Enoncés des exercices ...

TPs n°1 : les threads

Ce TP aborde une généralisation du problème des philosophes, dont la version initiale est due à E. Dijkstra, un mathématicien hollandais. L'objectif de ce problème était de tester le comportement de certains algorithmes d'allocation de ressources

Un ensemble de NBP philosophes (NBP vaut 5 typiquement) sont réunis autour d'une table dans un restaurant chinois. Chaque philosophe réalise une boucle comportant 2 sous-tâches :

```
loop :
    <discuter> ;
    <manger>
endloop
```

Question n°1 (le modèle d'avant Java5)
Identifier les tâches ainsi que les ressource d'exécution en supposant qu'un philosophe ne sait pas discuter et manger en parallèle (il n'est le siège que d'une seule activité : chaque itération dans la boucle du comportement consiste en 2 phases séquentielles).

Coder le problème en (1) associant à chaque philosophe un numéro (indiquant sa position autour de la table) et un nom. Par ailleurs <manger> et <discuter> se manifestent par l'affichage d'un message simple suivi d'un blocage d'une durée aléatoire (à l'aide de la méthode Thread.sleep((long)(1000 * Math.random())) par exemple).

Question n°2 (le modèle d'avant Java5)
Reprendre la modélisation et le codage en supposant qu'un philosophe sait discuter et manger en parallèle. Le comportement consiste en l'exécution parallèle des 2 boucles suivantes :

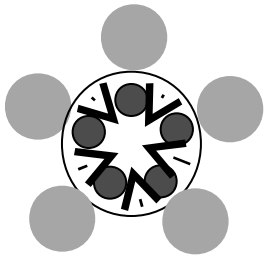
```
loop :
    <reflechir> ;
    <discuter>
endloop
loop :
    <manger>
    <nePasManger>
endloop
```

Chaque sous-tâche <manger>, <nePasManger>, <réfléchir>, <discuter> se manifeste par l'affichage d'un message simple et d'un blocage d'une durée aléatoire.

Question n°3 (le modèle d'après Java5)
On prend en compte maintenant le fait que chaque assiette doit être remplie lorsqu'elle est vide. On suppose ici que les assiettes sont initialement vides puis sont remplies toutes les NBI itérations (NBI vaut typiquement 5). Le remplissage est assuré par un ensemble de NBS serveurs implémenté par un pool de threads (NBS vaut typiquement 3). Modéliser et coder le problème.

Quelques idées simples : le remplissage des assiettes peut être réalisé via une tâche RemplirAssiette (une implémentation de Callable) soumise à un ExecutorService réalisé via un FixedThreadPool (figurant l'ensemble des serveurs). Le remplissage se manifeste par l'affichage d'un message simple suivi d'un blocage

Question n°4 (modélisation d'un problème de synchronisation)
On prend en compte maintenant le fait qu'un philosophe doit pour pouvoir manger, disposer de 2 baguettes. Or le nombre de baguettes disponibles peut être inférieur à 2 * NBP, ce qui implique donc qu'une éventuelle compétition entre les philosophes pour leur acquisition. Plus précisément l'attribution des baguettes aux philosophes est faite via son constructeur. Chaque philosophe doit **acquérir** ses baguettes avant de manger. Il doit par ailleurs les **libérer** lorsqu'il a fini la phase <manger> pour permettre à d'autres de manger à leur tour. Donner une modélisation simple du problème. Faire apparaître pour cela les points de synchronisation, les variables d'état...



Le problème consiste à synchroniser correctement le comportement des philosophes. La solution envisagée consiste à introduire un arbitre-gestionnaire de baguettes, avec lequel les philosophes communiquent.

Quelques idées simples :
- on peut associer un numéro à chaque baguette (numéro communiqué aux philosophes via leur constructeur)
- on peut représenter l'état du système (du point de vue de la gestion des baguettes et de la synchronisation associée) par un tableau de booléens : `boolean[] estUtilisee`

Question n°5a (programmation avec attente active des philosophes synchrones)
Coder le problème en réalisant la synchronisation par une attente active des philosophes à l'aide d'un moniteur.

