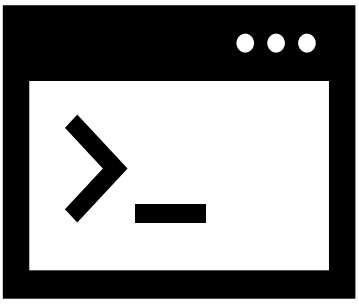
TP 1-Architectures Distribuées



- **Master Informatique**
- **Sujet :** -RMI
- **Membres** :

- 21908889: Adam DAIA **PARCOURS GL**

2023/2024

Table des matières :

I/ Introduction

Le projet que nous avons développé est une application de cabinet vétérinaire distribuée basée sur la technologie RMI (Remote Method Invocation). Le projet est structuré en utilisant un ensemble de packages, chacun jouant un rôle essentiel dans le fonctionnement global du système.

Le projet est découpé en quatre packages distincts, chacun remplissant une fonction particulière. Ces packages sont les suivants :

1. **Client** : Ce package contient l'implémentation du client, qui est responsable de la communication avec le serveur de cabinet vétérinaire distant. Le client utilise les interfaces distantes pour interagir avec le serveur.

2. **Serveur** : Le package Serveur contient l'implémentation du serveur de cabinet vétérinaire. Il comprend également des classes pour représenter différents aspects du cabinet, notamment les animaux, les espèces et les dossiers de suivi.

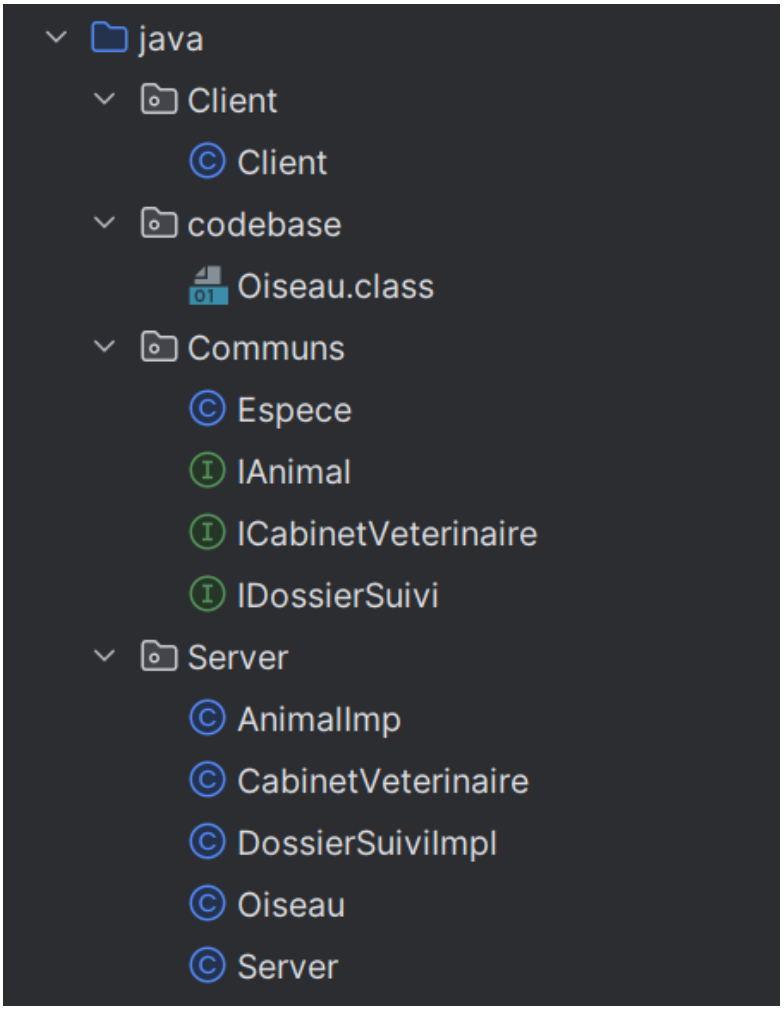
3. **Communs** : Dans ce package, nous avons placé les interfaces distantes partagées entre le client et le serveur. Les interfaces permettent de définir les méthodes et les objets distants utilisés pour accéder aux données et aux fonctionnalités du cabinet vétérinaire.

