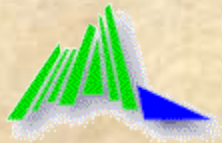


Programmation Orientée Objet
Langage Java™

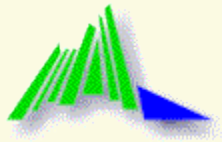
Manuel Munier

IUT des Pays de l'Adour - Mont de Marsan

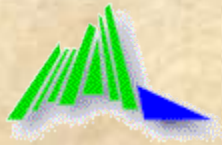
Département RT

2012-2013

Plan du cours



- Programmation Orientée Objet (POO)
  - modularité, encapsulation
  - objets, classes, instances, messages
  - héritage, liaison dynamique, généricité
- Langage Java
  - syntaxe du langage
  - classes et objets en Java
  - héritage, interfaces
  - paquetages, exceptions
  - architecture client/serveur (RMI)
  - Java et son environnement



Partie n°2

Langage Java™

Plan

- Pourquoi Java ?
- Syntaxe du langage
- Classes et objets en Java
- Héritage, interfaces
- Paquetages, exceptions
- Architecture client/serveur (RMI)
- Java et son environnement

Pourquoi Java ?

- Actuellement, C++ est le langage le plus utilisé, tant pour l'enseignement que pour le développement d'applications
- Toutefois, depuis 1996 Java est devenu un sérieux concurrent
  - fiabilité du langage
  - philosophie «write once, run everywhere»
  - API (Application Programming Interface)
  - explosion des applets sur le Web

Fiabilité

- Java supporte la majorité des concepts orientés objet
- La plupart des problèmes rencontrés en C++ par les novices sont absents de Java
  - pas de mélange entre POO et prog. structurée
  - compilateur bien plus strict sur les erreurs de programmation classiques
  - mécanisme de gestion des erreurs lors de l'exécution: exceptions
  - pas de type «pointeur» explicite

Portabilité

- Une fois compilé en bytecode, un programme Java peut être exécuté par n'importe quelle machine virtuelle Java
  - un entier a le même intervalle de valeurs dans toutes les implémentations de Java, quelle que soit l'architecture matérielle
- Idem pour toutes les classes de l'API
  - accès aux fichiers et au réseau, gestion des processus, interface graphique,…
- Contrepartie: interprétation du bytecode

API

- API = tout ce qui relie un programme avec son environnement + tout ce qui n'est pas défini directement dans le langage
- Développer avec Java c'est connaître le langage et l'API
- L'API est strictement la même sur toutes les plates-formes

API

java.lang: classes essentielles du langage (Object, String, Math, Thread,…)

java.io: entrées/sorties depuis des fichiers, chaînes de caractères, tubes,…

java.util: structure de données (dictionnaire, séquence, pile, table hachée,…) et autres (nombres aléatoires, dates,…)

java.net: utilisation d'un réseau (sockets, adresses Internet, URL,…)

java.awt: Abstract Window Toolkit

java.rmi: Remote Method Invocation

Applets

- Applet = petite application
- Une applet est un programme Java capable de s'exécuter au sein d'un navigateur Web
- Plusieurs mécanismes sont fournis
  - exécution dans la machine locale
  - chargement du code via le réseau
  - code minimal à la réalisation de la tâche
  - gestion de la sécurité (accès aux ressources locales, authentification,…)

Bilan

- Java en enseignement
  - syntaxe proche de celle du C++
  - propreté du langage vis-à-vis de la POO
  - rigueur du compilateur
  - erreurs d'exécution gérées par des exceptions
  - utilisation possible sous Windows et Unix
- Et pour le développement
  - portabilité ⇒ une seule version du logiciel
  - environnement standard complet (API)
  - applets dans les pages Web

Plan

- Pourquoi Java ?
- Syntaxe du langage
- Classes et objets en Java
- Héritage, interfaces
- Paquetages, exceptions
- Architecture client/serveur (RMI)
- Java et son environnement

Syntaxe du langage

- Nous allons tout d'abord étudier les éléments de base du langage Java
  - types de données
  - déclaration de variables
  - tableaux
  - opérateurs
  - conversions de type
  - structures de contrôle: tests, boucles,...

Commentaires

Syntaxe

- Java dispose de 3 types de commentaires
  - tout texte situé entre `//` et la fin de la ligne est ignoré par le compilateur
  - idem pour tout ce qui se trouve entre `/*` et le `*/` suivant
  - le 3ème type de commentaire commence par `/**` et se termine par `*/`; ces commentaires de documentation sont utilisés par l'outil **`javadoc`** pour générer automatiquement une doc HTML

Identificateurs

Syntaxe

- Pour nommer les variables, les classes, les méthodes, les packages, les programmes,…
- Caractères valides
  - les lettres `a..z` ou `A..Z`
  - les chiffres de `0` à `9`
  - les caractères `_` et `$`
  - les caractères Unicode supérieurs à `0x00C0` (caractères nationaux tels que `ç`,`ü`,…)
- 1er caractère dans `{a..z,A..Z,_,$}`

Identificateurs

Syntaxe

- Exemples d'identificateurs valides
  - `$valeur_system`
  - `dateDeNaissance`
  - `ISO9000`
- Exemples d'identificateurs invalides
  - `ça    // ç comme premier caractère`
  - `9neuf // 9 comme premier caractère`
  - `note# // # pas au-dessus de 0x00C0`
  - `long  // ok, mais mot réservé !`

Mots réservés

Syntaxe

```
abstract    extends     native      throw
boolean     false       new         throws
break       final       null        transient
byte        finally     package     true
case        float       private     try
catch       for         protected   void
char        goto        public      volatile
class       if          return      while
const       implements  short
continue    import      static
default     instanceof  super
do          int         switch
double      interface   synchronized
else        long        this
```

Types primitifs

Syntaxe

- Java dispose de types «primitifs» pouvant stocker des données directement compréhensibles par Java
  - **`boolean`** booléen valant **`true`** ou **`false`**
  - **`char`** caractère 16 bits Unicode
  - **`byte`** entier codé sur 8 bits (signé)
  - **`short`** entier codé sur 16 bits (signé)
  - **`int`** entier codé sur 32 bits (signé)
  - **`long`** entier codé sur 64 bits (signé)
  - **`float`** réel codé sur 32 bits
  - **`double`** réel codé sur 64 bits

Booléens

Syntaxe

- Deux valeurs possibles
  - `true` (vrai)
  - `false` (faux)
- Ne peut pas être assimilé à `0` ou `1` comme en langage C ou C++
- Exemples:
  - `boolean resultatAtteint = false;`
  - `boolean continuer = true;`

Entiers

Syntaxe

- Trois formats possibles
  - décimal
  - octal
  - hexadécimal
- En décimal ne commence jamais par un zéro
  - `int nbreMois = 12;`
- En octal, précédé d'un zéro
  - `int nbreDoigts = 012; // 10 en décimal`
- En hexadécimal, précédé d'un `0x` ou `0X`
  - `int dixHuit = 0x12;`

Réels

Syntaxe

- Un réel possède
  - une mantisse
  - éventuellement un exposant en puissance de 10
  - obligatoirement un point décimal ou un exposant
- Exemples de réels sans partie exposant:
  - `2.` | `.5` | `2.5` | `2.0` | `.001`
- Exemples de réels avec partie exposant:
  - `2E0` | `2E3` | `2E-3` | `.2e+3` | `2.5e-3`

Caractères

Syntaxe

- Un caractère est noté entre deux apostrophes (quotation simple)
  - `'x' | 'a' | '4'`
- Quelques caractères Unicode spéciaux

Continuation	`\`	Backslash (\)	`\\`
Nouvelle ligne (N)	`\n`	Apostrophe (')	`\ '`
Tabulation (HT)	`\t`	Guillement (")	`\"`
Retour arrière (BS)	`\b`	Car. octal (0377)	`\377`
Retour chariot (CR)	`\r`	Car. hexa (0xFF)	`\xFF`
Saut de page (FF)	`\f`	Car. Unicode (0xFFFF)	`\uFFFF`

Chaînes de caractères

Syntaxe

- C'est une suite de caractères entourée de guillemets (")
- Ce n'est pas un type primitif; ce sont des instances de la classe `String`

```
""
"\""
"ligne de texte"
"continuation\
d'une ligne"
"retour\n chariot"
```

affichage →

```

"
ligne de texte
continuation d'une ligne

retour
chariot
```

Déclaration de variables

Syntaxe

- Toute variable doit obligatoirement avoir été déclarée avant d'être utilisée (langage fortement typé)
- Une variable n'est visible qu'à l'intérieur du bloc d'instructions où elle a été définie
- Une variable définie dans un bloc peut être redéfinie dans un sous-bloc. Cette nouvelle variable masque la variable supérieure.
  - var. temporaires, des indices de boucles,…

Déclaration de variables

Syntaxe

- Le compilateur se charge de vérifier
  - la compatibilité des types dans les expressions
  - la visibilité des variables
- Exemples de variables:
  - `int i,j,k;`
  - `float note = 13.5;`
- Si on ajoute le mot-clé **`final`**, c'est une déclaration de constante
  - `final int nbreRoues = 4;`
  - `final float pi = 3.14159;`

Tableaux

Syntaxe

- Un tableau est une collection de variables du même type. L'indice est un entier
- Il suffit de post-fixer le type ou la variable par []
  - `int i[];  // vecteur d'entiers`
  - `int[] j;  // j a le même type que i`
  - `char motsCroises[][]; // matrice de car.`
- Les tableaux **ne sont pas contraints** lors de la déclaration

Tableaux

Syntaxe

- L'allocation est réalisée via l'opérateur **new**

```
// i est un vecteur de 100 entiers
int i[] = new int[100];

// tab est une matrice de 10 par 10
char tab[][] = new char[10][10];
```

- Les indices commencent à 0

```
i[0]=1;    // première position de i à 1
i[99]=100; // dernière position de i à 100

tab[0][0]='a'; // première case de tab
tab[9][9]='z'; // dernière case de tab
```

Tableaux

Syntaxe

- Lors de l'allocation, seule la 1ère dimension **doit** être contrainte pour un tableau ayant plusieurs dimensions
  - `char tab[][] = new char[10][];`
- La taille d'un tableau est fixée à sa création et ne peut plus être modifiée
- Les autres dimensions peuvent être déclarées à l'exécution
  - `tab[0] = new char[24];`
  - `tab[1] = new char[12];`

} *tailles différentes*

Tableaux

Syntaxe

- Au lieu de …

```
char tab[][] = new char[10][10];
```

- … on pourrait donc réaliser l'allocation à l'aide d'une boucle

```
char tab[][] = new char[10][];
for (int i=0; i<tab.length; i++)
    // allocation de chaque ligne de tab
    tab[i] = new char[10];
```

Affectation

Syntaxe

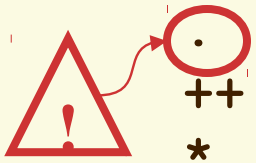
- L'affectation (opérateur =) assigne la valeur de son opérande droite (une expression) à son opérande gauche (une variable)

```
j = 2;
i = j*3;
k = i + 4 - j/(3*i^2)
```

- Les expressions sont évaluées de la gauche vers la droite en respectant la priorité des opérateurs (voir table de précédence)

Table de précédence

de la plus haute à la plus basse (et de la gauche vers la droite sur une même ligne)



```
.       []      ()
++      --      !       instanceof
*       /       %
+       -
<<      >>      >>>
<       >       <=      >=
==      !=
&       |       ^
&&      ||
?:
=       op=
```

Opérateurs numériques

Syntaxe

- Opérateurs unaires

-	négation	`i=-j;`
++	incrémentation de 1	`i++; ++j;`
--	décrémentation de 1	`i--; --j;`

- Bien évidemment, il faut toujours faire attention aux effets de bord induits par les opérateurs ++ et --

Opérateurs numériques

Syntaxe

- Opérateurs binaires

+ addition `i=j+k;`

+= `i+=2; // i=i+2`

- soustraction `i=j-k;`

-= `i-=j; // i=i-j`

* multiplication `x=2*y;`

*= `x*=x; // x=x*x`

/ division (tronque si les arg. sont entiers) `i=j/k;`

/= `x/=10; // x=x/10`

% modulo `i=j%k;`

%= `i%=2; // i=i%2`

Opérateurs relationnels

Syntaxe

- Opérateurs binaires

< plus petit que `x<i;`

> plus grand que `i>100;`

<= plus petit ou égal que `j<=k;`

>= plus grand ou égal que `c>='a';`

== égal à `i==20;`

!= différent de `c!='z';`

- Attention à ne pas confondre == et =. Mais grâce à la rigueur du compilateur Java, cela génèrera une erreur à la compilation…

Opérateurs logiques

Syntaxe

- Opérateurs binaires sur les booléens

! négation `!p`

& ET `p & (i<10)`

| OU `p |q`

^ OU exclusif `p ^ false`

&& ET évalué `p && q && r`

|| OU évalué `p || q || r`

!= négation assignée `p!=p;`

&= ET assigné `p&=q; // p=p&q`

|= OU assigné `p|=q; // p=p|q`

?: Si alors sinon `(i<10)?(j=2):(j=3)`

Opérateurs bit-à-bit

Syntaxe

- Les opérateurs **!**, **&**, **|** et **^** sont étendus à des opérations de manipulation binaire bit-à-bit sur des opérandes entières
- Opérateurs binaires de décalage

<< décalage à gauche
>> décalage à droite signé
>>> décalage à droite non signé
<<= décalage à gauche assignée
>>= décalage à droite signé assignée
>>>= décalage à droite non signé assignée

Conversion de type

Syntaxe

- Conversions **implicites**
  - `char` → `int`
  - `float` → `double`
  - `int` → `float`
  - `long` → `float` (perte d'info. possible)
- Les autres conversions doivent être **explicites** (mais n'empêchent pas les pertes d'information)

```
double d = 3.14159;
long l = (long)d;
```

Structures de contrôle

Syntaxe

- On retrouve en Java les mêmes structures de contrôle que dans les autres langages de programmation
  - si-sinon
  - aiguillage

  } *conditionnelles*

  - tant-que
  - faire-tant-que
  - pour

  } *boucles*

Bloc d'instructions

Syntaxe

- C'est une succession d'instructions séparées par des ;
- Un bloc est délimité par { et }
- Si les variables peuvent être définies n'importe où dans un bloc, il est préférable de le faire en début de bloc
- Les variables définies dans un bloc ne sont visibles que dans ce bloc (et ses sous-blocs)

Si-sinon

Syntaxe

- La condition est obligatoirement une **expression booléenne** comprise entre **()**

```
if (i==1)
   {
     s = "i est égal à 1";
     j = i;
   }
else
   {
     s = "i est différent de 1";
     j = 1770;
   }
```

Si-sinon

Syntaxe

- On peut cascader les conditions

```
if (i==1)
  s = "i est égal à 1";
else if (i==2)
  s = "i est égal à 2";
else if (i==3)
  s = "i est égal à 3";
else
  s = "i est différent de 1, 2 et 3";
```

Si-sinon

Syntaxe

- Penser à utiliser { } pour délimiter les blocs

```
i=0; j=3;
k=i+1;
if (i==0)
  if (j==2) k=i;
else k=j;
System.out.println(k);  // affichage de k
```

- Le programme va imprimer 3 (`k=j`) et non 1 (`k=i+1`) car le `else` se rapporte au `if` le plus imbriqué !

Aiguillage

Syntaxe

- Permet de distinguer plusieurs cas
- L'expression évaluée par le **`switch`** doit être de type **`char`**, **`byte`**, **`short`** ou **`int`**
- Le résultat de cette expression est ensuite comparé aux différentes **constantes** indiquées dans les **`case`**
- L'exécution commence alors à cet endroit (et exécute donc aussi les cas suivants)

Aiguillage

Syntaxe

- Si on veut isoler chaque cas, il faut utiliser l'instruction **`break`**
- On peut placer, à la fin du **`switch`**, un cas par défaut à l'aide de la clause **`default`**
- **`Case`** et **`default`** peuvent être considérés comme des étiquettes

Aiguillage

Syntaxe

```
int i=3;
switch (i)
  {
    case 1:  s = "I";   break;
    case 2:  s = "II";  break
    case 3:  s = "III"; break;
    case 4:  s = "IV";  break;
    case 5:  s = "V";   break;
    default: s = "pas de traduction";
  }
System.out.println(s);
```

Tant-que

Syntaxe

- La condition d'arrêt de la boucle est obligatoirement une **expression booléenne** comprise entre **()**

```
int i=100, somme=0, j=0;
// boucle sans effet de bord
while (j<=i)
   {
     somme += j;
     j++;
   }
System.out.println(somme);
```

Tant-que

Syntaxe

- Comme toujours, il faut faire attention aux effets de bord et ne les utiliser qu'avec parcimonie !

```
int i=100, somme=0, j=0;
// boucle avec effet de bord
while (j++<=i) somme += j;
System.out.println(somme);
```

Faire-tant-que

Syntaxe

- Dans ce cas, la condition d'arrêt est évaluée après l'exécution du corps de la boucle

```
int i=100, somme=0, j=0;
do
   {
     somme += j;
     j++;
   } while (j<=i);
System.out.println(somme);
```

Pour

Syntaxe

- Basé sur un itérateur contrôlé par la boucle elle-même (ou à la main dans le corps)

```
int n[] = new int[100];

// boucle 1: calcul de la somme des entiers
for (int i=1; i<100; i++)
  n[i] = n[i-1]+i;

// boucle 2: imprimer le résultat de 99 à 0
for (int j=99; j>=0;)
  {
    System.out.println("somme("+i+")="+n[i]);
    j++;  // modif. manuelle de l'itérateur
  }
```

Pour

Syntaxe

- Format général

```
for (instr1;cond;instr2) {bloc};
```

  - `instr1` : initialise l'itérateur (et le déclare)
  - `cond` : condition d'arrêt
  - `instr2` : modification de l'itérateur
  - `bloc` : corps de la boucle

- C'est équivalent à

```
instr1; while (cond) {bloc; instr2};
```

Pour

Syntaxe

- Format général

```
for (instr1;cond;instr2) {bloc};
```

- **instr1**, **instr2** et **cond** sont optionnels
  - si **intr2** est omis, l'itérateur doit être modifié manuellement dans le corps de la boucle (**bloc**)
  - si la condition **cond** est omise, il faudra utiliser un **break** pour quitter la boucle

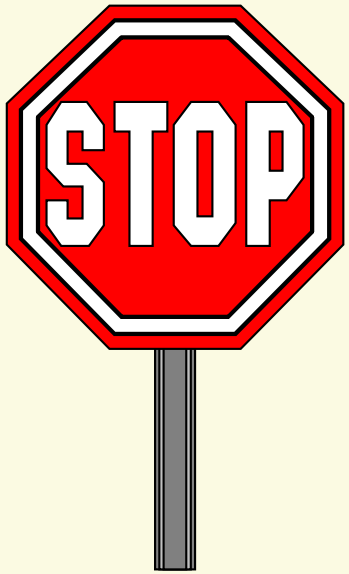
```
for (;;) System.out.println("boucle infinie");
```

On se fait un `break` ?

