Contents

1 Astuces et bonnes pratiques pour la programmation distribuée avec Java RMI

1.1 Rappels

La programmation en Java RMI permet à un objet s'exécutant dans une JVM d'invoquer des méthodes sur un objet s'exécutant dans une autre JVM (i.e., un objet distant s'exécutant dans un autre processus) et donc de construire des applications Java distribuées. Elle permet d'abstraire les détails de connexion, de transport, et de sérialisation/désérialisation des objets communiquant. Du point de vue de l'utilisateur, ceci se traduit par une interaction avec l'objet distant comme s'il était local (i.e., dans la même JVM que l'objet appelant).

Les composants figurant dans une application Java RMI sont le client, le serveur et son squelette, le registre RMI, les objets distribués et leurs stubs (proxies), et les codebases. Chaque entité ainsi devra avoir sa propre description et son propre rôle afin de pouvoir l'exploiter correctement et d'assurer une bonne conception de l'application.

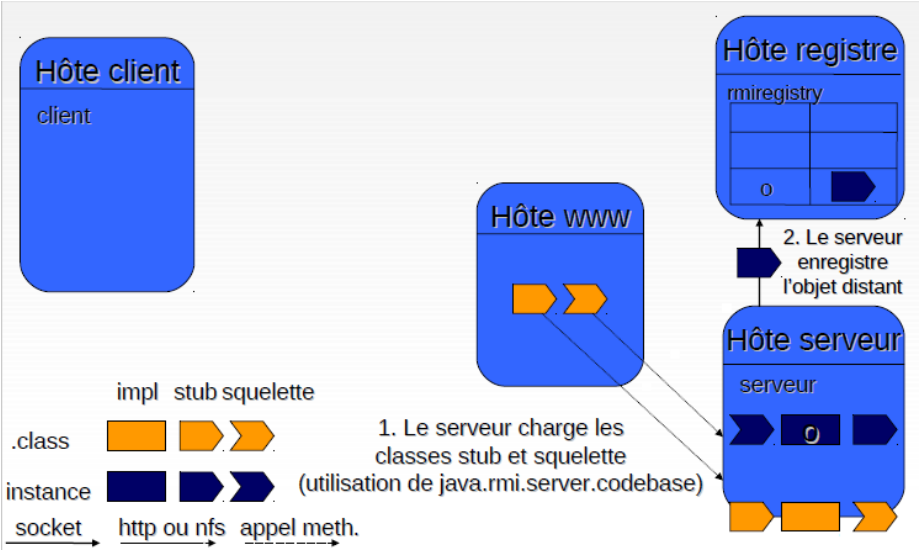

Figure 1: Enregistrement d'un OD

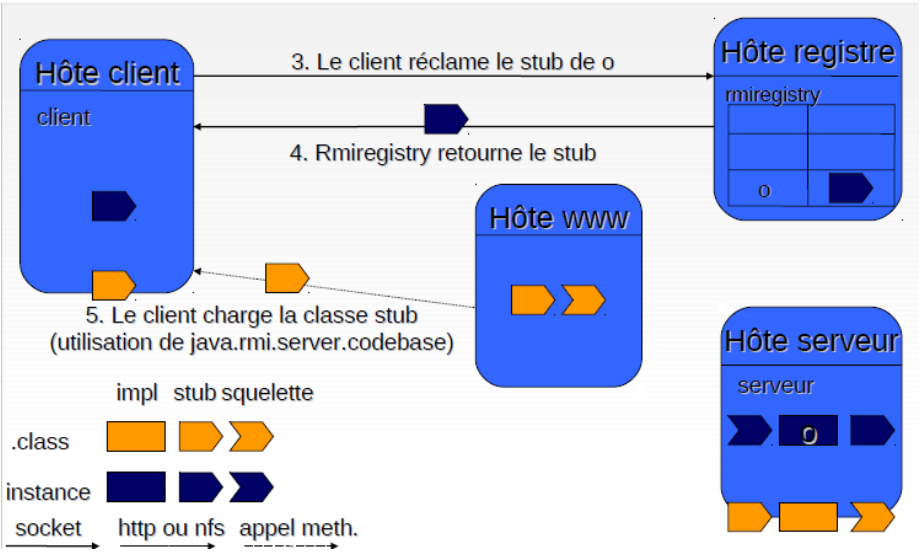

Figure 2: Récupération du stub

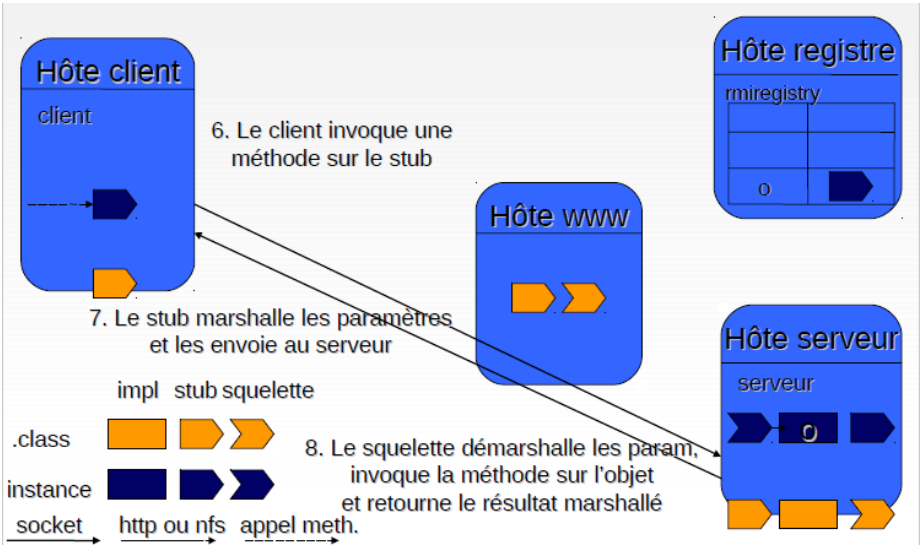

Figure 3: Invocation d'une méthode sur l'OD

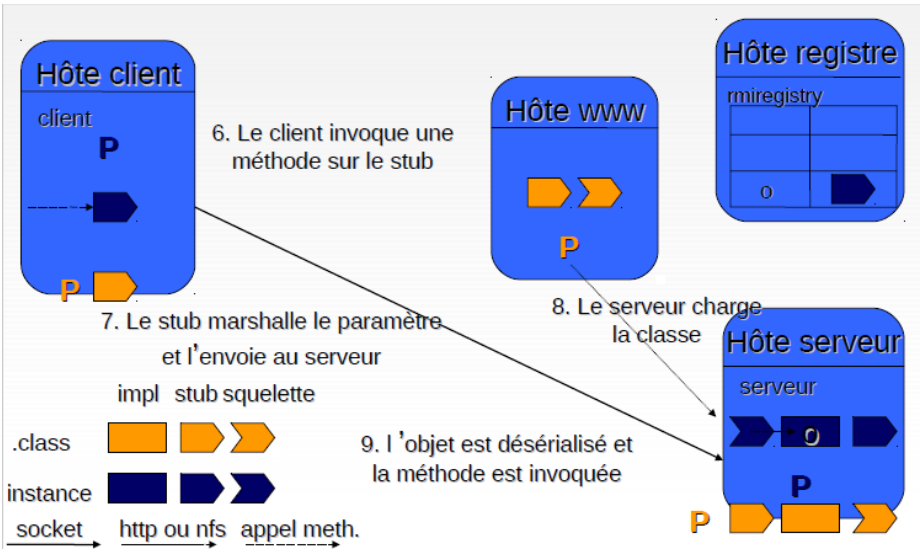

Figure 4: Passage d'un paramètre d'une classe inconnue du serveur

1.2 Workflow

1.2.1 Projets de l'application

Dans un environnement distribué, les composants d'une application distribuée tournent dans des processus différents, souvent localisés sur plusieurs machines. Dans le cadre de ce TP on créera un environnement distribué sur la même machine, en affectant à chaque composant de l'application son propre projet, tournant dans son propre processus. Pour distribuer notre application sur la même machine, on divise notre projet en trois :

1. un projet `common` contenant toutes les entités accessibles au serveur et client, telles que les interfaces des objets distants (*`java.rmi.Remote` ou toute sous-interface de celle-ci*) et éventuellement des classes désignant des objets non distants mais sérialisables (*implémentant l'interface `java.io.Serializable`*).
2. un projet client contenant la classe `Client` et la/les classe(s) qui devrai(en)t être reconnue(s) par le serveur via le codebase.
