GLG203, Cnam/Paris
Java et la Généricité

Cnam Paris
jean-michel Douin
version du 4 Octobre 2021

Notes de cours java :
Généricité en Java extrait en partie du cours de NFP121, http://jfod.cnam.fr/NFP121/

Sommaire

- **Généricité**
  - **Pourquoi, éléments de syntaxe**
- **Collection / Conteneur**
  - **Homogénéité**
- **Comment ?**
  - **À la compilation**
- **Contraintes pour son utilisation**
- **Conclusion**
  - Annexe 1 : Quiz
  - Annexe 2 : Les exceptions et la généricité
  - Annexe 3 : *Design pattern* et généricité
    - Patron Fabrique, Observateur MVC (utile/inutile)

Principale bibliographie

- Tutoriel sur la généricité en Java, indispensable
  - http://www.eecs.qmul.ac.uk/~mmh/APD/bloch/generics.pdf
- La base de ces notes de cours
  - https://www.csie.ntu.edu.tw/~htlin/course/oop10spring/doc/generics-tutorial.pdf
- Journal of Object Technology http://www.jot.fm
  - http://www.jot.fm/issues/issue_2004_07/column2.pdf
- Variance en POO
  - http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/papers/pdf/variance.TOPLAS.pdf

Généricité : Introduction

- **Motivation**

- **Inférence de type**
  - **Principe d *'erasure*, à la compilation**
  - **pas de duplication de code** (différent des templates de C++)

- **Aucune incidence sur la JVM**
  - **les *cast* restent en interne mais deviennent sûrs (sans levée d'exceptions)**
  - **mais ce n'est pas sans incidence…**

Généricité : Motivation

- **Tout est *Object* ==> non sûr**
  - **Une collection d'*object* peut être une collection hétérogène**

- **Exemple:**
  - **List l = new ArrayList();**
  - **l.add(new Integer(0));**
  - **l.add(new Boolean(true));**
  - **String s = (String) l.get(0);**
    - **// compilation : cast obligatoire, l.get(0)** retourne un « Object »

      // exécution : ClassCast Exception est levée !
  - **idem lors de l 'usage d 'un itérateur, (java.util.Iterator)**
  - **la méthode next() retourne un Object**
  - **String s = (String)l.iterator().next(); // exécution : ClassCast Exception**

<G>énéricité / <G>énéralités

- **Le type devient un < paramètre de la classe >,**
- **Le compilateur vérifie alors l'absence d'ambiguïtés,**
- **C'est une analyse statique (et uniquement),**
- **Cela engendre une inférence de types à la compilation.**

```
public class UneCollection<T>{   // une toute petite collection…
  private T elt;                  // un seul élément

  public void setElt(T elt){ this.elt = elt;}
  public T getElt(){ return elt;}
}


UneCollection<Integer> c1 = new UneCollection<Integer>();
Integer i = c1.getElt();

UneCollection<String> c2 = new UneCollection<String>();
String s = c2.getElt();
```

Un premier exemple Couple<T>

```
public class Couple<T>{                    // une autre collection:
                                           // un couple homogène
  private T a;
  private T b;

  public Couple(T a, T b){
    this.a = a;
    this.b = b;
  }

  public void échanger(){
    T local = a;
    a = b;
    b = local;
  }

  public String toString(){return "(" + a + ", " + b + ")";}
  public T getA(){ return this.a;} // accesseurs
  public T getB(){ return this.b;}
  public void setA(T a){ this.a=a;} // mutateurs
  public void setB(T b){ this.b=b;}
}
```

