

Java Remote Method Invocation

Java-RMI

- ▶ RMI signifie Remote Method Invocation
- ▶ Introduit dès JDK 1.1
- ▶ Partie intégrante du cœur de Java
- ▶ RMI = RPC en Java + chargement dynamique de code
- ▶ Même notions de stubs et skeletons
- ▶ Fonctionne avec l'API de sérialization (utilisée également pour la persistance)
- ▶ Possibilité de faire interagir RMI avec CORBA et DCOM

Notions de base

- ▶ RMI impose une distinction entre
 - ▶ Méthodes locales
 - ▶ Méthodes accessibles à travers le réseau
- ▶ Distinction dans
 - ▶ Déclaration
 - ▶ Usage (mais léger)
- ▶ Le stub est compatible avec l'appelé
 - ▶ Il a la “même tête”
- ▶ Le skeleton est générique
- ▶ Un objet qui à des méthodes accessibles à distance est appelé objet distant
- ▶ La partie publique de RMI est dans `java.rmi`

Implémenter un objet distant

- ▶ Les seules méthodes accessibles à distance seront celles spécifiées dans l'interface `Remote`
 - ▶ Écriture d'une interface spécifique à l'objet, étendant l'interface `java.rmi.Remote`
- ▶ Chaque méthode distante doit annoncer lever l'exception `java.rmi.RemoteException`
 - ▶ Sert à indiquer les problèmes liés à la distribution
- ▶ L'objet devra fournir une implémentation de ces méthodes

Implémenter un objet distant

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface MonInterfaceDistante extends Remote {  
    public void echo() throws RemoteException;  
}
```

Cette interface indique que l'objet qui l'implémentera aura la méthode echo() callable à distance

Implémenter un objet distant

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

public class MonObjetDistant extends UnicastRemoteObject
implements MonInterfaceDistante {

    public MonObjetDistant() throws RemoteException {}
    public void echo() throws RemoteException{
        System.out.println(« Echo »);
    }
}
```

L'objet distant doit finalement implémenter les méthodes de l'interface

Hériter de `java.rmi.server.UnicastRemoteObject`

Et avoir un constructeur sans paramètre levant aussi l'exception

Utiliser un objet distant

- ▶ Pour utiliser un objet distant il faut
 - ▶ Connaître son interface
 - ▶ Le trouver!
 - ▶ L'utiliser
- ▶ RMI fournit un service de nommage permettant de localiser un objet par son nom : le registry
 - ▶ L'objet s'enregistre sous un nom « bien connu »
 - ▶ Les clients demandent une référence vers cet objet

Utiliser un objet distant

```
import java.rmi.RemoteException;

public class Client {
    public static void main(String[] args) {
        MonInterfaceDistante mod = ... // du code pour
                                        //trouver l'objet

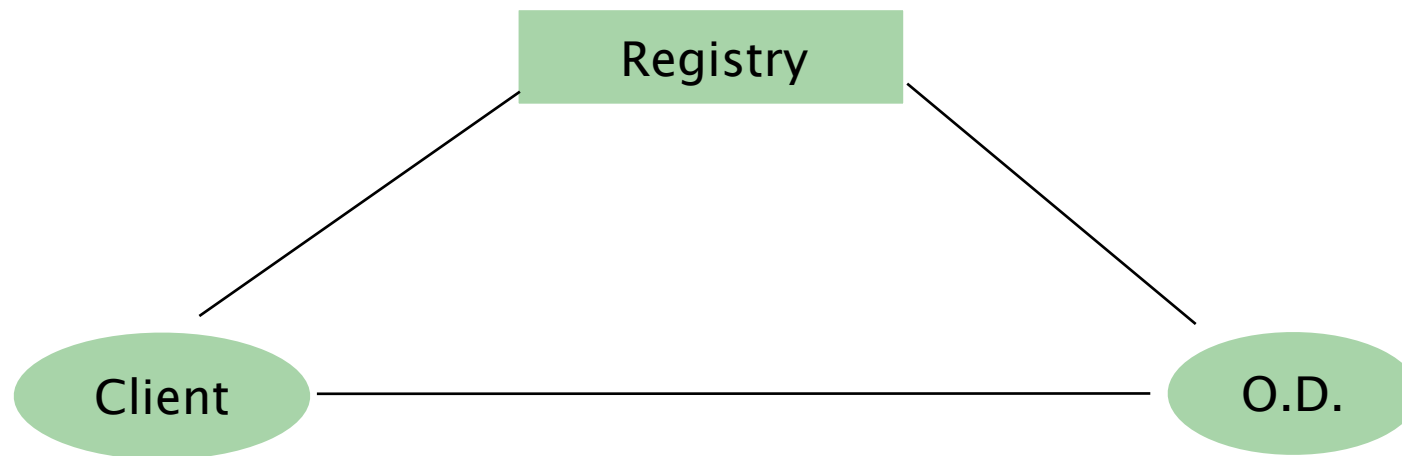
        try {
            mod.echo();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Trouver un objet distant

- ▶ L'objet doit s'enregistrer dans le registry
 - ▶ Programme lancé auparavant sur la même machine que l'objet distant : `rmiregistry`
 - ▶ Utilise le port 1099 par défaut
 - ▶ Possibilité de le démarrer depuis l'application (`LocateRegistry`)
- ▶ Il agit comme un service d'annuaire
- ▶ Les noms ressemblent à des URLs
 - ▶ `protocole://machine:port/nom`
 - ▶ Protocole, machine et port sont optionnels
 - ▶ Objet toto sur la machine locale: `///toto`
- ▶ Les méthodes de gestion sont regroupées dans la classe `Naming`
- ▶ L'objet distant appel `Naming.bind`
- ▶ Le client appel `Naming.lookup`

Démarrage d'une application RMI

- ▶ L'objet distant s'enregistre dans le registry
- ▶ Le client demande une référence au registry
- ▶ La référence sert ensuite pour appeler les méthodes



Générer les stubs et skeletons

- ▶ Une fois l'objet distant écrit, il est possible de générer les stubs et les skeletons
- ▶ Outils fournit dans Java: `rmic`
- ▶ Prends le nom complet de la classe distante (package+nom)
- ▶ Génère 2 fichiers (`_Stub` et `_Skel`)
- ▶ Ne met dans le stub que les méthodes spécifiées dans l'interface distante
- ▶ Possibilité de voir le code source avec l'option `-keep`
- ▶ Plus nécessaire depuis java 1.4

RMI: en résumé

- ▶ Écrire l'interface distante
- ▶ Écrire le code de l'objet distant
 - ▶ Implémenter l'interface
 - ▶ Ajouter le code pour le registry (en général dans le main ou le constructeur)
- ▶ Compiler
- ▶ Générer les stub et skeleton
- ▶ Écrire le client
 - ▶ Obtenir une référence vers l'objet distant
 - ▶ Appeler les méthodes
- ▶ Compiler
- ▶ Exécuter
 - ▶ Démarrer le serveur
 - ▶ Démarrer le client
 - ▶ Debugger :)

Résumé - RMI

- ▶ Un objet accessible à distance (objet distant) doit
 - ▶ Avoir une interface qui étend `Remote` et dont les méthodes lèvent une `RemoteException`
 - ▶ Sous classer `UnicastRemoteObject` et avoir un constructeur sans paramètre levant une `RemoteException`
- ▶ Pour trouver une référence vers un objet distant, on passe par un service d'annuaire, le registry

Compléments de RMI

Passage de paramètres

- ▶ Le but de RMI est de masquer la distribution
- ▶ Idéalement, Il faudrait avoir la même sémantique pour les passages de paramètre en Java centralisé et en RMI
- ▶ C'est-à-dire passage par copie
 - ▶ Copie de la valeur pour les types primitifs
 - ▶ Copie de la référence pour les objets (en C, c'est équivalent à passer un pointeur)
- ▶ En Java, on ne manipule jamais des objets!
 - ▶ La valeur d'une variable est soit un type primitif, soit une référence vers un objet

Passage de paramètres

