Les EJB 3.2

12/01/2017

Walid YAICH
walid.yaich@esprit.tn
Bureau E204

Plan

- EJB (Enterprise Java Bean)
- Types d’EJB
- Accés aux EJBs - local ou remote
- Accès local - Injection de dépendances
- Accès distant à un EJB - JNDI
- Session Bean - Stateless (sans état)
- Session Bean - Stateless - Lifecycle
- Session Bean - Stateful (avec état)
- Session Bean - Stateful - Lifecycle
- Session Bean - Singleton
- Création du premier EJB (sayHello)
- Création du client java (appel a sayHello)
- Exercice - Compte bancaire

EJB (Enterprise Java Bean)

- Les EJBs sont des **classes java** côté serveur.

- Le **conteneur EJB** est un environnement d'exécution pour les **EJB déployés** au sein du serveur d'application.

- Le **conteneur EJB** se lance au démarrage du serveur d'application.

- Le **conteneur EJB** gère le **cycle de vie des EJBs** (instantiation, injection, destruction ...).

Types d'EJB

Session Bean

- Modélise un traitement
- Représente: Une action, un verbe
Une logique métier, un algorithme
Un enchainement de tâches…
Exemple: *Saisie d'une commande, gestion d'un caddy ou d'un catalogue de produits, etc….*

- Statefull
- Stateless
- Singleton

Message Driven Bean (MDB)

- Consomme des messages asynchrones envoyés par des clients
- Permettent l'interconnexion avec des systèmes différents (non Java EE)

Accés aux EJBs - local ou remote

Selon d'où un client invoque un l'EJB session, la classe de ce dernier devra :

- Implémenter des interfaces locales (@Local)

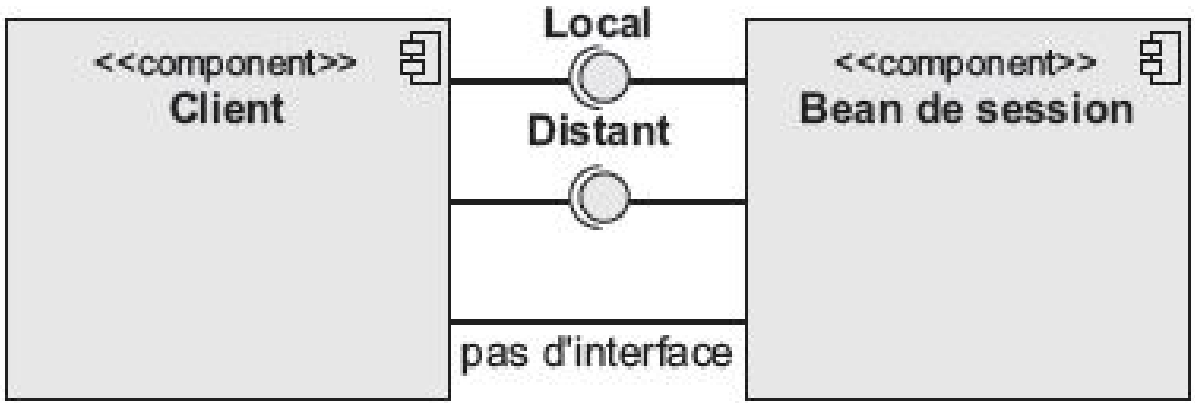
ou

- Implementer des interfaces distantes (@Remote)

```
@Remote
public interface CompteBancaireRemote {

    String versement(String nomPrenom, int
```

```
@Local
public interface CompteBancaireLocal {

    String versement(String nomPrenom, int monta
```

Elle pourra aussi ne pas implémenter aucune interface pour le cas d'accès local.



Accès local - Injection de dépendances

```java
public class Personne {

    private Voiture voiture;

    public Personne() {
    }

    void choisirMarqueVoiture(String marque){
        //faire quoi pour donner une marque a la voiture de cette personne ?
    }

}
```

```java
public class Voiture {

    private String marque;

    public Voiture() {

    }

    public String getMarque() {
        return marque;
    }

    public void setMarque(String marque) {
        this.marque = marque;
    }


}
```

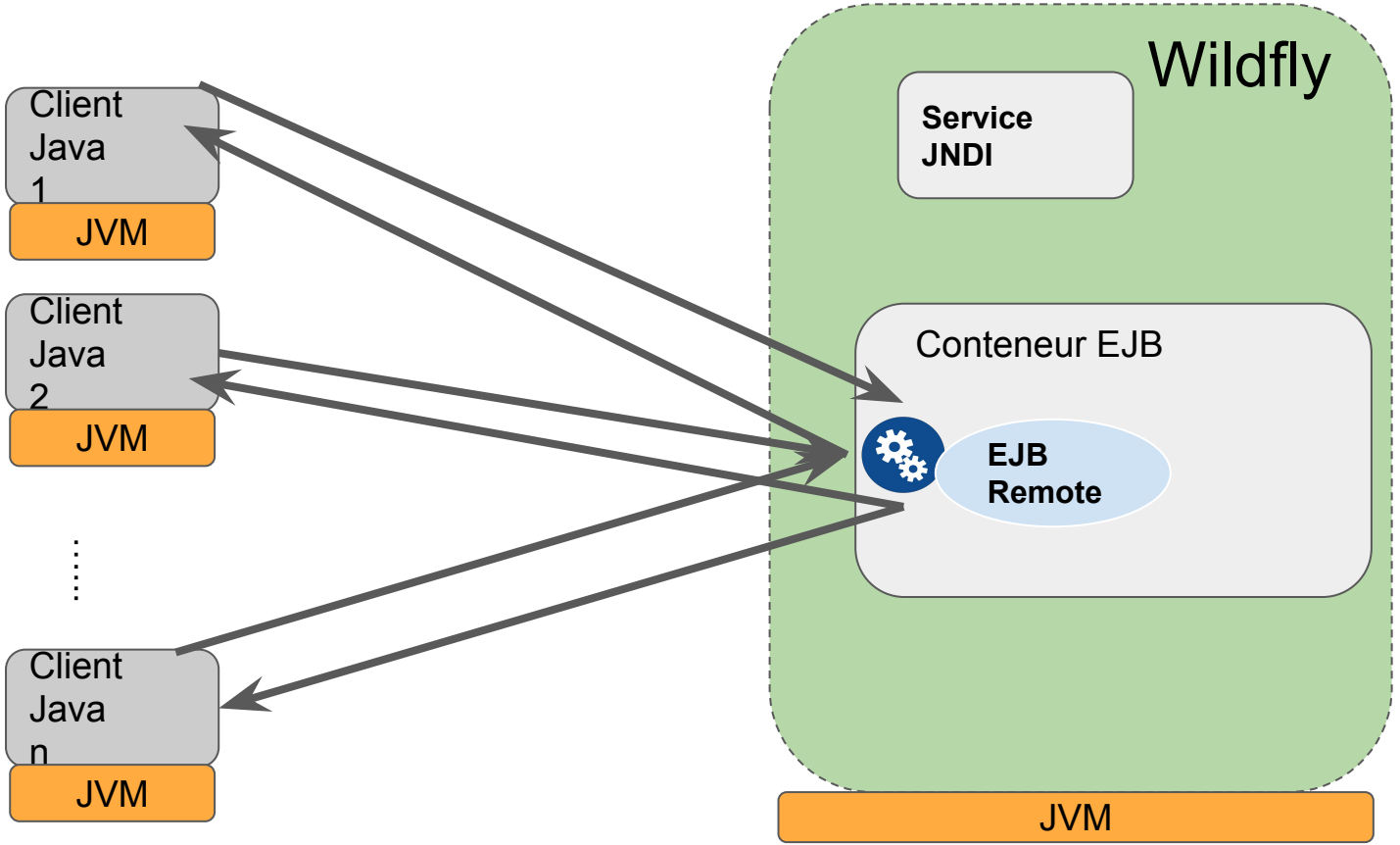
- Avec l'injection des dépendances, on n'invoque plus le constructeur nous même dans notre code, mais c'est plutôt le conteneur EJB qui invoque le constructeur.
- Pour injecter une dépendance il suffit d'annoter l'attribut avec @EJB.

