M2207
Consolidation des bases de la programmation

Manuel Munier

Département RT
IUT des Pays de l'Adour
manuel.munier@univ-pau.fr

Plan du cours

Plan du cours

Ce que dit le PPN

- Objectifs du module
  - → Proposer une solution logicielle orientée objet conforme à un cahier des charges
- Compétences visées
  - → Concevoir une application sous forme d'objets et de relations
  - → Développer des applications client-serveur dans un langage orienté objet
- Pré-requis
  - → M1207 : Bases de la programmation

Ce que dit le PPN

- Contenus
  - Principes de la programmation orientée objet
  - Concept de l'héritage simple
  - Mécanismes de gestion d'erreurs
  - Architectures client-serveur
  - Mettre en œuvre un service serveur mono-utilisateur
  - Mettre en œuvre un client interrogeant un service serveur
- Volume horaire : 30h
  - → Cours 6h / TD 6h / TP 15h
    (y compris exam & éval TP !!! ☹)
- Mots clés
  - langage objet, exceptions, modèle client-serveur, socket

Ce que l'on va (essayer) de faire...

- Contenu de ce module
  - ✓ Python ⇝ Java
  - ✓ Principes de la programmation orientée objet
  - ✓ Concept de l'héritage simple
  - ✓ Client-serveur & exceptions ⇒ 1 exo d'initiation en TP (sous forme de "recette de cuisine")
- **Ce que nous ne ferons pas ! ! !**
  - ✗ Algorithmique : boucles, tests, tableaux, fonctions,...
  - ✗ Ligne de commande : gestion fichiers & répertoires,...

  ⇒ `if (lacunes>0) then do_révisions(M1207);`

Pourquoi Java ?

- Langage réellement orienté objet
  (contrairement à Python ou C++ où l'on peut mélanger POO & prog. structurée ⇒ moins "propres" pour apprendre la POO)
- Compilateur strict & messages d'erreur clairs à l'exécution
  ⇒ *programmes plus faciles à corriger*
- « Write once, run everywhere »
  - JVM = Java Virtual Machine
  - code indépendant du matériel, de l'OS, de la JVM
  - environnement (très) complet (bibliothèques de classes)
  - langage très utilisé dans les entreprises

  ⇝ applets, servlets, android, midlets, cartes à puce,...

Comment ?

- Très peu de temps (3 cours d'1h30 ☹), donc :
  - pas le temps de faire de la théorie ⇒ apprentissage des concepts objet par la pratique, sur des exemples
  - syntaxe Java ⇒ au fur et à mesure des exercices
  - client-serveur & exceptions ⇒ 1 exo d'initiation en TP

Plan du cours

Le langage Java

- Syntaxe (très) proche du C/C++
- Qq différences notables avec Python
  - c'est un langage compilé $\Rightarrow$ code source $\rightsquigarrow$ exécutable
  - les variables sont typées et doivent être déclarées
  - les blocs sont délimités par { et }
  - point d'entrée (`main`) obligatoire
  - qq différences syntaxiques sur boucles `for` et tableaux

Le langage Java : compilation

Java est un langage compilé

Le **code source** doit être **compilé** pour être transformé en **code exécutable**.

- Python est un langage interprété

  un script Python est lu ligne par ligne, lesquelles sont exécutées au fur et à mesure

- Java
  1. le code source est un fichier texte (extension `.java`)
  2. le compilateur traduit le source en bytecode (fichier `.class`)

     ex : `javac Prog.java`
  3. ce bytecode est ensuite exécuté par la JVM

     ex : `java Prog`

Le langage Java : variables & types

Java est un langage fortement typé

Chaque **variable** doit être **déclarée** avec son **type** avant de pouvoir être utilisée.

- Exemples :
  - `int i; // i est un entier`
  - `double val; // val est un réel`
  - `char c; // c est un caractère`
  - `boolean bool; // bool vaut "true" ou "false"`
  - `String nom; // nom est une chaîne de caractères`

NB Une variable ne peut pas changer de type en cours d'exécution (contrairement à Python).

Le langage Java : blocs

Déclaration des blocs de code

Les blocs de code doivent être explicitement déclarés via les délimiteurs { et }.

NB Contrairement à Python, l'indentation ne suffit pas

- ⇝ il faut penser à mettre des { et } pour délimiter les blocs et les sous-blocs
- ⇝ mais le compilateur peut ainsi détecter des erreurs "bêtes" (ex : un espace en trop ou en moins en début de ligne. . .)

Le langage Java : `main`

Point d'entrée du programme

Une classe[a] contient un ensemble de méthodes[b]. L'une d'entre elles joue un rôle particulier : la méthode `main`. C'est le point d'entrée du programme, i.e. là où commence l'exécution.

---

a. un "programme"
b. des "fonctions"

NB En Python aussi il y a une fonction `main`. Elle se nomme `__main__` et, par défaut, c'est "ce qui se trouve à la fin du programme".

Le langage Java : 1er exemple

- Transition syntaxique de Python à Java
  - → `main` (dans une classe), délimiteurs de blocs, déclaration des variables, affichage à l'écran

Python

```
a=5
b=3
add=a+b
print 'résultat=',add
```

Java

```
class MonProg {
  public static void main(String[] args) {
    int a=5;
    int b=3;
    int add=a+b;
    System.out.println("résultat="+add);
  }
}
```

Le langage Java : 1er exemple

- Concrètement, comment ça se passe sur machine ?
  1. **édition** : via votre éditeur de texte préféré, créez le fichier `MonProg.java`
  2. **compilation** : `javac MonProg.java`
     ⇝ le compilateur crée le fichier `MonProg.class` (bytecode)
  3. **exécution** : `java MonProg`
     ⇝ la JVM charge le bytecode se trouvant dans le fichier `MonProg.class` et exécute la méthode `main`

NB Ceci est valable :

- peu importe le système d'exploitation et le matériel
- peu importe les "outils" utilisés (éditeur, IDE,...)
- peu importe le JDK et la JVM utilisés

Le langage Java : boucle `while`

- Transition syntaxique de Python à Java
  - → la syntaxe du `while` est quasiment la même

Python

```
add=0
i=0
while (i<10):
  add=add+i
  i=i+1
print 'résultat=',add
```

Java

```
class MonProg {
  public static void main(String[] args) {
    int add=0;
    int i=0;
    while (i<10) {
      add=add+i;
      i++;
    }
    System.out.println("résultat="+add);
  }
}
```

Le langage Java : boucle for

- Transition syntaxique de Python à Java
  - → la syntaxe du `for` est légèrement différente, mais le fonctionnement reste identique d'un point de vue algorithmique

Python

```
add=0
for i in range(10):
  add=add+i
print 'résultat=',add
```

Java

```
class MonProg {
  public static void main(String[] args) {
    int add=0;
    for (int i=0;i<10;i++) {
      add=add+i;
    }
    System.out.println("résultat="+add);
  }
}
```

Le langage Java : tableaux

- Transition syntaxique de Python à Java
  - → la syntaxe des tableaux est très similaire

Python

```
tab=[3,12,16,11,5]
add=0
i=0
while (i<5):
  print 'Valeur ',i,'=',tab[i]
  add=add+tab[i]
  i=i+1
print 'somme=',add
```