Syntaxe

- L'instruction **`break`** permet de sortir du **`switch`**, **`for`**, **`while`** ou **`do`** directement englobant
- A éviter autant que possible dans les boucles !
- Pire encore: un **`break`** permet de quitter n'importe qu'elle structure englobante en l'étiquettant…

Etiquettes



Il vous suffit de savoir que ça existe… juste au cas où…

On `continue`…

Syntaxe

- L'instruction **`continue`** saute directement à la fin du corps de la boucle, puis évalue l'expression booléenne qui contrôle la boucle
- Généralement utilisée pour éviter de traiter certains éléments dans une séquence de valeurs
- Un simple test peut faire l'affaire…

Plan

- Pourquoi Java ?
- Syntaxe du langage
- Classes et objets en Java
- Héritage, interfaces
- Paquetages, exceptions
- Architecture client/serveur (RMI)
- Java et son environnement

Classes et objets en Java

- Pseudo langage → Java
  - définition d'une classe
  - instanciation d'objets
  - constructeurs
  - appels de méthodes
  - surcharge de méthodes
  - membres statiques
  - ramasse-miettes
  - méthodes `main` et `toString`

Classe Compte

- Spécification en pseudo-langage objet:

```
classe Compte: spécification
    attributs:
        crédit, débit: réel;
        nom          : chaîne;

    méthodes:
        initialiser();
        déposer(réel);
        retirer(réel);
        donnerSolde(): réel;
fin Compte
```

Compte
crédit: réel débit : réel nom : chaîne
initialiser() déposer(réel) retirer(réel) donnerSolde():réel

Classe Compte

- Programmation (simple) en Java

```
class Compte {
  float credit, debit;
  String nom;

  Compte();  // ex-méthode initialiser()
  void deposer(float montant);
  void retirer(float montant);
  float donnerSolde();
}
```

Attributs

```
class Compte {
   float credit, debit;
   String nom;

   Compte();  // ex-méthode initialiser()
   void deposer(float montant);
   void retirer(float montant);
   float donnerSolde();
}
```

- Chaque objet de la classe **Compte** a ses propres attributs **credit**, **debit** et **nom**
  ⇒ état unique pour chaque objet

Instanciation

- Pour créer un nouvel objet, on doit:
  - déclarer une variable pouvant contenir une référence

```
Compte mon_compte;
```

    - ceci ne crée pas l'objet
    - la référence est initialisée à **`null`**
  - créer explicitement l'objet avec l'opérateur **`new`**

```
mon_compte = new Compte();
```

    - le runtime Java alloue la mémoire,
    - initialise l'objet (voir plus loin),
    - retourne une référence vers ce nouvel objet

Instanciation

- On peut ensuite utiliser l'objet:

```
mon_compte.credit = 0.0;
mon_compte.debit = 0.0;
mon_compte.nom = "munier";
mon_compte.deposer(10490.0);
float solde = mon_compte.donnerSolde();
```

- S'il n'y a plus de place en mémoire pour créer l'objet, **new** lance une exception **OutOfMemoryException** et le programme s'arrête

Constructeurs

```
class Compte {
  float credit, debit;
  String nom;

  Compte();  // ex-méthode initialiser()
  void deposer(float montant);
  void retirer(float montant);
  float donnerSolde();
}
```

- Les constructeurs ont le même nom que la classe qu'ils initialisent

Constructeurs

- Rôle des constructeurs:
  - dans certains cas, l'initialisation par défaut des attributs (ex: `int` à `0`, réf objet à `null`) ne suffit pas ⇒ construction de l'état initial
  - on est certain que les objets sont initialisés avant qu'un prog puisse les utiliser (via leur réf)
  - on peut paramétrer l'initialisation d'un objet
    - en passant des arguments au constructeur
    - en définissant plusieurs constructeurs pour une même classe

Constructeurs

```
class Compte {
  float credit, debit;
  String nom;

  Compte()   // ex-méthode initialiser()
  {
    credit = débit = 0.0;
    nom = "undefined";
  }

  Compte(String chaine) // un 2ème constructeur
  {
    this();  // invoque le constructeur vide
    nom = chaine;
  }
}
```

Constructeurs

- Le code d'allocation est simplifié:

```
Compte mon_compte = new Compte("munier");
mon_compte.deposer(10490.0);
float solde = mon_compte.donnerSolde();
```

- Le constructeur est invoqué lorsque l'opérateur **new** crée l'objet, mais après que les attributs aient été affectés à leur valeur initiale
- Quand **new** renvoie la référence, l'objet est dans un état « propre »

Constructeurs

- Consignes de programmation:
  - prendre l'habitude de toujours définir **au moins le constructeur vide** (sans arg) pour chaque classe
  - Java impose d'avoir défini le constructeur vide si on veut définir des constructeurs spécialisés (avec args)
  - veiller à ce que chaque constructeur **initialise tous les attributs**

Méthodes

```
class Compte {
  float credit, debit;
  String nom;

  Compte();  // ex-méthode initialiser()
  void deposer(float montant);
  void retirer(float montant);
  float donnerSolde();
}
```

- Rappel: ce sont les (seules) opérations qui permettent de manipuler l'objet

Méthodes

- Méthodes invoquées sur des objets, via des références, et en utilisant l'opérateur .

  référence.méthode(paramètres)

- En Java, chaque méthode a un **nombre fixé de paramètres**
- Une méthode a un type de retour (`void` si pas de résultat)
- Une méthode retourne un résultat via une instruction `return`

Méthodes

```
class Compte {
  float credit, debit;

  void deposer(float montant)
  {
    credit += montant;
  }

  float donnerSolde()
  {
    return (credit-debit);
  }
}
```

Méthodes

- En Java, tous les paramètres sont passés par valeur
- **Attention**: Pour un objet, c'est la **référence** qui est passée par valeur, **pas l'objet** ⇒ la méthode peut alors modifier cet objet via les méthodes de cet objet !

```
void hacker(Compte c)   // réf passée par valeur
{
  c.retirer(100000.0); // mais ceci a pour effet
  c.nom = "Merci..."   // de modifier l'objet c !
}
```

Méthodes

- Par défaut, on peut également accéder aux attributs d'un objet avec l'opérateur .

```
Compte mon_compte = new Compte("munier");
mon_compte.credit = 5000.0;
mon_compte.nom    = "Vil Coyote";
```

- Mais on peut vouloir
  - masquer ces attributs (abstraction de données)
  - contrôler leur accès (read-only, validité,…)
- Les attributs sont déclarés en **`private`**

Méthodes

```
class Compte {
  private float credit, debit;
  private String nom;

  Compte();  // ex-méthode initialiser()
  Compte(String chaine);  // un 2ème constructeur
  void deposer(float montant);
  void retirer(float montant);
  float donnerSolde();
}
```

Ces attributs ne sont accessibles que dans la classe elle-même

- On ne peut accéder à un objet qu'au travers de ses méthodes qui se chargent alors de tous les contrôles

Méthodes

- Visibilité
  - pas de variable globale en POO; on n'utilise que des attributs dans des objets
  - une méthode d'une classe a accès à toutes les variables et à toutes les méthodes de cette classe (même, et surtout, celles déclarées en **`private`**)
  - Rappel: une méthode peut être vue comme une fonction s'exécutant sur l'état d'un objet

Surcharge de méthodes

- La signature d'une méthode est composée:
  - de son nom
  - du nombre et du type de ses paramètres
  - de son type de retour
- Surcharge de méthodes en Java
  - deux méthodes peuvent avoir le même nom si le nombre ou les types de leurs paramètres diffèrent
- Résolution de la surcharge
  - effectuée par le compilateur
  - fonction du nombre et du type des paramètres

Surcharge de méthodes

- Exemple:
  - surcharge de la méthode **Longueur** dans la classe **Rectangle**

```
class Rectangle {
  private float ma_longueur;
  void Longueur(float value) {
    ma_longueur = value;
  }
  float Longueur() {
    return ma_longueur;
  }
```

Membres statiques

- Un **membre statique** est un membre (attribut ou méthode) dont il n'existe qu'**une seule copie par classe**, et non par instance de cette classe
- On parle également de **variables de classe** et de **méthodes de classe**

Attributs statiques

- Pour un attribut statique, il existe **exactement une variable**, indépendamment du nombre d'instances créées (y compris s'il n'y en a pas)
- Les attributs statiques d'une classe sont initialisés:
  - avant que n'importe quel attribut statique ne soit utilisé
  - avant que n'importe qu'elle méthode de la classe soit exécutée

Attributs statiques

```
class Planete {
  long idNum;
  String nom = "<sans-nom>";
  Planete orbite = null;

  // une seule et unique variable commune à
  // toutes les instances de cette classe
  static long nbre = 0;

  Planete() {
    idNum = ++nbre;
  }

  Planete(String n, Planete o) {
    this(); nom = n; orbite = o;
  }
}
```

Attributs statiques

- Ensuite, les objets de la classe **Planete** sont automatiquement numérotés lors de leur instanciation (via le constructeur)

```
// planète n°1
Planete soleil = new Planete("Soleil",null);

// planète n°2
Planete terre = new Planete("Terre",soleil);

// une seule variable nbre
System.out.println(soleil.nbre);  // affiche 2
```

Blocs statiques

- L'initialisation des attributs statiques d'une classe peut nécessiter un traitement plus complexe que l'affectation d'une valeur
- On peut alors définir un **bloc de code d'initialisation statique**
- Ce bloc sera exécuté **au chargement de la classe**

Blocs statiques

```
class Planete {
  long idNum;
  String nom = "<sans-nom>";
  Planete orbite = null;

  // une seule et unique variable commune à
  // toutes les instances de cette classe
  static long nbre;

  // initialisation complexe réalisée au
  // chargement de la classe
  static {
    nbre = 0;
  }
}
```

Méthodes statiques

- Une méthode statique est invoquée sur la classe, et non sur une instance particulière de cette classe
- Elle réalise un travail d'ordre général pour tous les objets de la classe: prochain numéro de série, …
- Pour l'invoquer, on utilise le nom de la classe plutôt qu'une référence sur un objet

Méthodes statiques

- Une méthode statique ne peut accéder qu'aux attributs et méthodes statiques de la classe
- Elle ne dispose pas de la référence `this` puisqu'elle n'opère pas sur un objet spécifique

Méthodes statiques

```
class Planete {
  // une seule et unique variable commune à
  // toutes les instances de cette classe
  static long nbre;

  // initialisation complexe réalisée au
  // chargement de la classe
  static {
    nbre = 0;
  }

  static long nombre() {
    return nbre;
  }
}
```

Méthodes statiques

```
// planète n°1
Planete soleil = new Planete("Soleil",null);

// planète n°2
Planete terre = new Planete("Terre",soleil);

// planète n°3
Planete lune = new Planete("Lune",terre);

// planète n°4
Planete mars = new Planete("Mars",soleil);

// Affiche le nombre de planètes créées (4)
System.out.println(Planete.nombre());
```

Membres statiques

- Conclusion:
  - `static` signifie défini sur la classe et non sur une instance particulière de cette classe
  - **attribut statique**: une seule variable pour toutes les instances de cette classe
  - **bloc statique**: exécuté lorsque la classe est chargée dans la machine virtuelle Java
  - **méthode statique**: invoquée sur la classe, et non sur une instance ⇒ restrictions de visibilité

Ramasse-miettes

- Java dispose d'un ramasse-miettes (ou *garbage collector*) qui s'occupe de supprimer (`free` en langage C) les objets qui ne sont plus utilisés
- Un objet est inutilisé lorsqu'il n'est plus référencé, i.e. qu'il n'existe plus de référence vers cet objet
- Ex: on affecte `null` aux références sur cet objet

Ramasse-miettes

- Plus de problème de référence invalide (référence sur un objet détruit)
- Vous ne savez pas quand est déclenché le ramasse-miettes:
  - quand il n'y a plus assez de mémoire
  - quand la programme a du temps libre
  - etc…
- Ce n'est plus votre problème: « ça marche »

Méthode `finalize`

- Une classe peut implémenter une méthode **`finalize`**

```
protected void finalize() throws Throwable
{
   super.finalize();
   // votre traitement
}
```

- Cette méthode est invoquée quand un objet doit être détruit
  - par le ramasse-miettes
  - quand la machine virtuelle Java se termine

Méthode `finalize`

- Utile dans le cas où vos objets utilisent des ressources externes (ex: des fichiers)
- Via la méthode **`finalize`**, vous veillerez à ce que tous les fichiers ouverts par cet objet aient bien été fermés
- Pensez à invoquer **`super.finalize()`** pour être certain que les traitements définis par les classes parentes soient également exécutés

Méthode `main`

- Pour exécuter une application Java (commande **`java`**), vous devez indiquer le nom d'une classe qui dirige l'application
- La machine virtuelle Java localise puis exécute la méthode **`main`** de cette classe
  - elle doit être publique et statique
  - son type de retour doit être **`void`**
  - elle n'accepte qu'un seul arg de type **`String[]`**

Méthode main

- Exemple d'application qui affiche ses paramètres:

```
class Echo {
  public static void main(String[] args)
  {
    System.out.print("Args: ");
    for (int i=0;i<args.length;i++)
      System.out.print(args[i] + " ");
    System.out.println();
  }
}
```

Méthode `main`

- En Java, une classe doit être définie dans un fichier de même nom et comportant l'extension `.java` (ici `Echo.java`)
- Commande pour compiler cette classe et générer le fichier `Echo.class` (bytecode):

```
javac Echo.java
```

- Commande pour exécuter cette application:

```
java Echo Vive le GTR
```

- Affichage: `Args: Vive le GTR`

Méthode `main`

- Une application peut comporter plusieurs méthodes **`main`** (mais une seule par classe)
- Seul le **`main`** de la classe passée en paramètre de la commande **`java`** sera exécuté
- Avoir un **`main`** dans chaque classe permet de tester chaque classe de façon plus ou moins indépendante

Méthode `toString`

- Si un objet dispose d'une méthode `toString` ne prenant pas d'argument, celle-ci sera automatiquement invoquée à chaque fois qu'une expression `+` ou `+=` reçoit cet objet alors qu'elle attend une chaîne de caractères

Méthode toString

```
class Compte {
  // ...
  public String toString() {
    return nom + " (" + donnerSolde() + " FF)" ;
  }
}
```

- La méthode suivante affiche tous les comptes du tableau tab

```
void afficherComptes(Compte[] tab) {
  for (int i=0;i<tab.length;i++)
    System.out.println(i + ": " + tab[i]);
}
```

Plan

- Pourquoi Java ?
- Syntaxe du langage
- Classes et objets en Java
- Héritage, interfaces
- Paquetages, exceptions
- Architecture client/serveur (RMI)
- Java et son environnement

Héritage

- Pseudo langage → Java
  - extension d'une classe
  - redéfinition de méthodes et d'attributs
  - mot-clé **`super`**
  - constructeurs des sous-classes
  - méthodes et classes finales
  - méthodes et classes abstraites
  - comparaison et duplication d'objets
  - notion d'interface
    - spécification pure
    - héritage multiple

Extension d'une classe

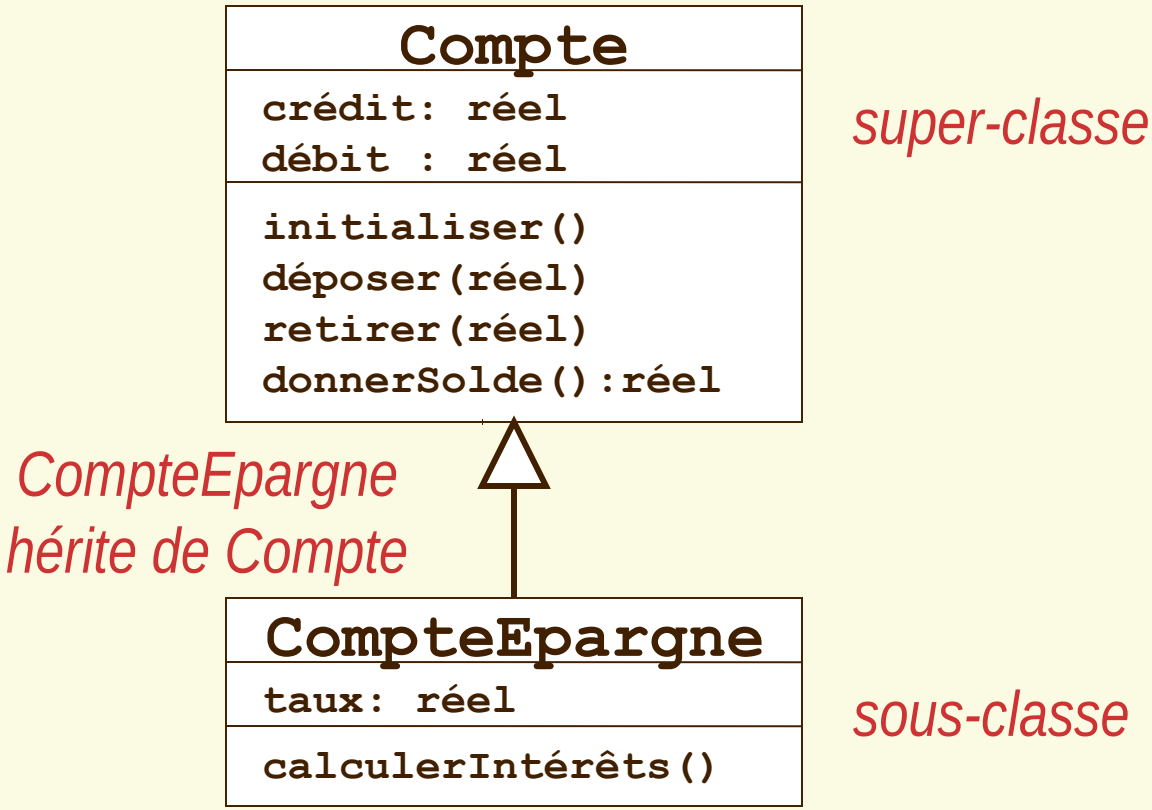
- Comme nous l'avons vu en POO, l'héritage permet d'étendre une classe existante sans la modifier
  - par enrichissement d'attributs et de méthodes
  - par redéfinition d'attributs et de méthodes
- A noter qu'en Java, toutes les classes étendent implicitement la classe `Object`

Extension d'une classe

- **Attention**: une sous-classe doit être un cas particulier de sa super-classe
  - soit une classe `Point` définissant deux coordonnées `X` et `Y`
  - un pixel est un point particulier, comportant en plus une couleur ⇒ la classe `Pixel` **hérite** de la classe `Point`
  - un cercle comporte un point (son centre) mais n'est pas un point particulier ⇒ la classe `Cercle` **utilise** la classe `Point`

Comptes bancaires

- Graphe de classes:



Comptes bancaires

- Classe **Compte** en Java

```
class Compte {
  private float credit, debit;
  private String nom;

  Compte() {…};                    // initialiser()
  Compte(String chaine) {…};       // 2ème constructeur
  void deposer(float montant) {…};
  void retirer(float montant) {…};
  float donnerSolde() {…};
  String nom() {…};
  String toString() {…};
}
```

Comptes bancaires

- En Java, on utilise le mot-clé **extends** pour indiquer que la classe **CompteEpargne** hérite de la classe **Compte**

```
class CompteEpargne extends Compte {
  private float taux;

  CompteEpargne() {…};
  CompteEpargne(String chaine, float t) {…};
  void calculerInterets() {…};
  float taux() {…};
  String toString() {…};
}
```

Héritage en Java

- Le mot-clé **`extends`** ne permet que l'héritage simple (une classe ne possède qu'une seule super-classe)
- Java supporte une certaine forme d'héritage multiple via la **notion d'interface** (vue plus loin)

Méthodes héritées

- On peut surcharger des méthodes héritées (même nom, mais signatures différentes)
- On peut redéfinir une méthode héritée (même signature)
  - NB: les **méthodes statiques** ne peuvent pas être redéfinies
- C'est le type réel de l'objet (celui du **new**) qui détermine l'implémentation à utiliser (liaison dynamique)

Attributs hérités

- Un attribut ne peut être ni redéfini, ni masqué
- Si, dans une sous-classe, on déclare un nouvel attribut (ex: `str`) avec le même nom qu'un attribut de la super-classe, ce dernier continue d'exister
  - `str` ⇒ attribut de la sous-classe
  - `super.str` ⇒ attribut de la super-classe

Attributs hérités

- **Attention**: c'est le type déclaré de la référence qui décide de l'attribut accédé

```
class Essai {
  String str = "One";
}
class EssaiEtendu extends Essai {
  String str = "Two";
}
void test() {
  EssaiEtendu ext = new EssaiEtendu();
  Essai       sup = ext;
  System.out.println(sup.str + "/" + ext.str);
}
```

même objet en mémoire

One *Two*

Mot-clé `super`

- Peut être utilisé dans toutes les méthodes non statiques
- **`super`** se comporte comme une référence sur l'objet courant, mais en supposant que celui-ci est une instance de sa super-classe
- Une invocation **`super.méthode`** utilise toujours l'implémentation trouvée dans la super-classe, et non celle éventuellement redéfinie dans la sous-classe

Restrictions d'accès

- Comme nous l'avons déjà signalé, on peut restreindre l'accès à un attribut ou à une méthode:
  - **`public`** ⇒ accès pour toutes les classes
  - **`protected`** ⇒ uniquement classe et sous-classes
  - **`private`** ⇒ accès pour la classe uniquement

Constructeurs

- Le constructeur d'une sous-classe doit invoquer un des constructeurs de la super-classe pour initialiser les attributs hérités
  - invocation par `super()` ou `super(args)`
  - `super()` doit être la **première instruction** du nouveau constructeur
  - si on n'invoque pas explicitement un constructeur de la super-classe, c'est son **constructeur vide** qui est invoqué (et doit donc exister)

Constructeurs

- Ordre d'initialisation lors de l'invocation d'un constructeur sur une sous-classe:
  1. invocation du constructeur de la super-classe
  2. initialisation des attributs déclarés dans la sous-classe avec leurs valeurs par défaut
  3. exécution du corps du constructeur
- **Attention**: Les constructeurs d'une classe ne sont pas hérités par ses sous-classes !

