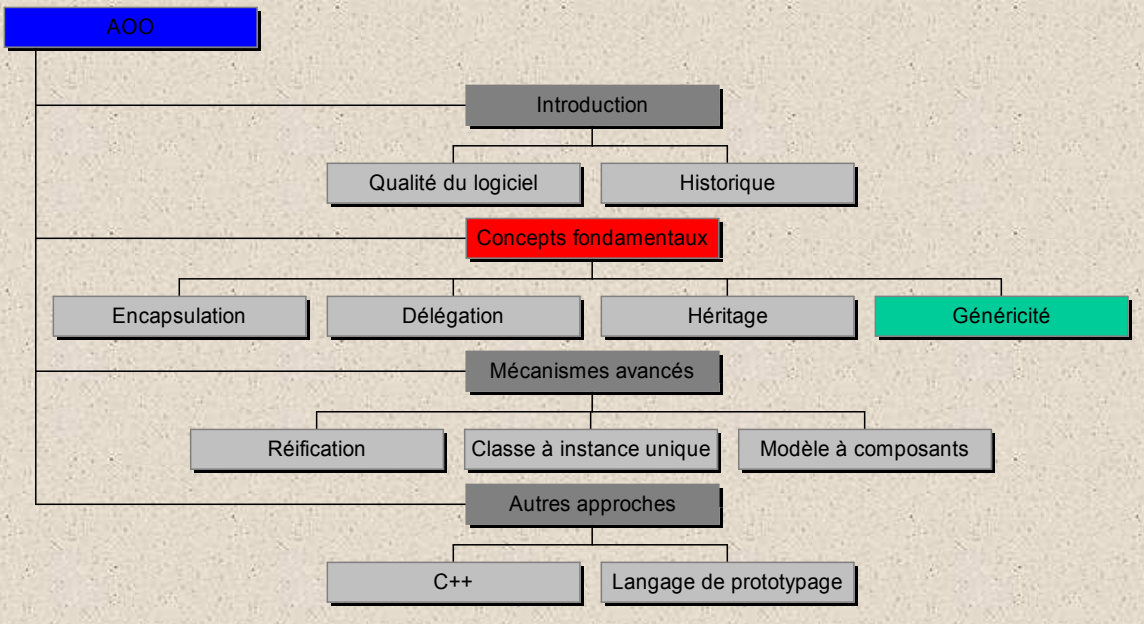
LA GENERICITE

Un paradigme pour l'abstraction

Sommaire



Sommaire

Généricité & Java

Ce chapitre avait peu d'intérêt pour Java, cependant IL me paraissait essentiel dans le cadre de la Conception Orientée Objet, là était la raison de sa présence dans ce cours.

Désormais, avec la version 1.5 de Java la généricité fait partie intégrante du langage

Vive les classes plurielles

La généricité

La notation (langage) doit permettre au niveau d'une classe d'exprimer le concept associé (ce que la classe représente), rien de plus :

- la sur-spécification est une gène.

