Opérations UML et leur implémentation par des méthodes Java

LIRMM / Université de Montpellier

Janvier 2017

Classes, opérations et méthodes

Partie précédente

- Définition de classes, d'attributs (partie structurelle)
- Définition de ce qu'est un objet, pas de ce qu'il fait

Méthodes et opérations

- Définissent des comportements des instances de la classe.
- Ex. Pour une classe voiture, exprimer ce que peut faire une voiture : klaxonner, fournir une assistance au parking, etc.
- Peuvent manipuler les attributs, ou faire appel à d'autres méthodes de la classe.
- Peuvent être paramétrées et retourner des résultats.
- Permettent la communication des instances par envoi de messages.

Sommaire

Opérations

- Les opérations sont des éléments du diagramme de classes représentant la dynamique du système

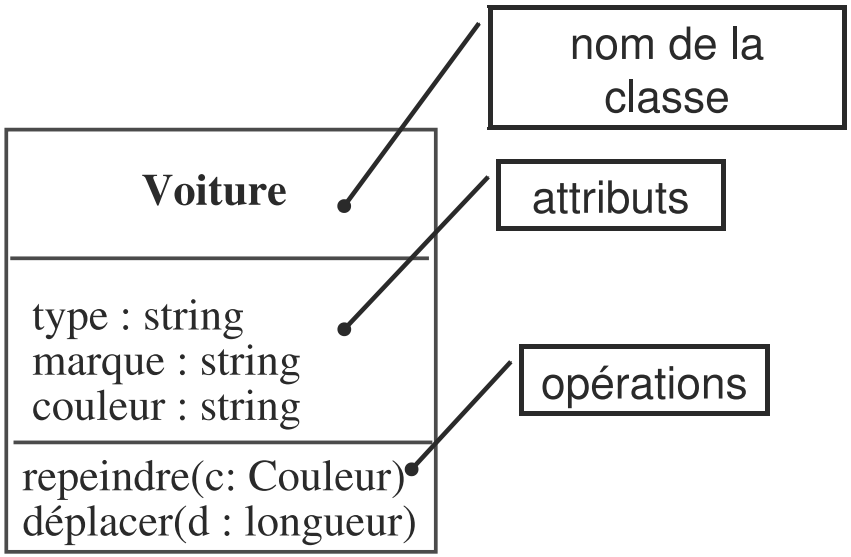


Figure : Les opérations dans les classes UML

Syntaxe

`[visibilité] nom ( lst-paramètres ) [: typeRetour]`
`[lst-propriétés]`
où la syntaxe de chaque paramètre est :
`[direction] nom : type[[multiplicité]] [= valeurParDéfaut]`
`[liste-propriétés]`
avec `direction` $\in \{in, out, inout\}$, et `multiplicité` définit une valeur (1, 2, n, ...) ou une plage de valeurs (1..*, 1..6, ...).

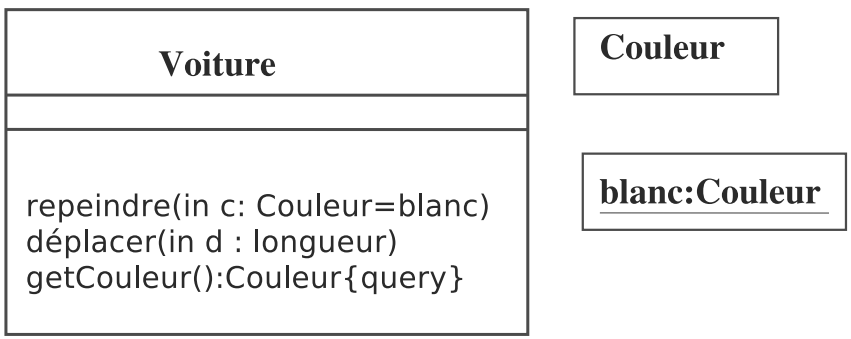


Figure : Exemple d'opérations en UML

Le nom

- Une opération a un nom
- Donner un nom portant le plus de sémantique possible
- Ex. "klaxonner", "déplacer", "repeindre", plutôt que : "o1", "o2", "op"

Visibilité

Dans les grandes lignes :

- Publique. Dénoté +. Signifie que cette opération pourra être appelée par n'importe objet
- Privée. Dénoté -. Signifie que cette opération ne pourra être appelée que par des objets instances de la même classe
- Paquetage. Dénoté $\sim$. Signifie que cette opération ne pourra être appelée que par des objets instances de classes du même paquetage.
- Protégée. Dénoté #. Signifie que cette opération ne pourra être appelée que par des objets instances de la même classe ou d'une de ses sous-classes (on verra plus tard ce que cela signifie exactement)

Paramètres

Modes de passage

in le paramètre est une entrée de l'opération, et pas une sortie : il n'est pas modifié par l'opération. C'est le cas le plus courant. C'est aussi le cas par défaut en UML.

out le paramètre est une sortie de l'opération, et pas une entrée. C'est utile quand on souhaite retourner plusieurs résultats : comme il n'y a qu'un type de retour, on donne les autres résultats dans des paramètres out.

inout le paramètre est à la fois entrée et sortie.

Propriétés

- Propriétés facultatives précisant le type d'opération
- Exemple. `{query}` : l'opération n'a pas d'effet de bord
- Propriétés entre accolades

Opérations de classe

- C'est une opération qui ne s'applique pas à une instance de la classe
- Elle peut être appelée sans avoir instancié la classe.

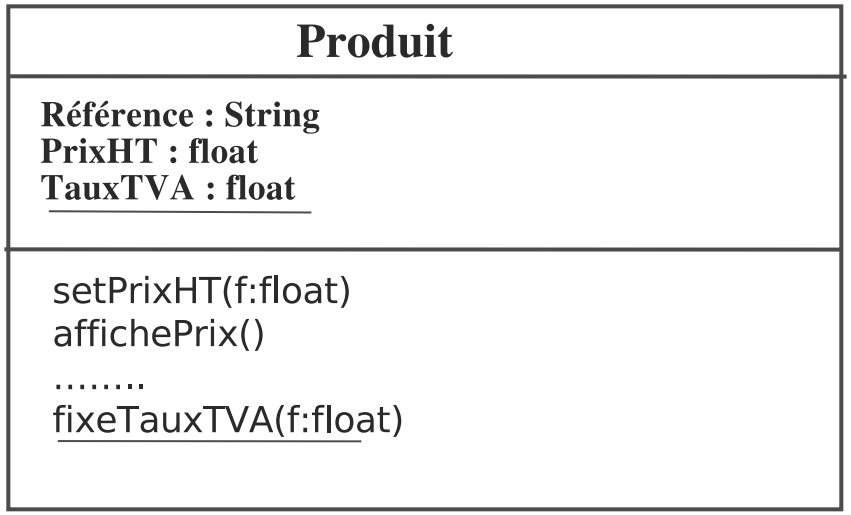


Figure : Opérations de classe

Constructeurs et destructeurs

Des opérations particulières

- Gestion de la durée de vie des instances
- Constructeur : création des instances
- Destructeur : destruction des instances

Constructeurs et destructeurs : notation

Notation

- stéréotypes <<create>> ou <<destroy>>
- stéréotypes : chaînes entre chevrons attachées aux éléments UML pour préciser la sémantique

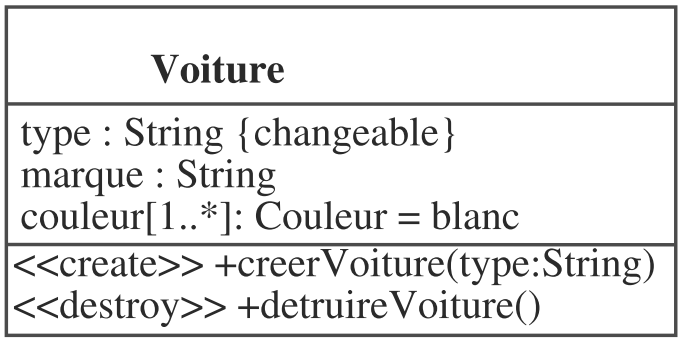


Figure : Constructeurs et destructeurs en UML

Le corps des opérations en UML

- langage d'action permettant de spécifier le comportement des opérations (Action Semantics dans Executable UML)
- Utilisation des diagrammes dynamiques pour spécifier le comportement des opérations
- Documentation avec du pseudo-code, dans une note de commentaire

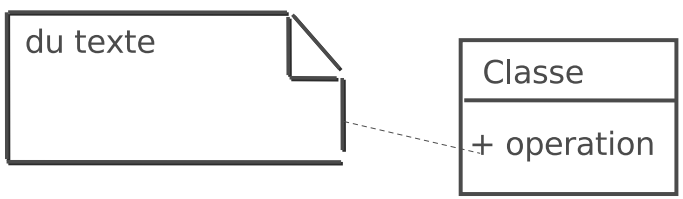


Figure : Note UML

Sommaire

Exemple

```
1  package ExemplesCours2;
2  public class Voiture
3  {
4      private String type;
5      private String marque;
6      private String couleur;
7      private static int nbrVoitures;
8      private static final int nbrRoues = 4;
9
10         public Voiture(String leType, String laMarque, String couleur){
11             type=leType;
12             marque=laMarque;
13             this.couleur=couleur;
14         }
15
16         public static int getNbrRoues(){
17             return nbrRoues;
18         }
19
20         public String getMarque(){
21             return marque;
22         }
23
24         private void setMarque(String m){
25             marque=m;
26         }
27
28         public void repeindre(String c){
29             couleur=c;
30         }
31 }
```

Déclaration de méthodes

- déclaration de constructeur : en Java, le constructeur doit avoir le même nom que la classe, et il n'y a pas de type de retour.
- pas vraiment de destructeur en Java. Il existe une méthode particulière nommée `finalize` qui est appelée quand le ramasse-miettes récupère l'espace alloué à l'objet car il n'est plus référencé.
- déclaration de méthode de classe : mot clef `static`. Appel : préfixer le nom de l'opération par le nom de la classe
- utilisation des mots clefs `private` et `public` pour définir la visibilité des méthodes
- il n'y a pas en Java la distinction entre paramètre in, out, ou inout.

Surcharge

Surcharge :

- dans une même classe
- plusieurs méthodes portant le même nom
- des signatures différentes

Exemple :

- une méthode `int add(int a, int b)`
- une méthode `float add(float a, float b)` .

Exemple

Listing 1 – Utilisation de la classe Voiture en java

```
1  package ExemplesCours2;
2  public class essaiVoiture{
3    public static void main(String[] arg){
4      Voiture v=new Voiture("C3", "Citroen", "rouge")
           ;
5      int nb=v.getNbrRoues();
6      System.out.println("Ma␣"+v.getMarque()+"␣a␣"+nb
         +"␣roues");
7    }
8  }
```

À noter ...

- la méthode étrange appelée `main` appelée par :

  ```
  > java ExemplesCours2.essaiVoiture
  ```

  - point d'entrée du programme
  - méthode statique : pas besoin de créer d'instance `essaiVoiture` pour utiliser la méthode `main`.
  - paramètre : arguments en ligne de commande (tableau de chaînes).
- la concaténation de chaînes pour l'affichage, et la traduction automatique d'entiers en chaînes.

Les accesseurs

Accès aux attributs, en lecture et en écriture
Par convention pour l'attribut `att` de type `T` :

```
1  T getAtt()
2  void setAtt(T valeur)
```

- `get` : statistiques sur les accès à l'attribut
- `set` : vérifications sur les valeurs, gestion des attributs dérivés

Les accesseurs – exemples

```
1  public class Point{
2      public static double dimension = 2 ;    //Variable de classe
3      private double x ;
4      private double y ;
5      public Point(){ //Constructeur par defaut
6          this(0,0) ;
7      }
8      public Point(double x , double y){ //Constructeur avec argument
9          this.setX(x) ;
10         this.setY(y) ;
11     }
12 //Accesseurs pour la variable x.
13     public double getX(){
14         return this.x ;
15     }
16     public void setX(double x){
17         this.x = x ;
18     }
19  //Accesseurs pour la variable y.
20     public double getY(){
21         return this.y ;
22     }
23     public void setY(double y){
24         this.y = y ;
25     }
26     public void symetrieSelonX(){
27         this.getY() = -this.getY() ;
28     }
29     public void symetrieSelonY(){
30         this.getX() = -this.getX() ;
31     }
32 }
```

Affectation, Déclaration de variables locales

- L'affectation se note =
- Dans le corps d'une méthode, on peut déclarer des variables locales. Par exemple :

```
int i;
int j=0;
Voiture v;
```

- Pas de visibilité : la portée de ces variables s'arrête à la fin de la méthode
- Pas de valeur initiale implicite

Création de nouvelles instances

Création : `new` + constructeur

```
Voiture v;
v=new Voiture("C4", "Citroen", "bleu");
```

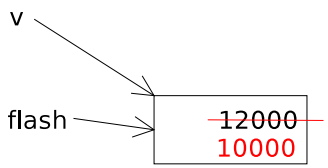
Le passage de paramètres

- Dans le corps d'une méthode, les paramètres sont comme des variables locales
- C'est comme si on avait des variables locales déclarées au début de la méthode, et qu'en début de méthode on affectait les valeurs des paramètres effectifs à ces variables locales
- Les paramètres de type simple (types de base en Java comme `int` et `boolean`, dont le nom commence par une minuscule) sont passés par valeur
- Tous les autres paramètres sont passés par référence (leur adresse est passée par valeur)

Illustration du passage de paramètres

```
public class Voiture{
  public int prix;
  public Voiture(int p){
   prix=p;
  }
}
```

```
public class Expertise{
  public void expertiser(v:Voiture){
    v.prix=10000;
 }
...
Voiture flash=new Voiture(12000);
...
expertiser(flash);
```



Pendant l'exécution de la méthode `expertiser`, `v` et `flash` désignent le même objet.

Figure : Passage de paramètres par référence

Exemple

Listing 2 – MonInt.java

```
1  package ExemplesCours2;
2  public class MonInt{
3
4    private int entier;
5
6    public MonInt(int e){
7      entier=e;
8    }
9
10   public int getEntier(){
11     return entier;
12   }
13
14   public void setEntier(int e){
15     entier=e;
16   }
17 }
```

Exemple

Listing 3 – Echange.java

```
1  package ExemplesCours2;
2  public class Echange{
3    public void fauxEchange(int a, int b){
4      System.out.println("a= "+a+" b= "+b);
5      int vi=a;
6      a=b;
7      b=vi;
8      System.out.println("a= "+a+" b= "+b);
9    }
10
11   public void pseudoEchange(MonInt a, MonInt b){
12     System.out.println("a.getEntier= "+a.getEntier()+" b.getEntier= "+b.
           getEntier());
13     int vi=a.getEntier();
14     a.setEntier(b.getEntier());
15     b.setEntier(vi);
16     System.out.println("a.getEntier= "+a.getEntier()+" b.getEntier= "+b.
           getEntier());
17   }
```

Exemple

Listing 4 – Echange.java

```
1     public static void main(String[] args){
2       int x=2, y=3;
3       System.out.println("x= "+x+" y= "+y);
4       Echange echange=new Echange();
5       echange.fauxEchange(x,y);
6       System.out.println("x= "+x+" y= "+y);
7
8       MonInt xx=new MonInt(2);
9       MonInt yy=new MonInt(3);
10      System.out.println("xx.getEntier= "+xx.getEntier()+" yy.getEntier= "
            +yy.getEntier());
11      echange.pseudoEchange(xx,yy);
12      System.out.println("xx.getEntier= "+xx.getEntier()+" yy.getEntier= "
            +yy.getEntier());
13    }
14
15  }
```

Test d'égalité sur des objets

- == teste l'égalité de référence
- la méthode equals, applicable sur n'importe quel objet, teste l'égalité de valeur

```
1  String s1="toto";
2  String s2="to";
3  s2+="to";
4  s1==s2; // false
5  s1.equals(s2); // true
```

Désignation de l'instance courante

- Receveur du message = instance courante
- Désigné par `this`
- Usage : conflit de noms ou utilisation directe

L'instruction `return`

- Retour d'un résultat
- Sortie immédiate de la méthode
- Cohérence entre valeur retournée et type de retour

Les commentaires

```
// ceci est un commentaire (s'arrête à la fin de la ligne)
/* ceci est un autre commentaire
   qui s'arrête quand on rencontre le marqueur de fin que voilà */
/** ceci est un commentaire particulier, utilisé par l'utilitaire javadoc **/
```

Affichage

On peut afficher des données sur la console grâce à une bibliothèque Java

```
System.out.println(‘‘affichage puis passage à la ligne’’);
System.out.print(‘‘affichage sans ’’);
System.out.print(‘‘passer à la ligne’’);
```

La méthode `toString()`

- Implicite dans toutes les classes
- Retourne une chaîne de caractères qui représente une instance ou son état sous une forme lisible et affichable
- Méthode par défaut : retourne une désignation de l'instance

La méthode `toString()` : exemple

Listing 5 – Personne.java

```
1  package ExemplesCours2;
2  public class Personne{
3    private String nom;
4    private int numSecu;
5
6    public String toString(){
7      String result=nom+" "+age;
8      return result;
9    }
10 }
```

Conditionnelle simple

Listing 6 – Conditionnelle en Java

```
1        if (expression booleenne) {
2          bloc1
3          }
4         else {
5          bloc2
6         }
```

- La condition doit être évaluable en true ou false et elle est obligatoirement entourée de parenthèses.
- Les points-virgules sont obligatoires après chaque instruction et interdits après }.
- Si un bloc ne comporte qu'une seule instruction, on peut omettre les accolades qui l'entourent.
- Les conditionnelles peuvent s'imbriquer.

Conditionnelle simple : exemple

Listing 7 – Conditionnelle en Java

```
1  int a =3;
2  int b =4;
3  System.out.print("Le plus petit entre "+a+" et "+b
       +" est : ");
4  if (b <a ) {
5    System.out.println(b);
6  }
7  else { System.out.println(a);
8  }
```

L'opérateur conditionnel ( ) ? ... : ...

Le : se lit *sinon*.

```
1  System.out.println( (b < a) ? b : a );
2  int c = (b < a) ? a-b : b-a ;
```

L'instruction de choix multiples

```
1  switch (expr entiere ou caractere) {
2  case i:
3  case j:
4  [bloc d'intructions]
5  break;
6  case␣k:
7  ...
8  default:
9  ...
10 }
```

- L'instruction `default` est facultative ; elle est à placer à la fin. Elle permet de traiter toutes les autres valeurs de l'expression n'apparaissant pas dans les cas précédents.
- Le `break` est obligatoire pour ne pas traiter les autres cas.