Constructeurs

```
class Compte {
  Compte()
  {
    credit = debit = 0.0;
  }
}

class CompteEpargne extends Compte {
  CompteEpargne(float t)
  {
    super();  // Constructeur vide de Compte
    taux = t;
  }
}
```

Constructeurs

```
class Fichier {
  Fichier(String name,String owner,String group,
          int droits)
  {…}
}

class Repertoire extends Fichier {
  Fichier liste[];

  Repertoire(String name,String owner,
             String group,int droits)
  {
    super(name,owner,group,droits);
    liste = new Fichier[30];
  }
}
```

Méthodes/Classes finales

- Une **méthode finale** (déclarée avec `final`) ne peut plus être redéfinie par aucune des sous-classes
- C'est la **version finale** de cette méthode
- Une **classe finale** ne peut pas être sous-classée ⇒ toutes ses méthodes sont donc implicitement finales
- But: **sécurité**, car le comportement de la méthode/classe ne changera plus

Méthodes/Classes finales

```
class GTRlogin {
  private String login;
  private String password;

  GTRlogin(String l, String p) {
    login    = l;
    password = p;
  }

  final boolean authentifier() {
    // même si on sous-classe GTRlogin, on ne
    // pourra pas redéfinir cette méthode ;-)
  }
}
```

Méthodes/Classes abstraites

- Une méthode/classe est déclarée abstraite avec le mot-clé **`abstract`**
- Une classe comportant **au moins** une méthode abstraite donc obligatoirement être elle-même déclarée abstraite
  - NB: si une classe hérite d'une méthode abstraite et qu'elle ne la redéfinit pas, elle est elle-même une classe abstraite

Méthodes/Classes abstraites

- On ne peut pas instancier d'objet à partir d'une classe abstraite
  - une classe abstraite n'étant pas complètement implémentée, une instance de cette classe possèderait donc des méthodes non implémentées !
- **Conséquence**: une classe abstraite ne devrait pas avoir besoin de constructeur
  - mais il est conseillé de lui définir au moins le constructeur vide (cf. cons. des ss-classes)

Méthodes/Classes abstraites

```
abstract class ObjetGraphique {
  abstract afficher();
  abstract effacer();
  abstract translater(Vecteur2D v);

  void deplacer(Vecteur2D v) {
    // méthode générique (cf. cours POO)
    this.effacer();
    this.translater(v);
    this.afficher();
  }
}
```

Méthodes/Classes abstraites

- Rôle d'une classe abstraite
  - définir ce qui est commun aux sous-classes
  - indiquer que l'on peut invoquer telle méthode sur une référence déclarée de cette classe, même si la méthode (abstraite) n'est effectivement implémentée que dans les sous-classes
  - là où on attend un objet de la classe **`A`**, on pourra passer un objet d'une classe **`B`** si **`B`** est une sous-classe de **`A`** (**`B`** est un cas particulier de **`A`** et dispose, au moins, des mêmes méthodes)

Comparaison/Duplication

- Toutes les classes héritent (implicitement) de la classe `Object`, et en particulier des méthodes suivantes:
  - `public boolean equals(Object obj);`
    `// test d'égalité (profonde)`
  - `public Object clone()`
    `throws CloneNotSupportedException;`
    `// copie (profonde) d'un objet`
  - `public final Class getClass();`
  - `protected void finalize()`
    `throws Throwable;`

Comparaison/Duplication

- Quand on compare deux objets:
  - vérifier qu'ils ont bien la même classe réelle (i.e. celle du **new**)
  - pour les types primitifs, il suffit de comparer les valeurs des attributs
  - pour les références sur des objets, ne pas comparer les références mais les objets référencés (récursivement) !
- Idem pour la copie profonde: ne pas copier les références, mais les objets eux-mêmes

Interfaces

- Dans une classe Java, la spécification et l'implémentation sont mélangées
- Interface Java = **spécification pure**
- Implicitement, toutes les méthodes d'une interface sont **abstraites** (car il n'y a pas d'implémentation)
- Les attributs d'une interface sont, quant à eux, implicitement `static` et `final`: ce sont des **constantes**

Interfaces

- Une interface peut étendre (**extends**) une autre interface

```
interface ICompte {
   void deposer(float montant);
   void retirer(float montant);
   float donnerSolde();
   String nom();
   String toString();
}
interface ICompteEpargne extends ICompte {
   void calculerInterets();
   float taux();
}
```

Interfaces

- Une classe implémente une interface (mot-clé **`implements`**)
- Si une classe n'implémente pas toutes les méthodes d'une interface, cette classe doit être déclarée abstraite
- Une interface définit le «**contrat**» que doit respecter une classe

Interfaces

- Exemple

```
class Compte implements ICompte {
  float credit,debit;
  // etc...
}

class CompteEpargne
    extends Compte
    implements ICompteEpargne
{
  float taux;
  // etc...
}
```

Interfaces

- Plusieurs classes peuvent implémenter une même interface
  - classe **`CompteFiable`** codée par le prof.
  - classe **`CompteBof`** codée par les étudiants
- Pour autant, elles n'ont pas forcément de lien de parenté
- Par contre, elles fournissent les mêmes services (même «contrat»)

Interfaces

- Si le type d'un paramètre est une interface `I`, cela signifie qu'il pourra contenir une référence sur une instance de n'importe quelle classe implémentant cette interface
  - la classe implémente directement l'interface `I`
  - elle hérite d'une classe qui implémente `I`
  - elle implémente une interface qui étend `I`
  - elle hérite d'une classe qui implémente une interface qui étend `I`

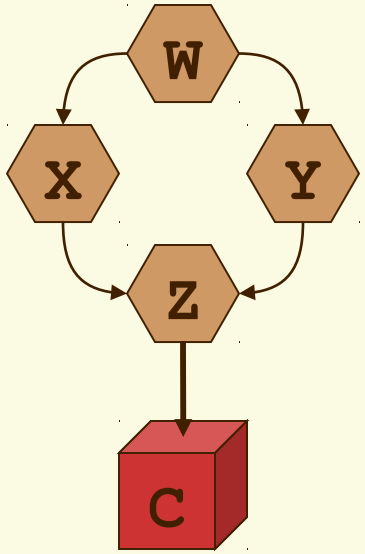
Interfaces

- Le concept d'interface nous apporte l'héritage multiple sur les classes
  - une interface peut étendre (**`extends`**) plusieurs interfaces
  - une classe peut implémenter (**`implements`**) plusieurs interfaces

Interfaces

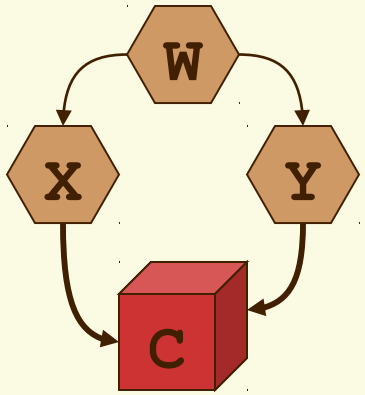
- Héritage multiple

```
interface W {…}
interface X extends W {…}
interface Y extends W {…}
interface Z extends X,Y {…}
class C implements Z {…}
```



- ou encore

```
interface W {…}
interface X extends W {…}
interface Y extends W {…}
class C implements X,Y {…}
```



Interfaces

- Pourquoi utiliser des interfaces ?
  - besoin d'héritage multiple
  - dissocier spécification et implémentation
    - plusieurs classes peuvent implémenter la même interface sans avoir aucun lien de parenté entre elles
    - envoyer la liste des services auxquels une instance d'une classe implémentant cette interface peut répondre (cf. objets répartis sur un réseau)
  - propreté de la conception de l'application (et tout ce qui va avec…)

Interfaces

- Exercice
  - Le cahier des charges nous demande de définir puis d'implémenter une classe DNS dont les instances devront être capables de répondre aux requêtes suivantes:
    - `enregistrer(String,String)` : mémorise une nouvelle association nom/adresse
    - `rechercherNom(String)` : retrouve l'adresse IP associée au nom passé en paramètre
    - `rechercherAdr(String)` : retrouve le nom associé à l'adresse IP fournie en argument

Plan

- Pourquoi Java ?
- Syntaxe du langage
- Classes et objets en Java
- Héritage, interfaces
- Paquetages, exceptions
- Architecture client/serveur (RMI)
- Java et son environnement

Paquetages Java

- En POO, une classe correspond à une entité élémentaire ⇒ dans une application, on se retrouve vite avec un grand nombre de classes
- Idée: organiser ces classes en définissant des modules appelés paquetages en conception OO
- De plus, en Java: 1 classe → 1 fichier

Paquetages Java

Package

- En Java, un paquetage permet de regrouper des **interfaces**, des **classes** et des **sous-paquetages** connexes
- Les classes/interfaces d'un paquetage ne peuvent pas entrer en conflit avec des classes/interfaces de même nom dans un autre paquetage
  - pas de confusion possible entre `GTR.String` et `java.lang.String`

Définition

- Chaque classe/interface appartenant au paquetage **GTR** affirme sa résidence avec une clause **package**
  - **package GTR;**
- Une clause **package** doit apparaître en premier dans le fichier, avant toute déclaration de classe/interface
- Il ne peut y avoir qu'une seule clause **package** par fichier

Définition

- Pour indiquer qu'une classe/interface appartient paquetage **Etudiants** qui est lui-même un sous-paquetage de **GTR**
  - `package GTR.Etudiants;`
- On construit ainsi tous les paquetages et sous-paquetages (les noms peuvent devenir très longs)

Utilisation

- Pour utiliser des classes/interfaces se trouvant dans un paquetage
  - on peut préfixer son nom par celui du paquetage

```
E=new GTR.Etudiants.Etudiant("Picavet");
```

  - on peut importer tout ou partie du package

```
import GTR.Etudiants.*;  // on importe tout
// puis, plus tard...
E=new Etudiant("Picavet");
```

- Une clause **import** se trouve en début de fichier (après **package**, mais avant quoi que ce soit d'autre)

Utilisation

- Un paquetage s'importe lui-même implicitement
  - tout ce qui est défini dans un paquetage est accessible sans préfixe aux autres membres de ce paquetage
- Importer un paquetage ne permet de nommer que les classes/interfaces de ce paquetage, pas celles de ses sous-paquetages

Restrictions d'accès

Package

- Visibilité des interfaces, classes, méthodes et attributs:
  - **`public`** ⇒ accès pour toutes les classes (quel que soit leur paquetage)
  - **`protected`** ⇒ uniquement classe et sous-classes
  - **`private`** ⇒ accès pour la classe uniquement
  - *<rien>* ⇒ accès réservé aux membres du même paquetage

Recommandations

- Dans un paquetage, on regroupe des interfaces, classes et paquetages qui ont un point commun (notion de module)
- En général, on fait correspondre la hiérarchie des paquetages à une hiérarchie de répertoires sur le système de fichiers
  - 1 paquetage → 1 répertoire
- Le découpage en paquetages doit également permettre une meilleure réutilisabilité

Conclusion

- Les paquetages permettent de découper une application en modules
- Ils peuvent être installés n'importe où dans le système de fichier
  - il suffit d'ajouter la racine du paquetage à la variable d'environnement **CLASSPATH** pour que Java sache le retrouver

Gestion des erreurs

- En programmation classique
  - on utilise des drapeaux (variables globales)
  - on provoque des effets de bord avec des champs additionnels ou des codes de retour

```
int sid = socket(AF_INET,SOCK_DGRAM,0);
if (bind(sid,&localaddr,sizeof(localaddr))<0)
  {
    printf("[serv] PB bind\n");
    exit(1);
  }
else
  printf("[serv] bind Ok. Reception...\n");
```

Si plusieurs erreurs possibles, il faut faire un switch...

Exceptions

- Les exceptions sont un moyen pratique de faire du contrôle d'erreurs sans alourdir le code
- Une exception permet d'interrompre l'exécution normale d'un programme pour signaler une erreur
- L'exception est alors remontée aux méthodes appelantes qui peuvent la capturer

Principe

- Fonctionnement des exceptions:
  - quand une exception est « **levée** », elle interromps l'exécution du programme et « **remonte** » dans la pile des appels de méthodes
  - une méthode peut « **capturer** » cette exception pour effectuer un certain traitement; le programme peut ensuite reprendre son exécution
  - si aucune méthode ne capture l'exception lors de sa remontée, le programme s'arrête et la pile d'appels est affichée

Exemple de trace

Erreur

```
Initialisation de l'écran...
java.lang.InternalError: Can't connect to X11 window server
    using 'marsan:0.0=' as the value of the DISPLAY variable.
        at sun.awt.motif.MToolkit.<init>(MToolkit.java:76)
        at java.awt.Toolkit.getDefaultToolkit(Toolkit.java:394)
        at java.awt.Window.getToolkit(Window.java:242)
        at java.awt.Frame.addNotify(Frame.java:203)
        at java.awt.Window.show(Window.java:157)
        at Ecran.<init>(Ecran.java:30)
        at Test2.main(Test2.java:12)
```

Pile des appels de méthodes

Exceptions en Java

- En Java, la plupart des exceptions sont des exceptions **contrôlées**
  - les méthodes doivent indiquer, dans leur signature, quelles sont les exceptions qu'elles sont susceptibles de lever
  - le compilateur vérifie que toute méthode invoquant une méthode susceptible de lever une telle exception
    - soit la capture
    - soit est elle-même susceptible de la propager

Définir une exception

- Les exceptions contrôlées sont dérivées de la classe **java.lang.Exception**

```
import java.lang.*;

public class NoSuchFileException extends Exception {
  public String name;
  public Repertoire dir;

  NoSuchFileException(String n, Repertoire d) {
    super("File \"" + n + "\" not found in " +
          "directory \"" + d.filename() + "\");
    name = n;
    dir  = d;
  }
}
```

Définir une exception

- Une exception étant un **objet** comme un autre, elle peut contenir un certain nombre d'informations (**attributs**) sur l'erreur qui s'est produite
- Elle peut également fournir des **méthodes** supplémentaires
- Le **type d'une exception** est une partie importante des données, car les exceptions seront capturées en fonction de leur type

Lever une exception

- Une méthode déclare pouvoir lever une exception avec la clause **throws**

```
public Fichier getFile(String name)
    throws NoSuchFileException
{…}
```

- Une méthode ne peut lever que les exceptions déclarées dans la clause **throws**
- Une méthode lève effectivement une exception avec l'instruction **throw**

```
throw new NoSuchFileException(name,this);
```

Lever une exception


```
public class Repertoire extends Fichier {
  public Fichier getFile(String name)
      throws NoSuchFileException
  {
    Fichier res=null;
    for (int i=0;(i<nbFiles)&&(res==null);i++)
      {
        if (liste[i].filename().equals(name))
          res = liste[i];
      }
    if (res==null)
      throw new NoSuchFileException(name,this);
    return res;
  }
}
```

Capturer une exception

- Si une méthode **M1** désire invoquer une méthode **M2** susceptible de lever une exception (clause **throws**)
  - soit **M1** englobe ces appels à **M2** dans des blocs **try…catch** afin de pouvoir capturer ces exceptions
  - soit la méthode **M1** doit explicitement déclarer qu'elle « **fera suivre** » une telle exception (cf. clause **throws** de **M1**)
- Ceci est imposé par le compilateur !

Capturer une exception

- Syntaxe de base d'un bloc **try**

```
try
    // bloc d'instructions susceptibles
    // de lever des exceptions
catch (exception_type identifier)
    // bloc de traitement
catch (exception_type identifier)
    // bloc de traitement
...
finally
    // bloc d'instructions
```

Capturer une exception

- Le corps de l'instruction `try` est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée
- Si une exception est levée, les clauses `catch` sont examinées **dans l'ordre** pour en trouver une qui capture cette classe d'exceptions ou une de ses super-classes
- Une instruction `try` peut avoir un nombre quelconque de clauses `catch` (voire aucune)

Capturer une exception

- Si aucune clause catch appropriée n'est trouvée, l'exception est propagée à l'extérieur de l'instruction `try`
  - vers une autre instruction `try` englobante qui pourrait la traiter
  - en dehors de la méthode si celle-ci a déclaré ce type d'exception dans sa clause `throws`

Capturer une exception

- Si une clause **`finally`** est présente dans un **`try`**, son code est exécuté une fois que le traitement du **`try`** a été effectué, qu'elle que soit la manière dont il s'est terminé
  - normalement
  - par une exception (capturée ou non)
  - par une instruction de contrôle comme **`return`** ou **`break`**

Capturer une exception

```
public void imprimerFichier(String name,
                            Repertoire dir,
                            Printer prn)
{
  prn.loadPage();
  try {
    Fichier f = dir.getFile(name);
    String  s = f.getContent();
    prn.print(s);
  }
  catch (NoSuchFileException e)
    prn.print("File not found !");
  finally
    prn.ejectPage();
}
```

bloc d'instructions susceptibles de lever une exception

capture et traitement des exceptions de la classe NoSuchFile...

ce bloc est toujours exécuté

Quand les utiliser ?

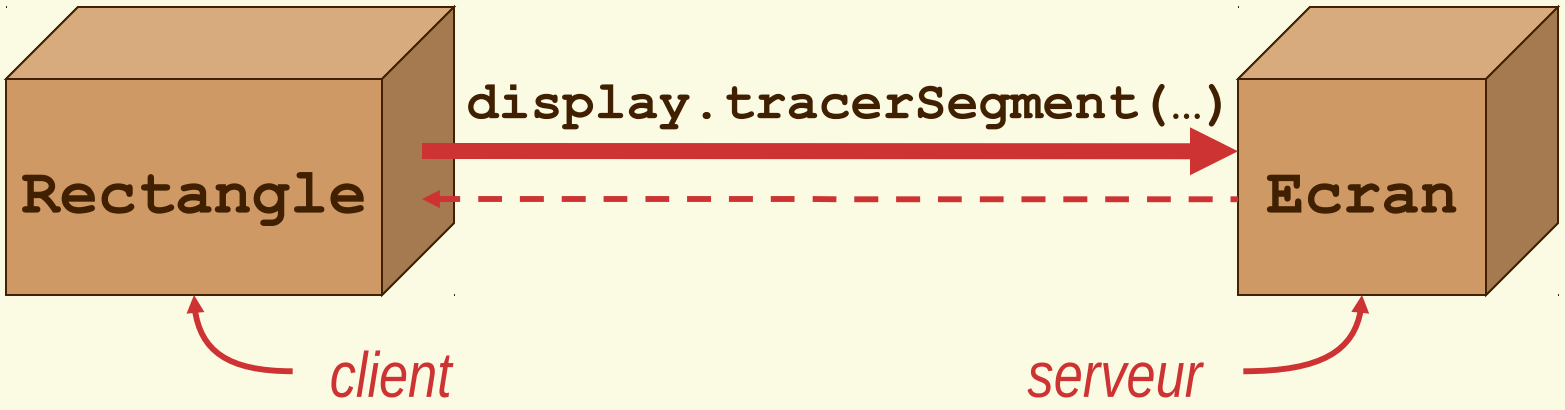
- Exception = condition d'erreur non prévue
- Dans les cas prévisibles, mieux vaut utiliser des tests et des conditions d'arrêt bien programmés
  - quand on parcourt tous les éléments d'une liste, atteindre la fin de la liste est un événement prévisible ⇒ **condition booléenne**
  - par contre, la rupture d'un flux d'entrée en pleine lecture est imprévisible ⇒ **exception**

Plan

- Pourquoi Java ?
- Syntaxe du langage
- Classes et objets en Java
- Héritage, interfaces
- Paquetages, exceptions
- Architecture client/serveur (RMI)
- Java et son environnement

Client/Serveur

- Architecture client/serveur:
  - le **client** est l'objet qui demande l'exécution d'un service (invocation d'une méthode)
  - le **serveur** est l'objet qui fournit le service (exécution de la méthode)



Objets distants

- Mais on parle plus généralement d'architecture client/serveur lorsque les objets clients et les objets serveurs s'exécutent sur des machines différentes
- Pour cela, il nous faut
  - connaître les objets distants
  - avoir la liste des services qu'ils proposent
  - invoquer ces services et passer des paramètres
  - récupérer les résultats
  - capturer des exceptions levées à distance

Objets distants

- Solutions actuelles:
  - programmer directement avec des **sockets**
    - vous devez tout faire !!!
  - sur Sun, utiliser les **RPC** (Remote Procedure Calls) et le mécanisme **XDR** (eXternal Data Representation)
    - ne gère que l'invocation de méthode et le passage de paramètres
  - Java propose les **RMI** (Remote Method Invocation)
    - fait (presque) tout, hétérogénéité des systèmes,...

