Programmation Objet
Java–Classes Génériques

Michail Lampis
michail.lampis@dauphine.fr

Typage Statique

- À quoi sert la déclaration des types ?
  - On utilise le système des types d'un langage « haut niveau » pour nous aider à trouver des erreurs.
  - Ex :

    a[2] = 5 ; //a est bien un tableau ?

    a.f(3) ;// la méthode f existe ? Elle prend un int ?

    x = y+z ;//peut-on additionner y et z?
- Typage Statique (Static Typing)
  - Le principe qui exige que le compilateur fasse le maximum possible des vérifications, en utilisant les types.
  - Désavantages :
    - Plus de boulot pour le programmeur
    - Beaucoup d'erreurs introuvable pendant la compilation
  - Avantages :
    - Programmes plus lisibles
    - Programmes plus correctes !

Typage Dynamique

- Typage Dynamique : on ne vérifie si les données ont le bon type que pendant l'exécution du programme
  - Exemple : Javascript, PHP, Python, ...
  - En javascript on peut écrire

    var a ;

    a = 2*a ;

    a[0] = a+5 +a.f();
  - Ce programme est accepté par l'interpréteur, mais ne marche pas !
- Si on avait de typage dynamique, cela nous aurait aidé trouver l'erreur avant l'exécution du programme...

Java - Typage Statique ?

- Java utilise un système de typage (quasi-)statique

  a.f(3) ;//le compilateur regarde le type déclaré de a

- ...avec une grande exception !

  class A { void f() {…} }

  class B extends A { void f() {…} }

  ..

  A a = … ;

  a.f() ; //Quelle méthode est appellée ?

- Pour l'appel d'une méthode on utilise le type réel d'un objet
  - → Vérification de type forcément pendant l'exécution, information non-connue pendant la compilation !

Typage Statique - Exemple

T [] tab = new T [10] ; //où T est un type

- Garantie du système des types
  - tab[2] a le même type que tab[3]
  - Impossible de mettre un élément d'un type autre que T dans tab[0]
    - Sauf si → Sous-type !
  - Expressions suivantes bien définies, si le type de T est approprié, sinon erreur.

    tab[0].f() ;

    tab[0][1] =5;

Typage Statique - Exemple

```
class StackInt {
    void push(Integer i){..}
    Integer pop() {…}
}
…
StackInt s = new StackInt() ;
for(int i=0 ; i<5 ; i++) s.push(i) ;
for(int i=0 ; i<5 ; i++) S.o.p(s.pop(i)*3) ;
```

- Programme correcte
  - Avantage : On ne peut mettre que des Integer dans le Stack

    → **s.push(''Hello'') ;** //→ Erreur de compilation
  - Désavantage : Quelle est la différence avec un StackDouble, StackString, ..?

    → Duplication de code !

Typage Statique - Exemple

```
class Stack {
    void push(Object i){..}
    Object pop() {…}
}
…
Stack s = new Stack() ;
for(int i=0 ; i<5 ; i++) s.push(i) ;
for(int i=0 ; i<5 ; i++) S.o.p(((Integer)s.pop(i))*3) ;
```

- Programme correcte
  - **Dés**Avantage : On peut mettre n'importe quoi dans le Stack, mélanger des types

    → **s.push(''Hello'') ;** //Pas de problème
  - Avantage : Il suffit d'écrire une seule version de la classe pour tout les types

    → **Pas de** duplication de code !

Généricité – Motivation

- Problème avec solution 1
  - Trop rigide → Beaucoup de complication de code
- Problème avec solution 2
  - N’utilise pas les avantages du système de typage
  - Aucune vérification des types des données pendant la compilation
  - Il faut faire de **casting** pour utiliser les données de la classe
    - Toujours une mauvaise indication !
- On cherche une solution qui est à la fois générique (couvre plusieurs types) est solide (ne se passe pas du système des types).