Par exemple :

@EJB

private Voiture voiture; //La classe Voiture devra être un EJB

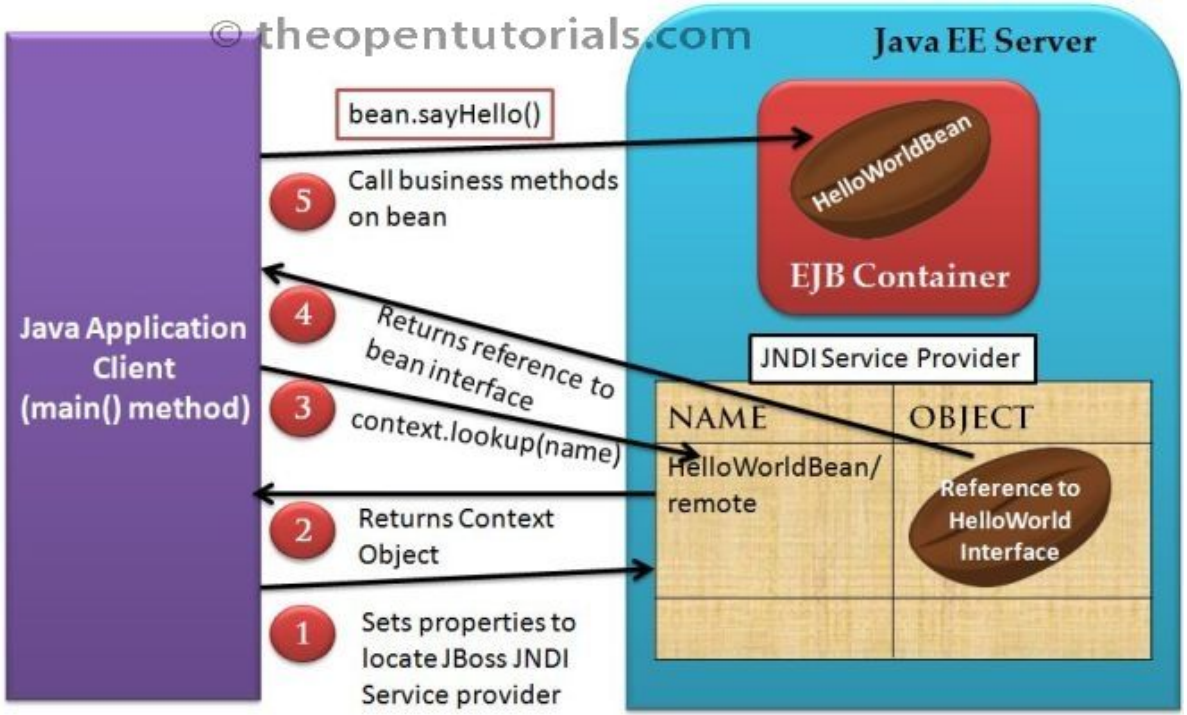
Accès Remote à un EJB



7

Accès distant à un EJB - JNDI (**Java Naming and Directory Interface**)

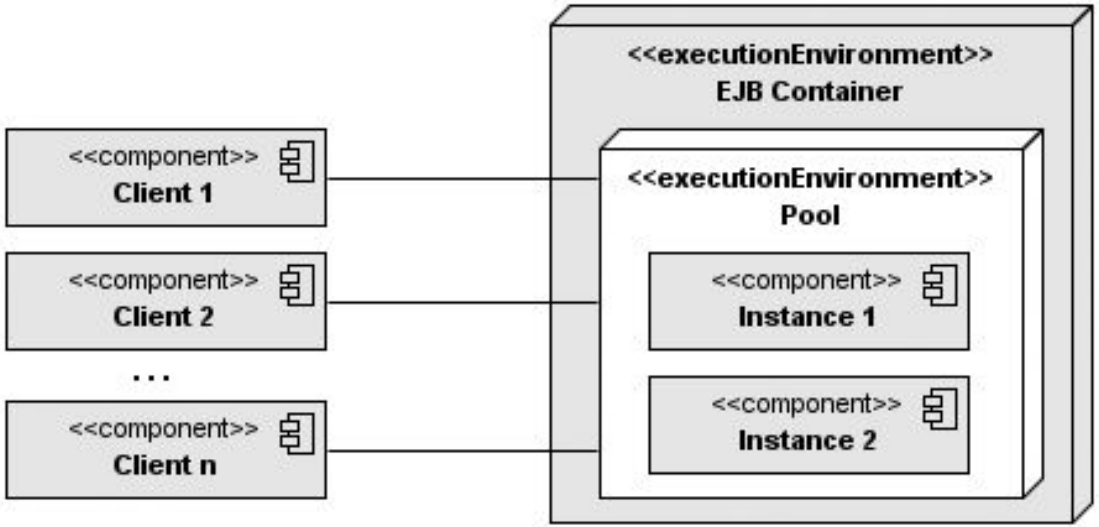
```
String jndiName="firstejb/HelloService!tn.esprit.firstejb.services.HelloServiceRemote";
Context context = new InitialContext();
HelloServiceRemote proxy=(HelloServiceRemote) context.lookup(jndiName);
System.out.println(proxy.sayHello("hello"));
```