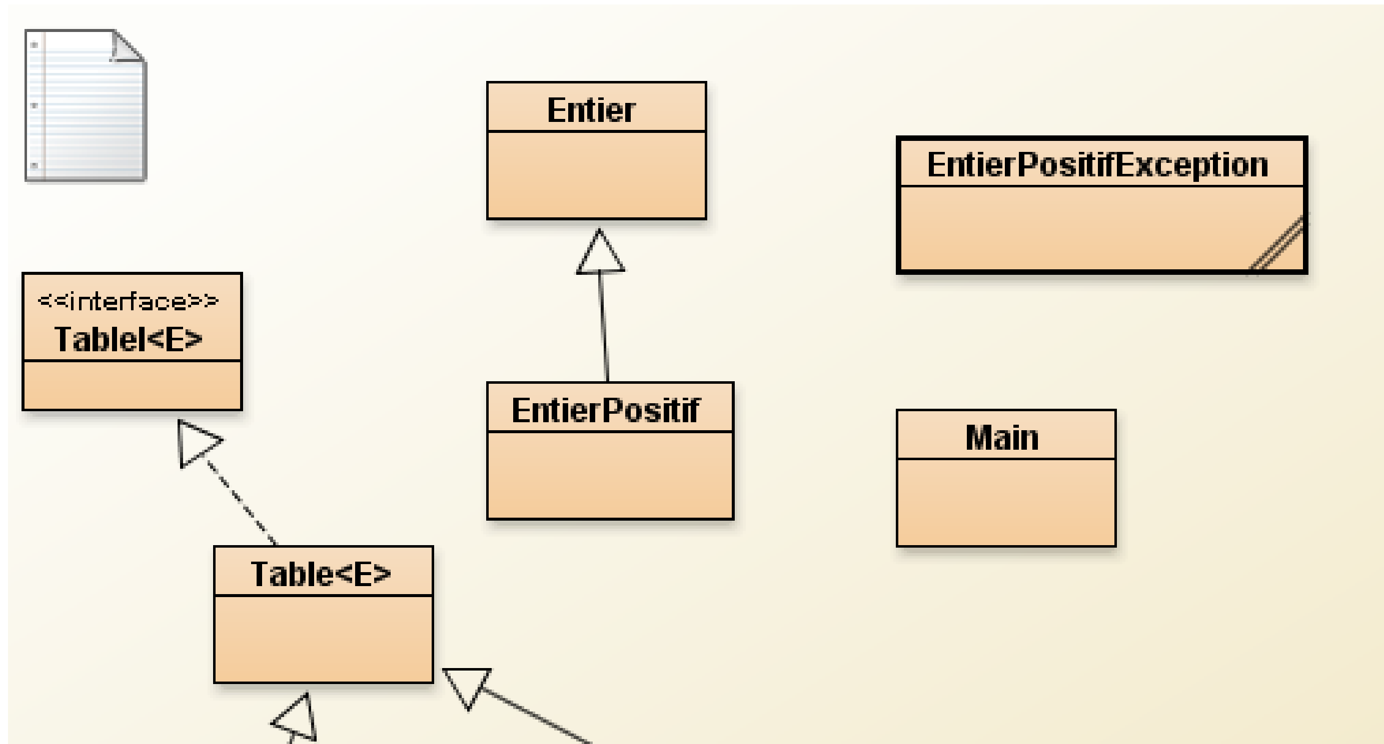
Un premier exemple : utilisation

```
Couple<String> c1 = new Couple<String>("fou", "reine");


Couple<Integer> c = new Couple<Integer>(new Integer(5),new Integer(6));
c.échanger();
Integer i = c.getA();
assert i==6;


Couple<Object> c2 = new Couple<Object>(new Integer(5), new String("cheval"));
Object o = c2.getA(); // Integer extends « Object »
```

divers

```
Couple<Integer> c = new Couple<Integer>(5, 7);  // auto boxing du 1.5...
Couple<int> c = new Couple<int>(5, 7);   //  erreur de syntaxe :
                                         // types primitifs non autorisés
```

Un autre exemple d'utilisation…




```
public interface TableI<E> extends Iterable<E>{

  public void ajouter(E e);
  public E lire(int index);
}
```

Utilisation suite…

```
TableI<Entier> t = new Table<Entier>();
t.ajouter(new Entier(3));
t.ajouter(new EntierPositif(3));
Entier e0 = t.lire(0);
Entier e1 = t.lire(1); // erreur si EntierPositif e1 ?

TableI<Object> to = new Table<Object>();
to.ajouter(new Entier(3));

TableI<Integer> ti = new Table<Integer>();
ti.ajouter(new Integer(3));
```

Démonstration

- http://jfod.cnam.fr/GLG203/exemples_cours_genericite.jar

Constats pour engager une réflexion…

- **Nous avons :**
  - **Object o = …;**
  - **Integer i = …;**
  - **o = i ; // correct !**

  - **Object[] to= …;**
  - **Integer[] ti= …;**
  - **to = ti; // correct !**

- **Avons-nous ?:**
  - **Couple<Object> co= …;**
  - **Couple<Integer> ci= …;**
  - **co = ci; // ???**

  - **Tablel<Object> to= …;**
  - **Tablel<Integer> ti= …;**
  - **to = ti; // ???**

Constats, Questions

- Nous avons :
  - Object o = …;
  - Integer i = …;
  - o = i ; // correct !