4. **Codebase** : Ce package est un élément clé de notre projet, où nous gérons les classes dynamiquement téléchargées. Il est utilisé pour illustrer le concept de "codebase" dans RMI. Vous y trouverez des fichiers de classe (.class) qui peuvent être téléchargés dynamiquement par le serveur pour ajouter de nouvelles espèces d'animaux, comme l'oiseau.

La structure du projet et l'organisation en packages visent à rendre le code plus lisible, maintenable et évolutif. De plus, la mise en place de la fonctionnalité de codebase permet au serveur d'ajouter de nouvelles espèces sans avoir à recompiler l'ensemble du projet.

II/ Architecture du Projet

Le projet se décompose en quatre packages distincts, chacun remplissant un rôle spécifique dans l'architecture globale. Cette structure bien organisée permet de concevoir et de développer un système distribué, où le serveur et le client interagissent de manière transparente. Voici un aperçu de la structure du projet et du contenu de chaque package :



Package Client :

- Le package Client abrite les composants de l'application client. Il comprend des classes responsables de l'interaction utilisateur et de la communication avec le serveur distant. Les fonctionnalités du client, telles que l'ajout et la recherche d'animaux, sont mises en œuvre dans ce package.

Package Communs :

- Le package Communs est destiné aux classes et aux interfaces partagées entre le client et le serveur. Il contient les définitions des interfaces distantes *IAnimal*, *ICabinetVeterinaire*, *IDossierSuivi* et une classe *Espece*. La communication entre le client et le serveur repose sur ces interfaces partagées.

Package Serveur :

- Le package Serveur est le cœur du système distant. Il contient les classes qui implémentent les interfaces distantes spécifiées dans le package Communs. Ces classes sont essentielles pour la gestion des animaux, des espèces et des dossiers de suivi au sein du cabinet vétérinaire. De plus, le serveur crée des objets distants et les met à la disposition des clients via le registre RMI.

Package Codebase :

- Le package Codebase est un ajout important à l'architecture du projet. Il est utilisé pour stocker les classes d'espèces telles que l'Oiseau Ces classes ne sont pas incluses initialement dans le projet du client ou du serveur. L'utilisation du mécanisme de codebase permet d'ajouter dynamiquement de nouvelles espèces sans nécessiter de modifications majeures dans le code du client ou du serveur.

Cette architecture modulaire et bien organisée favorise la scalabilité et la maintenance du système. Les clients peuvent ajouter de nouvelles espèces au cabinet vétérinaire distant en utilisant le codebase, simplifiant ainsi l'extension du système sans affecter son fonctionnement principal. La communication entre les parties cliente et serveur est transparente grâce aux interfaces distantes du package Communs.

III/ Explication des classes et interfaces

Package Communs :

Dans le package Communs, nous trouvons les éléments fondamentaux partagés entre le client et le serveur, facilitant ainsi la communication et l'interaction entre les deux. Ce package renferme plusieurs éléments clés :

1. **Interface IAnimal :**

L'interface *IAnimal* définit les méthodes que chaque animal dans le cabinet vétérinaire doit implémenter, tel *que getRace (), getNom_maitre (), getDossier (), getNom ()* et *getEspece ()*. Elle fournit des méthodes pour obtenir des informations sur l'animal, telles que son nom, son propriétaire, son espèce et son dossier de suivi. Grâce à cette interface, le client et le serveur peuvent interagir avec les animaux de manière uniforme.