Généricité - Principe

- Classe générique (Generic class) : une classe dont la définition est paramétrée avec un ou plusieurs types « variables ».

```
class Stack <T> {
    void push(T o){..}
    T pop() {..}
}
```

- Explication : T est une variable **de type**. L'idée est qu'on peut définir plusieurs versions de la même classe où T peut être remplacé par un autre type non-primitif.
- Dans la classe :
  - On utilise T comme si c'était un vrai type (avec quelques restrictions)
- En dehors de la classe :
  - Pour instancier un objet, il faut préciser le type T, p.ex. écrire **Stack<Integer>**

Généricité - Exemple

- Une liste chaînée générique :

```
class myLinkedList <T> {
    private T data ;
    private myLinkedList <T> next ;
    public myLinkedList(T d) { data = d ; }
    public myLinkedList(T d, myLinkedList<T> n){
        data = d ; next = n ; }
    //getter/setter ? Exemple
    public setData(T d) { data = d ; }
}
```

Généricité - Exemple

- Pour utiliser une classe générique il faut l'instancier

  myLinkedList<Integer> t = new myLinkedList<Integer>(5) ;

  - Détail : le compilateur nous permet de na pas préciser T quand il peut être inféré :

  myLinkedList<Integer> t = new myLinkedList<>(5) ;//OK

- Une fois instancié, on peut supposer que l'objet **t** a toutes les méthodes et variables de la classe, mais avec **T** remplacé par **Integer.**

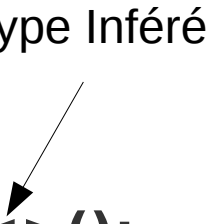
  - **setData(Integer d) ;**
  - **setNext(myLinkedList<Integer> n) ;**
  - **...**

Pile Générique - Exemple

```
class myList<T> {
        T data;
        myList<T> next;
}
class Stack<T> {
        myList<T> head;
    ...
```

Pile Générique - Exemple

```
class Stack<T> {
        myList<T> head;
        void push(T d){
                myList<T> t = new myList<>();
                t.data = d;
                t.next = head;
                head = t;
        }
```



Pile Générique - Exemple

```
class Stack<T> {
        myList<T> head;
        T pop(){
                if(head == null) return null;
                //Exception?
                T t = head.data;
                head = head.next;
                return t;
        }
```

Pile Générique – Intérêt

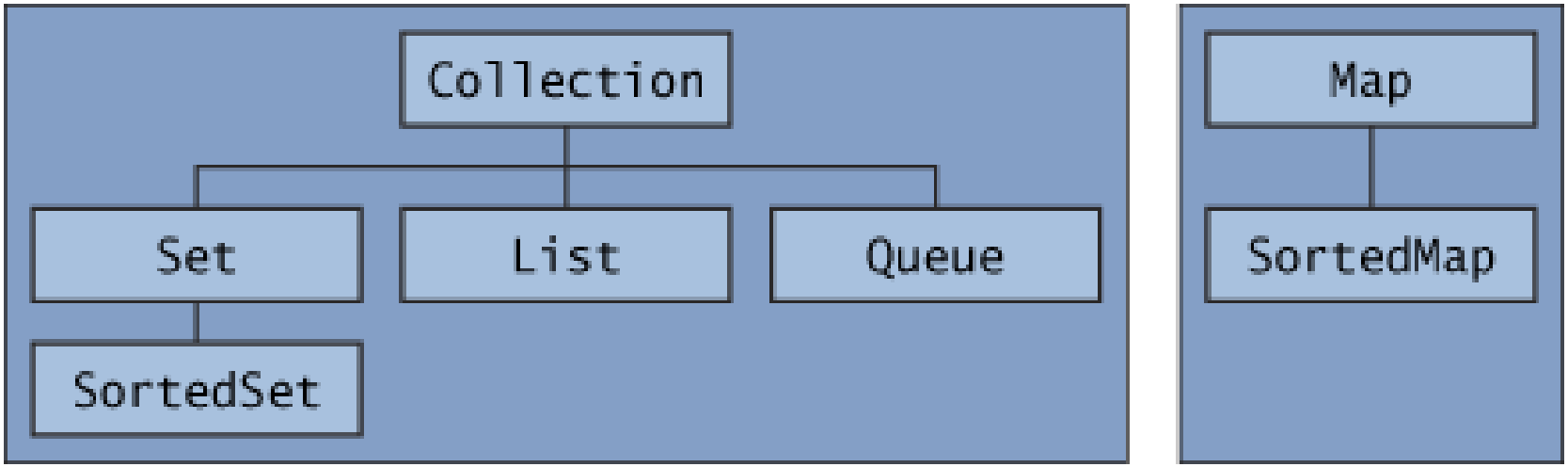
- Avec la classe générique on a deux avantages
  - La classe traite tous les types non-primitifs
  - L'intégrité des données est garantie par le compilateur
    - → Pas besoin de transtypage « à la main »

```
public static void main(String [] args){
        Stack<Integer> s1 = new Stack<>();
        Stack<String> s2 = new Stack<>();
        for(int i=0;i<5;i++) s1.push(i);
        for(int i=0;i<5;i++) s2.push(i+",");
        for(int i=0;i<2;i++) System.out.println(2*s1.pop());
        System.out.println(s1);
        System.out.println(s2);
}
```