Java

```
class MonProg {
  public static void main(String[] args) {
    int[] tab={3,12,16,11,5};
    int add=0;
    int i=0;
    while (i<5) {
      System.out.println("Valeur "+i+"="+tab[i]);
      add=add+tab[i];
      i++;
    }
    System.out.println("somme="+add);
  }
}
```

Le langage Java : tableaux

- Beaucoup de similitudes avec Python :
  - notation avec [ ], 1ère case à l'indice 0
  - possibilités de faire des tableaux de tableaux
  - `tab.length` retourne la taille **maximale** du tableau
- Qq petites différences par rapport à Python :
  - la taille d'un tableau est fixée **une fois pour toute** lors de son initialisation [1]
    - `int[] tab={3,12,16,11,5};` ⇝ tableau de 5 entiers (0 à 4)
    - `int[] tab; tab=new int[10];` ⇝ tableau de 10 entiers
  - toutes les cases du tableau **doivent** être du même type[1]
  - pas de fonctions[1] `append`, `sort`,...
  - index obligatoirement de type entier (i.e. pas de tableaux associatifs [2])

1. sinon utiliser des classes telles que `Vector`
2. il faudra utiliser des classes spécifiques (ex : `Hashtable`)

Le langage Java : affichage console

- Affichage d'un message dans le terminal
  - → en Java, uniquement pour les types simples : `int`, `double`, `boolean`, `char`,... (pas les tableaux)
  - → il existe un mécanisme supplémentaire pour les classes [3]

Python

```
valeur=...
print 'message=',valeur
```

Java

```
class MonProg {
  public static void main(String[] args) {
    ...
    System.out.println("message="+valeur);
  }
}
```

3. pour les connaisseurs : redéfinition de la méthode `toString`

Le langage Java : saisie clavier

- Lecture de valeurs depuis le terminal
  - → en Java, uniquement pour les types simples : `int`, `double`, `boolean`, `char`,... (pas les tableaux)
  - → pour les tableaux, à vous de faire une boucle et de lire les valeurs une par une ☺

Python

```
a=input('Valeur entière: ')
b=input('Valeur réelle: ')
s=input('Chaîne de caractères: ')
tab=input('Tableau de valeurs: ')
```

Java

```
import java.util.Scanner;
class MonProg {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Valeur entière: ");
    int a=sc.nextInt();
    System.out.print("Valeur réelle: ");
    double b=sc.nextDouble();
    System.out.print("Chaîne de caractères: ");
    String s=sc.nextLine();
  }
}
```

Le langage Java : 2ème exemple (sans fonction)

Python

```
# Somme et moyenne des valeurs d'un tableau
# 1ere version: sans fonction

tab = input('Tableau de valeurs: ')

somme = 0
for i in range(len(tab)):
    somme = somme + tab[i]

moyenne = somme / len(tab)

print 'somme   =',somme
print 'moyenne =',moyenne
```

Java

```
// Somme et moyenne des valeurs d'un tableau
// 1ere version: sans fonction

import java.util.Scanner;
class MonProg {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    double[] tab = new double[100];

    // Lecture du tableau
    System.out.print("Nb valeurs: ");
    int nb=sc.nextInt();
    for (int j=0;j<nb;j++) {
      System.out.print("Valeur réelle "+j+": ");
      tab[j]=sc.nextDouble();
    }

    double somme=0.0;
    for (int i=0;i<nb;i++) {
      somme=somme+tab[i];
    }

    double moyenne=somme/nb;

    System.out.print("somme   ="+somme);
    System.out.print("moyenne ="+moyenne);
  }
}
```

Le langage Java : fonctions

- Là, Java va commencer à diverger de Python !
- Reste qq similitudes :
  - fonctionnement général d'un appel de fonction
  - paramètres formels & paramètres effectifs
  - la fonction retourne au maximum 1 valeur
  - variables locales aux fonctions

"Fonctions" en Java

- Il n'est pas prévu d'avoir beaucoup de fonctions
  ⇒ on aura plutôt **des méthodes sur des objets**
- La signature d'une fonction/méthode doit être précise
  ⇒ comme pour les variables, il faut indiquer **le type de chaque paramètre** ainsi que **le type de retour**

Le langage Java : 2ème exemple (avec fonction)

Python

```
# Somme et moyenne des valeurs d'un tableau
# 2eme version: avec fonction

def sommeTab(tablo):
    somme = 0
    for i in range(len(tablo)):
        somme = somme + tablo[i]
    return somme

def moyenneTab(t):
    return sommeTab(t)/len(t)

# ----- Programme principal -----

tab = input('Tableau de valeurs: ')
print 'somme   =',sommeTab(tab)
print 'moyenne =',moyenneTab(tab)
```

Java

```
// Somme et moyenne des valeurs d'un tableau
// 2eme version: avec fonction

import java.util.Scanner;
class MonProg {
  private static double sommeTab(double[] tablo, int n) {
    double somme=0.0;
    for (int i=0;i<n;i++) {
      somme=somme+tablo[i];
    }
    return somme;
  }

  private static double moyenneTab(double[] t, int n) {
    return sommeTab(t,n)/n;
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    double[] tab = new double[100];

    // Lecture du tableau
    System.out.print("Nb valeurs: ");
    int nb=sc.nextInt();
    for (int j=0;j<nb;j++) {
      System.out.print("Valeur réelle "+j+": ");
      tab[j]=sc.nextDouble();
    }

    System.out.print("somme   ="+sommeTab(tab,nb));
    System.out.print("moyenne ="+moyenneTab(tab,nb));
  }
}
```

Le langage Java : fonctions

- Remarques :
  - visibilité
    - `private` ⇒ interne au programme
    - `public` ⇒ accessible à tous
  - `static` ⇝ "c'est comme ça" pour les fonctions... ☺
  - type de retour
    - `void` ⇒ pas de résultat attendu
    - sinon le `return` est obligatoire
  - pas d'ordre imposé (mais autant garder les bonnes habitudes)

Plan du cours

Concept d'objet : motivations

- Il est parfois nécessaire de **regrouper** plusieurs informations dans une seule variable
  - 3 entiers `jour`, `mois`, `année` pour représenter une date
  - `nom`, `prénom`, `date_naissance`,... pour une personne
  - `ref`, `libellé`, `prix`, `catégorie`,... pour un article
  - `point` (lui même composé de `X` et `Y`), `longueur`, `largeur` pour un rectangle

⇒ On définit pour cela des structures de données
  - ⇝ tuple en Python
  - ⇝ `record` en Pascal
  - ⇝ `struct` en langage C

Concept d'objet : motivations

Pb n°1 Si les données sont mieux structurées, cela reste néanmoins très "statique"

- les données d'un côté
- les traitements (fonctions) en "vrac" dans le programme

Pb n°2 Du coup, les fonctions doivent connaître la structure (interne) de ces données pour pouvoir les manipuler

- `lendemain(date)`
- `ajouter(date,nb_jours)`
- `surfaceRect(rectangle)`, `surfaceCercle(cercle)`,...
- `moyenne(tableau,nb_cases_remplies)`
- `ajouter(tableau,nb_cases_remplies,valeur)`

Concept d'objet

Définition d'un objet

L'idée consiste à **encapsuler** au sein d'une même entité à la fois

→ les données (appelées **attributs**)

→ les traitements (appelés **méthodes**)

- Les méthodes ont accès à tous les attributs de l'objet sur lequel elles s'exécutent (en plus de leurs paramètres)
- Exemples :
  - `date.lendemain()`
  - `date.ajouter(nb_jours)`
  - `rectangle.surface()`, `cercle.surface()`,...
  - `tableau.moyenne()`, `tableau.ajouter(valeur)`

Concept d'objet

Abstraction de données

Seules les méthodes d'un objet ont accès aux attributs de cet objet.