Lorsqu'un client distant appelle un EJB, il ne travaille pas directement avec une instance de cet EJB mais avec un **proxy** qui le présente sur le client.

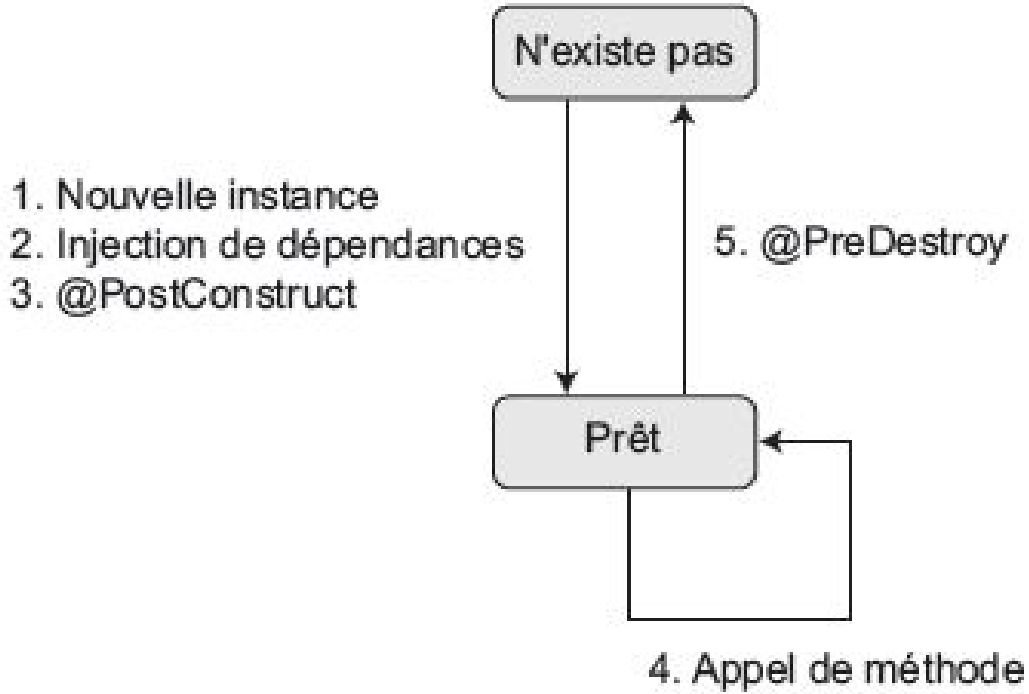
Session Bean - Stateless (sans état)

- le conteneur conserve en mémoire un certain nombre d'instances (un **pool**) de chaque EJB sans état et les partage entre les clients.
- Ces beans **ne mémorisent pas** l'état des clients.
- Lorsqu'un client appelle une méthode d'un bean sans état, le conteneur **choisit une instance du pool** et l'affecte au client.
- lorsque le client en a fini, **l'instance est retournée dans le pool** et pourra être réutilisé par un autre client.



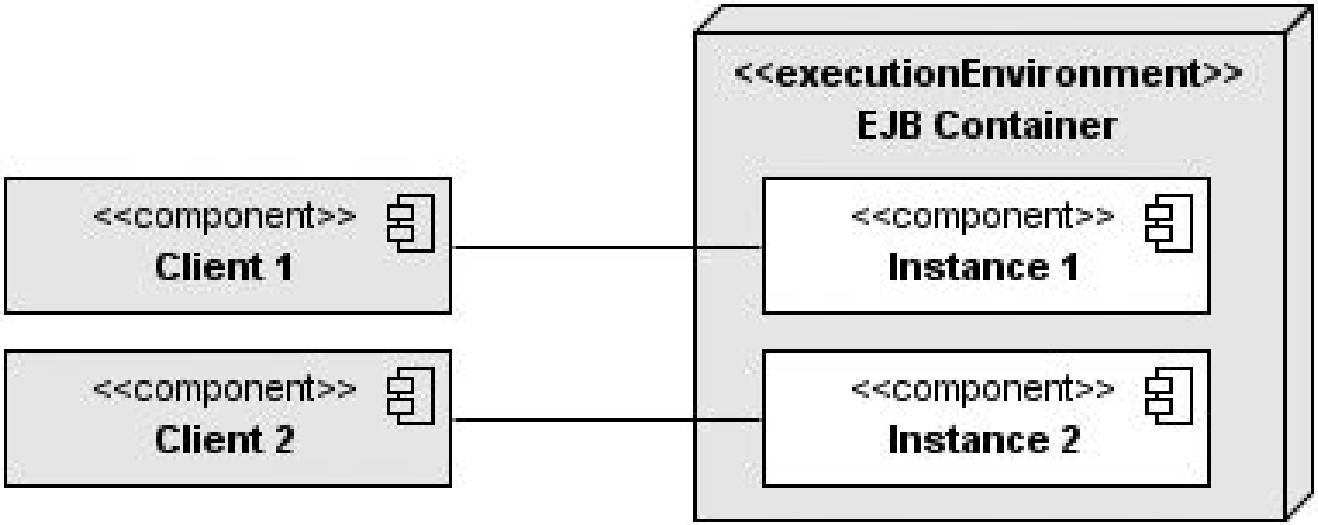
Il est possible de désactiver le pooling dans le serveur d'application wildfly.

