIENT">
      <generator class="increment"/>
    </id>
    <property name="nomClient" column="NOM_CLIENT"/>
    <property name="prenomClient" column="PRENOM_CLIENT"/>
    <property name="societe" column="SOCIETE"/>
    <property name="adresse" column="ADRESSE"/>
    <set name="factures" inverse="true">
      <key column="ID_CLIENT"/>
      <one-to-many class="Facture"/>
    </set>
  </class>
</hibernate-mapping>
```

# Mapper la classe Facture

```
<hibernate-mapping package="dao">
  <class name="Facture">
    <id name="idFacture" column="ID_FACTURE">
      <generator class="increment"/>
    </id>
    <property name="dateFacture" column="DATE_FACTURE"/>
    <property name="montant" column="MONTANT"/>
    <many-to-one name="client" column="ID_CLIENT"/>
  </class>
</hibernate-mapping>
```

---

# En utilisant les annotations JPA

## Facture.java

```
package dao;
import java.io.*;import java.util.*;import
    javax.persistence.*;
    @Entity
public class Facture implements Serializable{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
private Long idFacture;
private Date dateFacture;
private double montant;
    @ManyToOne
    @JoinColumn(name="ID_CLI")
private Client client;
// Constructeurs
// Getters et Setters
```

---

# En utilisant les annotations JPA

## Client.java

```
package dao;
import java.io.*;import java.util.*;import
    javax.persistence.*;
@Entity
public class Client implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long idClient;
    private String nomClient;
    private String prenomClient; private String societe;
    private String adresse;
    @OneToMany(mappedBy="client",targetEntity=Facture.class,
        fetch=FetchType.LAZY)
    private Set<Facture> factures=new HashSet<Facture>();
}
```



---

# Ajouter un client

```
public void addClient(Client cli){
    Session session=
        HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        session.save(cli);
        session.getTransaction().commit();
}

public void addFacture(Facture fact,Long idClient){
    Session session=
        HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        Client cli=(Client)session.get(Client.class,idClient);
        cli.addFacture(fact);
        session.save(fact);
        session.getTransaction().commit();
}
```

---

# Mapping d'une Association OneToMany et de l'héritage

---

# Enoncé

- Un client peut avoir plusieurs comptes.
- Il existe deux types de compte : CompteCourant et CompteEpargne
- Chaque compte peut subir des opérations.
- Description des entités:
  - Un client est défini par son code, son nom et son adresse
  - Un compte est défini par son numéro, son solde et sa date de création
  - Un compte courant est un compte qui a la particularité d'avoir un découvert
  - Un compte épargne est un compte qui possède un taux d'intérêt.
  - Une opération est défini par un numéro, la date d'opération, le montant de versement et le montant de retrait.

# Application à réaliser

Créer une application qui permet de saisir le numéro de compte et en validant, afficher les informations sur ce compte, son propriétaire et la liste des opérations de ce compte

The screenshot shows a web browser window with the address bar displaying a URL. The page contains a form for entering an account number, followed by a display of account details and a table of transactions.

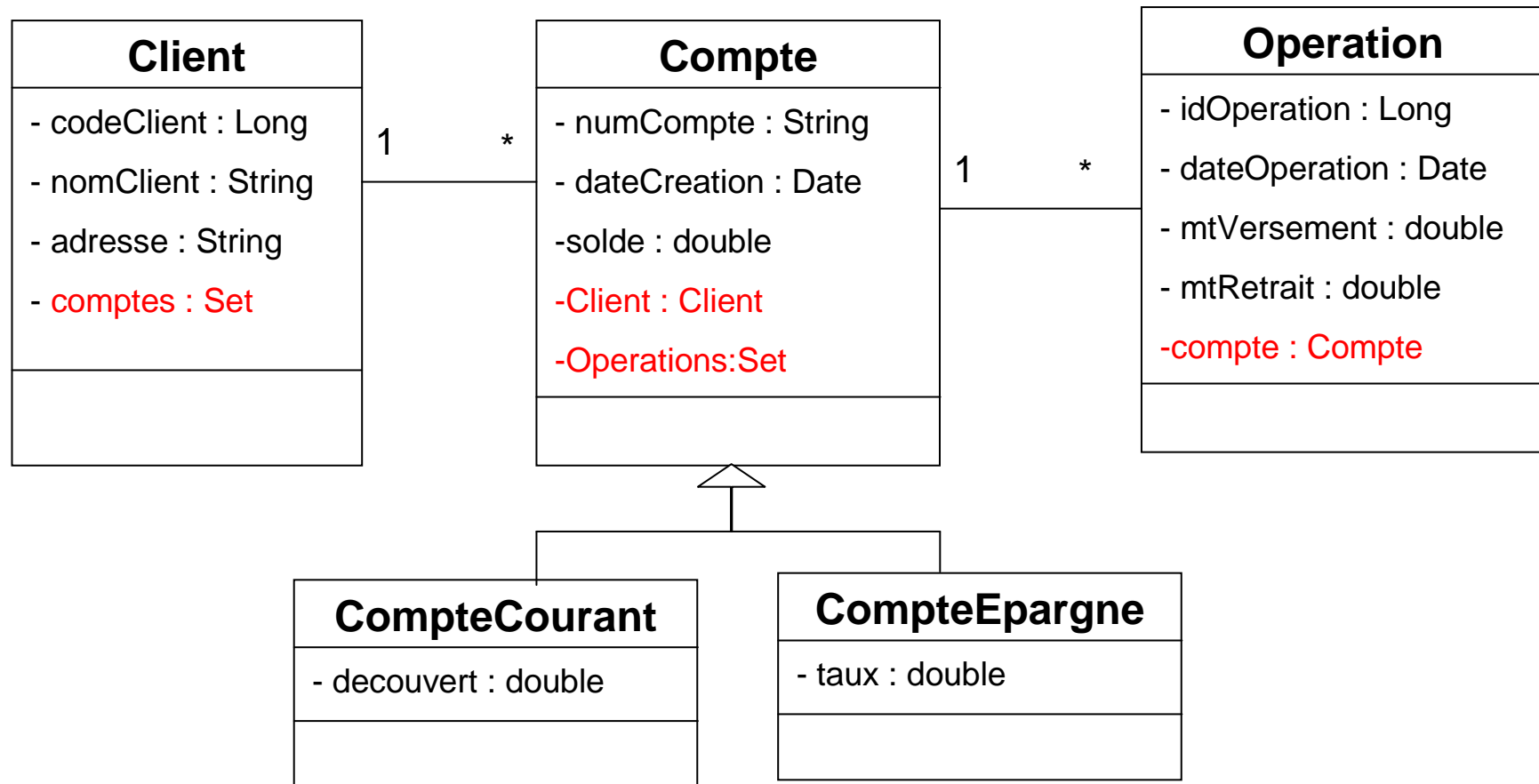
Num Compte:

Type de compte : CompteCourant  
Solde:13208,06  
Date création:2009-03-15 12:20:56.0  
Découvert:3978,28

Nom Client:Client1  
Adresse:Adresse1

NUM	Date	MTV	MTR
1	2009-03-15 12:20:56.0	4997.48565870688	0.0
2	2009-03-15 12:20:56.0	0.0	1969.01955462629
3	2009-03-15 12:20:56.0	1253.38586280278	0.0
4	2009-03-15 12:20:56.0	0.0	1309.33797361709

# Diagramme de classes et MLDR



## MLDR :

CLIENTS ( CODE\_CLIENT, NOM\_CLIENT, ADRESSE )

COMPTES ( NUM\_COMPTE, **TYPE\_CPTE**, DATE\_CREATION, SOLDE, DECOUVERT, TAUX **#CODE\_CLIENT** )

OPERATIONS ( ID\_OPERATION, DATE\_OPERATION, MT\_VERS, MT\_RETR, **#NUM\_COMPTE** )

---

# Mapper l'héritage

- Hibernate supporte les trois stratégies d'héritage de base :
  - une table par hiérarchie de classe
  - une table par classe concrète
  - une table pour la classe parente et une table pour chaque classe fille.

# Classe Client.java

```
package dao;
import java.util.HashSet; import java.util.Set;

public class Client {
    private Long codeClient;
    private String nom, adresse;
    private Set<Compte> comptes=new HashSet<Compte>();
    public Client() {
    }
    public Client(String nom, String adresse) {
        this.nom = nom;
        this.adresse = adresse;
    }
    public void addCompte(Compte c){
        comptes.add(c);
        c.setClient(this);
    }
    // Getters et Setters
}
```

# Compte.java

```
package dao;
import java.util.*;
public abstract class Compte {
    private String numCompte;private Date dateCreation;
    private double solde;private Client client;
    private Set<Operation> operations=new HashSet<Operation>();

    public Compte() {
    }
    public Compte(String numCompte,Date dateCreation,double solde)
    {
        this.numCompte = numCompte;
        this.dateCreation = dateCreation;
        this.solde = solde;
    }
    public void addOperation(Operation op){
        operations.add(op);
        op.setCompte(this);
    }
    // Getters et Setters
}
```



# Operation.java

```
package dao;
import java.util.Date;
public class Operation {
    private Long numOperation;
    private Date dateOperation;
    private double mtVersement,mtRetrait;
    private Compte compte;

    public Operation() {
    }
    public Operation(Date dateOperation, double
        mtVersement, double mtRetrait) {
        this.dateOperation = dateOperation;
        this.mtVersement = mtVersement;
        this.mtRetrait = mtRetrait;
    }
    // Getters et Setters
}
```

# CompteCourant.java

```
package dao;
import java.util.Date;
public class CompteCourant extends Compte {
    private double decouvert;

    public CompteCourant() {
        super();
    }

    public CompteCourant(String numCompte, Date
        dateCreation, double solde, double dec) {
        super(numCompte, dateCreation, solde);
        this.decouvert=dec;
    }
}
```

# CompteEpargne.java

```
package dao;
import java.util.Date;
public class CompteEpargne extends Compte {
    private double taux;

    public CompteEpargne() {
        super();
    }
    public CompteEpargne(String numCompte, Date
        dateCreation, double solde, double taux) {
        super(numCompte, dateCreation, solde);
        this.taux=taux;
    }
    //Getters et Setters
}
```

# Client.hbm.xml

```
<hibernate-mapping package="dao">
  <class name="Client" table="CLIENTS">
    <id name="codeClient" column="CODE_CLI">
      <generator class="native"></generator>
    </id>
    <property name="adresse"></property>
    <property name="nom"></property>
    <set name="comptes" inverse="true" lazy="true">
      <key column="CODE_CLI"></key>
      <one-to-many class="Compte" />
    </set>
  </class>
</hibernate-mapping>
```

# Compte.hbm.xml

```
<hibernate-mapping package="dao">
  <class name="Compte" table="COMPTE">
    <id name="numCompte" column="NUM_COMPTE"></id>
    <discriminator column="TYPE" type="string" length="2">
</discriminator>
    <property name="dateCreation"></property>
    <property name="solde"></property>
    <many-to-one name="client" column="CODE_CLI"></many-to-one>
    <set name="operations" inverse="true" lazy="true">
      <key column="NUM_COMPTE"></key>
      <one-to-many class="Operation"/>
    </set>
    <subclass name="CompteCourant" discriminator-value="CC">
      <property name="decouvert"></property>
    </subclass>
    <subclass name="CompteEpargne" discriminator-value="CE">
      <property name="taux"></property>
    </subclass>
  </class>
</hibernate-mapping>
```

# Operation.hbm.xml

```
<hibernate-mapping package="dao">
  <class name="Operation" table="OPERATIONS">
    <id name="numOperation" column="NUM_OP">
      <generator class="native"></generator>
    </id>
    <property name="dateOperation"></property>
    <property name="mtVersement"></property>
    <property name="mtRetrait"></property>
    <many-to-one name="compte"
column="NUM_COMPTE"></many-to-one>
  </class>
</hibernate-mapping>
```

# Gestion des entités

```
package dao;
import org.hibernate.Session;import util.HibernateUtil;
public class DaoImpl {
    /* AJOUTER UN NOUVEAU CLIENT */
    public void addClient(Client c){
        Session sess=
            HibernateUtil.getSessionFactory().getCurrentSession();
        sess.beginTransaction();sess.save(c);
        sess.getTransaction().commit();
    }
    /* AJOUTER UN NOUVEAU COMPTE D'UN CLIENT DONNE */
    public void addCompte(Compte cpte,Long idCli){
        Session sess=
            HibernateUtil.getSessionFactory().getCurrentSession();
        sess.beginTransaction();
        Client cli=(Client)sess.get(Client.class, idCli);
        cli.addCompte(cpte);sess.save(cpte);
        sess.getTransaction().commit();
    }
}
```

# Gestion des entités

```
/* AJOUTER UNE NOUVELLE OPERATION D'UN COMPTE DONNE */
public void addOperation(Operation op,String numCpte){
    Session sess=
        HibernateUtil.getSessionFactory().getCurrentSession();
    sess.beginTransaction();
    Compte cpte=(Compte)sess.get(Compte.class, numCpte);
    cpte.addOperation(op);sess.save(op);
    sess.getTransaction().commit();
}

/* CONSULTER UN COMPTE SACHANT SON CODE */
public Compte getCompte(String numCpte){
    Session sess=
        HibernateUtil.getSessionFactory().getCurrentSession();
    sess.beginTransaction();
    Object o=sess.get(Compte.class, numCpte);
    if(o==null) throw new RuntimeException("Compte
        introuvable");
    Compte cpte=(Compte)o;
    return cpte;
}
```



# Gestion des entités

```
/* CONSULTER UN CLIENT SACHANT SON CODE */
```

```
public Client getClient(Long idc){  
    Session sess=  
        HibernateUtil.getSessionFactory().getCurrentSession();  
    sess.beginTransaction();  
    Object o=sess.get(Client.class, idc);  
    if(o==null) throw new RuntimeException("Client introuvable");  
    Client cpte=(Client)o;  
    return cpte;  
  