```
public void foo(int a) {  
    a=a+1;  
}
```

```
public class MonInt {  
    public int i;  
    .....  
}  
public void foo(MonInt a) {  
    a.i=a.i+1;  
}
```

```
int x = 10;  
foo(x);  
// que vaut x ici?
```

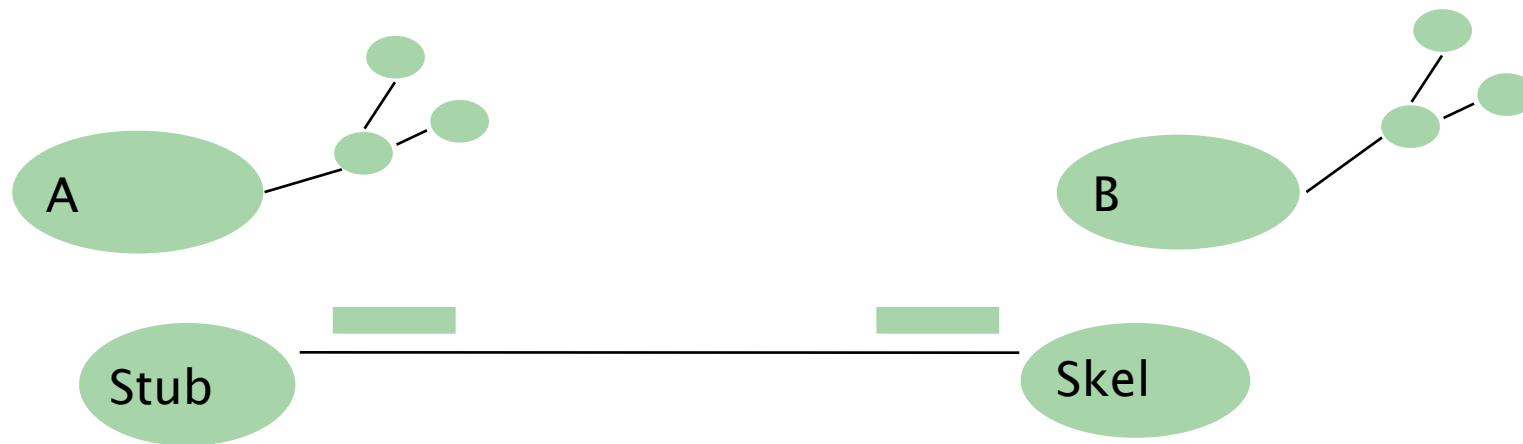
```
MonInt x = new MonInt(10);  
foo(x);  
// que contient x ici?
```

Passage de paramètre

- ▶ Peut-on faire la même chose en RMI
- ▶ Très facile pour les types primitifs, il suffit de les envoyer sur le réseau, ils sont automatiquement copiés
 - ▶ C'est le rôle du stub
- ▶ Plus compliqué pour les objets, car il faudrait
 - ▶ Envoyer la valeur de l'objet
 - ▶ Ramener les éventuelles modifications
 - ▶ Gestion de la concurrence non triviale
- ▶ Mais on peut avoir besoin d'un passage par référence
 - ▶ RMI le permet, seulement pour des références vers des objets distants
 - ▶ Que contient une référence vers un objet distant? Son Stub!
 - ▶ Il suffit donc de simplement copier le stub

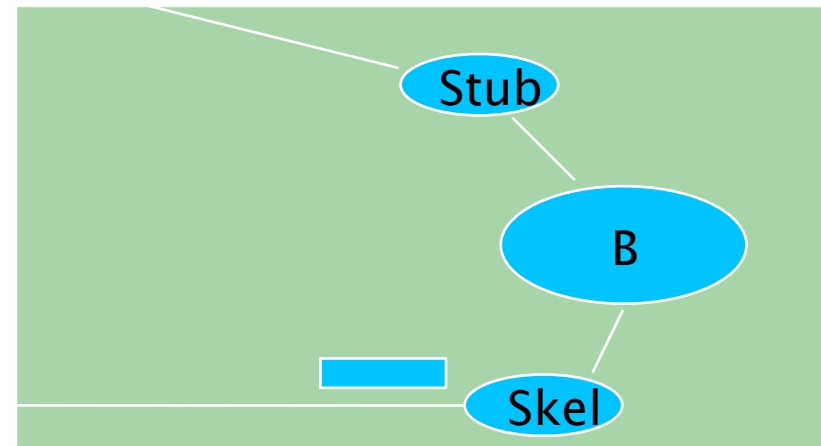
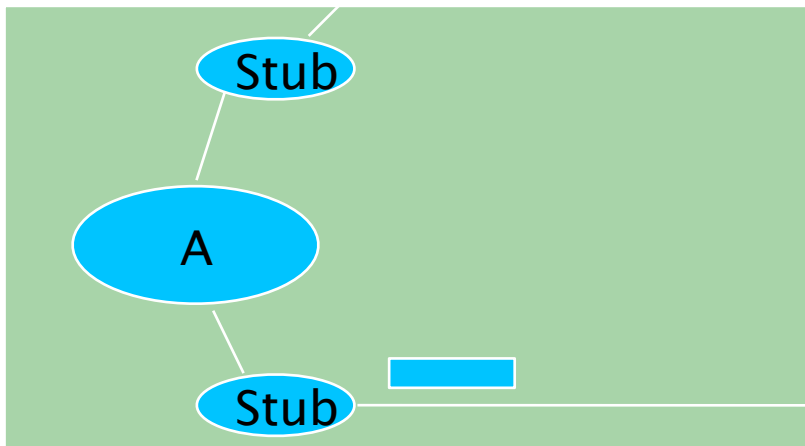
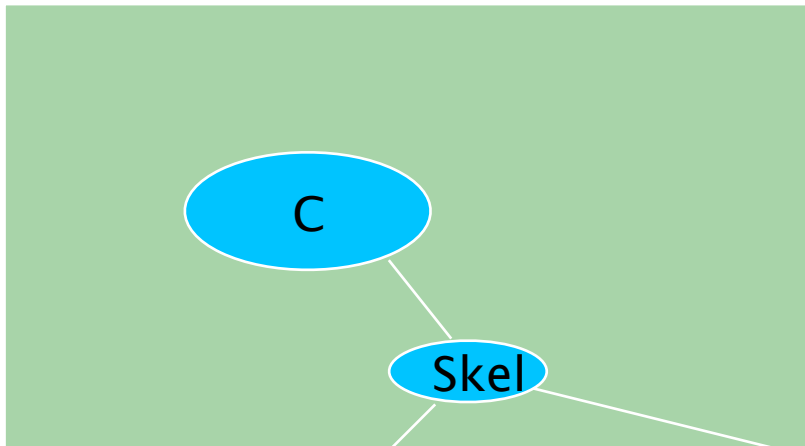
Passage de paramètre

- ▶ Un objet référencé est passé par copie profonde



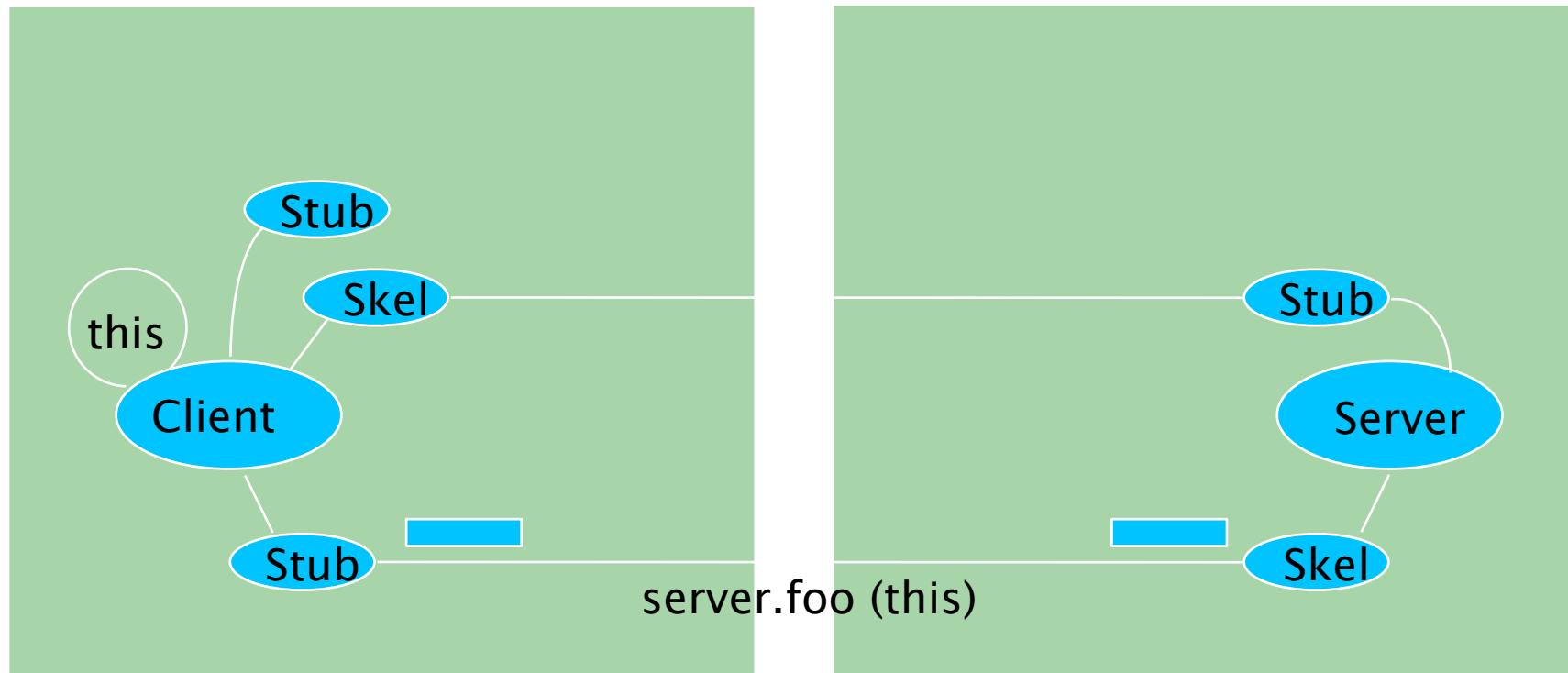
Passage de paramètres

Les références distantes sont passées par copie du stub



Passage de paramètres

Les références locales sont automatiquement converties en référence distante

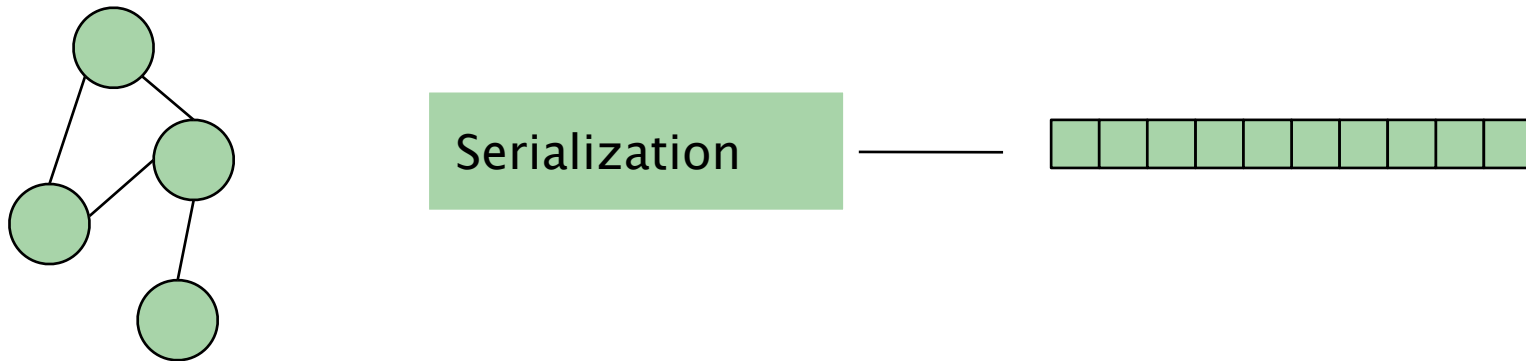


Résumons

- ▶ Toute variable primitive est passé par copie
- ▶ Tout objet est passé par copie
- ▶ Toute objet distant (abus de langage, on devrait dire référence distante) est passé par référence
- ▶ Mais comment copier un objet?
 - ▶ Il ne faut pas copier que l'objet
 - ▶ Il faut aussi copier toutes ses références
- ▶ Très fastidieux à faire à la main
- ▶ Heureusement, une API le fait pour nous: la Serialization

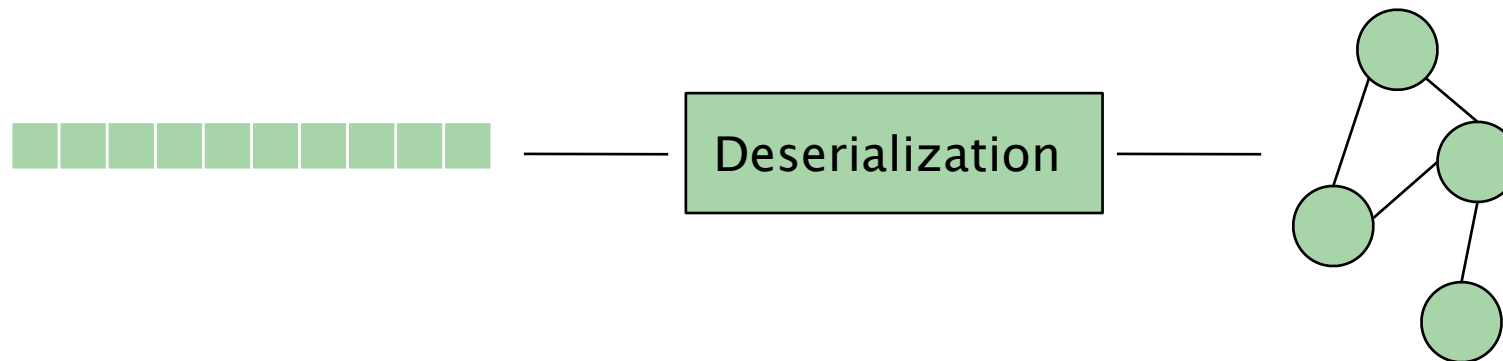
La Sérialisation

- ▶ Mécanisme générique transformant un graphe d'objets en flux d'octets
 - ▶ L'objet passé en paramètre est converti en tableau
 - ▶ Ainsi que tout ceux qu'il référence
 - ▶ Processus récursif (copie profonde)