Session Bean - Stateless - Lifecycle

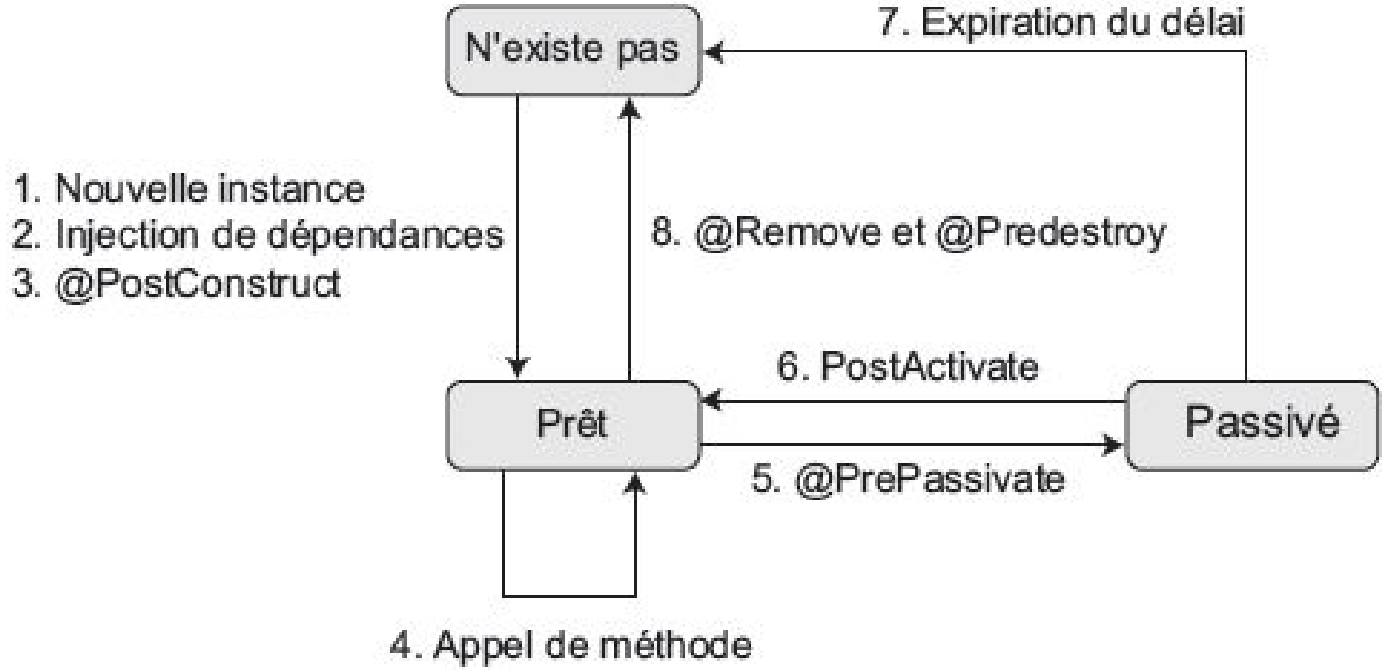


Session Bean - Stateful (avec état)

- Quand un client invoque un EJB avec état sur le serveur, le conteneur EJB doit fournir la **même instance a chaque appel de méthode**.
- Cette instance **ne peut pas** être réutilisé par un autre client.