Quelques idées simples :
- on peut utiliser l'API simple suivante pour l'arbitre-gestionnaire de baguette :

```
public interface IGestBaguettes {
   public boolean acqBaguettes(int bag1, int bag2) throws IllegalArgumentException;
   public void libBaguettes(int bag1, int bag2) throws IllegalArgumentException;
}
```

Question n°5b (programmation avec attente passive des philosophes synchrones)
Coder le problème en réalisant la synchronisation par une attente passive des philosophes à l'aide moniteur.

Question n°6 (optimisation de la programmation avec attente passive des philosophes synchrones)
On se propose d'optimiser le comportement en distinguant les files de blocage des threads des philosophes. Proposer des solutions n'introduisant pas d'interblocage.

Quelques idées simples :
- on peut utiliser remplacer le tableau `boolean[] estUtilisee` par un tableau `Baguette[] baguettes`, où `Baguette` une inner classe (équivalente à un `Boolean`) décrite ci dessous. Il est alors possible de bloquer sélectivement un philosophe sur `baguettes[i]` par un wait.

```
class Baguette  {
    boolean estUtilisee;
}
```

Question n°7 (programmation de philosophes asynchrones)

On se propose d'introduire des philosophes "asynchrones" qui possèdent 2 activités : une qui assure la tâche (supposée permanente) de réflexion, l'autre la tâche d'alimentation. Le comportement de cette seconde activité est le suivant :

```
loop :
    <délai avant demande de notification> ;
    <demander à être notifié de la disponibilité des baguettes et se bloquer>
endloop

sur réception d'une notification de disponibilité des baguettes
    <manger> puis <libérer baguettes>
```

On introduit pour cela 2 nouvelles interfaces PhiloCallback et IGestBaguettes1.

```
public interface PhiloCallback {
    public void baguettesLibres() throws RemoteException;          //indique la disponibilté des baguettes
}

public interface IGestBaguettes1 extends IGestBaguettes {
    public void resBaguettes(int b1, int b2, PhiloCallback itf) throws RemoteException;
}
```

Quelques idées simples :

- on peut stocker les réservations de baguettes (i.e. les interfaces de PhiloCallback) dans une List consultée à chaque libération de baguettes en utilisant une inner classe ReqBaguettesInfo :

```
private class ReqBaguettesInfo {
   int bag1;
   int bag2;
   IPhiloCallback philo;
}
```

TP n°2 : la communication par socket en mode stream

On se propose d'étendre la solution implémentée dans le TPs précédent à une infrastructure répartie : l'arbitre-gestionnaire de baguettes ne s'exécute pas dans la même JVM que les philosophes (les serveurs chargés de remplir les assiettes restent dans la même JVM que les philosophes). On considère dans un premier temps la solution reposant sur l'attente active avec des philosophes "synchrones". La nouvelle implémentation vise à réaliser la communication entre les philosophes et l'arbitre via des sockets en mode stream.

Question n°1 (modélisation)
Définir le protocole de communication entre les philosophes et le gestionnaire de baguettes : définir la nature, le contenu, le format des "messages" échangés, sachant que ces derniers seront constitués d'objets des classes RStRequest et StResponse échangés par sérialisation. Ces classes sont fournies dans le package philosophe.stream.

```
class StRequest implements Serializable {
   static final long serialVersionUID = -4321405821600571046L;

   private String req;
   private int bag1;
   private int bag2;

   public StRequest(String r,int b1, int b2) { req=r; bag1 = b1; bag2 = b2; }
   public String getRequete() { return req; }
   public int getBaguette1() { return bag1; }
   public int getBaguette2() { return bag2; }
}

class StResponse  implements Serializable {
   static final long serialVersionUID = -8296046979693023265L;

   private Object reponse;      //la réponse correspond soit à un Boolean soit à une IllegalArgumentException

   public StResponse(Object rp) { reponse=rp; }
   public Object getReponse() { return reponse; }
}
```

Question n°2a (programmation)
Coder l'aplication du coté philosophes (sachant que les String correspondant aux requêtes sont respectivement "acqBaguettes" et "libBaguettes").