- Les attributs sont "masqués"
  - ⇒ ils ne sont pas accessibles de l'extérieur
  - ⇒ l'objet ne peut être manipulé qu'au travers de ses méthodes

⇒ Modularité

- les objets peuvent être testés individuellement
- un changement sur les attributs n'impacte pas l'extérieur (à condition que les signatures des méthodes restent identiques)
- les algorithmes sont plus "clairs" ☺

Concept d'objet : exemple n°1 (Point)

- Nous voulons définir des objets "points" composés de 2 coordonnées X et Y

v1 Version "basique" : un simple tuple... pas vraiment POO !!!

Classe Point

```
class Point {
  // Attributs
  public double X,Y;
}
```

Programme de test

```
class MonProg {
  public static void main(String[] args) {
    // Déclaration d'une réf vers un (futur) objet Point
    Point pt;

    // Instanciation, i.e. création d'un nouvel objet
    pt = new Point();

    // Utilisation de l'objet (via la réf obtenue)
    pt.X = 10.5;
    pt.Y = 5.4;
    double alpha = pt.X;
    double beta  = pt.Y;
  }
}
```

Concept d'objet : instanciation

- En POO il faut impérativement distinguer 2 choses :
  - l'**objet** lui même
    - ⇝ *une zone de la mémoire contenant les attributs, les méthodes, sa classe (~type de l'objet), etc...*
  - une **référence** sur cet objet
    - ⇝ *"l'adresse" de cette zone mémoire (~pointeur)*
    - ⇝ *plusieurs références (variables différentes) peuvent désigner le même objet (même "adresse" comme valeur)*

Instanciation

Une **instance** est un objet créé à partir d'une classe par un mécanisme appelé **instanciation** et matérialisé par l'opérateur `new`.

Concept d'objet : exemple n°1 (Point)

v2 Abstraction de données + méthodes d'accès

Classe Point

```
class Point {
  // Attributs
  private double X,Y;

  // Méthodes d'accès
  public double getX() { return X; }
  public void setX(double a) { X=a; }
  public double getY() { return Y; }
  public void setY(double b) { Y=b; }
}
```

Programme de test

```
class MonProg {
  public static void main(String[] args) {
    // Déclaration d'une réf vers un (futur) objet Point
    Point pt;

    // Instanciation, i.e. création d'un nouvel objet
    pt = new Point();

    // Utilisation de l'objet (via la réf obtenue)
    pt.setX(10.5);
    pt.setY(5.4);
    double alpha = pt.getX();
    double beta  = pt.getY();
  }
}
```

Constructeurs

- Modularité & tests (GL) $\rightsquigarrow$ une méthode prend un objet "stable" et le rend dans un état "stable"

Pb Quel est l'état[4] de l'objet lors de son instanciation ?

Constructeur

Un **constructeur** est une méthode particulière qui est invoquée automatiquement lors de l'instanciation d'un objet. Son objectif est d'initialiser tous les attributs de l'objet.

- En Java, un constructeur porte le même nom que la classe

4. valeurs de ses attributs

Constructeurs : exemple n°1 (Point)

v3 Notion de constructeur pour initialiser "proprement" l'objet

Classe Point

```
class Point {
  // Attributs
  private double X,Y;

  // Constructeur
  public Point() {
    X=Y=0.0;
  }

  // Méthodes d'accès
  public double getX() { return X; }
  public void setX(double a) { X=a; }
  public double getY() { return Y; }
  public void setY(double b) { Y=b; }
}
```

Programme de test

```
class MonProg {
  public static void main(String[] args) {
    // Déclaration d'une réf vers un (futur) objet Point
    Point pt;

    // Instanciation, i.e. création d'un nouvel objet
    pt = new Point();

    // Utilisation de l'objet (via la réf obtenue)
    pt.setX(10.5);
    pt.setY(5.4);
    double alpha = pt.getX();
    double beta  = pt.getY();
  }
}
```

Constructeurs : exemple n°1 (Point)

v3.1 Ajout de constructeurs paramétrés

Classe Point

```
class Point {
  // Attributs
  private double X,Y;

  // Constructeurs
  public Point() {
    X=Y=0.0;
  }

  public Point(double a, double b) {
    X=a;
    Y=b;
  }

  // Méthodes d'accès
  public double getX() { return X; }
  public void setX(double a) { X=a; }
  public double getY() { return Y; }
  public void setY(double b) { Y=b; }
}
```

Programme de test

```
class MonProg {
  public static void main(String[] args) {
    Point pt = new Point(10.5, 5.4);
    double alpha = pt.getX();
    double beta  = pt.getY();
  }
}
```

Constructeurs : exemple n°1 (Point)

v3.2 Un constructeur peut appeler un autre constructeur

Classe Point

```
class Point {
  // Attributs
  private double X,Y;

  // Constructeurs
  public Point() {
    this(0.0, 0.0);
  }

  public Point(double a, double b) {
    X=a;
    Y=b;
  }

  // Méthodes d'accès
  public double getX() { return X; }
  public void setX(double a) { X=a; }
  public double getY() { return Y; }
  public void setY(double b) { Y=b; }
}
```

Programme de test

```
class MonProg {
  public static void main(String[] args) {
    Point pt = new Point(10.5, 5.4);
    double alpha = pt.getX();
    double beta  = pt.getY();
  }
}
```

Remarques

- En POO, toute la "mécanique" est cachée dans les objets
  - organisation des attributs
  - algorithmes des méthodes
- Les objets "extérieurs" doivent utiliser les "services" proposés par les objets
  - → constructeurs pour instancier des objets
  - → méthodes (publiques) pour consulter / modifier les objets

⇒ "Propreté" d'un point de vue GL
  - traitements locaux
  - interfaces bien identifiées
  - impact d'un changement minimisé

    Ex *modification des spécifications, ajout de fonctionnalités, correction de bug,...*

Ramasse-miettes

- Java dispose d'un ramasse-miettes (ou *garbage collector*) qui s'occupe de supprimer les objets qui ne sont plus utilisés[5]
- 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*

⇒ En Java, **plus de problème de référence invalide**[6]

- Vous ne savez pas quand est déclenché le ramasse-miettes :
  - quand il n'y a plus assez de mémoire
  - quand le programme a du temps libre

⇒ Ce n'est plus votre problème : "ça fonctionne"

5. équivalent du `free` en langage C
6. référence sur un objet qui a été détruit

Exemple n°2 (Bulletin)