L'instruction de choix multiples : exemple

```
1  int mois, nbJours;
2  switch (mois) {
3   case 1:
4   case 3:
5   case 5:
6   case 7:
7   case 8:
8   case 10:
9   case 12:
10   nbJours = 31;
11   break;
12  case 4:
13  case 6:
14  case 9:
15  case 11:
16   nbJours = 30;
17   break;
18  case 2:
19    if ( ((annee % 4 == 0) && !(annee % 100 == 0)) || (annee % 400 == 0)
          )
20      nbJours = 29;
21    else
22      nbJours = 28;
23    break;
24  default nbJours=0;
25       }
26      }
```

Switch et énumérations

```
1  public enum Day {
2      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
3      THURSDAY, FRIDAY, SATURDAY
4  }
5
6  public class EnumTest {
7      Day day;
8
9      public EnumTest(Day day) {
10         this.day = day;
11     }
12
13     public void tellItLikeItIs() {
14         switch (day) {
15             case MONDAY:
16                 System.out.println("Mondays are bad.");
17                 break;
18
19             case FRIDAY:
20                 System.out.println("Fridays are better.");
21                 break;
22
23             case SATURDAY: case SUNDAY:
24                 System.out.println("Weekends are best.");
25                 break;
26
27             default:
28                 System.out.println("Midweek days are so-so.");
29                 break;
30         }
31     }
32 }
```

while

Syntaxe :

```
1 while (expression) {
2   bloc
3 }
```

Exemple :

```
1 int max = 100, i = 0, somme = 0;
2 while (i <= max) {
3   somme += i;       // somme = somme + i
4   i++ ;
5 }
```

do while

Syntaxe :

```
1        do
2          { bloc }
3        while (expression)
```

```
1  int max = 100, i = 0, somme = 0 ;
2  do {
3   somme += i ;
4   i++;
5  }
6  while ( i <= max );
```

for

Syntaxe :

```
1 for ( expression1; expression2 ; expression3){
2     bloc
3 }
```

- utilisée pour répéter N fois un même bloc d'instructions
- `expression1` : initialisation. Précise en général la valeur initiale de la variable de contrôle (ou compteur)
- `expression2` : la condition à satisfaire pour rester dans la boucle
- `expression3` : une action à réaliser à la fin de chaque boucle. En général, on actualise le compteur.

for : exemple

```
1  int somme = 0, max = 100;
2  for (int i =0 ; i <= max ; i++ ) {
3   somme += i ;
4  }
```

Modélisation et Programmation par Objets
HLIN406

Marianne Huchard, Clémentine Nebut

LIRMM / Université de Montpellier

2017

Sommaire

Langages et paradigmes de programmation

- paradigme de programmation : une approche pour définir, organiser, utiliser la connaissance.
- Principaux paradigmes
  - Programmation fonctionnelle, centrée sur une décomposition en fonctions (scheme, camel)
  - Programmation impérative, centrée sur une structuration “ séquentielle “ des traitements (pascal, C)
  - Programmation objet, centrée sur les données associées à leurs comportements (C++, Java)

Le raisonnement classificatoire

- Identification d'objets de base (Estelle, la voiture d'Estelle)
- Utilisation de ces objets comme des prototypes (la voiture d'Estelle vue comme une voiture caractéristique, à laquelle ressemblent les autres voitures, moyennant quelques modifications)
  ▷ modèles et langages à prototypes
- Regroupement des objets partageant des propriétés structurelles et comportementales en classes
  ▷ modèles et langages à classes

Classes

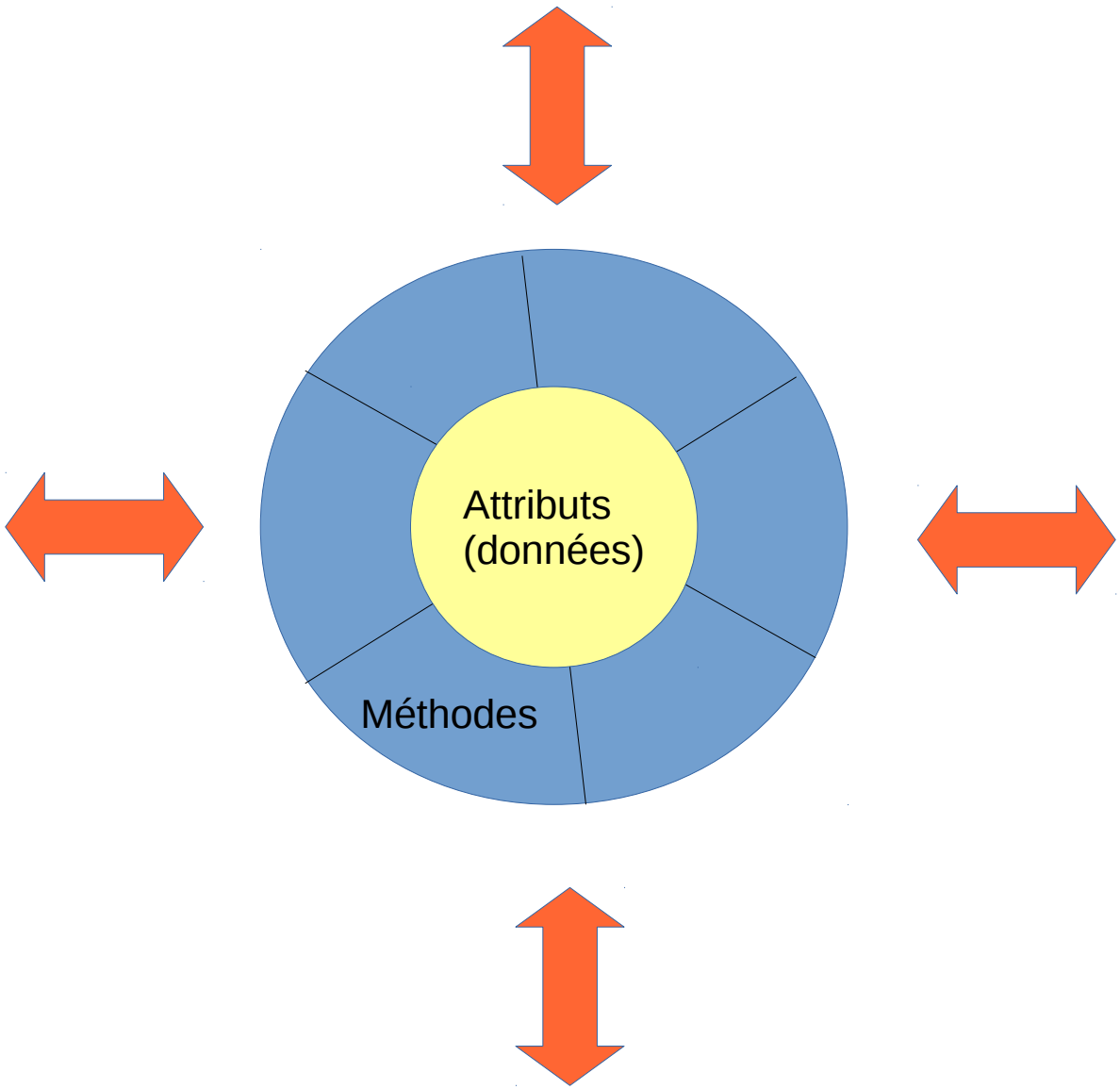
Concept du domaine sur lequel porte :

- le logiciel ▷ voiture ou compte bancaire
- le domaine du logiciel ▷ un type de données abstrait tel que la pile

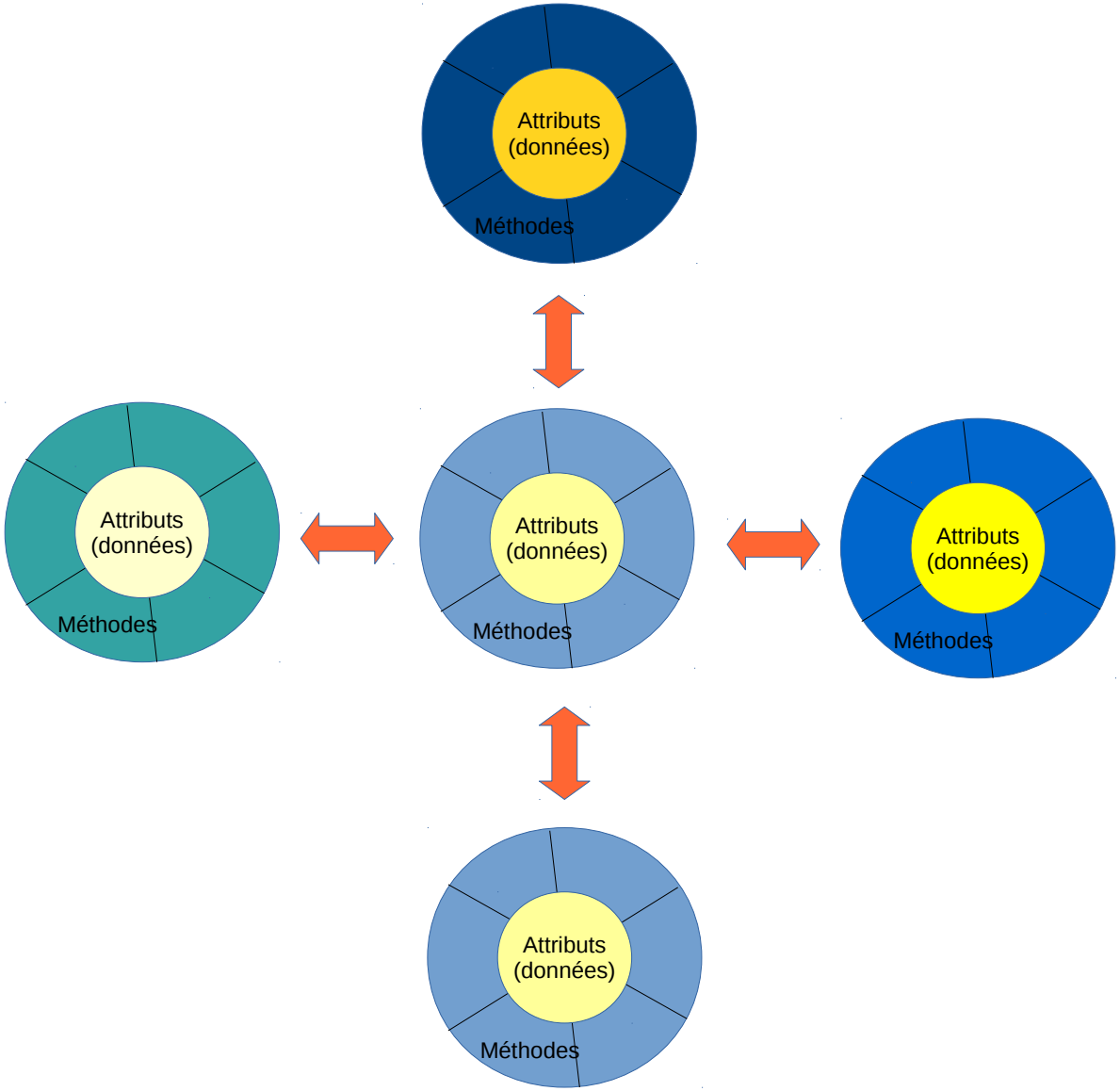
Une classe peut se voir selon trois points de vue :

- un aspect *extensionnel* : l'ensemble des objets (ou instances) représentés par la classe,
- un aspect *intensionnel* : la description commune à tous les objets de la classe, incluant les données (partie statique ou attributs) et les opérations (partie dynamique),
- un aspect *génération* : la classe sert à engendrer les objets.

Encapsulation des données



Encapsulation des données et communication entre objets



Sommaire

Les classes et instances en UML

Modèle statique (ou structurel)

- Diagrammes d'objets ou diagrammes d'instances
  - objets du domaine modélisé ou de la solution informatique (par exemple des personnes, des comptes bancaires)
  - liens entre ces objets (par exemple le fait qu'une personne possède un compte bancaire) ;
- Diagrammes de classes
  - abstraction des diagrammes d'objets
  - classes, associations, attributs, opérations

Représentation des classes

Pour "ranger" les classes : les paquetages

- regroupement logique d'éléments UML, par exemple de classes
- structuration d'une application
- utilisés dans certains langages, notamment Java, ce qui assure une bonne traçabilité de l'analyse à l'implémentation
- liés par des relations de dépendance

Les paquetages en UML

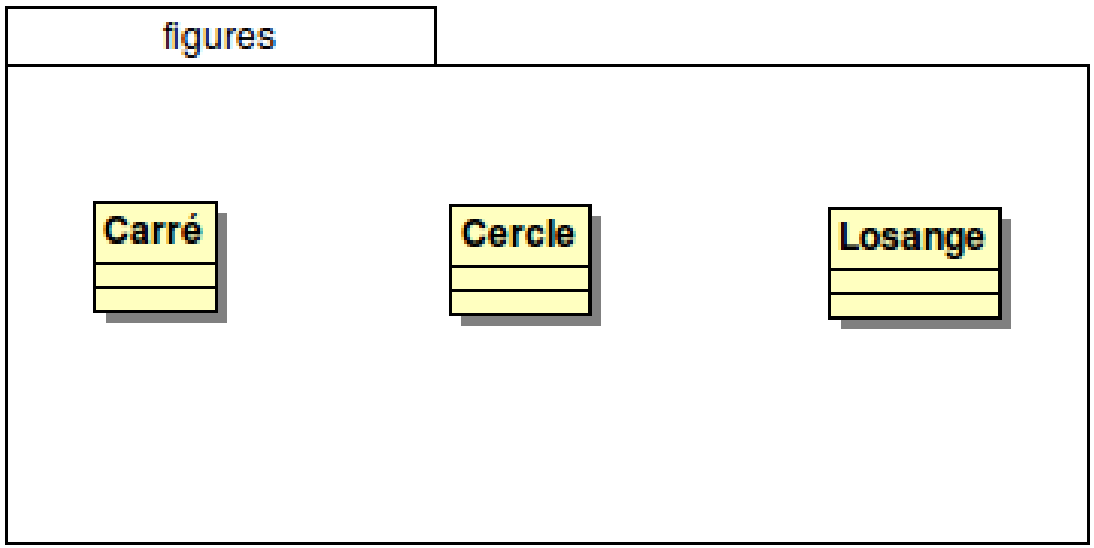


Figure – Paquetages en UML

Instances et classes en notation UML

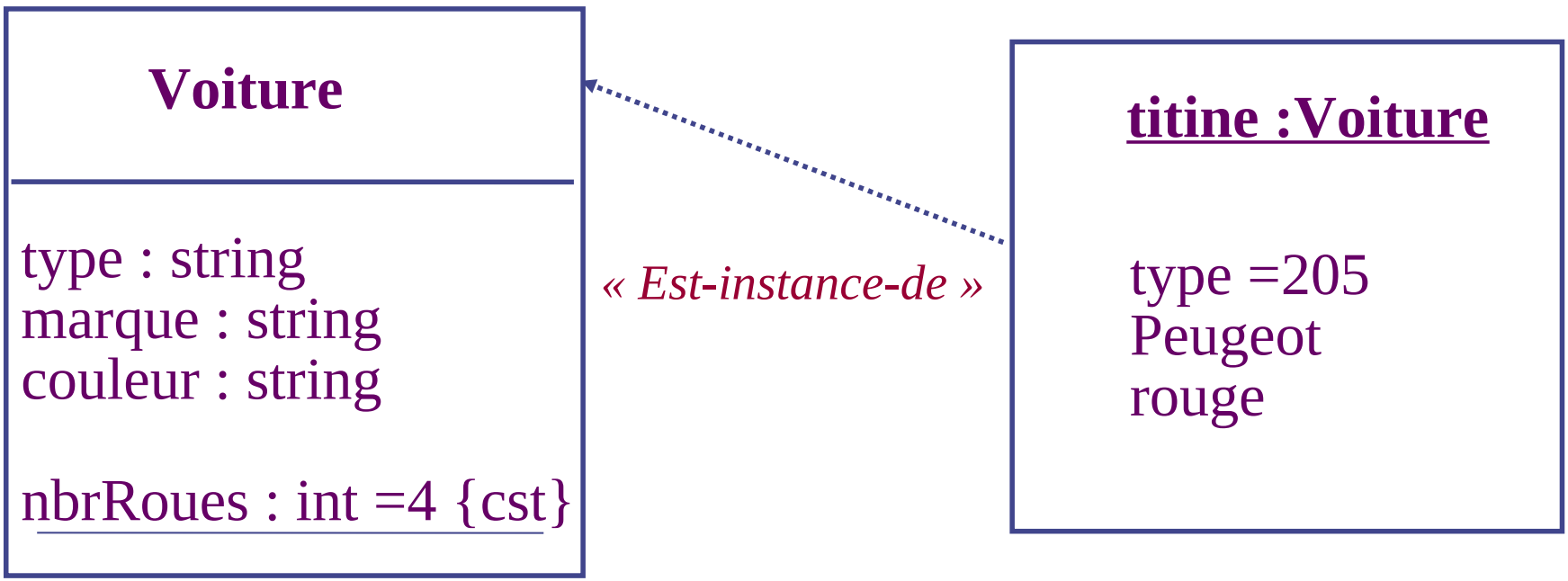


Figure – Classe (à gauche) et objet/instance (à droite)

Attributs

Quelles informations pour décrire un attribut ? Principalement :

- son nom
- son type
- sa visibilité (qui peut le voir/l'utiliser ?)
- la multiplicité (combien de valeurs sont portées simultanément par l'attribut ?)

Description des attributs

`[visibilité][/]nom[:type][[multiplicité]][= valeurParDéfaut]`

- `visibilité` $\in \{+, -, \#, \sim\}$, et `multiplicité` définit une valeur (1, 2, n, ...) ou une plage de valeurs (1..*, 1..6, ...).
- visibilité : d'où est visible l'attribut ?
  - Public. + est la marque d'un attribut accessible partout (public)
  - Privé. - est la marque d'un attribut accessible uniquement par sa propre classe (privé)
  - Package. $\sim$ est la marque d'un attribut accessible par tout le paquetage
  - Protected. # est la marque d'un attribut accessible par les sous-classes de la classe
- le nom est la seule partie obligatoire de la description

Description des attributs

`[visibilité][/]nom[:type][[multiplicité]][= valeurParDéfaut]`

- la multiplicité décrit le nombre de valeurs que peut prendre l'attribut (à un même moment)
- le type décrit le domaine de valeurs
- la valeur initiale décrit la valeur que possède l'attribut à l'origine
- des propriétés peuvent préciser si l'attribut est constant (`{constant}`), si on peut seulement ajouter des valeurs dans le cas où il est multi-valué (`{addOnly}`), etc.