- ▶ Fonctionnement de base
 - ▶ Encode le nom de la classe
 - ▶ Encode les valeurs des attributs



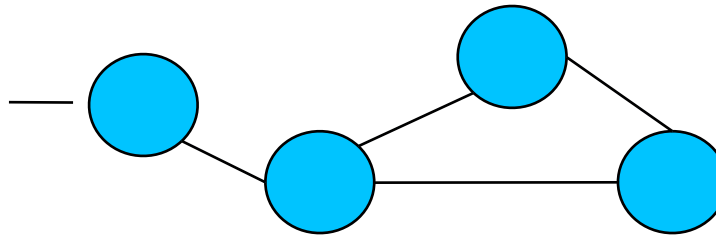
La Désérialisation

- ▶ Processus symétrique de la sérialisation
 - ▶ Prend en entrée un flux d'octets
 - ▶ Crée le graphe d'objet correspondant
- ▶ Fonctionnement de base
 - ▶ Lis le nom de la classe
 - ▶ Fabrique un objet de cette classe
 - ▶ Lis les champs dans le flux, et met à jour leur valeur dans la nouvelle instance



Gestion des cycles

- ▶ La sérialisation est un processus récursif
- ▶ Que se passe-t-il quand il y a un cycle dans le graphe d'objets?



Un algorithme naïf bouclerait à l'infini

Solution:

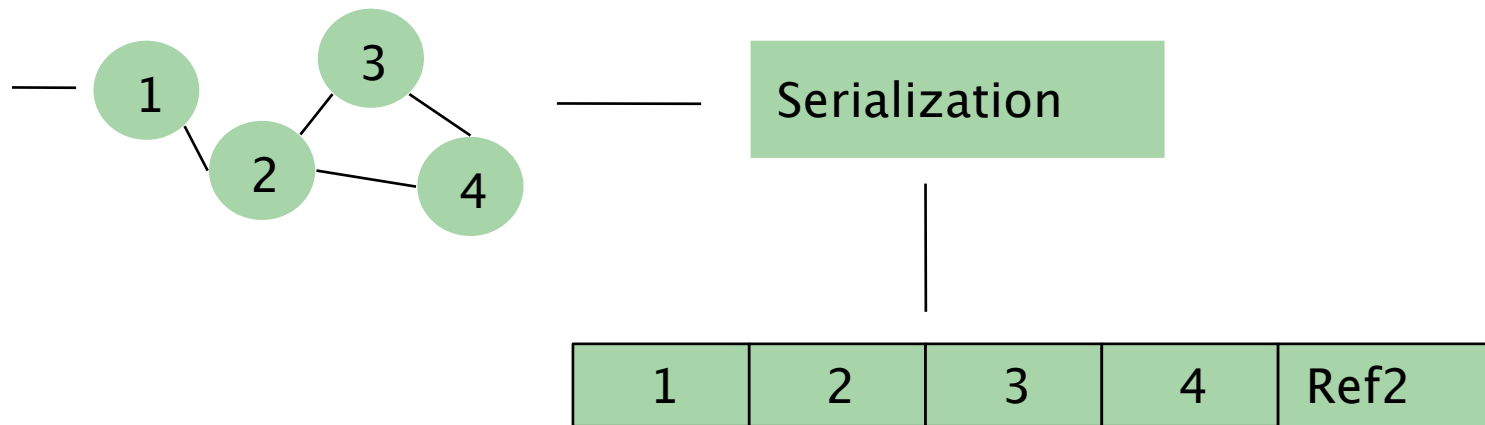
Repérer les cycles

La sérialisation se souvient des objets déjà sérialisés

Si on veut en sérialiser un à nouveau, alors met juste une référence

Gestion des cycles

- ▶ Le flux contient donc 3 types d'information
 - ▶ Le nom de la classe
 - ▶ Les valeurs des attributs
 - ▶ Des références vers d'autres parties du flux



Utiliser la sérialisation

- ▶ Par défaut, un objet n'est pas sérialisable
 - ▶ Problème de sécurité: la sérialisation ignore les droits d'accès (private...)
 - ▶ Levée d'une `NotSerializableException`
- ▶ Il faut donc explicitement indiquer qu'un objet est sérialisable
- ▶ Marquage au niveau de la classe
 - ▶ Toutes les instances seront sérialisable
 - ▶ Les sous classes d'une classe sérialisable sont sérialisable
- ▶ Utilisation de l'interface `java.io.Serializable`
 - ▶ Interface marqueur, aucune méthode à implémenter

Utiliser la sérialisation

- ▶ RMI fait appel à la sérialisation
 - ▶ Totalement transparent
- ▶ Mais on peut aussi l'utiliser manuellement
 - ▶ Très pratique pour copier des objets
- ▶ Étapes
 - ▶ Bien vérifier que les objets sont sérialisables
 - ▶ Créer des flux d'entrée et de sortie (input et output streams)
 - ▶ Utiliser ces flux pour créer des flux objets (object input et object output streams)
 - ▶ Passer l'objet à copier à l'object output stream
 - ▶ Le lire depuis l'object input stream

Utiliser la sérialisation