}  
  
}
```

# Application de Test

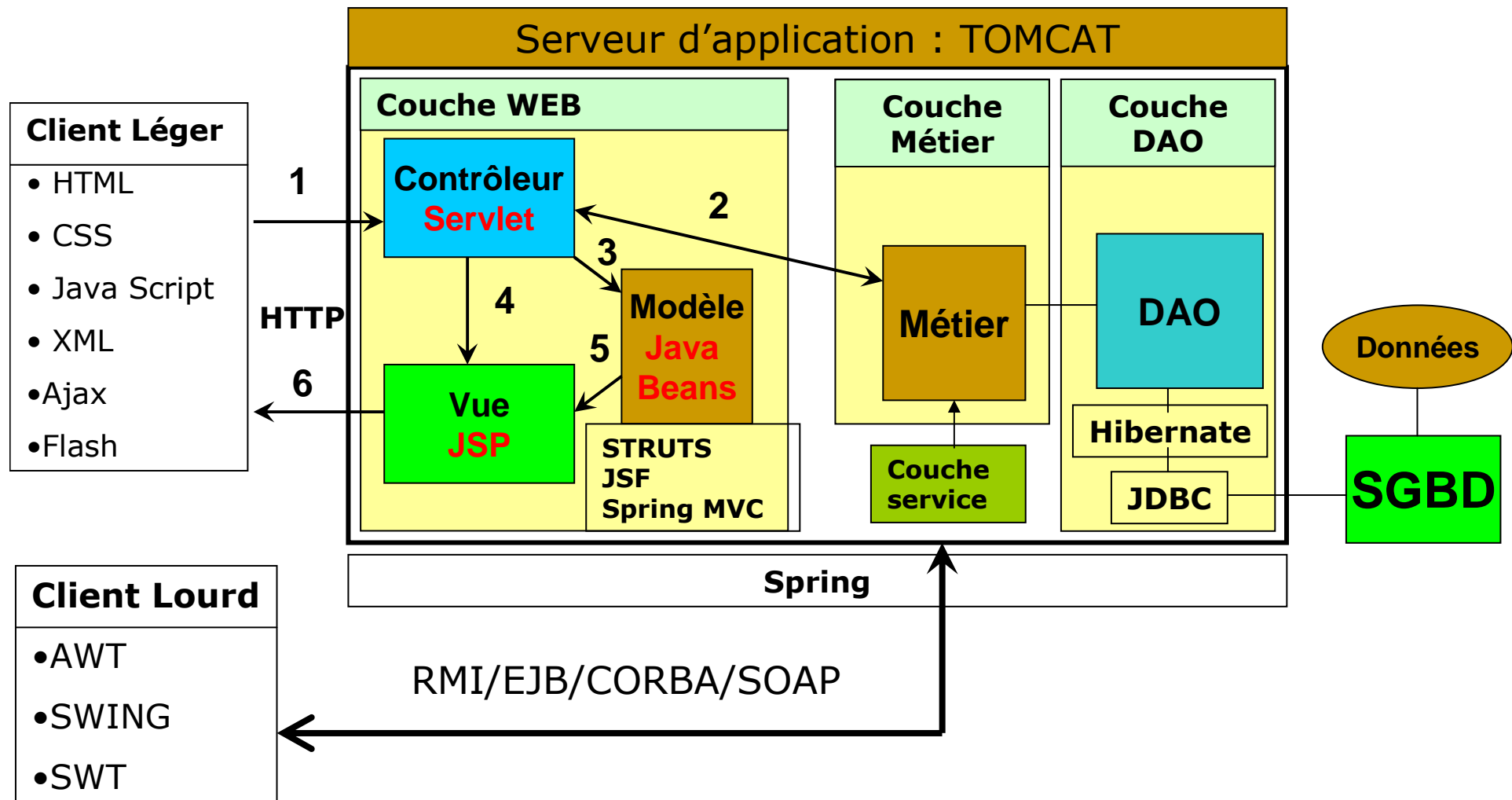
```
package dao;
import java.util.Date;
public class TestDao {
public static void main(String[] args) {
    DaoImpl dao=new DaoImpl();
    dao.addClient(new Client("CLI1","ADR1"));
    dao.addClient(new Client("CLI2","ADR2"));
    dao.addCompte(new CompteCourant("CC1",new Date(),5000,8000),1L );
    dao.addCompte(new CompteEpargne("CE1",new Date(),8000,5.5),1L );
    dao.addOperation(new Operation(new Date(),4000, 0),"CC1");
    dao.addOperation(new Operation(new Date(),0, 7000),"CC1");
    try{
        Compte c=dao.getCompte("CC1");
        System.out.println("Solde:"+c.getSolde());
        System.out.println("Type de Compte:"+c.getClass().getSimpleName());
        System.out.println("Nom Client:"+c.getClient().getNom());
        Iterator<Operation> ops=c.getOperations().iterator();
        while(ops.hasNext()){
            Operation op=ops.next();
            System.out.println(op.getMtVersement()+"--"+op.getMtRetrait());
        }
    }
}
catch (Exception e) { System.out.println(e.getMessage());
}}}
```

---

# Spring FrameWork

M.Youssfi

# Architecture J2EE



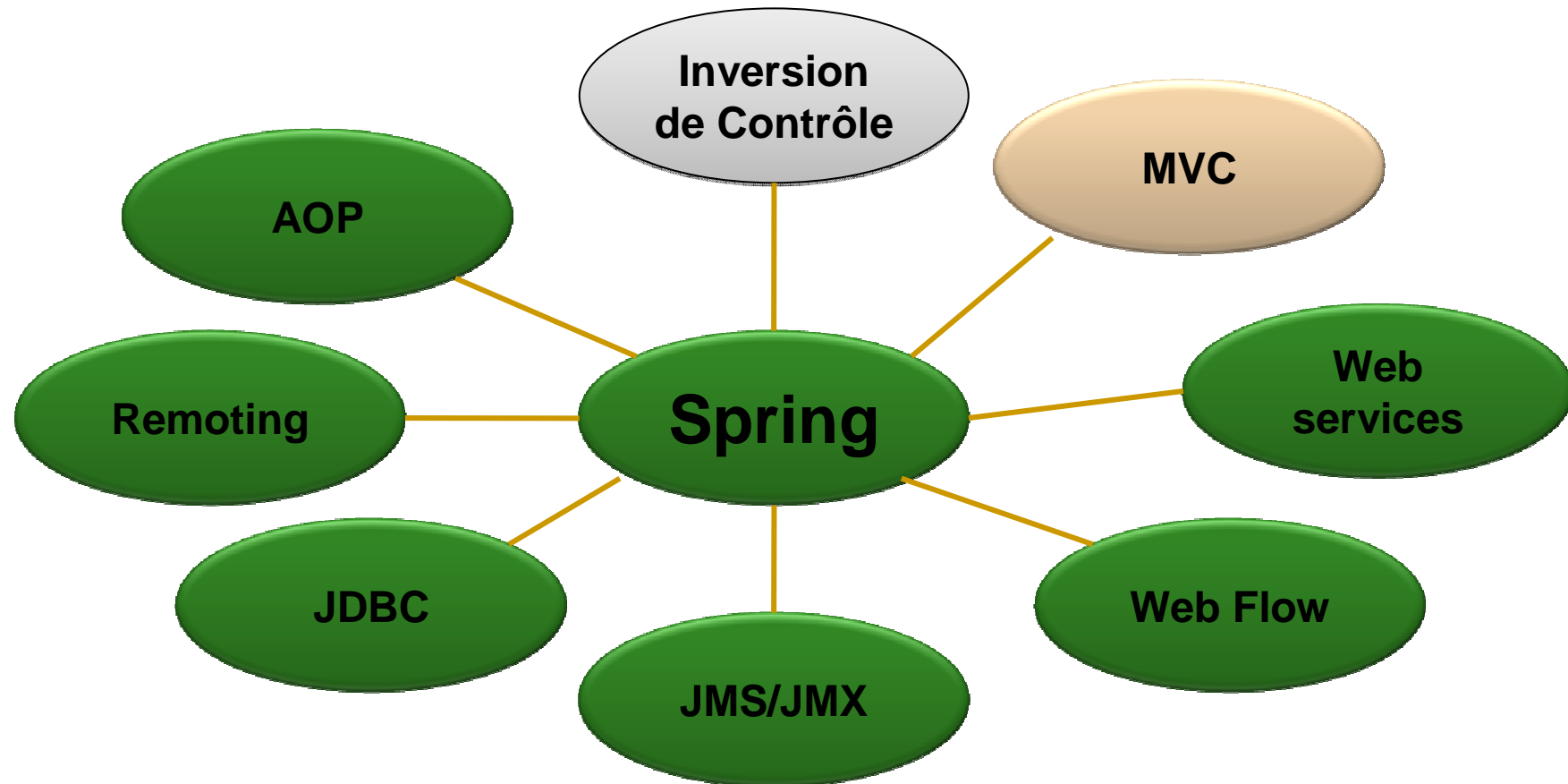
# Architecture d'une application

- La couche [**dao**] s'occupe de l'accès aux données, le plus souvent des données persistantes au sein d'un SGBD. Mais cela peut être aussi des données qui proviennent de capteurs, du réseau, ...
- La couche [**metier**] implémente les algorithmes " métier " de l'application. Cette couche est indépendante de toute forme d'interface avec l'utilisateur.
  - C'est généralement la couche la plus stable de l'architecture.
  - Elle ne change pas si on change l'interface utilisateur ou la façon d'accéder aux données nécessaires au fonctionnement de l'application.
- La couche [**Présentation**] qui est l'interface (graphique souvent) qui permet à l'utilisateur de piloter l'application et d'en recevoir des informations.

---

# Spring FrameWork

# Projets Spring



# Rôle de Spring

- Spring est un framework qui peut intervenir dans les différentes parties d'un projet J2EE :
  - ❑ Spring IOC: permet d'assurer l'injection des dépendances entre les différentes couches de votre application (Inversion du contrôle), de façon à avoir un faible couplage entre les différentes parties de votre application.
  - ❑ Spring MVC: Implémenter le modèle MVC d'une application web au même titre que STRUTS et JSF
  - ❑ Spring AOP: permet de faire la programmation orientée Aspect.
  - ❑ Peut être utilisé au niveau de la couche métier pour la gestion des transactions.
  - ❑ Spring JDBC: Peut être exploiter au niveau de la couche d'accès aux bases de données
  - ❑ Spring Remoting : offre un support qui permet de faciliter l'accès à un objet distant RMI ou EJB.
  - ❑ Spring Web Services : permet de faciliter la création et l'accès aux web services
- ❑ Etc...



---

# Spring IOC

## Inversion de Contrôle

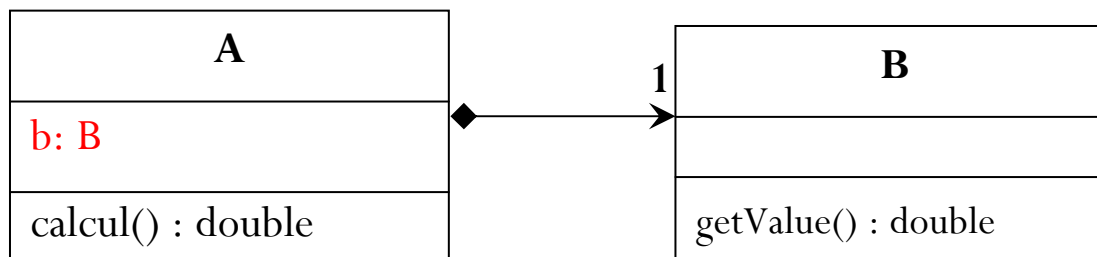
---

# Rappels de quelque principes de conception

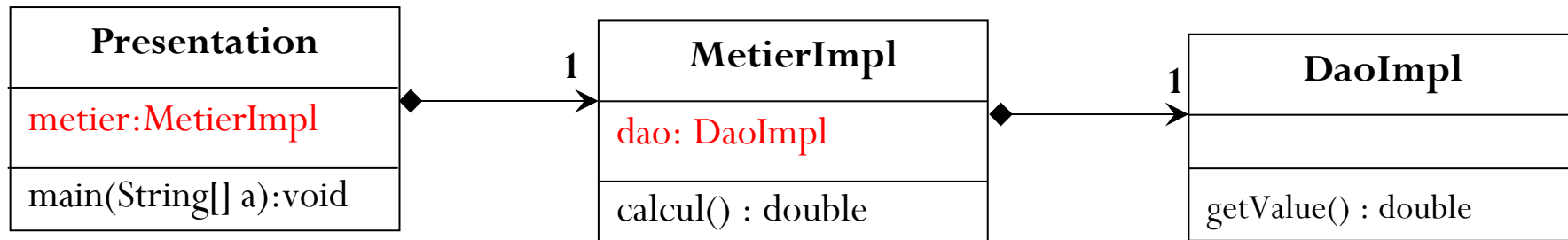
- Une application qui n'évolue pas meurt.
- Une application doit être fermée à la modification et ouverte à l'extension.
- Une application doit s'adapter aux changements
- Efforcez-vous à coupler faiblement vos classes.
- Programmer une interface et non une implémentation
- Etc..

# Couplage fort

- Quand une classe A est liée à une classe B, on dit que la classe A est fortement couplée à la classe B.
- La classe A ne peut fonctionner qu'en présence de la classe B.
- Si une nouvelle version de la classe B (soit B2), est créée, on est obligé de modifier dans la classe A.
- Modifier une classe implique:
  - ❑ Il faut disposer du code source.
  - ❑ Il faut recompiler, déployer et distribuer la nouvelle application aux clients.
  - ❑ Ce qui engendre un cauchemar au niveau de la maintenance de l'application



# Exemple de couplage fort



```

package metier;
import dao.DaoImpl;
public class MetierImpl {
    private DaoImpl dao;
    public MetierImpl() {
        dao=new DaoImpl();
    }
    public double calcul(){
        double nb=dao.getValue();
        return 2*nb;
    }
}
  
```

```

package dao;
public class DaoImpl {
    public double getValue(){
        return(5);
    }
}
  
```

```

package pres;
import metier.MetierImpl;
public class Presentation {
    private static MetierImpl metier;
    public static void main(String[]
        args) {
        metier=new MetierImpl();
        System.out.println(metier.calcul());
    }
}
  
```

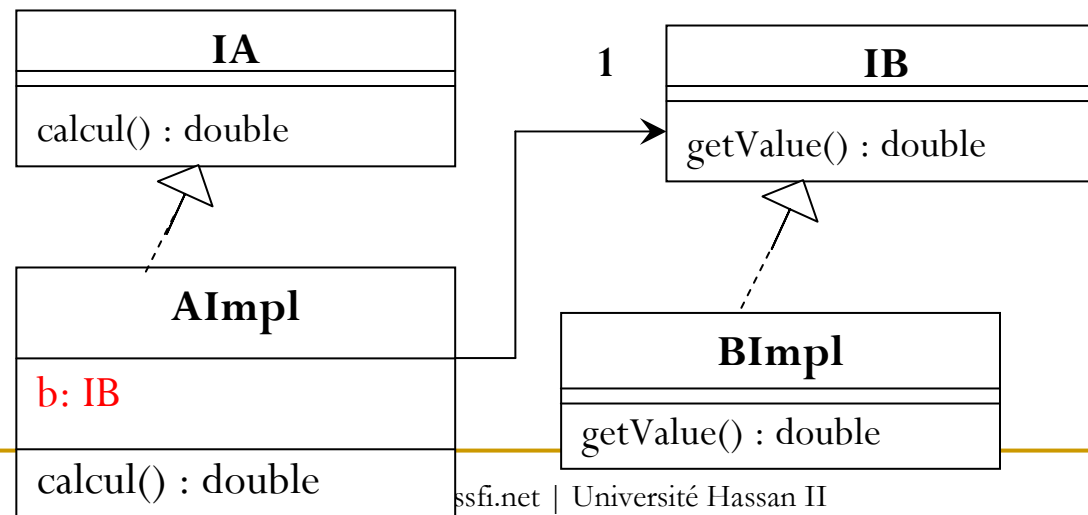
---

# Problèmes du couplage fort

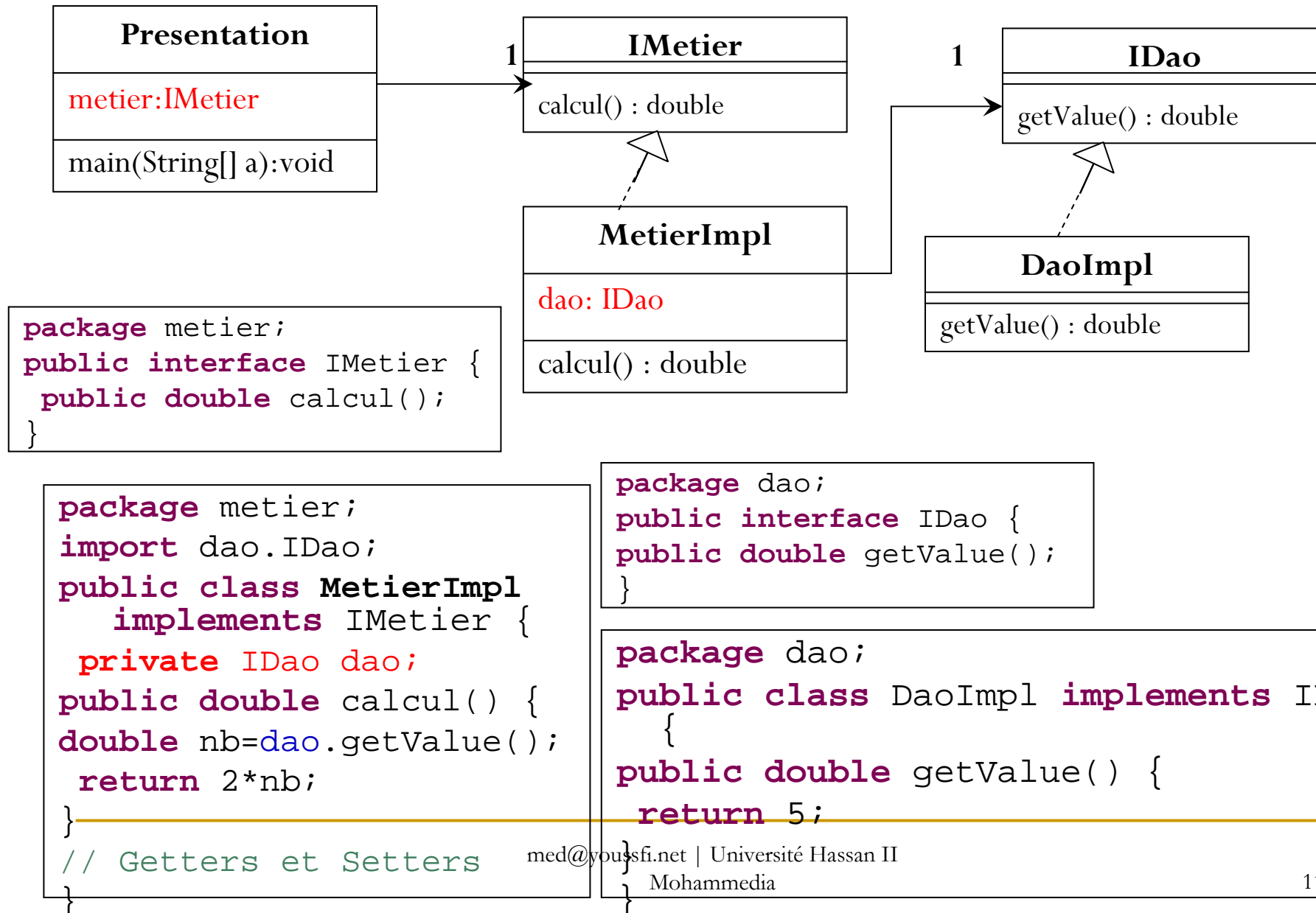
- Dans l'exemple précédent, les classes MetierImpl et DaoImpl sont liées par un couplage fort. De même pour les classe Presentation et MetierImpl
- Ce couplage fort n'a pas empêché de résoudre le problème au niveau fonctionnel.
- Mais cette conception nous ne a pas permis de créer une application fermée à la modification et ouverte à l'extension.
- En effet, la création d'une nouvelle version de la méthode getValue() de la classe DaoImpl, va nous obliger d'éditer le code source de l'application aussi bien au niveau de DaoImpl et aussi MetierImpl.
- De ce fait nous avons violé le principe « une application doit être fermée à la modification et ouverte à l'exetension »
- Nous allons voir que nous pourrons faire mieux en utilisant le couplage faible.

# Couplage Faible.

- Pour utiliser le couplage faible, nous devons utiliser les interfaces.
- Considérons une classe A qui implémente une interface IA, et une classe B qui implémente une interface IB.
- Si la classe A est liée à l'interface IB par une association, on dit que le classe A et la classe B sont liées par un couplage faible.
- Cela signifie que la classe B peut fonctionner avec n'importe quelle classe qui implémente l'interface IA.
- En effet la classe B ne connaît que l'interface IA. De ce fait n'importe quelle classe implémentant cette interface peut être associée à la classe B, sans qu'il soit nécessaire de modifier quoi que se soit dans la classe B.
- Avec le couplage faible, nous pourrions créer des application fermée à la modification et ouvertes à l'extension.



# Exemple de coupage faible



# Injection des dépendances

- Fichier texte de configuration : config.txt  
ext.DaoImp  
metier.MetierImpl
- Injection par instantiation statique :