3. un projet serveur contenant la classe `Serveur` et toutes les classes des objets distants, étendant `java.rmi.server.UnicastRemoteObject` et implémentant chacune l'interface correspondante de son objet distant (*i.e., la classe `ObjectDistantImpl` implémente l'interface `IObjetDistant`*).

Pour que les projets du serveur et du client puissent reconnaître les entités communes dans le projet common, il faut ajouter le **build path** de ce dernier à leurs build paths. Pour ce faire : clic droit sur le projet client/serveur → `Build Path` → `Configure Build Path...` → Sous l'onglet `Projects` clic sur `Classpath` → `Add` → ajouter le projet `common`.

1.2.2 Objets distants

1.2.2.1 Définition

un objet distribué est un objet qui implémente une interface distante (*i.e., l'interface `java.rmi.Remote` ou n'importe quelle sous-interface personnalisée de celle-ci*) et qui éventuellement héritera de `java.rmi.server.UnicastRemoteObject` (*bonne pratique pour hériter le comportement d'exportation de l'objet*). Ceci nous permettra de référencer notre objet distant au sein d'un registre RMI qui sera ensuite "queriable" par un client via la méthode `java.rmi.registry.Registry#lookUp(String name)`.

Chaque méthode d'une interface distante doit déclarer l'exception `RemoteException` dans sa clause `throws` lors de la déclaration de sa signature. Cette exception permet de capturer les différentes erreurs pouvant survenir lors de l'exécution dans un environnement distribué. Par exemple, une exception retournée lors de la coupure de la connexion entre les sockets utilisées pour établir une connexion réseau, ou lors de l'accès distant à une référence non existante, ...

```java
/* SYNTAXE GÉNÉRIQUE D'UNE INTERFACE DISTANTE */
// dans le projet commons
package rmi.commons.workflow;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IObjetDistant extends Remote {
    typeRetour methodeDistante([params]) throws RemoteException;
    ...
}

/* SYNTAXE GÉNÉRIQUE D'UNE CLASSE D'UN OBJET DISTANT */
// dans le projet serveur
package rmi.server.workflow;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

import rmi.commons.workflow.IObjetDistant;

public class ObjetDistantImpl extends UnicastRemoteObject {
    /* attributs */
    /* méthodes */
    @Override
    typeRetour methodeDistante([params]) throws RemoteException{
        // implémentation
    }
    ...
    // autres méthodes : getters/setters, implémentations privées internes, etc.
}
```