2. **Interface ICabinetVeterinaire :**

L'interface *ICabinetVeterinaire* expose les opérations du cabinet vétérinaire, tel que :

ajouterAnimal(IAnimal animal),

ajouterAnimal(String nom, String nom_maitre, Especе especе, String race, IDossierSuivi suivi),

supprimerAnimal(IAnimal animal),

chercherAnimalParNom(String nom),

obtenirAnimaux()

Elle permet aux clients d'accéder aux fonctionnalités du cabinet vétérinaire, en utilisant des méthodes telles que *ajouterAnimal* et *chercherAnimal*.

3. **Interface IDossierSuivi :**

L'interface *IDossierSuivi* définit deux méthodes importantes *getDossierSuivi() et setDossierSuivi(String dossier)*. Ces méthodes permettent d'accéder aux informations du dossier de suivi de l'animal, ainsi que de les mettre à jour. Le dossier de suivi est essentiel pour suivre l'état de santé, les traitements et d'autres détails importants liés à l'animal.

4. **Classe Especе :**

La classe *Especе* implémente l'interface *Serializable*. Elle sert de base pour la création de différentes espèces d'animaux, telles que le *Félin* ou *l'Hominidé*. Elle stocke des informations sur le nom de l'espèce et sa durée de vie. L'utilisation de la sérialisation permet de transférer des objets d'espèces entre le client et le serveur.

Package Serveur :

Dans le package Serveur, nous trouvons les classes responsables de la mise en œuvre du serveur RMI et de la gestion des objets distants. Voici une brève explication des classes et interfaces de ce package

1. **Classe Serveur :**

La classe *Serveur* est le point d'entrée du serveur. Elle est chargée de la création du serveur RMI, de l'enregistrement des objets distants dans le registre RMI et de la mise à disposition de ces objets pour les clients. Cette classe joue un rôle essentiel dans l'initialisation et la gestion du serveur RMI, permettant au client d'accéder aux fonctionnalités offertes par le serveur.

2. **Classe CabinetVeterinaire :**

La classe *CabinetVeterinaire* est l'implémentation concrète de l'interface *ICabinetVeterinaire*. Elle gère la liste des animaux dans le cabinet, permettant leur ajout, leur suppression et leur recherche. Cette classe maintient également des informations importantes sur le cabinet, comme le nombre total d'animaux.

3. **Classe DossierSuiviImpl:**

La classe *DossierSuiviImpl* implémente l'interface *IDossierSuivi* et représente le dossier de suivi d'un animal. Cette classe est utilisée pour suivre l'état de santé et le suivi médical des animaux.

4. **Classe AnimalImp :**

La classe *AnimalImp* implémente l'interface *IAnimal* et représente un animal spécifique. Elle stocke des informations telles que le nom de l'animal, le nom du propriétaire, l'espèce à laquelle il appartient, la race de l'animal et son dossier de suivi. Les instances de cette classe sont créées et gérées par le serveur pour représenter les animaux du cabinet vétérinaire.

5. **Classe Oiseau :**

La classe *Oiseau* hérite de la classe *Espece* et représente une espèce spécifique. Cette classe illustre comment vous pouvez ajouter de nouvelles espèces au cabinet en étendant la classe *Espece*. Les instances de cette classe peuvent être ajoutées au cabinet vétérinaire pour gérer les oiseaux.

Package "Codebase" :

1. **Oiseau.class :**

Dans le package Codebase, vous trouverez le fichier compilé *Oiseau.class*. Ce fichier est utilisé pour illustrer le mécanisme de codebase, qui permet d'ajouter dynamiquement de nouvelles classes au système sans nécessiter de modifications majeures. Dans ce cas, le fichier *Oiseau.class* représente la classe *Oiseau* du package Serveur et il peut être accessible et utilisé par les clients pour gérer cette nouvelle espèce, sans nécessiter une connaissance préalable de son implémentation. Le mécanisme de codebase offre une flexibilité essentielle pour étendre le système avec de nouvelles classes, telles que des espèces, de manière transparente.

