POLYTECH NANCY

TP 1 : Brefs rappels

Exercice 1 *(RMI)*

L'objectif de cet exercice est de vous introduire, à travers un exemple simple, les étapes à suivre pour publier un objet distant, en utilisant RMI (*Remote Method Invocation*), dont l'objectif est d'exposer du code au travers d'interfaces. Autrement dit, pour que le client puisse interroger le serveur, il aura besoin uniquement des interfaces, ce qui éviterait de générer du code (métier) côté client. Concrètement, l'appel des méthodes se fait via un proxy (*stub*) du côté client et le *skeleton* du côté serveur.

Un exemple simple serait de mettre à disposition des utilisateurs une méthode permettant la conversion d'un montant, passé en paramètre, de l'€ au $.

Commençons par développer la partie serveur, en suivant les étapes suivantes.

— Créer un projet Java *RmiServeur*.

— Créer une interface *InerfaceRemote*. Cette interface doit hériter de l'interface *java.rmi.Remote* et toutes ces méthodes doivent lever l'exception *RemoteException*.

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  public interface InterfaceRemote extends Remote{
4      public double convertir(double mt) throws RemoteException;
5  }
```

— Définir une classe *ConversionServiceI* qui implémente l'interface *InerfaceRemote*. Cette classe hérite de la classe *UnicastRemoteObject*, fournie par RMI, qui permet de définir, entre autre, un *Skeleton*. En effet, à chaque définition d'un objet, un *skeleton* est crée permettant d'établir la communication à distance (voir le cours sur les *Sockets*).

— Définir un constructeur sans paramètres qui, lui aussi, lève l'exception *RemoteException*. Vous pouvez fixer le numéro de port à utiliser, en définissant un paramètre de type entier.

```
1  import java.rmi.RemoteException;
2  import java.rmi.server.UnicastRemoteObject;
3  public class ConversionServiceI extends  UnicastRemoteObject
4  implements InterfaceRemote {
5      protected ConversionServiceI() throws RemoteException{
6          super();
7      }
8      @Override
9      public double convertir(double mt) throws RemoteException {
10         return mt*1.9;
11     }}
```

— Une fois l'interface et l'implémentation sont définies, la prochaine étape consiste à définir un serveur. Pour cela, créer une classe *ServeurRmi*. Dans la méthode principale, on crée d'abord un objet distant et puis on publie sa référence dans l'annuaire, sans oublier de démarrer l'annuaire (soit à l'intérieur de l'application elle même, en utilisant la classe *LocateRegistry* ou en ligne de commande).

Le code du serveur est donné ci-dessous (vous le trouverez aussi sur Arche) :

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.util.logging.Level;
import java.util.logging.Logger;
public class ServeurRmi {
    public static void main(String[] args) {
        try {
            // créer un objet distant ....
            ConversionServiceI od = new ConversionServiceI();
            // publier l'objet ...
            // nom de l'objet dans l'annuaire ...
            // démarrer le serveur, vous pouvez le faire aussi en ligne de commande ....
            LocateRegistry.createRegistry(1099);
            Naming.rebind("rmi://localhost:1099/Conversion", od);
            // afficher des informations sur l'objet distant ....
            System.out.println(od.toString());
        } catch (Exception ex) {
            Logger.getLogger(ServeurRmi.class.getName()).log(Level.SEVERE, null, ex);
        }}  }
```

Intéressons-nous à présent à la définition d'un client, permettant l'accès à distance et invoquer la méthode *convertir*. Pour cela, suivre les étapes suivantes.

- Définir une classe Java *ClientRmi*. Le client a besoin de l'interface de l'objet distant (copier donc l'interface dans le projet du client). On peut aussi générer un fichier .jar, c'est plus propre !
- Dans la méthode principale, créer un *Stub* et appeler la méthode *convertir*.

Le code du client est donné ci-après (vous le trouverez aussi sur Arche) :

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.logging.Level;
import java.util.logging.Logger;
public class ClientRmi {
    public static void main(String[] args) {
        try {
            // on a besion de l'interface ....
            // définition d'un stub ....
            InterfaceRemote stub = (InterfaceRemote)
            Naming.lookup("rmi://localhost:1099/Conversion");
            System.out.println(stub.convertir(11));
        } catch (Exception ex) {
            Logger.getLogger(ClientRmi.class.getName()).log(Level.SEVERE, null, ex);
        }}}
```

En s'inspirant de l'exemple précédent, proposer une solution pour gérer des comptes bancaires (définir la partie serveur et la partie client). Pour la persistance des objets, vous pouvez utiliser une base de données de type MySql ou les sauvegarder tout simplement dans un fichier text.

Exercice 2 *(Mapping XML, Json-Objet)*

L'objectif de cet exercice est d'introduire le mapping Objet-XML, en utilisant quelques annotations JAXB (*Java Architecture for XML Binding*), une spécification qui permet de faire une

correspondance entre un objet d'une class et un document XML et vice versa, en utilisant deux opérations appelées respectivement sérialisation et désérialisation (*marshalling/unmarshalling*).

