



IMT Atlantique

Bretagne-Pays de la Loire
École Mines-Télécom

Type, Variances et Généricité

A. Beugnard
DCL – MAPD – C6
2023

1. Conséquences de l'héritage sur les types
2. Retour sur la généricité (variance de type)
3. Définir des classes génériques

Contribution aux compétences

1. Spécification
2. Conception
3. Développement

Je vous invite à prendre des notes...

- 1 Introduction
- 2 Généricité, des besoins
- 3 Généricité par l'exemple en Java

- ▶ Définit un ensemble de valeurs (instances) qui partagent des propriétés :
 - ▶ comportementales : les opérations que l'on peut faire avec
 - ▶ La structure ou la représentation des instances n'est pas considérée
- ▶ En programmation objet, on distingue :
 - ▶ Le type apparent ou déclaré (dit *type statique*)
 - ▶ Le type réel ou instancié (dit *type dynamique*)

- ▶ Sous-typage (idéal) : un type D est un sous-type de U si et seulement si toute propriété de U est une propriété de D
- ▶ Sous-typage (Objet) : la relation d'héritage définit une relation de sous-typage : si D hérite de U alors $D <: U$
- ▶ Sous-typage (Java) : la relation d'héritage (entre classes ou entre interfaces) ainsi que la relation `implements` définissent une relation de sous-typage :
 - ▶ si D hérite de U alors $D <: U$
 - ▶ si D `implements` I alors $D <: I$
- ▶ Le sous-typage $D <: U$ implique une propriété de substitution (*Substituabilité*) : partout où est attendu un $u : U$ on peut mettre un $d : D$ sans risque d'erreurs de type


Avec un système de type sûr la relation *dynamique* $<:_{\text{statique}}$ doit toujours être vérifiée pour n'importe quelle expression correcte du langage

Mais avec un tel système il faut respecter les règles de substituabilité.

- ▶ Soit un objet a de type A
- ▶ Soit un objet r de type R
- ▶ Le compilateur autorise $a = r$; ssi $R <: A$
- ▶ Le compilateur autorise $o.m(r)$ ssi $R <: A$ et $o : O$ a la propriété $m(A)$

- ▶ Soit un objet a de type A (avec la propriété $m(T\ t)$)
- ▶ Soit r tel que $R <: A$
- ▶ Tous les objets r possèdent la propriété $m(T\ t)$
- ▶ En cas d'invocation $r.m(t)$ quel corps de méthode est exécuté ?
 - ▶ Celui de A ? [Peut être abstrait, d'une interface]
 - ▶ Le premier concret en descendant à partir de A ?
 - ▶ Celui de R ? [Peut-être hérité]
 - ▶ Le premier concret en montant de R ?

La plupart des langages objet choisissent : LE DERNIER

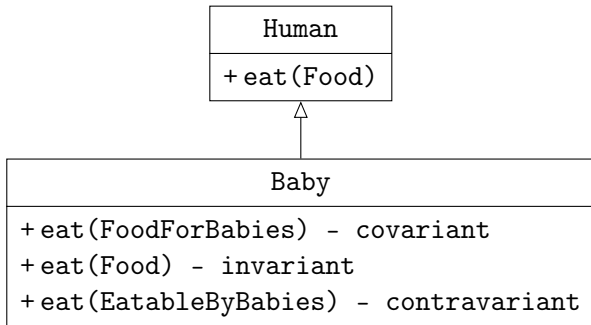
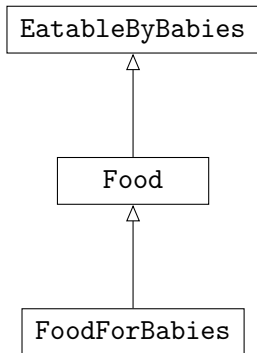
- ▶ signalé par eclipse : triangle vert 
- ▶ signalé (intention) en Java : `@Override`

- ▶ même nom, mais nombre ou type de paramètres différents (en Java)

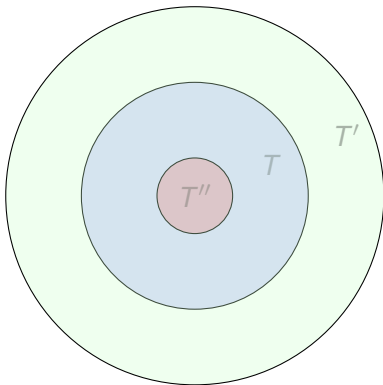
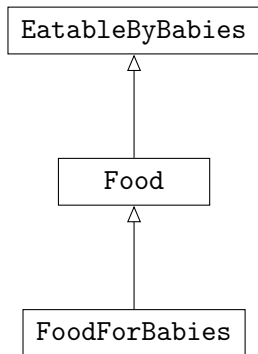
- ▶ (Invocation) Que se passe-t-il pour un appel `r.m(t')` :
 - ▶ avec t' de type réel $T' ::> T^1$ [SUBSTITUTION INCORRECTE, le compilateur rejette]
 - ▶ avec t' de type réel $T' <: T$ [SUBSTITUTION CORRECTE]
car t' sait faire tout ce qui est attendu par T
- ▶ (Définition) Est-ce une redéfinition avec `m(T' t)` ?
 - ▶ si $T' ::> T$ [SURCHARGE ; pas de liaison dynamique]
 - ▶ si $T' <:: T$ [SURCHARGE ; pas de liaison dynamique]
 - ▶ si $T' = T$ [oui ; liaison dynamique]