Package Client :

Dans le package Client, nous trouvons les classes responsables de l'interaction utilisateur et de la communication avec le serveur distant. Voici une brève explication de la classe principale de ce package :

1. **Classe Client**

La classe *Client* est le point d'entrée de l'application client. Elle permet aux utilisateurs d'interagir avec le cabinet vétérinaire distant via une interface utilisateur. Cette classe offre des fonctionnalités telles que l'ajout et la suppression d'animaux, la recherche d'animaux et la consultation des informations des animaux tel que leur nom, espèce, durée de vie ou bien leur dossier de suivi. La classe *Client* communique avec le serveur RMI pour réaliser ces opérations.

IV/ Fonctionnalités clés

L'application développée pour la gestion d'un cabinet vétérinaire offre un ensemble de fonctionnalités robustes conçues pour simplifier la gestion des patients.

Parmi ces fonctionnalités, l'ajout d'animaux se démarque en offrant une flexibilité totale, permettant aussi bien au client qu'au serveur de contribuer aux dossiers du cabinet. Les utilisateurs peuvent ajouter de nouvelles espèces au cabinet sans perturber le système existant, grâce à une architecture modulaire qui stocke les nouvelles classes d'espèces dans le package Codebase.

De plus, la possibilité de supprimer, rechercher et consulter les animaux facilite la gestion quotidienne des vétérinaires. Cette application polyvalente permet aux professionnels de la santé animale de gérer efficacement leur cabinet, tout en ayant la possibilité d'ajouter de nouvelles espèces au fur et à mesure que leur pratique se développe.

Dans cette section, nous allons explorer plus en détail ces fonctionnalités clés et expliquer comment elles sont mises en œuvre pour garantir une gestion fluide et évolutive du cabinet vétérinaire.

1. **Ajout d'Animaux :**

Notre application offre une flexibilité totale en ce qui concerne l'ajout d'animaux au cabinet vétérinaire. Cela peut être fait aussi bien par le client que par le serveur.

L'ajout par le serveur est réalisé en créant une instance d'animal avec les informations nécessaires, telles que le nom, le maître, l'espèce, la race et le dossier de suivi. Cette instance est ensuite ajoutée à la liste des animaux dans le cabinet.

L'ajout d'animaux par le client est similaire, permettant aux utilisateurs de créer des instances d'animaux directement via le client, en fournissant toutes les informations pertinentes.

2. **Suppression d'Animaux :**

Les utilisateurs ont la possibilité de retirer des animaux du cabinet à l'aide de la méthode *supprimerAnimal()*. Lorsqu'un animal est supprimé, il est retiré de la liste des animaux du cabinet, réduisant ainsi le nombre de patients. Cette fonctionnalité permet aux vétérinaires de gérer efficacement la liste des animaux en consultation.

3. **Recherche d'Animaux** :

La recherche d'animaux par nom est une fonctionnalité essentielle pour identifier et accéder rapidement à des animaux spécifiques dans le cabinet. En utilisant la méthode *chercherAnimalParNom (),* les utilisateurs peuvent trouver un animal en saisissant simplement son nom. Si l'animal est trouvé, ses informations sont renvoyées. Dans le cas contraire, un résultat nul est retourné, indiquant qu'aucun animal correspondant n'a été trouvé.

4. **Liste des Animaux** :

La possibilité d'obtenir une liste complète de tous les animaux dans le cabinet est essentielle pour une gestion efficace. La méthode *obtenirAnimaux ()* renvoie une copie de la liste des animaux, ce qui permet aux utilisateurs de consulter l'ensemble des patients du cabinet.

Chaque animal est accompagné de son nom, de son espèce, de sa durée de vie moyenne et de son dossier de suivi. Cela facilite le suivi et la gestion de l'ensemble des patients du cabinet.

5. **Ajout de Nouvelles Espèces** :

La fonctionnalité clé de votre application réside dans la capacité à ajouter de nouvelles espèces d'animaux sans nécessiter de modifications majeures dans le code du client ou du serveur. Cette flexibilité est réalisée grâce au package Codebase, qui stocke les classes des nouvelles espèces, telles qu'Oiseau.