On doit pouvoir s'abstraire des informations non pertinentes

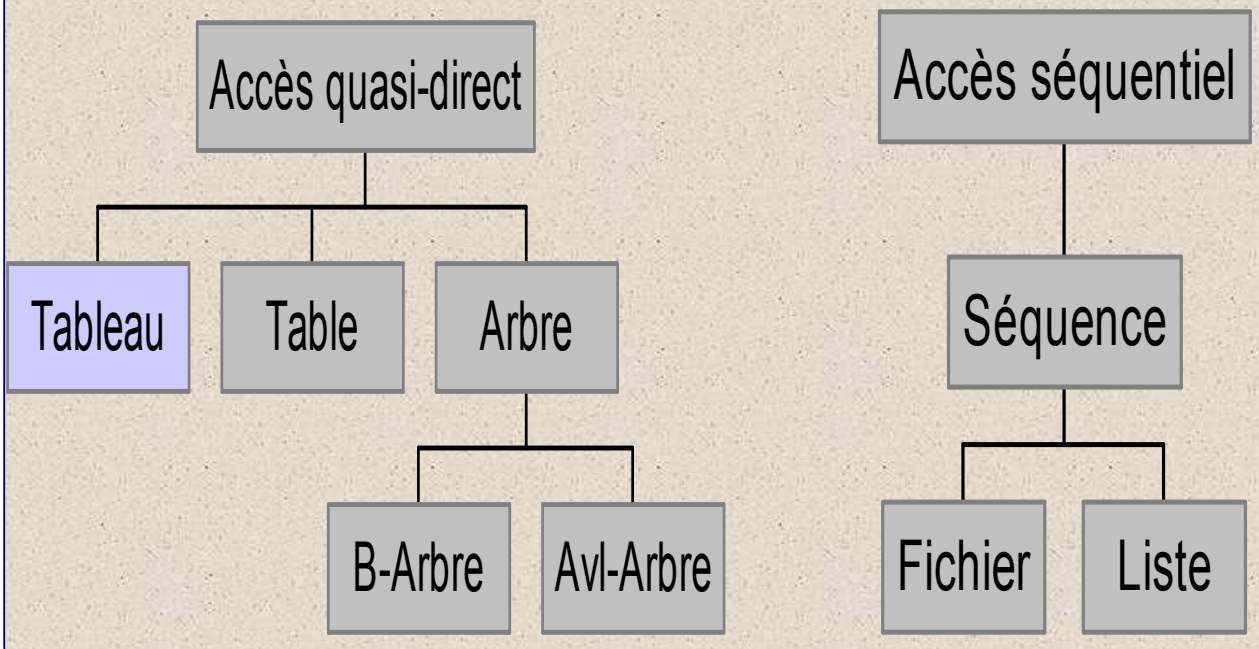
- Un mécanisme d'abstraction : le paramétrage
- Au niveau modèle (classe) le paramètre est un type (classe)
  - On nomme ce paramétrage : la généricité

La généricité n'a de sens que dans le cadre de langage compilé

- Un langage non typé est de facto générique
- Cependant certains langages comme C++ intègre dans ce niveau des éléments qui relèvent de l'instanciation (valeurs).

Les structures de données

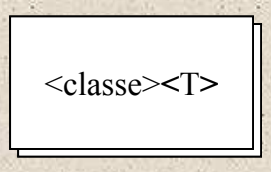
Par définition les structures de données sont des objets génériques



Généricité simple

Le type formel T est utilisé comme type d'une ou plusieurs caractéristiques,

- la classe manipule des informations de nature T. Elle ignore tout de ces informations.



Le type formel T est utilisé comme type effectif lors d'un ou plusieurs héritages génériques,

- après dérivation le type formel T n'est utilisé que comme type d'une ou plusieurs caractéristiques.

Cellule<T>

```
/**
* modélise les fonctions d'une cellule capable
* d'héberger une information
* @author P.Morat
* @version    1.0
* date : 1/9/99
*/
public class Cellule<T> {
/**
* la valeur hébergée
*/
private T valeur;
/**
* Affecte la cellule avec une information
* @param v    l'information à stocker
*/
public void set(T v) {
    valeur = v;
}
/**
* restitue l'information contenue dans la cellule
* @return     l'information contenue
*/
public T get() {
    return valeur;
}
}
```

Type formel

Contrôle des types applicable

Cellule<Integer> cellule

-- Si une classe possède une telle déclaration de cellule, c'est comme si cellule était du type dont la définition suit :

```
/**
* utilisation d'une cellule
* @author P.Morat
* @version    1.0
* date : 1/9/99
*/
public class Essai {
/**
* Une cellule
*/
private Cellule<Integer> cellule;
/**
* une méthode qqc
*/
public void m() {
    cellule = new Cellule<Integer>();

    cellule.set(new Integer(3));
    Integer i = cellule.get();
    ...
```

Type effectif

```
/** en JAVA
* modélise les fonctions d'une cellule capable
* d'héberger une information
* @author P.Morat
* @version    1.0
* date : 1/9/99
*/
public class CelluleInteger {
/**
* la valeur hébergée
*/
private Integer valeur;
/**
* Affecte la cellule avec une information
* @param v    l'information à stocker
*/
public void set(Integer v) {
    valeur = v;
}
/**
* restitue le contenu de la cellule
* @return    l'information contenue
*/
public Integer get() {
    return valeur;
}
}
```


Limites de la généricité sous cette forme

Trop permissive dans un contexte où l'on cherche à être précis.