Question n°2b (programmation)
Coder l'aplication du coté arbitre-gestionnaire de baguettes.

Question n°3 (architecture)
Reprendre le codage en cherchant à séparer le code applicatif du code technique de communication. Introduire pour cela des stubs et skeletons.

Question n°5
Etendre la solution précédente pour traiter le cas des philosophes "asynchrones".

Quelques idées simples :

- Une façon de procéder consiste à créer dynamiquement un skeleton de callback ainsi que le stub permettant d'y accéder (il implémente IPhiloCallback) et de l'envoyer comme argument de reqBaguettes (il implémente aussi Serializable)

TP alternatif

On considère une application simple formée d'une machine qui possède une température interne variable au cours du temps. Cette machine implémente l'interface.

```
public interface IMachine {
        public double getTemperature();
}
```

Un client affiche la température de la machine dans une boucle. Le client et la machine ne s'exécute pas dans la même JVM. La nouvelle implémentation vise à réaliser la communication entre le client et la machine via des sockets en mode stream.

Question n°1 (modélisation du problème)
Définir le protocole de communication entre le client et la machine : définir la nature, le contenu, le format des "messages" échangés, sachant que ces derniers seront constitués d'objets des classes RStRequest et StResponse échangés par sérialisation. Ces classes sont fournies dans le package capteurs.stream.

```
class StRequest implements Serializable {
  static final long serialVersionUID = -4321405821600571046L;
}

class StResponse  implements Serializable {
  static final long serialVersionUID = -8296046979693023265L;

  private Object reponse;      //la réponse correspond à un Double

  public StResponse(Object rp) { reponse=rp; }
  public Object getReponse() { return reponse; }
}
```

Question n°2a (programmation)
Coder l'aplication du coté client.

Question n°2b (programmation)
Coder l'aplication du coté machine.

Question n°3 (architecture)
Reprendre le codage en cherchant à séparer le code applicatif du code technique de communication. Introduire pour cela des stubs et skeletons.

TP n°4 : la réflexion et les annotations

L'objectif consiste à remplacer le skeleton spécifique du Tp précédent par un skeleton générique (i.e. un skeleton utilisable pour n'importe quel type d'objet de service). On remplace pour cela les classes utilisées dans le Tp précédent par : GenericRequest et GenericReponse du package reflect.

```
public class GenericRequest implements Serializable {
    private static final long serialVersionUID = 6658340337605443 60L;

    private String methodName;
    private Class[] paramTypes;
    private Class resultType;
    private Object[] values;
    private long ident;

    public GenericRequest(String m, Class[] p, Object[] v, Class r, long i) {
        methodName = m;
        paramTypes = p;
        values = v;
        resultType = r;
        ident = i;
    }

    public String getMethodName() { return methodName; }
    public Class[] getParameterTypes() { return paramTypes; }
    public Class getResultType() { return resultType; }
    public Object[] getValues() { return values; }
    public long getIdent() { return ident; }
}

public class GenericResponse implements Serializable {
    private static final long serialVersionUID = 2077113375563019860L;

    private Object result;
    private long ident;
    private boolean isException;

    public GenericResponse(Object r, long i,  boolean iE) {
        result = r;
        ident = i;
        isException = iE;
    }

    public GenericResponse(Object r, long i) { this(r, i); }

    public Object getResult() { return result; }
    public long getIdent() { return ident; }
    public boolean isException() { return isException; }
}
```

On encapsule par ailleurs les exceptions techniques dans un wrapper qui étend RuntimeException : JAPRemoteException

```
public class JAPRemoteException extends RuntimeException {
    private static final long serialVersionUID = 506453249625055305L;
    private Throwable wrapped;
```

```
    public JAPRemoteException(Throwable ex) { wrapped = ex; }
    public JAPRemoteException(String msg) { super(msg); }

    public String getMessage() {
        if (wrapped != null) return wrapped.getMessage();
        else return super.getMessage();
    }

    public String toString() {
        if (wrapped != null) return wrapped.toString();
        else return super.toString();
    }
}
```

Question n°1 (programmation du skeleton)
Ecrire le skeleton générique : GenericSkel utilisant la réflexion. L'objet de service à invoquer est un argument du constructeur.