```
import metier.MetierImpl;  
import dao.DaoImpl;  
public class Presentation {  
public static void main(String[] args) {  
    DaoImpl dao=new DaoImpl();  
    MetierImpl metier=new MetierImpl();  
    metier.setDao(dao);  
    System.out.println(metier.calcul());  
}  
}
```



# Injection des dépendances

- Injection par instanciation dynamique par réflexion :

```
import java.io.*;import java.lang.reflect.*;
import java.util.Scanner; import metier.IMetier;
import dao.IDao;
public class Presentation {
public static void main(String[] args) {
try {
Scanner scanner=new Scanner(new File("config.text"));
String daoClassname=scanner.next();
String metierClassName=scanner.next();
Class cdao=Class.forName(daoClassname);
IDao dao= (IDao) cdao.newInstance();
Class cmetier=Class.forName(metierClassName);
IMetier metier=(IMetier) cmetier.newInstance();
Method meth=cmetier.getMethod("setDao",new Class[]{IDao.class});
meth.invoke(metier, new Object[]{dao});
System.out.println(metier.calcul());
} catch (Exception e) { e.printStackTrace(); }
}
}
```

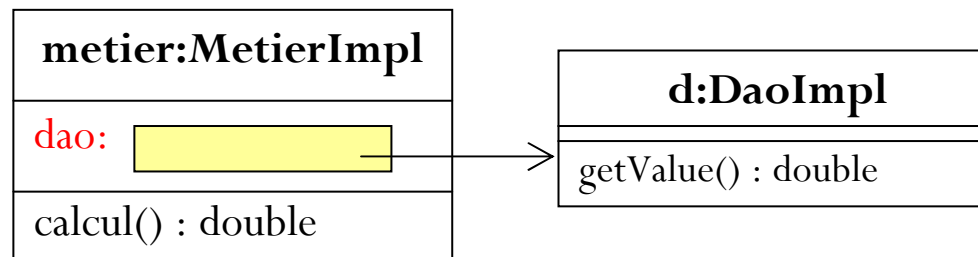
---

# Injection des dépendances avec Spring.

- L'injection des dépendance, ou l'inversion de contrôle est un concept qui intervient généralement au début de l'exécution de l'application.
- Spring IOC commence par lire un fichier XML qui déclare quelles sont différentes classes à instancier et d'assurer les dépendances entre les différentes instances.
- Quand on a besoin d'intégrer une nouvelle implémentation à une application, il suffirait de la déclarer dans le fichier xml de beans spring.

# Injection des dépendances dans une application java standard

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-
    2.0.dtd" >
<beans>
  <bean id="d" class="dao.DaoImpl2"></bean>
  <bean id="metier" class="metier.MetierImpl">
    <property name="dao" ref="d"></property>
  </bean>
</beans>
```

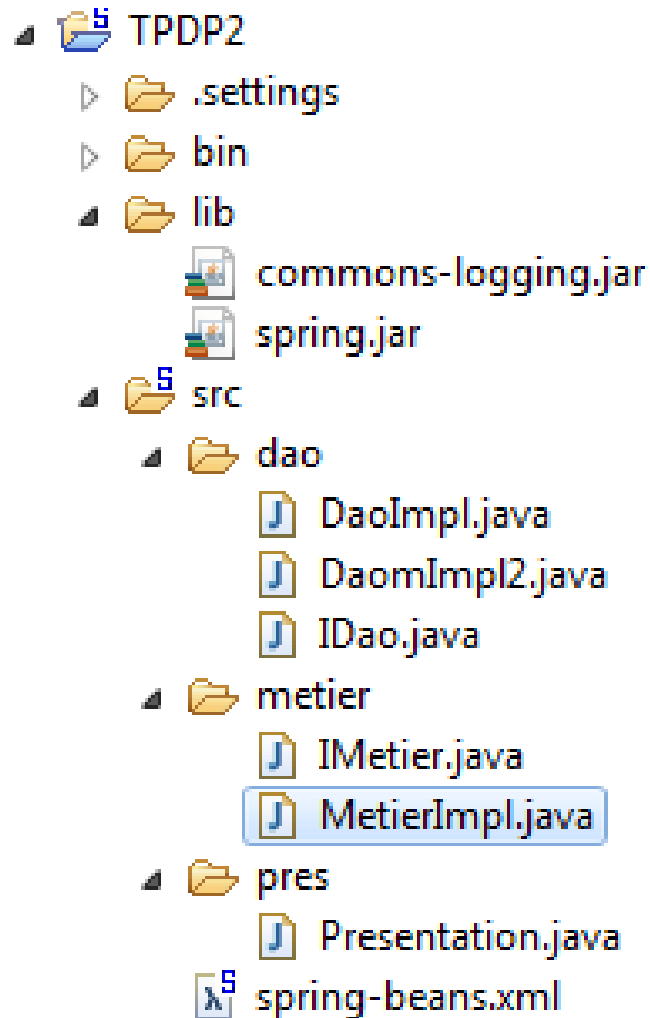


# Injection des dépendances dans une application java standard

```
package pres;
import metier.IMetier;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
public class Presentation {
public static void main(String[] args) {
    XmlBeanFactory bf=new XmlBeanFactory(new
        ClassPathResource("spring-beans.xml"));
    IMetier m=(IMetier) bf.getBean("metier");
    System.out.println(m.calcul());
}
}
```

Pour une application java classique, le fichier « spring-beans.xml » devrait être enregistré dans la racine du classpath. C'est-à-dire le dossier src.

# Structure du projet

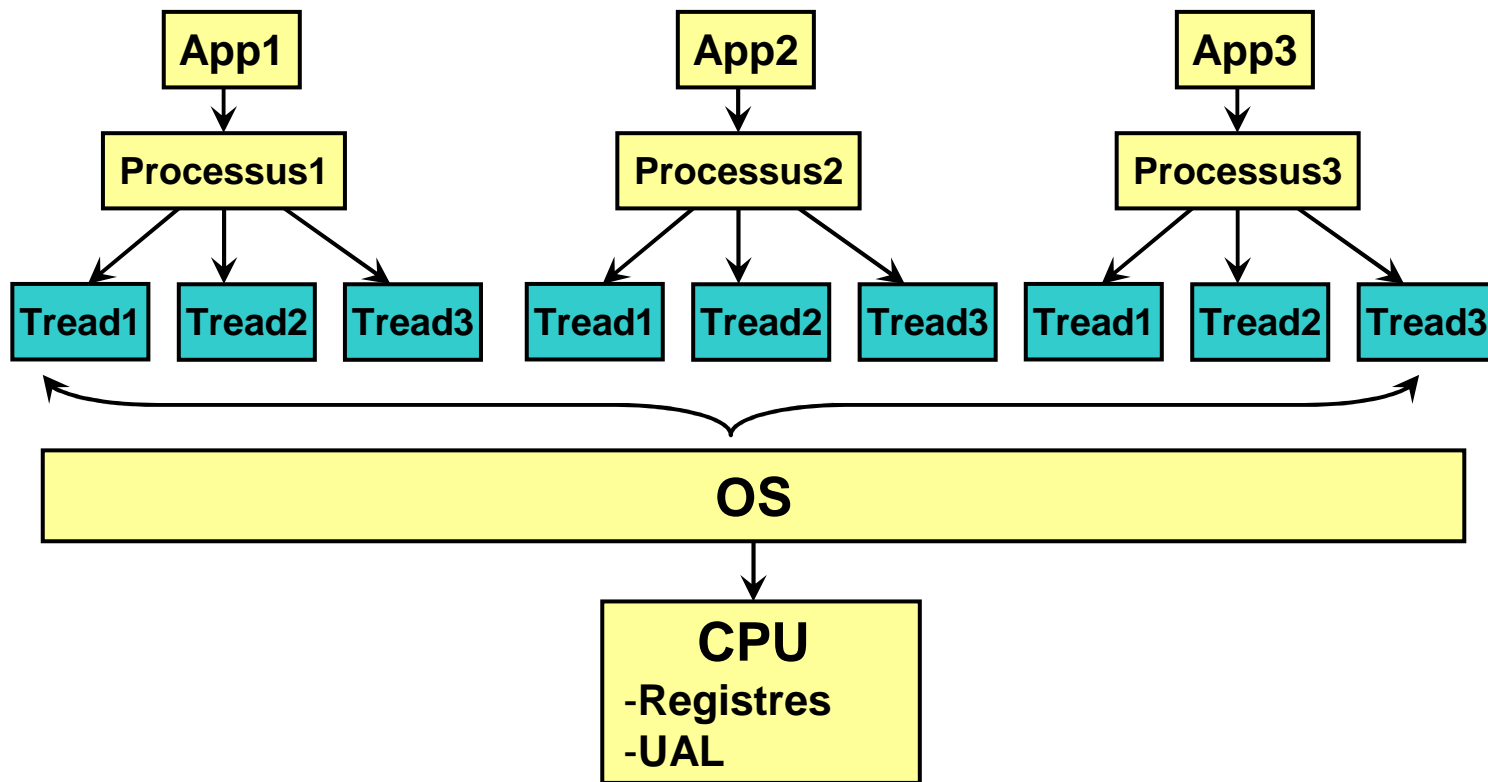


# Les Threads en Java

M.Youssfi

# Qu'est ce qu'un thread

- Java est un langage multi-threads
- Il permet d'exécuter plusieurs blocs d'instructions à la fois.
- Dans ce cas, chaque bloc est appelé Thread ( tâche ou fil )



---

# Création et démarrage d'un thread

En Java, il existe, deux techniques pour créer un thread:

- Soit vous dérivez votre thread de la classe `java.lang.Thread`,
- soit vous implémentez l'interface `java.lang.Runnable`.

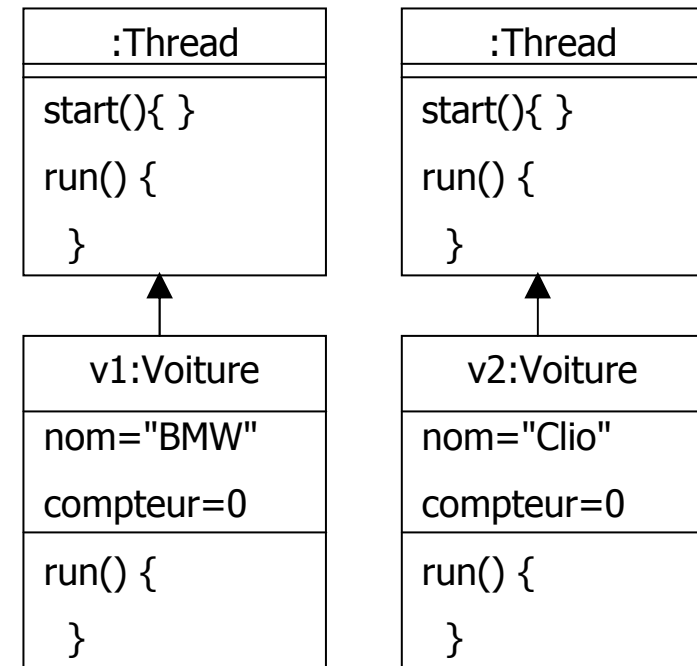


# Créer un thread en dérivant de la classe java.lang.Thread.

```
public class ThreadClass extends Thread {
    // Les attributs de la classe
    // Les constructeurs de la classe
    // Méthode de la classe
    @Override
    public void run() {
        // Code exécuté par le thread
    }
    public static void main(String[] args) {
        ThreadClass t1=new ThreadClass();
        ThreadClass t2=new ThreadClass();
        t1.start();t2.start();
    }
}
```

# Exemple de classe héritant de Thread

```
public class Voiture extends Thread {  
    private String nom;  
    private int compteur;  
    public Voiture(String nom) {  
        this.nom=nom;  
    }  
    public void run(){  
        try{  
for(int i=0;i<10;i++){  
            ++compteur;  
            System.out.println("Voiture:"+nom+"  
I="+i+"Compteur="+compteur);  
            Thread.sleep(1000);  
        }}catch(InterruptedException e){  
            e.printStackTrace(); }  
        }  
    public static void main(String[] args){  
        Voiture v1=new Voiture("BMW");  
        Voiture v2=new Voiture("Clio");  
        v1.start();  
        v2.start();  
    }  
}
```



## Exécution :

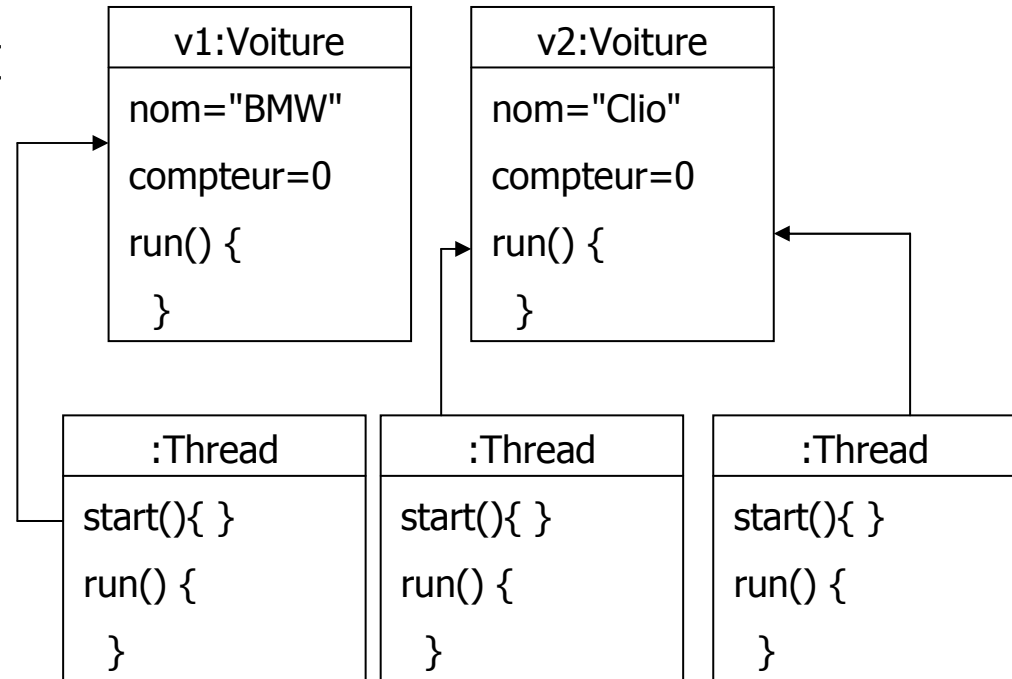
Voiture BMW I=0 Compteur=1  
Voiture Clio I=0 Compteur=1  
Voiture BMW I=1 Compteur=2  
Voiture Clio I=1 Compteur=2  
Voiture BMW I=2 Compteur=3  
Voiture Clio I=2 Compteur=3

# Créer un thread en implémentant l'interface java.lang.Runnable.

```
public class ThreadClass implements Runnable {
    // Les attributs de la classe
    // Les constructeurs de la classe
    // Méthode de la classe
    @Override
    public void run() {
        // Code exécuté par le thread
    }
    public static void main(String[] args) {
        ThreadClass t1=new ThreadClass();
        ThreadClass t2=new ThreadClass();
        new Thread(t1).start();new Thread(t2).start();
    }
}
```

# Exemple de classe implémentant Runnable

```
public class Voiture implements Runnable {
    private String nom;
    private int compteur;
    public Voiture(String nom) {
        this.nom=nom;
    }
    public void run(){
        try{
            for(int i=0;i<10;i++){
                ++compteur;
                System.out.println("Voiture:"+nom+"
I="+i+"Compteur="+compteur);
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
    }
    public static void main(String[]largs){
        Voiture v1=new Voiture("BMW");
        Voiture v2=new Voiture("Clio");
        new Thread(v1).start();
        new Thread(v2).start();
        new Thread(v2).start();
    }
}
```



## Exécution :

Voiture BMW I=0 Compteur=1  
Voiture Clio I=0 Compteur=1  
Voiture Clio I=0 Compteur=2  
Voiture BMW I=1 Compteur=2  
Voiture Clio I=1 Compteur=3  
Voiture Clio I=1 Compteur=4

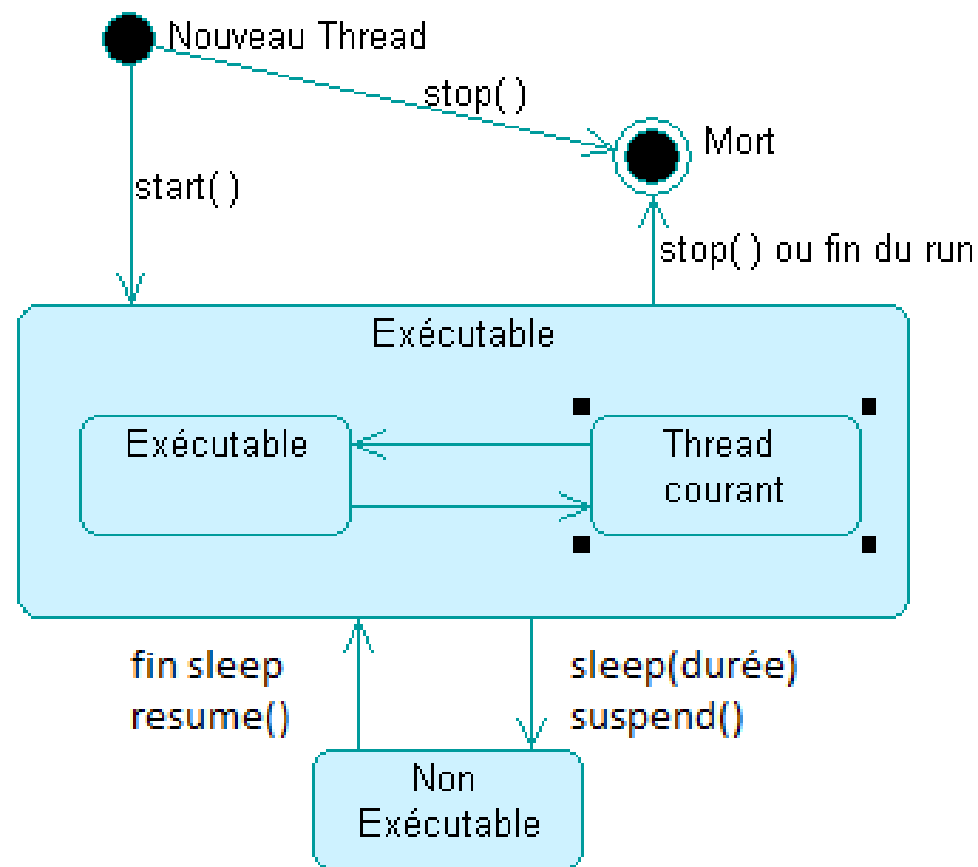
.....;

# Quelle technique choisir ?

Méthode	<i>Avantages</i>	<i>Inconvénients</i>
<b>extends Thread</b>	Chaque thread a ses données qui lui sont propres.	On ne peut plus hériter d'une autre classe.
<b>implements Runnable</b>	L'héritage reste possible. En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs de votre classe sont partagés pour tous les threads qui y sont basés. Dans certains cas, il peut s'avérer que cela soit un atout.