  - Object[] to= …;
  - Integer[] ti= …;
  - to = ti; // correct !

- Avons nous ?
  - Couple<Object> co= …;
  - Couple<Integer> ci= …;
  - co = ci; // incorrect ! ???????

Couple<Object> = Couple<Integer> ???

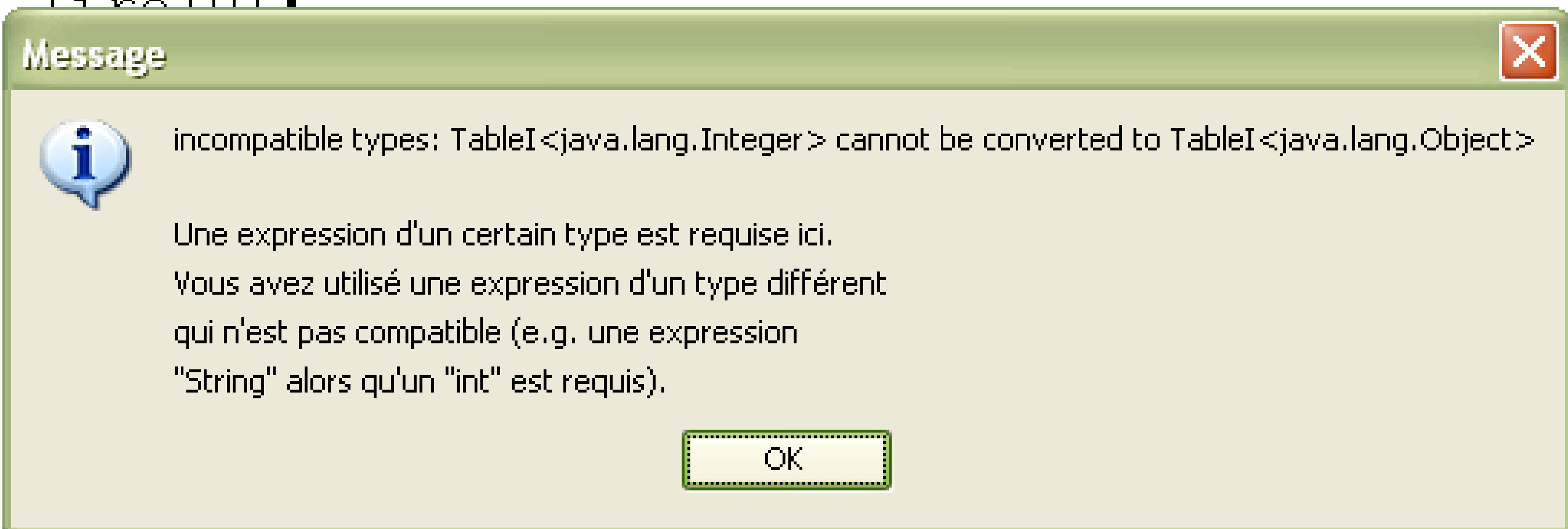
```
Couple<Object> c = new Couple<Object>(…)

Couple<Integer> c1 = new Couple<Integer>(…);


c = c1;  // Couple<Object> affecté par un Couple<Integer>

Erreur de compilation … démonstration diapositive suivante
```

```
incompatible types - found
 Couple<java.lang.Integer> but
 expected Couple<java.lang.Object>
```

- `c = c1;` // est donc incorrect ! ?