Description des attributs

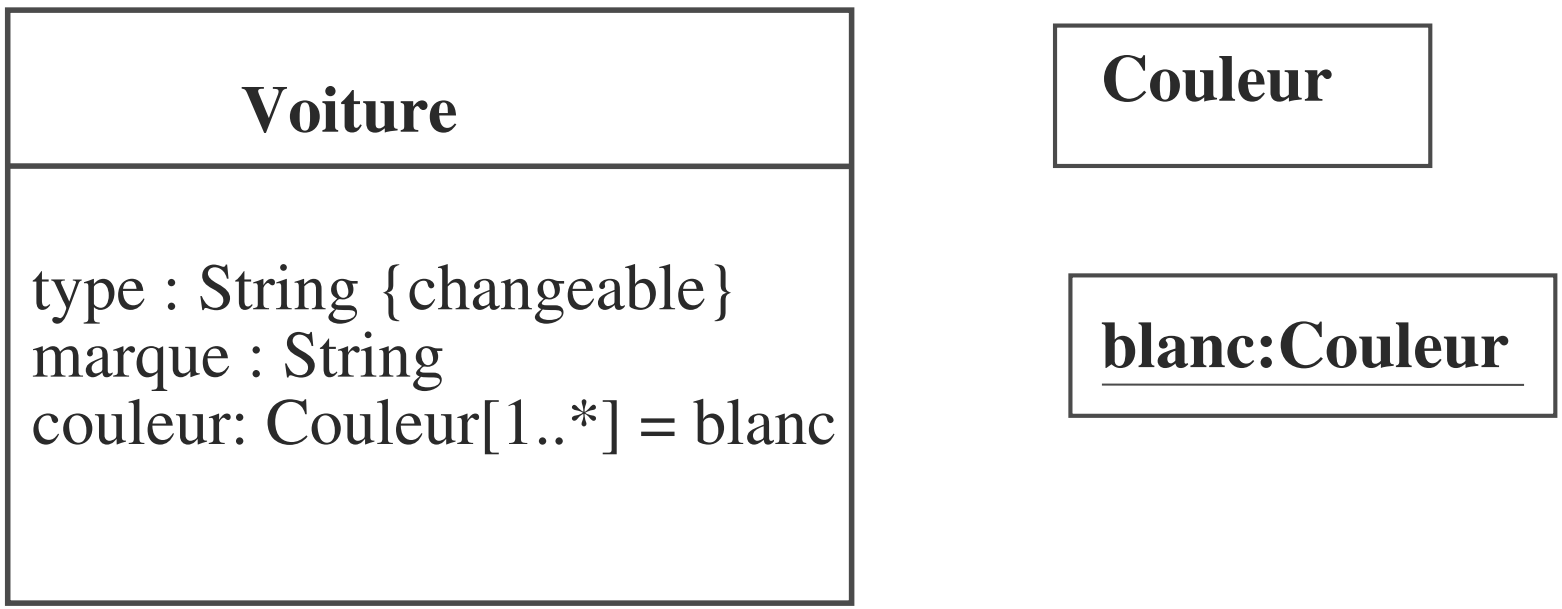


Figure – Détails sur la syntaxe de description des attributs

Sommaire

Types de base en Java

- `boolean`, constitué des deux valeurs `true` et `false`. Les opérateurs se notent :

Java	non	égal	différent	et alors / et	ou sinon / ou
	!	==	!=	&&	||
				&	|

- `int`, entiers entre -2^{31} et $2^{31}-1$
- `float`, `double`, ces derniers sont des réels entre $-1.79E+308$ et $+1.79E+308$ avec 15 chiffres significatifs.
- `char`, caractères représentés dans le système Unicode. Les constantes se notent entre apostrophes simples, par exemple `'A'`.
- `String`, qui est ... une classe. Les chaînes de caractères constantes se notent entre guillemets, par exemple `"hello world"`.

Écriture des classes

```
package exemplesCours1;
public class Voiture
{
private String type; // null
private String marque; // null
private String couleur; // null
private static int nbrVoitures; // 0
private static final int NBR_ROUES = 4;
}
```

Conventions :

- le nom de la classe commence par une majuscule,
- le nom du package commence par une minuscule,
- le nom des attributs commence par une minuscule,
- les constantes sont en majuscules.

Création des instances

Déclaration d'une variable

```
Voiture titine;
```

Création d'une instance

```
titine = new Voiture();
```

Accès aux attributs, affectations

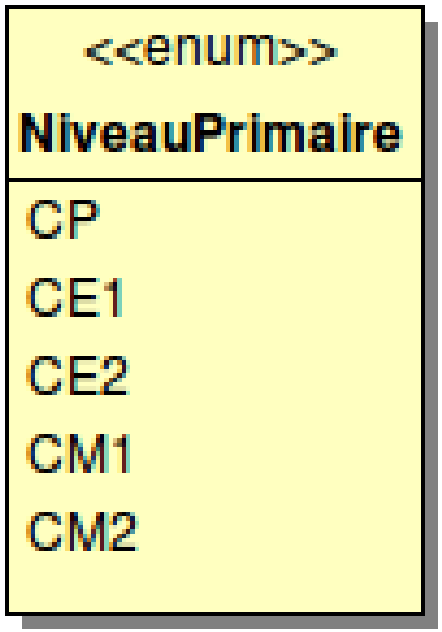
Notation '.' Exemple `titine.type`
Ecriture de valeurs dans des attributs d'instance

```
titine.type = "205";
titine.marque = "Peugeot";
titine.couleur = "rouge";
```

Sommaire

Les énumérations

- Un type de données dont on peut énumérer toutes les valeurs possibles
- Exemples :
  - Civilité → Mme, M, Mlle
  - Stations de ski du grand domaine → Valmorel, Combelouvière, Saint-François-Longchamp
  - Niveaux à l'école primaire → CP, CE1, CE2, CM1, CM2



Petit aperçu des énumérations en Java

```
public enum Niveau{
CP, CE1, CE2, CM1, CM2;
}

...

Niveau n=Niveau.CP;
...
```

Attributs dérivés

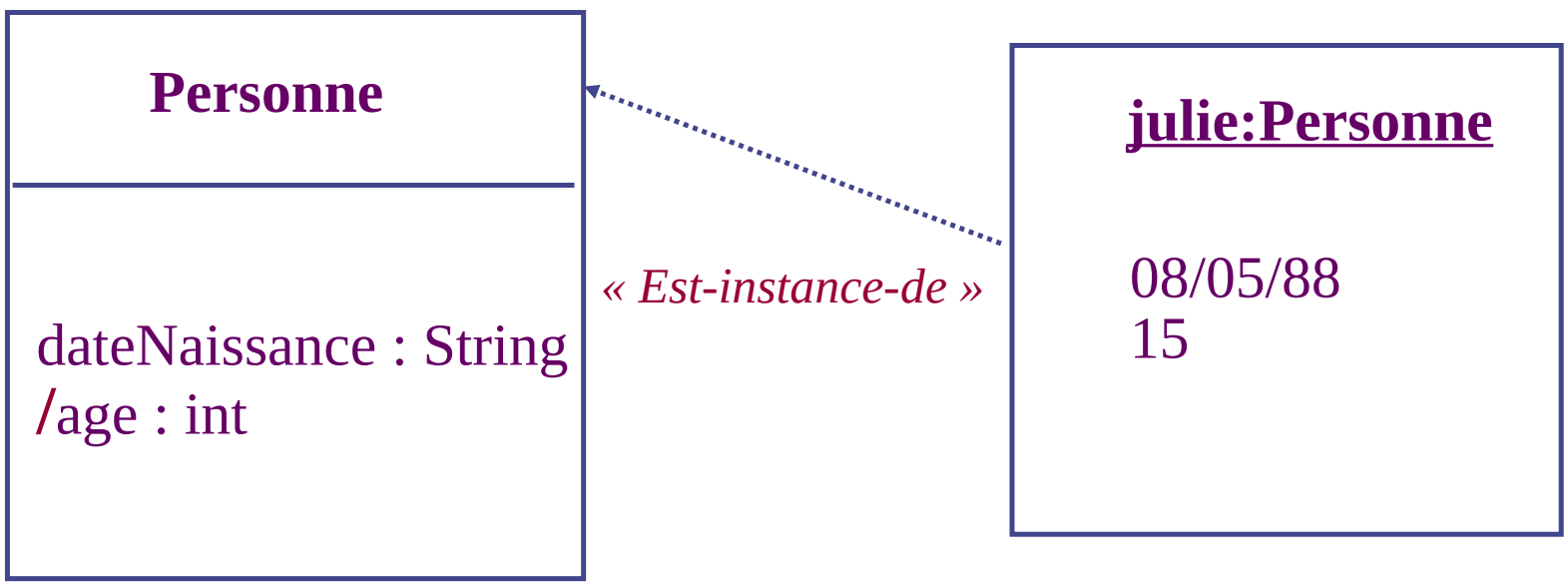
Valeur déduite de la valeur d'autres attributs ou d'autres éléments décrivant l'objet

Classe

Attributs dérivés

Instance

Valeurs d'attributs (État)



{age = date du jour – date de naissance}

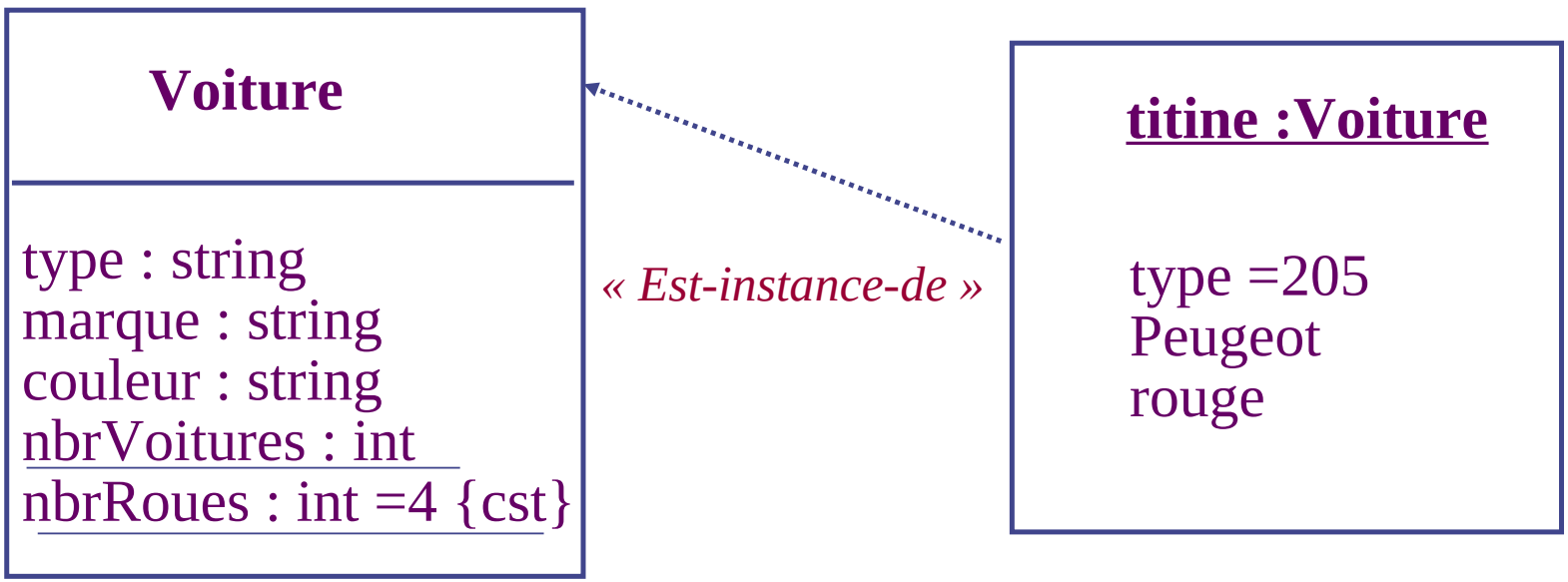
Peut-être une opération déguisée ? (stockage optionnel)

Attributs *de classe*

Valeur partagée par toutes les instances

Classe
Attributs de classe

Instance
Valeurs d'attributs (État)
Sauf des attributs de classe



Attribut de classe ~ caractéristique partagée
Révèle souvent une modélisation à approfondir

Figure – Attributs de classe

Accès aux attributs de classe

Notation : Nom de la classe ou de l'instance '.' nom de l'attribut

```
Voiture.nbrVoitures = 3;
Voiture titine=new Voiture();
titine.nbrVoitures=4;
```

Sommaire

Prochaine étape

- Animer les objets
- Opérations décrivant la manière dont les objets répondent aux messages

Mais avant ...

On s'intéresse aux puzzles 2D. Leur dimension se définit par une longueur et une largeur donnée généralement en centimètres. Ils ont un certain nombre de pièces (20, 50, 100, 1000, etc.) et en général un nom. Ils sont des reproductions d'images (photos, peintures, ou dessins). On trouve des puzzles en bois ou en carton.

- Réalisez un diagramme de classes pour les puzzles.
- Réalisez un diagramme d'objets pour un puzzle en carton, nommé "Femmes au marché", de 98x75cm, étant une reproduction de peinture, et ayant 2000 pièces.

Le diagramme de classe

Héritage et spécialisation/généralisation

Faculté des Sciences / Université de Montpellier
Modélisation et programmation par objets

2017

Héritage et spécialisation/généralisation

Concept

- Extension : ensemble des objets couverts
- Intension : prédicats, caractéristiques des objets couverts
- Classes et interfaces Java, classes C++

Concept Rectangle

- Extension : ensemble des rectangles
- Intension : ensemble des caractéristiques des rectangles, posséder quatre côtés parallèles 2 à 2, deux côtés consécutifs forment un angle droit, notion de largeur, de hauteur, etc.

Héritage et spécialisation/généralisation

Classification par organisation des concepts

- inclusion des extensions
- héritage et raffinement des intensions

`Concept Carré` spécialise `Concept Rectangle`

- Point de vue extensionnel : l'ensemble des carrés est inclus dans l'ensemble des rectangles
- Point de vue intensionnel : les propriétés des rectangles s'appliquent aux carrés et se spécialisent (largeur = hauteur)

Héritage et spécialisation/généralisation

Cas d'étude

- Contexte d'une agence immobilière
- Gestion de la location d'appartement

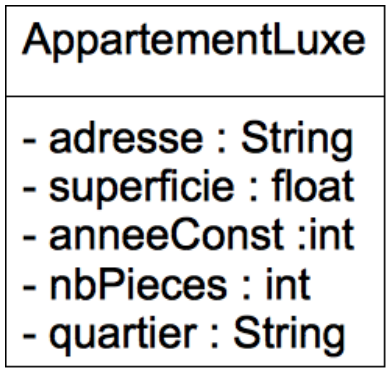
Appartement

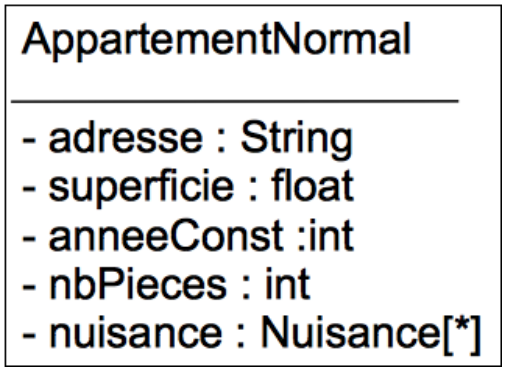
- tous sont décrits par : une adresse, une année de construction, une superficie, un nombre de pièces
- deux sous-catégories nous intéressent
  - les appartements de luxe décrits en plus par leur quartier
  - les appartements normaux décrits en plus par les nuisances de leur environnement

Sans spécialisation/généralisation

Solution 1

Réaliser deux classes





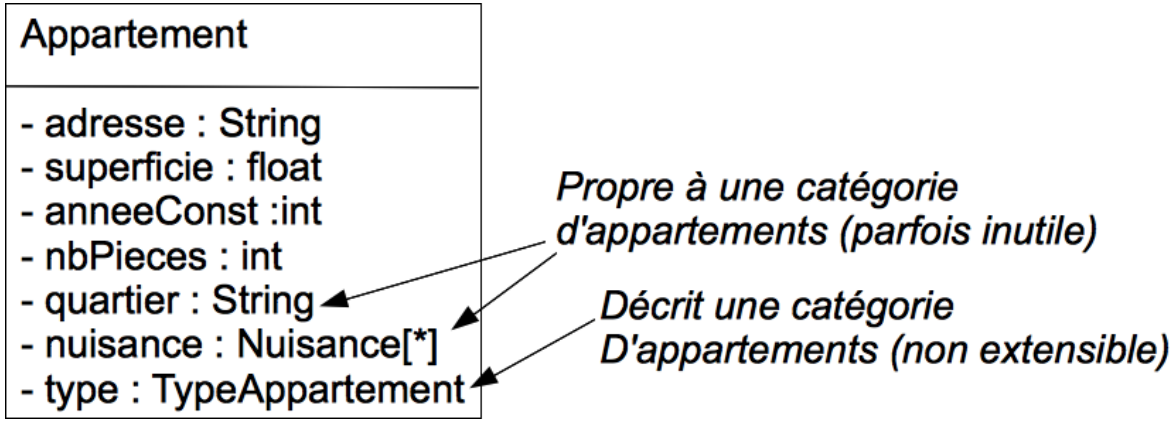
Inconvénients

- répétition des attributs et de parties dans le code des méthodes
- risque d'incohérence (sur les parties communes) entre les différentes sortes d'appartements
- toute modification sur les parties communes demandera d'intervenir à plusieurs endroits (avec de nouveaux risques d'erreurs)

Sans spécialisation/généralisation

Solution 2

Réaliser une seule classe



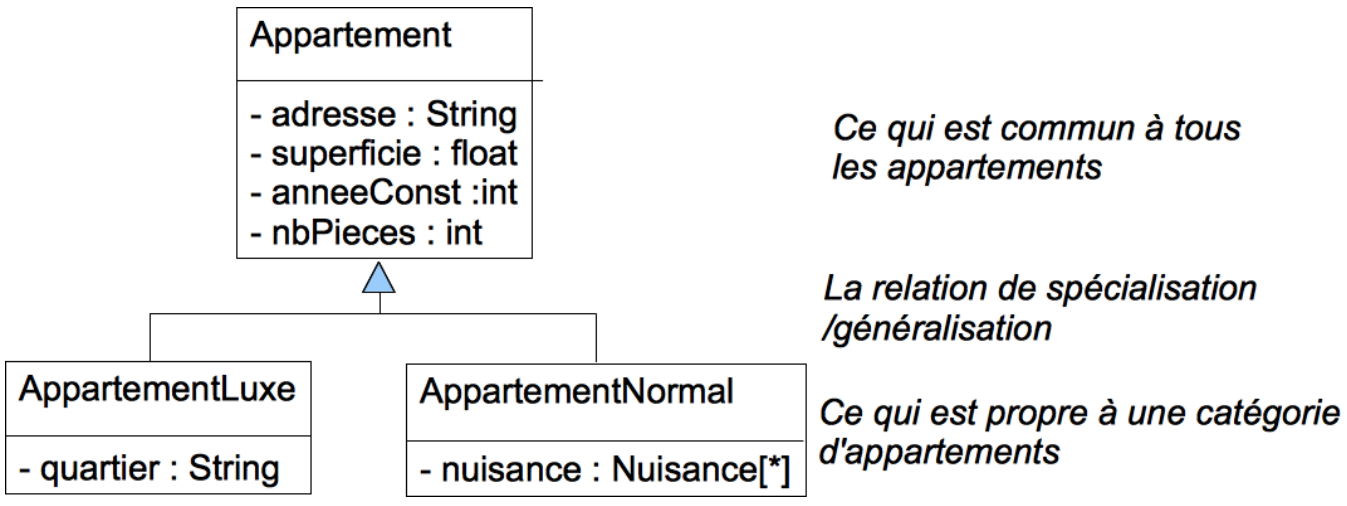
Inconvénients

- certains attributs (quartier, nuisance) ne sont pas toujours utilisés
- on ne peut pas ajouter facilement une sous-catégorie (ex. appartement de fonction)