# Gestion des threads

## ■ *Cycle de vie d'un thread*



# Démarrage, suspension, reprise et arrêt d'un thread.

- **public void start()** : cette méthode permet de démarrer un thread. En effet, si vous invoquez la méthode *run* (au lieu de *start*) le code s'exécute bien, mais aucun nouveau thread n'est lancé dans le système. Inversement, la méthode *start*, lance un nouveau thread dans le système dont le code à exécuter démarre par le *run*.
- **public void suspend()** : cette méthode permet d'arrêter temporairement un thread en cours d'exécution.
- **public void resume()** : celle-ci permet de relancer l'exécution d'un thread, au préalable mis en pause via *suspend*. Attention, le thread ne reprend pas au début du *run*, mais continue bien là où il s'était arrêté.
- **public void stop()** : cette dernière méthode permet de stopper, de manière définitive, l'exécution du thread. Une autre solution pour stopper un thread consiste à simplement sortir de la méthode *run*.

# Gestion de la priorité d'un thread.

- Vous pouvez, en Java, jouer sur la priorité de vos threads.
- Sur une durée déterminée, un thread ayant une priorité plus haute recevra plus fréquemment le processeur qu'un autre thread. Il exécutera donc, globalement, plus de code.
- La priorité d'un thread va pouvoir varier entre 0 et 10. Mais attention, il n'est en aucun cas garanti que le système hôte saura gérer autant de niveaux de priorités.
- Des constantes existent et permettent d'accéder à certains niveaux de priorités : `MIN_PRIORITY` (0) - `NORM_PRIORITY` (5) - `MAX_PRIORITY` (10).
- Pour spécifier la priorité d'un thread, il faut faire appel à la méthode `setPriority(int p)` de la classe `Thread`
- Exemple :
  - ❑ `Thread t=new Thread();`
  - ❑ `t.setPriority(Thread.MAX_PRIORITY);`



---

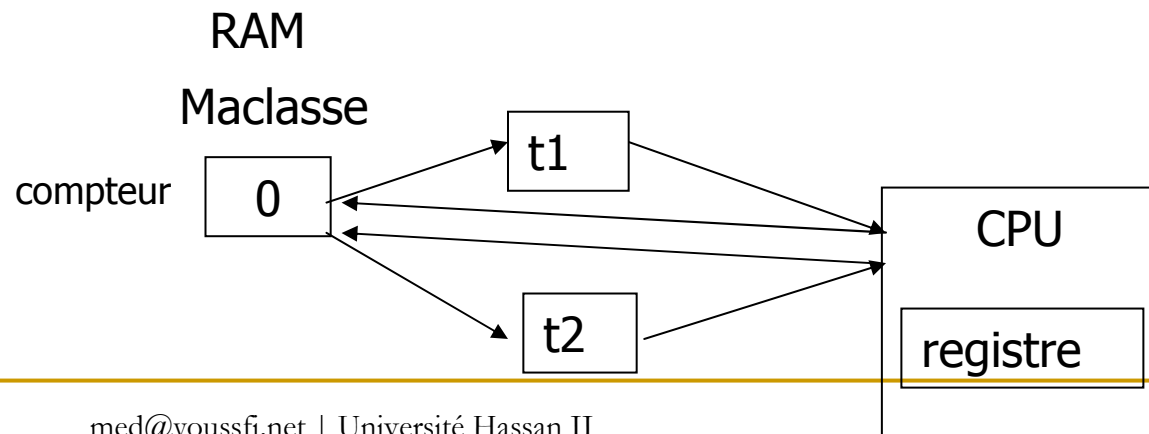
# Synchronisation de threads et accès aux ressources partagées.

- Lorsque que vous lancez une JVM (Machine Virtuelle Java), vous lancez un processus.
- Ce processus possède plusieurs threads et chacun d'entre eux partage le même espace mémoire.
- En effet, l'espace mémoire est propre au processus et non à un thread.
- Cette caractéristique est à la fois un atout et une contrainte.
- En effet, partager des données pour plusieurs threads est relativement simple.
- Par contre les choses peuvent se compliquer sérieusement si la ressource (les données) partagée est accédée en modification
- il faut synchroniser les accès concurrents. Nous sommes certainement face à l'un des problèmes informatiques les plus délicats à résoudre.

# Synchronisation de threads et accès aux ressources partagées.

- Soit l'exemple suivant:

```
public class MaClasse extends Thread{  
    private static int compteur;  
    public void run(){  
        compteur=compteur+1;  
        System.out.println(compteur);  
    }  
    public static void main(String[] Args){  
        MaClasse t1=new MaClasse();  
        MaClasse t2=new MaClasse();  
        t1.start(); t2.start();  
    }  
}
```



# Synchronisation de threads et accès aux ressources partagées.

- Imaginez qu'un premier thread évalue l'expression *MaClasse.Compteur + 1* mais que le système lui hôte le cpu, juste avant l'affectation, au profit d'un second thread.
- Ce dernier se doit lui aussi d'évaluer la même expression. Le système redonne la main au premier thread qui finalise l'instruction en effectuant l'affectation, puis le second en fait de même.
- Au final de ce scénario, l'entier aura été incrémenté que d'une seule et unique unité.
- De tels scénarios peuvent amener à des comportements d'applications chaotiques.
- Il est donc vital d'avoir à notre disposition des mécanismes de synchronisation.

# Notions de verrous

- L'environnement Java offre un premier mécanisme de synchronisation : les verrous (locks en anglais).
- Chaque objet Java possède un verrou et seul un thread à la fois peut verrouiller un objet.
- Si d'autres threads cherchent à verrouiller le même objet, ils seront endormis jusqu'à que l'objet soit déverrouillé.
- Cela permet de mettre en place ce que l'on appelle plus communément une section critique.
- Pour verrouiller un objet par un thread, il faut utiliser le mot clé **synchronized**.
- En fait, il y a deux façons de définir une section critique. Soit on synchronise un ensemble d'instructions sur un objet, soit on synchronise directement l'exécution d'une méthode pour une classe donnée.
- Dans le premier cas, on utilise l'instruction *synchronized* :  

```
synchronized(object) {  
    //Instructions de manipulation d'une ressource partagée.  
}
```
- Dans le second cas, on utilise le qualificateur *synchronized* sur la méthode considérée:  

```
public synchronized void meth(int param) {  
// Le code de la méthode synchronisée.}
```

## Exemple : Synchroniser les threads au moment de l'incrément de la variable compteur

```
public class Voiture implements Runnable {
    private String nom;
    private int compteur;
    public Voiture(String nom) {
        this.nom=nom;
    }
    public void run(){
        try{
            for(int i=0;i<10;i++){
                synchronized (this) {
                    ++compteur;
                }
            }
            System.out.println("Voiture:"+nom+" I="+i+"Compteur="+compteur);
            Thread.sleep(1000);
        }catch(InterruptedException e){
            e.printStackTrace(); }
    }
    public static void main(String[]largs){
        Voiture v1=new Voiture("BMW"); Voiture v2=new Voiture("Clio");
        new Thread(v1).start();new Thread(v2).start();
        new Thread(v2).start();
    }
}
```

## Exemple : Synchroniser les threads avant l'accès à la méthode run

```
public class Voiture implements Runnable {
    private String nom;
    private int compteur;
    public Voiture(String nom) {
        this.nom=nom;
    }

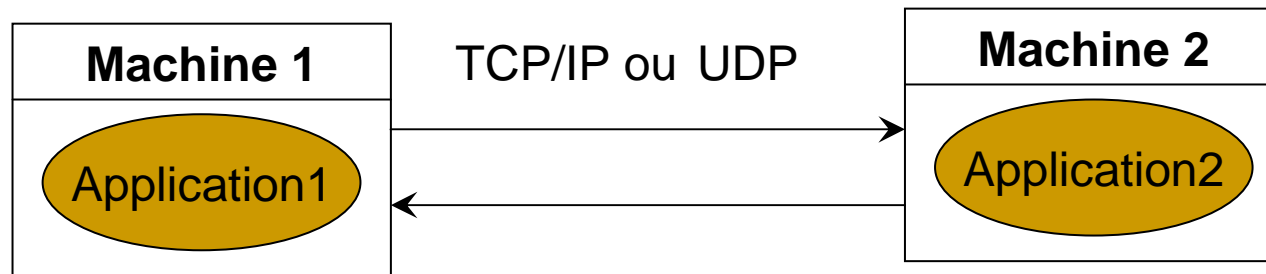
    public synchronized void run(){
        try{
            for(int i=0;i<10;i++){
                ++compteur;
                System.out.println("Voiture:"+nom+" I="+i+"Compteur="+compteur);
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace(); }
        }

        public static void main(String[] args){
            Voiture v1=new Voiture("BMW"); Voiture v2=new Voiture("Clio");
            new Thread(v1).start();new Thread(v2).start();
            new Thread(v2).start();
        }
    }
}
```

# Sockets dans java

Par M.Youssfi

# Applications distribuées



## Technologies d'accès :

- Sockets ou DataGram
- RMI (JAVA)
- CORBA (Multi Langages)
- EJB (J2EE)
- Web Services (HTTP+XML)

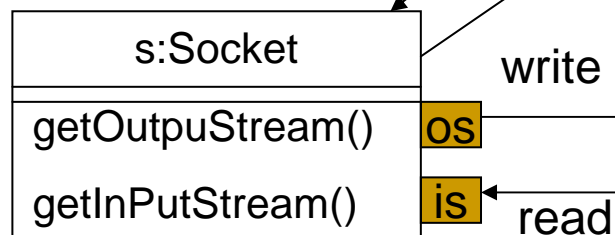


# Principe de base

## Création d'un client :

```
Socket s=new Socket("192.168.1.23",1234)
```

```
InputStream is=s.getInputStream();  
OutputStream os=s.getOutputStream();  
os.write(23);  
int rep=is.read();  
System.out.println(rep);
```



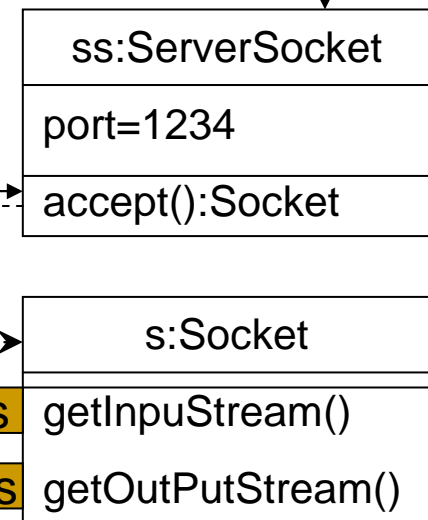
## Création d'un serveur:

```
ServerSocket ss=new ServerSocket(1234);
```

```
Socket s=ss.accept();
```

```
InputStream is=s.getInputStream();  
OutputStream os=s.getOutputStream();  
int nb=is.read();  
int rep=nb*2;  
os.write(rep);
```

Connexion



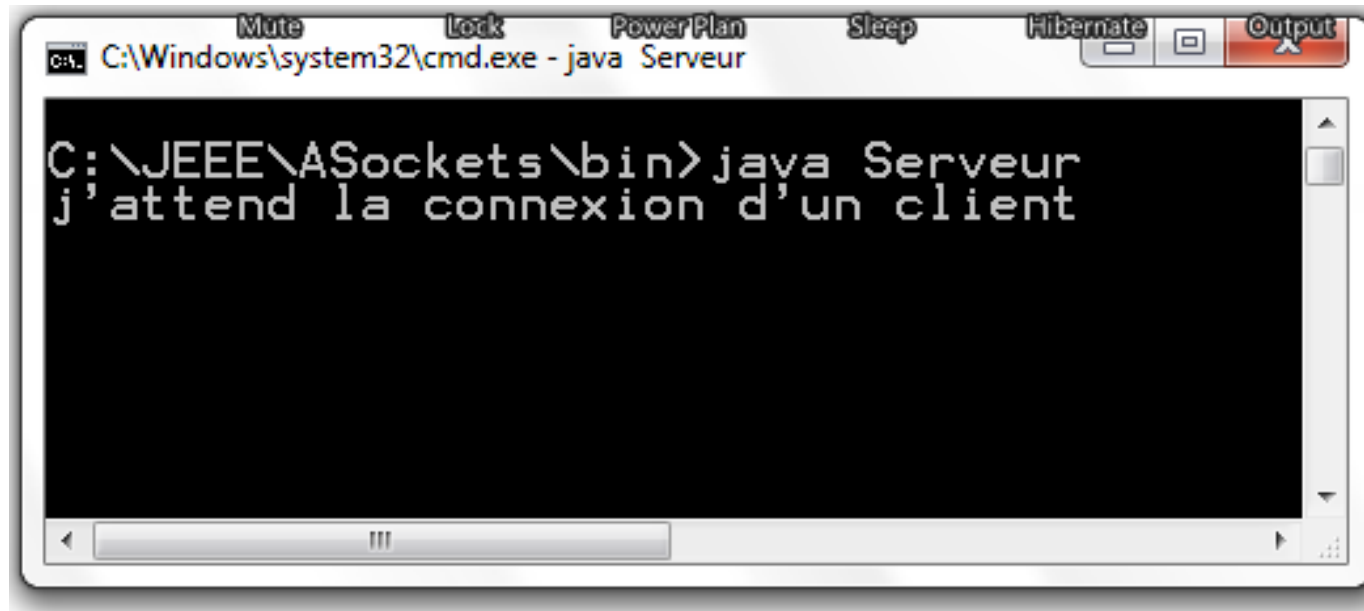
# Exemple d'un serveur simple

```
import java.io.*;import java.net.*;
public class Serveur {
public static void main(String[] args) {
    try {
ServerSocket ss=new ServerSocket(1234);
System.out.println("j'attends la connexion d'un client");
Socket clientSocket=ss.accept();
System.out.println("Nouveau client connecté");
System.out.println("Génération de objet InptStream et
    OutputStream de la socket");
InputStream is=clientSocket.getInputStream();
OutputStream os=clientSocket.getOutputStream();
System.out.println("J'attends un nombre (1 octet)!");
int nb=is.read();
System.out.println("J'envoie la réponse");
os.write(nb*5);
System.out.println("Déconnexion du client");
clientSocket.close();
    } catch (IOException e) {
e.printStackTrace();
    }
}
}
```

# Exemple d'un client simple

```
import java.io.*;import java.net.*;import java.util.Scanner;
public class Client {
    public static void main(String[] args) {
    try {
        System.out.println("Créer une connexion au serveur:");
        Socket clientSocket=new Socket("localhost", 123);
        System.out.println("Génération de objet InptStream et OutputStream
de la socket");
        InputStream is=clientSocket.getInputStream();
        OutputStream os=clientSocket.getOutputStream();
        System.out.print("Lire un nombre au clavier NB=");
        Scanner clavier=new Scanner(System.in);
        int nb=clavier.nextInt();
        System.out.println("Envoyer le nombre "+nb+" au serveur");
        os.write(nb);
        System.out.println("Attendre la réponse du serveur:");
        int rep=is.read();
        System.out.println("La réponse est :"+rep);
    } catch (Exception e) {
        e.printStackTrace();
    }
    }
}
```