1.2.2.2 Passage par référence et passage par valeur

En général, un passage par valeur veut dire passage par copie de l'objet lui-même, alors qu'un passage par référence veut dire passage de l'adresse de l'objet au lieu de l'objet lui-même. Dans le cadre de Java, ceci est abstrait du programmeur. Tous les types primitifs (*char*, *byte*, *int*, *long*, *double*, *boolean*, ...) sont passés en valeur, alors que tous les objets sont passés par référence.

1.2.2.3 Passage par référence vs passage par valeur en Java RMI

Dans le cadre du Java RMI faire un passage par référence signifie faire un passage d'un objet distribué qui sera référencé par son objet proxy correspondant et c'est ce dernier qui sera utilisé par toute entité souhaitant utiliser l'objet distant et qui

sera retourné par le registre RMI. Alors, on ne peut pas faire passer un objet non distant par référence en Java RMI. On ne peut le faire que “par valeur”. Pour ce faire, il faut alors le rendre sérialisable, et cet objet sérialisable ne sera associé à aucun proxy le référençant.

1.2.2.4 Sérialisation vs. Désérialisation

Un objet sérialisable est un objet qui possède une représentation textuelle permettant de le stocker sur des supports de persistance, mais aussi de le transmettre à travers un réseau. Désérialiser un objet consisterai ainsi à le reconstruire à partir de sa représentation textuelle. En Java, rendre un objet sérialisable signifie qu'il doit implémenter l'interface `java.io.Serializable`.

1.2.2.4.1 Exemple

```java
public class Person implements Serializable {
    /* ATTRIBUTES */
    private String name;
    private int age;

    /* CONSTRUCTORS */
    public Person() {

    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    /* METHODS */
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "(Name: "+name+", Age: "+age+")";
    }
}
```

1.2.2.4.2 En Java RMI

Quand un client essaye d'invoquer une méthode sur l'objet distant, le proxy sérialise (ou **marshalle**) les arguments passés et l'envoie au serveur. Quand le serveur reçoit du client une invocation distante, il l'envoie au squelette qui désérialise (ou **démarshalle**) les arguments passés, invoque la méthode sur l'objet distant réel, et retourne le résultat au serveur qui le renvoie au client. La sérialisation et la désérialisation des arguments passés sont automatiquement gérées par Java RMI.

Pour plus d'informations sur la sérialisation/désérialisation en Java : https://dzone.com/articles/what-is-serialization-everything-about-java-serial.

1.2.2.5 Les proxys

Proxy est un patron de conception (*design pattern*) dans lequel un objet peut être référencé par un autre objet qui se comporte en tant que son intermédiaire. Les deux types d'objets sont conformes à une interface commune. L'avantage de cette conception est de pouvoir effectuer des contrôles et/ou des calculs avant/après l'invocation actuelle d'une méthode ciblée d'un objet. Toute méthode ciblée fait partie ainsi de l'interface commune d'un objet et de son proxy.

1.2.2.5.1 Exemple

```
package structural.proxy;

/**
 * an Internet interface that plays the role of Subject
 * in the Proxy design pattern.<br/>
 * It provides an interface for connecting to the Internet
 * that we want to limit access to using a proxy.
 * @author anonbnr
 */
public interface Internet {

    /* METHODS */
    /**
     * Connects to serverHost
     * @param serverHost An Internet host to which we wish to connect
     */
    void connectTo(String serverHost);
}

package structural.proxy;

/**
```

```
 * a RealInternet concrete class that plays the role of RealSubject
 * in the Proxy design pattern.<br/>
 * It implements the Internet interface to allow access to the Internet.
 * @author anonbnr
 *
 */
public class RealInternet implements Internet {

    /* METHODS */
    @Override
    public void connectTo(String serverHost) {
        System.out.println("Standard Console: Connecting to " + serverHost);
    }
}
```

```
package structural.proxy;

import java.util.ArrayList;
import java.util.List;

/**
 * a ProxyInternet concrete class that plays the role of Proxy
 * in the Proxy design pattern.<br/>
 * It provides a proxy to classes implementing Internet, particularly
 * to ban Internet connections to some hosts.
 * @author anonbnr
 *
 */
public class ProxyInternet implements Internet {

    /* ATTRIBUTES */
    /**
     * The proxied Internet connection
     */
    private Internet internet;

    /**
     * The list of banned sites
     */
    private static List<String> bannedSites;

    static {
        bannedSites = new ArrayList<>();
        bannedSites.add("abc.com");
        bannedSites.add("def.com");
        bannedSites.add("ijk.com");
```

```
        bannedSites.add("lnm.com");
    }

    /* METHODS */
    /**
     * Only allows the proxied Internet connection to connect to hosts
     * that are not in the banned sites, otherwise denies access to the host.
     * It also creates the Internet connection, only if it hasn't already
     * been created
     */
    @Override
    public void connectTo(String serverHost) {
        if (bannedSites.contains(serverHost))
            System.err.println("Standard Error: Access Denied to " + serverHost);

        else {
            if (internet == null)
                internet = new RealInternet();

            internet.connectTo(serverHost);
        }
    }
}

package structural.proxy;

/**
 * a Test class
 * @author anonbnr
 *
 */
public class Test {
    public static void main(String[] args) {
        Internet internet = new ProxyInternet();
        internet.connectTo("abc.com");
        internet.connectTo("google.com");
    }
}

/* OUTPUT */
// Standard Error: Access Denied to abc.com
// Standard Console: Connecting to google.com
```