OK pour Integer !
(Auto-Unboxing)

Collections

- Dans la bibliothèque Java on retrouve
  - interfaces collection
  - implémentations
  - Algorithmes



Collections

Il s’agit d’interfaces génériques

- Collection<E>: base de la hierarchy
  - Set<E>: ensemble d’éléments de type E (sans duplication)
    - SortedSet<E>: ensembles ordonnés
  - List<E>: suite d’éléments de type E (avec duplication)
  - Queue<E>: file d’elements de type E
- Map<K,V>: association clés-valeurs (clés de type K, valeurs de type V)
  - SortedMap<K,V> avec un ordre sur les clefs

Listes

- L’interface List<E> est implémentée par deux classes :
  - ArrayList<E>
  - LinkedList<E>
- Exemple :

```
List<String> l = new LinkedList<String>();
l.add(“abc”); l.add(“bcd”);
String s = l.get(0);
```

Collections

- Les classes de la bibliothèque de Java qui implémentent l'interface Collection sont extrêmement utiles.
  - Structures des données avec de fonctionnalité comme
    - File
    - Pile
    - Tableau Dynamique (on peut ajouter/supprimer des éléments au fur et à mesure)
    - Tableau associative
    - Ensemble
    - ...
- Dans les version modernes de Java toutes les classes de la bibliothèque sont génériques
  - Cependant, on a aussi des versions non-génériques...

L’interface Iterable

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

- Un objet Iterable est un objet dont on peut parcourir les éléments “contenus”.
  - Soit avec la construction “for each” :

```
for ( Object o : monIterable ) System.out.println ( o ) ;
```

- Soit en utilisant explicitement un itérateur :

```
Iterator <String> it = monIterable.iterator() ;
while(it.hasNext () ) {
    String s = it . next () ;
    System.out.println (s) ;
}
```

Listes (Non-)Génériques

- Exemple :

  List l = new LinkedList();

  l.add(“abc”); l.add(“bcd”);

  String s = (String)l.get(0);

- Version Java < 5
  - Retenue pour des raisons de compatibilité
- **À éviter !!**
  - Ne garantit pas que tous les éléments de la liste ont le même type
    - → get() retourne Object
  - Nécessite un type casting pour utiliser l’élément retourné.

Méthodes Génériques

- On peut également définir des méthodes génériques (i.e. avec leur propres paramètres de type)
  - E.g. une méthode qui peut recevoir un tableau de type arbitraire et renvoie un tableau du même type

  static <T> T pick(T[] tab) {…}

  - Déclaration du type variable après les modificateurs et avant le type de retour
  - Elles peuvent appartenir à des classes normales, ainsi qu'a des classes génériques

Méthodes Génériques

```
public class Utilite {
public static <T> T[] permuter (T[] tab){...}
...
}
```

- On peut explicitement préciser le type:

```
String[] tab ={“Bonjour”, “Tout”, “le monde”};
String[] s = Utilite.<String>permuter (tab);
```

- ..ou pas

```
String[] s = Utilite.permuter (tab);
```

Effacement

- Comment ça marche la généricité ?