Classe Bulletin

```
class Bulletin {
  // Attributs
  private double[] notes;
  private int[]    coefs;
  private int      nb;

  // Constructeur
  public Bulletin() {
    notes = new double[50];
    coefs = new int[50];
    nb    = 0;
  }

  // Méthodes d'accès
  public int nbNotes() { return nb; }

  public void ajouterNote(double n, int c) {
    notes[nb] = n;
    coefs[nb] = c;
    nb++;
  }

  public double calculerMoyenne() {
    double sommeNotes = 0.0;
    int    sommeCoefs = 0;
    for (int i=0; i<nb; i++) {
      sommeNotes += notes[i]*coefs[i];
      sommeCoefs += coefs[i];
    }
    return sommeNotes/sommeCoefs;
  }
}
```

Programme de test

```
class MonProg {
  public static void main(String[] args) {
    Bulletin munier = new Bulletin();
    munier.ajouterNote(19.5,3);
    munier.ajouterNote(18,2);
    munier.ajouterNote(20,4);

    Bulletin abenia = new Bulletin();
    abenia.ajouterNote(4,3);
    abenia.ajouterNote(8.5,2);
    abenia.ajouterNote(6,4);

    System.out.println("munier -> "+munier.calculerMoyenne()+"/20");
    System.out.println("abenia -> "+abenia.calculerMoyenne()+"/20");

    System.out.println("That's all folks ;-)");
  }
}
```

Méthodes

- Les méthodes sont des "fonctions" qui s'exécutent sur des objets
  - une méthode a donc accès à tous les attributs de l'objet sur lequel elle s'exécute (en plus de ses paramètres)

  NB (idéalement) il n'y a pas de "fonctions globales" ni de "variables globales" en POO $\rightsquigarrow$ d'ailleurs impossible en Java

- Les méthodes sont invoquées sur des objets, via leurs références, en utilisant l'opérateur .

  ```
  référence.méthode(paramètres)
  ```

Méthodes

- En Java, chaque méthode a un nombre fixe de paramètres
- Une méthode a un type de retour (`void` si pas de résultat)
  - ⚠ *en Java, un constructeur n'a pas de type de retour !*
- Une méthode retourne un résultat via une instruction `return`
  - ⚠ *le `return` est obligatoire pour une méthode non `void`*
  - ⇒ *le compilateur va vérifier tous les "chemins" permettant de terminer la méthode*
- En Java, tous les paramètres sont passés par valeur
  - ⚠ *pour un objet, c'est **la référence** qui est passée par valeur, **pas l'objet** ⇒ la méthode peut modifier cet objet via les méthodes de cet objet !*

Méthodes : surcharge

- 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 (formels) diffèrent
- Résolution de la surcharge
  - effectuée par le compilateur
  - en fonction du nombre et du type des paramètres (effectifs)

Méthodes : surcharge

- Exemple de surcharge (pas d'ambiguïté possible) :
  - identificateur sans (...) ⇒ attribut
  - identificateur avec (...) ⇒ méthode
    - ⇝ invocation avec 1 paramètre réel ⇒ $1^{\text{ère}}$ méthode
    - ⇝ invocation sans paramètre ⇒ $2^{\text{ème}}$ méthode

Classe Rectangle

```
class Rectangle {
  private double longueur;

  void longueur(double valeur) {
    longueur = valeur;
  }

  double longueur() {
    return longueur;
  }
}
```

Synthèse

- En POO, un programme est un ensemble de petites entités autonomes (les **objets**) qui interagissent et communiquent par messages (les **appels de méthodes**)

- L'accent est mis sur l'autonomie de ces entités et sur les traitements locaux

$\Rightarrow$ Programmer avec des objets nécessite un changement d'état d'esprit de la part du programmeur

Synthèse

- 3 critères de qualité en Génie Logiciel
  - **fiabilité** : un composant fonctionne dans tous les cas de figure
  - **extensibilité** : on peut ajouter de nouvelles fonctionnalités sans modifier l'existant
  - **réutilisabilité** : les composants peuvent être réutilisés (en partie ou en totalité) pour construire de nouvelles applications

$\Rightarrow$ 1 mot : **modularité**

Synthèse

- Cette modularité a un coût
  - → à partir du cahier des charges, il faut identifier les objets

- En amont de la phase de programmation ont ainsi été développées des méthodes d'analyse et/ou de conception orientées objet
  - Ex UML (*the Unified Modeling Language*)

- **Ne concerne pas le module M2207 ☺**
  - ⇝ les sujets préciseront les objets/méthodes à programmer
  - ⇝ UML sera abordé dans le module M4207C (au S3) "Application informatique dédiée aux R&T"

Plan du cours

Concept d'héritage

Définition

L'**héritage** est un mécanisme permettant à une nouvelle classe de posséder automatiquement les attributs et les méthodes de la classe dont elle **dérive**.

- On parle de **classe mère** (ou **super-classe**) et de **classe fille** (ou **sous-classe**)

Ex Un pixel est point "normal" (i.e. 2 coordonnées `X` et `Y`) mais avec, en plus, un attribut `color`

⇒ on dérive la classe `Pixel` à partir de la classe `Point`

NB Module M2207 ⇝ héritage **simple** entre **classes**

Concept d'héritage : exemple n°1 (Pixel)

- On indique que la classe `Pixel` hérite de la classe `Point` via la directive `extends Point`

Classe Point

```
class Point {
  // Attributs
  private double X,Y;

  // Constructeurs
  public Point() {
    this(0.0, 0.0);
  }

  public Point(double a, double b) {
    X=a;
    Y=b;
  }

  // Méthodes d'accès
  public double getX() { return X; }
  public void setX(double a) { X=a; }
  public double getY() { return Y; }
  public void setY(double b) { Y=b; }
}
```

Classe Pixel

```
class Pixel extends Point {
  // Attributs
  private int color;

  // Constructeurs
  public Pixel() {
    super();
    color = -1;
  }

  public Pixel(double a, double b, int c) {
    super(a,b);
    color = c;
  }

  // Méthodes d'accès
  public int getColor() { return color; }
  public void setColor(int c) { color=c; }
}
```

Concept d'héritage : exemple n°1 (`Pixel`)

- Toutes les instances de la classe `Pixel` ont implicitement :
  - les attributs `X` et `Y` (définis dans la classe `Point`)
  - les méthodes `getX`, `setX`, `getY` et `setY` (héritées de `Point`)

⇒ On ne caractérise que les différences avec la classe mère

- En Java, les constructeurs devant porter le même nom que la classe, il est nécessaire de les réécrire
  - → `super(...)` permet d'invoquer un constructeur de la classe mère ; ce doit être la 1ère instruction
  - → indispensable pour initialiser les attributs `private` de la classe mère ! (présents, mais inaccessibles dans la sous-classe)

Objectifs

1. **Enrichissement** : une classe `A` est "incomplète"
   - → on définit une classe `B` qui hérite de `A` et on lui ajoute de nouvelles fonctionnalités (attributs et/ou méthodes)
   - → on étend les fonctionnalités de la classe existante
     ⇒ **enrichissement** d'attributs et de méthodes (ex : `Pixel`)

2. **Substitution** : une classe `A` est satisfaisante, "sauf dans certains cas particuliers"
   - → on définit une classe `B` qui hérite de `A` et on y redéfinit certaines méthodes de `A`
   - → on parle de **substitution** ⇒ pour les instances de `B`, les méthodes redéfinies dans `B` sont **prioritaires** par rapport à celles héritées de `A`
   - ⚠ on modifie le **corps** de la méthode, pas sa **signature** !