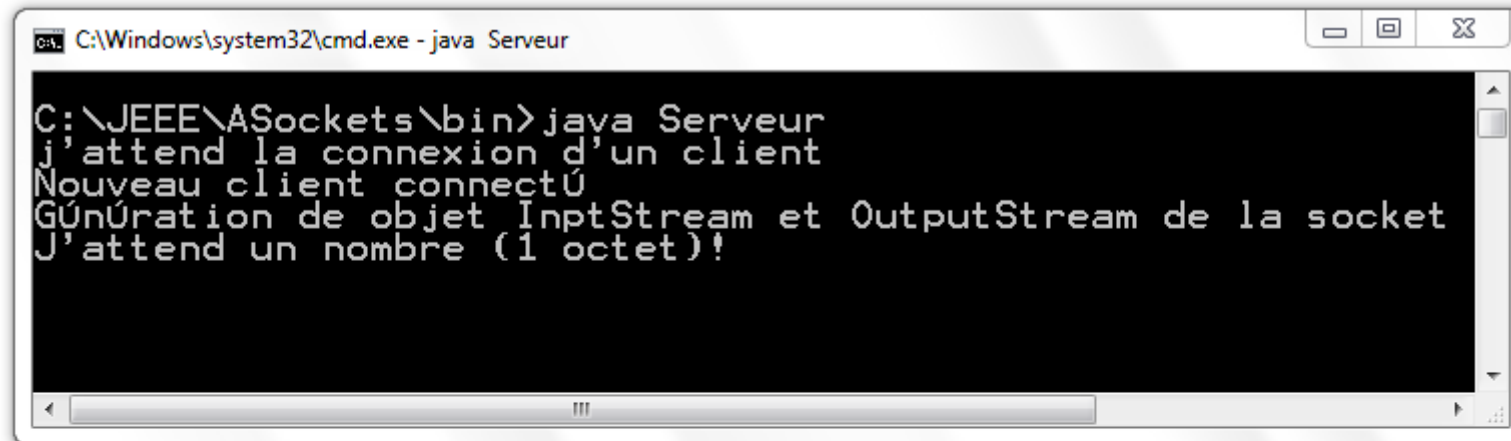
# Lancement du serveur sur ligne de commandes



The image shows a Windows command prompt window. The title bar includes standard Windows window controls (minimize, maximize, close) and a status bar with 'Mute', 'Lock', 'Power Plan', 'Sleep', 'Hibernate', and 'Output' buttons. The address bar shows 'C:\Windows\system32\cmd.exe - java Serveur'. The command prompt itself has a black background with white text. The prompt is 'C:\JEEE\ASockets\bin>' and the command entered is 'java Serveur'. The output of the command is 'j'attend la connexion d'un client'.

```
C:\Windows\system32\cmd.exe - java Serveur  
C:\JEEE\ASockets\bin>java Serveur  
j'attend la connexion d'un client
```

# Lancement du client sur ligne de commandes



```
C:\Windows\system32\cmd.exe - java Serveur

C:\JEEE\ASockets\bin>java Serveur
j'attend la connexion d'un client
Nouveau client connecté
Génération de objet InptStream et OutputStream de la socket
J'attend un nombre (1 octet)!
```



```
C:\Windows\system32\cmd.exe - java Client

C:\JEEE\ASockets\bin>java Client
Créer une connexion au serveur:
Génération de objet InptStream et OutputStream de la socket
Lire un nombre au clavier NB=_
```

# Saisir un nombre par le client et l'envoyer au serveur



```
C:\Windows\system32\cmd.exe

C:\JEEE\ASockets\bin>java Serveur
j'attend la connexion d'un client
Nouveau client connecté
Génération de objet InptStream et OutputStream de la socket
J'attends un nombre (1 octet)!
J'envoie la réponse
Déconnexion du client

C:\JEEE\ASockets\bin>
```



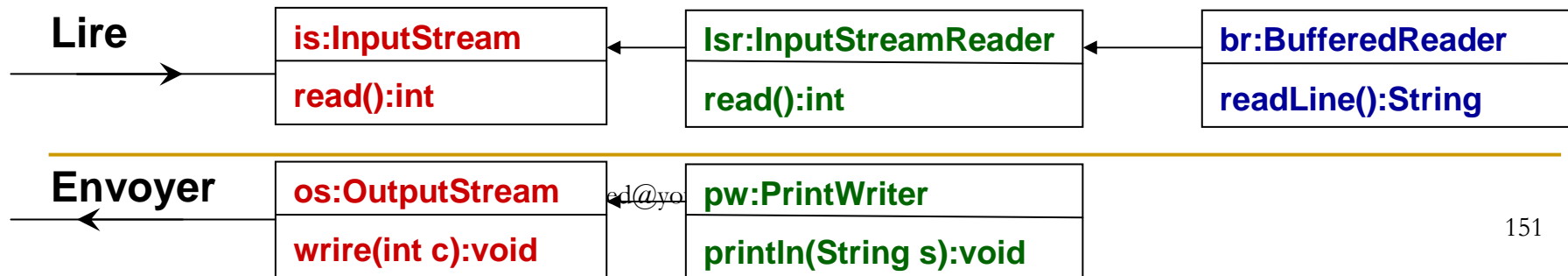
```
C:\Windows\system32\cmd.exe

C:\JEEE\ASockets\bin>java Client
Créer une connexion au serveur:
Génération de objet InptStream et OutputStream de la socket
Lire un nombre au clavier NB=12
Envoyer le nombre 12 au serveur
Attendre la réponse du serveur:
La réponse est :60

C:\JEEE\ASockets\bin>
```

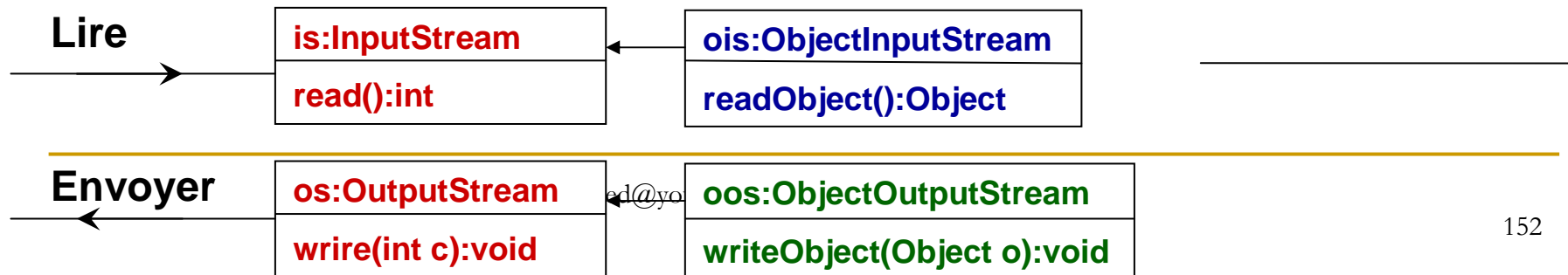
# Recevoir et envoyer des chaînes de caractères

- Création de l'objet ServerSocket
  - `ServerSocket ss=new ServerSocket(2345);`
- Attendre une connexion d'un client
  - `Socket sock=ss.accept();`
- Pour lire une chaîne de caractère envoyée par le client :
  - `InputStream is=sock.getInputStream();`
  - `InputStreamReader isr=new InputStreamReader(is);`
  - `BufferedReader br=new BufferedReader(isr);`
  - `String s=br.readLine();`
- Pour envoyer une chaîne de caractères au client
  - `OutputStream os=sock.getOutputStream();`
  - `PrintWriter pw=new PrintWriter(os,true);`
  - `pw.println("Chaîne de caractères");`



# R/E des objets (Sérialisation et désérialisation)

- Une classe Sérializable:
  - ❑ public class Voiture implements Serializable{
  - ❑ String mat;int carburant;
  - ❑ public Voiture(String m, int c) { mat=m; carburant=c;}
  - ❑ }
- Pour sérialiser un objet (envoyer un objet vers le client)
  - ❑ `OutputStream os=sock.getOutputStream();`
  - ❑ `ObjectOutputStream oos=new ObjectOutputStream(os);`
  - ❑ `Voiture v1=new Voiture("v212",50);`
  - ❑ `oos.writeObject(v1);`
- Pour lire un objet envoyé par le client ( désérialisation)
  - ❑ `InputStream is=sock.getInputStream();`
  - ❑ `ObjectInputStream ois=new ObjectInputStream(is);`
  - ❑ `Voiture v=(Voiture) ois.readObject();`



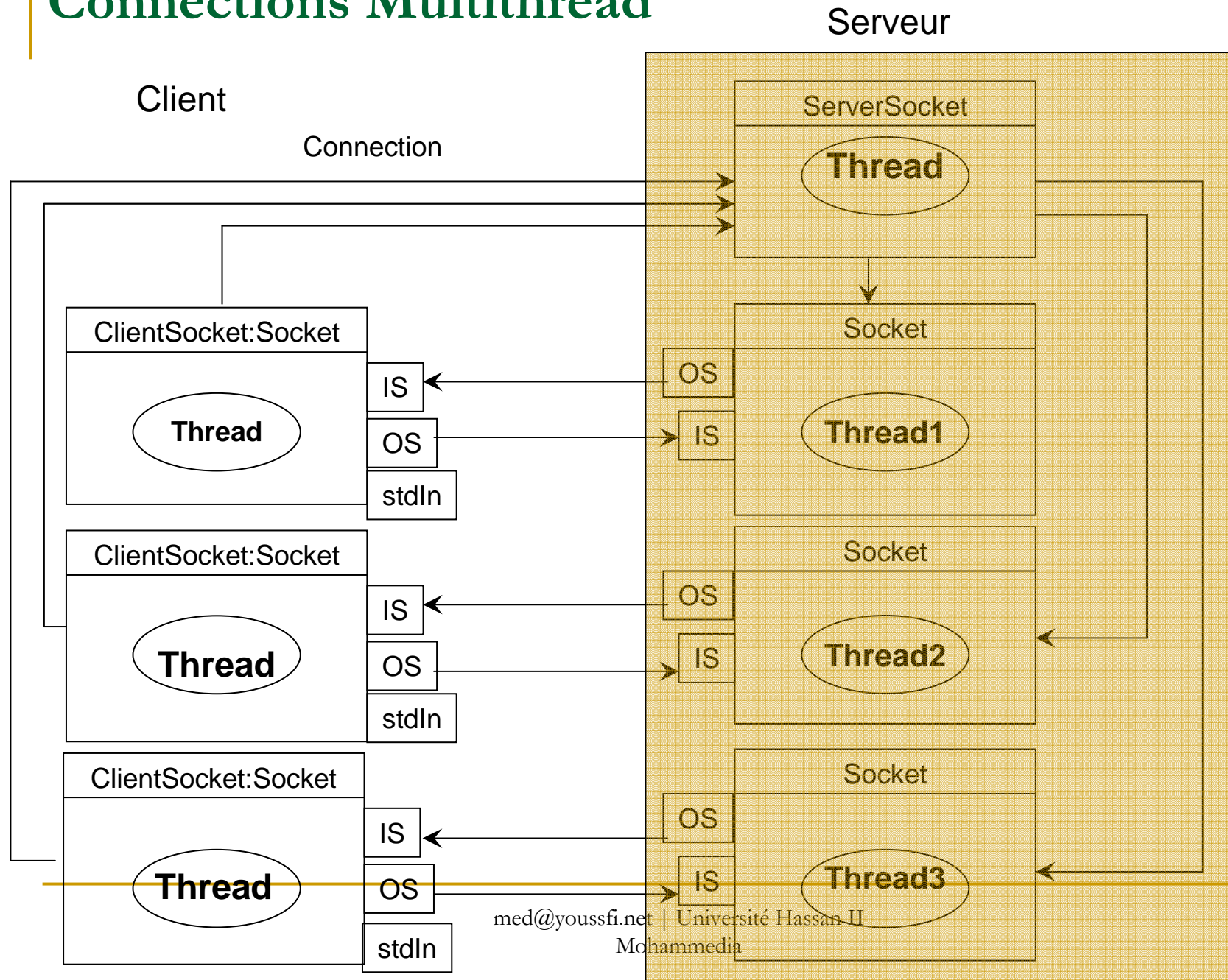


---

# Serveur Multi-Threads

- Pour qu'un serveur puisse communiquer avec plusieurs client en même temps, il faut que:
  - ❑ Le serveur puisse attendre une connexion à tout moment.
  - ❑ Pour chaque connexion, il faut créer un nouveau thread associé à la socket du client connecté, puis attendre à nouveau une nouvelle connexion
  - ❑ Le thread créé doit s'occuper des opérations d'entrées-sorties (read/write) pour communiquer avec le client indépendamment des autres activités du serveur.

# Connections Multithread



# Implémentation d'un serveur multi-thread

- Le serveur se compose au minimum par deux classes:
  - `ServeurMultiThread.java` : serveur principal

```
import java.io.*; import java.net.*;
public class ServeurMultiThread extends Thread {
    int nbClients=0; // pour compter le nombre de clients connectés
    public void run(){
        try {
            System.out.println("J'attend une connexion");
            ServerSocket ss=new ServerSocket(4321);
            while(true){
                Socket s=ss.accept();
                ++nbClients;
                new ServiceClient(s,nbClients).start();
            }
        } catch (IOException e) {e.printStackTrace();}
    } // Fin de la méthode run
    public static void main(String[] args) {
        new ServeurMultiThread().start();
    }
}
```

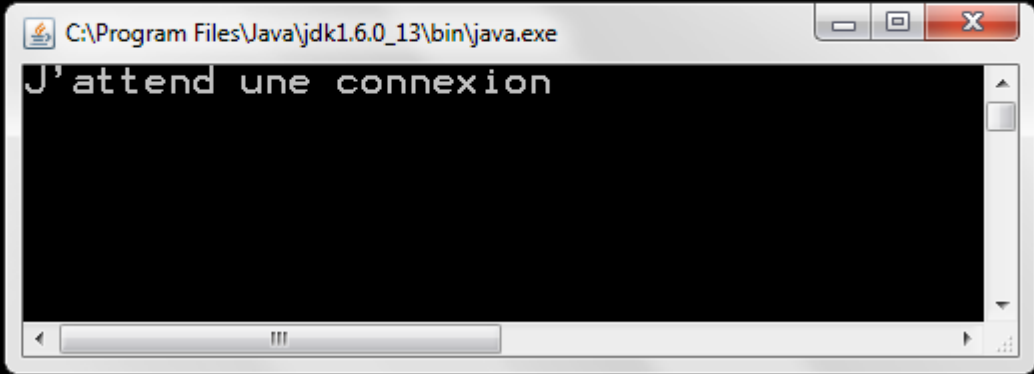
# Implémentation d'un serveur multi-thread

- La classe qui gère les entrées-sorties : ServiceClient.java

```
import java.io.*; import java.net.*;
public class ServiceClient extends Thread {
    Socket client; int numero;
    public ServiceClient(Socket s,int num){client=s;numero=num; }
    public void run(){
        try {
            BufferedReader in=new BufferedReader(
                new InputStreamReader(client.getInputStream()));
            PrintWriter out=new PrintWriter(client.getOutputStream(),true);
            System.out.println("Client Num "+numero);
            out.println("Vous etes le client num "+numero);
            while(true){
                String s=in.readLine();
                out.println("Votre commande contient "+ s.length()+" caractères");
            } } catch (IOException e) { e.printStackTrace();}
    }
}
```

# Lancement du serveur multithreads

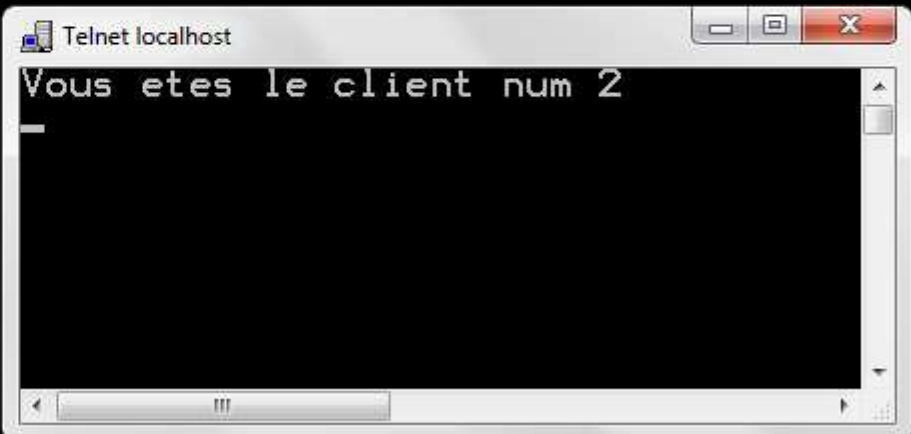

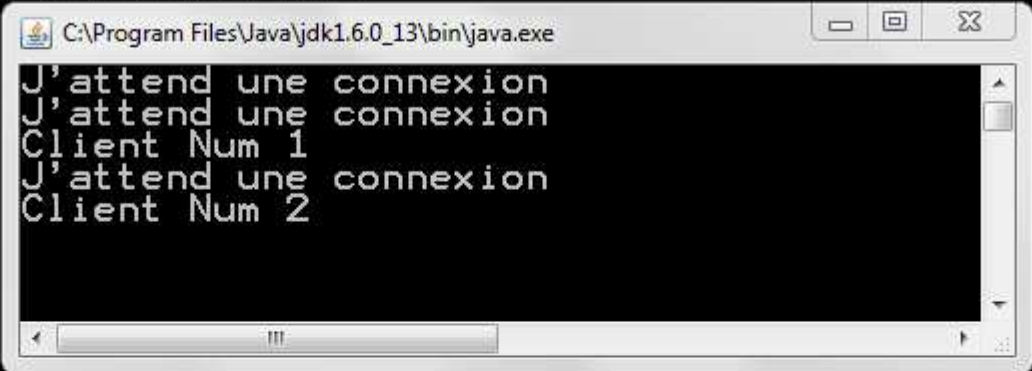
```
C:\JEEE\ASockets\bin>start java ServeurMultiThread  
C:\JEEE\ASockets\bin>
```