Objets distants

- Problème des RMI
  - mécanisme spécifique à Java, i.e. pas question d'avoir un client en Java et un serveur en C++ (ou alors on l'encapsule dans un objet Java)
- Solution plus générale: **CORBA**
  - Common Object Request Broker Architecture
  - interconnexion d'objets distants, s'exécutant sur des plates-formes différentes, et implémentés dans des langages différents
  - archi d'égal-à-égal, notion de service,...

Objets distants

- Tableau comparatif des solutions

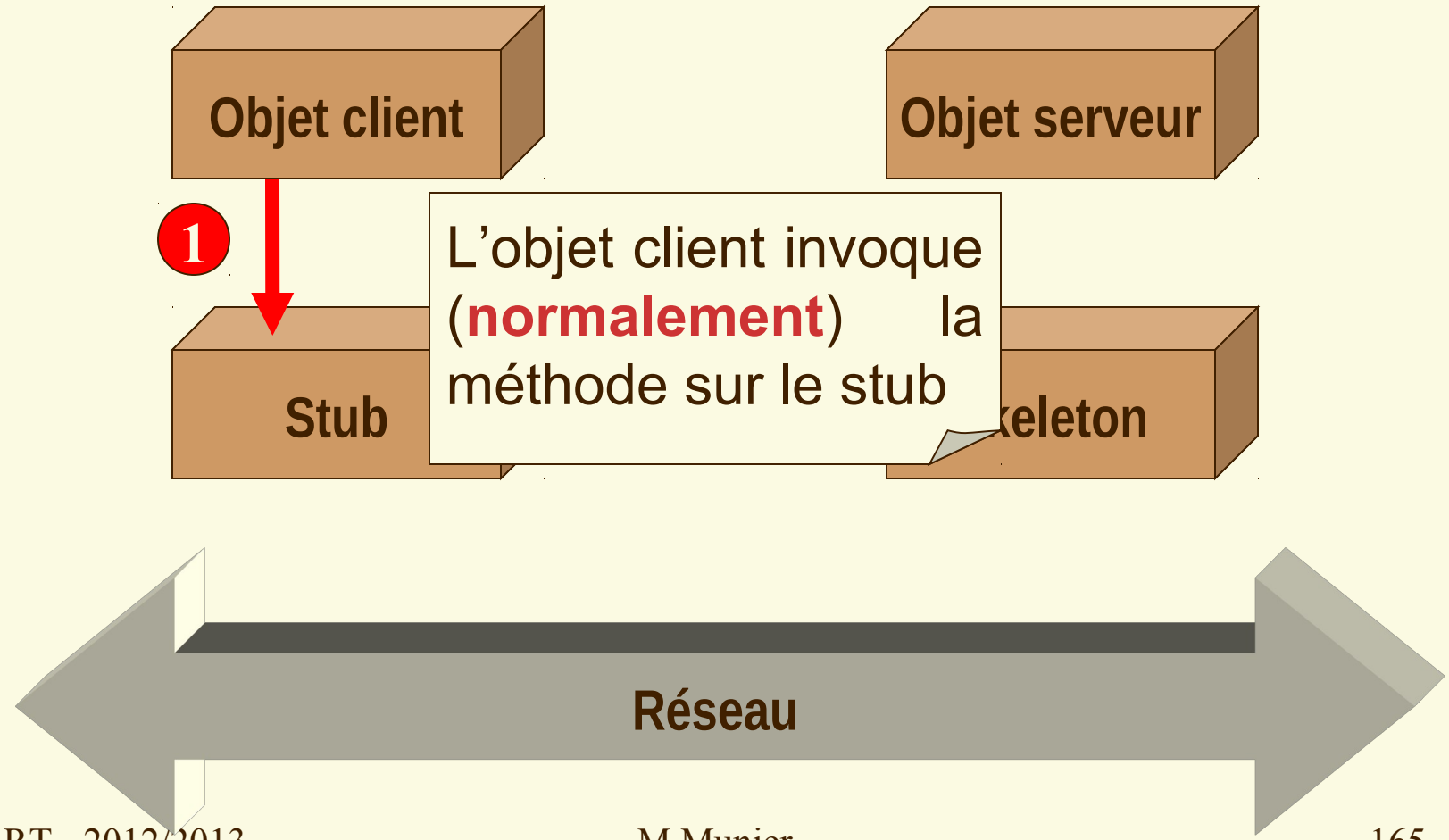
	réseau	invocation de méthodes	plates-formes différentes	annuaire	langages différents
sockets	oui				
RPC	oui	oui			
RPC+XDR	oui	oui	oui		
RMI	oui	oui	oui	oui	
CORBA	oui	oui	oui	oui	oui
WS	oui	oui	oui (XML)	oui	oui

- Comme on ne fait que du Java, les RMI nous suffisent

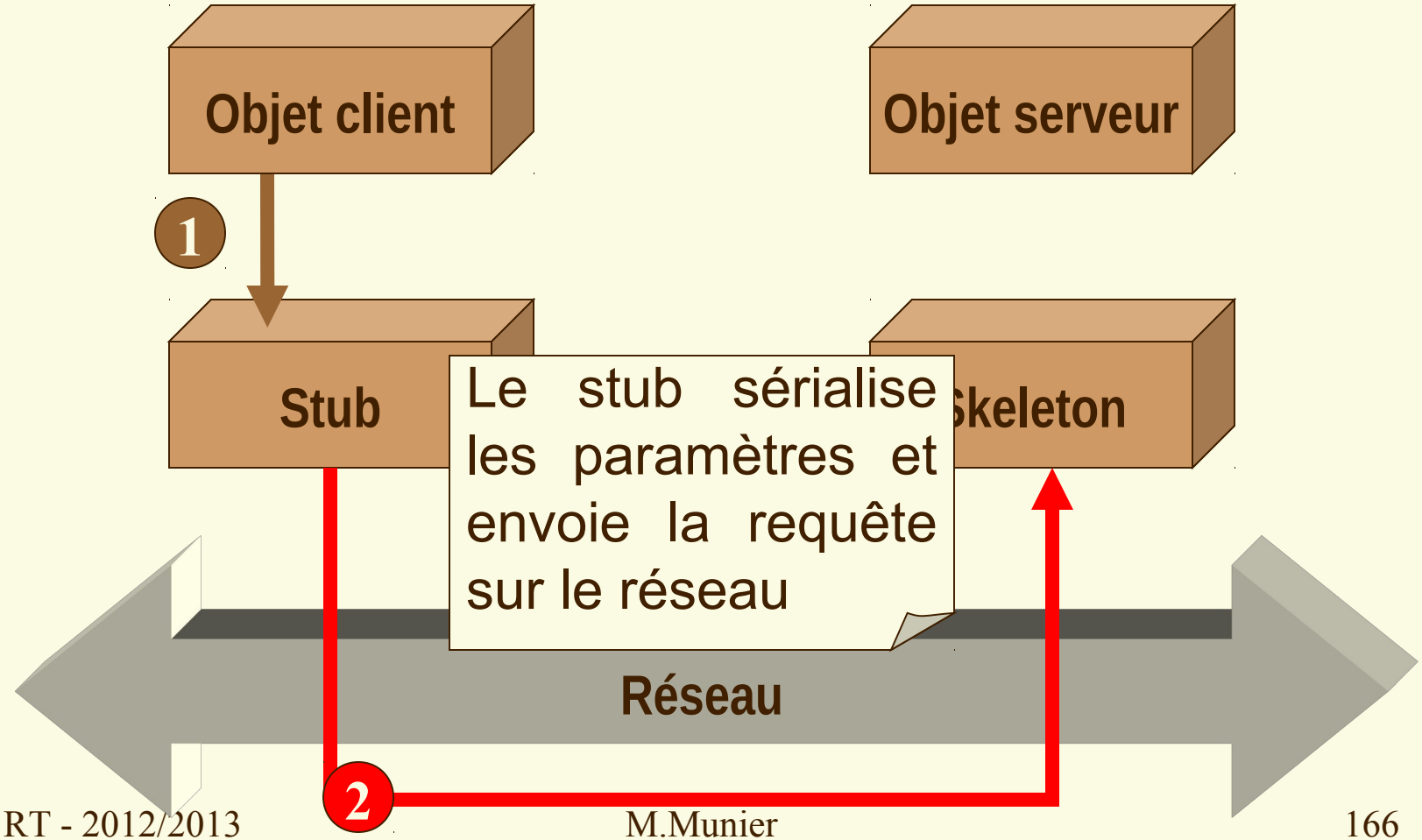
Principe

- L'objet client ne communique pas directement avec l'objet serveur, mais avec un proxy local: le **stub**
  - sérialisation des paramètres
  - construction de la requête réseau
  - attente du résultat et/ou des exceptions
- De son côté, l'objet serveur est relié au réseau via un **skeleton**
  - désérialisation des paramètres
  - invocation de la méthode sur le serveur
  - sérialisation du résultat et/ou des exceptions