La solution de la spécialisation/généralisation

Solution spécifique des approches à objets

Réaliser trois classes et les connecter par spécialisation



Avantages

- pas de répétition, pas de risque d'incohérence, pas d'attributs inutiles
- facile à étendre, par une nouvelle sous-classe

Quelques instances

a1 : AppartementLuxe
adresse = « 8, ch. des lilas, Mulhouse »
anneeconst = 1988
superficie = 150
nbPieces = 4
quartier = « La petite rivière »

a2 : AppartementNormal
adresse = « 10, rue H Berlioz, Mulhouse »
anneeconst = 1988
superficie = 40
nbPieces = 2
nuisance = {centreVille}

Précisions sur les relations de généralisation/spécialisation

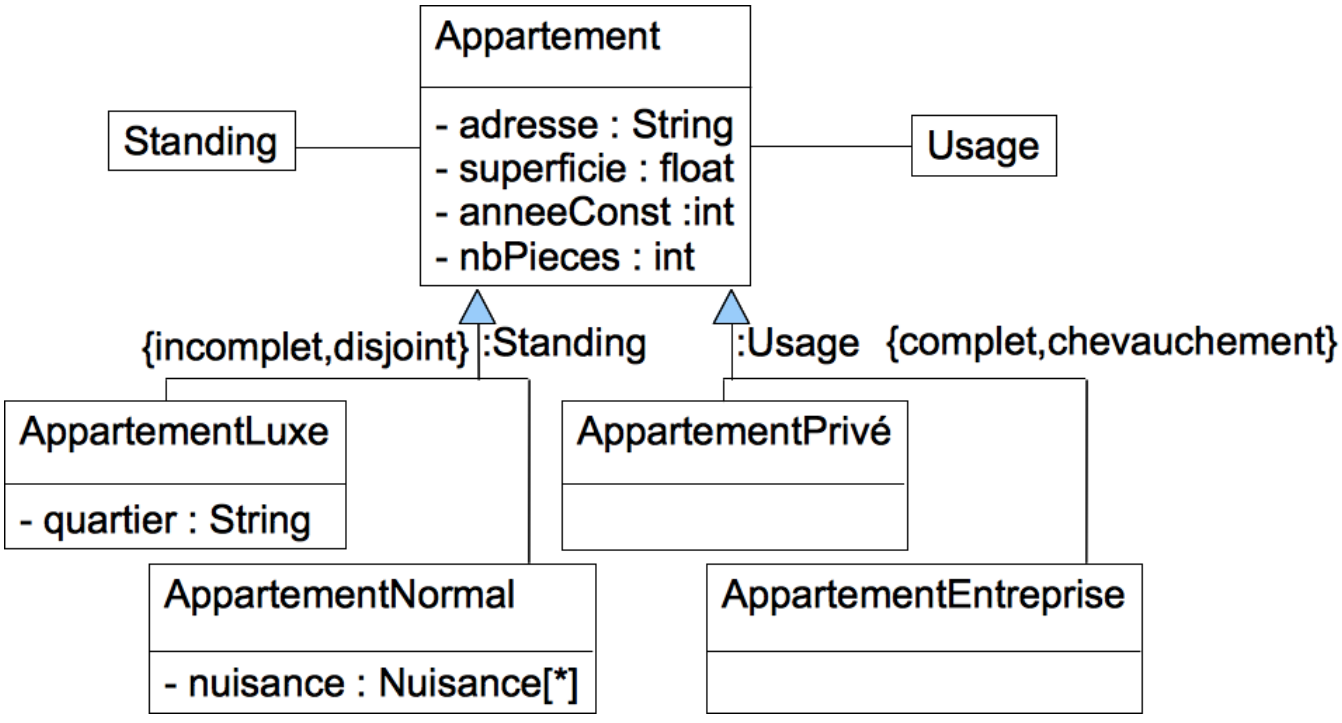
Discriminant

- critère de classification
- est représenté par une classe et par une annotation sur un ensemble de relations de spécialisation/généralisation

Contraintes

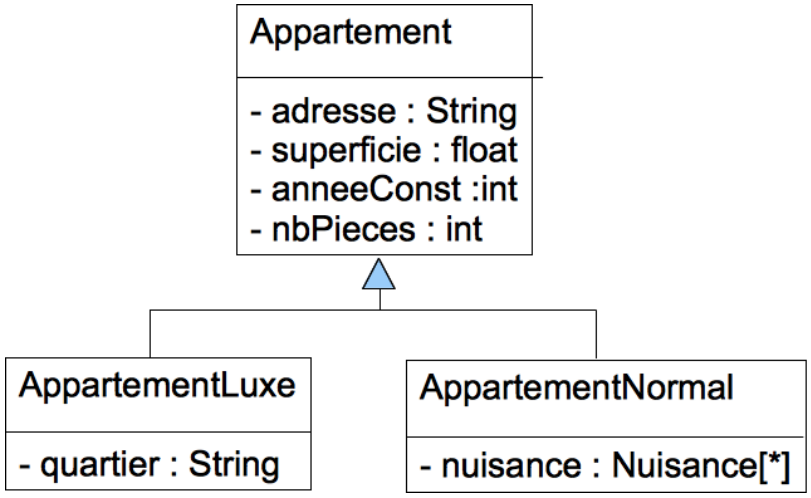
- annotent, entre accolades, un ensemble de relations de spécialisation/généralisation
- il en existe quatre (deux paires) :
  - complet, incomplet : les sous-classes couvrent (ne couvrent pas) l'ensemble des objets de la superclasse
  - disjoint, chevauchement : les sous-classes ne peuvent pas (peuvent) avoir d'instances communes

Discriminants et contraintes



- il existe des appartements autres que de luxe ou normaux
- un appartement ne peut être en même temps de luxe et normal
- un appartement est soit privé, soit pour une entreprise (pas d'autre cas)
- un appartement peut être utilisé à la fois pour un usage privé et pour un usage par une entreprise

Traduction en Java



Ce qui est commun à tous les appartements

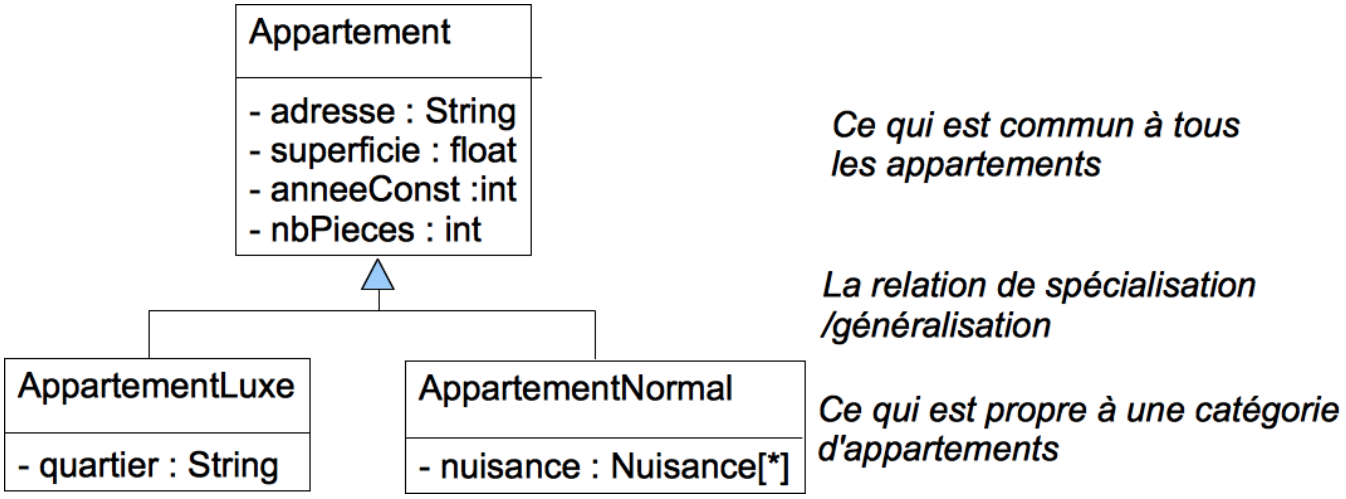
La relation de spécialisation /généralisation

Ce qui est propre à une catégorie d'appartements

Listing 1 – Classe Appartement / attributs

```
1  public class Appartement {
2  private String adresse;
3  private int anneeConst;
4  private float superficie;
5  private int nbPieces;
6  .....
7  }
```

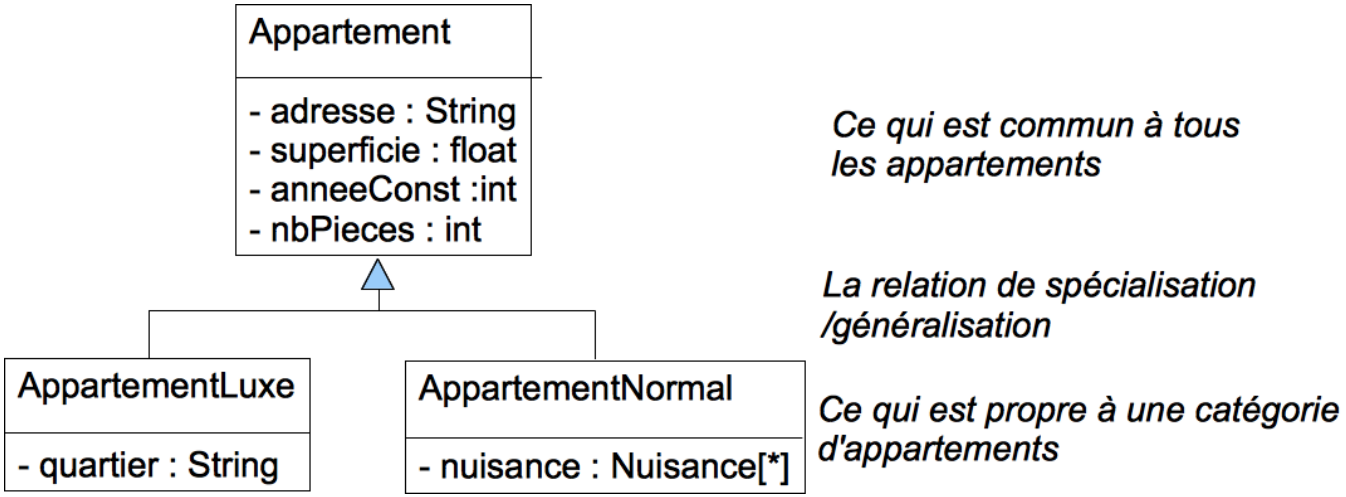
Traduction en Java



Listing 2 – Classe Appartement de luxe/ attributs

```
1  public class AppartementLuxe extends Appartement{
2  private String quartier;
3  .....
4  }
```

Traduction en Java



Listing 3 – Classe Appartement de normal / attributs

```
1  public class AppartementNormal extends Appartement{
2  private Nuisance[] nuisances;
3  .....
4  }
```

Traduction en Java

Listing 4 – enum Nuisance

```
1  public enum Nuisance{
2  autoroute, aeroport, dechetterie, centre_ville,
3  .....
4  }
```

Création d'objets

Listing 5 – Creation

```
1  AppartementLuxe a1 = new AppartementLuxe(...);
2  Appartement a2 = new AppartementLuxe(...);
3  Object a3 = new AppartementLuxe(...);
```

Les lignes 2 et 3 sont des affectations polymorphes :
la variable a un type (Appartement) différent du type de l'instance (AppartementLuxe)

Notez que l'affectation polymorphe se fait "en remontant" :
on affecte une instance de la sous-classe à une variable de la superclasse

`Object` est la super-classe implicite de toutes les classes en Java.

Constructeurs / dans la super-classe

Listing 6 – Constructeurs dans la super-classe

```
1  public class Appartement{
2  public Appartement()
3  {adresse = "inconnue"; anneeConst = 1970; superficie = 15;
       nbPieces = 1;}
4  public Appartement(String ad, int ac, float s, int n)
5  {adresse = ad; anneeConst = ac; superficie = s; nbPieces = n
       ;}
6  }
```

Règle

- Toujours écrire un constructeur sans paramètre
- Le constructeur initialise les attributs propres à la classe

Constructeurs / dans la sous-classe

Listing 7 – Constructeurs dans la sous-classe

```
1  public class AppartementLuxe extends Appartement{
2  public AppartementLuxe(){quartier = "inconnu";}
3  public AppartementLuxe(String ad, int ac, float s, int n,
       String q)
4  {super(ad, ac, s, n); quartier = q;}
5  }
```

Règle

- `super` : transmet les paramètres au constructeur de la super-classe
- L'appel au constructeur de la super-classe, s'il a lieu, est forcément la première instruction exécutable du constructeur.
- La super-classe doit disposer d'un constructeur admettant ces paramètres

Constructeurs / Appel

Listing 8 – Creation

```
1  Appartement a1 = new AppartementLuxe();
2  Appartement a2 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
```

Constructeurs exécutés

- les constructeurs de toutes les super-classes sont appelés du haut vers le bas
- Ligne 1 : Object() ; Appartement() ; AppartementLuxe()
- Ligne 2 : Object() ; Appartement(String ad, int ac, float s, int n) ; AppartementLuxe(String ad, int ac, float s, int n, String q)

Définition des méthodes

Trois configurations

- Héritage : la méthode est écrite dans une classe et accessible dans ses sous-classes
- Masquage : la méthode est écrite dans une classe et entièrement réécrite dans une sous-classe
- Spécialisation : la méthode est écrite dans une classe ; dans la sous-classe on fait référence à la méthode de la super-classe pour modifier légèrement le comportement initial

Héritage de méthode

Listing 9 – Héritage de méthode

```
1  public class Appartement {
2  ...
3  public float valeurLocativeBase()
4     {return superficie * 5 * (1 + nbPieces/n);}
5  }
6  public class AppartementLuxe extends Appartement {
7  ...
8  // rien qui concerne la valeur locative de base
9  }
10 ...
11 AppartementLuxe a1 = new AppartementLuxe("8, ch. Lilas, 
       Mulhouse", 1988, 150, 4, "La petite riviere");
12 System.out.println(a1. valeurLocativeBase());
```

Masquage de méthode

Listing 10 – Masquage de méthode

```
1  public class Appartement {
2  public String toString(){return "appt - adresse ="+ adresse
       + "superficie = "+superficie+" m2";}
3  }
4  public class AppartementNormal extends Appartement {
5  public String toString(){return " annee de construction = "+
       anneeConst;}
6  }
7  ...
8  Appartement a2 = new AppartementNormal("8, ch. Lilas , 
       Mulhouse", 1988, 150, 4, Nuisance.centre_ville);
9  System.out.println(a2.toString());
```

Ligne 11 : annee de construction = 1988

Spécialisation de méthode

Listing 11 – Spécialisation de méthode

```
1  public class Appartement {
2  ...
3  public String toString(){return "appt - adresse ="+ adresse
       + "superficie = "+superficie+" m2";}
4  }
5  public class AppartementLuxe  extends Appartement {
6  ...
7  public String toString(){return super.toString()+" quartier
       = "+quartier;}
8  }
9  ...
10 Appartement a3 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
11 System.out.println(a3.toString());
```

Ligne 11 : appt - adresse = 8, ch. Lilas, Mulhouse superficie = 150 m2 quartier = La petite riviere

Classes et méthodes abstraites

Loyer

Le loyer se calcule comme produit de :

- la valeur locative de base
- le coefficient modérateur

Coefficient modérateur

Le coefficient modérateur vaut :

- 1.1 pour les appartements de luxe
- 1 - 0.1*nombre de nuisances

La méthode calculant le coefficient modérateur est donc nécessaire pour écrire la méthode de calcul de loyer, mais elle ne peut être écrite de manière générale pour les appartements.