L'utilisation du mécanisme de codebase permet d'accéder à ces nouvelles classes d'espèces au moment de l'exécution, facilitant ainsi l'ajout dynamique de nouvelles espèces au cabinet vétérinaire. Cela garantit que votre application est évolutive et peut accueillir de nouvelles espèces sans perturber le fonctionnement existant.

L'ensemble de ces fonctionnalités fait de notre application un outil puissant et polyvalent pour la gestion d'un cabinet vétérinaire. Elle offre des options flexibles pour ajouter, supprimer, rechercher et consulter les animaux du cabinet.

De plus, la capacité à ajouter de nouvelles espèces sans perturber le fonctionnement existant grâce au mécanisme de codebase renforce la polyvalence de l'application. Cette caractéristique est essentielle pour répondre aux besoins variés des professionnels vétérinaires, leur offrant un outil évolutif et adaptatif.

En résumé, notre application simplifie la gestion des patients et offre une extensibilité qui la rend prête à s'adapter aux évolutions futures du domaine vétérinaire.

V/ Exemples d'utilisation :

1. **Ajout d'Animaux au Cabinet Vétérinaire distant** :

Utilisation côté Client :

stubCabinetVeterinaireMTP.ajouterAnimal("Aigle", "Adam", oiseau, "Aigle_Royal", stubDossierAigle);

Dans cet exemple, le client utilise l'interface distante *ICabinetVeterinaire* pour ajouter un animal au cabinet vétérinaire distant. La méthode *ajouterAnimal()* est appelée avec plusieurs paramètres, ce qui permet de créer et d'ajouter un nouvel animal nommé "Aigle" dans le cabinet.

- `"Aigle"` : C'est le nom de l'animal que le client souhaite ajouter. Il peut s'agir de n'importe quel nom pour le nouvel animal.
- `"Adam"` : Il s'agit du nom du maître de l'animal, ce qui permet de connaître à qui appartient cet animal.
- `oiseau` : L'objet `oiseau` est une instance d'une Espèce . Il spécifie l'espèce de l'animal, ce qui est essentiel pour identifier et classifier l'animal correctement.
- `"Aigle_Royal"` : C'est la race de l'animal. Il indique une caractéristique spécifique de l'animal qui le distingue, par exemple, un "Aigle Royal" est une race d'aigle.
- `stubDossierAigle` : C'est un objet de dossier de suivi qui est lié à l'animal. Chaque animal a un dossier de suivi qui contient des informations sur sa santé, ses traitements, etc.

L'idée derrière cette ligne de code est que le client, en utilisant l'interface distante, est capable de créer un nouvel animal en fournissant toutes les informations nécessaires.

Le client peut interagir avec le système distant sans avoir à se soucier des détails de l'implémentation de la classe sous-jacente, ce qui simplifie considérablement le processus et offre une isolation entre les composants du client et du serveur. Cela garantit également que le client peut ajouter un nouvel animal en toute confiance, sans avoir besoin de connaître les complexités internes du cabinet vétérinaire distant.

Ensuite, en appelant *ajouterAnimal()*, le nouvel animal est ajouté au cabinet vétérinaire distant, où il peut être géré et suivi par le vétérinaire. De plus, après chaque ajout d'animal, la variable "nombrePatients" est incrémentée pour refléter le nombre actuel de patients enregistrés au cabinet.

Utilisation côté Serveur :

```
Espece félin =new Espece("félin", 20);

DossierSuiviImpl dossierTigre =new DossierSuiviImpl ("DossierDuTigre_Vierge");

IAnimal tigre= new AnimalImp("Tigre", "Adam", félin, "Race1", dossierTigre);

cabinetVeterinaireMTP.ajouterAnimal(tigre);
```

Dans cet exemple, le serveur est responsable de créer et d'ajouter un nouvel animal au cabinet vétérinaire distant. Pour ce faire, plusieurs étapes sont nécessaires.