Une idée simple : reprendre le skeleton développé dans le Tp sur les sockets en mode stream en modifiant uniquement la méthode run du thread chargée d'analyser les requêtes entrantes et de construire les réponses.

Tester le code en utilisant le stub RfStub du package reflect

Question n°2 (programmation du stub "serveur")
On introduit une annotation d'interface : @JAPRemote. Cette annotation équivalente à une interface de tag autorise l'accès à ses méthodes par une invocation distante. On modifie donc le stub de façon à ce qu'il vérifie que la méthode à invoquer est déclarée dans une interface annotée par @JAPRemote. Dans le cas contraire il répond par l'envoi d'une exception JAPRemoteException.

Tester le code en annotant une méthode de l'arbitre.

TP alternatif

On se propose de modifier le programme de test présenté dans le chapitre sur la réflexion (MyJunit) en remplaçant la convention sur le nommage des méthodes par une annotation @Test.

Question n°1 (modélisation du problème)
Créer l'annotation.
Modifier le programme MyJunit.

TP n°5 : la communication via RMI

L'objectif consiste à remplacer le protocole de communication par sockets entre les philosophes et l'arbitre développé dans les Tps précédents par une communication à l'aide de RMI. On se propose d'éviter les problèmes spécifiques posées par le chargement dynamique des stubs en disposant des classes de stubs client en local.

On se propose par ailleurs de ne pas "polluer" le code applicatif déjà développé par des spécificités RMI et d'introduire par conséquent des adapteurs.

On ne considère pas dans un premeier temps les philosophes "asynchrones"

Question n°1 (modélisation du problème)
Modéliser le problème en précisant les interfaces utilisées.

Question n°2 (programmation avec attente active des philosophes)
Coder les adapteurs du coté client et serveur de façon à pouvoir utiliser le code déjà développé via RMI

Question n°3 (programmation avec callbacks)
Etendre la solution aux philosophes "asynchrones".

TP alternatif

On se propose d'implémenter l'application client-machine du TP alternatif via RMI. Définir l'interface de service IRMIMachine correspondant à IMachine

Question n°1
Coder l'application.

TPs n°6 : la programmation via JMX

L'objectif consiste à introduire des fonctions d'administration dans l'application des philosophes communiquant avec le Gestionnaire de baguettes via RMI. On désire :

- observer/modifier certains paramètres temporels utilisés par les philosophes (durée pendant laquelle il mange, il digère et surtout délai entre 2 requêtes successives en cas de refus d'accorder les baguettes)
- observer la durée de communication entre un philosophe et le gestionnaire
- observer le nombre de refus d'accorder les baguettes.
- pouvoir suspendre/reprendre les interactions entre les philosophes et l'arbitre gestionnaire de baguettes.

Question n°1 (modélisation du problème)
Déterminer le niveau d'administrabilité requis pour les philosophes et identifier les modifications nécessaires. Déterminer les fonctions que l'on peut implémenter sans intervention dans le code applicatif (i.e. les philosophes et le gestionnaire de baguettes).

Question n°2 (conception de l'application)
Identifier les MBeans, les listeners et les intercepteurs à introduire.

Question n°3
Programmer l'administration de l'application.

TP alternatif

L'objectif consiste à introduire des fonctions d'administration dans l'application utilisant la machine à température interne.

On désire :

- être notifié lorsque la température sort d'un intervalle (temperatureMin, temperatureMax) paramétrable à partir de la couche d'administration.
- faire des statistiques sur les températures retournées par la méthode getTemperature()

Question n°1 (modélisation du problème)
Déterminer le niveau d'administrabilité des ressources.