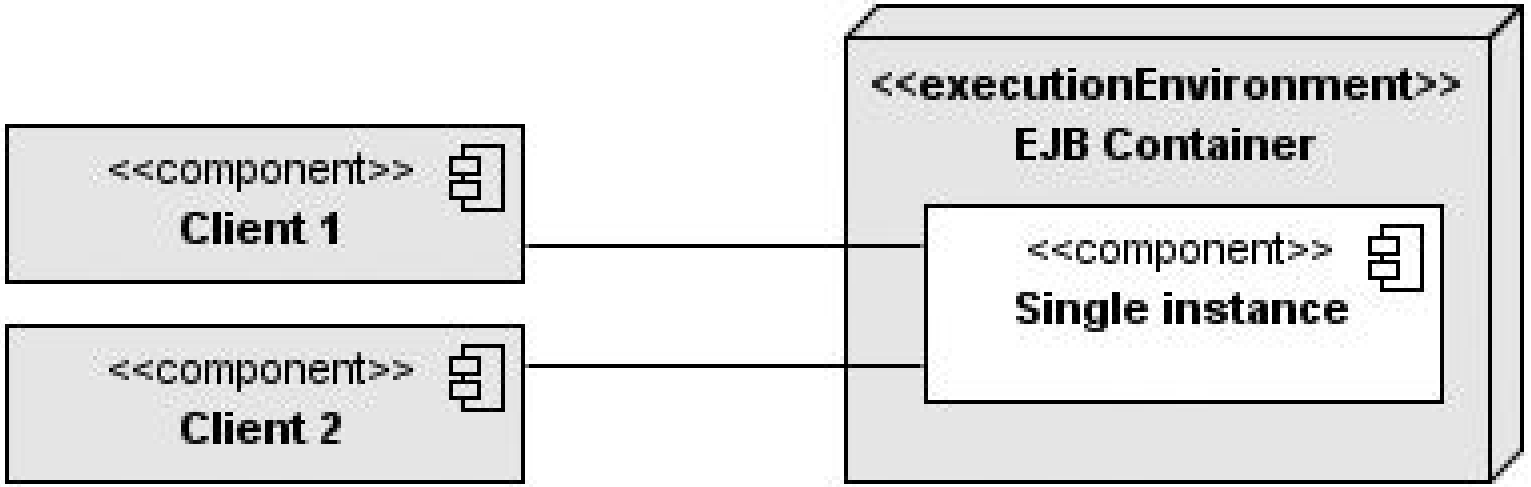
Session Bean - Stateful - Lifecycle



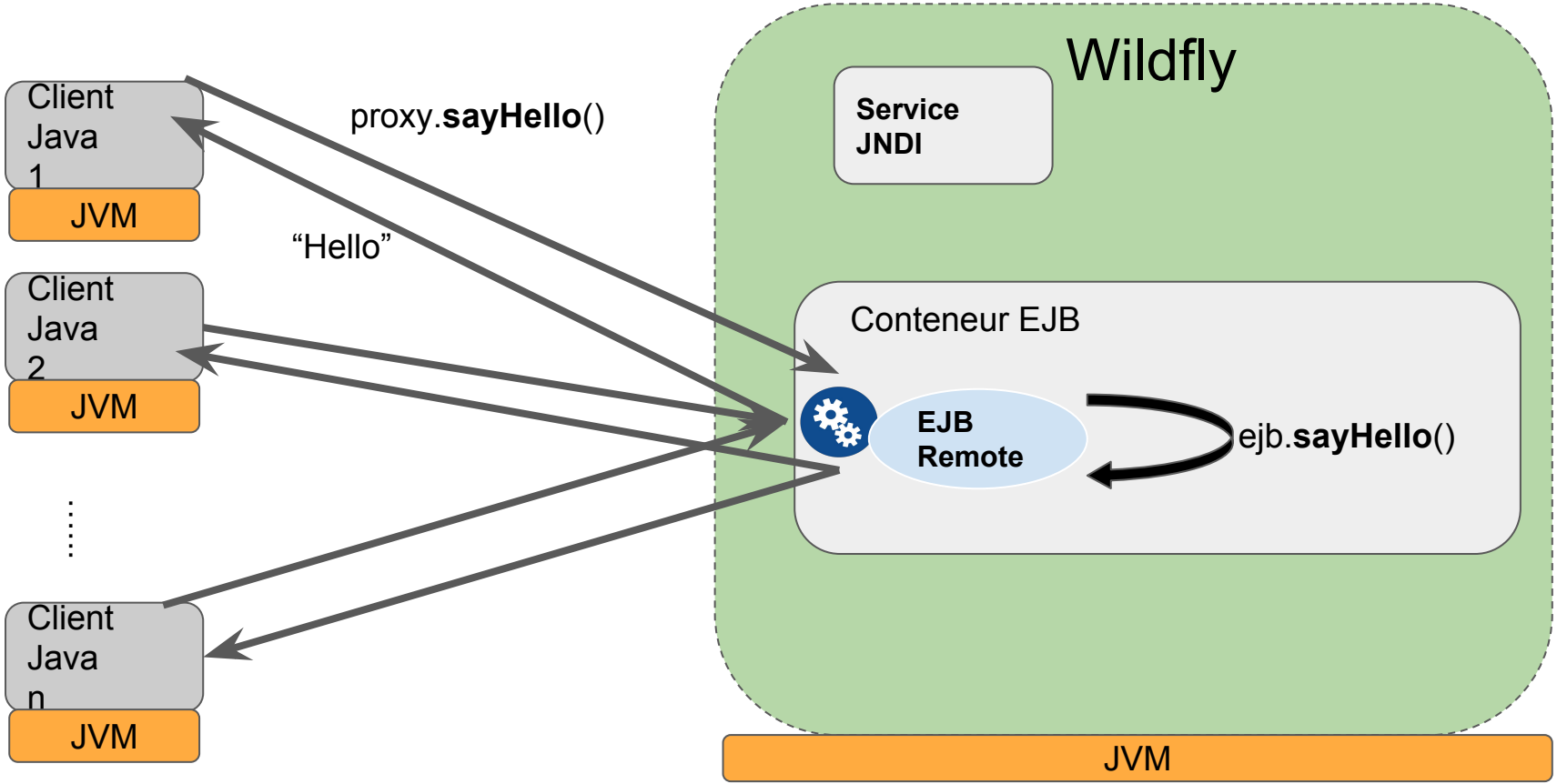
Session Bean - Singleton

- Un EJB singleton est simplement un bean de session qui n'est **instancié qu'une seule fois** par application.

- Meme cycle de vie que Stateful (Slide précédent)



Création du premier EJB (sayHello)



Création du premier EJB (sayHello)

1) File->new->project; Maven Project
2) Cocher “create simple project”
3) Spécifier le group id et l’artifact id
4) Finish
5) Aller au pom.xml, ajouter le packaging ejb, la dépendance java ee et le plugin EJB
6) Bouton droit sur le projet; maven->update project.
7) Vérifier la view “problems”

```
<project xmlns="http://maven.apache.org/
  <modelVersion>4.0.0</modelVersion>
  <groupId>tn.esprit.firstejb</groupId>
  <artifactId>firstejb</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>ejb</packaging>

  <dependencies>
      <dependency>
          <groupId>javax</groupId>
          <artifactId>javaee-api</artifactId>
          <version>7.0</version>
          <scope>provided</scope>
      </dependency>

  </dependencies>

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ejb-plugin</artifactId>
        <version>2.5</version>
        <configuration>
            <ejbVersion>3.2</ejbVersion>
            <jarName>${project.artifactId}</jarName>
        </configuration>
    </plugin>
```

Création du premier EJB (sayHello)

8) Créer deux interfaces, HelloServiceLocal et HelloServiceRemote

```java
import javax.ejb.Remote;

@Remote
public interface HelloServiceRemote {
    String sayHello(String msg);
}
```

```java
import javax.ejb.Local;

@Local
public interface HelloServiceLocal {
    String sayHello(String msg);
}
```