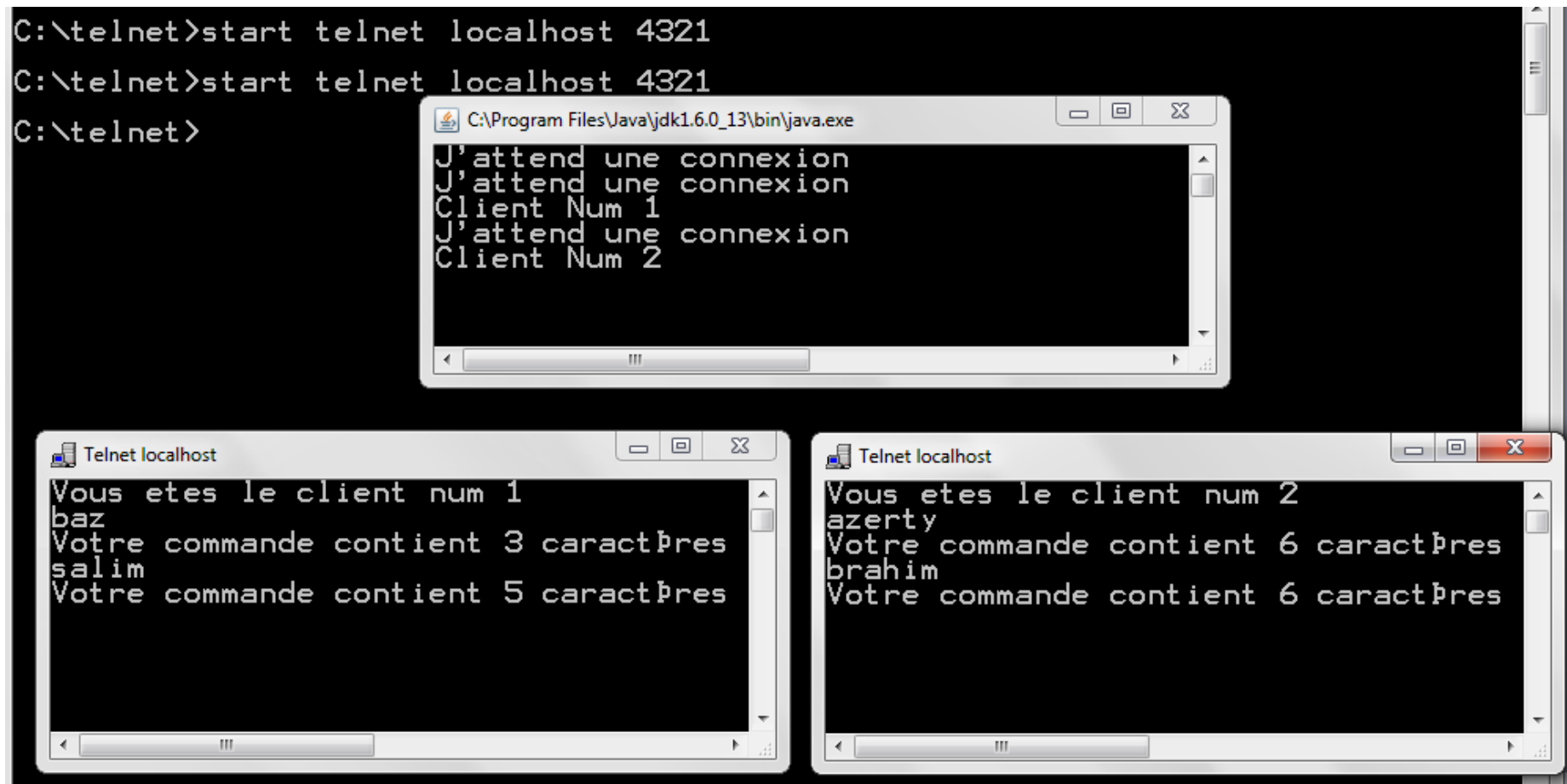
The screenshot shows a Windows command prompt window with a black background. The text 'C:\JEEE\ASockets\bin>start java ServeurMultiThread' is entered on the first line, and 'C:\JEEE\ASockets\bin>' is on the second line. Overlaid on this is a standard Windows application window titled 'C:\Program Files\Java\jdk1.6.0\_13\bin\java.exe'. This window has a white title bar with minimize, maximize, and close buttons. The main content area of the window is black and displays the text 'J'attends une connexion' in white. A vertical scrollbar is visible on the right side of the window.

# Lancement des clients avec telnet

```
C:\telnet>start telnet localhost 4321
C:\telnet>start telnet localhost 4321
C:\telnet>
```



# Communication entre les clients et le serveur



The screenshot displays a Telnet server interface and two client windows. The server window, titled 'C:\telnet>', shows the command 'start telnet localhost 4321' being executed twice. A Java window, titled 'C:\Program Files\Java\jdk1.6.0\_13\bin\java.exe', shows the server's output: 'J'attend une connexion', 'Client Num 1', 'J'attend une connexion', and 'Client Num 2'. The two client windows, both titled 'Telnet localhost', show the server's response to client input. The left client window shows 'Vous etes le client num 1', 'baz', 'Votre commande contient 3 caractères', and 'salim'. The right client window shows 'Vous etes le client num 2', 'azerty', 'Votre commande contient 6 caractères', and 'brahim'.

```
C:\telnet>start telnet localhost 4321
C:\telnet>start telnet localhost 4321
C:\telnet>
```

```
C:\Program Files\Java\jdk1.6.0_13\bin\java.exe
J'attend une connexion
J'attend une connexion
Client Num 1
J'attend une connexion
Client Num 2
```

```
Telnet localhost
Vous etes le client num 1
baz
Votre commande contient 3 caractères
salim
Votre commande contient 5 caractères
```

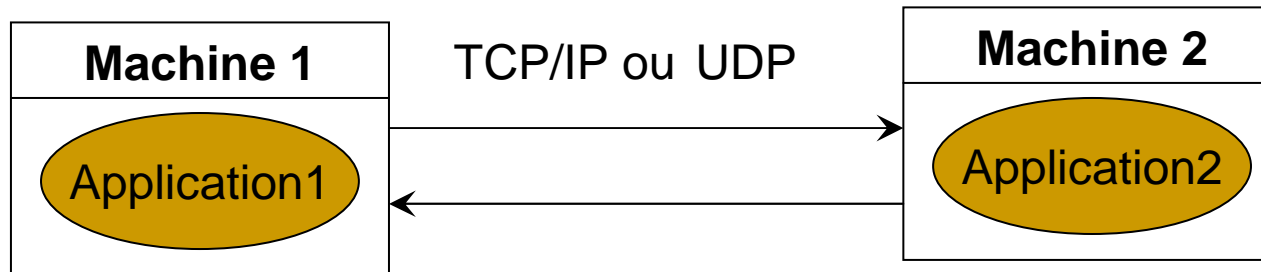
```
Telnet localhost
Vous etes le client num 2
azerty
Votre commande contient 6 caractères
brahim
Votre commande contient 6 caractères
```

# Architectures distribuées

Par M.Youssfi



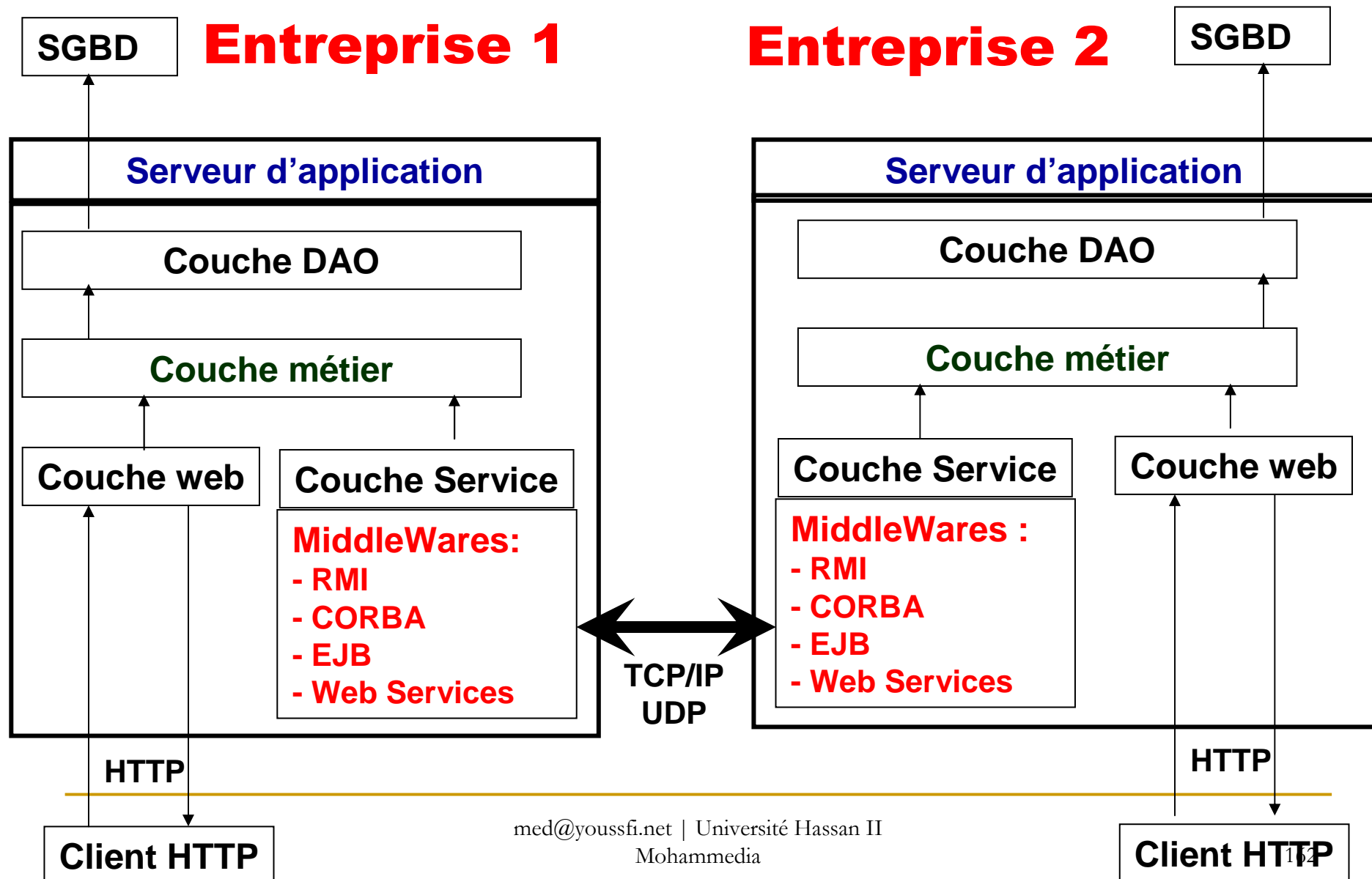
# Applications distribuées



## Technologies d'accès :

- Sockets
- RMI (JAVA)
- CORBA (Multi Langages)
- EJB (J2EE)
- Web Services (HTTP+XML)

# Architectures Distribuées



---

# Applications distribuées

- Une application distribuée est une application dont les classes sont réparties sur plusieurs machines différentes.
- Dans de telles applications, on peut invoquer des méthodes à distance.
- Il est alors possible d'appeler les méthode d'un objet qui n'est pas situé sur la machine locale.

---

# Applications distribuées

## ■ RPC : Remote Procedure Calls

- ❑ Déjà dans le langage C, il était possible de faire de l'invocation à distance en utilisant RPC.
- ❑ RPC étant orienté "structure de données", il ne suit pas le modèle "orienté objet".

## ■ RMI : Remote Method Invocation

- ❑ RMI est un middleware qui permet l'utilisation d'objets sur des JVM distantes.
- ❑ RMI va plus loin que RPC puisqu'il permet non seulement l'envoi des données d'un objet, mais aussi de ses méthodes. Cela se fait en partie grâce à la sérialisation des objets
- ❑ RMI est utilisé pour créer des applications distribuées développées avec Java.

---

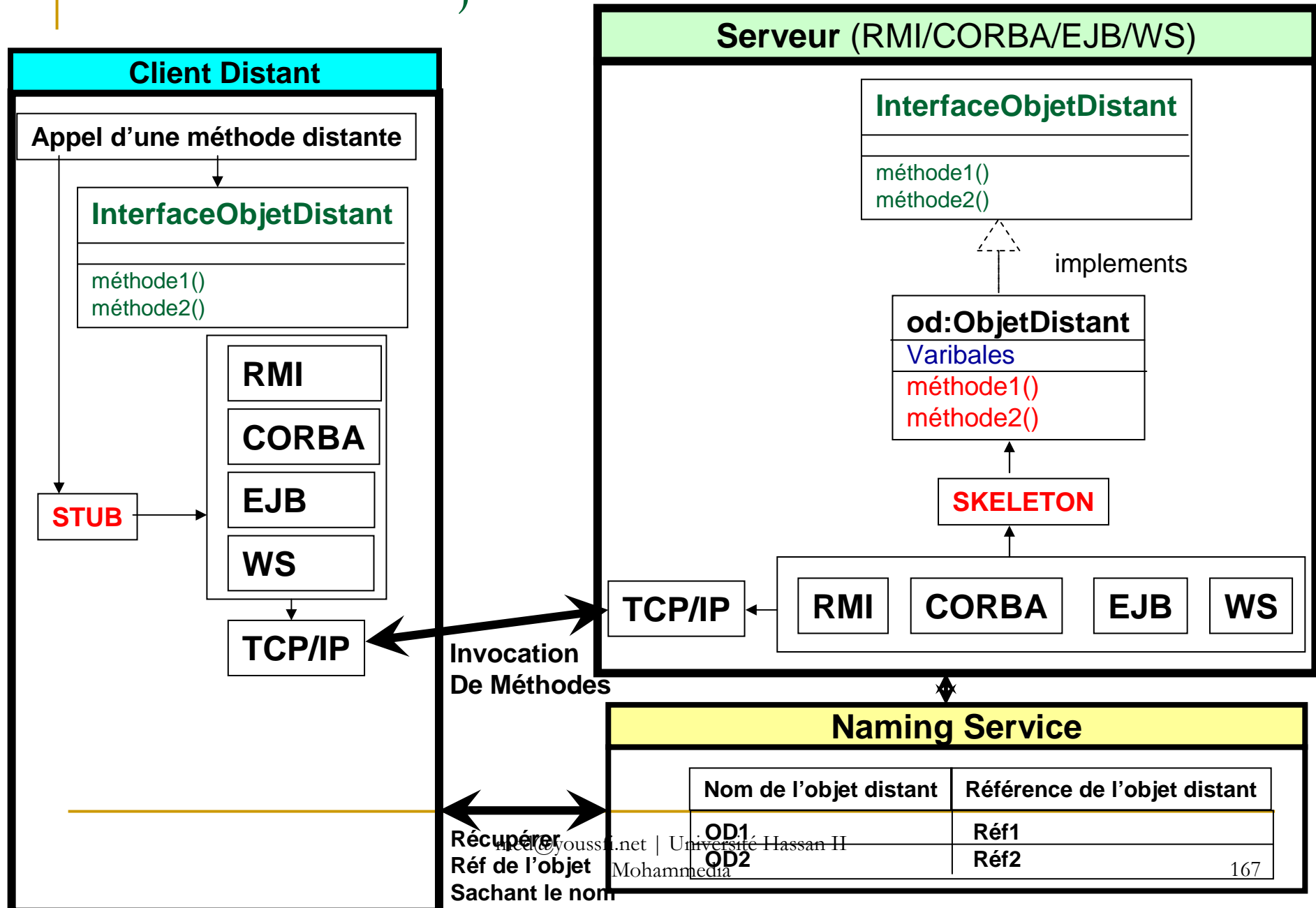
# Application distribuée

- CORBA : Common Object Request Broker Architecture
  - Il existe une technologie, non liée à Java, appelée CORBA qui est un standard d'objet réparti.
  - CORBA a été conçue par l'OMG (Object Management Group), un consortium regroupant 700 entreprises dont Sun.
  - Le grand intérêt de CORBA est qu'il fonctionne sous plusieurs langages, mais il est plus lourd à mettre en place.