Héritage : exemple n°1 (`Pixel`)

- Exemple n°1 : classes `Point` et `Pixel`
  - la classe `Point` définit 2 attributs `X` et `Y`
  - la sous-classe `Pixel` hérite de ces 2 attributs
  - la sous-classe `Pixel` ajoute un 3ème attribut `color`

  $\Rightarrow$ enrichissement

Héritage : exemple n°2 (`Bulletin`)

- Exemple n°2 : bulletins de notes
  - la classe `BulletinSansCoef` définit 2 attributs (1 tableau de notes et 1 entier pour le nbre de notes saisies) et 2 méthodes `ajouterNote(double)` et `calculerMoyenne()`
  - la classe `BulletinAvecCoef`
    - → ajoute 1 attribut (tableau de coefs)
    - → redéfinit les 2 méthodes `ajouterNote(double)` (coef 1 par défaut) et `calculerMoyenne()`
    - → ajoute 1 méthode `ajouterNote(double,int)` (note avec coef)

  ⇒ enrichissement + substitution

Héritage : exemple n°2 (Bulletin)

Classe BulletinSansCoef

```
class BulletinSansCoef {
  // Attributs
  protected double[] notes;
  protected int nb;

  // Constructeur
  public BulletinSansCoef(int taille) {
    notes = new double[taille];
    nb = 0;
  }

  // Méthodes
  public void ajouterNote(double note) {
    notes[nb] = note;
    nb++;
  }

  public double calculerMoyenne() {
    double moy = 0.0;
    for (int i=0; i<nb; i++) {
      moy += notes[i];
    }
    return moy/nb;
  }
}
```

Classe BulletinAvecCoef

```
class BulletinAvecCoef extends BulletinSansCoef {
  // Attribut supplémentaire
  private int[] coefs;

  // Constructeur
  public BulletinAvecCoef(int taille) {
    super(taille);
    coefs = new int[taille];
  }

  // Méthodes
  public void ajouterNote(double note, int coef) {
    coefs[nb] = coef;
    super.ajouterNote(note);
  }

  public void ajouterNote(double note) {
    this.ajouterNote(note, 1);
  }

  public double calculerMoyenne() {
    double moy = 0.0;
    int    tot = 0;
    for (int i=0; i<nb; i++) {
      moy += notes[i]*coefs[i];
      tot += coefs[i];
    }
    return moy/tot;
  }
}
```

Héritage : exemple n°2 (`Bulletin`)

- Dans `BulletinSansCoef` les attributs sont définis en `protected` afin que la classe `BulletinAvecCoef` puisse y avoir accès directement
  - ⇝ si on garde `private`, seule la classe `BulletinSansCoef` peut y accéder ⇒ la classe `BulletinAvecCoef` devrait passer par des méthodes d'accès
- Les constructeurs des sous-classes doivent toujours invoquer un des constructeurs de la classe mère (via `super(...)`)
  - ⇝ c'est obligatoirement la 1ère instruction du constructeur
- Référence à soi-même (*self reference*)

`this` on commence à chercher dans le code de la sous-classe
`super` on cherche directement dans le code de la classe mère

Méthodes & classes abstraites

- Idée : définir une méthode dans une classe sans donner son implémentation
  - ⇝ on ne donne que la signature de la méthode ; pas son corps
- Pourquoi ?
  - déclarer qu'une méthode existera sur une classe et ses sous-classes (pour le compilateur)
  - mais pouvoir l'implémenter de manière différente sur chaque sous-classe (code choisi à l'exécution en fonction de la classe des objets)

Méthodes & classes abstraites

- Exemple : objets graphiques
  - La classe `ObjetGraphique` déclare 3 méthodes `afficher()`, `effacer()` et `translater(`δ`x,`δ`y)`
  - Ces 3 méthodes ne peuvent pas être implémentées sur un objet graphique "en général"
    - ⇒ méthodes définies `abstract` (i.e. incomplètes)
    - ⇒ classe `ObjetGraphique` définie `abstract` (i.e. incomplète)
  - Ces 3 méthodes seront implémentées (différemment) par chaque sous-classe `Rectangle`, `Cercle`, `Triangle`,... en fonction de leurs attributs respectifs

Exemple : ObjetGraphique & ses sous-classes

Classe ObjetGraphique

```
abstract class ObjetGraphique {
  // Méthodes abstraites (signature uniquement)
  abstract void afficher();
  abstract void effacer();
  abstract void translater(double dx, double dy);

  // Méthode "normale" (avec implémentation)
  void déplacer(double dx, double dy) {
    this.effacer();
    this.translater(dx,dy);
    this.afficher();
  }
}
```

- s'il existe au moins une méthode abstraite ⇒ la classe est abstraite
- classe abstraite ⇒ instanciation impossible (`new` refusé)

Classe Rectangle

```
class Rectangle extends ObjetGraphique {
  void afficher() {
    //...
  }

  void effacer() {
    //...
  }

  void translater(double dx, double dy) {
    //...
  }
}
```

- méthodes (re)définies ⇒ il n'y a plus de méthode abstraite ⇒ classe "normale" ("complète") ⇒ instanciation possible

Synthèse

Héritage

L'héritage induit un nouveau style de programmation : à partir de classes prédéfinies (classes "de base") on procède par raffinages successifs pour construire de nouvelles classes (plus "évoluées").

- Empêcher l'héritage ? → mot clé `final`
  - sur une classe ⇒ interdit les sous-classes
  - sur une méthode ⇒ interdit la redéfinition dans les sous-classes
  - sur une variable ⇒ interdit la modification (⇒ constante)

Synthèse

Liaison dynamique

Quand une méthode est redéfinie dans les sous-classes, le choix de la "bonne" méthode (i.e. code à exécuter) se fait à l'exécution en fonction de la classe de l'objet qui exécute la méthode.

Ex La classe `Rectangle` utilisera **sa** méthode `afficher()` ; la classe `Cercle` utilisera la sienne, etc...

Synthèse

- Remarques quant à la liaison dynamique :
  - l'héritage permet de modifier des méthodes existantes (substitution) ou d'en ajouter de nouvelles (enrichissement), **mais pas d'en supprimer**

  ⇒ si une méthode existe dans une classe, **elle existera obligatoirement dans toutes les sous-classes** (éventuellement avec une implémentation différente)

  ⇒ si une variable/paramètre est de type `ClasseA` alors on peut y stocker des références à des objets :
    - de la classe `ClasseA`
    - d'une classe `ClasseB` héritant de `ClasseA` (directement ou indirectement)

  ⇒ pratique pour mélanger des objets "de types différents" dans un même tableau par exemple

    ⇝ il suffit de les regrouper sous une super-classe commune

Plan du cours

Notion de client et de serveur

Définition

- Le **client** est celui qui demande l'exécution d'un service
  → *l'objet qui invoque une méthode*
- Le **serveur** est celui qui fournit le service
  → *l'objet qui exécute la méthode*

Ex Un objet de la classe `MonProg` (le client) invoque la méthode `calculerMoyenne()` sur un objet de la classe `Tableau` (le serveur)