```
public class Serialisation {
    public static void main(String[] args) {
        try {
            JAXBContext jc = JAXBContext.newInstance(Personne.class);
            Personne p_1 = new Personne("Youcef", "Augustin", new Date(17, 3, 2017));
            Marshaller m = jc.createMarshaller();
            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
            m.marshal(p_1, new File("resultat.xml"));
        } catch (Exception ex) {
            Logger.getLogger(Serialiser.class.getName()).log(Level.SEVERE, null, ex);
        }}}
```

— Changer le nom de l'entête du document XML généré.
— Quelle annotation doit-on utiliser pour que l'attribut *solde* ne soit pas un élément mais un attribut dans le document XML.
— Quelle annotation doit-on utiliser pour que l'attribut *solde* n'apparaisse pas dans le document XML.
— Quelle annotation doit-on utiliser pour imposer un ordre d'apparition des élément dans le document XML.

Vous trouverez à cette adresse `http://docs.oracle.com/javaee/6/api/javax/xml/bind/annotation/package-summary.html` d'autres annotations.

Intéressons-nous maintenant à l'opération inverse, c'est-à-dire générer un objet à partir d'un document XML.

```
public class Desirialisation {
    public static void main(String[] args) {
        try {
            JAXBContext jc = JAXBContext.newInstance(Personne.class);
            Unmarshaller um = jc.createUnmarshaller();
            Personne p = (Personne)um.unmarshal(new File("resultat.xml"));
            System.out.println(p.getNom());
        } catch (Exception ex) {
            Logger.getLogger
            (Desirialisation.class.getName()).log(Level.SEVERE, null, ex);
        }}}
```

En s'inspirant des deux exemples précédents, proposer une solution pour la sérialisation et la désirialisation d'une liste de personnes.

Pour générer un schéma XML à partir d'une classe java, on utilise l'interface *SchemaOutputResolver*, en spécifiant dans la méthode *createOutput* l'emplacement où sera sauvegarder le schéma XML généré. Un exemple d'utilisation de cette interface est donnée ci-après.

```
public class GenerationSchema {
    public static void main(String[] args) {
        try {
            JAXBContext jc = JAXBContext.newInstance(Personne.class);
            jc.generateSchema(new SchemaOutputResolver() {
                @Override
 public Result createOutput(String namespaceUri, String suggestedFileName)
 throws IOException {
                    File f = new File("personne.xsd");
                     StreamResult r = new StreamResult(f);
```

```
                    return r; }
            });
        } catch (Exception ex) {
            Logger.getLogger
            (GenerationSchema.class.getName()).log(Level.SEVERE, null, ex);
        }}}
```

Quant à la génération d'une classe à partir d'un schéma XML, on peut utiliser l'utilitaire xjc (Java Compiler).

Intéressons-nous maintenant à la séarilisation des objets en Json (*JavaScript Object Notation*), un format de données textuelles dérivé de la notation des objets du langage JavaScript.

Il existe plusieurs librairies Java pour parser des flux Java en Json et vice versa, dont la plus populaire est Jackson, que vous trouverez sur Arche à l'adresse suivante `https://arche.univ-lorraine.fr/course/view.php?id=19522`. Une fois la librairie ajoutée à votre projet, vous définissez un objet de la classe *ObjectMapper* pour appeler ensuite la méthode *writeValue*, en spécifiant le fichier de sauvegarde du flux Json et l'objet concerné par cette sérialisation.

```
Personne p_1 = new Personne("Youcef", "Samir", new Date(1, 1, 1980));
ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(new File("resultats.json"), p_1);
```

Quant à la transformation d'un flux Json en objet Java, il suffit de faire appel à la méthode *readValue* de la classe *ObjectMapper*. Un exemple simple est donné ci-après.

```
ObjectMapper mapper = new ObjectMapper();
String s =  "{\"nom\": \"Youcef\", \"prenom\": \"Augustin\"}";
Personne p_2 = mapper.readValue(s, Personne.class);
```

Exercice 3 *(Bref rappel sur les Servlets)*

Sans rentrer dans les détails, l'objectif dans cet exercice est de vous montrer à travers un exemple simple - mais suffisant pour développer notamment un clients d'un service Web sous-forme de *servlet* - les étapes à suivre pour développer une *servlet* simple [1]. Ce étapes sont illustrées à travers un exemple permettant à un utilisateur de saisir deux entier et d'afficher la somme, que vous trouverez sur Arche.

1. C'est uniquement pour des raisons pédagogique que l'on utilisera des servlet, car il existe d"autres façon plus simples, que l'on verra dans les prochaines séances, pour consommer un service web