```
static public Object deepCopy(Object oldObj) throws Exception
{
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;
    try {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        oos = new ObjectOutputStream(bos);
        // serialize and pass the object
        oos.writeObject(oldObj);
        oos.flush();

        ByteArrayInputStream bin = new ByteArrayInputStream(bos.toByteArray());
        ois = new ObjectInputStream(bin);
        // return the new object
        return ois.readObject();
    }
    catch(Exception e) {
        System.out.println("Exception in ObjectCloner = " + e);
        throw(e);
    }
    finally {
        oos.close();
        ois.close();
    }
}
```

Source: <http://www.javaworld.com/javaworld/javatips/jw-javatip76-p2.html>

Contrôler la sérialisation

- ▶ Marquer une classe avec l'interface Serializable indique que tout ses champs seront sérialisés
- ▶ Pas forcément acceptable
 - ▶ Sécurité
 - ▶ Efficacité (pourquoi copier ce qui pourrait être recalculé plus rapidement?)
- ▶ Possibilité de contrôle plus fin
 - ▶ Marquage d'attributs comme étant non sérialisables: mots clé transient
 - ▶ Donner à un objet la possibilité de se sérialiser

Contrôler la sérialisation

- ▶ Pour modifier la sérialisation par défaut, il faut implémenter 2 méthodes
 - ▶ `writeObject()` : sérialisation
 - ▶ `readObject()` : désérialisation
- ▶ Leur signature est
 - ▶ `private void writeObject(ObjectOutputStream s) throws IOException`
 - ▶ `private void readObject(ObjectInputStream o) throws ClassNotFoundException, IOException`
- ▶ Elles seront automatiquement appelées et remplaceront le comportement par défaut
- ▶ On écrit dans ces méthodes du code spécifique à l'objet

Contrôler la sérialisation

- ▶ Dans les méthodes readObject/writeObject il est possible de tout faire
 - ▶ pas de limitation théorique
 - ▶ Manipulation/modification des attributs de l'objet possibles
- ▶ Basé sur les flots (streams de java.io)
 - ▶ Implémentation FIFO
 - ▶ Donc lecture dans le même ordre que l'écriture
- ▶ Symétrie
 - ▶ Normalement, lire tout ce qui a été écrit
 - ▶ En pratique, RMI délimite le flux et évite les mélanges

Écriture - Lecture

- ▶ Utilisation des méthodes de `ObjectOutputStream` et `ObjectInputStream`
 - ▶ Types primitifs
 - ▶ `{writelread}Double`, `{writelread}Int...`
 - ▶ Objets
 - ▶ `{writelread}Object`
 - ▶ Provoque une nouvelle serialization
- ▶ Possible de rappeler l'ancienne implémentation
 - ▶ Méthodes `defaultWriteObject()` et `defaultReadObject()` des streams
 - ▶ Très pratique pour ajouter une fonctionnalité

Exemple: comportement par défaut