Table<Object> = Table<Integer> ?

```
TableI<Object> to = new Table<Object>(…)

TableI<Integer> t1 = new Table<Integer>(…);

to = t1;
// Table<Object> affecté par un Table<Integer>
```



Par l'absurde, si cela était possible

```
Couple<Object> c = new Couple<Object>(…)
Couple<Integer> c1 = new Couple<Integer>(…);
```

si c = c1; était possible

nous pourrions écrire

- **c.setA(new Object());**
- **Integer i = c1.getA();** ***/// !!!! Integer = Object !!!***

donc

Couple<Object> ne peut recevoir qu'un Couple<Object>

Collection<Object> ne peut recevoir qu'une Collection<Object>

```
public static void afficher(java.util.Collection<Object> c){
    for( Object o : c)
      System.out.println(o);
  }
```

ne peut …. qu'afficher qu 'une java.util.Collection<Object> et rien d'autre

```
public static void afficher(java.util.Collection<Couple<Object>> c){
    for(Couple<Object> o : c)
      System.out.println(o);
  }
```

ne peut afficher qu'une Collection de couple d'Object et rien d'autre,

Donc …

```
public static void afficher(java.util.Collection<Boolean> c){
   for( Boolean o : c)
     System.out.println(o);
  }
```

ne peut …. qu'afficher qu 'une java.util.Collection<Boolean> et rien d'autre

```
public static void afficher(java.util.Collection<Couple<Boolean>> c){
   for(Couple<Boolean> o : c)
     System.out.println(o);
  }
```

- **ne peut afficher qu'une Collection de couple de booléen et rien d'autre !!!!**
- **Et Tablel et les List ?**

Couple et List<Object> = List<String> idem

```
List<Object> t = …;
List<String> s = …;        // String extends Object


t=s;  // si cette affectation était possible


t.set(0,new Object());

String s = s.get(0);   // nous aurions une erreur ici
```

Contraignant mais légitime n'est-ce pas ?

Et pourtant les tableaux

```
String[] s = new String[1];
Object[] t = s; // ok
```

Nous avons la **covariance** des tableaux en java

Mais nous avons bien une erreur ici à l'exécution si

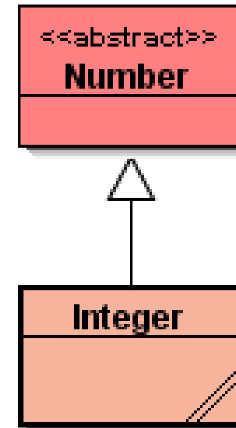
```
t[0] = new Boolean(true);  // compilation réussie
  // A l'exécution : exception
```

cette affectation déclenche une exception ArrayStoreException
dès l'affectation t=s, t est contraint à ne recevoir que des String …

http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

Déjà vu ? : Covariance NFP121, cours 3-1

```
public class A{
  public Number f(){ return 0; }
}

public class B extends A{
  @Override
  public Integer f(){ return 1; }
}
```




```
A a = new B();
Number n = a.f();     //Integer extends Number
```

- // n peut recevoir n'importe quelle instance d'une classe fille de Number,
- // et quelque soit le type constaté de a

- **Covariance ici**
  Le type de retour de la méthode redéfinie peut être un sous-type de celui de la classe mère

Impasse ? Non, <?> à la rescousse

Collection<Object> ne peut recevoir qu'une Collection<Object>

Alors joker '?'

```
public static void afficher(java.util.List<?> liste){
    for( Object o : liste)
      System.out.println(o);
}

afficher(new java.util.ArrayList<Object>());
afficher(new java.util.ArrayList<Integer>());
afficher(new Couple<String>());

List<?> l = new ArrayList<Integer>();
```

Mais

<?> la contre-partie <?>

? **Représente le type inconnu et si le type est inconnu (analyse statique) alors**

```
public static void ajouter(java.util.Collection<?> c, Object o){
    c.add( o);
}
```

engendre cette erreur de compilation

– **add(capture of ?) in java.util.Collection<capture of ?> cannot be applied to (java.lang.Object)**

Le compilateur ne peut supposer que o est un bien un sous-type de ***?*** *(ce qui sera transmis…)*

si cela était possible nous pourrions écrire

Collection<String>cs = …;

ajouter(cs, new Integer(1)); // et dire: Ah ! Palsambleu ! Ventrebleu !

<?> confer (cf.) l'exemple sur wikipedia

- List<String> a = **new** ArrayList<String>();
- a.add("truc");

- List<?> b = a; *// ici b est déclarée **une liste de n'importe quoi***

- Object c = b.get(0);

*// **correct**, le type "?" est bien (ne peut-être qu') un sous-type de Object*

// mais l'ajout d'un entier à b.

- b.add(**new** Integer (1));

// engendre bien une erreur de compilation, « signature inconnue »
// il n'est pas garanti que Integer soit un sous-type du paramètre "?"