# Application distribuée

- **EJB : Entreprise Java Beans**
  - ❑ Avec Le développement des serveur d'applications J2EE Sun, a créé la technologie EJB qui permet également de créer des composants métier distribués.
  - ❑ Cette technologie est dédiée pour les applications J2EE
  - ❑ Les EJB utilise RMI et CORBA
- **Web Services : ( Protocole SOAP = HTTP + XML )**
  - ❑ Avec le développement du web et de la technologie XML, La technologie de distribution « Web Services » a été développée.
  - ❑ Avec les web services, on peut développer des applications distribuées multi langages et multi plateformes.
  - ❑ Les web services sont plus facile à mettre en œuvre.
  - ❑ Les web services exploitent deux concepts XML et HTTP
- **Toutes ces technologies de création des applications collaboratives s'appuient sur les SOCKETS**

# Accès à un objet distant via un middleware

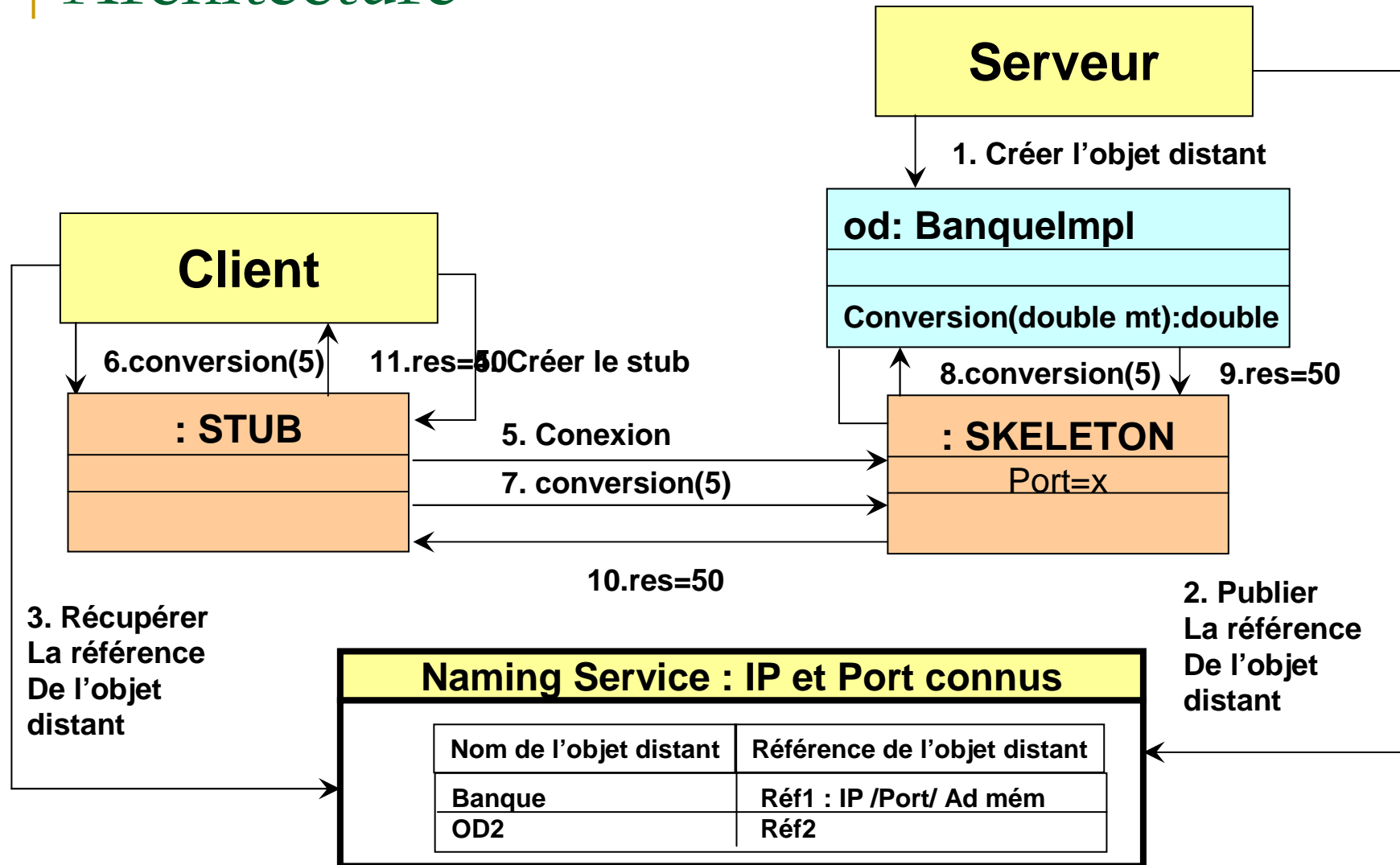


---

# RMI



# Architecture



---

# RMI

- Le but de RMI est de créer un modèle objet distribué Java
- RMI est apparu dès la version 1.1 du JDK
- Avec RMI, les méthodes de certains objets (appelés objets distants) peuvent être invoquées depuis des JVM différentes (espaces d'adressages distincts)
- En effet, RMI assure la communication entre le serveur et le client via TCP/IP et ce de manière transparente pour le développeur.
- Il utilise des mécanismes et des protocoles définis et standardisés tels que les sockets et RMP (Remote Method Protocol).
- Le développeur n'a donc pas à se soucier des problèmes de niveaux inférieurs spécifiques aux technologies réseaux.

---

# Rappel sur les interfaces

- Dans java, une interface est une classe abstraite qui ne contient que des méthodes abstraites.
- Dans java une classe peut hériter d'une seule classe et peut implémenter plusieurs interfaces.
- Implémenter une interface signifie qu'il faut redéfinir toutes les méthodes déclarées dans l'interface.

# Exemple

- Un exemple d'interface

```
public interface IBanque{  
    public void verser(float mt);  
}
```

- Un exemple d'implémentation

```
public class BanqueImpl implements IBanque{  
    private float solde ;  
  
    public void verser(float mt){  
        solde=solde+mt;  
    }  
}
```

---

# Architecture de RMI

## ■ Interfaces:

- ❑ Les interfaces est le cœur de RMI.
- ❑ L'architecture RMI est basé sur un principe important :
  - La définition du comportement et l'exécution de ce comportement sont des concepts séparés.
- ❑ La définition d'un service distant est codé en utilisant une interface Java.
- ❑ L'implémentation de ce service distant est codée dans une classe.

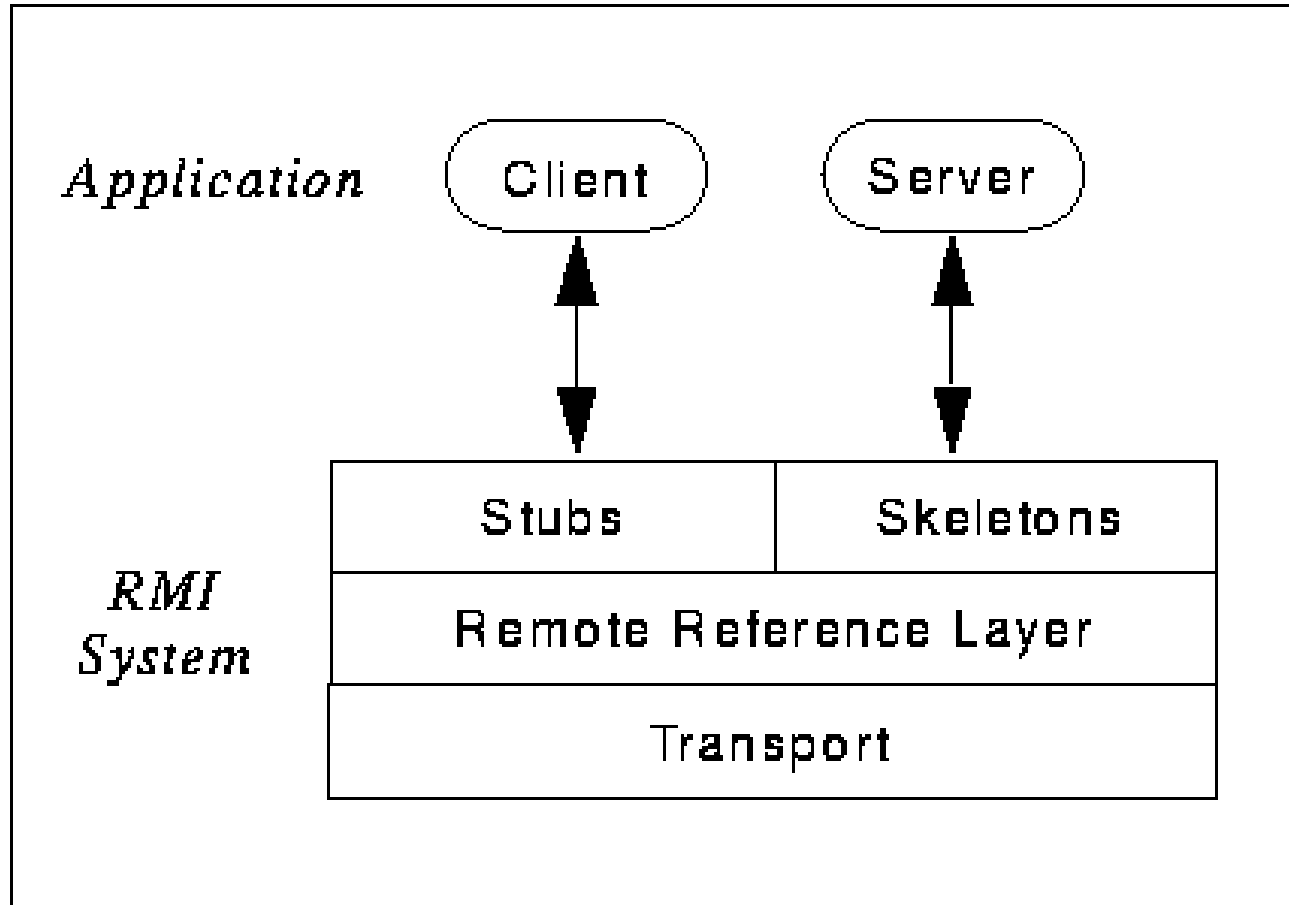
---

# Coches de RMI

- RMI est essentiellement construit sur une abstraction en trois couches.
  - Stubs et Skeletons (Souche et Squelette)
  - Remote Reference Layer (Couche de référencement distante)
  - Couche Transport

# Architecture de RMI

## ■ Architecture 1



---

# Stubs et Skeletons

- Les stubs sont des classes placées dans la machine du client.
- Lorsque notre client fera appel à une méthode distante, cet appel sera transféré au stub.
- Le stub est donc un relais du côté du client. Il devra donc être placé sur la machine cliente.
- Le stub est le représentant local de l'objet distant.
- Il emballe les arguments de la méthode distante et les envoie dans un flux de données vers le service RMI distant.
- D'autre part, il déballe la valeur ou l'objet de retour de la méthode distante.
- Il communique avec l'objet distant par l'intermédiaire d'un skeleton.



---

# Stubs et Skeletons

- Le squelette est lui aussi un relais mais du côté serveur. Il devra être placé sur la machine du serveur.
- Il débale les paramètres de la méthode distante, les transmet à l'objet local et embale les valeurs de retours à renvoyer au client.
- Les stubs et les skeletons sont donc des intermédiaires entre le client et le serveur qui gèrent le transfert distant des données.
- On utilise le compilateur `rmic` pour la génération des stubs et des skeletons avec le JDK
- Depuis la version 2 de Java, le skeleton n'existe plus. Seul le stub est nécessaire du côté du client mais aussi du côté serveur.

---

# Remote Reference Layer

- La deuxième couche est la couche de référence distante.
- Ce service est assuré par le lancement du programme **rmiregistry** du côté du serveur
- Le serveur doit enregistrer la référence de chaque objet distant dans le service rmiregistry en attribuant un nom à cet objet distant.
- Du côté du client, cette couche permet l'obtention d'une référence de l'objet distant à partir de la référence locale (le stub) en utilisant son nom rmiregsitry

---

# Couche transport

- La couche transport est basée sur les connexions TCP/IP entre les machines.
- Elle fournit la connectivité de base entre les 2 JVM.
- De plus, cette couche fournit des stratégies pour passer les firewalls.
- Elle suit les connexions en cours.
- Elle construit une table des objets distants disponibles.
- Elle écoute et répond aux invocations.
- Cette couche utilise les classes Socket et ServerSocket.
- Elle utilise aussi un protocole propriétaire R.M.P. (Remote Method Protocol).

---

# Démarche RMI

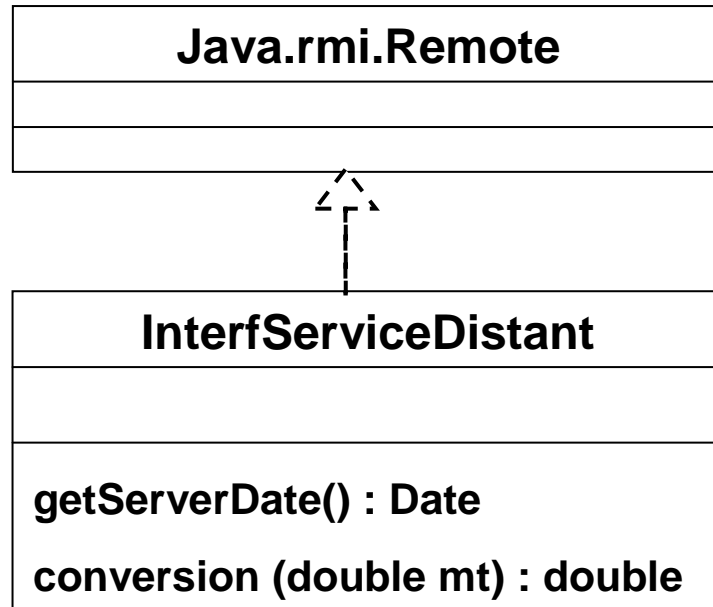
1. Créer les interfaces des objets distants
2. Créer les implémentation des objets distants
3. Générer les stubs et skeletons
4. Créer le serveur RMI
5. Créer le client RMI
6. Déploiement Lancement
  - Lancer l'annuaire RMIREGISTRY
  - Lancer le serveur
  - Lancer le client

---

# Exemple

- Supposant qu'on veut créer un serveur RMI qui crée un objet qui offre les services distants suivant à un client RMI:
  - ❑ Convertir un montant de l'euro en DH
  - ❑ Envoyer au client la date du serveur.

# 1- Interface de l'objet distant



```
import java.rmi.Remote;import java.rmi.RemoteException;
import java.util.Date;
public interface InterfServiceDistant extends Remote {
    public Date getServerDate() throws RemoteException;
    public double convertEuroToDH(double montant) throws
RemoteException;
}
```

---

# Interfaces

- La première étape consiste à créer une interface distante qui décrit les méthodes que le client pourra invoquer à distance.
- Pour que ses méthodes soient accessibles par le client, cette interface doit hériter de l'interface **Remote**.
- Toutes les méthodes utilisables à distance doivent pouvoir lever les exceptions de type **RemoteException** qui sont spécifiques à l'appel distant.
- Cette interface devra être placée sur les deux machines (serveur et client).

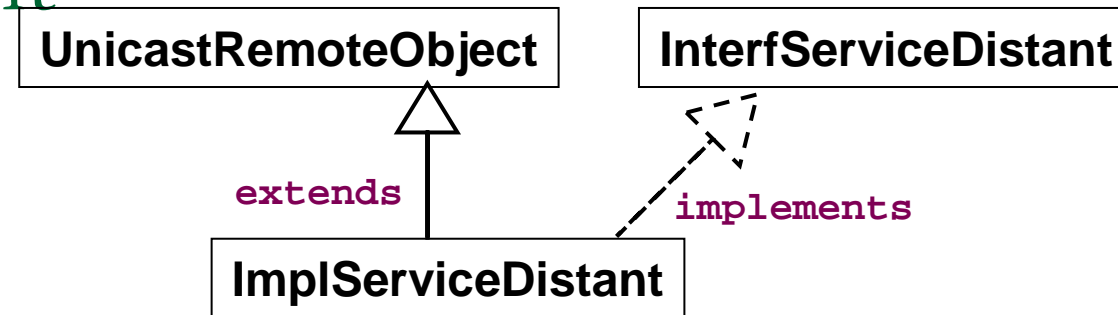
---

# Implémentation

- Il faut maintenant implémenter cette interface distante dans une classe. Par convention, le nom de cette classe aura pour suffixe Impl.
- Notre classe doit hériter de la classe `java.rmi.server.RemoteObject` ou de l'une de ses sous-classes.
- La plus facile à utiliser étant `java.rmi.server.UnicastRemoteObject`.
- C'est dans cette classe que nous allons définir le corps des méthodes distantes que pourront utiliser nos clients



## Deuxième étape : Implémentation de l'objet distant



```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class ImplServiceDistant extends UnicastRemoteObject implements
    InterfServiceDistant {
    public ImplServiceDistant() throws RemoteException{

    }

    public Date getServerDate() throws RemoteException {
    return new Date();
    }

    public double convertEuroToDH(double montant) throws RemoteException {
    return montant*11.3;
    }
}
```

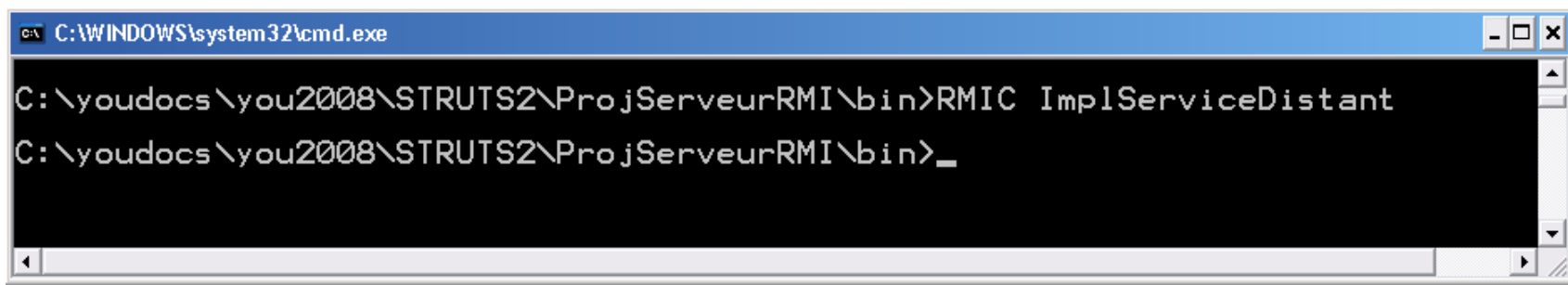
---

# Implémentation

- Il faut maintenant implémenter cette interface distante dans une classe. Par convention, le nom de cette classe aura pour suffixe Impl.
- Notre classe doit hériter de la classe `java.rmi.server.RemoteObject` ou de l'une de ses sous-classes.
- La plus facile à utiliser étant `java.rmi.server.UnicastRemoteObject`.
- C'est dans cette classe que nous allons définir le corps des méthodes distantes que pourront utiliser nos clients