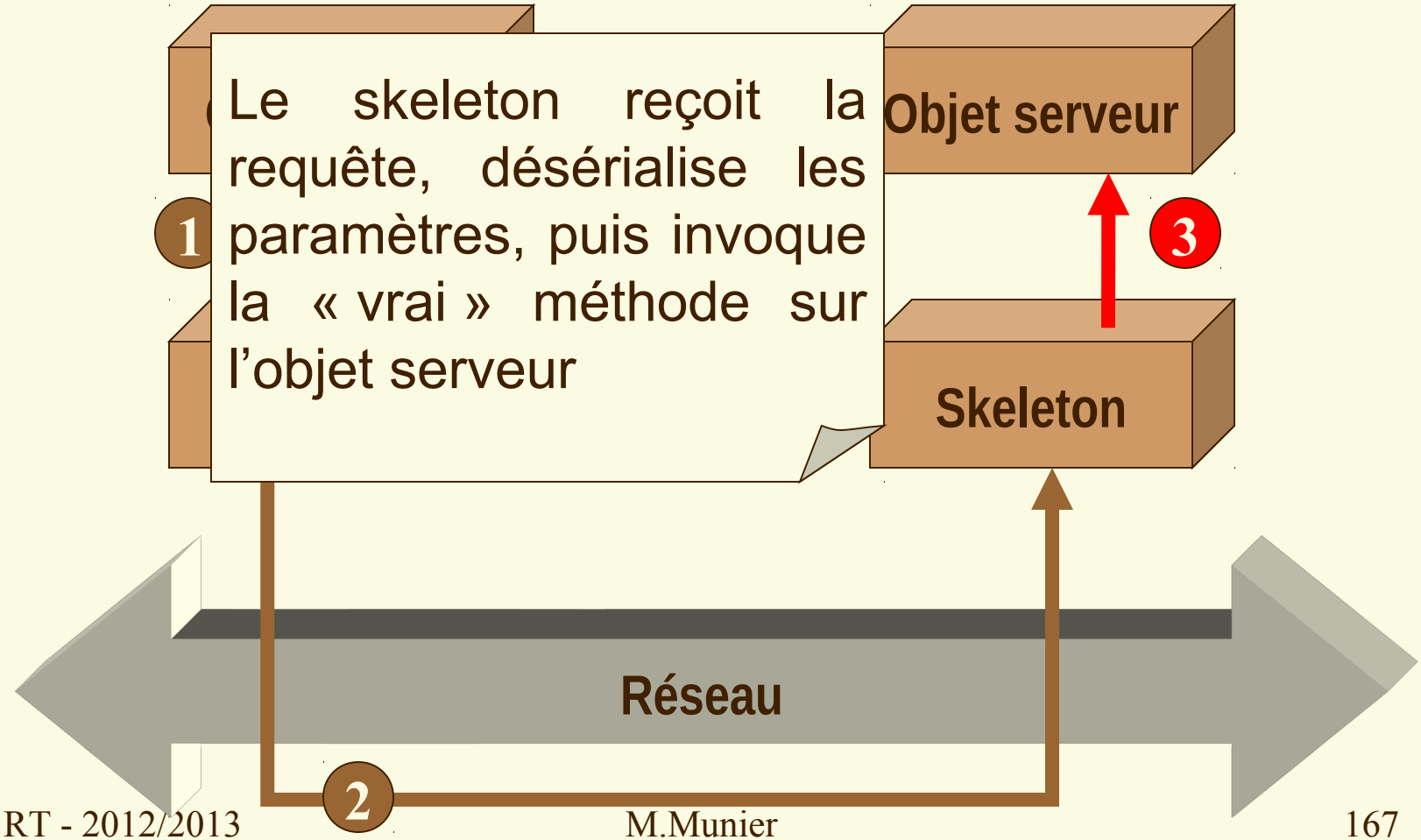
Architecture



Architecture

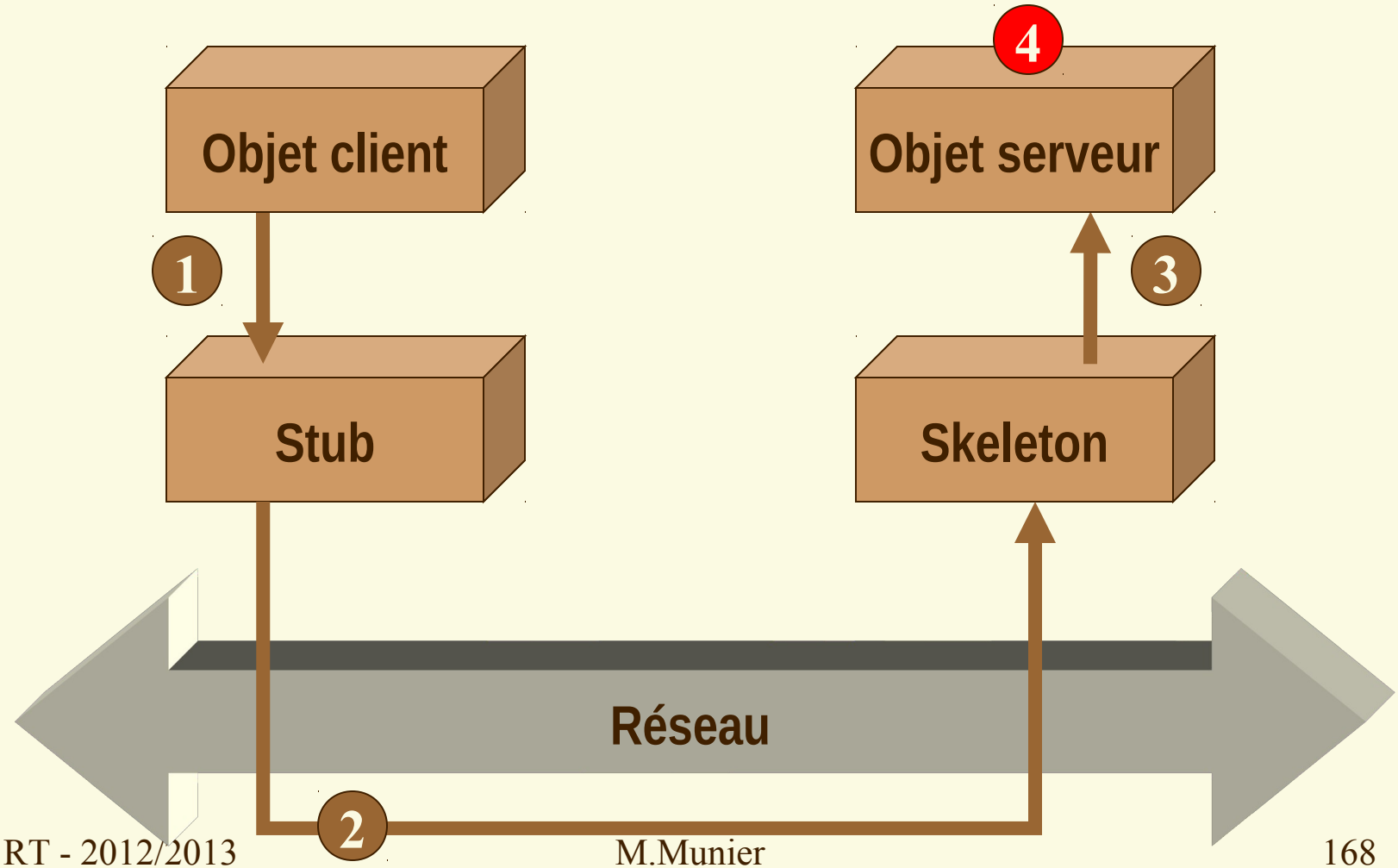


Architecture

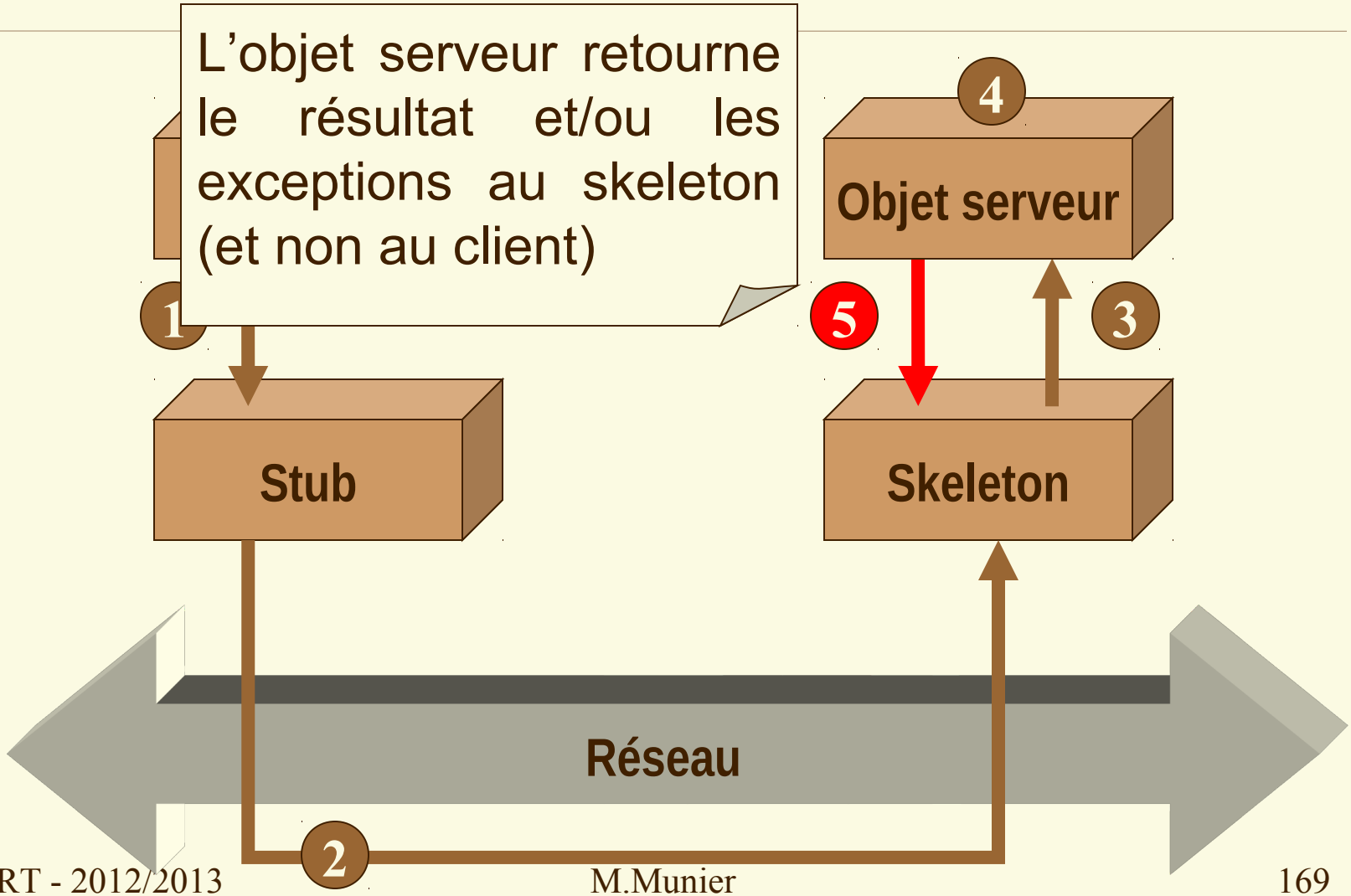


Architecture

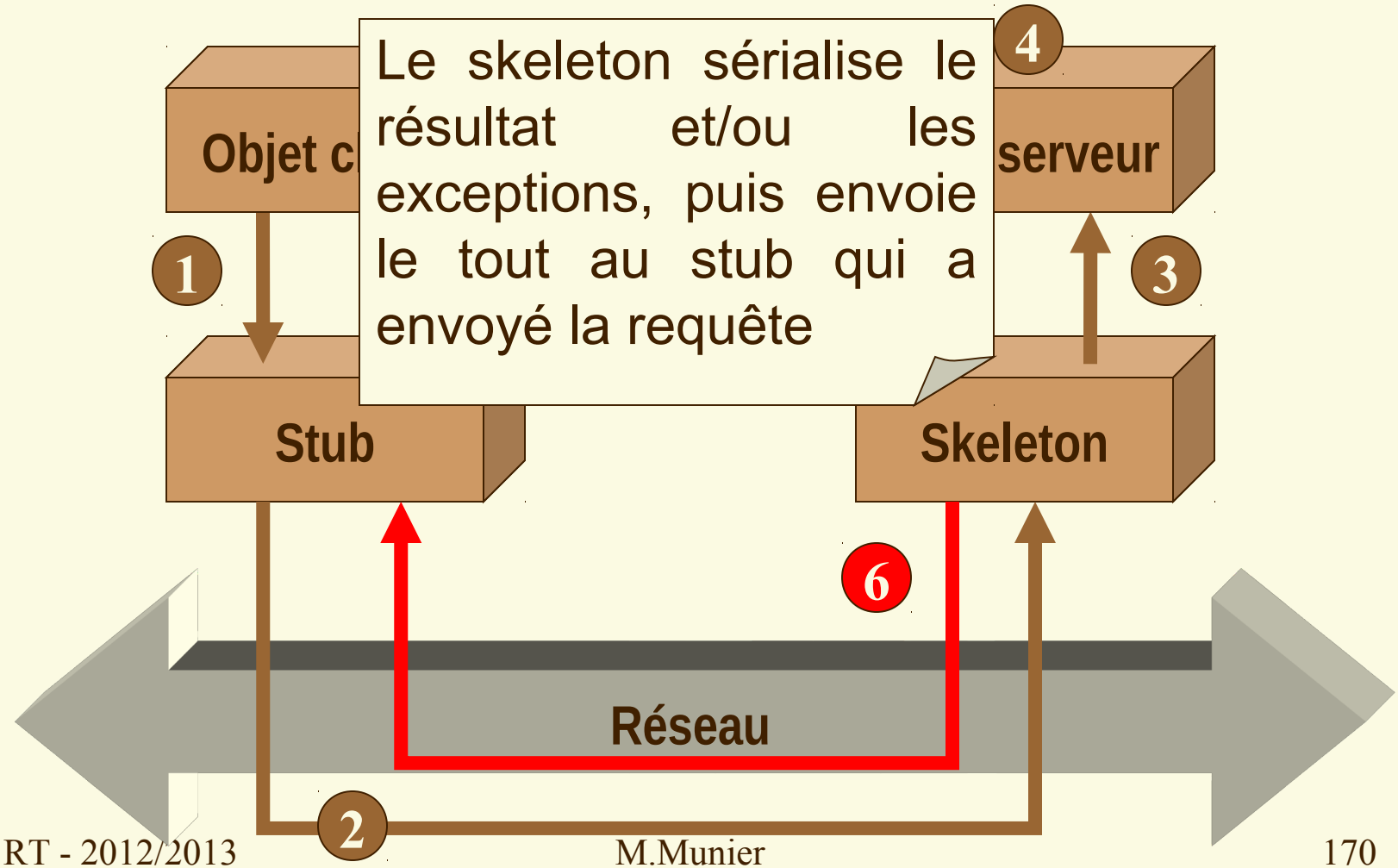
L'objet serveur exécute la méthode (**normalement**)



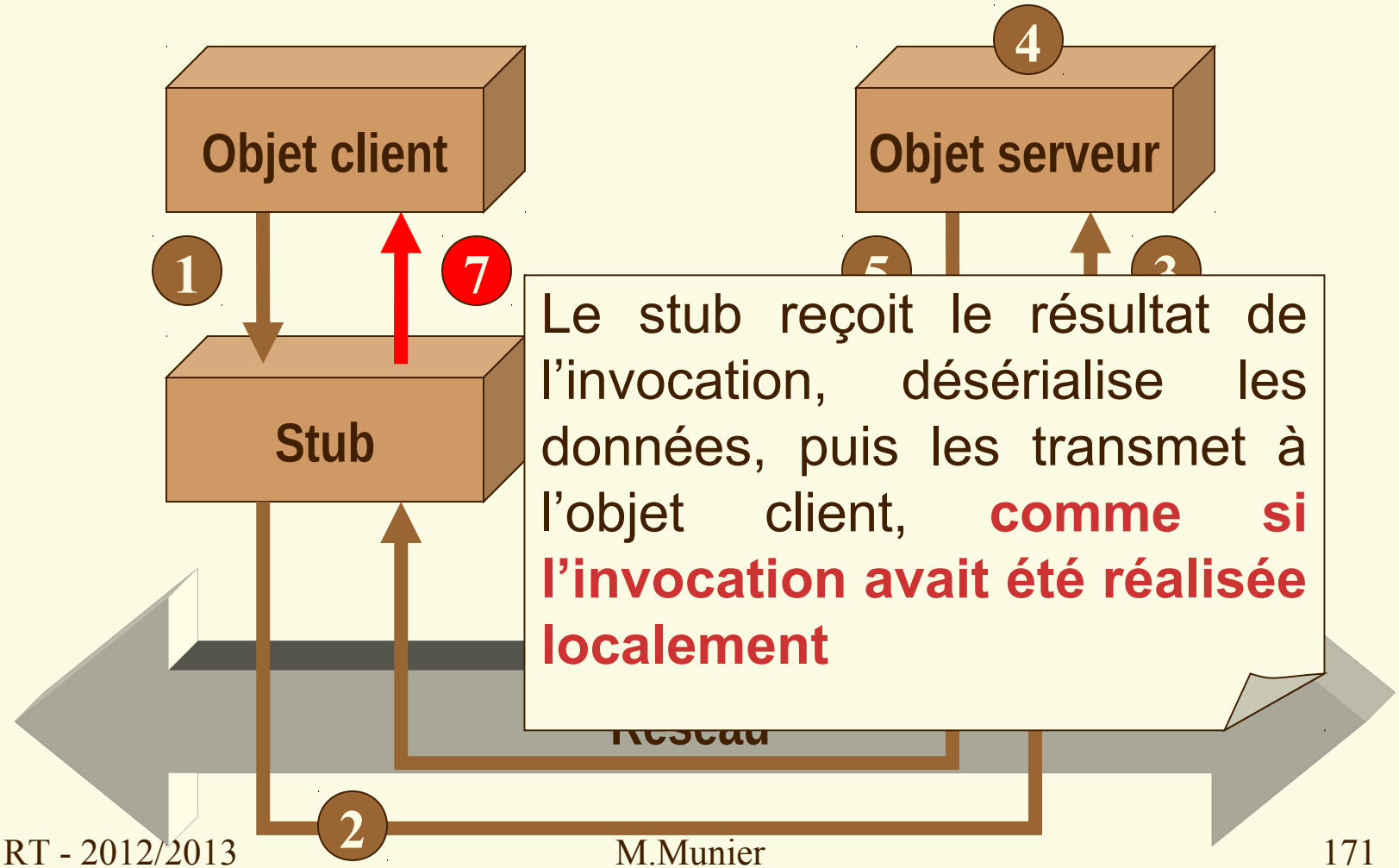
Architecture



Architecture



Architecture



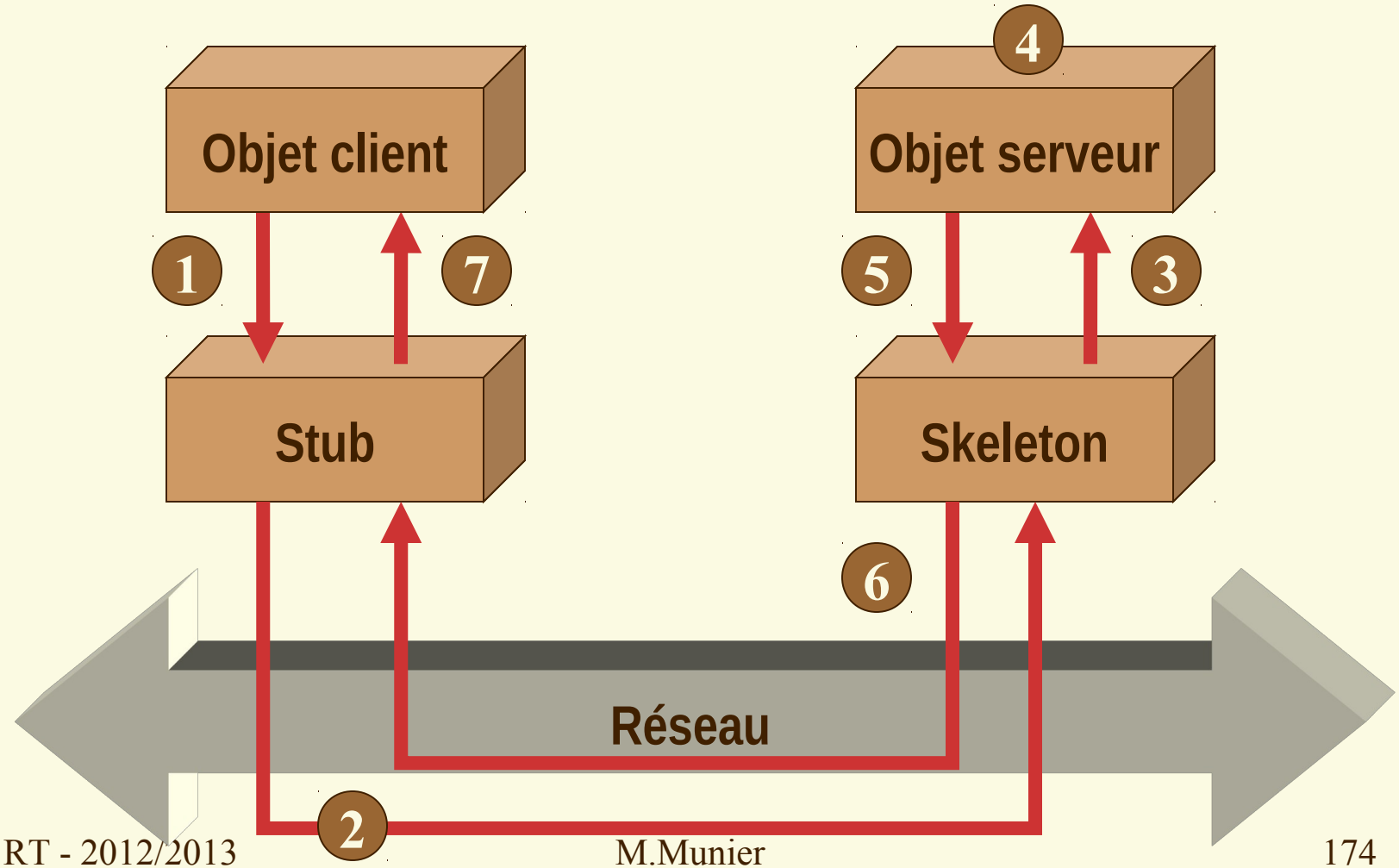
Fonctionnement

1. L'objet client invoque (**normalement**) la méthode sur le stub
2. Le stub sérialise les paramètres et envoie la requête sur le réseau
3. Le skeleton reçoit la requête, désérialise les paramètres, puis invoque la « vrai » méthode sur l'objet serveur
4. L'objet serveur exécute la méthode

Fonctionnement

5. L'objet serveur retourne le résultat et/ou les exceptions au skeleton (et non au client)
6. Le skeleton sérialise le résultat et/ou les exceptions, puis envoie le tout au stub qui a envoyé la requête
7. Le stub reçoit le résultat de l'invocation, désérialise les données, puis les transmet à l'objet client, **comme si l'invocation avait été réalisée localement**

Architecture



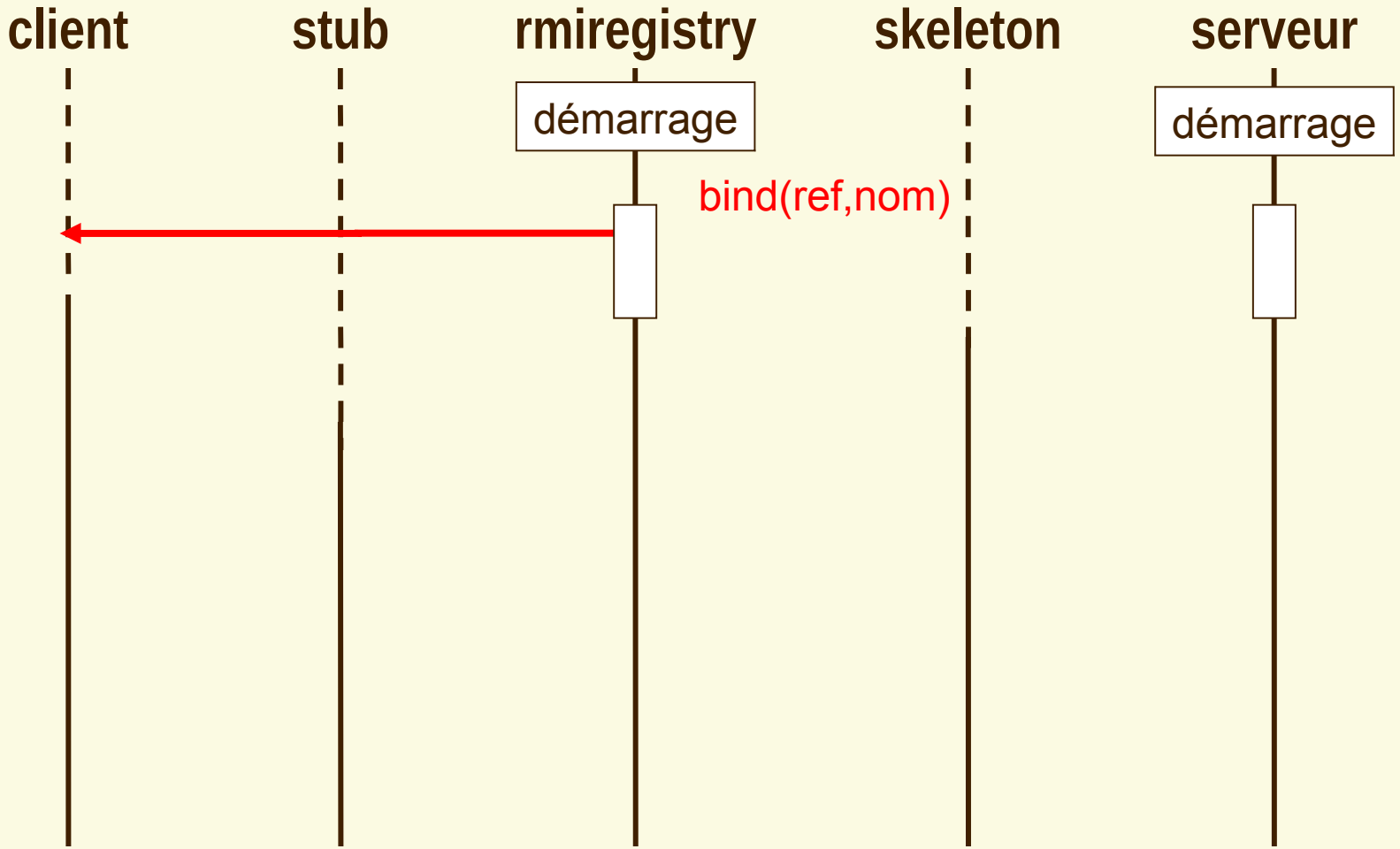
Programmation

- Le stub n'est qu'une interface permettant au client d'accéder, via le réseau, aux services proposés par le serveur
- Pour construire le stub, seule l'interface du serveur est nécessaire; il n'est pas nécessaire de connaître son implémentation
- Quand on programme en client/serveur, il est « indispensable » de bien distinguer l'interface et l'implémentation du serveur

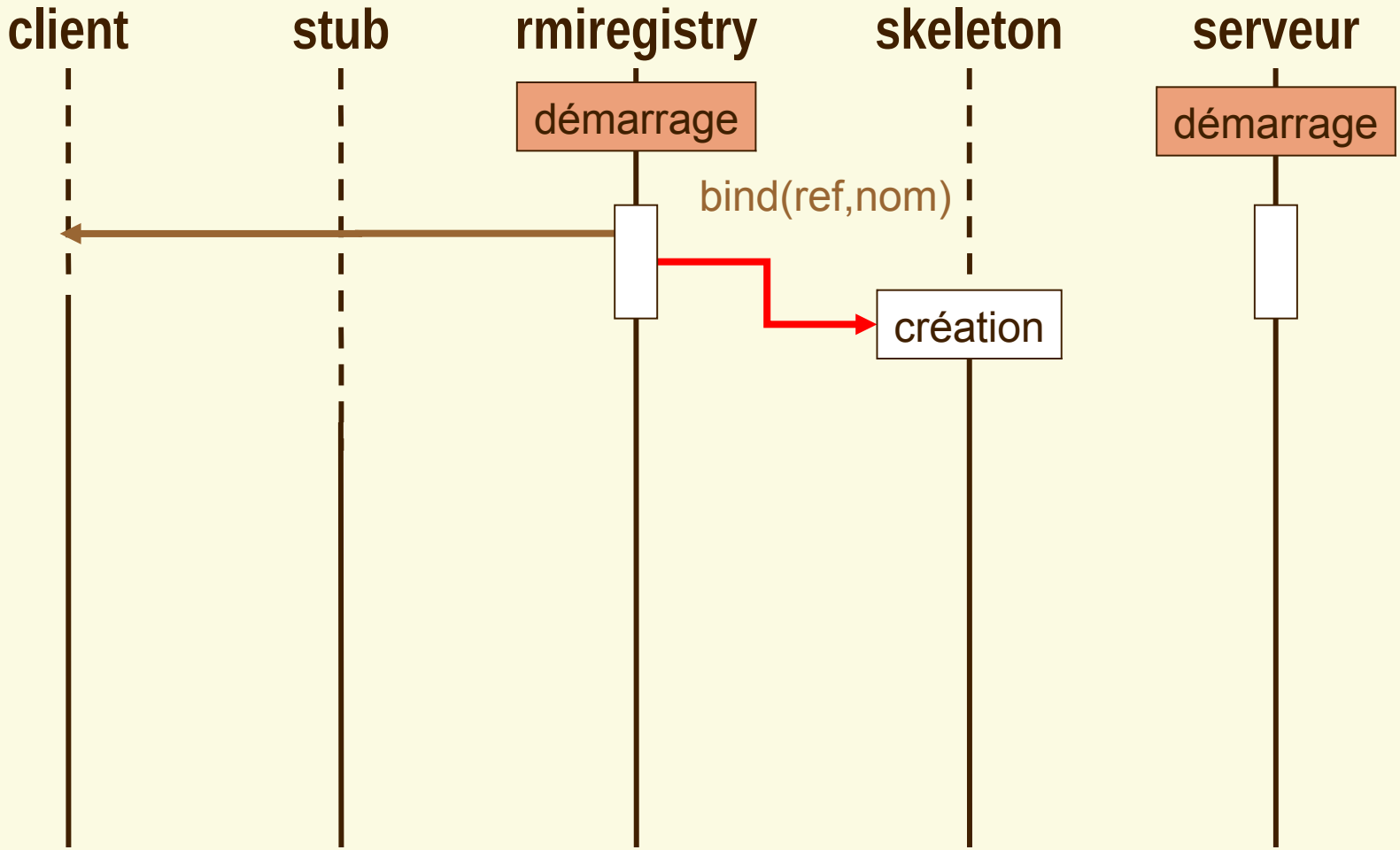
Programmation

- Le programmeur se contente de développer les interfaces et les implémentations, comme avant
- La seule contrainte est que les interfaces des objets distants doivent étendre l'interface `java.rmi.Remote`
- Les stubs et les skeletons sont générés automagiquement par la commande `rmic`