```
public class Defaut implements Serializable {
    public Defaut() { }

    private void writeObject(ObjectOutputStream s) throws
        IOException {
        s.defaultWriteObject();
    }

    private void readObject(ObjectInputStream s) throws
        IOException, ClassNotFoundException {
        s.defaultReadObject();
    }
}
```

Exemple: sauvegarde d'un entier

```
public class Defaut implements Serializable {
    private int valeur;
    private double valeur2
    public Defaut() { }

    private void writeObject(ObjectOutputStream s) throws
        IOException {
        s.writeInt(valeur);
    }

    private void readObject(ObjectInputStream s) throws
        IOException, ClassNotFoundException {
        valeur = s.readInt();
    }
}
```

Sérialisation et héritage

- ▶ Les sous classes d'une classe sérialisable sont sérialisables
- ▶ Mais une classe peut-être sérialisable, alors que son parent ne l'est pas
 - ▶ La sous-classe est responsable de la sauvegarde/restauration des champs hérités
 - ▶ Lors de la désérialisation, les constructeurs sans paramètres seront appelés pour initialiser les champs non sérialisables
 - ▶ Le constructeur vide de la classe parent sera appelé
- ▶ Source de bugs très difficiles à identifier

RMI avancé

Répartition des classes

- ▶ RMI distingue deux types d'objets
 - ▶ Ceux accessibles à distance
 - ▶ Les autres
- ▶ Ils sont souvent sur des machines différentes (un client et un serveur)