⇝ la notion de client/serveur est "relative" à 1 appel de méthode

⇝ au fil de l'exécution, 1 même objet pourra alternativement jouer les rôles de client et de serveur

Notion de client et de serveur

- Mais plus généralement on parle d'architecture client-serveur lorsque les objets clients et les objets serveurs s'exécutent sur des machines différentes.
- Pour cela il faut être capable :
  - → de connaître les objets distants
  - → d'avoir la liste des services qu'ils proposent
  - → d'invoquer ces services et de passer des paramètres
  - → de récupérer les résultats
  - → de capturer les exceptions [7] levées à distance

7. Une exception est un mécanisme permettant d'interrompre "proprement" l'exécution d'un programme en cas d'erreur.

Technologies

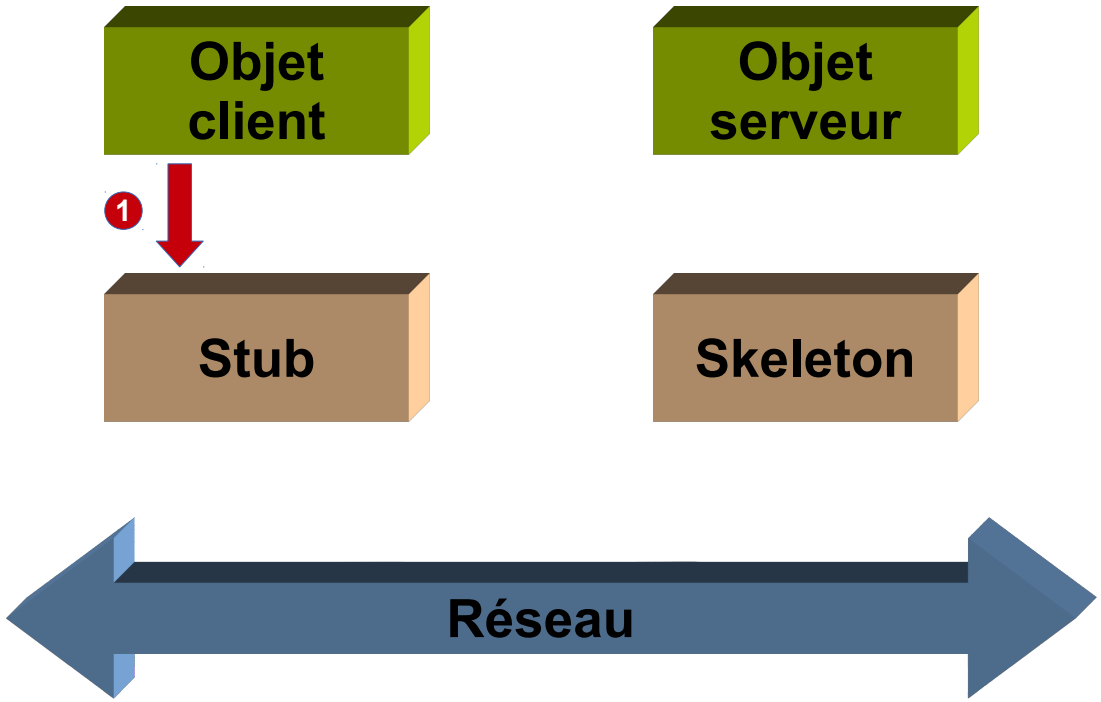
	réseau	invocation de méthodes	plates-formes différentes	annuaire	langages différents
sockets	oui				
RPC	oui	oui			
RPC + XDR	oui	oui			
RMI (Java)	**oui**	**oui**	**oui**	**oui**	
CORBA	oui	oui	oui	oui	oui
WS	**oui**	**oui**	**oui (XML)**	**oui**	**oui**

- M2207 ≡ Java ⇒ les RMI suffisent
- si besoin d'hétérogénéité ⇒ *Web Services*

RMI

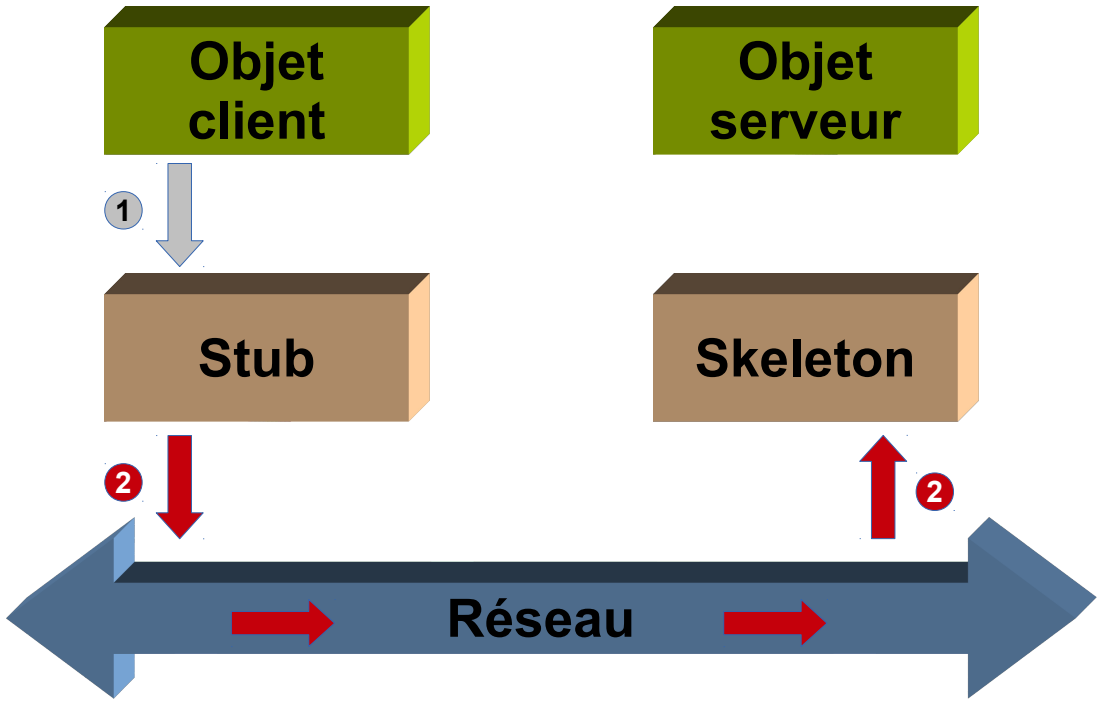
- RMI ≡ *Remote Method Invocation*
- L'objet client ne communique pas directement avec l'objet serveur, mais avec un "proxy local" : le **stub**
  - → sérialisation des paramètres
  - → construction de la requête réseau
  - → attente du résultat ou des exceptions
- De son côté, l'objet serveur est relié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