-Tout d'abord, une instance d'une espèce est créée, en l'occurrence un "félin," avec ses caractéristiques spécifiques telles que le nom de l'espèce ("félin") et une durée de vie moyenne (20 ans). De même, un dossier de suivi, nommé "DossierDuTigre_Vierge," est initialisé pour le futur animal.

-L'étape clé réside dans la création de l'animal. Un nouvel objet animal, ici un "Tigre," est instancié, comportant divers détails tels que le nom de l'animal, le nom de son maître, l'espèce (dans ce cas félin), la race et le dossier de suivi. Chacun de ces éléments est fourni en tant que paramètre lors de la création de l'animal.

-Une fois que l'animal est créé, il est prêt à être intégré au cabinet vétérinaire distant. La méthode *ajouterAnimal(tigre*) est invoquée sur l'instance du cabinet vétérinaire, ce qui a pour effet d'inclure le nouveau tigre dans la liste des animaux pris en charge par le cabinet. De plus, après chaque ajout d'animal, la variable "nombrePatients" est incrémentée pour refléter le nombre actuel de patients enregistrés au cabinet.

Cette illustration met en évidence la capacité du serveur à ajouter des animaux au cabinet vétérinaire en créant manuellement une instance d'animal et en l'intégrant dans le cabinet. Il souligne l'importance de préparer à l'avance des espèces et des dossiers de suivi pour faciliter l'ajout ultérieur d'animaux sans entrer dans les détails de ces préparatifs. Elle souligne également l'importance de suivre le nombre de patients, ce qui peut être crucial pour la gestion des capacités du cabinet vétérinaire distant.

Cette approche offre une grande flexibilité et un contrôle total sur la gestion des patients du cabinet vétérinaire distant.

2. **Suppression d'Animaux**

```
IAnimal stubGorille = (IAnimal) registry.lookup("Gorille");

stubCabinetVeterinaireMTP.supprimerAnimal(stubGorille);
IAnimal rechecheGorille2 = stubCabinetVeterinaireMTP.chercherAnimalParNom("Gorille");
System.out.println("Gorille a été supprimé : "+ rechecheGorille2);
```

Dans cette séquence, "stubGorille" est une référence distante à l'animal "Gorille" que l'on souhaite supprimer du cabinet. La méthode *supprimerAnimal(stubGorille)* est appelée sur l'instance du cabinet vétérinaire pour effectuer la suppression.

Après la suppression, une recherche est effectuée pour vérifier si l'animal "Gorille" a été réellement retiré du cabinet. La méthode "*chercherAnimalParNom("Gorille")*" est utilisée pour rechercher un animal au nom de "Gorille." Comme "Gorille" a été supprimé, la recherche renverra "null," indiquant que l'animal n'est plus présent dans le cabinet.

Après la suppression réussie, la variable "nombrePatients" est décrémentée, maintenant le nombre de patients à jour. Cette opération garantit que la liste de patients du cabinet reste précise et reflète toujours le nombre d'animaux présents.

La suppression d'animaux est une opération essentielle pour maintenir la base de données du cabinet en ordre et à jour, tout en garantissant que les ressources du cabinet sont correctement gérées.

3. Recherche d'Animaux

```
IAnimal perroquet = stubCabinetVeterinaireMTP.chercherAnimalParNom("Paco");

if (perroquet != null) {
   System.out.println("Le perroquet a été ajouté au cabinet avec succès!");
} else {
   System.out.println("Le perroquet n'a pas été trouvé dans le cabinet.");
}
```

L'une des fonctionnalités clés de cette application est la possibilité de rechercher un animal spécifique par son nom dans le cabinet vétérinaire. Dans l'exemple fourni, le client cherche un perroquet nommé "Paco" pour vérifier s'il a été enregistré dans le cabinet vétérinaire.

Pour cela on utilise la méthode distante *chercherAnimalParNom()* via l'interface distante *stubCabinetVeterinaireMTP*. Cette méthode parcourt la liste des animaux actuellement enregistrés dans le cabinet vétérinaire, et compare le nom de chaque animal avec le nom fourni (ici "Paco").