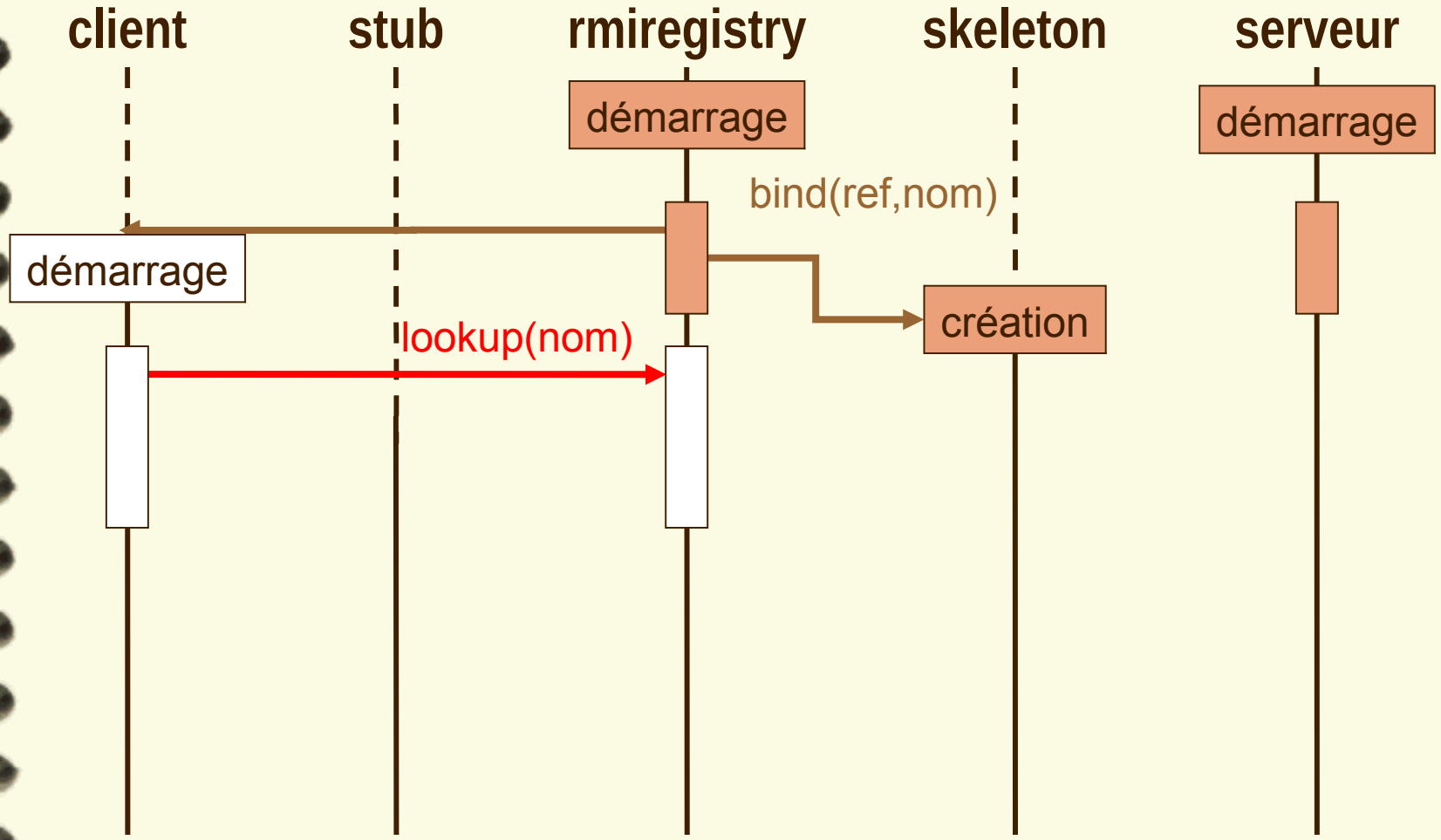
Fonctionnement



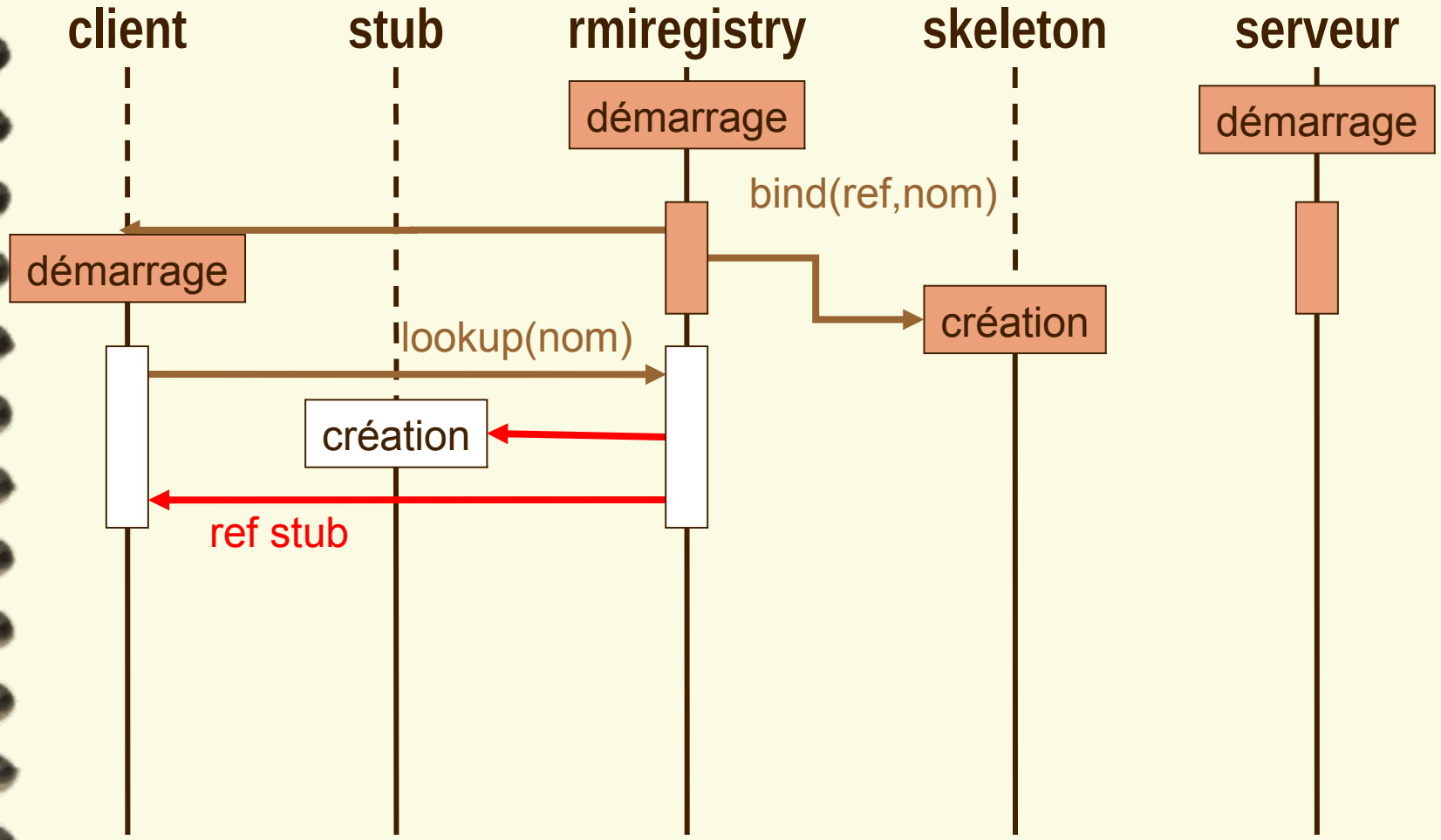
Fonctionnement



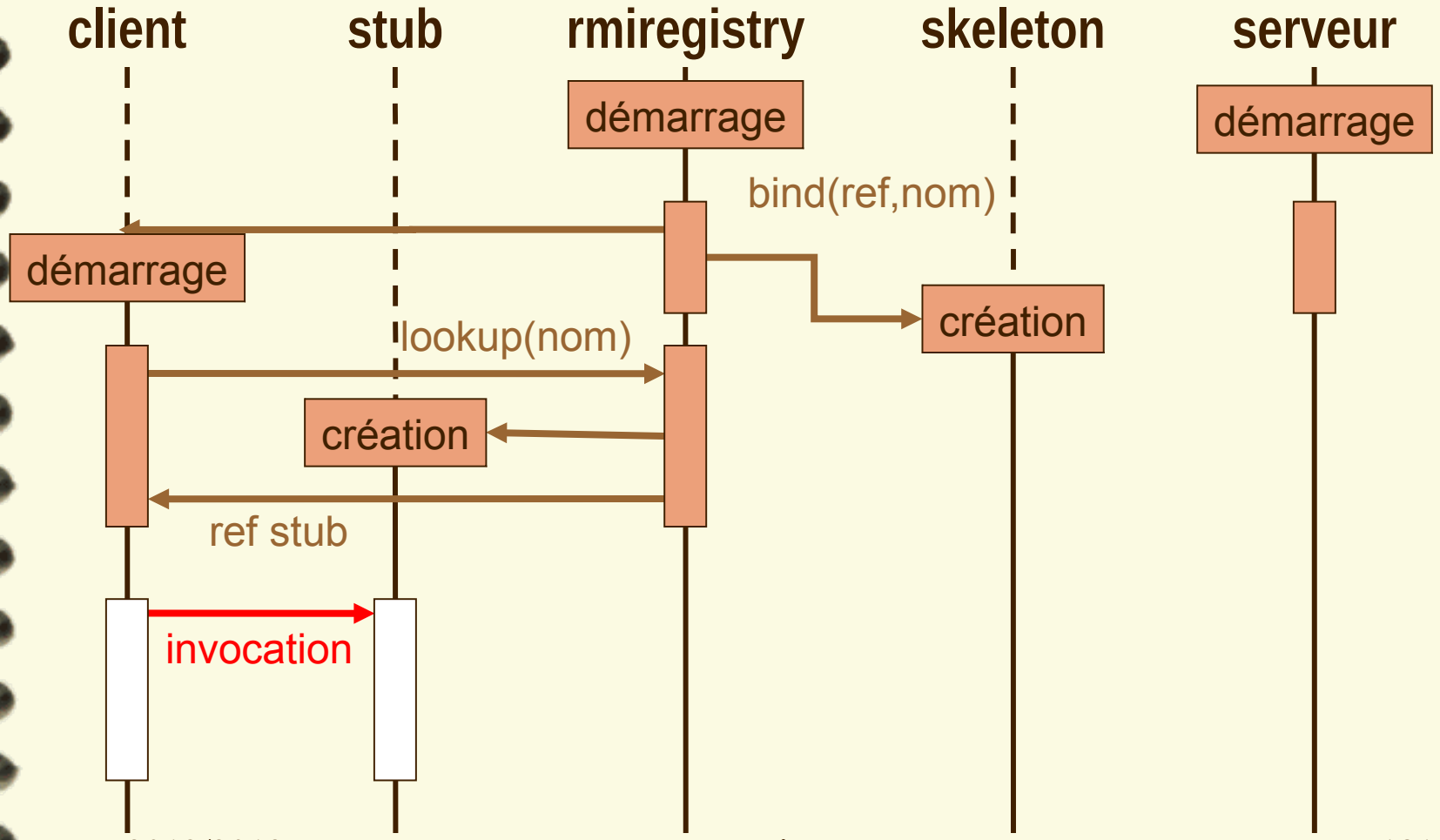
Fonctionnement



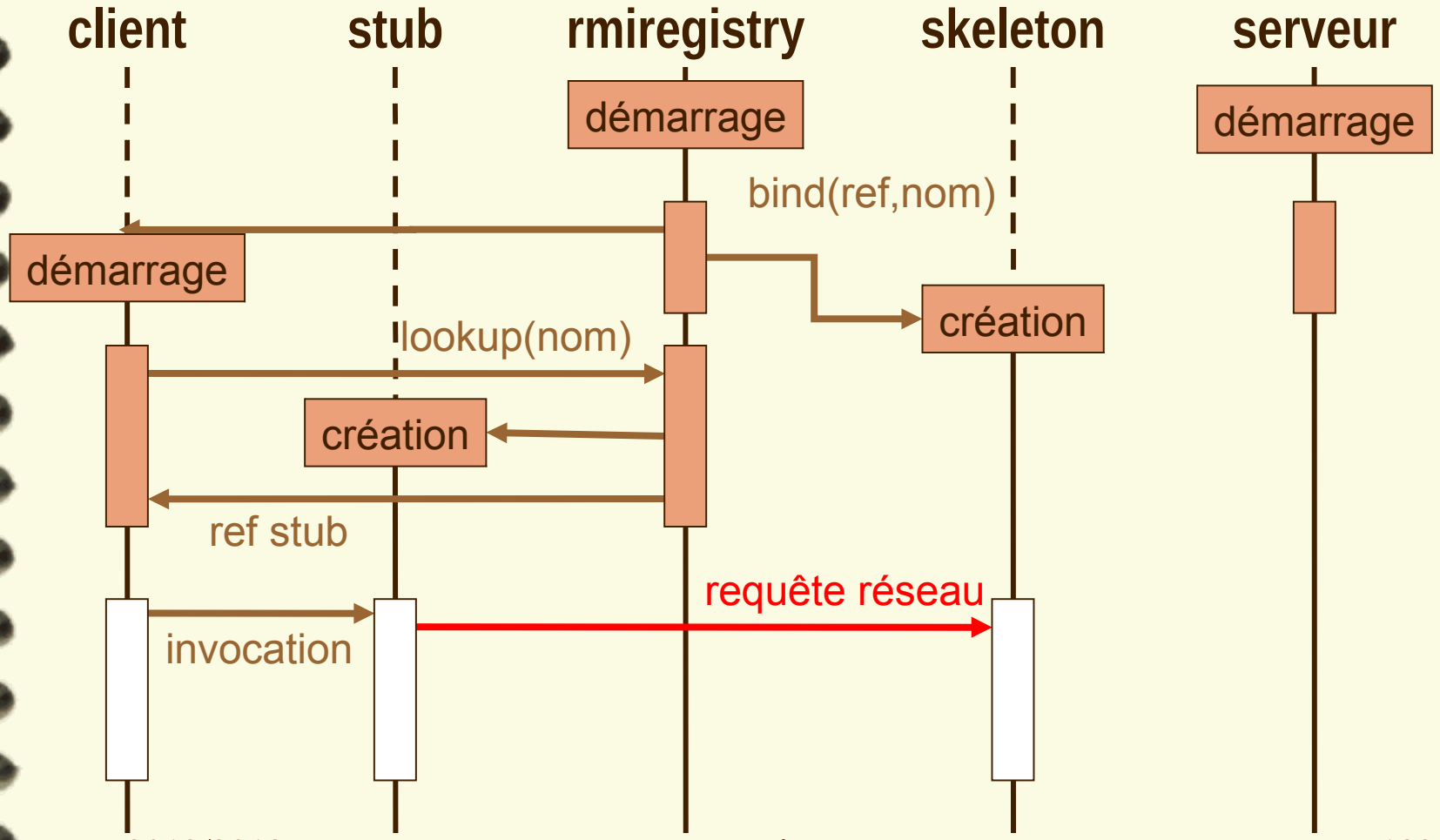
Fonctionnement



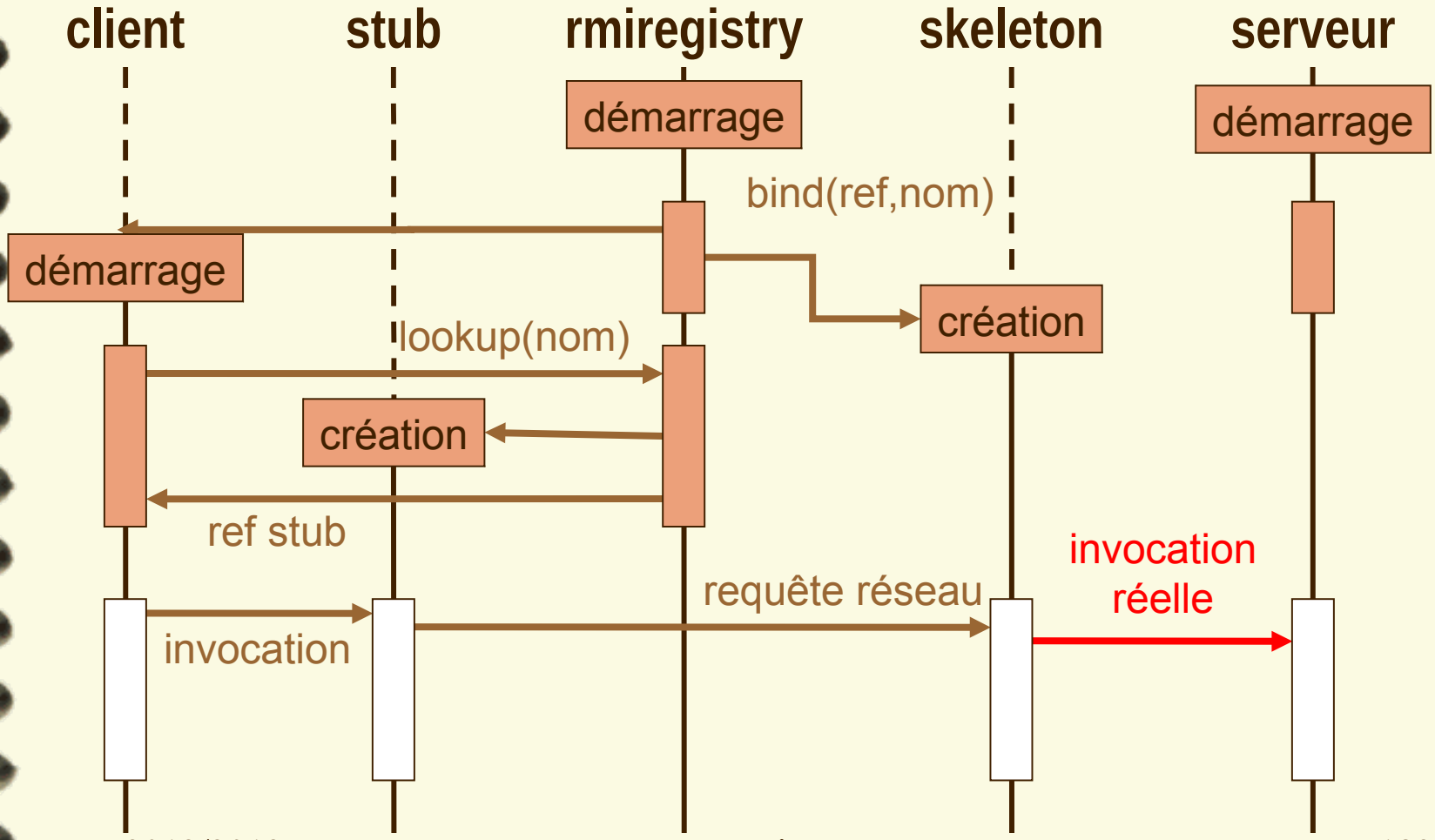
Fonctionnement



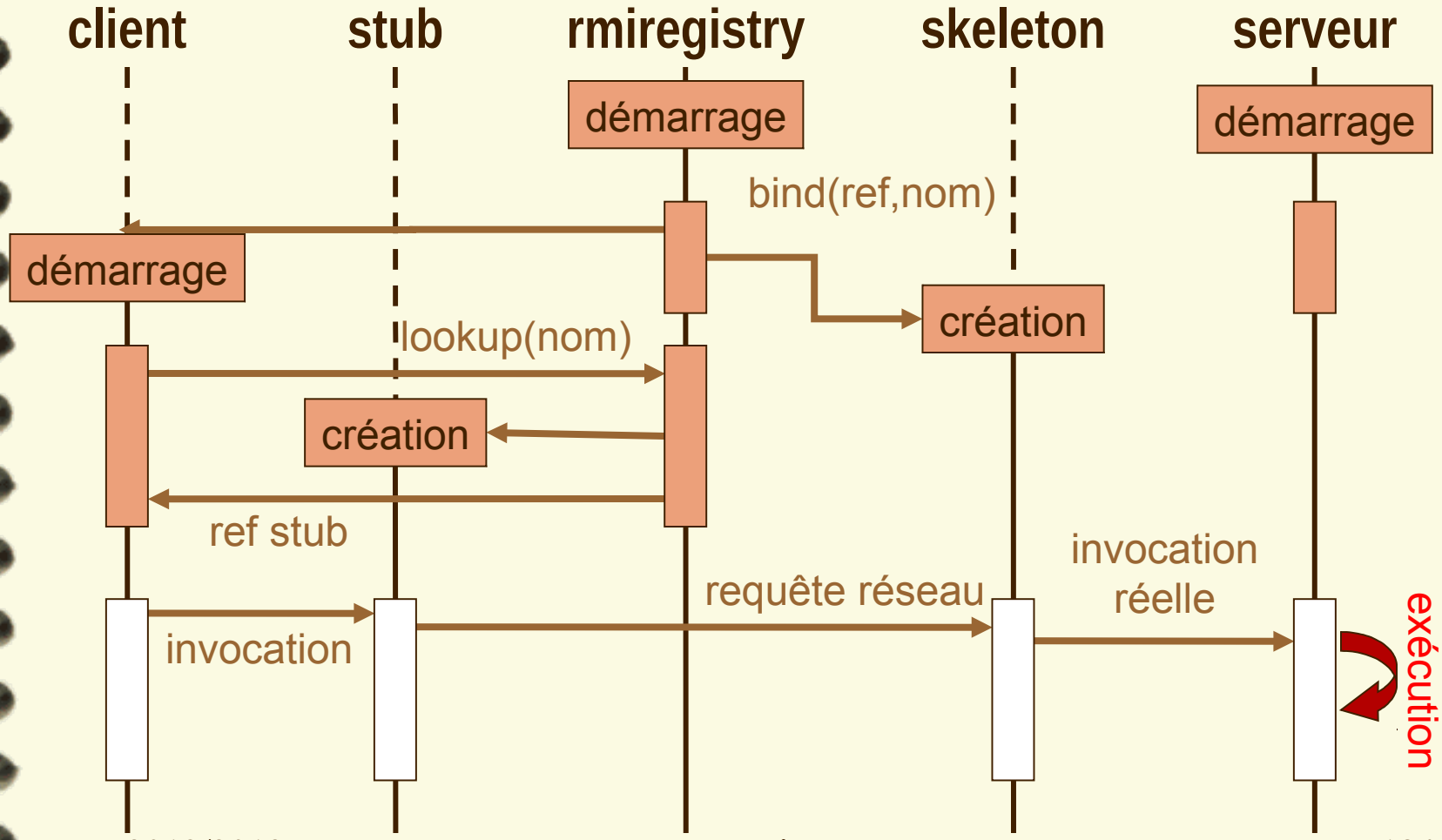
Fonctionnement



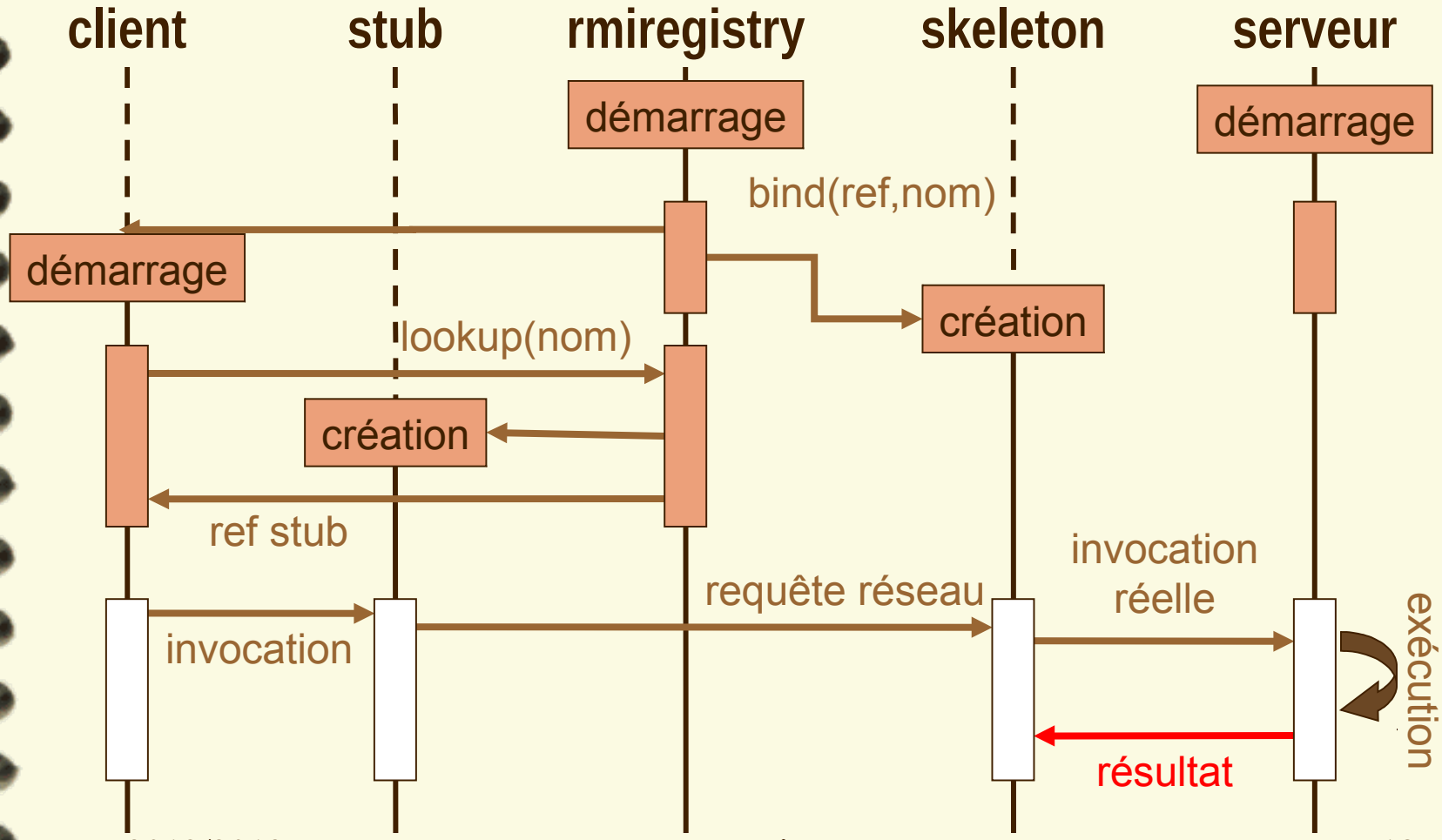
Fonctionnement



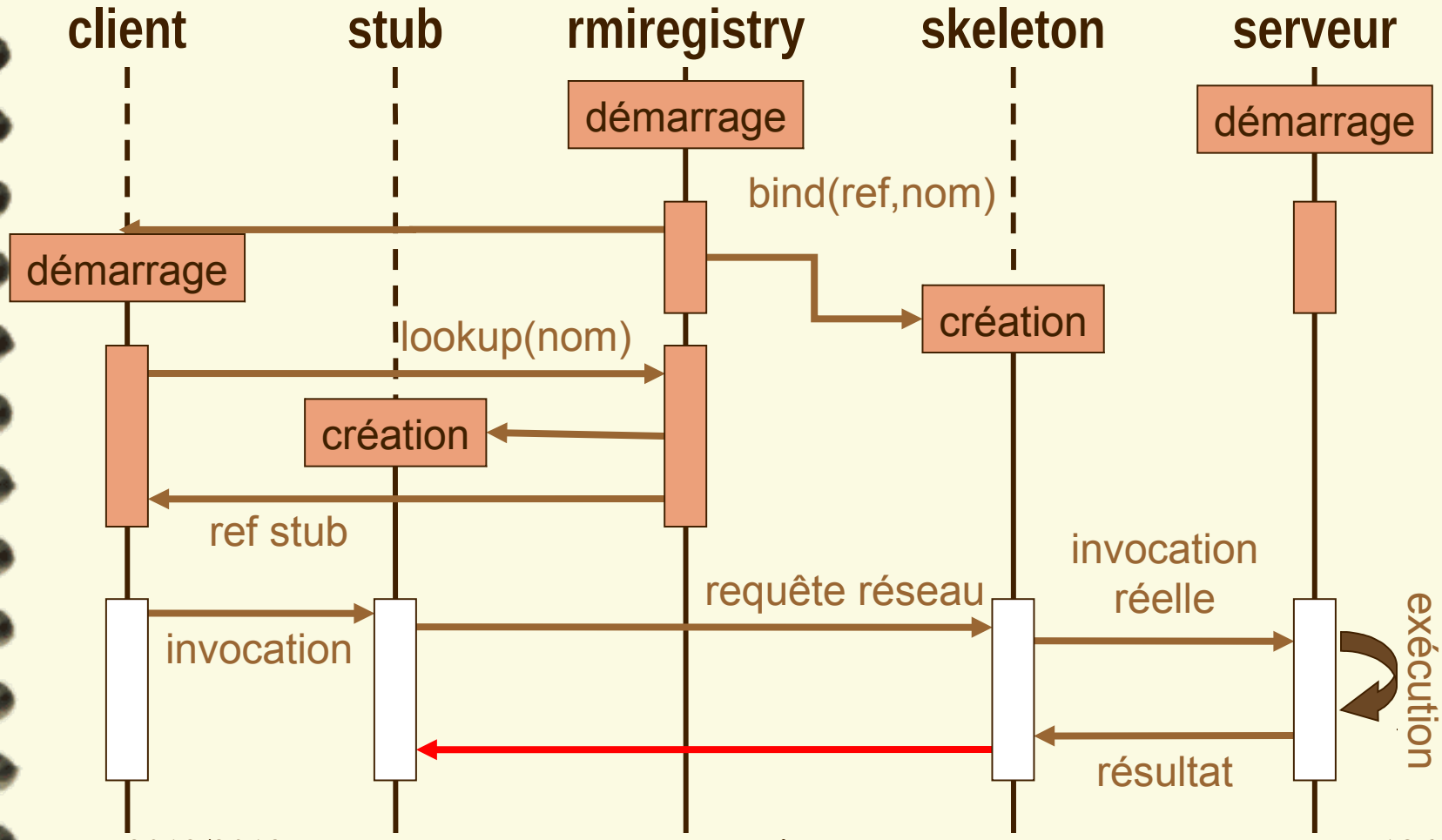
Fonctionnement



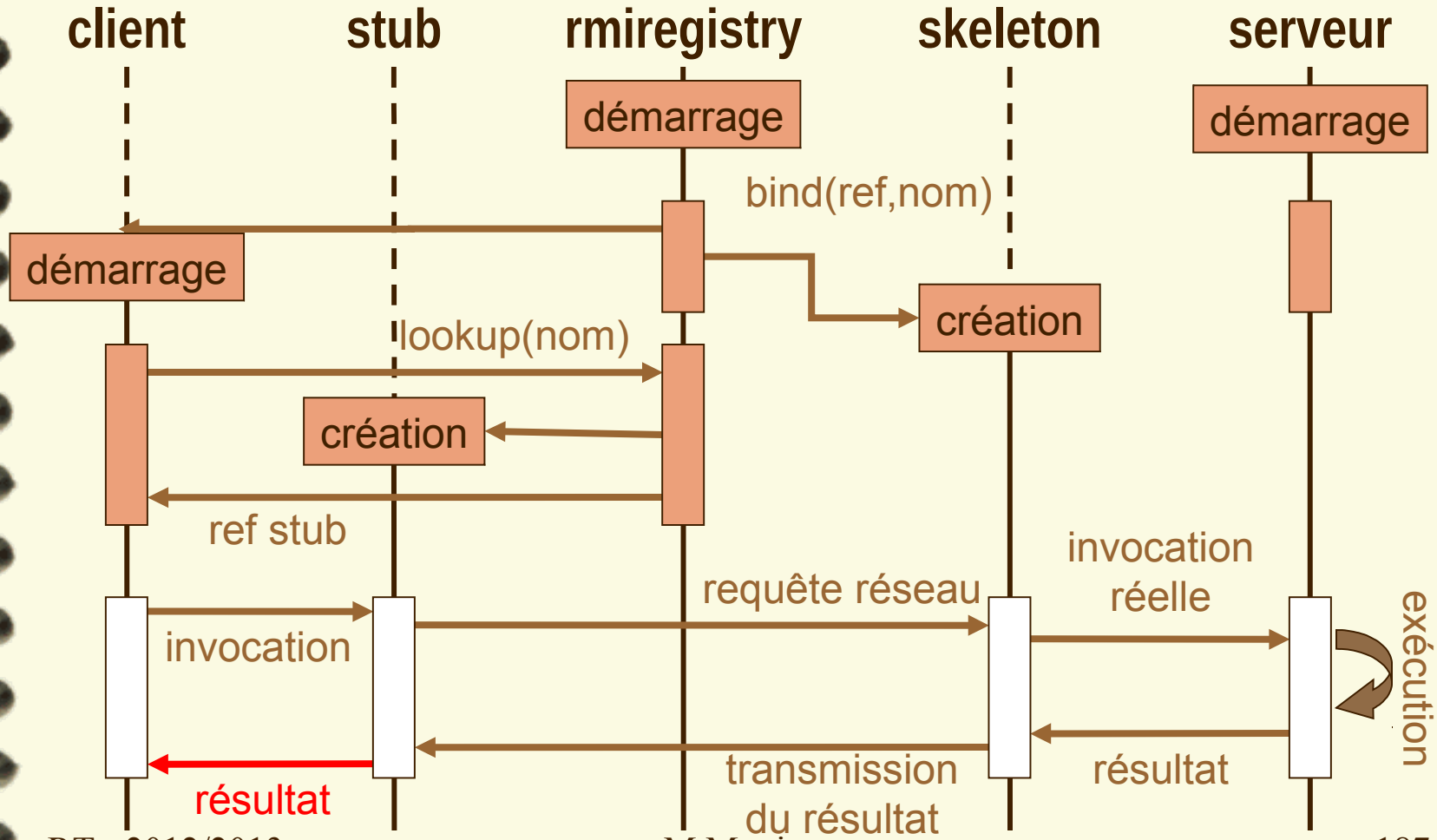
Fonctionnement



Fonctionnement



Fonctionnement



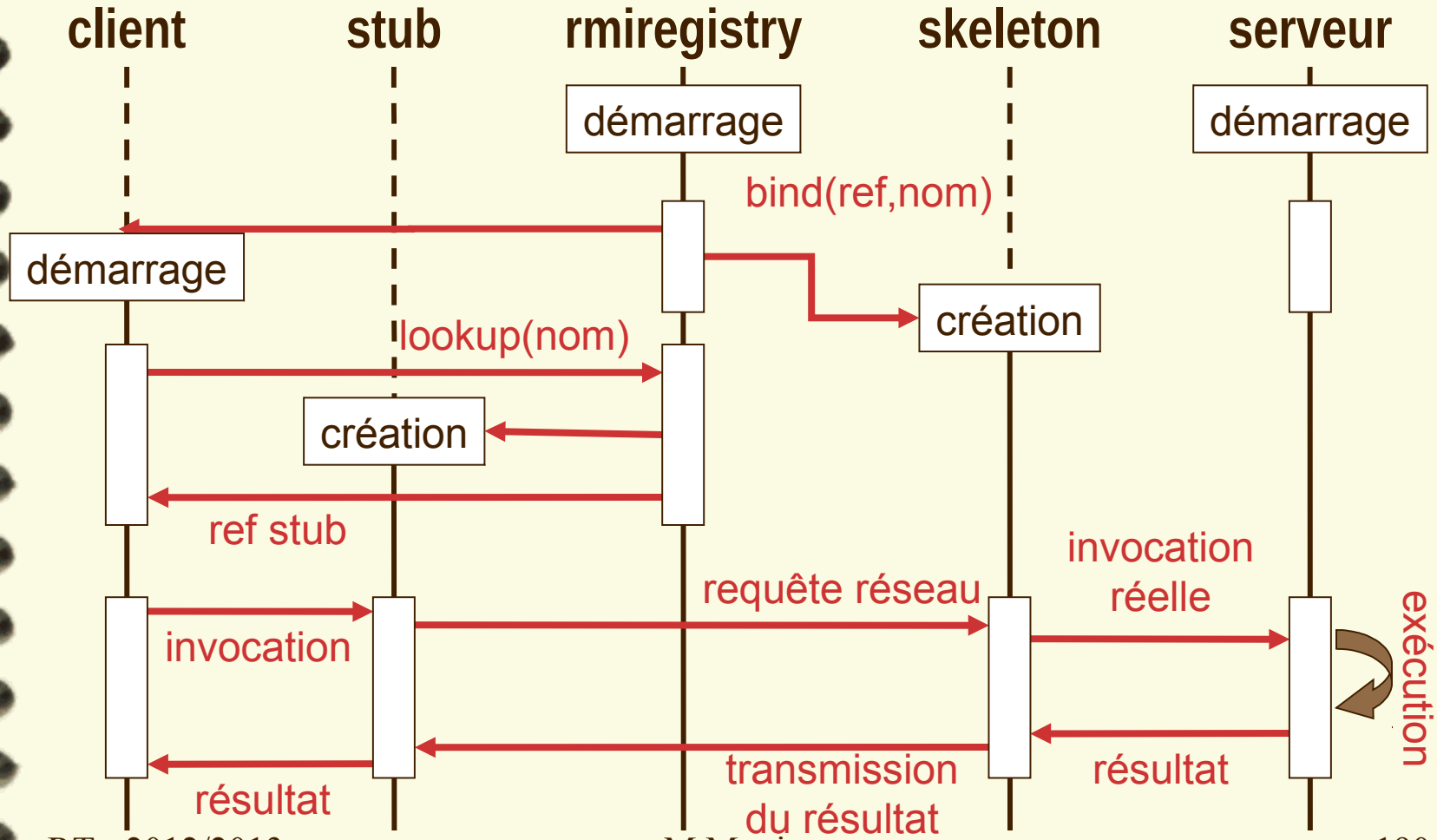
Programmation

- Au démarrage du serveur, celui-ci doit se faire connaître du système
  - il doit s'enregistrer (**`bind`**) auprès d'un annuaire: le **`rmiregistry`**
  - pour cela il fournit une référence sur l'objet qui répondra aux requêtes ainsi qu'un nom qui permet de l'identifier

Programmation

- De son côté, le client doit se connecter au serveur pour pouvoir utiliser ses services
  - il interroge le `rmiregistry` en donnant le nom du serveur (`lookup`)
  - celui-ci lui retourne une référence sur un stub connecté à ce serveur (via un skeleton)
  - le client peut alors invoquer des méthodes sur le stub qui se charge de transmettre les requêtes au skeleton puis au serveur

Fonctionnement



Conclusion

- Les RMI permettent à des objets s'exécutant dans des machines virtuelles Java différentes de communiquer par invocation de méthodes
- Ceci est indépendant de la plate-forme: des objets Java sous Windows peuvent interagir avec des objets Java sous Unix
- Attention aux interblocages, car tous les objets s'exécutent en parallèle !