1.2.2.5.2 En Java RMI

En Java RMI, les objets distants sont référencés par des stubs (proxy), qui seront liés à des noms uniques et enregistrés dans un registre RMI. Quand un client réclame un objet distant par son nom au registre RMI, celui-ci lui renvoie le stub associé à l'objet demandé. Ce stub sera ensuite utilisé par le client pour invoquer des méthodes sur l'objet distant. Le stub se chargera ainsi de sérialiser les paramètres des invocations, et faire les appels réseaux nécessaires pour communiquer avec le serveur. Lorsque le serveur reçoit l'appel, il l'envoie au squelette qui désérialise les paramètres, invoque la méthode sur l'objet distant réel, et retourne le résultat au serveur qui le renvoie au client.

Les objets stubs, le squelette, et leurs comportements sont automatiquement générés et contrôlés par Java RMI et on ne s'en soucie pas. Ainsi, on continuera à construire notre programme client comme si on était en train d'invoquer une méthode sur un objet local, alors qu'on est en train d'invoquer une méthode sur l'objet proxy d'un objet distant.

1.2.2.6 Dans le TP

Dans le cadre de ce TP, un animal est un objet distant qui devrait être manipulé par le client, et donc il faut qu'il implémente une interface distante (`IAnimal` qui dérive de l'interface `java.rmi.Remote`) et qu'il étend la classe `java.rmi.server.UnicastRemoteObject`. Par contre, son espèce est un objet qui ne devrait pas être manipulé directement par le client, mais qui devrait quand même être reconnue par lui lorsque ce dernier va manipuler des objets de type `IAnimal` (*des objets proxy*). Pour ce faire, l'espèce doit être ainsi déclarée en tant qu'une classe sérialisable dont l'implémentation doit être accessible au client et au serveur. C'est pour cette raison qu'on l'installe au sein du projet `commons`.

1.2.3 Serveur

Un serveur doit effectuer les fonctionalités suivantes :

- éventuellement créer une instance d'un registre RMI dans le même processus : `Registry registry = LocateRegistry.createRegistry(RMI_REGISTRY_PORT);` si l'outil `rmiregistry` a été lancé déjà via la ligne de commande dans un processus séparé.
- récupérer une instance du registre RMI : `Registry registry = getRegistry(RMI_REGISTRY_PORT);`

- créer les objets distants ;
- lier les objets distants à des noms uniques au sein d'un registre RMI ;
- recevoir une requête du client consistant à invoquer une méthode distante sur l'objet distant via son proxy, la traiter, et retourner le résultat.

Le serveur doit être mis en place dans l'ordre suivant :

1. mise en place de son gestionnaire de sécurité (et de sa politique de sécurité).
2. éventuellement mise en place de son codebase.
3. mise en place de sa politique de sécurité.
4. mise en place de l'instance du registre RMI.
5. instanciation et distribution d'objets distants dans le registre RMI.

1.2.3.1 Serveur et Main

En général, la méthode `main()` (côté serveur) est ajouté à la classe du serveur directement. Toutefois, ceci peut devenir limitant pour l'extensibilité et la réutilisabilité, surtout si on souhaite éventuellement avoir une hiérarchie de serveurs de différentes fonctionnalités et qui peuvent partager des fonctionnalités communes par héritage. Pour éviter ces limitations, on peut introduire une classe `Main` dédiée, qui contiendra la méthode `main()` créant et mettant en place une instance du serveur.

```
// Exemple
public class Main {
    public static void main(String[] args) {
        Server server = new Server();
        try {
            server.setUp();
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

1.2.4 Client

Un client doit effectuer les fonctionnalités suivantes :

- récupérer une instance du registre RMI.
- rechercher les objets distants d'intérêt.
- invoquer des méthodes sur un objet distant pour accomplir une logique métier (*e.g., afficher des informations sur un objet distant, créer/modifier/supprimer un objet distant, ...*)

1.2.4.1 Client et Main

En général, la méthode `main()` (*côté client*) est ajoutée à la classe du client directement. Toutefois, pour les mêmes raisons susmentionnées, on peut introduire une classe `Main` dédiée, qui contiendra la méthode `main()` créant et mettant en place une instance du client. De plus, cette classe se comportera

comme un contrôleur, faisant le lien entre les fonctionnalités du client et les interactions de l'utilisateur (*i.e., une CLI ou un GUI*).