– C<T> exprime "T ∈ l'ensemble S des classes existantes "

Elle ne permet qu'une manipulation globale de l'information dont on s'abstrait.

– pas de possibilité de fixer le niveau d'abstraction désiré

– par la déclaration "T x" : x ne peut être vu que comme un objet (classe Object).

La généricité contrainte

La généricité contrainte permet de fixer la nature minimale de l'information dont on s'abstrait.

- Réduire le type de paramètre effectif de la classe "D<T ⊆ C>" à un sous-ensemble de classes $Sc \subseteq S \mid E \in Sc \Rightarrow E$ hérite de C
- Forcer le paramètre effectif à avoir une propriété particulière. Si $E \in Sc$ alors E à au moins les propriétés de C.

La généricité simple correspond à une généricité contrainte où la contrainte est la plus large possible (Object).

- Le paramètre effectif est d'un type descendant de la classe la plus générale.
  - D<T> ≡ D<T ⊆ Object>

Le contrôle de type (principe général)

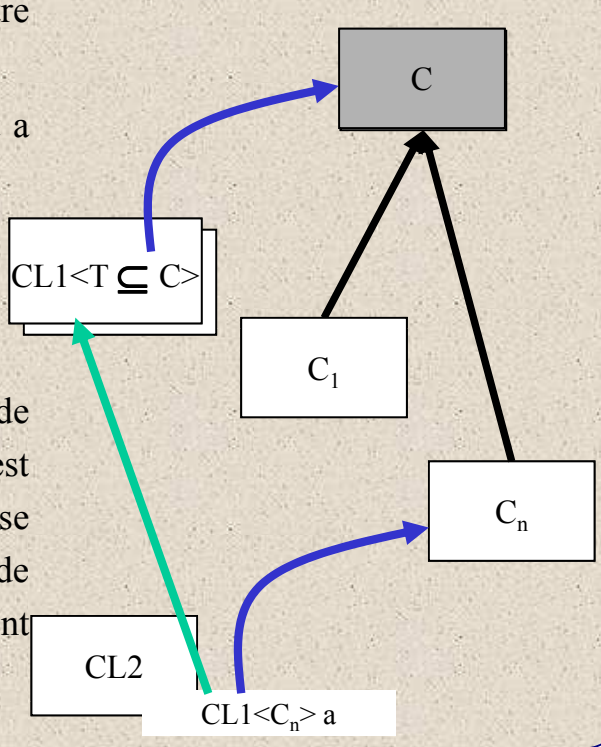
- $T_1 \subseteq T_1' \Leftrightarrow T_1' \leftarrow T_1$
- $T_1<E_1,E_2,\ldots,E_n> \subseteq T_1'<E_1',E_2',\ldots,E_n'>$ ssi
  - $T_1' \leftarrow T_1, E_1' \leftarrow E_1, E_2' \leftarrow E_2, E_n' \leftarrow E_n$

En java l'opérateur ⊆ se note extends

En java tout ceci n'est pas si simple car tout n'est pas si simple

La généricité contrainte : principe

- Soit une classe CL1 ayant un paramètre T contraint par C
- Soit une classe CL2 ayant un attribut a de type CL1 de type effectif C_n
  - T est un nom formel de classe
  - C est un nom de classe effective
  - C_n doit être une sous-classe de C
- Classiquement C est une classe de définition de propriété. A priori c'est une classe abstraite. C_n est une classe qui appartient à un arbre de spécialisation et qui hérite directement ou indirectement de C.

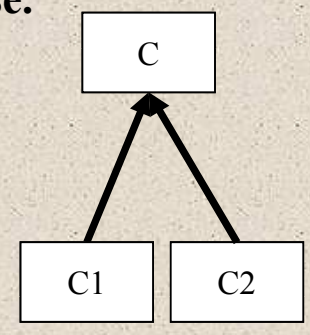


Généricité vs non généricité

On peut remplacer le paramètre générique par sa contrainte.

Ne modifie pas le contrôle interne de la classe.