- ▶ Comment sont réparties les classes?
- ▶ Client:
 - ▶ Implémentation du client
 - ▶ Interface distante
 - ▶ Stub
- ▶ Serveur:
 - ▶ Implémentation du serveur
 - ▶ Interface Distante
 - ▶ Skeleton

Téléchargement de classes

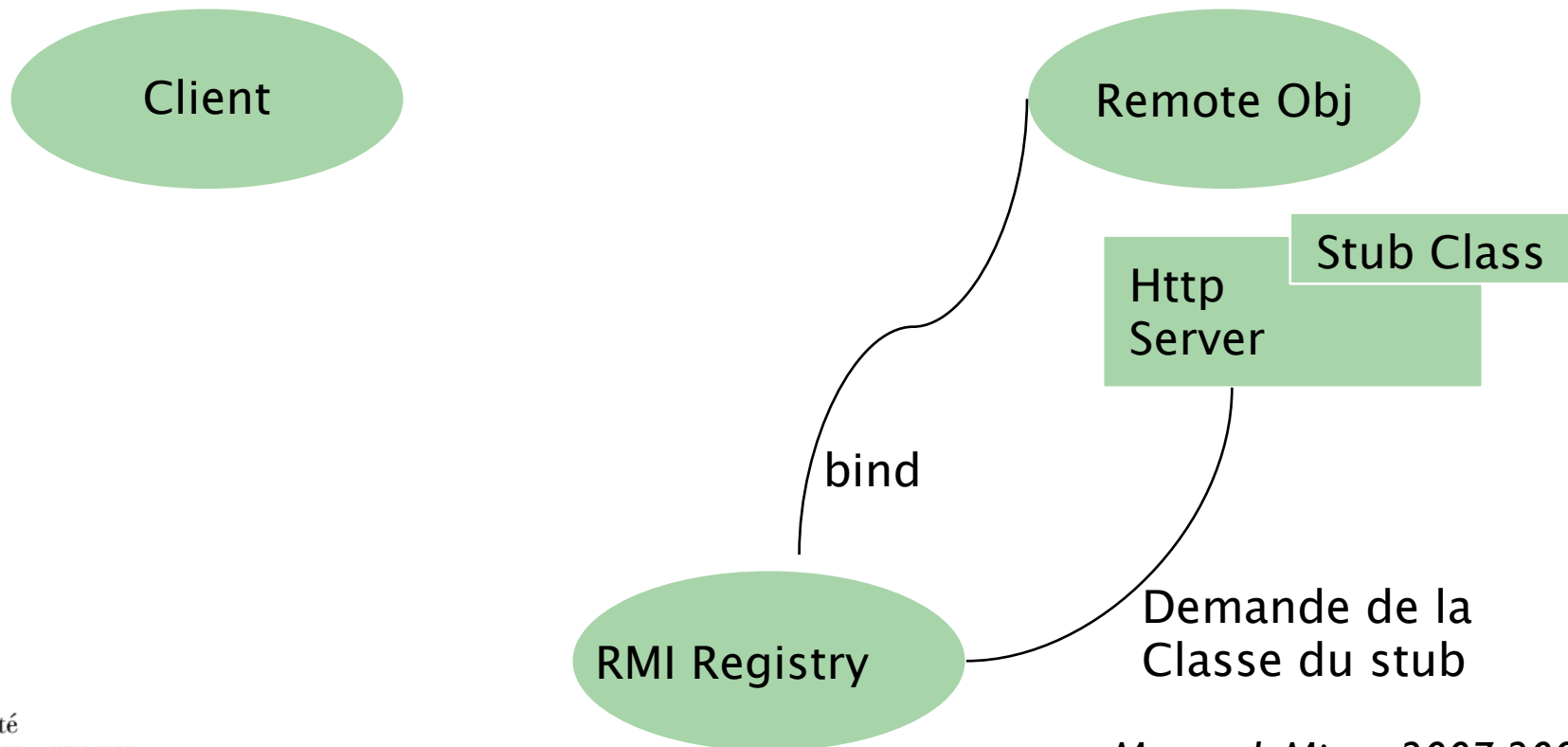
- ▶ Dans un monde parfait
 - ▶ Les classes sont distribuées
 - ▶ Rien ne change, tout est connu
- ▶ En pratique
 - ▶ Les classes sont plus ou moins bien distribuées
 - ▶ Certains ordinateurs n'ont pas les classes nécessaires
 - ▶ Ex: Appel d'une méthode distante avec en paramètre une sous classe de la classe déclarée dans la signature
- ▶ Solution: pouvoir télécharger les classes manquantes

Téléchargement de classes

- ▶ Mécanisme fourni par RMI
- ▶ Utilise HTTP
 - ▶ Permet de passer tous les firewalls
 - ▶ Mais nécessite un serveur HTTP
- ▶ Principe:
 - ▶ Les flots de sérialisation sont annotés avec des codebase
 - ▶ Ils indiquent où peut être téléchargé une classe si nécessaire
 - ▶ Lors de la désérialisation, si une classe manque, le serveur indiqué par le codebase est contacté
 - ▶ Si la classe est disponible, le programme continue, sinon, `ClassNotFoundException`

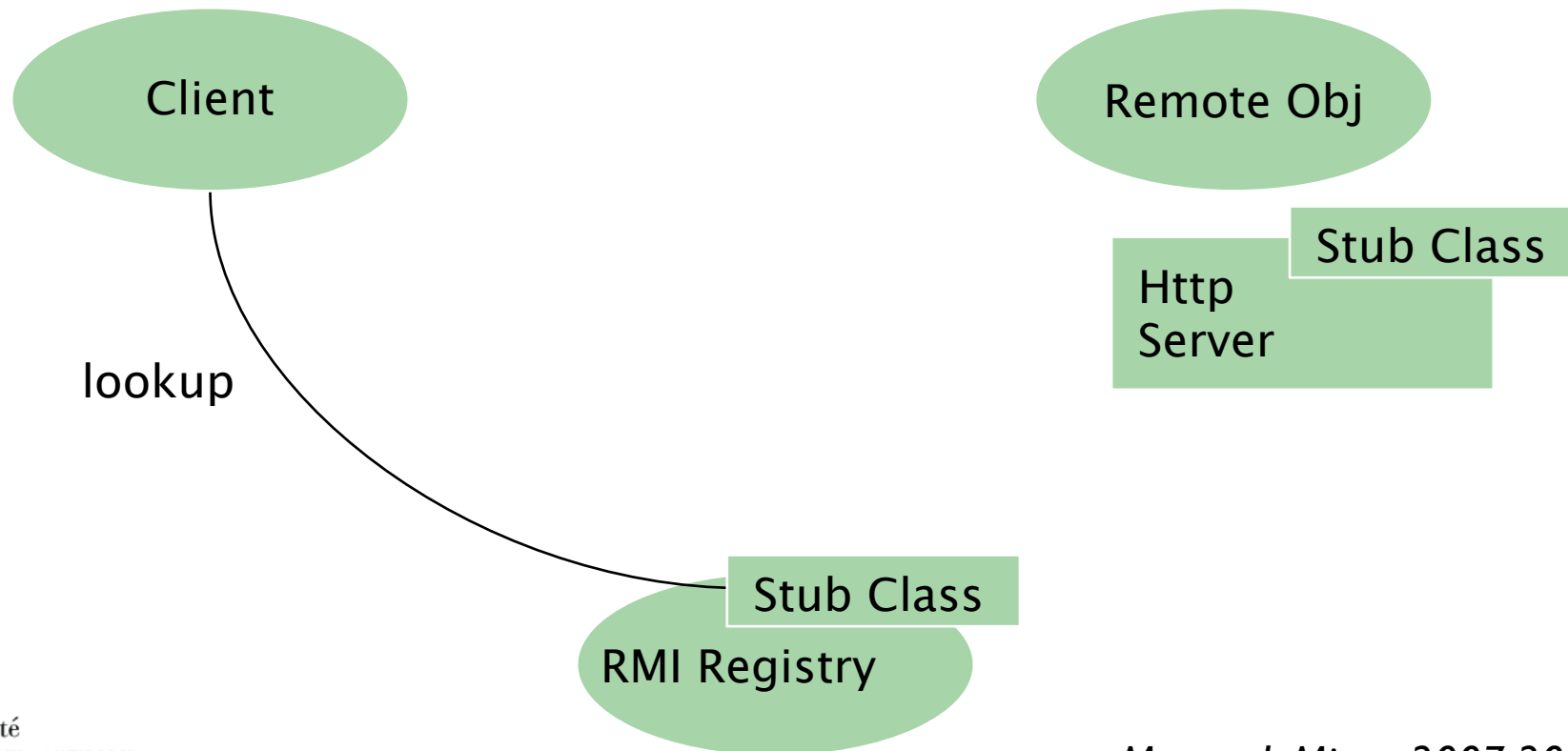
Exemple

- ▶ Déploiement minimal
 - ▶ Le stub n'est pas disponible pour le registry ou pour le client
 - ▶ Seul le remote object possède la classe

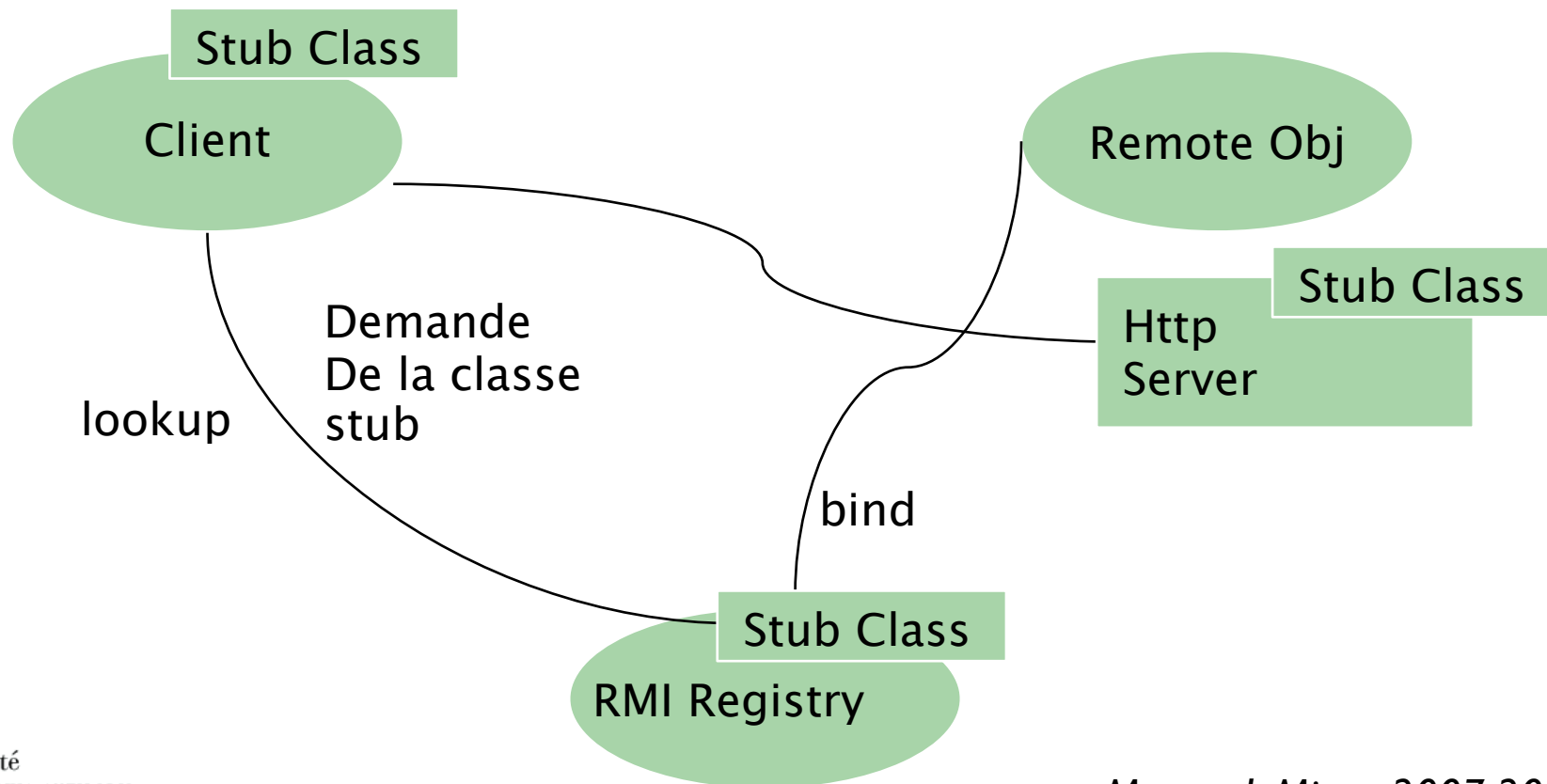


Exemple

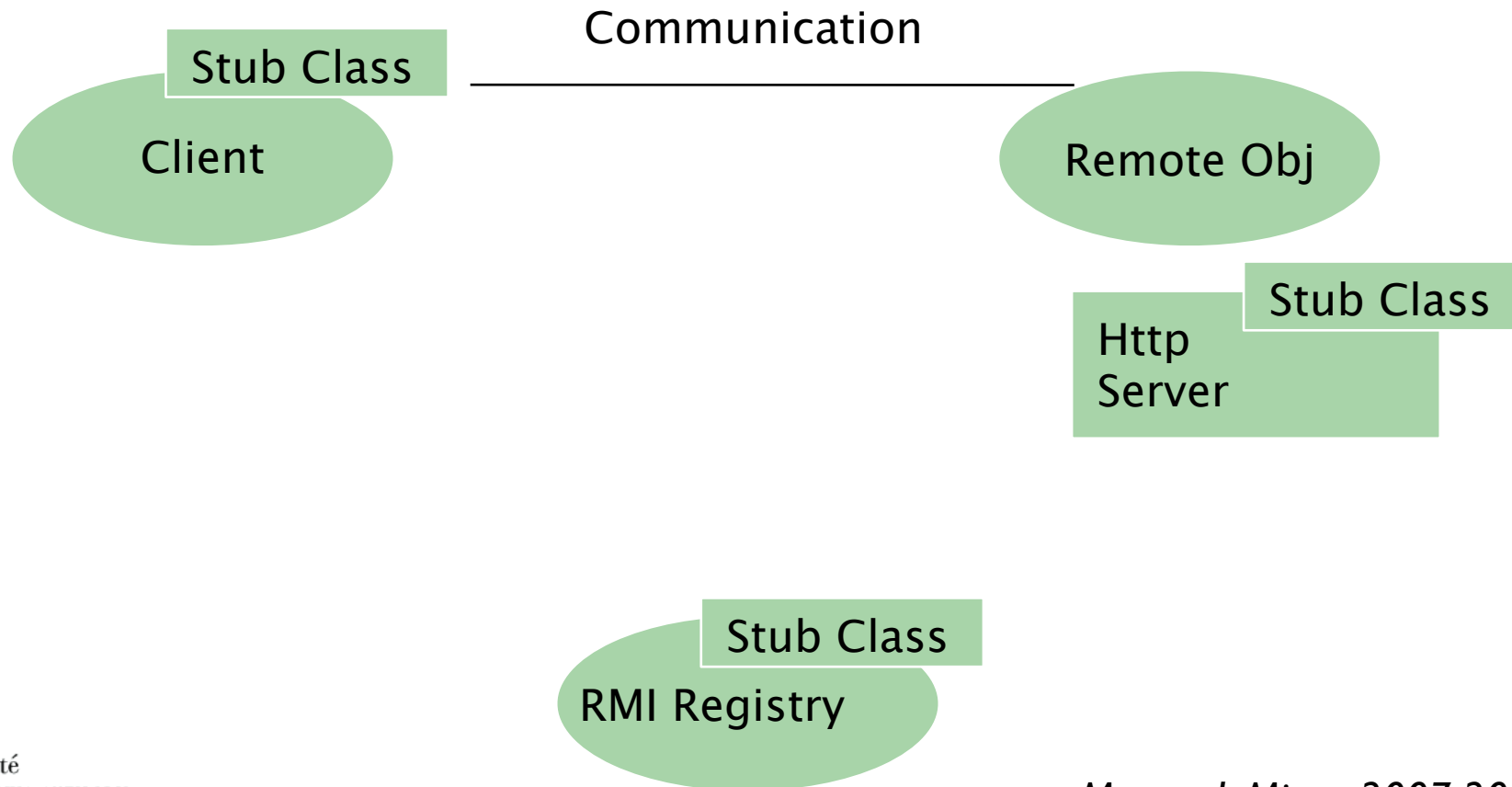