1. On note $::>$ ou $<::$ la relation stricte.

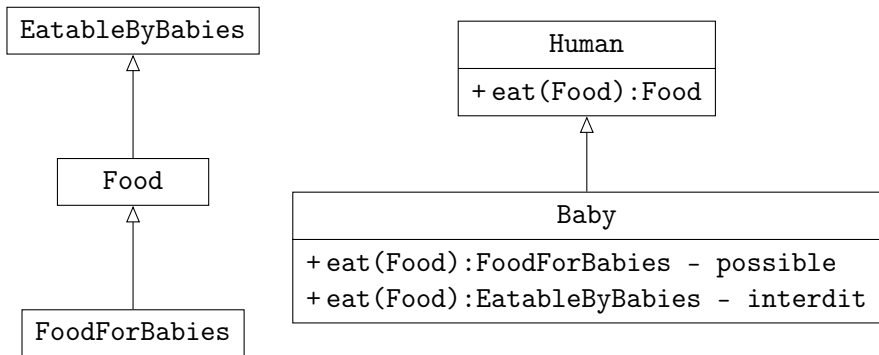
- Trois cas de redéfinition sont possibles avec $m(T', t)$:
- si $T' ::> T$ [Contravariance]
 - si $T' <:: T$ [Covariance]
 - si $T' = T$ [Invariance]



- Il faut des paramètres au moins plus généraux : $T' :=> T$
[Contravariance]
- S'ils sont plus restreints (riches/spécialisés) $T'' <: T$
[Covariance], on risque d'appeler avec des valeurs $v \notin T'' \wedge v \in T$



- ▶ Si $m(T \ t) : X$ est redéfini en $m(T \ t) : X'$
- ▶ Quelle relation est nécessaire entre X et X' ?
- ▶ Comme on peut avoir $X \ x = o. m(T \ t) ;$, il faut :
- ▶ $X' < X$ donc la COVARIANCE du type de retour.



- ▶ Redéfinition \neq Surcharge
- ▶ Redéfinition déclenche (est indissociable de) la liaison dynamique
- ▶ Pour une redéfinition on doit avoir : Type de retour covariant
- ▶ Pour une redéfinition on doit avoir : Paramètre contravariant
- ▶ Java choisit la version restrictive : invariance des types des paramètres
- ▶ Noter que la visibilité des propriétés peut grandir (`private` < `⌀` (default) < `protected` < `public`, elle est contravariante)
- ▶ Et autant ou moins d'exceptions (ou des sous-classes) pour la redéfinition

- ▶ Le compilateur (`javac`) vérifie les types et les règles de Substituabilité (il autorise ou non)
- ▶ À l'exécution (`java`), on applique la liaison dynamique ; si le système de type est sûr tout se passe bien.
 - ▶ (Java) on va voir un cas mal traité...

- ▶ Écrire une interface (c'est plus abstrait)
 - ▶ écrire les méthodes
- ▶ Écrire une classe [abstraite ou non]
 - ▶ définir les attributs (`private` ou `protected`)
 - ▶ écrire les constructeurs (qui initialisent les attributs, chaque niveau responsable de ses attributs)
 - ▶ écrire les méthodes (obligation de programmation, accesseurs, etc.)

- 1 Introduction
- 2 **Généricité, des besoins**
- 3 Généricité par l'exemple en Java

Que penser de $A[]$ et $B[]$ si $A <: B$?

- ▶ $A[] <: B[]$? (covariance de $[\cdot]$)
- ▶ $A[] :> B[]$? (contravariance de $[\cdot]$)
- ▶ incomparable ?

Le choix de Java... $A[] <: B[]$

C'est une erreur !

- ▶ Analyse statique correcte impossible
- ▶ Les erreurs sont détectées à l'exécution

```
// substitutability  
Number[] nums = new Number[5];  
nums[0] = new Integer(1); // Ok  
nums[1] = new Double(2.0); // Ok
```

```
Integer[] ints = new Integer[5];
```

```
// covariance  
nums = ints; // Ok
```

```
nums[0] = 1.23; // compile but fails  
// nums value is an array of Integer
```

Que penser de `ArrayList<A>` et `ArrayList` si $A <: B$?