- String s = b.get(0);

// incorrect, il n'est pas garanti que String soit un sous-type du paramètre "?"

- b.add(**new Object()**);
- *// incorrect, il n'est pas garanti que Object soit compatible avec "?"*
- b.add(**null**);

// correct, c'est la seule exception

http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

Tout est <?> , le super type, covariance…

- **Couple<?> c = new Couple<Integer>();**

Attention, inférence de c en Couple<Object>

- **Object o = c.getA();**

et non

- **Integer o = c.getA();** *// erreur de compilation !*

- **c.setA(new Integer(3));** *// erreur de compilation !, rappel*

- *https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html*

Contraintes

Quelles contraintes sur l’arbre d’héritage

- ? extends T

- ? super T

- <?>

- **{co|contra|bi|in}variance Voir : Introducing Variance into the Java Programming Language. A quick Tutorial. C.Plesner Hansen,E. Ernst, M. Torgesen, G. Bracha, June 2003**
  - https://docplayer.net/13447865-Introducing-variance-into-the-java-programming-language-draft.html
  - https://www.researchgate.net/publication/221496192_On_Variance-Based_Subtyping_for_Parametric_Types

? extends T

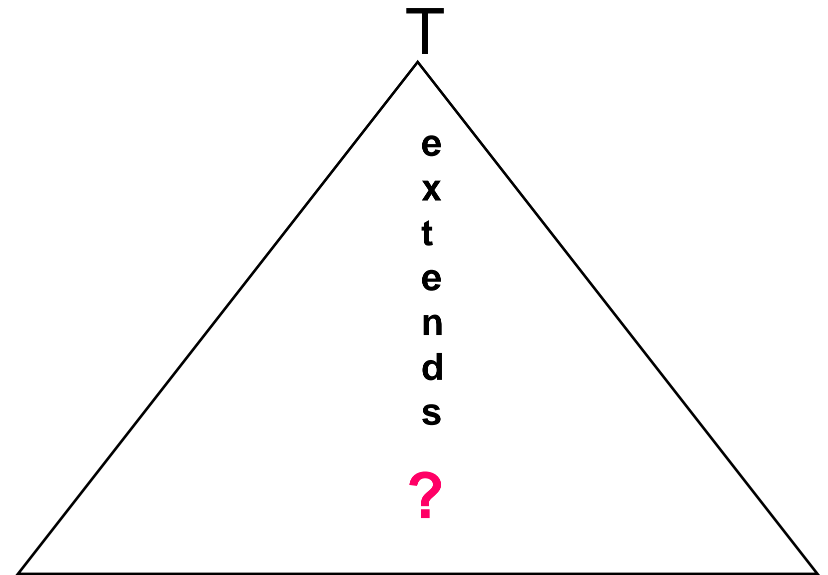
- ? extends T
  - T représente la borne supérieure (super classe)
  - Syntaxe:

    Vector<? extends Number> vn =new Vector<Integer>();

    Integer est une sous classe de Number alors

    Vector<Integer> est bien une sous classe de Vector<? extends Number>

La classe C<T> est en covariance dans T
si pour toute classe A,B :
B une sous_classe de A,
C<B> est une sous_classe de C<A>



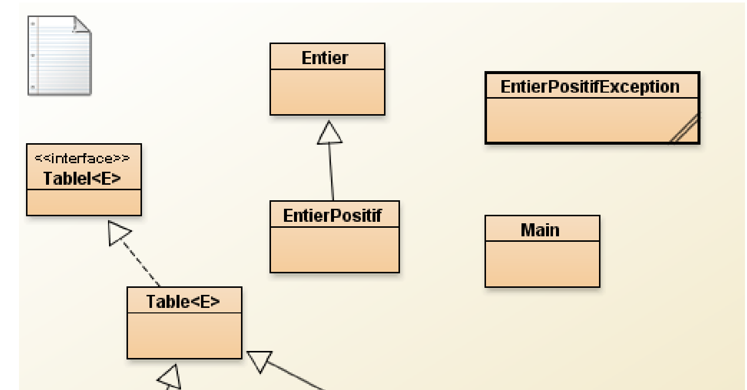
Exemples, ? extends T

- **List<? extends Number> c = new ArrayList<Integer>();**
  - **Quel est le type retourné ? Object, Number, Integer ?**
    - **???? n = c.get(0)**
  - **c.add(3);**
    - **// correct ou erreur de compilation ?**