```
// Exemple
/*
* Classe modélisant le contrôleur
* qui gère l'interaction entre le client
* et le serveur.
*/
public class Main {
    public static void main(String[] args) {
        try {

            Client client = new Client();
            client.setUp();
            client.lookupCabinet("Animal Care");
            ICabinet cabinet = client.getCabinet();
            System.out.println("Objet proxy de l'objet cabinet: "+cabinet);
            /*
            * OUTPUT
            * ======
            * Objet proxy de l'objet cabinet: Proxy[ICabinet,
            * RemoteObjectInvocationHandler [UnicastRef [liveRef:
            * [endpoint:[127.0.1.1:34553]
            * (remote),objID:[6cc201a2:17cc940e953:-7fff,
            * 604015115438757631]]]]]
            */
            browseCabinet(client, cabinet);

        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (NotBoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1.2.5 Gestionnaire et politique de sécurité

Un **gestionnaire de sécurité** pour une classe en Java permet de lui définir une politique de sécurité et de vérifier si cette politique de sécurité est bien respectée. Par défaut, une classe n'a pas un gestionnaire de sécurité. Pour en créer un : `System.setSecurityManage(new SecurityManager());`

Une **politique de sécurité** permet de contrôler les interactions possibles avec une classe en spécifiant des **permissions**. Toute interaction non autorisée par la politique de sécurité engendre une exception de type `java.security.SecurityException`. Par défaut, une classe n'a ni un gestionnaire de sécurité, ni une politique de sécurité. Pour lier une classe à un fichier définissant sa politique de sécurité : `System.setProperty("java.security.policy", "path/to/security.policy");`

Dans le cadre de ce TP, on créera une politique de sécurité à titre indicatif en donnant toutes les permissions à tout le monde. Pour ce faire :

```
// path/to/security.policy
grant {
    permission java.security.AllPermission;
};
```

1.2.6 Codebases

1.2.6.1 Motivation

Dans une prochaine étape de l'exercice on introduit la notion d'un cabinet médical qui contiendra la liste des animaux suivis et qui sera le point d'interfaçage entre le client et le serveur. Le cabinet médical introduira des méthodes permettant de consulter les animaux suivis, mais aussi d'en ajouter. Ainsi, un cabinet médical est un objet distant créé de la même manière que précédemment avec les objets animaux. On suppose que l'interface distante `ICabinetMedical` fournit plusieurs méthodes dont les méthodes suivantes pour ajouter des animaux :

```
boolean addAnimal(String name, String ownerName, String speciesName,
    int speciesAverageLife, String race,
    String state) throws RemoteException;
boolean addAnimal(String name, String ownerName, Species species,
    String race, String state) throws RemoteException;
```

La première permet d'ajouter un animal sans besoin de spécifier son espèce explicitement (*via une instance de la classe* `Species`). C'est-à-dire l'objet `Species` est créé à l'intérieur du constructeur de l'animal et dont les paramètres `String speciesName`, `int speciesAverageLife` sont passés via la méthode `addAnimal()`.

Par contre, la deuxième version de la méthode permet d'ajouter un animal en spécifiant explicitement son espèce via une instance de la classe `Species`. Ceci est possible parce que la classe `Species` est dans le projet `commons`, et donc le client et le serveur la reconnaissent. Ainsi l'objet `Species` est créé et initialisé du côté client *avant* d'être passé à la méthode `addAnimal()`. Le constructeur de l'animal ajouté se contentera ainsi d'affecter l'instance `Species` passée à l'attribut correspondant au niveau de l'instance de l'animal créé.

Voici donc deux implémentations simples de ces deux méthodes au niveau de la classe `CabinetMedicalImpl` (*côté serveur*) implémentant `ICabinetMedical` :

```
@Override
public boolean addAnimal(String name, String ownerName, String speciesName,
      int speciesAverageLife, String race, String state)
            throws RemoteException {
   IAnimal patient = new AnimalImpl(name, ownerName, speciesName,
         speciesAverageLife, race, state);
   return patients.add(patient);
}

@Override
public boolean addAnimal(String name, String ownerName, Species species,
      String race, String state) throws RemoteException {
   IAnimal patient = new AnimalImpl(name, ownerName, species, race, state);
   return patients.add(patient);
}
```

Voici les deux constructeurs de la classe AnimalImpl (côté serveur) implémentant IAnimal :

```
protected AnimalImpl(String name, String ownerName, String speciesName,
      int speciesAverageLife, String race, String state)
      throws RemoteException {
   this.name = name;
   this.ownerName = ownerName;
   this.species = new Species(speciesName, speciesAverageLife); // création de l'espèce
   this.race = race;
   this.followUpFile = new FollowUpFileImpl(state);
}

protected AnimalImpl(String name, String ownerName, Species species, String race,
      String state) throws RemoteException {
   this.name = name;
   this.ownerName = ownerName;
   this.species = species; // affectation de l'espèce créée auparavant
   this.race = race;
   this.followUpFile = new FollowUpFileImpl(state );
}
```

Maintenant supposons que le client souhaite ajouter au cabinet médical un `IAnimal` dont l'espèce est une sous-classe de `Species` (*e.g., une classe `Dog`*). Le client peut déclarer ainsi une classe `Dog` de son côté qui étend `Species` et ensuite utiliser la deuxième version de `addAnimal()` en lui passant une instance de `Dog` pour ajouter l'animal au cabinet médical. Toutefois, le serveur va se plaindre de son côté parce qu'il ne reconnaît pas la classe `Dog`. En effet, il ne connaît que la classe `Species` dans le projet `commmons` qui est ajouté à son class-

path. Donc on aura une exception de type `ClassNotFoundException` quand on essaye d'ajouter l'animal ayant une espèce `Dog` au cabinet médical.