  Class <T> myList { T d ; .. }

- En C++

  template <class T>

  T f(T x) { .. }

  ..

  f(2) ; f(2.1) ; f(‘’Hello’’) ;

- → Compile trois versions de f separés
  - f(int), f(double), f(String)

Effacement

- Comment ça marche la généricité ?

  Class <T> myList { T d ; .. }

- Le compilateur de Java ne produit qu'une seule version de la classe myList
  - T est remplacé par Object
    - → On ne peut utiliser que des types non-primitifs pour T
  - Pendant l'exécution le programme ne retient aucune information pour le type T
    - T sert seulement pendant la compilation pour vérifier l'intégrité des types des données utilisées.

Effacement

- Puisque les informations concernant T sont effacées, pour garantir l'intégrité des données pendant l'exécution le compilateur ajoute des casts.

  public class Suite <E> { E premier ; E getFirst() ; … }

- → En réalité le compilateur, **après vérification des types**, quand on utilise cette classe produit la classe

  public class Suite { Object premier ;…}

- → Où on a utilisé des instances de cette classe (avec un type E spécifique), on ajoute des casts

  Suite <Integer> s = new Suite<>();..;int x=s.getFirst() ;

- →

  Suite s = new Suite();..;int x=(Integer)s.getFirst() ;

Effacement → Conséquences

- Une classe générique doit être invoquée avec un type non-primitif (sous-classe de Object)

  ArrayList<int> //ne marche pas !!

- Deux objets avec un type différent pendant la compilation peuvent avoir le même type pendant l'exécution

  C<A> a = new C<A>() ; C<B> b = new C<B>() ;

  a.getClass() == b.getClass() // → True !

  a.instanceOf C, b.instanceOf C sont aussi True

  - Pour la JVM il existe une seule version de la classe générique C<T>, appelée simplement C.

Effacement → Conséquences

- Dans une classe générique on ne peut pas instancier un objet du type variable !

```
Class MaClasse<E> {
    public E g() {.. } //OK
    public E f() {
        E v ; //OK
        v = new E() ; // !!!
```

- Instanciation pas permise !
  - **new E() ;** serait traduit en **new Object()…**
- Par contre **v = g() ;** est OK

Effacement → Conséquences

- Dans une classe générique on ne peut pas créer un tableau du type variable !
  - Même si créer un tableau n'appelle pas new…

```
Class MaClasse<E> {
    public E f() {
        E [] v  = new E[10] ; // !!!!
```

- Raison : ça équivaut a un tableau de Object
  - On ne peut pas garantir que tous les éléments du tableau auront le même type…
- Solution :
  - Utiliser ArrayList/LinkedList dans une classe générique

Bornes Supérieures

- Une variable de type T doit représenter un type sous-classe de Object
  - Trop générique…
- On peut imposer des contraintes additionnelles.

```
class A<T extends Iterable> {
        T d;
   public void f(){
        for(Object o:d)
           System.out.println(o); }
}
```

Bornes Supérieures

- Une variable de type T doit représenter un type sous-classe de Object
  - Trop générique…
- On peut imposer des contraintes additionnelles.
- On peut imposer plusieurs bornes supérieures à la fois avec l'opérateur &
  - Les bornes supérieures peuvent être des classes ou des interfaces

  class A<T extends B & C>

  - → T doit être une sous-classe (extends ou implements) de B et C

Bornes Supérieures

- L'interface Comparable

```
interface Comparable <T> {
    int compareTo(T t)
}
```

- Une classe qui implémente cette interface promet que ces objets peuvent être comparés avec des objets de type T
  - La méthode compareTo retourne un int négatif, zero, ou positif, si l'objet actuel est inférieur, égal, ou supérieur à l'objet t.

```
class Rational implements Comparable<Rational> {
    int n,d;
    public int compareTo(Rational r){
        return n*r.d - d*r.n;
    }
}
```

Bornes Supérieures

```
class Rational implements Comparable<Rational> {
        int n,d;
        public int compareTo(Rational r){
                return n*r.d - d*r.n;
        }
}
static <T extends Comparable<T>> T [] sort(T [] tab){
                return .....;
}
Rational [] t = new Rational [10] ;
sort(t) ; //OK
```