Classes et méthodes abstraites

Listing 12 – Classe et méthode abstraite

```
1  public abstract class Appartement {
2  ...
3  public abstract float coeff(); // pas de corps !!!
4  }
5  ...
6  public class AppartementNormal extends Appartement{
7  ...
8  public float coeff()
9  {return 1 - 0.1 * nuisances.lenght;}
10 }
11 ...
12 public class AppartementLuxe extends Appartement{
13 ...
14 public float coeff(){return 1.1;}
15 }
```

Classes et méthodes abstraites

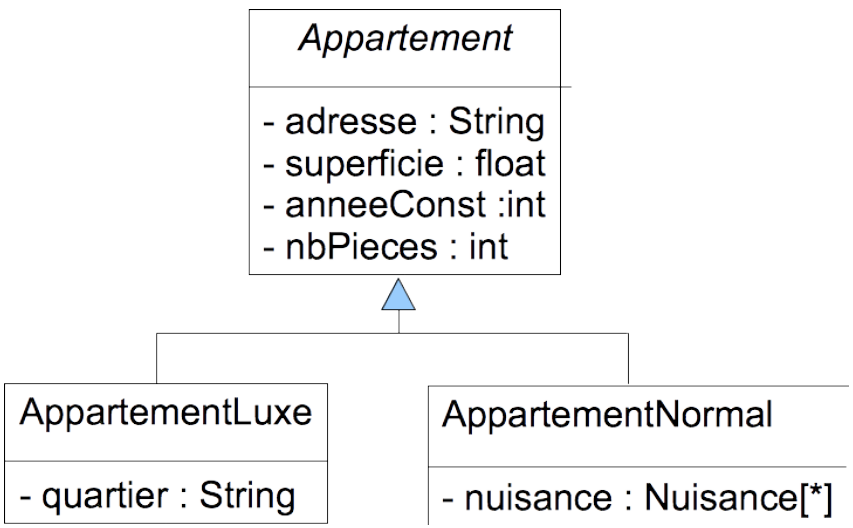
- la classe abstraite peut toujours servir de type pour des variables
- on ne peut plus créer d'instance de la classe abstraite (car sa définition est incomplète)

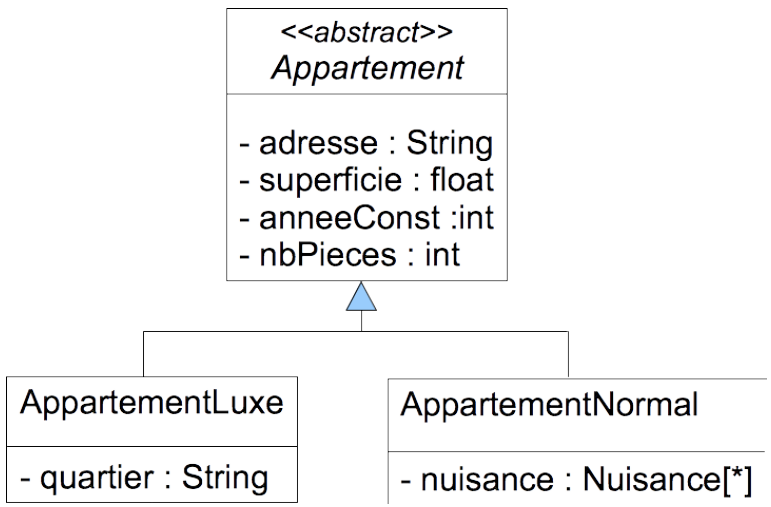
Listing 13 – Classe abstraite

```
1  Appartement a1 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
2  ...
3  // DEVIENT IMPOSSIBLE : Appartement a2 = new Appartement("8,
       ch. Lilas, Mulhouse", 1988, 150, 4);
```

Classes et méthodes abstraites

Représentation en UML (italiques, stéréotype)





Liaison dynamique

Choix d'une méthode à appeler

- Lorsqu'une méthode est appelée, elle est recherchée à partir de la classe de l'objet et en remontant vers ses superclasses jusqu'à la trouver
- On parle de liaison dynamique car le choix est réalisé lors de l'exécution

Listing 14 – Classe abstraite

```
1  Appartement a1 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
2  System.out.println(a1.loyer());
```

Exécute : loyer et valeurLocativeBase de Appartement ; coeff de AppartementLuxe

Associations UML et leur implémentation en Java

LIRMM / Université de Montpellier

Février 2017

Sommaire

Définition

- association : relation entre 2 ou plusieurs classes qui décrit les connexions structurelles entre leurs instances
- exemple : entre Pays et Ville : *a pour capitale* (ou dans l'autre sens : *est la capitale de*).
- au niveau instance : lien

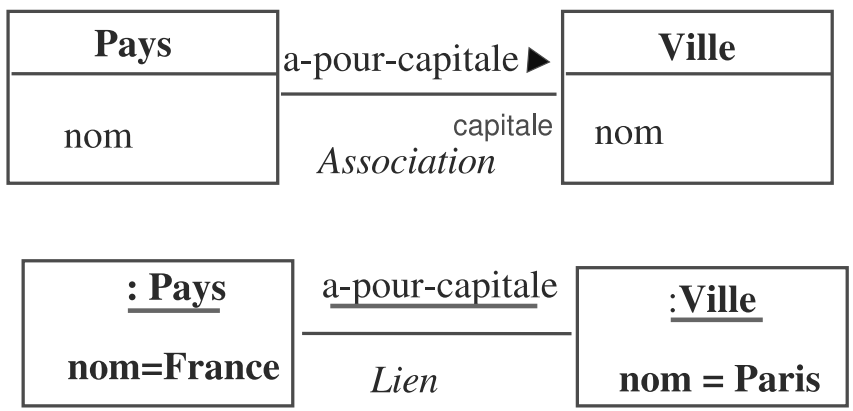


Figure – Associations et liens

Représentation des associations

Association binaire : un trait entre 2 classes avec possiblement :

- le nom de l'association,
- le nom des rôles aux extrémités de l'association,
- la multiplicité des extrémités,
- la navigabilité.

Figure – Association binaire

Associations et attributs

- Lisibilité : on voit bien mieux les liens entre classes avec une association qu'avec des attributs ;
- Liens dépendants : avec une association, on définit 2 liens dépendants : les 2 extrémités de l'association. Deux attributs de part et d'autre ne sont pas équivalents à une association ;
- Associations complexes : on peut définir des associations complexes.

Convention utilisée dans la suite de ce module

- Convention : les attributs sont uniquement de type simple (entiers, flottants, booléens, ...).
- Conséquence logique : on ne mettra jamais un attribut de type complexe, on préférera alors une association.

Sommaire

Comment traduire les associations en Java ?

- On ne traduit que les extrémités navigables.
- Si extrémité de cardinalité $\leq 1 \rightarrow$ attribut.
- Si extrémité de cardinalité $> 1 \rightarrow$ types de bibliothèque représentant des collections : listes, tableaux, ensembles, ...
- Si association bidirectionnelle $\rightarrow$attention à bien à faire les mises à jour des 2 côtés. Ex : voiture et propriétaire.
- Si composition $\rightarrow$attention à bien respecter la contrainte de non partage de composants.

Tableaux

- Déclaration
- Construction effective (avec une taille donnée)
- Initialisation

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

Exemple de tableau d'int

```
int[] tab = new int[10];
for(int i = 0; i < 10; i++)
    tab[i] = i;
// autre solution
int[] tab2={0,1,2,3,4,5,6,7,8,9};
```

Tableaux à plusieurs dimensions

```
typ[][] tab;
tab = new typ[N][M]; // N lignes, M colonnes
int[][] tab = {{1,2,3},{4,5,6},{7,8,9}};
```

Qu'est-ce qu'une collection en Java ?

- Une classe qui définit une structure destinée à regrouper plusieurs objets
- Exemple : Pile, File, Liste chaînée, ...
- Une instance de Collection permet de manipuler les éléments de la structure (de pile, de file, de liste, ...) grâce à des méthodes bien pratiques : ajout d'élément, recherche, etc.

Les collections Java

- Beaucoup de types complexes définis dans la bibliothèque Java pour les collections :
  - ensembles
  - listes
  - tableaux associatifs
  - ...
- Collections génériques
  - collections d'éléments de type E (où E est de type non primitif).
  - déclaration du type des éléments qui seront rangés dans la collection : collection d'entiers (en utilisant le type `Integer` et non `int`), de voitures, de personnes, etc.
- Opérations définies pour toutes les collections : l'ajout et la suppression d'éléments, l'obtention de la taille de la collection, etc.

Qu'est-ce que la généricité ?

Le paramétrage d'une classe par un type.

Qu'est-ce que la généricité ?

Exemple d'une classe Pile

- Pile de quoi ?
- Dans quoi stocker les éléments de la pile ?
- Peut-on mettre n'importe quoi dans une pile ? Des assiettes et des feuilles ?

L'exemple de la pile

- Côté programmation de la pile, on n'a pas précisément besoin de savoir ce qu'on mettra dedans : les piles d'assiettes et les piles de feuilles se ressemblent. Mais il nous faut quand même avoir la connaissance du type des objets manipulés ...
- Côté utilisation, on voudrait pouvoir dire :
  - "je veux une pile d'assiettes", et ne pouvoir y ranger que des assiettes
  - "je veux une pile de feuilles", et ne pouvoir y ranger que des feuilles
- Bilan : on veut fabriquer des piles contenant des éléments d'un certain type T, que l'utilisateur définira quand il déclarera sa pile.
  - Le programmeur de pile programme des piles de T
  - L'utilisateur précise ce qu'il veut pour T lors de l'utilisation d'une pile : T+assiette, ou T=Feuille, etc.

Création de classes génériques

Rendez-vous l'année prochaine ...

Utilisation de classes génériques

- Lors de la déclaration d'une variable dont le type est une classe générique
  - On précise entre $<$ $>$ le type des éléments que l'on veut manipuler
  - `Pile<Assiette> maPileAssiettes; // une pile d'assiette`
  - `Pile<Feuille> maPileFeuilles; // une pile de feuilles`
- Utilisation du constructeur
  - `maPileAssiettes=new Pile<Assiette>()`
- Ensuite on utilise normalement les méthodes prévues dans la classe générique
  - Si dans la classe `Pile<T>`, il y a une méthode `push(T elem)`
  - Alors
    - `maPileAssiettes.push(new Assiette());`
    - et surtout pas : `mapileAssiettes.push(new Feuille());`

Collections Java : c'est facile !

Un peu de méthode Coué ne peut pas nuire

Les collections Java sont très simples à manipuler, beaucoup plus que des tableaux classiques.

Exemple des vecteurs

- Collections qui ressemblent aux tableaux, mais de taille non prédéfinie

```
Vector<MonType> v; // déclaration
v=new Vector<MonType>(); // création
```

Déclaration et création en une seule ligne :

```
Vector<MonType> v=new Vector<MonType>();
```

Méthodes classiques des vecteurs

- `void addElement(E obj)` ou plus court `void add(E obj)`. Ajoute l'objet `obj` à la fin du vecteur, et augmente sa taille de 1.
- `E get(int index)`. Retourne l'objet placé en position `index` dans le vecteur.
- `boolean isEmpty()`. Teste si le vecteur n'a aucun élément.
- `int size()`. Retourne la taille du vecteur.
- `E remove(int index)`. Supprime l'objet en position `index` et le retourne.
- `boolean remove(Object o)`. Supprime la première occurrence de `o` rencontrée (laisse le vecteur inchangé si l'objet `o` n'est rencontré, et retourne alors faux).

Exemple de vecteur de chaînes

```
Vector<String> prenoms = new Vector<String> ();
prenoms.addElement("Thomas");
prenoms.addElement("Sophie");
// prenoms.addElement(new Voiture()); -> interdit

//affichage des prénoms
for (int i = 0; i < prenoms.size(); i++)
{
   System.out.println(prenoms.get(i));
}
// autre affichage des prénoms avec la boucle
// for existant depuis Java 1.5
for (String prenomCourant:prenoms){
  System.out.println(prenomCourant);
}
```

Tableau et Vector : un exemple

On veut manipuler des familles de mots. Comparons l'usage d'un tableau par rapport à un vector.

```
1   package tableauxVsVectors;
2
3   public class Mot {
4     private String mot;
5     private String traduction;
6
7     public Mot(String mot){
8         this.mot=mot;
9     }
10
11    public Mot(String mot, String traduction){
12        this.mot=mot;
13        this.traduction=traduction;
14    }
15
16    public String toString(){
17        String result=mot;
18        if (traduction!=null) result=result+"("+traduction+")";
19        return result;
20    }
21  }
```

Programme manipulant les familles de mots

```
1   public static void main(String[] args){
2           ManipMotsTableaux mmt=new ManipMotsTableaux();
3
4           Mot ga=new Mot("ga");
5           Mot bu=new Mot("bu");
6           Mot zo=new Mot("zo");
7           Mot meu=new Mot("meu");
8           Mot zobuga=new Mot("ZoBuGa","pomper avec une petite pompe");
9
10          mmt.ajoutMot(ga);
11          mmt.ajoutMot(bu);
12          mmt.ajoutMot(zo);
13          mmt.ajoutMot(meu);
14          mmt.ajoutMot(zobuga);
15          System.out.println("il y a "+mmt.nbMots()+" mots");
16          System.out.print(ga+" existe ? ");
17          System.out.println(mmt.existeMot(ga));
18          System.out.print(zobuga+" existe ? ");
19          System.out.println(mmt.existeMot(zobuga));
20      }
```

Manipulation d'une famille de mots stockée dans un tableau

```
1  package tableauxVsVectors;
2
3  public class ManipMotsTableaux {
4
5      // on va manipuler un tableau de mots
6      private Mot[] tabMots;
7
8      public ManipMotsTableaux(){
9          // Il faut creer le tableau, donc en fixer la taille
10         tabMots=new Mot[4];
11     }
12 ...
```

Manipulation d'une famille de mots stockée dans un tableau

```
1   ...
2       // on ajoute le mot m
3       public void ajoutMot(Mot m){
4           int i=0; // indice auquel on peut placer m
5           boolean placeTrouvee=false; // on a trouve une place pour m
6           // on cherche une place vide
7           while (i<tabMots.length&&!placeTrouvee){
8               if (tabMots[i]==null){placeTrouvee=true;} // on a trouve
9               else {i++;} // on avance
10          }
11          if (placeTrouvee){
12              tabMots[i]=m;
13              System.out.println("ajout du mot "+m);
14          } else{
15              System.out.println("plus de place !");
16          }
17      }
18  ...
```

Manipulation d'une famille de mots stockée dans un tableau

```
1   ...
2       public boolean existeMot(Mot mot){
3           boolean trouve=false;
4           int i=0;
5           while (i<tabMots.length&&!trouve){
6               if (tabMots[i]==mot){trouve=true;} // on a trouve
7               else {i++;} // on avance
8           }
9           return trouve;
10      }
11
12      public int nbMots(){
13          int result=0;
14          for (int i=0;i<tabMots.length;i++){
15              if (tabMots[i]!=null){result++;}
16          }
17          return result;
18      }
19  ...
```

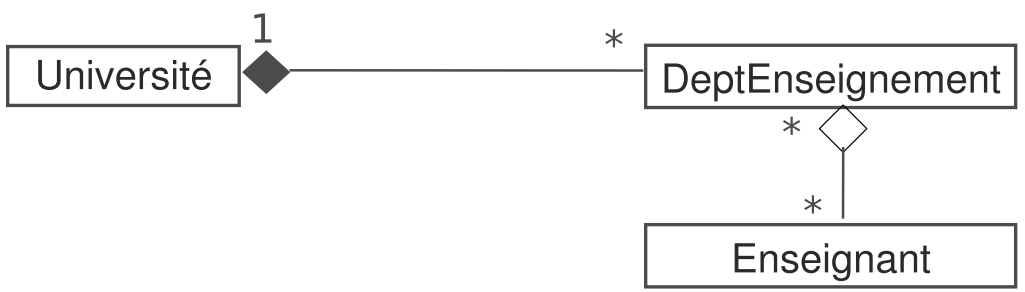
Manipulation d'une famille de mots stockée dans un Vector

```
1  package tableauxVsVectors;
2  import java.util.Vector;
3  public class ManipMotsVector {
4      // on va manipuler un vecteur de mots
5      private Vector<Mot> vMots;
6
7      public ManipMotsVector(){
8          // Il faut creer le vector mais pas necessairement en fixer la
               taille
9          vMots=new Vector<Mot>();
10     }
11
12     // on ajoute le mot m
13     public void ajoutMot(Mot m){
14         vMots.add(m);
15         System.out.println("ajout du mot "+m);
16     }
17
18     public boolean existeMot(Mot mot){
19         return vMots.contains(mot);
20     }
21
22     public int nbMots(){
23         return vMots.size();
24     }
25 }
```

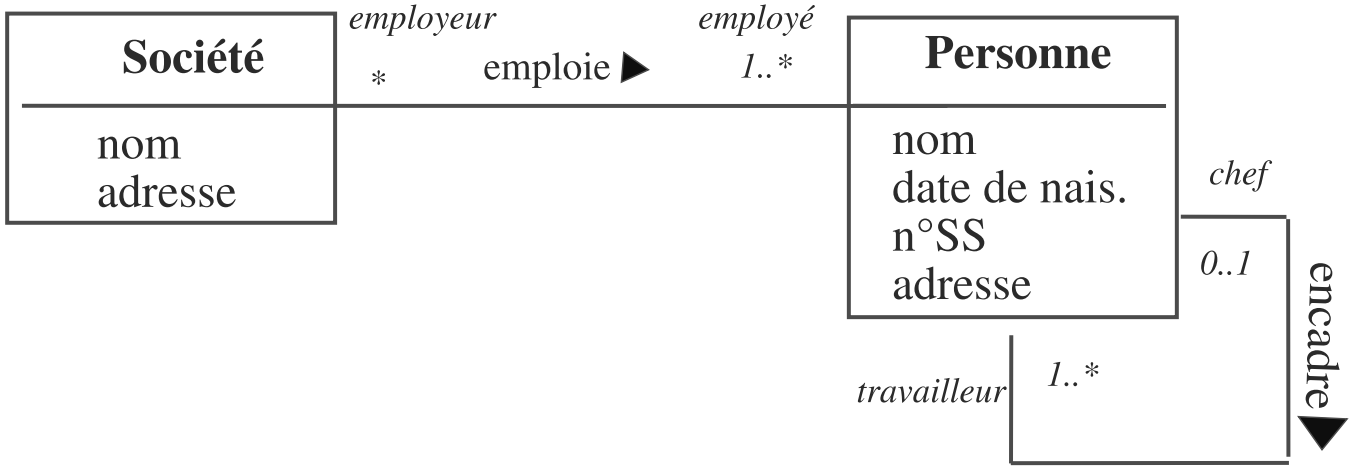
Sommaire

Agrégation et composition

- Agrégation : relation *ensemble-élément*, dénotée par un losange non rempli du côté de l'ensemble.
- Composition : relation de composition, on la note avec un losange plein du côté du composite. Notion d'exclusivité : un composant ne peut pas être partagé par plusieurs composites.
- "Despite the semantic linked to aggregation everyone think (for different reasons) that it is necessary. Consider it as a placebo" – James Rumbaugh.