9) Créer la classe HelloService qui implémente les deux interfaces

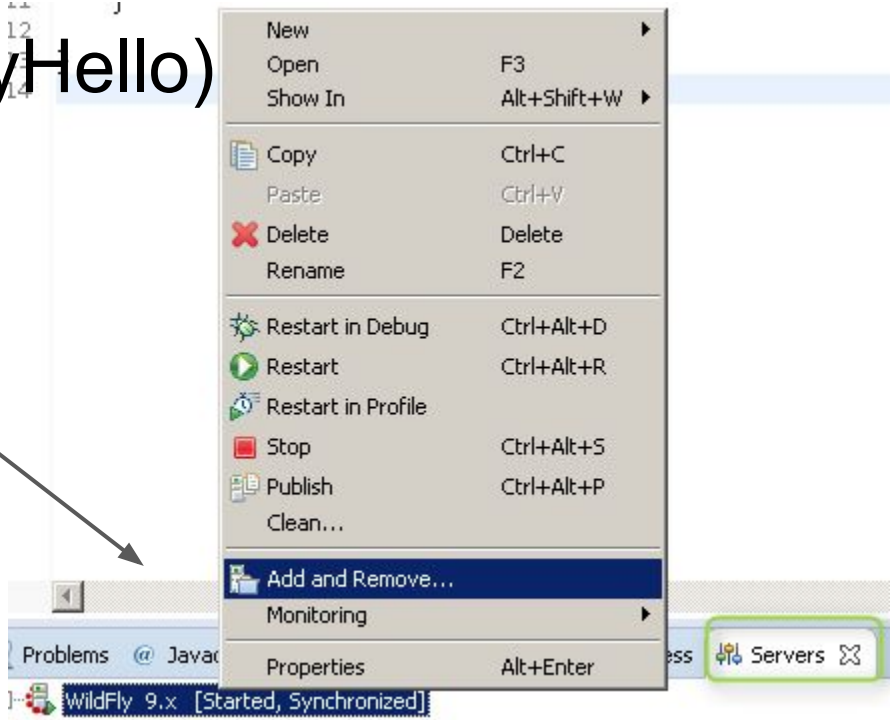
```java
import javax.ejb.Stateless;

@Stateless
public class HelloService implements HelloServiceLocal, HelloServiceRemote {

    @Override
    public String sayHello(String msg) {
        return msg;
    }

}
```

Création du premier EJB (sayHello)

10) clic droit sur le serveur wildfly et ajouter le projet en question, puis “finish”.

11) Démarrer le serveur

12) Dans les logs, on devra voir les EJB déployés.




```
java:global/firstejb/HelloService!tn.esprit.firstejb.services.HelloServiceRemote
java:app/firstejb/HelloService!tn.esprit.firstejb.services.HelloServiceRemote
java:module/HelloService!tn.esprit.firstejb.services.HelloServiceRemote
java:jboss/exported/firstejb/HelloService!tn.esprit.firstejb.services.HelloServiceRemote
java:global/firstejb/HelloService!tn.esprit.firstejb.services.HelloServiceLocal
java:app/firstejb/HelloService!tn.esprit.firstejb.services.HelloServiceLocal
java:module/HelloService!tn.esprit.firstejb.services.HelloServiceLocal
```

Création du client java (appel a sayHello)

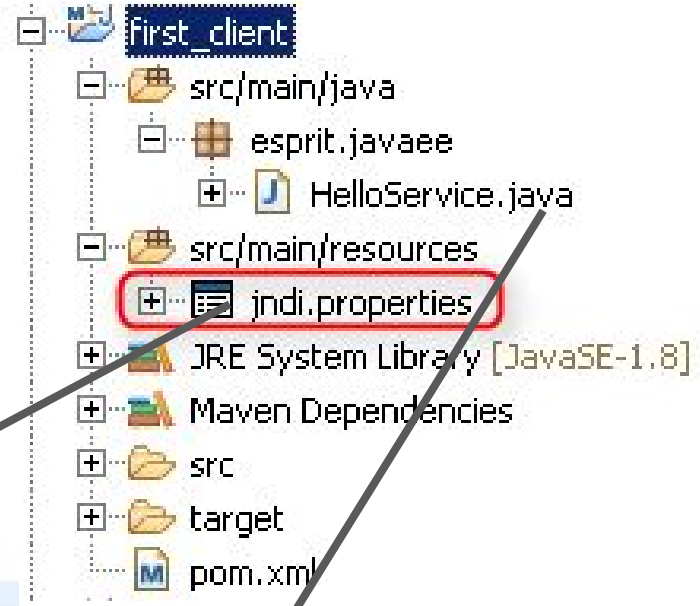
1) File->new->project; Maven Project
2) Cocher “create simple project”
3) Spécifier le group id et l’artifact id
4) Finish
5) Aller au pom.xml, ajouter la dépendance **wildfly-ejb-client-bom** et la dépendance du projet serveur ‘firstejb’.
6) Bouton droit sur le projet; maven->update project et cocher “force update of snapshot/releases”.
7) Vérifier la view “problems”

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
  <modelVersion>4.0.0</modelVersion>
  <groupId>tn.esprit</groupId>
  <artifactId>firstClient</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
      <maven.compiler.target>1.8</maven.compiler.target>
      <maven.compiler.source>1.8</maven.compiler.source>
  </properties>

  <dependencies>
      <dependency>
          <groupId>tn.esprit</groupId>
          <artifactId>firstEJB</artifactId>
          <version>0.0.1-SNAPSHOT</version>
          <type>ejb</type>
      </dependency>

      <dependency>
          <groupId>org.wildfly</groupId>
          <artifactId>wildfly-ejb-client-bom</artifactId>
          <version>9.0.0.Alpha1</version>
          <type>pom</type>
      </dependency>
  </dependencies>