Modifie l'utilisation que l'on peut faire de la classe.




```
CL1<T ⊆ C>
  T a;
  public void set(T v) {a=v};
  public T get() {return a}
```

```
CL1
  C a;
  public void set(C v) {a=v};
  public C get() {return a}
```

```
CL2
  CL1<C2> u;
  C1 u1;
  C2 u2;
  public void M() {
  u.set(u1);
  u.set(u2);
  u2=u.get();
  }
```

statiquement correct

statiquement incorrect

```
CL2
  CL1 u;
  C1 u1;
  C2 u2;
  public void M() {
  u.set(u1);
  u.set(u2);
  u2=u.get();
  }
```

statiquement incorrect

statiquement correct

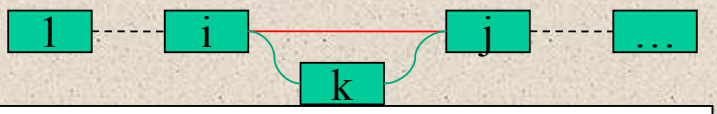
La Généricité avec Java 1.5

Ensemble d'exemples de mise en œuvre de la généricité

- Utilisation de la généricité
  - E1 : Contrainte
    - Cet exemple montre l'intérêt de la généricité contrainte en mettant en évidence la nécessité de connaître certaines capacités liées aux types formels. L'exemple utilisé est la classe **SortedList**.
  - E1' & E1'' : Classe multi-générique
    - Cet exemple montre un cas avec plusieurs paramètres formels, l'exemple utilisé est la classe **Dictionary**
  - E2 : Propriété du type formel vs propriété du type générique
    - Cet exemple montre la différence entre les propriétés du type générique et celles du type formel qui le paramètre. L'exemple utilisé est la classe **SortedListComparable**.
  - E3 : Portée d'une déclaration de type formel
    - Cet exemple montre l'intérêt d'avoir un type formel lié à une entité conceptuelle précise. L'exemple utilisé est la classe **Liste**.
- Généricité & héritage
  - E4 : Classe générique héritant d'un classe générique
    - Cet exemple montre la possibilité d'avoir des hiérarchies d'héritage où se mélangent des classes génériques et non génériques. L'exemple utilisé est la classe **SortedListComparable** (version2)**.**
  - E5 : Classe non générique héritant d'une classe générique
    - Cet exemple montre la possibilité d'avoir des hiérarchies d'héritage où se mélangent des classes génériques et non génériques . L'exemple utilisé est la classe **IntegerSortedList**.
  - E6 : Class générique héritant d'une classe non générique
    - Cet exemple montre la possibilité d'avoir des hiérarchies d'héritage où se mélangent des classes génériques et non génériques . L'exemple utilise les classes **Indice** et **Number**.
    - Le cas des interfaces correspond à l'attribution d'une propriété (héritage de comportement) qui ne nécessite pas de généricité. Comparable est un contre-exemple.
  - E7 : Co-variance des types des paramètres d'une méthode induite par la généricité
    - Comparable
    - Le problème de Cloneable
- Généricité & polymorphisme
  - E8 : Paramètre générique et polymorphisme
    - Cet exemple montre l'intérêt des types effectifs libres co-variants. L'exemple utilisé est une amélioration de la classe **Liste**.
  - E9 : Paramètre générique et polymorphisme
    - Cet exemple montre l'intérêt des types effectifs libres contravariants. L'exemple utilisé est une amélioration de la classe **Liste**.

E1 : SortedList <T⊆Comparable >

```
/**
* modélise les fonctions d'une liste ordonnée sans doublon.
* @author  P.Morat
* @version 1.0
* date : 1/9/99
*/
```



E1 : Contrainte

Cet exemple montre l'intérêt de la **généricité contrainte** en mettant en évidence la nécessité de *connaître certaines capacités* liées aux **types formels**.

L'exemple utilisé est la classe **SortedList** qui permet de construire et gérer des listes dans lesquelles les éléments sont rangés selon la relation d'ordre qui les caractérise.

```
...
}
```


E1' : Dictionary <K,V>

E1' : multi-générique

Cet exemple montre l'intérêt d'avoir plusieurs types formels dans la déclaration d'une classe.