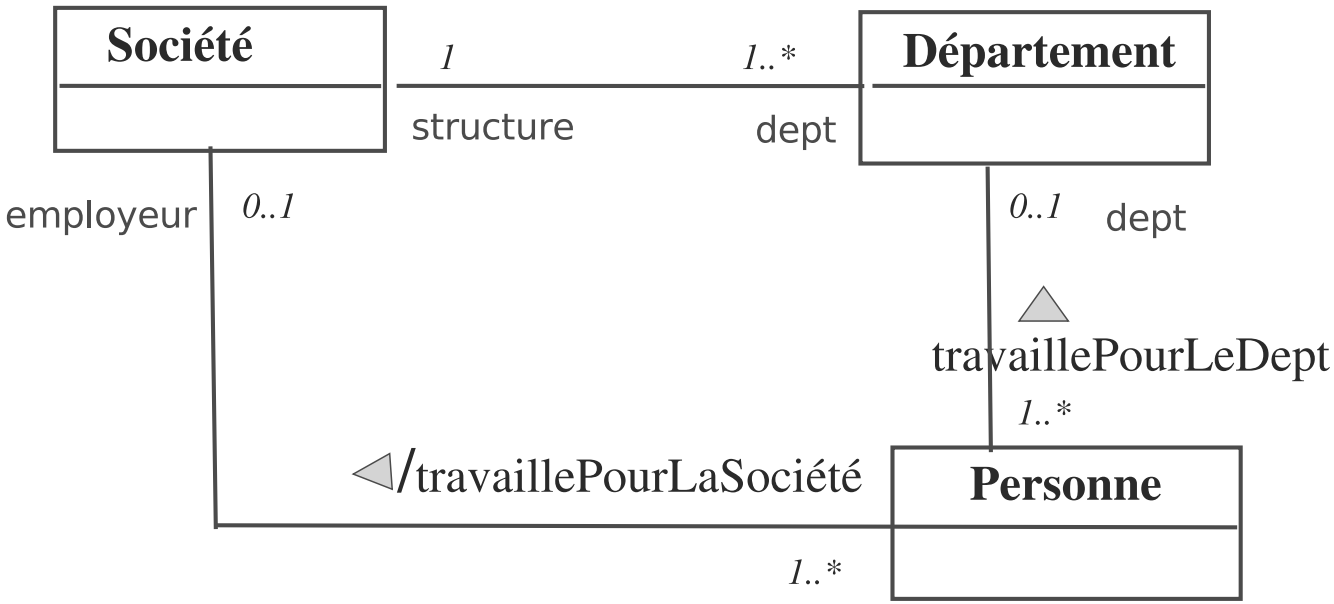


Association réflexive



Associations dérivées

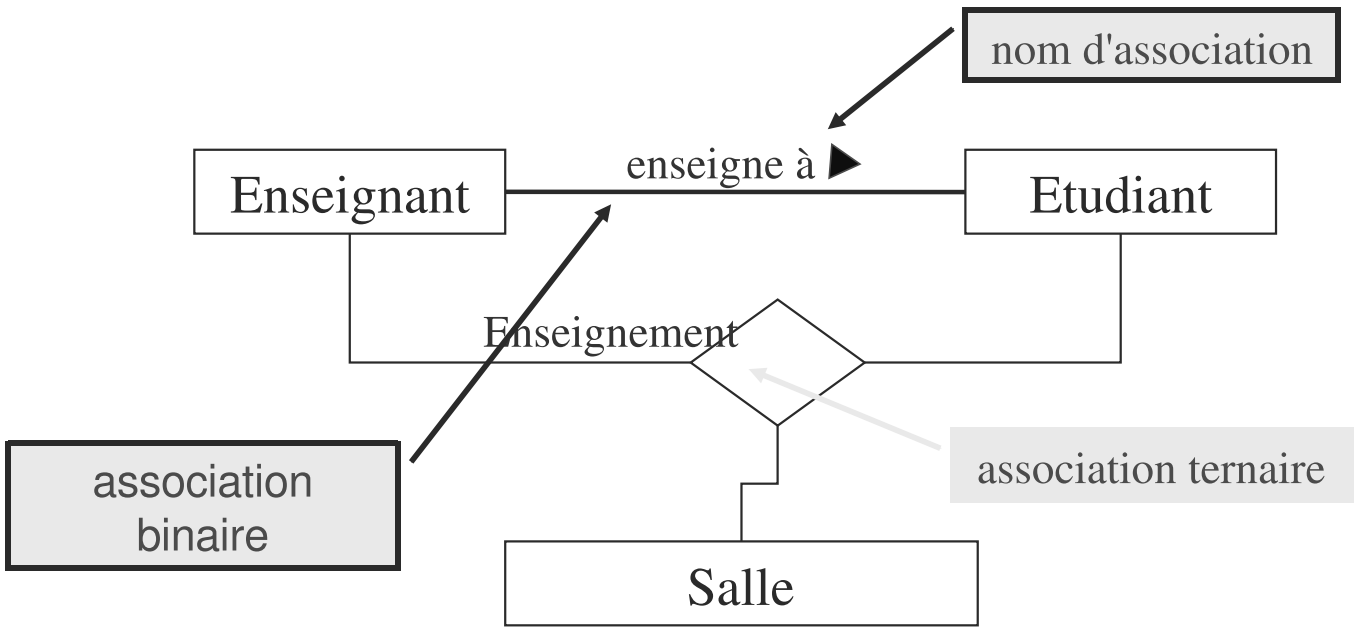
Association dérivée d'autres associations, marquée par : /.



{Personne.employeur=Personne.dept.structure}

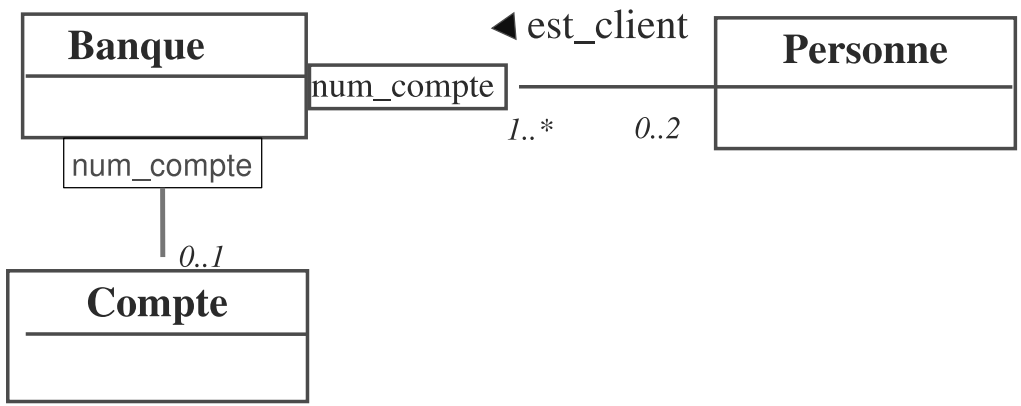
Associations n-aires

Les associations peuvent être d'arité 3, 4, ...



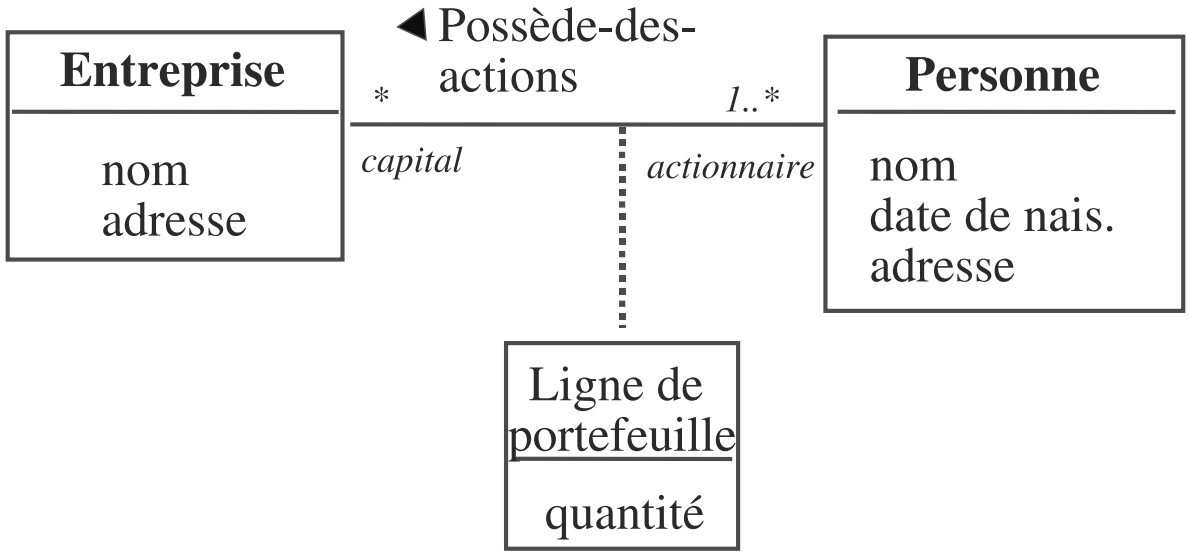
Associations qualifiées

- Un qualifieur sur une association permet de sélectionner un sous-ensemble dans l'association
- Un qualifieur peut être typé (comme un attribut)

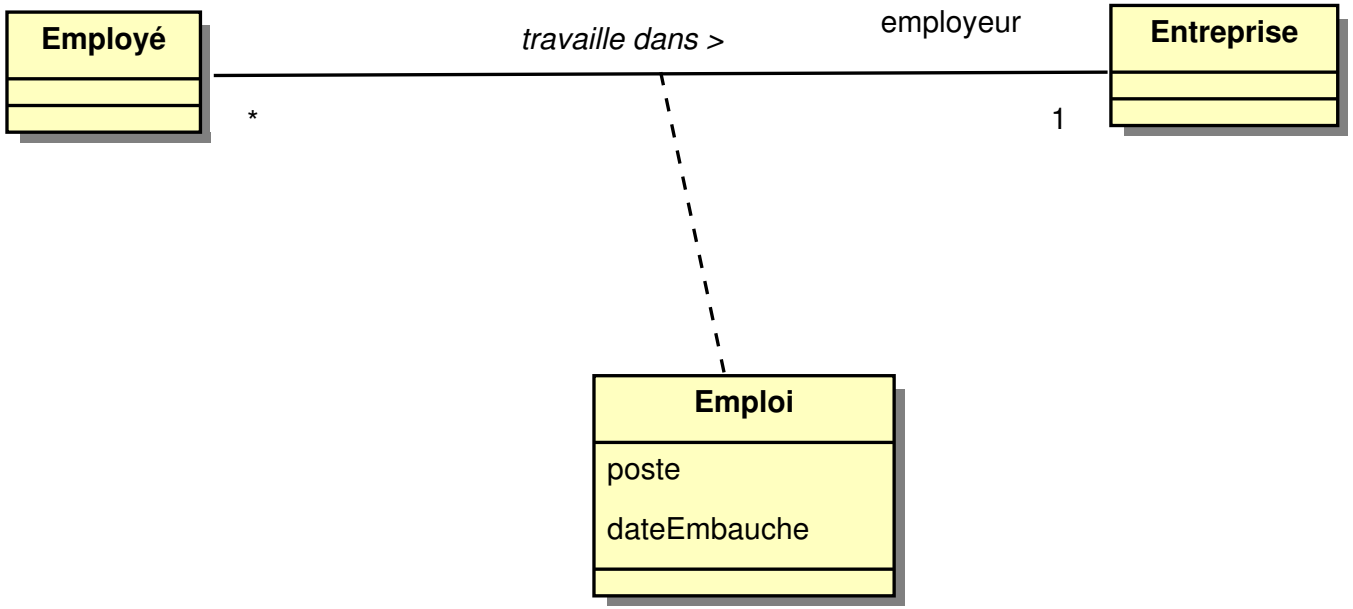


Classes d'association

- Permet de rajouter des informations sur une association complexe (de réifier l'association)
- C'est une classe à part entière ⇒ peut être liée à d'autres classes



Classes d'association, autre exemple

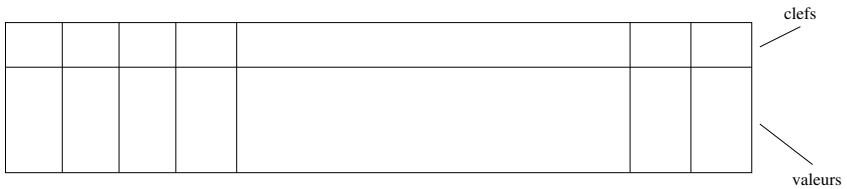


Sommaire

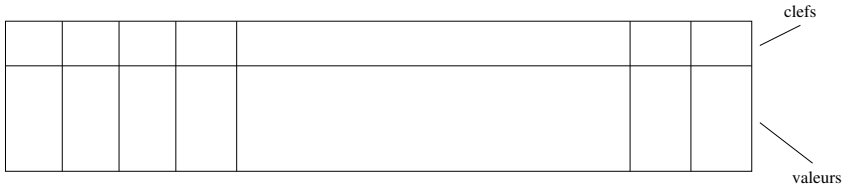
Les tables de hachage : principe d'utilisation

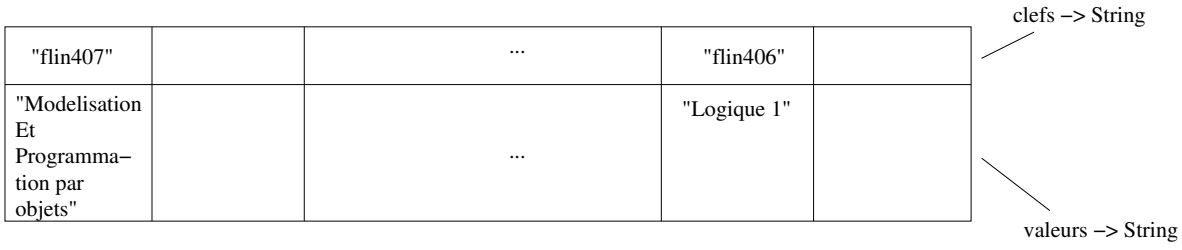
- Une table de hachage est une table indexée par une clef = elle implémente une table d'association entre une clef (un index) et une valeur.
- Par ex. utilisée pour traduire une association qualifiée.
- Exemple. Une table de hachage de comptes bancaires, indexés par leur numéro de compte, que l'on sait unique.
  - clef = numéro de compte, de type entier.
  - La table contient des comptes, accès direct grâce à la clef
- Intérêt des tables de hachage : accès très rapide à une valeur à partir de la clef
- Nous n'étudions pas ici le mécanisme sous-jacent mais juste l'utilisation des tables de hachage

Tables de hachage : illustration

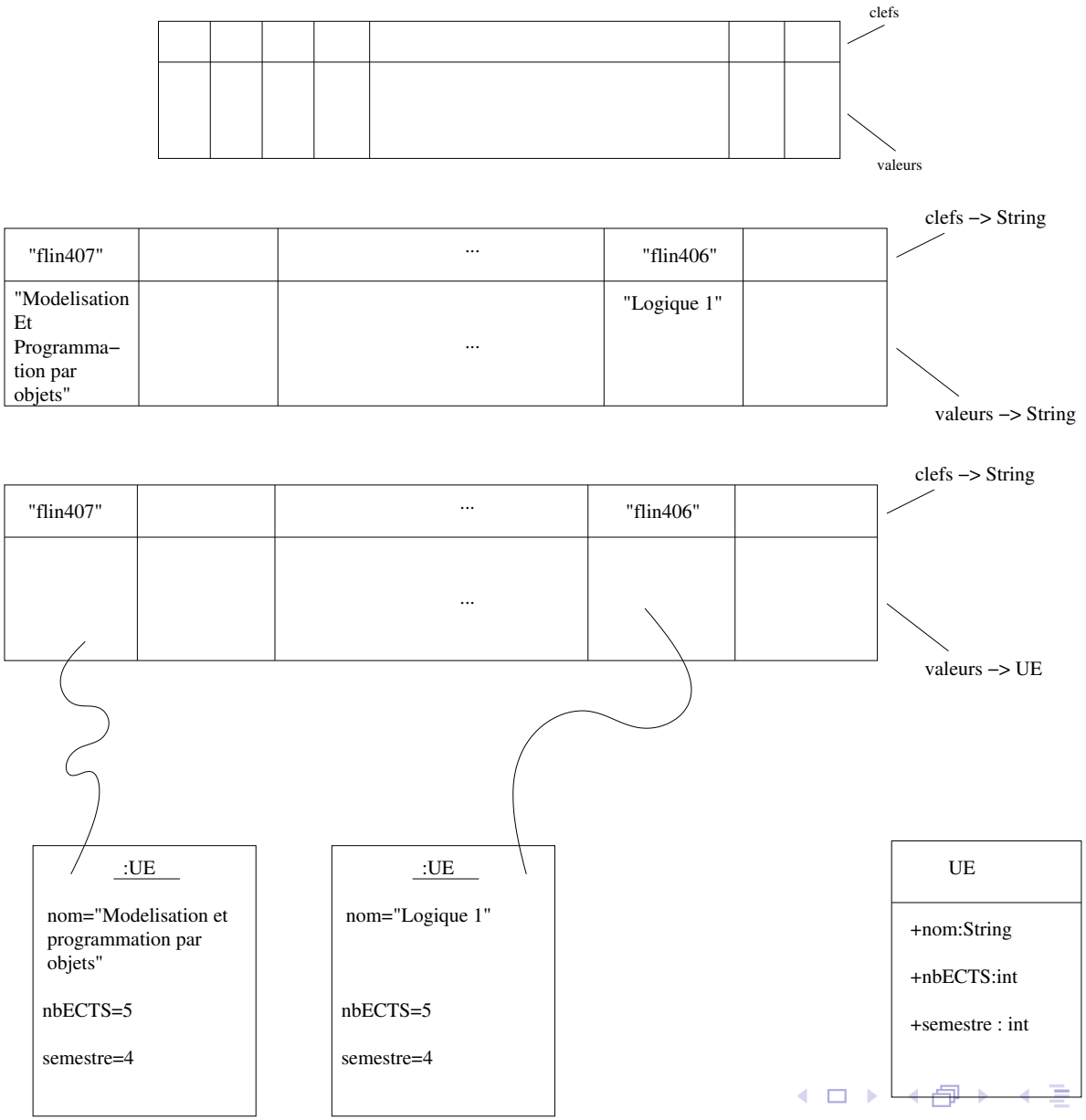


Tables de hachage : illustration





Tables de hachage : illustration



Les tables de hachage : Hashtable

Une classe pour gérér les tables de hachage

```
Hashtable<TClef, TValeur> table=new Hashtable<TClef, TValeur>();
```

Exemples de méthodes pour une table d'objets de type V et de clefs de type K :

- `Object get(K key)` Retourne l'objet de clef `key` ou `null` s'il n'y en a pas.
- `V put(K key, V value)` Ajoute l'élément `value` avec comme clef `key`. S'il existait déjà un élément de même clef, cet élément est retourné (et écrasé dans la table par `value`).
- `V remove(Object key)` Retire l'élément de clef `key` de la table.
- `Collection<V> values()` Retourne une collection des valeurs contenues dans la table

Les tables de hachage : exemple Java

```
Hashtable<String, Integer> numbers
  = new Hashtable<String, Integer>();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));

Integer n = numbers.get("two");
if (n != null) {
   System.out.println("two = " + n);
}
```

Autre exemple en Java

```
Hashtable<String, UE> ues_licence
  = new Hashtable<String, UE>();
ues_licence.put("flin407", new UE("Modelisation et
  programmation orientée objects",5,4));
ues_licence.put("flin406", new UE("Logique 1",5,4));

UE mpoo = ues_licence.get("flin407");
```

UML vers Java : ce qu'on ne peut pas traduire directement

- Les classes d'association : on les transforme en général en classes "normales" fortement associées aux autres classes
- Les associations ternaires ou n-aires, avec n>2 : on réifie en général l'association (elle est implémentée comme une classe)