1.2.6.2 Solution : Les codebases

Pour résoudre ce problème, il faut que le serveur puisse reconnaître la classe `Dog`, notamment son *bytecode* (*i.e.,* *`Dog.class`*). Pour ce faire on utilise ce qu'on appelle un **codebase** désignant un mécanisme **classpath distribué**. Il s'agit simplement d'un dossier contenant des fichiers bytecode de classes.

Chaque entité pourra avoir son propre codebase où seront installés les fichiers bytecode des classes qui devraient être accessibles aux autres composants de l'application. En gros, si le client définit et utilise des classes qui figurent dans l'application et qui doivent être reconnues par le serveur, il installera leurs fichiers bytecode dans son codebase, et le serveur ira pendant l'exécution consulter le codebase du client, où il trouvera ces fichiers bytecode qui n'existent pas directement dans son classpath. Bien sûr ça peut être l'inverse aussi ou le client ira chercher des classes qu'il ne reconnaît pas depuis le codebase du serveur.

Ce codebase peut être localisé sur une machine distante et servi par **un serveur web** via HTTP et qui pourra être consulté via son URL (*e.g.,* *`http://localhost/codebaseURI/`*), comme il peut être défini comme dossier sur le **système de fichiers** de la même machine d'un composant de l'application et pourra être consulté via son chemin (*e.g.,* *`file:/home/someUser/path/to/codeBase`*).

Pour spécifier un/plusieurs codebases qui pourront être consultés par un composant, on spécifie leurs URIs via la propriété `"java.rmi.server.codebase"` du composant concernée de la manière suivante : `System.setProperty("java.rmi.server.codebase", "URICodebase1 URICodebase2 ...");`. Dans notre cas, il faut définir cette propriété pour le serveur, vu que c'est lui qui va aller chercher l'implémentation de la classe `Dog` depuis le codebase du client.

Il faut aussi faire attention en spécifiant le chemin du codebase; il faut que le dossier du codebase ne soit pas un package contenant la classe qu'on souhaite partager, parce que la classe est reconnue par son **nom complètement qualifié** (*Fully Qualified Name*).

Pour rappel, le nom complètement qualifié d'une classe désigne son nom et le nom du package qui la contient (*e.g., une classe* *`Animal`* *qui existe dans un package* *`com.example.some-package`* *aura* *`com.example.some-package.Animal`* *comme nom complètement qualifié*). Ceci est essentiel parce que deux classes peuvent avoir le même nom tout en étant localisées dans deux packages différents. Ainsi le seul moyen de les distinguer sera à travers leurs noms complètement qualifiés.

Dans un projet Java, tout dossier sous `src/` ou `bin/` est un package et donc ne peut pas être utilisé en tant qu'un dossier codebase. Alors si le serveur essaye

de chercher une classe du client qu'il ne reconnaît pas à partir du chemin du codebase, si on inclut le nom d'un package contenant la classe dans le chemin, le serveur ne pourra jamais la trouver. Dans cette situation :

1. soit on spécifie le chemin du dossier `bin/` du client comme codebase contenant tous les fichiers bytecode de toutes les classes du projet client (*ce qui peut être problématique si on ne souhaite pas exposer toutes les classes du client au serveur*).
2. soit on peut copier les fichiers bytecode des classes que l'on souhaite exposer et les mettre dans un dossier parent (*qui n'est pas un package*) dans le projet (*ou ailleurs sur le système de fichiers*). Ainsi, le chemin du codebase désignera le chemin de ce dossier parent.

1.2.6.3 Dernières remarques

1. Dans le cadre de ce TP nous avons créé un projet `commons` qui contiendra les interfaces distantes et les classes non distantes accessibles au client et serveur, puis on a ajouté son classpath aux classpaths du client et serveur. Cependant, ceci est une approche simplifiée pour le but du TP. En réalité, ce projet sera un codebase dédié qui sera consulté par le client et le serveur pour récupérer les fichiers bytecode des interfaces distantes et les classes non distantes qui leur sont accessibles.
2. Le client ne se comportera pas comme un serveur que si le serveur a besoin de manipuler explicitement des objets distants provenant du client, et dans ce cas là on peut parler d'un mode **Peer-to-Peer** (**P2P**). Si le serveur se contente uniquement de récupérer le fichier bytecode d'une classe `C` utilisée par le client et dont une instance est transmise au serveur sans que ce dernier puisse la reconnaître, ceci n'est pas du P2P. En effet, le serveur ne manipule pas explicitement des variables de type `C` dans son code.
3. Quand on a une application P2P, les interfaces des objets distants créés par le client et qui seront manipulées explicitement par le serveur doivent être dans le projet `commons`.

1.3 Erreurs fréquentes

1.3.1 Conflit entre les instances du serveur

Parfois vous faites tourner plusieurs instances du serveur par accident et vous tombez ainsi sur une erreur vous indiquant que le port de votre serveur est déjà occupé, et qu'il ne peut pas être lancé. Dans cette situation, avant de relancer votre programme, vérifier bien dans votre console qu'aucune instance du serveur n'est en train de tourner (i.e., elle est "terminated"). En gros, il faut arrêter toutes les instances du serveur avant de relancer une nouvelle instance.