Covariance

- Rappel B est sous-type de A (B est cas spéciale de A) si
  - B extends A
  - B implements A
- →
  - A a = new B() ; //OK, mettre cas spéciale dans variable générale
  - B b = new A() ; // !!!
  - B b = (B) new A() ;// !!! Accepté par le compilateur, mais exception
- Covariance : quelle est la relation entre les types A [] et B [] ?
  - B est sous-type de A →
  - B [] est sous-type de A []

Covariance

- Exemple :

  void f(A [] tab) {…}

  class A {..}

  class B extends A {..}

  B [] tb = new B [10] ; …

  f(B) ;

- OK ! La méthode f veut recevoir un tableau d'objets de type A. Chaque objet du tableau qu'on passe comme paramètre a comme type B. Mais B est un sous-type de A
  - On peut utiliser un B où on peut utiliser un A
- Alors, le compilateur accepte...

Covariance – Pièges

```
class A { }
class B extends A { void f() {} };
class Test {
        public static void main(String [] args){
                B [] tabB = new B[10];
                A [] tabA;
                tabA = tabB; //OK?
                tabA[0] = new A(); //OK?
                tabB[0].f(); //OK?
        }
}
```

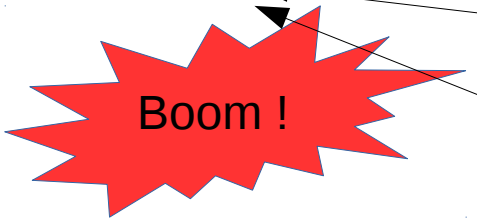
Covariance – Pièges

```
class A { }
class B extends A { void f() {} };
class Test {
        public static void main(String [] args){
                B [] tabB = new B[10];
                A [] tabA;
                tabA = tabB; //OK?
                tabA[0] = new A(); //OK?
                tabB[0].f(); //OK?
        }
}
```

Bon. B[] sous-classe de A[]

Bon. tabA[0] est de type A.

Bon. tabB[0] est de type B, donc contient une f().



ArrayStoreException !

Covariance des Tableaux

- Est-ce une bonne idée
  - Pour :
    - La covariance est très intuitive (si B cas spéciale de A, alors B[] cas spéciale de A[])
    - Simplifie le code
  - Contre :
    - Nécessite de vérifications pendant l'exécution
    - → Le programme peut échouer et lancer une exception si les données n'ont pas le bon type

Covariance et Généricité

- Rappel : Motivation de la Généricité est de vérifier le maximum possible pendant la compilation
  - Sinon, on aurait utilisé Object à la place de T
- Covariance → Situations impossibles à vérifier avant exécution
- Covariance + Généricité →
  - Impossible à vérifier avant exécution (voir tableaux)
  - Impossible à vérifier pendant l'exécution (effacement de types)
  - !?
- Résultat : les types génériques n'ont pas de covariance
  - Même si B est sous-classe de A, List<B> **n'est pas sous-classe** de List<A>.

Covariance et Généricité

```
public class Person {...}
public class Student extends Person {...}
public static void changeAddress (Person p, String ad) {...}
public static void invite (ArrayList<Person> l) {...}
...
Student s1 = new Student(...); Student s2 = new Student(...);
ArrayList<Student> l = new ArrayList<Student>();
l.add(s1); l.add(s2);
changeAddress (s1, “5 rue Monge”); // OK
invite (l); //ERREUR : ArrayList<Student> n’est pas un
            //sous-type de ArrayList<Person>
```

Wildcards

- Lors de l'utilisation d'un type générique, on peut décider, au lieu de le concrétiser, de remplacer un paramètre par un "?" (symbolisant un joker, un wildcard).

  static double somme(List<? extends Number> liste){ ... }

- Équivalent à

  static <T > double somme (List < T extends Number > liste) { ... }

- Différence : quand on a plusieurs ? ils représentent des types différents
- Différence : on peut déclarer aussi des bornes inférieures

  static void remplir(List<? super Number> liste){ ... }

- Annonce que la fonction remplir va traiter une liste dont chaque élément a un type T, qui est plus générale que Number (i.e. Number est une sous-classe de T)