L'exemple utilisé est l'interface **Dictionary** qui spécifie une fonction d'un ensemble K dans un ensemble V

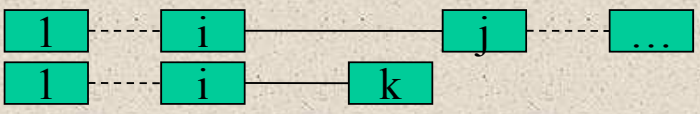
```
}
```


E1'' : Dictionary <K ⊆ Comparable, V>

```
public interface Dictionary<K extends Comparable,V> {
/**
* insère le couple {key,value} dans la tabulation,
* si la clé key existe déjà l'association est modifiée
* avec la nouvelle valeur value.
* @param key  la clé
* @param value      F(key)
* @return la valeur insérée
*/
public V put(K key, V value);
/**
* restitue la valeur associée à la clé key
* @param key la clé
* @return la valeur associée à la clé ou null si celle-ci
*            n'est pas présente.
*/
public V get(K key);
}
```


E2 : SortedListComparable <T⊆Comparable>

```
/**
* modélise les fonctions d'une liste ordonnée sans doublon.
* @author  P.Morat
* @version 1.0
* date : 1/9/99
*/
```



E2 : Propriété du type formel vs propriété du type générique

Cet exemple montre la différence entre les propriétés du **type générique** et celles du **type formel** qui le paramètre.

L'exemple utilisé est la classe **SortedListComparable** qui gère des informations sur lesquelles il existe une relation d'ordre et pour laquelle on définit un relation d'ordre permettant de dire si une telle liste est plus petite ou plus grande qu'une autre.

E3 : Liste <A>

```
public class Couple<A,B>{
public A fst; public B snd;
public Couple (A fst, B snd){this.fst=fst; this.snd=snd;}
```

E3 : Portée d'une déclaration de type formel

Cet exemple montre l'intérêt d'avoir un type formel lié à une entité conceptuelle précise. La **portée** d'un **type formel** ne doit pas être trop importante au risque de nuire à *l'expressivité (sur-spécification)*.

L'exemple utilisé est la classe **Liste** dans laquelle on souhaite définir une méthode «zip» qui étant donnée une liste de même taille, dont le type des éléments est quelconque, construit la liste des couples comme ci-dessous :

{"a", "b", "c"}.zip({1,2,3}) = {<"a",1>,<"b",2>,<"c",3>}.

E3 : Inférence de type

Lors de l'utilisation d'un type générique, on fixe le type effectif qui se substitute au type formel.

- Dans le cas classique :
  - Liste<Integer> … c'est le type effectif Integer qui est affecté au type formel A.
- Dans le cas d'un type formel à portée réduite à une méthode, comment ce type effectif est-il déterminé ?
  - Première solution : de façon déclarative comme dans le cas général,
    - Liste<Integer> l1 = new Liste<Integer>();
    - l1.…;
    - l1.<Integer>zip(l1);
  - Seconde solution : on laisse le compilateur inférer lui-même le type effectif
    - l1.zip(l1);
    - Dans ce cas l1 étant de type Liste<Integer> le type formel B est substitué par le type effectif Integer
- Dans des cas plus complexes le type effectif peut être le résultat du compromis de plusieurs contraintes (borne supérieure)

E4 : SortedList<T ⊆ Comparable>

```
public class SortedList<T extends Comparable> {
List<T> l;
...
}
```

E4 : Classe générique héritant d'un classe générique

Cet exemple montre la possibilité d'avoir des hiérarchies d'héritage où se mélangent des classes génériques et non génériques.

L'exemple utilisé est la classe **SortedList** (version2). Dans la première version de cette classe nous avions utilisé une délégation (SortedList utilisait une Liste).

E5 : IntegerSortedList

```
/**
* modélise les fonctions d'une liste ordonnée sans doublon.
* @author  P.Morat
* @version 1.0
* date : 1/9/99
*/
```

E5 : Classe non générique héritant d'une classe générique

Cet exemple montre la possibilité d'avoir des hiérarchies d'héritage où se mélangent des classes génériques et non génériques .

L'exemple utilisé est la classe **IntegerSortedList**. Cette forme correspond à l'instanciation d'un paquetage ADA ou C++. Son seul intérêt est de faire disparaître la généricité dans un contexte où cela n'aurait pas d'utilité.