L'implémentation de la méthode *chercherAnimalParNom*() utilise une boucle pour parcourir chaque animal dans la liste. Si elle trouve un animal avec le nom correspondant, elle renvoie cet animal. Si aucun animal ne correspond, la méthode renvoie `null`.

De retour dans le code client, le résultat de la recherche l'objet `perroquet` est vérifié. Si `perroquet` n'est pas `null`, cela signifie que le perroquet a été trouvé et un message approprié est affiché, à savoir *"Le perroquet a été ajouté au cabinet avec succès* !". Si le résultat est `null`, cela signifie que le perroquet n'est pas présent dans le cabinet, et le message *"Le perroquet n'a pas été trouvé dans le cabinet."* est affiché à la place.

4. Liste des Animaux

```
List<IAnimal> AnimauxCabinetMTP= stubCabinetVeterinaireMTP.obtenirAnimaux();


 for (IAnimal animal : AnimauxCabinetMTP){
    System.out.println("\nNom de l'animal : "+ animal.getNom());
    System.out.println("Espèce : "+ animal.getEspece().getNom());
    System.out.println("Durée de vie moyen  : "+ animal.getEspece().getDuréeVie());
    System.out.println("Dossier de suivi : "+ animal.getDossier().getDossierSuivi());
 }
System.out.println("\nFin du listing des animaux dans le CabinetVeterinaireMTP" );
```

La fonction de listage des animaux dans le cabinet vétérinaire est une fonctionnalité clé qui permet aux utilisateurs de visualiser la liste complète des animaux actuellement enregistrés dans le cabinet. Cela peut être utile pour le personnel vétérinaire ou pour les propriétaires d'animaux pour vérifier rapidement les patients actuels et leurs informations associées.

L'exemple de code fourni permet d'illustrer comment cette fonctionnalité est réalisée. En utilisant l'interface distante `stubCabinetVeterinaireMTP`, le client appelle la méthode *obtenirAnimaux()* . Cette méthode est implémentée du côté serveur et renvoie une copie de la liste des animaux actuellement enregistrés dans le cabinet.

Ensuite, le client parcourt la liste des animaux à l'aide d'une boucle `for` pour chaque animal de la liste. Puis on affiche sur la console les diverses informations sur chaque animal, telles que le nom de l'animal, l'espèce à laquelle il appartient, la durée de vie moyenne de son espèce, et le contenu de son dossier de suivi.

Cette fonctionnalité offre une visibilité complète sur les patients du cabinet vétérinaire, facilitant la gestion et la surveillance des animaux. Elle permet aux utilisateurs de récupérer des informations importantes sur les patients en un seul coup d'œil, améliorant ainsi l'efficacité et la qualité des soins prodigués aux animaux. De plus, le fait de renvoyer une copie de la liste garantit que les données existantes ne sont pas altérées lors de la consultation.

5. Ajout de Nouvelles Espèces via le codebase :

Une fonctionnalité cruciale de l'application est la possibilité d'ajouter de nouvelles espèces d'animaux via le codebase. Cette caractéristique offre une flexibilité essentielle, car elle permet de mettre à jour et d'élargir la liste des espèces prises en charge sans avoir à modifier le code source de l'application elle-même. Voici comment cela fonctionne avec l'exemple de la classe `Oiseau` :

```
public class Oiseau extends Espece implements Serializable {
   public Oiseau(String nom, int dureeVie) {
      super(nom, dureeVie);
   }
}
```

- **Création de nouvelles espèces** : Pour ajouter une nouvelle espèce, il faut créer une nouvelle classe qui représente cette espèce. Dans notre exemple, la classe `Oiseau` est créé, et elle hérite de la classe abstraite `Espece`. Cette nouvelle classe doit implémenter les attributs et les méthodes nécessaires pour définir cette espèce, tels que le nom de l'espèce et la durée de vie moyenne.