- ▶ `ArrayList<A> <: ArrayList` ? (covariance de `<>`)
- ▶ `ArrayList<A> :> ArrayList` ? (contravariance de `<>`)
- ▶ incomparable ?

Le choix de Java...incomparable

C'est mieux !

- ▶ Il existe des structures de données qui sont naturellement homogènes pour toute une famille de types (voire tous)
- ▶ Liste de ..., ensemble de ..., tableau de ..., paire de ..., etc
- ▶ Il existe des algorithmes faisant un traitement homogène (invariant pour les types) sur ces données
 - ▶ Le tri d'une liste,
 - ▶ le minimum sur des nuplets numériques,
 - ▶ la recherche d'un élément dans une liste, ...
 - ▶ En général cela repose sur l'existence de fonctions (procédures) générales/génériques comme l'égalité, la comparaison, le hashcode etc
- ▶ Besoin de *généricité* : définir ces entités d'une façon concise en fonction d'un ou de plusieurs *paramètres de types*

```
public class Paire <T> { // T paramètre de type
    T premier ;
    T second;

    public Paire (T a, T b){ premier a; second = b; }

    public T getFirst(){ return premier; }

    public T getSecond(){ return second; }
}
```

T est un paramètre de type ; la classe l'introduit (<T>) ; le compilateur s'assure de la correction des types

Une utilisation simple de l'exemple précédent

```
Paire p = new Paire<Integer>(new Integer(1),  
                             new Integer(2));
```

// Si Pixel extends Point

```
Paire<Point> p = new Paire<Point>(new Point(... ),  
                                 new Pixel(...));
```

Le second cas est possible par substituabilité (voir 6)

On produit du code où on remplace T par le type effectif :
`Integer`, `String`, `Product`, etc.

Exemple d'utilisation avec `String` :

```
Paire<String> p = new Paire<String> ("abc", "xyz");
```

```
String x = p.getFirst(); // pas de cast
```

```
Double y = p.getSecond();  
// erreur de compilation (type mismatch)
```

- ▶ Le code est dupliqué
- ▶ Les paramètres de types sont substitués
- ▶ Le code résultant est compilé
- ▶ C'est le fonctionnement de base en C++
- ▶ Il y a autant de classes générées que de types effectifs différents

On utilise le type le plus général (`Object` par exemple) puis, à la compilation, on génère des `cast` (forçage de type) pour les types adéquats

```
public class Paire {  
    Object premier ;  
    Object second;  
    public Paire (Object a, Object b){  
        premier = a; second = b;  
    }  
    public Object getFirst(){  
        return premier;  
    }  
    public Object getSecond(){  
        return second;  
    }  
}
```

```
Paire p = new Paire ("abc", "xyz");  
String x = (String)p.getFirst();  
// le casting est obligatoire
```

```
Double y = (Double)p.getSecond();  
// runtime exception
```

On va avoir besoin de cast en général et en particulier si on veut un code sûr et « homogènement » bien typé

- ▶ Java propose une version homogène mais dont l'impémentation est basée sur l'héritage plutôt que les templates
- ▶ En Java il y a effaçage des informations de types avant la compilation et donc pas de duplication de code
- ▶ La compatibilité avec la machine virtuelle existante (< 1.5) est assurée
- ▶ Mais besoin de classes/interfaces comme paramètres effectifs pour les variables de type (en réalité des sous types d' `Object`, comme `Array`, `List`, `Number`, `Integer`, ...)
- ▶ Notations et exemples pour les classes
- ▶ Notations et exemples pour les méthodes
- ▶ Des règles à retenir

- 1 Introduction
- 2 Généricité, des besoins
- 3 Généricité par l'exemple en Java**

Les collections d'objets sont de bons supports à la compréhension de la généricité

```
// E is type parameter
public interface List<E> extends Collection<E>
// Some methods
// E as an argument type or return type
boolean add(E e) // (1)
E get(int index) // (1)
// an unknown type - the wildcard (joker)
boolean containsAll(Collection<?> c) // (2)
// a bounded wildcard (un joker borné) : extend
boolean addAll(Collection<? extends E> c) // (3)
// no reference to E is also possible!
void clear() // (4)
boolean contains(Object o) // (4)
```

Une méthode peut être générique, sans que la classe ne le soit

Par exemple la classe `java.util.Collections`

```
// from java.util.Collections
```

```
static <T> List<T>    emptyList()
```

```
static <K,V> Map<K,V> emptyMap()
```

```
// a bounded wildcard (un joker borné) : super
```

```
static <T> boolean    addAll(Collection<? super T> c,  
                           T... elements)
```

```
static <T extends Object & Comparable<? super T>> T  
        min(Collection<? extends T> coll)
```

? représente n'importe quelle classe

Permet de lire, mais pas d'écrire.