Interfaces et spécialisation multiple

Conception par Objets, GLIN404

6 mars 2017

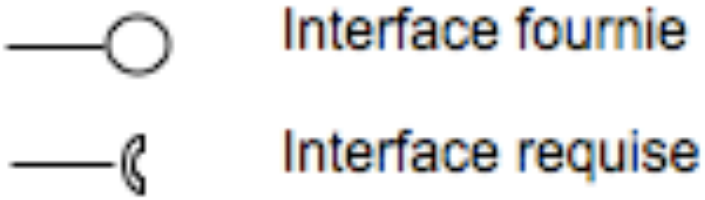
Sommaire

La notion d'interface dans le cadre de la modélisation

- zone de contact
- façade pour une implémentation (classe, programme)
- décrit par exemple pour une classe :
  - ce qu'elle peut offrir comme services à un programme
  - ce qu'elle requiert comme services de son environnement ou du programme
  - l'un des rôles qu'elle peut jouer

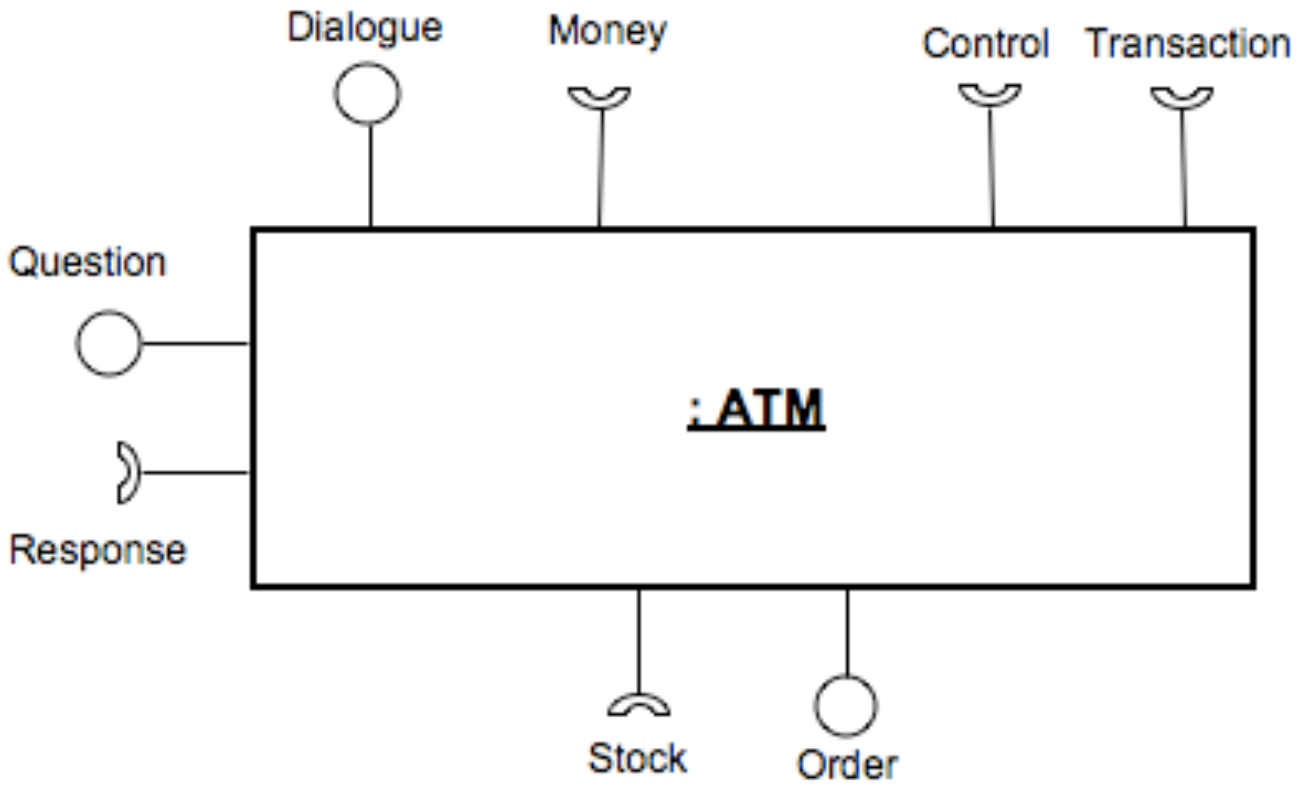
Interface en UML

- Ensemble nommé de propriétés publiques
- Constituant un service cohérent offert par une classe
- Ne spécifie pas la manière dont ce service est implémenté
- Si l'interface déclare un attribut, celui-ci n'est pas forcément implémenté dans la classe, mais doit apparaître aux observateurs

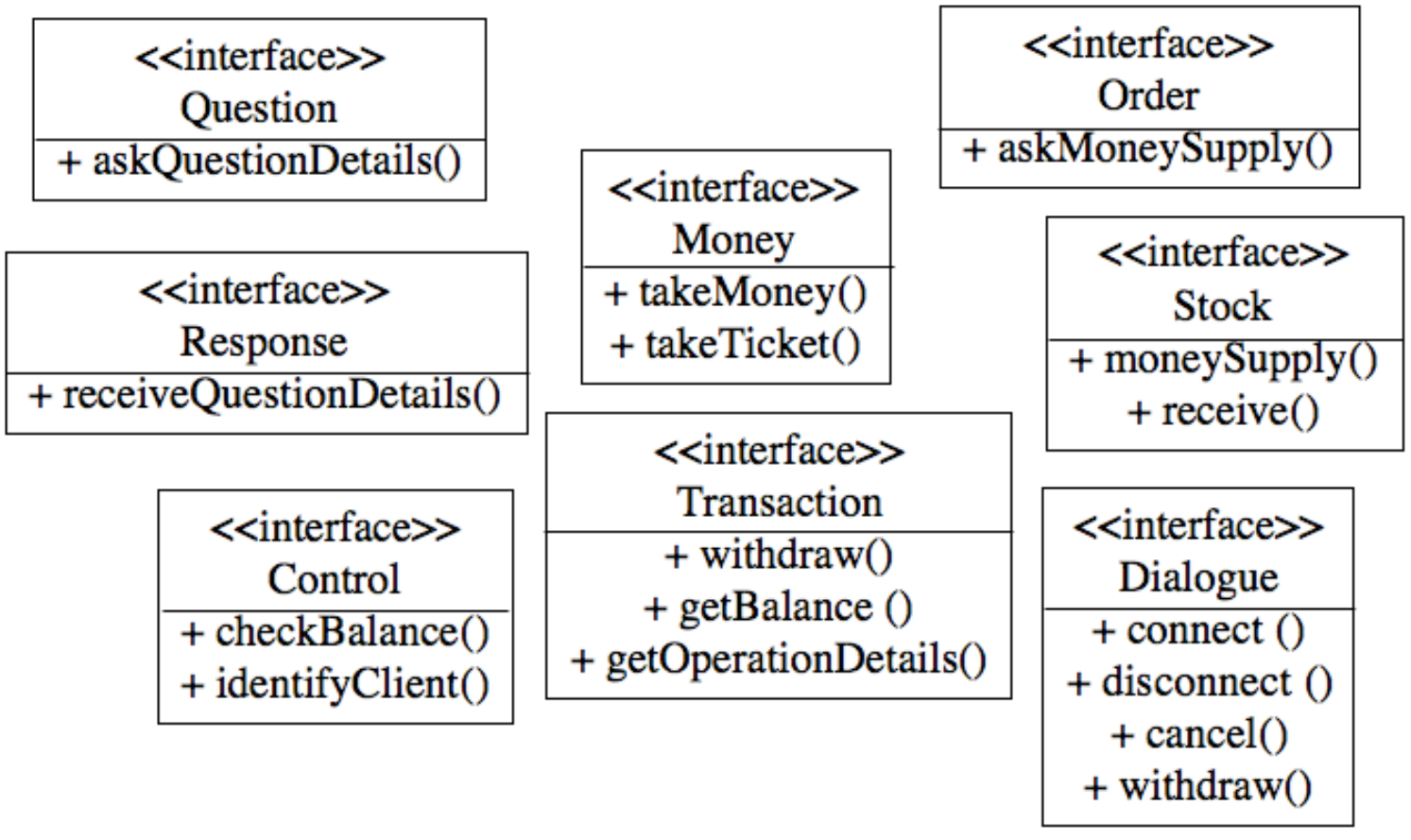


Un composant en UML

Brique logicielle de haut niveau décrite par des interfaces

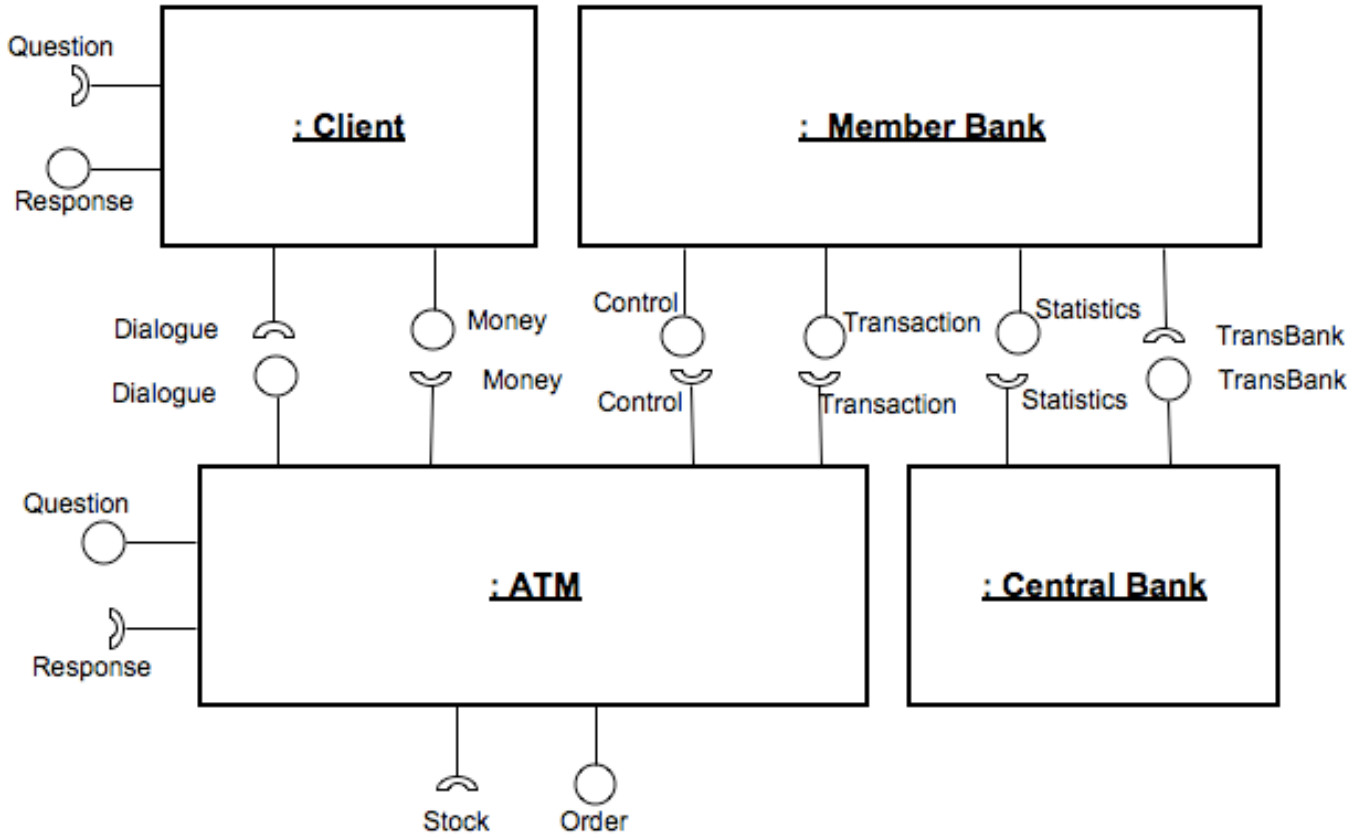


Interfaces en UML



Un assemblage de composants en UML

Architecture de haut niveau d'un logiciel



Interface en Java

- Se limite à présenter des services fournis par une classe
- Un ensemble de :
  - signatures de méthodes publiques (méthodes d'instances abstraites)
  - variables de classes constantes et publiques (constantes statiques)

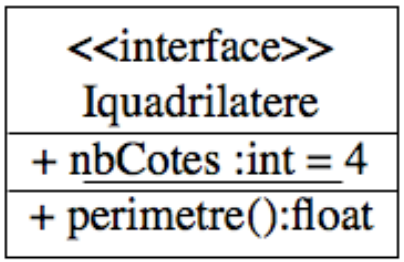
Interface en Java

Améliorer le code :

- on décrit des types de manière plus **abstraite** qu'avec les classes et par conséquent ces types sont plus **réutilisables** ;
- c'est une technique pour **masquer l'implémentation** puisqu'on découple la partie publique d'un type de son implémentation ;
- on peut écrire du **code plus générique** (plus général), au sens où il est décrit sur ces types plus abstraits ;
- les relations de spécialisation entre les interfaces (d'une part) et entre les classes et les interfaces (d'autre part) relèvent de la **spécialisation multiple**, ce qui facilite l'organisation des types d'un programme.

Sommaire

Une interface quadrilatère en UML



Définition d'une Interface

```
public interface Iquadrilatere
{
  public static final int nbCotes = 4;
  public abstract float perimetre();
}
```

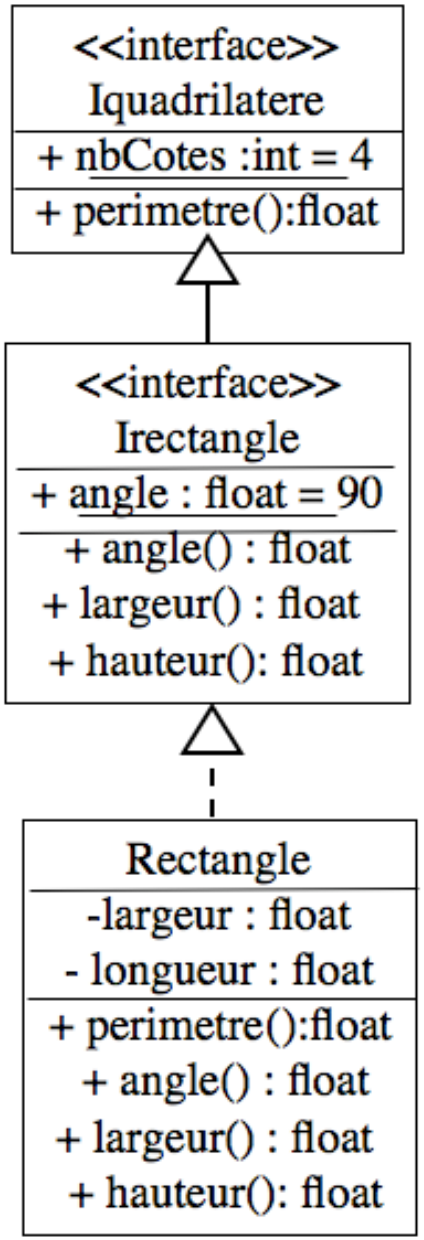
ou (en enlevant les mots-clefs obligatoires)

```
public interface Iquadrilatere
{
  int nbCotes = 4;
  float perimetre();
}
```

Spécialisation d'une interface

```
public interface Irectangle extends Iquadrilatere
{
float angle = 90;
float angle();
float largeur();
float hauteur();
}
```

Interface et réalisation par une classe en UML



Interface et réalisation par une classe en Java / solution 1

L'implémentation utilise deux attributs de type réel

```
public class Rectangle implements Irectangle
{
private float largeur,hauteur;
public Rectangle(){}
public Rectangle(float l, float h){largeur=l; hauteur=h;}

public float perimetre(){return 2*largeur()+2*hauteur();}
public float angle(){return Irectangle.angle;}
public float largeur(){return largeur;}
public float hauteur(){return hauteur;}
}
```

Interface et réalisation par une classe en Java / solution 2

L'implémentation uilise un tableau de deux réels

```
class RectangleTab implements Irectangle
{
private float tab[]=new float[2];
public RectangleTab(){}
public RectangleTab(float l, float h){tab[0]=l; tab[1]=h;}

public float perimetre(){return 2*largeur()+2*hauteur();}
public float angle(){return Irectangle.angle;}
public float largeur(){return tab[0];}
public float hauteur(){return tab[1];}
}
```

Limite des interfaces

On ne peut pas factoriser de code commun dans une interface
On peut introduire une classe (abstraite) pour cette factorisation

```
abstract public class RectangleAbs implements Irectangle
{
public RectangleAbs(){}
public float perimetre(){return 2*largeur()+2*hauteur();}
public float angle(){return Irectangle.angle;}
}
```

Abstraite car :

- Elle ne propose pas de modèle d'implémentation en mémoire (pas d'attributs)
- Elle n'implémente pas `largeur` ni `hauteur`

Interface et réalisation par une classe en Java
Retour sur la solution 1

```
public class Rectangle extends RectangleAbs
{
private float largeur,hauteur;
public Rectangle(){}
public Rectangle(float l, float h){largeur=l; hauteur=h;}

public float largeur(){return largeur;}
public float hauteur(){return hauteur;}
}
```

Interface et réalisation par une classe en Java
Retour sur la solution 2

```
public class RectangleTab extends RectangleAbs
{
private float tab[]=new float[2];
public RectangleTab(){}
public RectangleTab(float l, float h){tab[0]=l; tab[1]=h;}

public float largeur(){return tab[0];}
public float hauteur(){return tab[1];}
}
```

Sommaire

Description de comportements génériques

Noter : seules des interfaces et méthodes abstraites sont invoquées

```
public class StockRectangle
{
Vector<Irectangle> listeRectangle = new Vector<Irectangle>();

public void ajoute(Irectangle r){listeRectangle.add(r);}

public float sommePerimetres()
{
  float sp=0;
  for (int i=0; i<listeRectangle.size(); i++)
  {sp+=listeRectangle.get(i).perimetre();}
  return sp;
}
}
```

Description de comportements génériques

Noter : on peut mettre des rectangles de toutes sortes dans le stock de rectangles

```
public class TestStockRectangle
{
  public static void main (String [] arg)
  {
        StockRectangle st = new StockRectangle();
        st.ajoute(new RectangleTab(3,8));
        st.ajoute(new Rectangle(2,9));
.....
  }
}
```

Sommaire

Spécialisation multiple

- moins contraignante
- plus naturelle pour exprimer des relations de classification quelconques
- permet une factorisation maximale dans tous les cas

Complétons nos interfaces

```
public interface ILosange extends Iquadrilatere
{
float cote();
}
```

Un carré est à la fois un rectangle et un losange :

```
public interface ICarre extends IRectangle, ILosange
{
}
```

Discussion

- En utilisant seulement des classes (pour lesquelles on n'a que de l'héritage simple), les carrés ne pourraient être à la fois des rectangles et des losanges, comme c'est pourtant le cas dans le domaine des mathématiques. Conséquence : une méthode admettant des losanges en paramètre ne pourrait admettre des carrés !
- Toutes les propriétés ne pourraient pas factorisées, certaines seraient redondantes, comme `cote()` ou `angle()`.
- La multi-spécialisation que nous avons pu faire sur les interfaces limite ces problèmes.