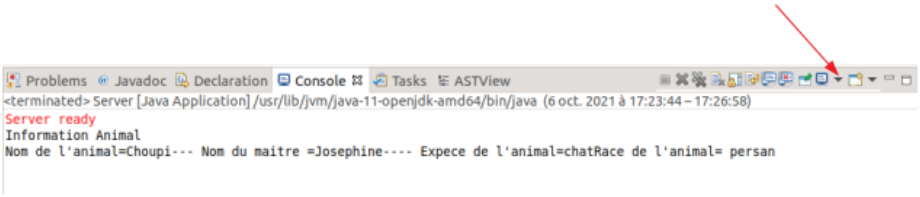

Figure 5: Consulter les programmes en cours d'exécution sous Eclipse

1.3.2 Lancer le programme d'un composant dans le mauvais ordre

Parfois vous essayez de lancer le programme de votre client alors que le programme de votre serveur ne l'est pas encore. Dans ce cas le client ne pourra pas ainsi se connecter au serveur. Rappelez-vous de l'ordre d'exécution :

1. lancer une instance du registre RMI :
   - soit en utilisant `rmiregistry` depuis la ligne de commandes pour un processus séparé,
   - soit en créant une instance du registre au sein du processus du serveur en utilisant `createRegistry(int port)`, et qui sera ainsi lancée quand vous lanceriez votre serveur.
2. lancer une instance du serveur.
3. lancer une instance du client.

Pour lancer plusieurs instances de votre client, il faut que le programme de votre client soit terminable manuellement via son interface fournie (e.g., via une option `quit` pour une CLI ou un bouton `fermer` pour une GUI).

1.3.3 Création d'un projet Java modulaire

Quand vous créez votre projet Java, le wizard de création vous demandera si vous voulez créer un fichier `module-info.java` qui permet de rendre votre projet modulaire. Si vous le faites, votre projet n'arrivera pas à localiser le module contenant les classes du package `java.rmi` pour qu'il puisse se lancer. Dans ce cas, vous pouvez préciser les modules nécessaires dont votre projet dépend selon la syntaxe de description des modules en Java (*au délà de la portée de ce TP*).

Je vous invite ainsi à éviter de créer un projet Java modulaire (cf. 4.png ci-joint), sauf si vous maîtrisez déjà tous ses facettes et que vous le souhaitez, mais ce n'est pas le but du TP.