Si je veux faire des listes de n'importe quel type d'objet :

```
List<?> stuff = new ArrayList<>();  
// stuff.add("abc"); ne compile pas  
// stuff.add(new Object()); ne compile pas  
// stuff.add(3); ne compile pas  
int numElements = stuff.size(); // 0
```

// la solution

```
List<Object> stuff = new ArrayList<>();  
stuff.add("abc");  
stuff.add(new Object());  
stuff.add(3);  
int numElements = stuff.size(); // 3
```

```
import java.util.ArrayList;
import java.util.List;

public class GenericityTests {
    public static void main (String [] a) {
        List<?> stuffWildcard = new ArrayList<>();
        //stuff.add("abc");
        //stuff.add(new Object());
        //stuff.add(3);
        printList(stuffWildcard);

        List<Object> stuffObject = new ArrayList<>();
        stuffObject.add("abc");
        stuffObject.add(new Object());
        stuffObject.add(3);
        printList(stuffObject);
    }
}
```



```
List<Number> stuffNumber = new ArrayList<>();  
//stuffNumber.add("abc");  
//stuffNumber.add(new Object());  
stuffNumber.add(3);  
printList(stuffNumber);  
}  
  
private static void printList(List<?> list) {  
    System.out.println(list);  
}  
}
```

Si l'on veut sommer les éléments d'une liste de nombres ?

```
private static double sumList(List<?> list) {  
    double r = 0.0;  
    for (Number n: list) {  
        r += n.doubleValue(); //  
    }  
    return r;  
}
```

`doubleValue()` n'a de sens que sur des `Number`.

Ça ne marche pas

Car en fait ce sont des n'importe quoi !

Si l'on veut sommer les éléments d'une liste de nombres ?

```
private static double sumList(List<? extends Number> list) {  
    double r = 0.0;  
    for (Number n: list) {  
        r += n.doubleValue(); //  
    }  
    return r;  
}
```

Les objets de la liste sont des sous-types de `Number` et donc acceptent `doubleValue()`.

Le type peut être une **classe** ou une **interface**.

On utilise `super` ;

`List<? super Number>` indique que le type ne peut représenter que `List<Number>` ou `List<Object>`.

Le type peut être une `classe` ou une `interface`.

```
// Machin peu utile mais correct
```

```
private static double truc(List<? super Number> list) {  
    double r = 0.0;  
    for (Object n : list) {  
        r += n.hashCode();  
    }  
    return r;  
}
```

La borne supérieure permet d'extraire des valeurs et de les utiliser (on connaît l'interface disponible)

La borne inférieure permet de fournir des valeurs et de modifier la structure générique.

Cette règle pragmatique s'appelle PECS (*Producer Extends, Consumer Super*)

- ▶ Si un type paramétré fournit des objets ; c'est un producteur : utiliser `extends`
- ▶ Si on ajoute des objets dans un type paramétré ; c'est un consommateur : utiliser `super`
- ▶ Si on veut faire les deux : utiliser le type explicitement (seule intersection entre les deux ensembles)

Les collections en Java sont invariantes, sauf si :

- ▶ On utilise un joker `extends` qui permet la covariance
- ▶ On utilise un joker `super` qui permet la contra-variance

Un type paramétré peut avoir plusieurs bornes ; elles sont séparées par un `&`.

Il peut y avoir autant de bornes `interface` que l'on veut.

Il ne peut y avoir qu'au plus une borne `class` et ce doit alors être la première.

```
T extends Runnable & AutoCloseable
```

- ▶ Les types : garde-fou puissant, mais compliqué avec l'héritage
- ▶ Simplification : faire de la redéfinition invariante
- ▶ Simplification : éviter la surcharge (sauf des constructeurs*²)
- ▶ Programmer générique...c'est délicat

Enfin, tous les langages objet n'ont pas les mêmes règles !

- ▶ Type et fonction générique
- ▶ Subsumption, polymorphisme, substituabilité
- ▶ Variance, covariance, contravariance
- ▶ Variable de type, type paramétré
- ▶ Instanciation d'un paramètre générique
- ▶ Joker

Organisation

- ▶ Documenter
- ▶ Vérifier, valider
- ▶ Faire relire

Conception

- ▶ Chercher la modularité
- ▶ Réutiliser
- ▶ Contractualiser
- ▶ Couplage et cohésion

Programmation

- ▶ Séparer la spécification de sa (ou ses) réalisation(s)
- ▶ Déclarer abstrait, instancier concret