Plan

- Pourquoi Java ?
- Syntaxe du langage
- Classes et objets en Java
- Héritage, interfaces
- Paquetages, exceptions
- Architecture client/serveur (RMI)
- Java et son environnement

Environnement Java

- Quand on parle de l'**environnement Java** cela englobe à la fois:
  - le **compilateur Java**
  - l'**API** avec tous les paquetages standards:
    - entrées/sorties, graphisme (AWT, Swing), réseau (sockets), objets distants (RMI, CORBA), connexion BdD (JDBC), processus (Thread), …
  - **différents outils** d'aide au développement
    - documentation « automatique » avec javadoc,
    - debuggeur, rmic, …

Environnement Java

JDK

- Pour conclure ce cours sur le langage Java, on va (rapidement) étudier plusieurs éléments de l'API:
  - les mécanismes d'**entrées/sorties**
  - des **utilitaires standard** tels que les classes `Vector`, `Enumeration`, …
  - les **Threads**
  - la connexion à une BdD via le **JDBC**

Entrées/sorties

JDK

- En Java, toutes les I/O sont définies en termes de **flots**
- Un **flot** est une séquence ordonnée d'informations possédant une **source** (flot d'entrée) ou une **destination** (flot de sortie)
- Les flots isolent le programmeur du système d'exploitation sous-jacent

Entrées/sorties

JDK

- Le paquetage I/O de Java se nomme **`java.io`**
- Y sont définis des flots de base (classes abstraites) ainsi que des flots spécifiques (comme ceux dédiés à la gestion des fichiers)
- Bon nombre de méthodes de **`java.io`** utilisent la classe **`IOException`** pour signaler des erreurs

Entrées/sorties

JDK

- **InputStream**
  - classe abstraite pour lire un flot d'octets en entrée à partir d'une source donnée
  - quelques méthodes (publiques):
    - `InputStream()`
    - `abstract int read() throws IOException`
    - `int read(byte[] buf) throws IOException`
    - `long skip(long count) throws IOException`
    - `int available() throws IOException`
    - `void close() throws IOException`

Entrées/sorties

JDK

- **`OutputStream`**
  - classe abstraite pour manipuler un flot d'octets en sortie vers une destination donnée
  - quelques méthodes (publiques):
    - `OutputStream()`
    - `abstract void write(int b) throws IOExcep…`
    - `void write(byte[] buf) throws IOException`
    - `void flush() throws IOException`
    - `void close() throws IOException`

Entrées/sorties

JDK

- **`PrintStream`**
  - classe dérivée de **`OutputStream`**
  - fournit les méthodes **`print`** et **`println`** pour afficher des données sous forme lisible
  - ces méthodes sont implémentées pour les types suivants:
    - **`char`**, **`int`**, **`float`**, **`Object`**, **`boolean`**, **`char[]`**, **`long`**, **`double`** et **`String`**
  - **`System.out`** et **`System.err`** appartiennent à cette classe

Entrées/sorties

JDK

- **`FileInputStream`** et **`FileOutputStream`**
  - dérivées de **`InputStream`** et **`OutputStream`**
  - gèrent les I/O vers des fichiers
  - chacune possède 3 constructeurs qui prennent respectivement en paramètre:
    - soit un **`String`** contenant le nom du fichier
    - soit un **`File`** faisant référence à un fichier (**`getName()`**, **`getPath()`**, **`getParent()`**, **`exists`**, **`canRead`**, **`isFile`**, **`length()`**, **`mkdir()`**,...)
    - soit un objet de type **`FileDescriptor`**

Entrées/sorties

JDK

```
import java.io.*;

// compte le nombre de caractères et d'espaces
// au sein d'un fichier

class CountSpace {
  public static main(String[] args)
    throws IOException
  {
    InputStream in;
    if (args.length == 0)
      in = System.in;  // entrée standard
    else
      in = new FileInputStream(args[0]);
```

Entrées/sorties

JDK

```
    int ch;
    int nbChars  = 0;
    int nbSpaces = 0;

    while ( (ch=in.read()) != -1 )
      {
        nbChars++;
        if ( Character.isSpace((char)ch) )
          nbSpaces++;
      }

    System.out.println(nbChars  + " caractères" +
                       nbSpaces + " espaces");
  }
}
```

Entrées/sorties

JDK

- **`DataInputStream`** et **`DataOutputStream`**
  - bien que la lecture et l'écriture d'octets soient utiles, il est souvent intéressant de transmettre des données d'un type particulier
  - permettent de convertir des types primitifs (**`char`**, **`int`**, **`double`**,…) en **flots d'octets**
  - ATTENTION: flot d'octets ≠ chaîne de caractères
    - ➔ équivalent au **format binaire** dans les autres langages de programmation

Entrées/sorties

JDK

- `DataInputStream` et `DataOutputStream`

Read	Write	Type
`readBoolean`	`writeBoolean`	boolean
`readChar`	`writeChar`	char
`readByte`	`writeByte`	byte
`readShort`	`writeShort`	short
`readInt`	`writeInt`	int
`readLong`	`writeLong`	long
`readFloat`	`writeFloat`	float
`readDouble`	`writeDouble`	double
`readUTF`	`writeUTF`	String au format UTF

Entrées/sorties

JDK

- Généralisation avec **`ObjectInputStream`** et **`ObjectOutputStream`**
  - comme **`Data…Stream`** mais pour la sérialisation d'objets (et même de graphes d'objets)
  - seuls les objets implémentant l'interface **`java.io.Serializable`** ou l'interface **`java.io.Externalizable`** peuvent être sérialisés dans des **`Data…Stream`**
    - ➔ méthodes **`writeObject`** et **`readObject`**

Entrées/sorties

JDK

- Exemple d'écriture d'un objet dans un **ObjectOutputStream** :

```
FileOutputStream ostream = new FileOutputStream("t.tmp");
ObjectOutputStream p = new ObjectOutputStream(ostream);
p.writeInt(12345);
p.writeObject("Today");
p.writeObject(new Date());
p.flush();
ostream.close();
```

Entrées/sorties

JDK

- Exemple de lecture de l'objet depuis un **ObjectInputStream** :

```
FileInputStream istream = new FileInputStream("t.tmp");
ObjectInputStream p = new ObjectInputStream(istream);
int i = p.readInt();
String today = (String)p.readObject();
Date date = (Date)p.readObject();
istream.close();
```

Entrées/sorties

JDK

- Le mécanisme de sérialisation **traverse automatiquement** les références vers d'autres objets, sauvegardant et restaurant ainsi tout le graphe d'objets
- Si un traitement spécial est nécessaire:

```
private void writeObject(java.io.ObjectOutputStream stream)
    throws IOException;

private void readObject(java.io.ObjectInputStream stream)
    throws IOException, ClassNotFoundException;
```

Entrées/sorties

JDK

- Si on veut écrire sur des flux de caractères
  - classe **BufferedWriter**
    - `public void write(int c) throws IOException`
    - `public void write(String s,int off,int len) throws IOException`
    - `public void newLine() throws IOException`
    - `public void flush() throws IOException`
    - `public void close() throws IOException`
    - `...`

Entrées/sorties

JDK

- Si on veut lire des données sur des flux de caractères
  - classe **BufferedReader**
    - `public int read() throws IOException`
    - `public String readLine() throws IOException`
    - `public long skip(long n) throws IOException`
    - `public boolean ready() throws IOException`
    - `public void close() throws IOException`
    - `...`

Entrées/sorties

JDK

```
import java.io.*;

class Saisie1 {

    public static void main(String args[])
    {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    String  buf = null;
    double  d   = 0;
    boolean ok  = false;

    while (!ok)
        {
    System.out.print("Saisie d'un double\t: ");

    // On essaye de lire une chaîne de caractères
    try {
        buf = in.readLine();
    }
    catch (IOException e2)
        {
    System.out.println("### Problème de saisie !!! ###");
        }
```

Entrées/sorties

JDK

```
            // On essaye de convertir la chaîne lue en double
            try {
                d   = java.lang.Double.valueOf(buf).doubleValue();
                ok  = true;
            }
            catch (NumberFormatException e1)
                {
                      System.out.println("### Erreur de conversion !!! ###");
                }
        }

    System.out.println("--> "+d);
    }

}
```

Entrées/sorties

JDK

- **IOException**
  - le paquetage **java.io** définit 4 sous-classes spécifiques de IOException:
    - **EOFException**: la fin du fichier est atteinte
    - **FileNotFoundException**: le fichier est introuvable
    - **InterruptedIOException**: une opération d'E/S a été détournée par un interruption de thread
    - **UTFDataFormatException**: la syntaxe UTF de la chaîne en cours de lecture n'est pas viable

Entrées/sorties

JDK

- Conclusion:
  - On pourrait aller beaucoup plus loin sur les flots en Java avec:
    - les flots de type `Piped`,
    - les `StreamTokenizer`,
    - les `SequenceInputStream`,
    - les flots de type `Filter`,
    - etc…
  - Pour obtenir plus d'informations, consulter la documentation du paquetage `java.io`

Utilitaires standard

JDK

- Le paquetage Java **`java.util`** définit de nombreuses classes et interfaces standard telles que
  - **`StringTokenizer`**: découpage d'une chaîne
  - **`Vector`**: tableaux dynamiques
  - **`Dictionary`**: dictionnaires
  - **`Hashtable`**: tables de hachage
  - **`Enumeration`**: itérateurs sur des ensembles
  - **`Date`**: dates (granularité d'une seconde)
  - **`Random`**: nombres pseudo-aléatoires

Utilitaires standard

JDK

- **StringTokenizer**
  - cette classe découpe une chaîne de caractères en unités lexicales en se basant sur des délimiteurs
    - `StringTokenizer(String str, String delim)`
    - `StringTokenizer(String str)`
      - est équivalent à `StringTokenizer(str,"\t\n\r")`
    - `boolean hasMoreTokens()`
    - `String nextToken()`
    - `String nextToken(String delim)`
    - `int countTokens()`

Utilitaires standard

JDK

- **`Vector`**
  - cette classe fournit un **tableau redimensionnable** d'objets `Object`
  - on peut ajouter des éléments au début, ou milieu ou à la fin du vecteur
  - on peut accéder à tout élément dans le vecteur par un indice de tableau
  - **Attention:** les éléments vont nous être retournés en tant qu'`Object` ⇒ il faut donc faire un cast

Utilitaires standard

JDK

- **`Vector`**
  - constructeurs pour créer un vecteur
    - **`Vector(int initialCapacity, int capInc)`**
      - un incrément de capacité `capInc` de 0 signifie que le buffer double chaque fois qu'il a besoin de grandir
    - **`Vector(int initialCapacity)`**
      - est équivalent à `Vector(initialCapacity,0)`
    - **`Vector()`**
      - est équivalent à `Vector(0,0)`

Utilitaires standard

JDK

- **Vector**
  - méthodes pour modifier le vecteur
    - `void setElementAt(Object obj, int index)`
    - `void removeElementAt(int index)`
    - `void insertElementAt(Object obj, int index)`
    - `void addElement(Object obj)`
    - `boolean removeElement(Object obj)`
    - `void removeAllElements()`

Utilitaires standard

JDK

- **Vector**
  - méthodes pour extraire des valeurs du vecteur
    - `Object ElementAt(int index)`
    - `boolean contains(Object obj)`
    - `int indexOf(Object obj, int index)`
    - `int indexOf(Object obj)`
    - `int lastIndexOf(Object obj, int index)`
    - `int lastIndexOf(Object obj)`
    - `void copyInto(Object[] anArray)`
    - `Enumeration elements()`
    - `Object firstElement()`
    - `Object lastElement()`

Utilitaires standard

JDK

- **Vector**
  - méthodes pour gérer la taille du vecteur
    - `int size()`
    - `boolean isEmpty()`
    - `void trimToSize()`
    - `void setSize(int newSize)`
    - `int capacity()`
    - `void ensureCapacity(int minCapacity)`

Utilitaires standard

JDK

- **`Enumeration`**
  - cette interface est un moyen d'itérer sur les éléments d'une collection
  - elle définit deux méthodes:
    - `boolean hasMoreElements()`
    - `Object nextElement()`
  - **Attention:** de nouveau, les éléments vont nous être retournés en tant qu'`Object` ⇒ il faut donc faire un cast

Utilitaires standard

JDK

```
import java.util.*;

class DNSImpl implements DNS {

    // Attributs

    private Vector listeComp;

    // Constructeurs

    public DNSImpl()
    {
        listeComp = new Vector();
    }

    // Methodes

    public void register(ComposantRZO comp)
    {
        listeComp.addElement(comp);
    }
```

Utilitaires standard

JDK

```
public ComposantRZO lookup(AdresseRZO addr) throws UnknownHostException
    {
        ComposantRZO result = null;
        Enumeration  e      = listeComp.elements();

        while ( e.hasMoreElements() && (result==null) )
        {
            ComposantRZO comp = (ComposantRZO)e.nextElement();
            if ( addr.compareTo(comp.adresse()) == 0 )
                result = comp;
        }

        if (result==null)
            throw new UnknownHostException(addr);

        return result;
    }

...
```

Utilitaires standard

JDK

- **`Dictionary`**
  - cette classe (abstraite) permet d'associer des **clés** à des **valeurs**
  - une clé donnée est associée à une valeur au maximum
  - les clés et les valeurs doivent être des `Object` non `null`
  - les objets servant de clés doivent disposer d'une méthode `equals`

Utilitaires standard

JDK

- **Dictionary**
  - méthodes disponible
    - `abstract int size()`
    - `abstract boolean isEmpty()`
    - `abstract Enumeration keys()`
    - `abstract Enumeration elements()`
    - `abstract Object get(Object key)`
    - `abstract Object put(Object key,Object val)`
    - `abstract Object remove(Object key)`

Utilitaires standard

JDK

- **`Hashtable`**
  - classe dérivée de **`Dictionary`**
  - utilise une table de "hachage" pour optimiser les recherches
  - outre la méthode **`equals`**, les objets servant de clés doivent également implémenter la méthode **`hashCode`**

Utilitaires standard

JDK

- **Date** (et **Calendar** avec Java 2)
  - gestion des dates et des instants
    - `int getYear()` `void setYear(int year)`
    - `int getMonth()` `void setMonth(int month)`
    - `int getDay()` `void setDay(int day)`
    - `int getHours()` …
    - `int getMinutes()` …
    - `int getSeconds()` …
    - `long getTime()`
    - `boolean before(Date other)`
    - `boolean after(Date other)`
    - `boolean equals(Date other)`

Utilitaires standard

JDK

- **Random**
  - génération de nombres pseudo-aléatoires
    - `void setSeed(long seed)`
    - `int nextInt()`
    - `int nextInt(int n)`
    - `long nextLong()`
    - `boolean nextBoolean()`
    - `float nextFloat()`
    - `double nextDouble()`
    - `double nextGaussian()`

Thread

JDK

- Pour l'instant, nos programmes s'exécutent pas à pas: les méthodes sur les différents objets sont invoquées les unes après les autres
- Mais il est fréquent que l'on veuille avoir plusieurs processus qui s'exécutent en même temps et qui communiquent
- Bonne nouvelle: c'est plus simple qu'en C

Thread

JDK

- En Java, point de **`fork`**: on crée un nouveau processus (léger) en instanciant un nouvel objet à partir de la classe **`Thread`** (paquetage **`java.lang`**)

```
Thread processus = new Thread();
```

- C'est la méthode **`run`** qui détermine ce que va faire ce processus

Thread

JDK

- Par défaut, la méthode **`Thread.run`** ne fait rien ⇒ on doit donc
  - soit étendre **`Thread`** pour fournir une nouvelle méthode **`run`**
  - soit créer un objet **`Runnable`** et le passer au constructeur du **`Thread`**
- Quand la méthode **`run`** d'un thread retourne, le thread se termine

Thread

JDK

- Une fois qu'un thread a été créé, on peut:
  - le configurer (priorité initiale, nom,…)
  - l'exécuter en invoquant sa méthode **`start`**
  - le stopper explicitement via sa méthode **`stop`**
  - suspendre son exécution avec **`suspend`**
  - reprendre son exécution avec **`resume`**

Thread

JDK

```
import java.lang.*;

class PingPong extends Thread {
  String word;   // mot à afficher
  int delay;     // durée de la pause

  PingPong(String whatToSay, int delayTime)
  {
    word  = whatToSay;
    delay = delayTime;
  }
```

Thread

JDK

```
public void run()
{
  try {
    for (;;) {
      System.out.println(word + " ");
      sleep(delay);  // attente jusqu'à la
                     // prochaine fois
    }
  }
  catch (InterruptedException e) {
    // si la méthode sleep lance une telle
    // exception, on termine le thread
    return;
  }
}
```

Thread

JDK

```
  public static void main(String[] args)
  {
    // affiche ping tous les 1/30 de seconde
    new PingPong("ping",33).start();

    // affiche pong tous les 1/100 de seconde
    new PingPong("pong",10).start();
  }

}  // fin classe
```

Thread

JDK

- On peut donner un nom à un thread
  - en passant une `String` au constructeur
  - en invoquant la méthode `setName`
- On peut récupérer le nom d'un thread avec la méthode `getName`
- Les noms de thread ne servent qu'aux programmeurs: ils ne sont pas utilisés par le support d'exécution Java
- `Thread.currentThread()` retourne le thread en cours d 'exécution

Thread

JDK

- Synchronisation:
  - si un thread invoque une méthode **synchronized** sur un objet, cet objet est verrouillé
  - si un autre thread invoque lui-aussi une méthode **synchronized** sur ce même objet, ce second thread se bloque jusqu'à ce que le verrou soit relâché
  - le verrou est relâché quand la méthode **synchronized** se termine

Thread

JDK

- Synchronisation:
  - quand une méthode **`synchronized`** est invoquée sur un objet verrouillé par le même thread, elle est exécutée
  - mais le verrou ne sera relâché que lorsque la méthode **`synchronized`** la plus externe se terminera

Thread

JDK

```
class BankAccount {
  private double balance;

  public BankAccount(double initialDeposit) {
    balance = initialDeposit;
  }

  public synchronized double getBalance() {
    return balance;
  }

  public synchronized void deposit(double val) {
    balance += val;
  }
}
```

Thread

JDK

- Bloc **`synchronized`** :
  - on peut exécuter du code synchronisé sans invoquer de méthode **`synchronized`**

`synchronized (expr) statement`

  - **`expr`** doit produire un objet à verrouiller
  - **`statement`** est un bloc d'instructions

  - il n'est pas nécessaire que l'objet produit par **`expr`** soit effectivement utilisé dans le bloc **`statement`**

Thread

JDK

- Synchronisation avec **`wait`** et **`notify`** :
  - **`synchronized`** est suffisant pour empêcher des threads d'interférer l'un avec l'autre
  - un **`wait`** permet de mettre un thread en attente jusqu'à ce qu'il soit réveillé par un **`notify`**
  - les méthodes **`wait`** et **`notify`** sont définies dans la classe **`Object`** et sont donc héritées par tous les objets

Thread

JDK

- Il y a un modèle standard qu'il est important d'utiliser avec **wait** et **notify**
- Le thread qui attend une condition doit faire qqch comme ceci:

```
synchronized doWhenCondition()
{
  while (!condition)
    wait();
  // … fait ce qui doit être fait quand la
  // condition vaut true …
}
```

Thread

JDK

- La méthode est **`synchronized`** car la condition ne doit plus changer une fois qu'on l'a testée à **`true`**
- Le **`wait`** relâche le verrou quand il suspend le thread; le verrou est repris quand le thread est redémarré
- Le **`wait`** est dans une boucle car la condition n'est pas forcément **`true`** quand le thread est débloqué

Thread

JDK

- De l'autre côté, la méthode **notify** est invoquée par des méthodes qui changent des données sur lesquelles un autre thread peut être en attente sur cet objet

```
synchronized changeCondition()
{
   // …change une valeur dans une condition…
   notify();
}
```

- La méthode **notifyAll** permet de réveiller tous les threads en attente

Thread

JDK

- Ordonnancement:
  - on peut modifier la priorité d'un thread avec `setPriority(p)` où `p` est compris entre les constantes `MIN_PRIORITY` et `MAX_PRIORITY` de Thread
  - `getPriority()` donne la priorité d'un thread
  - `sleep(t)` met le thread en sommeil pour `t` millisecondes
  - `yield()` abandonne le thread en cours d'exécution et passe la main à l'ordonnanceur

Thread

JDK

- Les threads permettent donc à plusieurs objets de s'exécuter simultanément
- Les communications entre threads sont tout simplement réalisées par les invocations de méthodes
- Bien évidemment, il faut faire attention aux **interblocages** quand les threads tentent d'acquérir des verrous

Environnement Java

- **Rappel:** l'environnement Java c'est aussi
  - l'AWT et Swing pour les applis graphiques
  - les applets (exécution dans un browser web)
  - la réfléxivité du langage
  - Enterprise JavaBeans™ Architecture
  - JavaServer Pages™
  - Java™ Servlet
  - Java Naming and Directory Interface™ (JNDI)
  - Java™ IDL
  - JDBC™

Environnement Java

- **Rappel:** l'environnement Java c'est aussi
  - Java™ Message Service (JMS)
  - Java™ Transaction (JTA)
  - Java™ Transaction Service (JTS)
  - JavaMail
  - RMI-IIOP
  - JavaPhone™ API
  - Java TV™ API
  - Jini™ Network Technology
  - Java3D

Plan

- ✓ Pourquoi Java ?
- ✓ Syntaxe du langage
- ✓ Classes et objets en Java
- ✓ Héritage, interfaces
- ✓ Paquetages, exceptions
- ✓ Architecture client/serveur (RMI)
- ✓ Java et son environnement