E6 : Indice

```
public abstract class Number {
```

E6 : Class générique héritant d'une classe non générique

Cet exemple montre la possibilité d'avoir des hiérarchies d'héritage où se mélangent des classes génériques et non génériques.

L'exemple utilise les classes **Indice** et **Number**. T est le type permettant la représentation des valeurs caractérisant l'indice.

Le cas des interfaces correspond à l'attribution d'une propriété (héritage de comportement) qui ne nécessite pas de généricité. *Comparable est un contre-exemple*.

E6' : Indice

```
public abstract class Number {
```

E6 : Class générique héritant d'une classe non générique

Cet exemple montre la possibilité d'avoir des hiérarchies d'héritage où se mélangent des classes génériques et non génériques.

L'exemple utilise les classes **Indice** et **Number**. T est le type permettant la représentation des valeurs caractérisant l'indice.

Le cas des interfaces correspond à l'attribution d'une propriété (héritage de comportement) qui ne nécessite pas de généricité. *Comparable est un contre-exemple*.

E7 : Cloneable

```
public class Object {
  protected Object clone() throws CloneNotSupportedException{
  ...
}
...
}
public interface Cloneable {}
```

E7 : Problème lié à l'interface Cloneable. L'exemple que l'on utilise ici consiste à mettre en place une forme de passage de paramètre par valeur. Pour assurer que la classe Cellule est garante de l'information qu'on lui confit, on décide qu'elle conserve une copie de la valeur que l'on veut sauvegarder (passage par valeur) et qu'elle restitue une copie de cette information sauvegardée (passage par valeur).

Spécification de l'interface Comparable

```
public interface Comparable {public int compareTo(Object v);}
```

La fonction de comparaison est : compareTo : T_1 x T_2 -> int

Pour l'instant T_2 = Object, mais il est "préférable" que $T_2 \subseteq T_1$ pour que la comparaison ait un sens. Dans l'AOO T_1 est fixé par le destinataire du message compareTo, la spécification ne dépend alors plus que de T_2. La fonction n'est pas totale : compareTo n'a de sens que si on compare des choses comparables. Une manière de faire serait d'imposer la compatibilité du type du paramètre au destinataire public int compareTo(Like(this) v);

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

Il n'en reste pas moins que la comparaison peut ne pas être signifiante car on n'impose pas $T \subseteq T_1$. Pour forcer cet propriété, il faut que le type effectif substitué à T soit compatible à T_1.

Covariance avec les types génériques

Dans cette forme la méthode compareTo support la covariance du type du paramètre car (1) devient (2) quand T est substitué par CL.

(1) `public int compareTo(T  v){...}`

(2) `public int compareTo(CL v){...}`

On a là une contradiction avec les méthodes sans généricité où la surcharge s'impose quand les profils changent.

```
public class CL implements Comparable<CL> {
public int compareTo(CL v){...}
}
public class SortedList<T extends Comparable<T>> {
List<T> l;
public void insert(T v) {
       if(l.size()==0) l.add(v);
       ListIterator<Comparable<T>> it = l.listIterator();
       Comparable<T> e;
       for(   e=it.next();
                    it.hasNext() && e.compareTo(v)< 0;
                    e=it.next());
       if(e.compareTo(v)==0) return;
       if(e.compareTo(v)> 0) it.previous();
       it.add(v);
}
```


Type générique & Polymorphisme

Les types génériques ne sont pas variant-compatibles

- $T<E_1> \not\subset T<E'_1>$ même si $E_1 \subset E'_1$
  - – List<Integer> ⊄ List<Number>
    - » List<Number> toto = new List<Integer>(); // est incorrect
  - – Permet d'éviter les inconsistances à l'exécution :
    - » toto.add(new Double(33.33));
  - – Cependant ne garantit pas une sécurité absolue :
    - » List l = new List<Integer>();
    - » …
    - » List<Double> tutu = (List<Double>) l;

erreur un peu volontaire

Cependant sans possibilité de polymorphisme, l'utilité est réduite

- – On a une forte contradiction avec le principe de base de l'AOO
  - Un type générique ne se comporte pas comme un autre type (non générique)