RMI



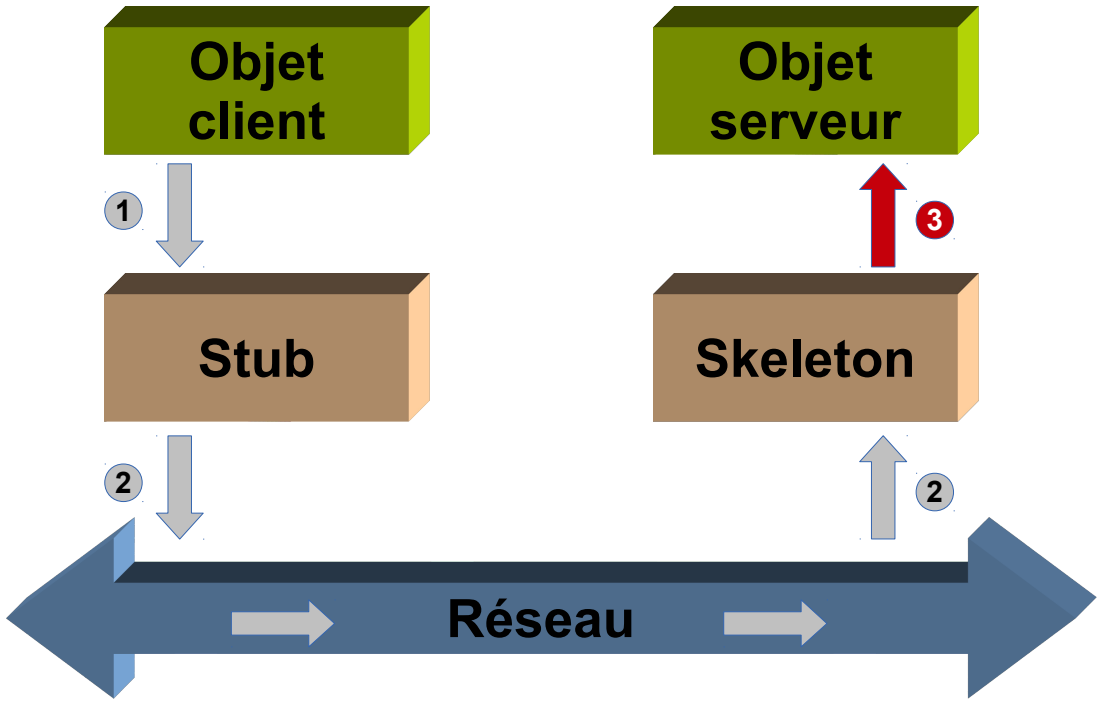
1 L'objet client invoque (normalement) la méthode sur le stub qui joue le rôle de proxy.

RMI



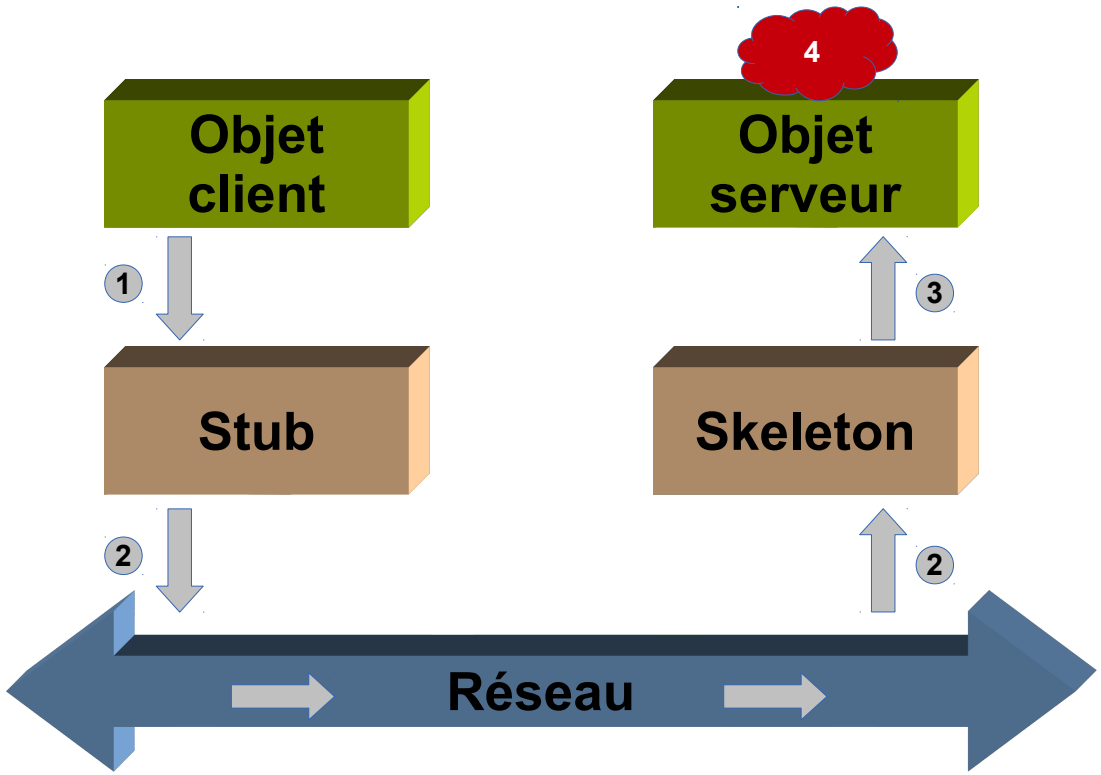
2 Le stub sérialise les paramètres et envoie la requête sur le réseau (au final, des paquets TCP/IP).

RMI



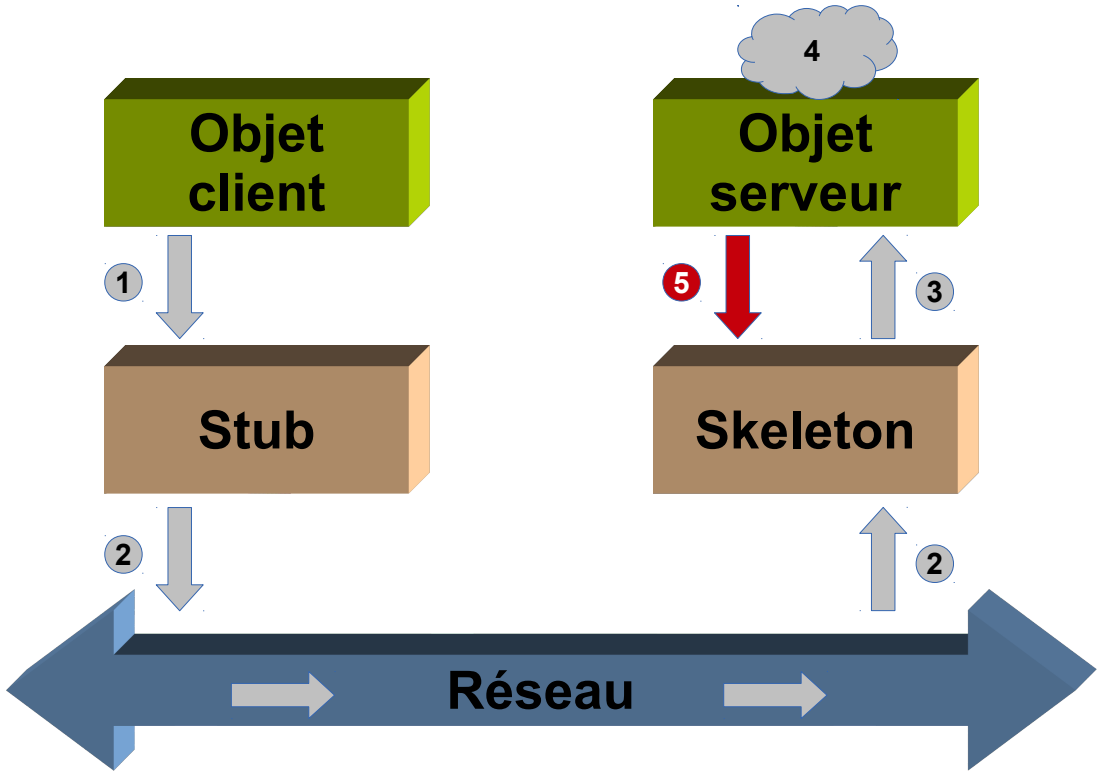
3 Le skeleton reçoit la requête, désérialise les paramètres, puis invoque la "vraie" méthode sur l'objet serveur.