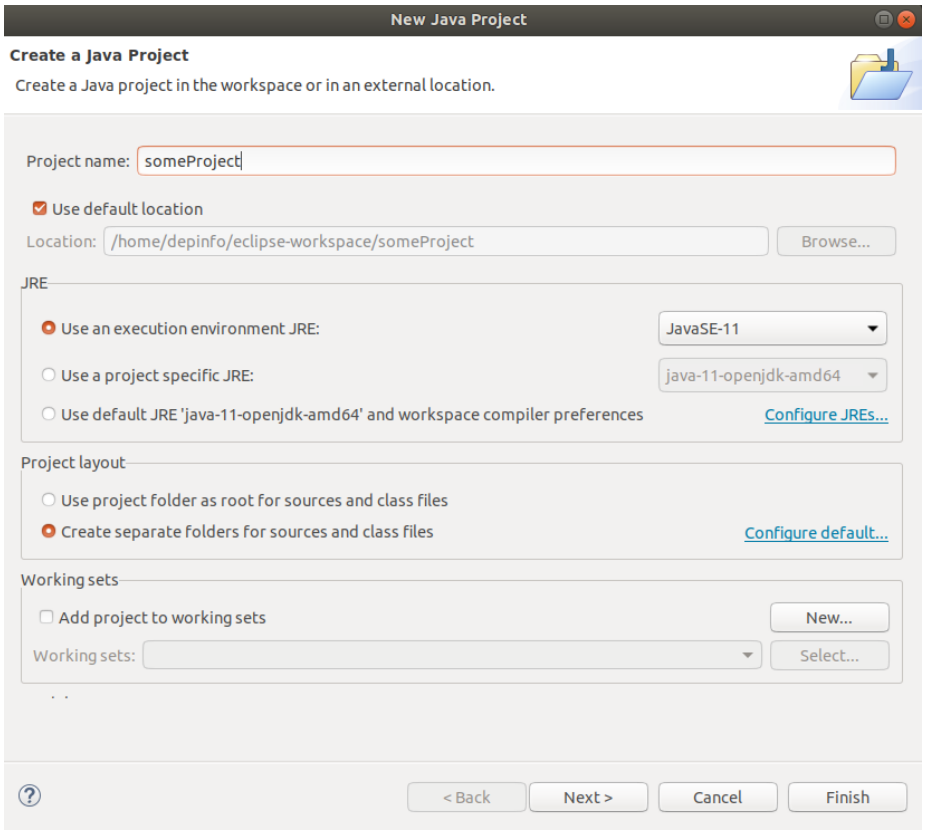

Figure 6: Le wizard pour la création de projets Java sous Eclipse

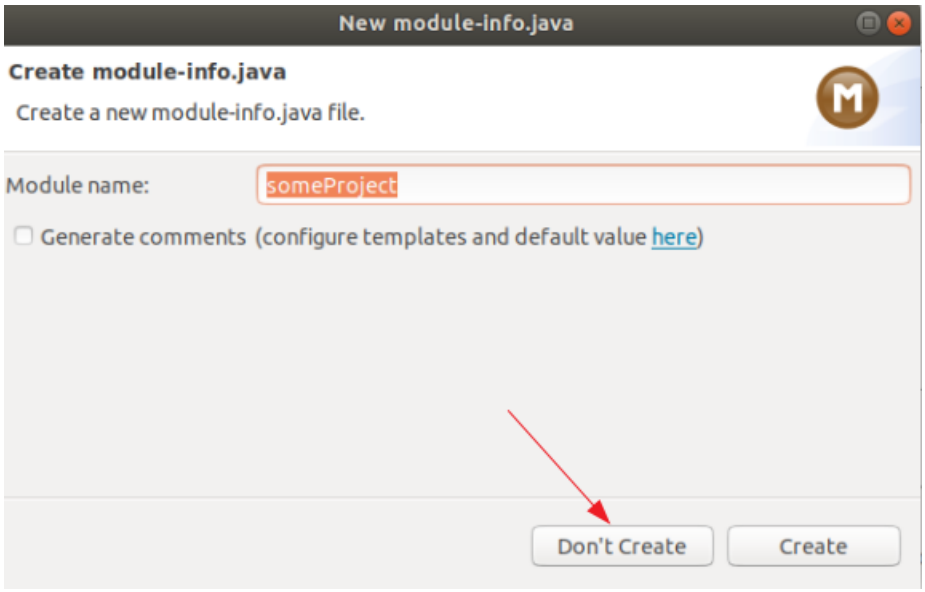

Figure 7: Créer un projet Java non modulaire sous Eclipse

1.3.4 Erreurs liées aux codebases

Quand vous spécifier l'ensemble des codebases consultables par des classes, vous pouvez éventuellement lancer des exceptions de types divers. Dans ce cas, vérifiez que :

1. le nom de la propriété pour lier des codebases est bien formé.
2. l'URI (URL ou chemin dans le système de fichiers) est bien formé.
3. votre dossier existe et ne désigne pas un package pour les classes du codebase.

1.4 Astuces

1.4.1 Rappels pour la POO en Java

1.4.1.1 Principe d'encapsulation

1. toute classe s'occupe de ses propres responsabilités, encapsule ses propriétés (attributs et méthodes), et impose des restrictions sur leur accès pour les autres class.

2. les attributs d'une classe `C` doivent toujours être `private` et accessibles uniquement à travers des **getters** (*lecture*) et éventuellement des **setters** (*écriture*).

3. on peut les déclarer `protected` si on souhaite donner un accès directe aux sous-classes de `C`.

1.4.1.2 Constructeurs par défaut et constructeurs paramétrés

1. un constructeur par défaut est une méthode spéciale d'une classe qui permet d'en créer un objet vide, éventuellement avec des initialisations par défaut pour ses attributs.
2. un constructeur paramétré est une méthode spéciale d'une classe qui permet d'en créer un objet et d'initialiser ses attributs par le biais de ses paramètres.
3. quand une classe possède plusieurs attributs, on peut déclarer plusieurs constructeurs paramétrés pour fournir plusieurs manières différentes de créer et initialiser des objets.

1.4.1.3 Nommer ses éléments de programmation

1. **conventions** :
   - `camelCase` pour les variables/attributs/méthodes : *`nomVariable`*, *`nomChamps`*, *`nomMethode([params])`*.
   - `CamelCase` pour les classes/interfaces/énumérations : *`NomClasse`*, *`NomInterface`*, *`NomEnum`*.
2. on essaye toujours de choisir des noms pertinents pour nos éléments de programmation :
   - List liste = new ArrayList<>(); → non parce que le nom ne spécifie pas ce que la liste doit contenir
   - List animaux = new ArrayList<>(); → ok parce qu'on peut savoir maintenant ce que cette liste contiendra.
   - List listeAnimaux = new ArrayList<>(); → ok, bien que la partie "liste" n'est pas forcément nécessaire parce qu'on peut le savoir directement depuis la déclaration de la variable. Toutefois, ça peut être utile si cette variable est utilisée dans un endroit dans le programme loin de sa ligne de déclaration.
   - List ListeAnimaux = new ArrayList<>(); → le nom choisi est ok mais la convention de nommage n'est pas respectée (`camelCase`).
3. parfois on est ramené à l'implémentation d'un concept en deux parties : une **interface** et une classe l'**implémentant**. Si cette interface est destinée à avoir une classe fournissant une implémentation par défaut (*e.g., implémentation du concept par un serveur ou une factory dédiée*), on adoptera l'un des styles de nommage suivants pour l'interface et la classe respectivement :
   - `Concept/ConceptImpl` : *e.g.,* *`Animal/AnimalImpl`*, *`DossierSuivi/DossierSuiviImpl`*.
   - `IConcept/Concept` : *e.g.,* *`IAnimal/Animal`*, *`IDossierSuivi/DossierSuivi`*.
   - `IConcept/ConceptImpl` : *e.g.,* *`IAnimal/AnimalImpl`*, *`IDossierSuivi/DossierSuiviImpl`*.

1.4.2 Divers

Il faut imaginer votre système d'une manière orientée objet. Pensez bien aux patrons de conception, aux principes de conception SOLID, et aux relations entre vos objets. Réifier quand c'est possible et nécessaire. Imaginer une interface utilisateur (CLI ou GUI) qui nous permettra de tester votre travail d'une manière ergonomique.

N'oubliez pas de commiter votre code sur un dépôt distant après avoir implémenté chaque feature. De cette manière vous diviserez vos tâches convenablement et vous laisserez des traces pour suivre l'évolution de votre performance.