</project>
```

Il faut pointer sur le projet serveur

Création du client java (appel a sayHello)

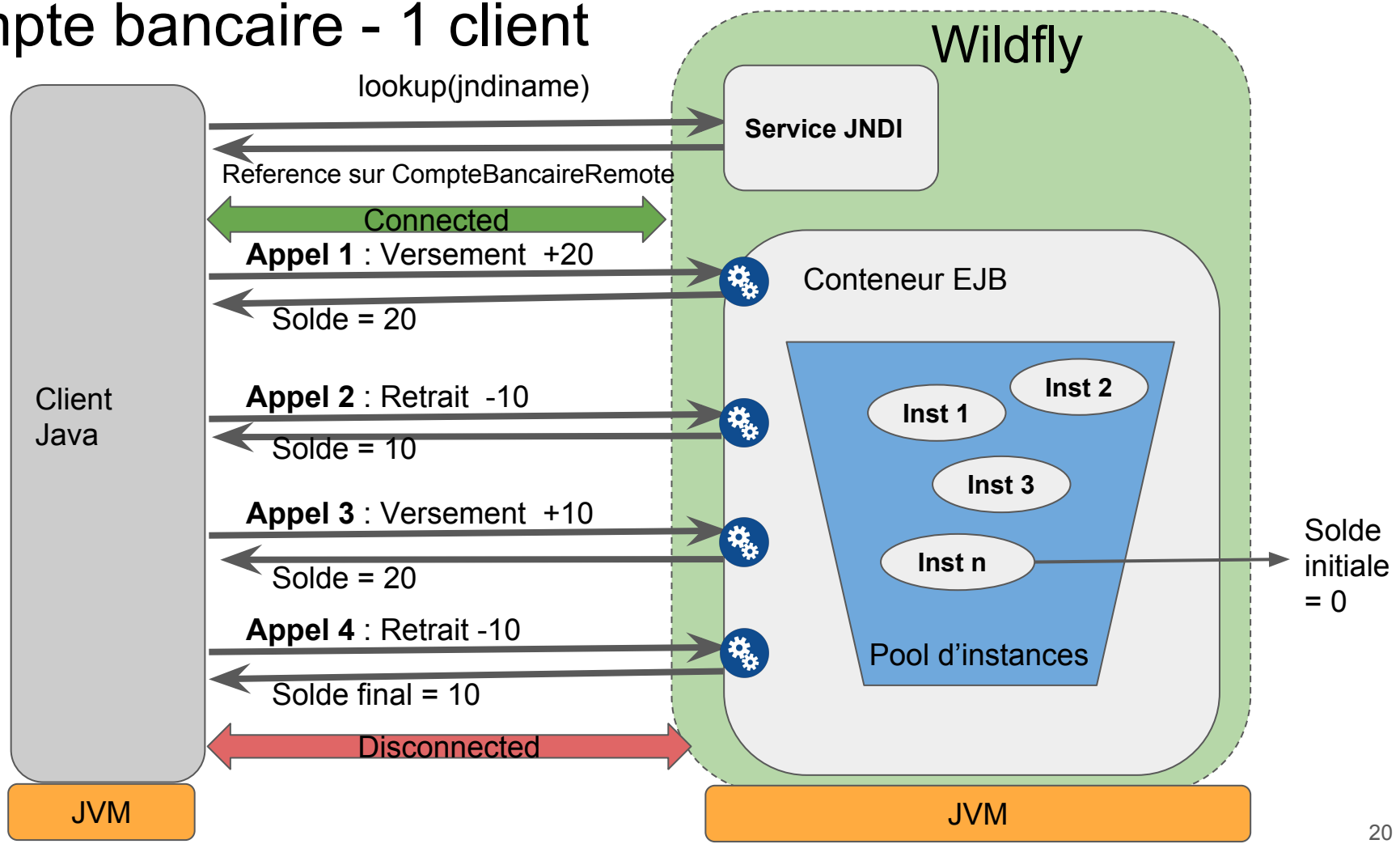
1) Créer une classe qui contient la méthode <main>. Le **jndiName** devra être récupéré de la console serveur (voir slide 17)
2) Ajouter le fichier jndi.properties
3) Sur la classe main, bouton droit, run as->java application (Output : hello)



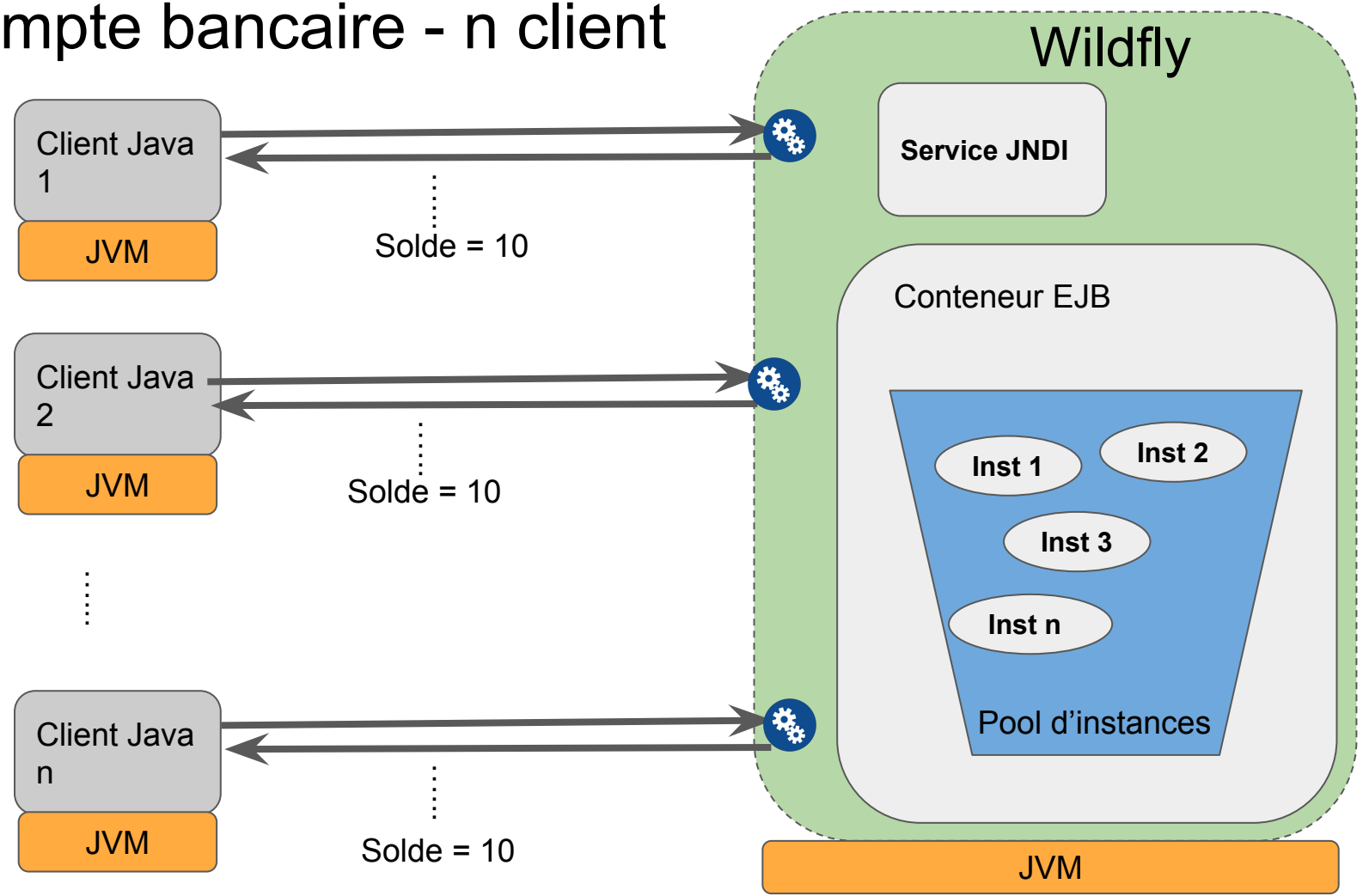

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://localhost:18080
jboss.naming.client.ejb.context=true
```

```
public class HelloService {

    public static void main(String[] args) throws NamingException {

        String jndiName="firstejb/HelloService!tn.esprit.firstejb.services.HelloServiceRemote";
        Context context = new InitialContext();
        HelloServiceRemote proxy=(HelloServiceRemote) context.lookup(jndiName);
        System.out.println(proxy.sayHello("hello"));
    }

}
```