- ▶ Quand il contacte le registry, le client doit télécharger la classe



Exemple



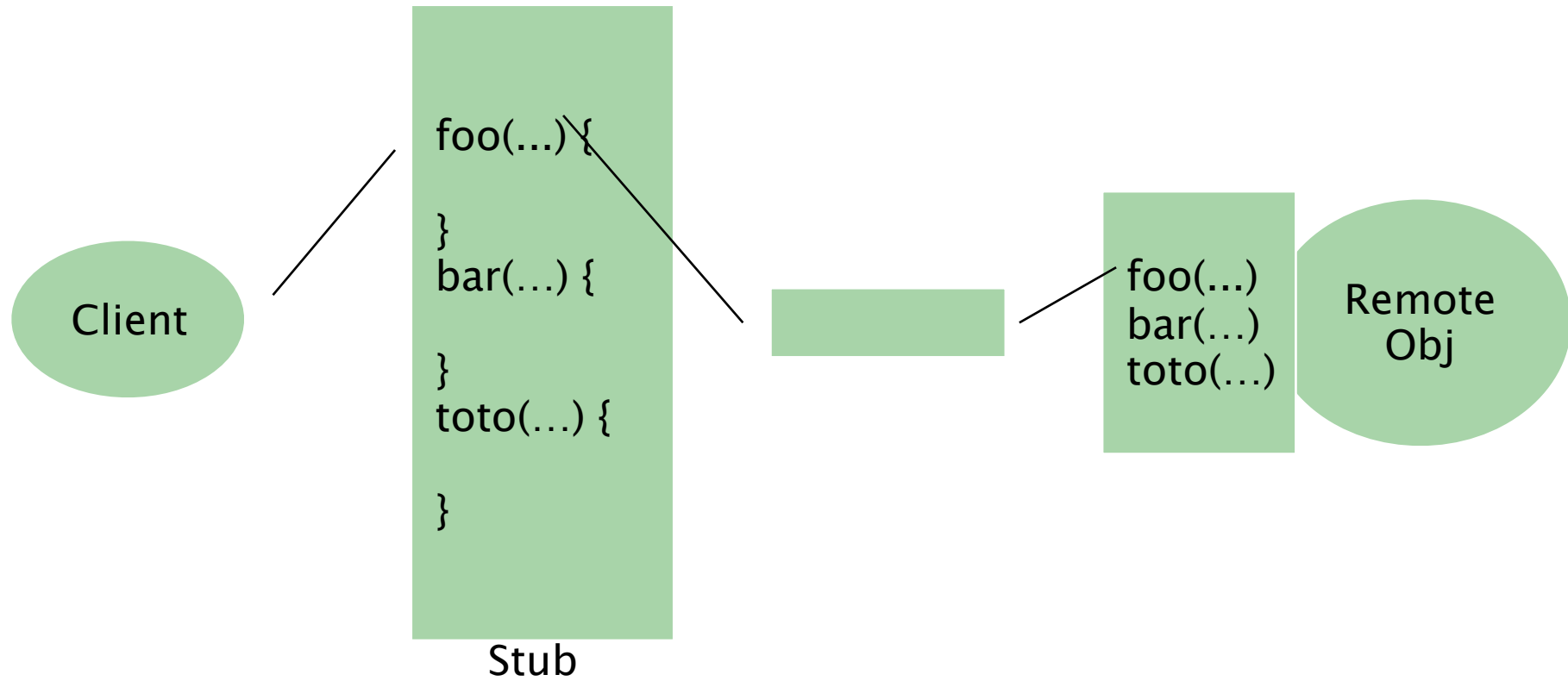
Exemple



Le Stub

- ▶ Le rôle du stub est de se faire passer pour l'objet distant
 - ▶ Il implémente l'interface distante
- ▶ Il doit aussi convertir l'appel de méthode en flot
 - ▶ Facile pour les paramètres
 - ▶ Pour la méthode, il suffit de la code sur quelques octets, convention avec le skeleton
- ▶ Et attendre le résultat en retour
 - ▶ Lecture sur une socket et désérialisation
- ▶ Donc un stub, c'est très simple!
- ▶ Tellement simple qu'on peut le générer à l'exécution (cf Java 1.5)

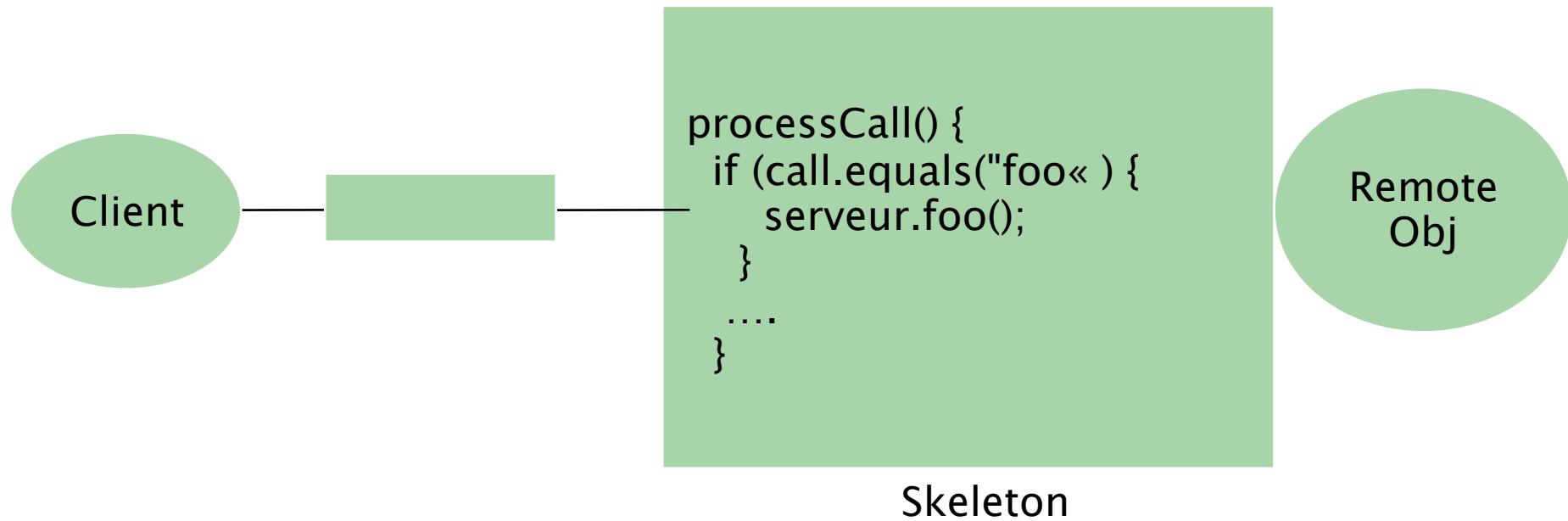
Le Stub



Le Skeleton

- ▶ Le skeleton appelle les méthodes sur l'objet distant
- ▶ Et retourne le résultat
- ▶ Est-il dépendant de l'objet distant?
 - ▶ Oui si implémentation statique (naïf)

Le Skeleton (version naïve)



Le Skeleton

- ▶ Y'a-t-il moyen de séparer le skeleton de l'objet appelé?
 - ▶ Oui, si on a un moyen de dire "je veux appeler la méthode dont le nom est foo" sans l'écrire explicitement
- ▶ Reflection
 - ▶ Capacité qu'à un programme à observer ou modifier ses structures internes de haut niveau
 - ▶ Concrètement, le langage permet de manipuler des objets qui représentent des appels de méthodes, des champs...
 - ▶ On fabrique un objet qui représente une méthode, et on demande son exécution
 - ▶ Partie intégrante de Java. Vital pour RMI, la sérialization...

Exemple de reflexion