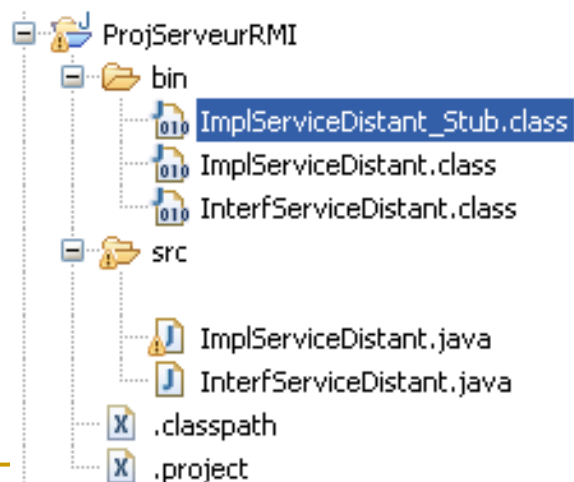
# STUBS et SKELETONS

- Si l'implémentation est créée, les stubs et skeletons peuvent être générés par l'utilitaire rmic en écrivant la commande:
- Rmic NOM\_IMPLEMENTATION



```
C:\WINDOWS\system32\cmd.exe

C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>RMIC ImplServiceDistant
C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>_
```



---

# Naming Service

- Les clients trouvent les services distants en utilisant un service d'annuaire activé sur un hôte connu avec un numéro de port connu.
- RMI peut utiliser plusieurs services d'annuaire, y compris Java Naming and Directory Interface (JNDI).
- Il inclut lui-même un service simple appelé (rmiregistry).
- Le registre est exécuté sur chaque machine qui héberge des objets distants (les serveurs) et accepte les requêtes pour ces services, par défaut sur le port 1099.

---

# Utilisation de RMI

- Un serveur crée un service distant en créant d'abord un objet local qui implémente ce service.
- Ensuite, il exporte cet objet vers RMI.
- Quand l'objet est exporté, RMI crée un service d'écoute qui attend qu'un client se connecte et envoie des requêtes au service.
- Après exportation, le serveur enregistre cet objet dans le registre de RMI sous un nom public qui devient accessible de l'extérieur.
- Le client peut alors consulter le registre distant pour obtenir des références à des objets distants.

---

# Le serveur

- Notre serveur doit enregistrer auprès du registre RMI l'objet local dont les méthodes seront disponibles à distance.
- Cela se fait grâce à la méthode statique **bind()** ou **rebind()** de la classe Naming.
- Cette méthode permet d'associer (enregistrer) l'objet local avec un synonyme dans le registre RMI.
- L'objet devient alors disponible par le client.
  - ❑ `ObjetDistantImpl od = new ObjetDistantImpl();`
  - ❑ `Naming.rebind("rmi://localhost:1099/NOM_Service",od);`

# Code du serveur RMI

```
import java.rmi.Naming;
public class ServeurRMI {
    public static void main(String[] args) {
        try {
            // Créer l'objet distant
            ImplServiceDistant od=new ImplServiceDistant();
            // Publier sa référence dans l'annuaire
            Naming.rebind("rmi://localhost:1099/SD",od);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

# Client

- Le client peut obtenir une référence à un objet distant par l'utilisation de la méthode statique **lookup()** de la classe Naming.
- La méthode lookup() sert au client pour interroger un registre et récupérer un objet distant.
- Elle retourne une référence à l'objet distant.
- La valeur retournée est du type Remote. Il est donc nécessaire de caster cet objet en l'interface distante implémentée par l'objet distant.

```
import java.rmi.Naming;
public class ClientRMI {
    public static void main(String[] args) {
        try {
            InterfServiceDistant stub=
                (InterfServiceDistant)Naming.lookup("rmi://localhost:1099/SD");
            System.out.println("Date du serveur:"+stub.getServerDate());
            System.out.println("35 euro vaut "+stub.convertEuroToDH(35));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



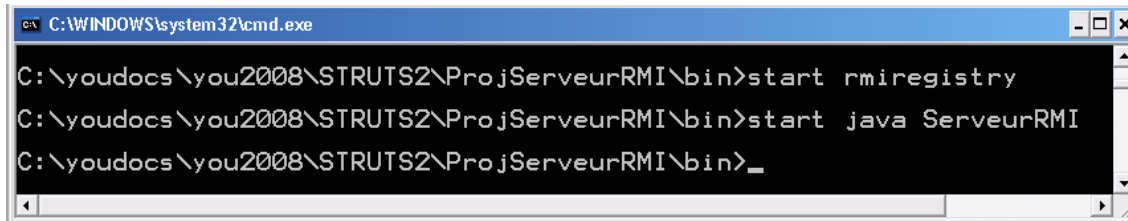
---

# Lancement

- Il est maintenant possible de lancer l'application. Cela va nécessiter l'utilisation de trois consoles.
  - ❑ La première sera utilisée pour activer le registre. Pour cela, vous devez exécuter l'utilitaire **rmiregistry**
  - ❑ Dans une deuxième console, exécutez le serveur. Celui-ci va charger l'implémentation en mémoire, enregistrer cette référence dans le registre et attendre une connexion cliente.
  - ❑ Vous pouvez enfin exécuter le client dans une troisième console.
- Même si vous exécutez le client et le serveur sur la même machine, RMI utilisera la pile réseau et TCP/IP pour communiquer entre les JVM.

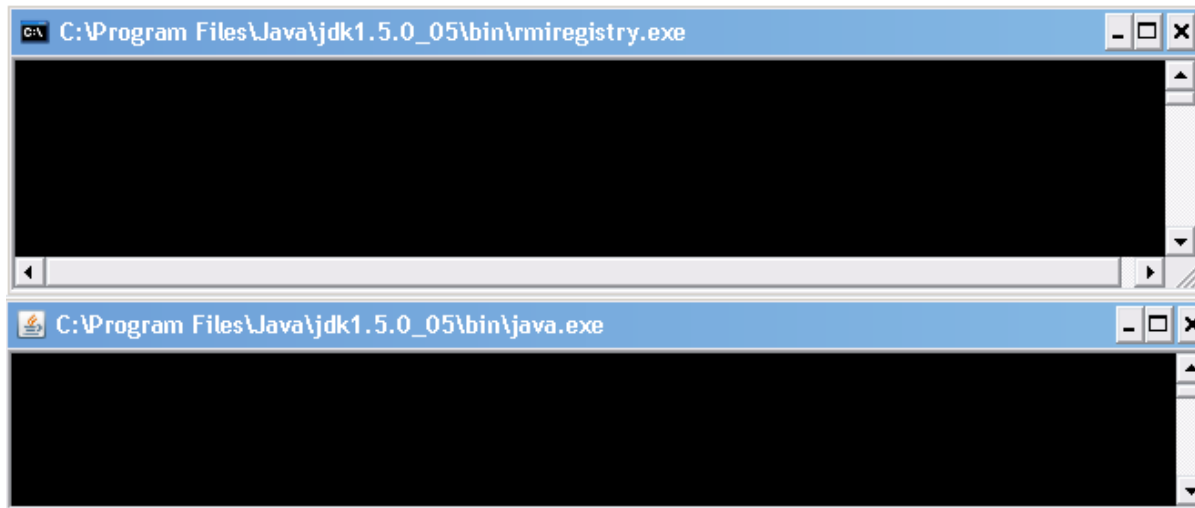
# Lancement

Lancement de l'annuaire puis du serveur rmi :

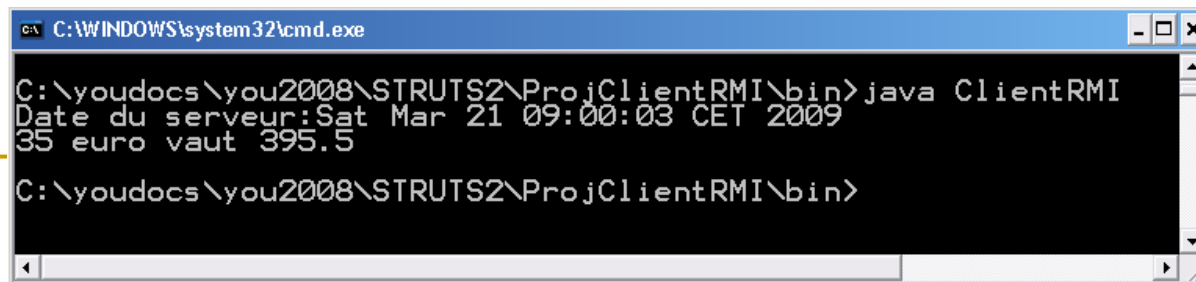


```
C:\WINDOWS\system32\cmd.exe

C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>start rmiregistry
C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>start java ServeurRMI
C:\youdocs\you2008\STRUTS2\ProjServeurRMI\bin>_
```



Lancement du client RMI :



```
C:\WINDOWS\system32\cmd.exe

C:\youdocs\you2008\STRUTS2\ProjClientRMI\bin>java ClientRMI
Date du serveur:Sat Mar 21 09:00:03 CET 2009
35 euro vaut 395.5
C:\youdocs\you2008\STRUTS2\ProjClientRMI\bin>
```

---

# JNDI (Java Naming and Directory Interface)

- JNDI est l'acronyme de Java Naming and Directory Interface.
- Cette API fournit une interface unique pour utiliser différents services de nommages ou d'annuaires et définit une API standard pour permettre l'accès à ces services.
- Il existe plusieurs types de service de nommage parmi lesquels :
  - DNS (Domain Name System) : service de nommage utilisé sur internet pour permettre la correspondance entre un nom de domaine et une adresse IP
  - LDAP(Lightweight Directory Access Protocol) : annuaire
  - NIS (Network Information System) : service de nommage réseau développé par Sun Microsystems
  - COS Naming (Common Object Services) : service de nommage utilisé par Corba pour stocker et obtenir des références sur des objets Corba
  - RMI Registry : service de nommage utilisé par RMI
  - etc, ...

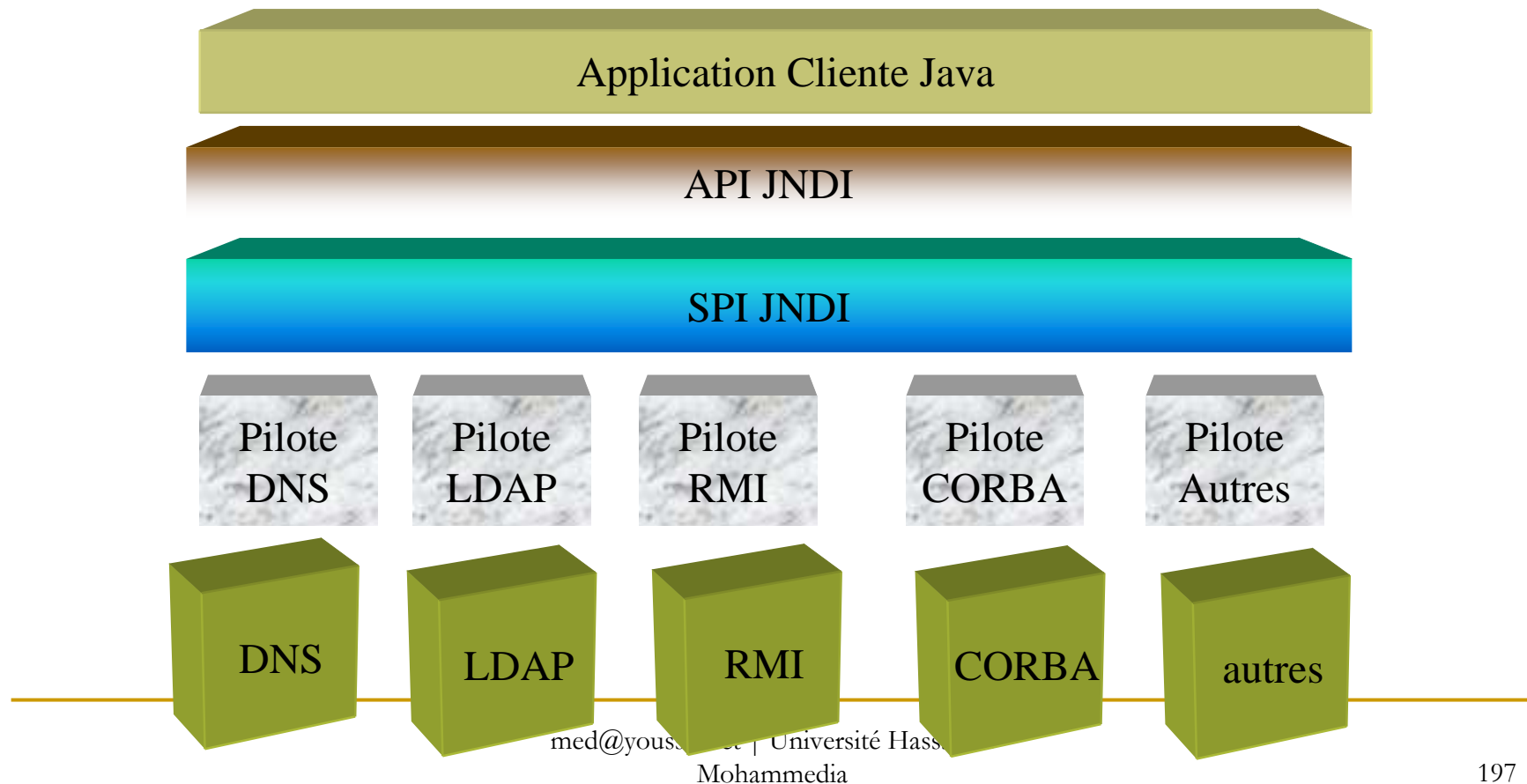
---

# JNDI (Java Naming and Directory Interface)

- Un service de nommage permet d'associer un nom unique à un objet et de faciliter ainsi l'obtention de cet objet.
- Un annuaire est un service de nommage qui possède en plus une représentation hiérarchique des objets qu'il contient et un mécanisme de recherche.
- JNDI propose donc une abstraction pour permettre l'accès à ces différents services de manière standard. Ceci est possible grâce à l'implémentation de pilotes qui mettent en œuvre la partie SPI de l'API JNDI. Cette implémentation se charge d'assurer le dialogue entre l'API et le service utilisé.

# Architecture JNDI

- L'architecture de JNDI se compose d'une API et d'un Service Provider Interface (SPI). Les applications Java emploient l'API JNDI pour accéder à une variété de services de nommage et d'annuaire. Le SPI permet de relier, de manière transparente, une variété de services de nommage et d'annuaire ; permettant ainsi à l'application Java d'accéder à ces services en utilisant l'API JNDI



---

# Architecture JNDI

- JNDI est inclus dans le JDK Standard (Java 2 SDK) depuis la version 1.3. Pour utiliser JNDI, vous devez posséder les classes JNDI et au moins un SPI (JNDI Provider).
- La version 1.4 du JDK inclut 3 SPI pour les services de nommage/annuaire suivant :
  - Lightweight Directory Access Protocol (LDAP)
  - Le service de nommage de CORBA (Common Object Request Broker Architecture) Common Object Services (COS)
  - Le registre de RMI (Remote Method Invocation) rmiregistry
  - DNS

# Serveur RMI utilisant JNDI

```
package service; import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
public class ServeurRMI {
public static void main(String[] args) {
try {
    // Démarrer l'annuaire RMI REgistry
    LocateRegistry.createRegistry(1099);
    // Créer l'Objet distant
    BanqueImpl od=new BanqueImpl();
    // Afficher la référence de l'objet distant
    System.out.println(od.toString());
    // Créer l'objet InitialContext JNDI en utilisant le fichier
    jndi.properties
    Context ctx=new InitialContext();
    // Publier la référence de l'objet distant avec le nom BK
    ctx.bind("SD", od);
} catch (Exception e) { e.printStackTrace();}
}
}
```

**Fichier jndi.properties :**

```
java.naming.factory.initial=com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url=rmi://localhost:1099
```

# Client RMI utilisant JNDI

```
import javax.naming.*;
import rmi.IBanque;
public class ClientRMI {
public static void main(String[] args) {
try {
    Context ctx=new InitialContext();
    ctx.addToEnvironment(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.rmi.registry.RegistryContextFactory");
    ctx.addToEnvironment("java.naming.provider.url",
        "rmi://localhost:1099");
    IBanque stub=(IBanque) ctx.lookup("BK");
    System.out.println(stub.test("test"));
} catch (Exception e) {
e.printStackTrace();
}}}
```

- Dans ce cas, on pas besoin de créer le fichier jndi.properties