Compte bancaire - 1 client



Compte bancaire - n client



Exercice - Compte bancaire

Créer un nouveau projet Maven (avec <packaging>ejb</packaging>) qui représente un compte bancaire :

1) Créer l'interface **remote**

```
@Remote
public interface CompteBancaireRemote {

    String versement(String nomPrenom, int montant);

    String retrait(String nomPrenom, int montant);

}
```

2) Créer **l'implémentation** en rajoutant 2 attributs, solde et nomPrenom.
   - Le solde initial est 0.
   - Chaque méthode devra changer le solde et retourner le nouveau solde au client java.
   - Chaque méthode devra logger sur la console du serveur le nom prénom, l'opération et le montant.
3) **Déployer** ce projet sur wildfly et vérifier l'affichage du **JNDI** name sur la console de Wildfly

Exercice - Compte bancaire

4) Créer le projet client java qui fait appel a l'EJB remote déjà déployé.

```
public static void main(String[] args) throws NamingException, InterruptedException {

    //Deux executions signifie 2 clients differents !
    String jndiName="[illegible]";
    Context context = new InitialContext();
    CompteBancaireRemote proxy=(CompteBancaireRemote) context.lookup(jndiName);
    System.out.println(proxy.versement("Walid_YAICH", 20)); //Le client demande un versement
    System.out.println(proxy.retrait("Walid_YAICH", 10));    //Le client demande un retrait
    System.out.println(proxy.versement("Walid_YAICH", 10));  //Le client demande un versement
    System.out.println(proxy.retrait("Walid_YAICH", 10));    //Le client demande un retrait
    //EXIT
}
```

L'output devra ressembler a :

Votre nouveau Solde est : 20

Votre nouveau Solde est : 10

Votre nouveau Solde est : 20

Votre nouveau Solde est : **10**

Exercice - Compte bancaire

5) Changer l'EJB CompteBancaire vers Stateless et expliquer le résultat
6) Changer l'EJB CompteBancaire vers Stateful et expliquer le résultat
7) Changer l'EJB CompteBancaire vers Singleton et expliquer le résultat
8) Même avec Stateful, entre deux exécutions du programme java client, le solde du client n'est pas maintenu, comment expliquer ce comportement ?
9) Pour maintenir le solde entre deux appels du client java, je propose de :

A. Créer une classe **Comptes** annoté "@singleton" qui contient une Hashmap<nomClient, solde>
B. Injecter la classe **Comptes** dans la classe CompteBancaire.
C. Créer deux méthodes **versementPermanent** et **retraitPermanent** (comme les méthodes versement et retrait) qui utilisent la classe **Comptes** pour stocker le couple (nomPrenom, solde) et pour récupérer le solde d'un client X.
D. Vérifier qu'entre deux exécutions du client java, le programme côté serveur mémorise le solde.

```
public class CompteBancaire implements CompteBancaireLocal, CompteBancaireRemote{

    @EJB //Injection de la classe Comptes dans la classe CompteBancaire
    Comptes comptes;
```

Compte bancaire - 3 clients en stateless

```
appel 1 : Vous etes user0, Vous avez utilisé le bean de instance1, votre nouveau Solde est : 20   + 20
appel 2 : Vous etes user0, Vous avez utilisé le bean de user0, votre nouveau Solde est : 10    - 10
appel 3 : Vous etes user0, Vous avez utilisé le bean de user2, votre nouveau Solde est : 20    + 10
appel 4 : Vous etes user0, Vous avez utilisé le bean de user0, votre nouveau Solde est : 10    - 10

appel 1 : Vous etes user1, Vous avez utilisé le bean de instance2, votre nouveau Solde est : 20   + 20
appel 2 : Vous etes user1, Vous avez utilisé le bean de user1, votre nouveau Solde est : 10    - 10
appel 3 : Vous etes user1, Vous avez utilisé le bean de user1, votre nouveau Solde est : 20    + 10
appel 4 : Vous etes user1, Vous avez utilisé le bean de user0, votre nouveau Solde est : 0     - 10

appel 1 : Vous etes user2, Vous avez utilisé le bean de instance3, votre nouveau Solde est : 20   + 20
appel 2 : Vous etes user2, Vous avez utilisé le bean de user2, votre nouveau Solde est : 10    - 10
appel 3 : Vous etes user2, Vous avez utilisé le bean de user1, votre nouveau Solde est : 30    + 10
appel 4 : Vous etes user2, Vous avez utilisé le bean de user1, votre nouveau Solde est : -10   - 10
```


Explication user1:

appel 1 : **user1** a demandé une instance, aucune instance disponible dans le pool
==> le conteneur EJB va lui créer une nouvelle instance "instance2".
appel 2 : **user1** a demandé une instance, il existe au moins une instance dans le pool
==>le conteneur EJB va lui choisir une instance par hasard, l'instance choisi est celle qui l'a utilisé dans l'appel1
appel 3 : **user1** a demandé une instance, il existe au moins une instance dans le pool
==>le conteneur EJB va lui choisir une instance par hasard, l'instance choisi est celle qui l'a utilisé dans l'appel2
appel 4 : **user1** a demandé une instance, il existe au moins une instance dans le pool
==>le conteneur EJB va lui choisir une instance par hasard, l'instance choisi est **celle du user0 (qui contient une mauvaise valeur)**
==========> résultat incorrect, le résultat attendu est 10 alors que le résultat obtenu est 0