Sommaire

Interfaces marqueurs

- vides d'opérations et de constantes de classes
- précisent la sémantique et indiquent dans quel contexte leurs objets peuvent être utilisés
- implémenter ces interfaces marqueurs n'est pas toujours suffisant pour obtenir le comportement attendu, mais c'est nécessaire

cloneable les objets peuvent être clonés : on peut leur appliquer une méthode `clone`, `protected` dans la classe `Object`, et qui doit être redéfinie `public` dans la classe concernée.
serializable les objets peuvent être « sérialisés » c'est-à-dire écrits dans un flux de données. `readObject` et `writeObject` sont réécrites si on désire une sérialisation particulière.

Comparaison d'objets et tris

L'API définit une interface `Comparable` dont le code est le suivant.

```
public interface Comparable<T>
{
int compareTo(T o);
}
```

`compareTo` retourne -1, 0, ou 1 suivant si l'objet receveur est plus petit, égal ou plus grand que le paramètre. Cette opération est notamment utilisée pour les opérations de tri de la classe `Collections`

Collections et itérateurs

L'API 1.5 a ajouté pour les collections une interface bien pratique qui se définit comme suit.

```
public interface Iterable<T>
{
Iterator<T> iterator();
}
```

Collections et itérateurs

Elle permet de parcourir les objets qui l'implémentent avec les méthodes usuelles de `Iterator`, en l'occurrence `hasNext()` et `next()` :

```
public float sommePerimetres()
{
  float sp=0;
  Iterator<Irectangle> It=listeRectangle.iterator();
  while (It.hasNext()){sp+=It.next().perimetre();}
  return sp;
}
```

Collections et itérateurs

... mais également avec une forme particulière de l'instruction `for` :

```
public float sommePerimetres()
{
  float sp=0;
  for (Irectangle r:listeRectangle)
   {sp+=r.perimetre();}
  return sp;
}
```

Hiérarchie des collections

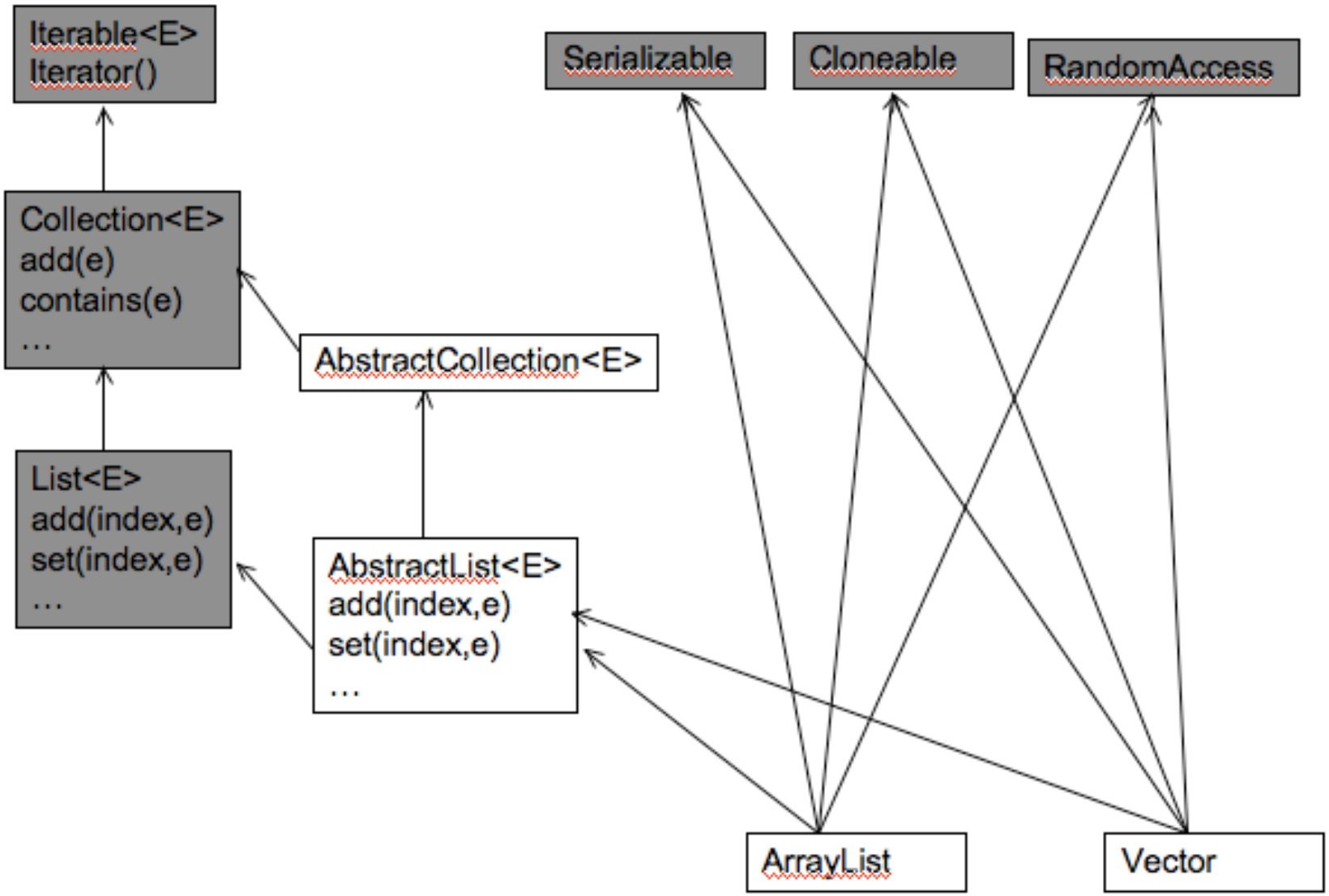


Figure – Extrait des collections

Illustration de l'usage des interfaces
Tri
Sérialisation

Conception par Objets 1, HLIN406

6 mars 2017

Sommaire

Interface IPassager

On représente les passagers d'une compagnie aérienne par l'interface suivante

```
public interface IPassager {
      String nomPrenom(){}
      String numeroPlace(){}
      String programmeFidelite(){}
}
```

Commentez ...

Interface IBouteille

On représente les bouteilles par l'interface suivante

```
public interface IBouteille {
      double volumeMax;
      double volume();
}
```

Commentez ...

Interface IBouteille

On représente les bouteilles par l'interface et la classe suivante

```
public interface IBouteille {
     double volumeMax;
     double volume();
}

public class BouteilleEau extends IBouteille{
...
}
```

Commentez ...

Interface IProduit

On représente les produits d'un magasin par l'interface et la classe suivantes

```
public interface IProduit {
     double tauxTVA = 19.6;
     double prix();
}
public class Produit implements IProduit
{
     private double prix;
     public Produit(double prix) {this.prix = prix;}
     public double getPrix() {return prix;}
     public void setPrix(double prix) {this.prix = prix;}
}
```

Commentez ...

Interface IProduit

On représente des jardins par l'interface et la classe suivantes

```
public interface Ijardin {
     double surface();
     String nomProprietaire();
     String coordGPS();}

public class Jardin implements Ijardin{
// ...
     public double surface(){return 0;}
     public String nomProprietaire(){return "inconnu";}
     public String coordGPS(){return "coordInconnues";}
     public static void main(String[] arg){
          Ijardin jardin1 = new IJardin();
          Ijardin jardin2 = new Jardin();
          Jardin jardin3 = new Jardin();
}}
```

Commentez ...

Interface Ifigure

On représente des figures géométriques et des cercles par les interfaces suivantes

```
public interface IFigureGeometrique {
      int coordX(); int coordY();
      void dessiner();
      void deplacer(int x, int y);
      boolean memeCoordonneesQue(IFigureGeometrique f);
}

public interface ICercle extends IFigureGeometrique {
      double rayon();
      boolean memeCoordonneesQue(ICercle f);
}
```

Commentez ...

Interface Ifigure

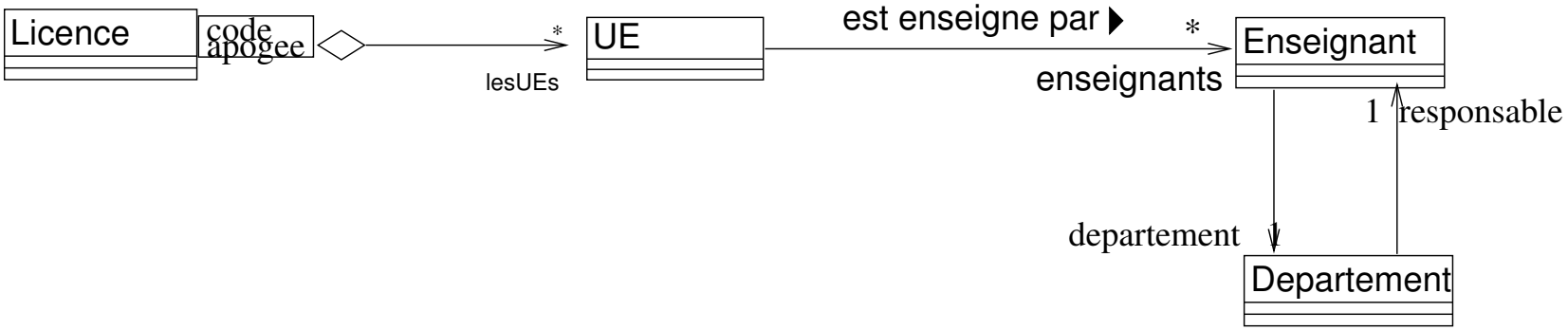
On représente des figures géométriques et des cercles par l'interface et la classe suivantes

```
public interface IFigureGeometrique {
      int coordX(); int coordY();
      void dessiner();
      void deplacer(int x, int y);
      boolean memeCoordonneesQue(IFigureGeometrique f);
}

abstract class Cercle implements IFigureGeometrique {
      public abstract double rayon();
      public boolean memeCoordonneesQue(Cercle f)
                {/* ...  */ return true;}
}
```

Sommaire

Etude de cas Licence, UE, Enseignant, Département



Quelques éléments sur la classe Licence

```
public class Licence{
    private String specialite;
    private int nbInscriptions;
    private Hashtable<String,UE> lesUEs
        =new Hashtable<String,UE>();
        // ...
    public UE getUEByCode(String codeApogee){
        return lesUEs.get(codeApogee);}
    public void ajoutUE(UE ue){
        lesUEs.put(ue.getCodeApogee(), ue);}
}
```

Quelques éléments sur la classe UE

```
public class UE{
    private String codeApogee;
    private String nom;
    private Vector<Enseignant> enseignants
           = new Vector<Enseignant>();

...
}
```

Quelques éléments sur la classe Enseignant

```
public class Enseignant{
    private String nom;
    private String prenom;
    private int anneeRecrutement;
    private Departement departement;
//.....
```

Quelques éléments sur la classe Département

```
public class Departement
{
private String nom;
private Enseignant responsable;

//....
}
```

Questions de tris

- Comparer les départements d'enseignement selon :
  - leur nom
- Trier les enseignants d'une UE selon :
  - leur nom
  - leur ancienneté
  - leur département

Trier les enseignants d'une UE selon un critère

- Savoir comparer les enseignants selon ce critère
- Trier le vecteur `enseignants` de la classe `UE`

Trier Collections.sort

- `public static void sort(List<T> list)`
- Sorts the specified list into ascending order, according to the natural ordering of its elements.
- All elements in the list must implement the Comparable interface.

```
public Interface Comparable<T>{
int compareTo(T o) }
// ...
e1.compareTo(e2)
// Returns a negative integer, zero, or a positive integer
// as this object is
// less than, equal to, or greater than the specified object.
```

Trier les enseignants par leur nom

- Donner un ordre naturel aux enseignants d'après l'ordre lexicographique de leur nom
- Utilise le fait qu'il existe un ordre naturel sur les String (par implémentation de `Comparable` et de `compareTo`)

```
public class Enseignant
     implements Comparable<Enseignant>{

   public int compareTo(Enseignant e){
      return this.getNom().compareTo(e.getNom());}
}
```

Trier les enseignants par leur nom

- Appeler la méthode `sort` qui utilise cet ordre naturel
- Le vecteur `enseignants` est modifié par le tri

```
public class UE{
//....
   public void trieEnseignantParNom()
      {Collections.sort(enseignants);}
}
```

Trier les enseignants par leur ancienneté

- Ecrire un nouvel ordre naturel ?
- Mais on ne peut le faire qu'en réécrivant la méthode `compareTo` de `Enseignant` (on perd l'autre)
- solution : écrire une classe `Comparator` spécialisée dans la comparaison des enseignants suivant l'ancienneté

Trier avec un comparateur Collections.sort

- public static void sort(List<T> list, Comparator<T> c)
- Sorts the specified list according to the order induced by the specified comparator.

```
public Interface Comparator<T>{
   int compare(T o1, T o2)
   // Compares its two arguments for order.
   boolean  equals(Object obj)
   // Indicates whether some other object is "equal to"
   // this comparator.
}
```

Comparateur d'enseignants par ancienneté

```
public class ComparatorEnseignantParAnciennete
      implements Comparator<Enseignant>
{
      // retourne un nb > 0 si e1 est plus ancien que e2
      // 0 si même ancienneté
      // un nb < 0 si e1 est moins ancien que e2
      public int compare(Enseignant e1,Enseignant e2){
            return e1.getAnneeRecrutement()
                  - e2.getAnneeRecrutement();}
}
```

Tri des enseignants par ancienneté

```
public class UE{
//....
public void trieEnseignantParAnciennete(){
   Collections.sort(enseignants,
       new ComparatorEnseignantParAnciennete());}
}
```

Tri des enseignants par département

```
public class Departement
     implements Comparable<Departement>{

public int compareTo(Departement d)
       // sur le nom du département
{
   return this.getNom().compareTo(d.getNom());
}

}
```

Tri des enseignants par département

```
public class CompareEnseignantParDepartement
     implements Comparator<Enseignant>
{
    public int compare(Enseignant e1,Enseignant e2){
        int resultat=e1.getDepartement()
             .compareTo(e2.getDepartement());
        if (resultat==0){ // même département
            resultat=e1.compareTo(e2);
        }
        return resultat;
    }
}
```

Tri des enseignants par département

```
public class UE{
//....
public void trieEnseignantParDepartement(){
   Collections.sort(enseignants,
      new CompareEnseignantParDepartement());
}
```

Tri des enseignants par tout critère passé en paramètre sous forme d'un comparateur

```
public class UE{
//....
public void trieEnseignantParCritere
       (Comparator<Enseignant> critere){
              Collections.sort(enseignants, critere);
}
```

Appel des méthodes de tri

```
public static void main(String[] args){
    Departement info = new Departement("Informatique");
    Departement math = new Departement("Mathematique");
    Licence l=new Licence();
    l.setSpecialite("info");
    UE ue=new UE();
    ue.setCodeApogee("ULIN407");
    ue.setNom("Conception et programmation par objets");
    Enseignant mh=new Enseignant("Huchard", "M",info, 1992);
    Enseignant cn=new Enseignant("Nebut", "C",info, 2005);
    Enseignant mc=new Enseignant("Cuer", "M",math, 2000);
    ue.addEnseignant(mh);
    ue.addEnseignant(mc);
    ue.addEnseignant(cn);
    l.ajoutUE(ue);
// ....
 }
```

Appel des méthodes de tri

```
 public static void main(String[] args){
     // ....

System.out.println("Liste triee par dept ------");
ue.trieEnseignantParDepartement();

System.out.println("Liste triee par anciennete ------");
ue.trieEnseignantParAnciennete();

System.out.println("Liste triee par nom ------");
ue.trieEnseignantParNom();

System.out.println("Liste triee par anciennete
          en passant le critère ------");
ue.trieEnseignantParCritere
         (new ComparatorEnseignantParAnciennete());
  }
```

Sommaire

Flot

- Flot = objet capable de transférer une suite de données
  - d'une source externe (ex. fichier, clavier) vers le programme
  - du programme vers une cible externe (ex. fichier, écran)
- Flots en Java - plusieurs dizaines de classes
  - flots binaires (octets)
  - flots de caractères (texte)
    - avec transformation
    - UNICODE -> format texte de la machine hôte

Fichier texte, exemple

```
public static void main(String[] arg) throws IOException{
  BufferedReader fc = new BufferedReader
    (new InputStreamReader (System.in));

  BufferedWriter ff = new BufferedWriter
    (new FileWriter ("essai2015.txt"));

  System.out.println
              ("Entrez des lignes (Return pour terminer)");
  String s = fc.readLine();
  while (s.length() != 0)
    {
    ff.write(s);
     ff.newLine();
    s = fc.readLine();
  }
  ff.close();
}
```

Fichier d'objets

- la classe de l'objet à sauvegarder et de ses sous-objets doit implémenter l'interface `java.io.Serializable`
- l'objet est décomposé selon ses attributs jusqu'au niveau où les données sont de type primitif ou String

Par exemple on veut sauver les informations d'une licence

```
public class Licence implements Serializable{ ...
public class UE implements Serializable{ ...
public class Enseignant
       implements Comparable<Enseignant>, Serializable{ ...
public class Departement
       implements Comparable<Departement>, Serializable{ ...
```

Fichier d'objets

Le fait d'implémenter `Serializable` autorise à utiliser :

- java.io.ObjectOutputStream
- public final void writeObject(Object obj) throws IOException

- java.io.ObjectInputStream
- public final Object readObject() throws OptionalDataException, ClassNotFoundException, IOException

Fichier d'objets - sauvegarde

```
 public static void main(String[] args)
      throws FileNotFoundException, IOException{
{
// ....
  ObjectOutputStream flotS =
              new ObjectOutputStream
                    (new FileOutputStream ("essai2015.txt"));
  flotS.writeObject(l);
  flotS.close();
}
```

Fichier d'objets - récupération

```
 public static void main(String[] args)
      throws FileNotFoundException, IOException,
           ClassNotFoundException
{
// ....
  ObjectInputStream flotE =
              new ObjectInputStream
                    (new FileInputStream ("essai2015.txt"));
  Object lic = flotE.readObject();
  Licence licRecuperee = (Licence) lic;
  flotE.close();
}
```

Fichiers d'objets

- automatique (il suffit de déclarer que les classes implémentent `Serializable`)
- permet de gérer facilement des objets complexes
- les fonctions readObject et writeObject peuvent être redéfinies dans la classe (Licence par exemple) pour modifier ce qui est stocké
- inconvénient : les fichiers ne sont pas lisibles (binaires)