RMI



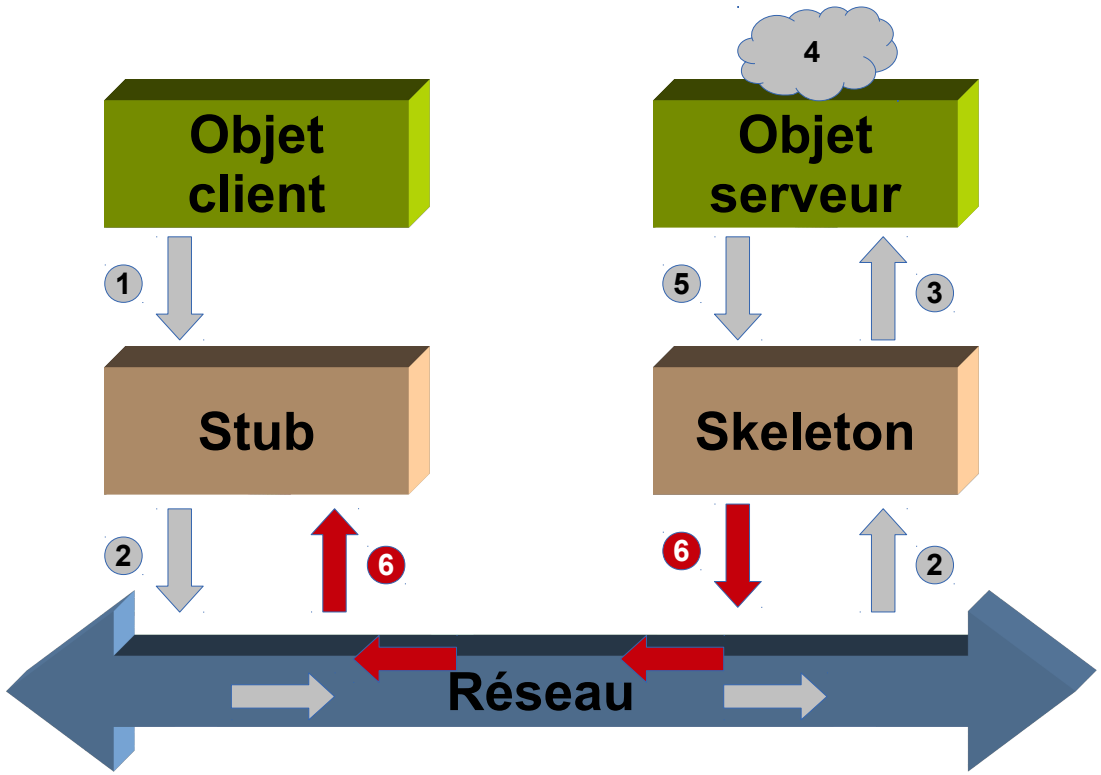
4 L'objet serveur exécute la méthode (normalement), sans même savoir s'il s'agit d'une invocation locale ou distante.

RMI



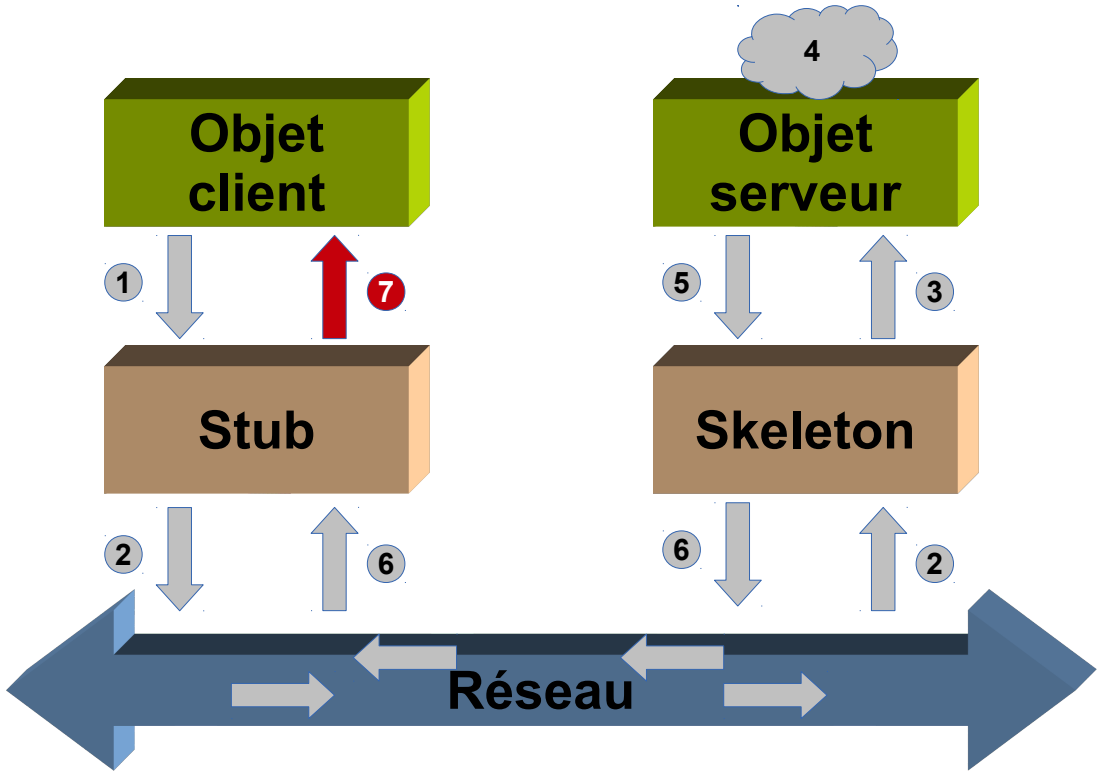
5 L'objet serveur retourne le résultat (ou l'exception) au skeleton (celui qui avait invoqué la méthode, et non l'objet client).

RMI



6 Le skeleton sérialise la résultat (ou l'exception) puis le transmet au stub qui avait envoyé la requête réseau.

RMI



7 Le stub désérialise le résultat puis les transmet à l'objet client, **comme si l'invocation avait été réalisée localement.**

RMI

- Le stub n'est qu'une interface permettant au client d'accéder, via le réseau, aux services proposés par le serveur.
- Les stubs et les skeletons sont générés automagiquement par la commande `rmic`

- Mise en œuvre $\rightsquigarrow$ voir TP... ☺