- **Sérialisation** : Une fois la classe de la nouvelle espèce créée, elle doit être compilée en bytecode et placée dans le dossier spécifié par le `codebase`. Cette classe doit également être sérialisable, car elle sera transmise via RMI. Cela signifie qu'elle doit implémenter l'interface `Serializable`.

- **Inscription auprès du registre** : Pour que le cabinet vétérinaire puisse reconnaître et gérer la nouvelle espèce, la classe `Oiseau` doit être inscrite auprès du registre RMI. Cela se fait généralement dans la méthode `main` du serveur. Une fois que la classe `Oiseau` est enregistrée, elle est accessible à distance par les clients.

- **Utilisation dans l'application** : Une fois que la classe `Oiseau` est inscrite, les clients de l'application peuvent l'utiliser comme n'importe quelle autre espèce existante. Par exemple, lors de la création d'un nouvel animal, le client peut spécifier la nouvelle espèce en créant une instance de `Oiseau`. L'ensemble des espèces disponibles est accessible via le `codebase`.

Cette fonctionnalité offre une grande extensibilité à l'application, car de nouvelles espèces peuvent être ajoutées dynamiquement sans avoir à modifier le code source de l'application elle-même. Cela facilite la gestion de différentes espèces d'animaux au sein du cabinet vétérinaire, tout en conservant la transparence pour les clients qui peuvent ajouter des animaux de ces nouvelles espèces sans effort supplémentaire.

6. **Gestion des Alertes en Fonction du Nombre d'Animaux :**

La fonction *verifierSeuilsAlerte()* est une fonctionnalité clé de l'application qui permet de surveiller le nombre d'animaux dans le cabinet vétérinaire distant. Son rôle est de détecter lorsque le nombre d'animaux dépasse certains seuils prédéfinis, déclenchant ainsi des alertes. Voici comment cette fonctionnalité fonctionne :

- **Surveillance du Nombre d'Animaux** : La méthode *verifierSeuilsAlerte()* est appelée chaque fois qu'une opération d'ajout ou de suppression d'animal a lieu. Elle vérifie le nombre actuel d'animaux dans le cabinet vétérinaire via la variable `nombrePatients`.

- **Détection des Seuils** : La méthode compare `nombrePatients` à plusieurs seuils prédéfinis (ici 3, 4 et 5 animaux). Si le nombre d'animaux dépasse l'un de ces seuils, une alerte est déclenchée.

- **Alertes** : Si le nombre d'animaux atteint ou dépasse l'un des seuils, la méthode *verifierSeuilsAlerte()* génère un message d'alerte approprié en fonction du seuil dépassé. Par exemple, si le nombre d'animaux atteint 3, le message "Alerte : Le nombre de patients a dépassé 3 !" est imprimé.

Cette fonctionnalité est essentielle pour la gestion du cabinet vétérinaire, car elle permet au personnel de surveiller et de contrôler son nombre de patients. Lorsqu'un seuil critique est atteint, des mesures peuvent être prises pour gérer efficacement la situation, telles que l'affectation de plus de ressources ou l'ajout de personnel. Cela garantit que le cabinet fonctionne de manière optimale et que les patients reçoivent les soins nécessaires sans délai.

VI. Conclusion

Ce projet de gestion de cabinet vétérinaire montre une solution complète et flexible pour répondre aux besoins des vétérinaires. En mettant en œuvre des fonctionnalités telles que l'ajout, la suppression, la recherche et la consultation des animaux, ainsi que la possibilité d'introduire de nouvelles espèces sans perturber le fonctionnement existant, l'application offre un outil puissant pour la gestion efficace des patients.

L'utilisation de la technologie permet une communication transparente entre le client et le serveur, garantissant que les utilisateurs puissent interagir avec le cabinet vétérinaire à distance. De plus, le mécanisme de codebase offre une flexibilité sans précédent en matière d'extension de nouvelles espèces, assurant que l'application reste évolutive pour s'adapter aux futurs besoins du domaine vétérinaire.