```
String firstWord = "blih";
String secondWord = "blah";
String result = null;
Class c = String.class;
Class[] parameterTypes = new Class[] {String.class};
Method concatMethod;
Object[] arguments = new Object[] {secondWord};

concatMethod = c.getMethod("concat",

parameterTypes);
result = (String) concatMethod.invoke(firstWord,arguments);
```

RMI et les threads

- ▶ Un appel RMI est initié par un thread côté appelant
- ▶ Mais exécuté par un autre thread côté appelé
- ▶ Le thread de l'appelant est bloqué jusqu'à ce que le thread du côté appelé ai fini l'exécution
- ▶ Si multiples appelants, multiples threads côté appelé
 - ▶ Un objet distant est par essence multithread!
 - ▶ Il faut gérer la synchronisation des threads (synchronized, wait, notify)
- ▶ Pas de lien entre le thread appelant et le thread côté appelé

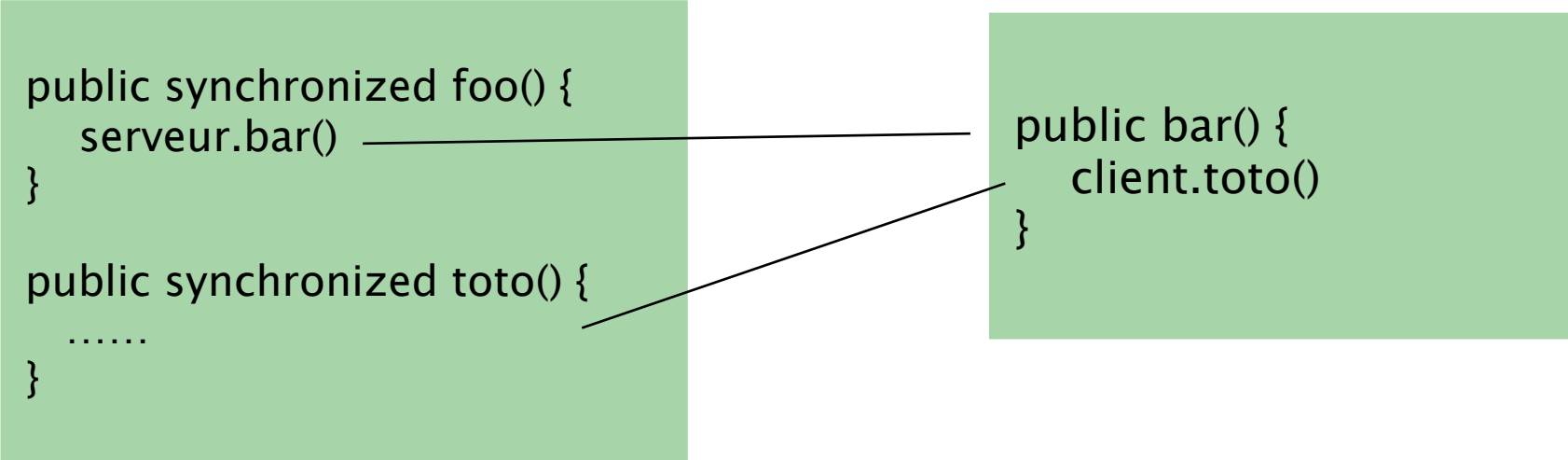
RMI et les threads

- ▶ L'implémentation n'est pas spécifiée
- ▶ En pratique
 - ▶ Lorsqu'un appel arrive, RMI crée un thread pour l'exécuter
 - ▶ Une fois l'appel fini, le thread attend un nouvel appel
 - ▶ Si multiples appels simultanés, de nouveaux threads sont créés
- ▶ Technique du thread pool
- ▶ Problème des appels ré-entrants
 - ▶ A fait un appel distant sur B, qui fait un appel distant sur A
 - ▶ Très courant: cycle dans le graph d'objets
 - ▶ Pas de problèmes dans la plupart des cas (si ce n'est la latence)
 - ▶ Gros problèmes si les méthodes sont synchronized

Deadlock distribué

```
public synchronized foo() {  
    serveur.bar()  
}  
  
public synchronized toto() {  
    .....  
}
```

```
public bar() {  
    client.toto()  
}
```



2 objets distants communiquent

Cycle dans le graph d'appels

Le thread ne peut pas entrer dans la méthode toto() tant que l'autre thread n'a pas fini d'exécuter foo

Deadlock!

Très difficile à identifier

Pas de vue globale de l'application

Pas d'information de la JVM