Wildcards et Covariance

- Intérêt :
  - Le problème avec les tableaux et la covariance est qu'on peut à la fois lire et écrire au tableau, en le regardant comme un tableau de A ou B.
- Les Wildcards nous donnent la possibilité de définir soit une borne inférieure, soit une borne supérieure.
- Cela nous permet d'implémenter de manière sûre soit la lecture, soit l'écriture sur une classe générique, **avec covariance.**

Covariance et Généricité

```
public class Person {...}
public class Student extends Person {...}
public static void changeAddress (Person p, String ad) {...}
public static void invite (ArrayList<Person> l) {...}
...
Student s1 = new Student(...); Student s2 = new Student(...);
ArrayList<Student> l = new ArrayList<Student>();
l.add(s1); l.add(s2);
changeAddress (s1, “5 rue Monge”); // OK
invite (l); //ERREUR : ArrayList<Student> n’est pas un
            //sous-type de ArrayList<Person>
```

Covariance et Généricité

```
public class Person {...}
public class Student extends Person {...}
public static void changeAddress (Person p, String ad) {...}
public static void invite (ArrayList< ? extends Person> l) {...}
...
Student s1 = new Student(...); Student s2 = new Student(...);
ArrayList<Student> l = new ArrayList<Student>();
l.add(s1); l.add(s2);
changeAddress (s1, “5 rue Monge”); // OK
invite (l); //OK !
```

Conséquences

- Considérer la méthode **invite**

  public static void invite (ArrayList< ? extends Person> l){

- Le type de l est ArrayList<T>, où T est une sous-classe de Person

  Person p = l.get(0) ;

- OK ! Le type de l.get(0) est T, une sous-classe de Person
- Par contre…

  l.add(p) ;// !!!

- Ne marche pas, parce que la méthode add prend comme paramètre un objet de type T, et on tente de passer un objet de type Person (super-classe de T).
- → On peut lire la liste, mais pas écrire, puisqu'on ne connaît pas un type sûr qui peut être ajouté à l.
- → Ça nous convient pour cette exemple (on appelle avec ArrayList<Student> comme paramètre. C'est souhaitable qu'on ne puisse pas mettre un Person dans cette liste.

Conséquences

- Par contre, avec super on peut écrire mais pas lire !
- Considérer la méthode **invite**. Si on avait mis

  public static void invite (ArrayList< ? super Person> l){
- Le type de l est ArrayList<T>, où T est une **super**-classe de Person

  Person p = l.get(0) ;
- Ne marche pas ! Person est un cas spéciale de T, le type de l.get(0) ;
- Par contre…

  l.add(p) ;// OK !
- Est accepté, puisque p est de type Person, add prend comme paramètre T, qui est super-classe de Person.
- → Il faut décider si la méthode a besoin d'écrire où lire. Si on fait l'un des deux, on a de la covariance sûre !
- Covariance sûre → L'intégrité de la collection est garantie.

Conséquences

class Rational implements Comparable<Rational> {..}

class OrderedSet<E extends Comparable<E>>{..}

- Peut-on dire **OrderedSet<Rational>** ?

Conséquences

class Point implements Comparable<Point> {..}

class OrderedSet<E extends Comparable<E>>{..}

- Peut-on dire **OrderedSet<Point>** ?
  - Oui !
- Cependant on a un problème potentiel avec l'héritage :
  - E extends Comparable<E> est trop fort
- **class ColoredPoint extends Point {..}**
  - On ne redéfinit pas compareTo (pas nécessaire)
- Peut-on dire **OrderedSet<ColoredPoint>** ?
  - Non ! ColoredPoint implements Comparable<Point> (méthode héritée), mais n'implémente pas Comparable<ColoredPoint>
- ?!

Conséquences

class Point implements Comparable<Point> {..}

class OrderedSet<E extends Comparable< ? super E>>{..}

- Peut-on dire **OrderedSet<ColoredPoint>** ?
  - Oui ! ColoredPoint implements Comparable<Point> (méthode héritée)
  - Cela (Point) correspond à une super-classe de ColoredPoint
- La bibliothèque Java fait une utilisation massive de ces formes de généricité...