Question n°2 (conception de l'application)
Décrire l'architecture. Identifier pour cela les MBeans, les listeners et les intercepteurs à introduire.

Question n°3
Programmer l'administration de l'application.

TPs n°7 : la communication via JMS

L'objectif consiste à implémenter la communication entre les philosophes et l'arbitre-gestionnaire de baguettes à l'aide de JMS. On ignore dans un premier temps les philosophes "asynchrones".

Question n°1 (modélisation du problème)
Modéliser le problème en précisant les messages et les files utilisés.

Quelques idées simples :
- solution1 :
  - associer au gestionnaire une destination et un consommateur (réception des requêtes)
  - associer à chaque philosophe une destination et un producteur (envoi des requêtes)
  - créer pour chaque philosophe une destination temporaire et un consommateur (réception des réponses). Le gestionnaire crée de façon dynamique des producteurs pour l'envoi des réponses.
  - des MapMessages dont les clés sont "meth", "bag1", "bag2".

- solution2 :
  - associer au gestionnaire une destination et un consommateur (réception des requêtes)
  - associer au gestionnaire une destination et un producteur (envoi des réponses)
  - associer à chaque philosophe une destination et un producteur (envoi des requêtes)
  - créer une destination unique pour la réception des réponses et associer à chaque philosophe un consommateur associé à un filtre utilisant une propriété : (philosophe = position)
  - des MapMessages dont les clés sont "meth", "bag1", "bag2".

Question n°2 (programmation avec attente active des philosophes)
Coder les philosophes et le gestionnaire de baguettes à l'aide de l'API JMS.
On se propose d'utiliser la plateforme JORAM.

TP alternatif

L'objectif consiste à implémenter la communication entre le clients et la machine JMS.

Question n°1 (modélisation du problème)
Modéliser le problème et coder l'application.

Quelques idées simples :
- solution1 :
  - associer au compteur une destination et un consommateur (réception des requêtes)
  - associer à chaque client une destination et un producteur (envoi des requêtes)
  - créer pour chaque client une destination temporaire et un consommateur (réception des réponses). Le compteur crée de façon dynamique des producteurs pour l'envoi des réponses.
  - des MapMessages dont les clés sont "meth", "value".

- solution2 :
  - associer au compteur une destination et un consommateur (réception des requêtes)

- associer au compteur une destination et un producteur (envoi des réponses)
- associer à chaque client une destination et un producteur (envoi des requêtes)
- créer une destination unique pour la réception des réponses et associer à chaque client un consommateur associé à un filtre utilisant une propriété : (client = numero)
- des MapMessages dont les clés sont "meth", "value".

TP n°8 : les chargeurs de classes

Question n°1 (modélisation du problème)
Ecrire un ClassLoader qui :

- détecte qu'une interface <interf> est annotée par @JAPRemote (définie dans le package reflect)
- génère le stub correspondant au skeleton générique défini dans le Tp précédent, l'enregistre sous le nom <interf>_Stub

Quelques idées simples :

- on peut utiliser javassist pour générer la classe de stub.
- on peut utiliser la réflexion java pour analyser l'interface <interf>

TP n°9 : la communication par socket en mode datagram

On se propose d'étendre à un ensemble de philosophes répartis la solution implémentée dans les TPs précédents. On considère les solutions reposant sur les clients actifs avec l'arbitre-gestionnaire de baguettes. La nouvelle implémentation vise à réaliser la communication entre les philosophes et l'a rbitre via des sockets en mode datagram.

Question n°1 (modélisation)
Définir le protocole de communication entre les philosophes et le gestionnaire de baguettes : définir la nature, le contenu, le format des messages échangés.

Question n°2 (programmation)
Coder les philosophes et le gestionnaire.

Question n°3 (architecture)
Reprendre le codage en cherchant à séparer le code applicatif du code technique de communication. Introduire pour cela des stubs "client" et "serveur".