- *Classe et Interface même syntaxe ici*
- **List<? extends Serializable> l = new ArrayList<String>();**
  - **Quel est le type retourné ? Object, Serializable, String ?**
    - **???? s = l.get(0)**

Utilisation <? extends T>

```
Table<? extends Entier> t1 = new Table<Entier>();
//t1.ajouter(new EntierPositif(3));
//t1.ajouter(new Entier(3));
Entier e1 = t1.lire(0);
Object o1 = t1.lire(0);
```




```
Table<? extends Entier> t2 = new Table<EntierPositif>();
//t2.ajouter(new EntierPositif(3));
//t2.ajouter(new Entier(3));
Entier e2 = t2.lire(0);
Object o2 = t2.lire(0);
```

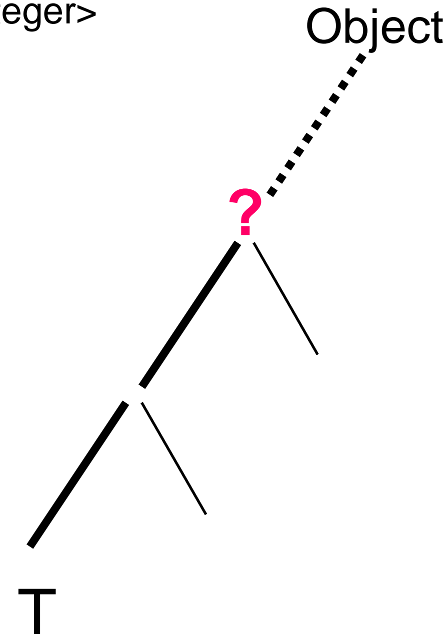
? super T

- ? super T
  - T représente la borne inférieure (sous classe)
  - Syntaxe:

    Vector<? super Integer> vn =new Vector<Number>();

    Integer est une sous classe de Number alors

    Vector<Number> est une sous classe de Vector<? super Integer>

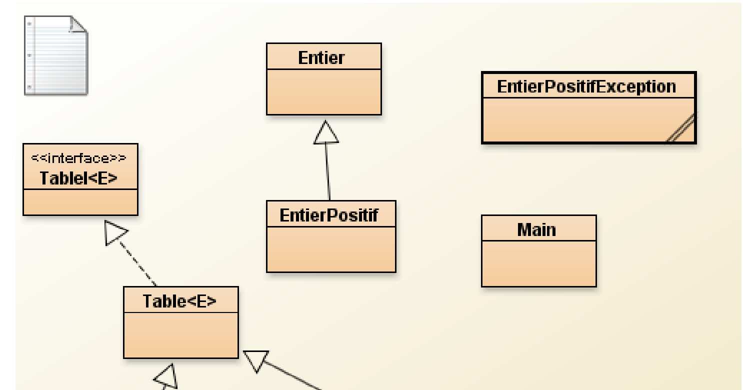


La classe C<T> est en contravariance dans T
si pour toute classe A,B :
B une sous_classe de A,
C<A> est une sous_classe de C<B>

Exemples, ? super T

- List<? super Integer> c = new ArrayList<Integer>();
  - Integer i = c.get(0)
    - // correct ou erreur de compilation ?
  - c.add(new Integer(3));
    - // correct ou erreur de compilation ?

Utilisation <? super T>




```
Table<? super EntierPositif> t3 = new Table<Entier>();
t3.ajouter(new EntierPositif(3));
//t3.ajouter(new Entier(3));
//EntierPositif e3 = t3.lire(0);
Object o3 = t3.lire(0);
```

Contraintes sur les interfaces

- **Interfaces implémentées**

- **Contraintes souhaitées**

Composition : &

- TypeVariable keyword $Bound_1$ & $Bound_2$ & … & $Bound_n$

- pour l 'exemple : une liste ordonnée, « serializable » et « clonable »
- Dont les éléments possèdent une relation d'ordre

```
public class SortedList<T extends Object & Comparable<? super T>
                                          & Cloneable & Serializable>
    extends AbstractCollection<T>
    implements Iterable<T>, Cloneable, Serializable{
```

Composition &, suite de l'exemple

```
public class Entier
      implements java.io.Serializable,Cloneable,Comparable<Entier>{
   private int valeur;
   public Entier(int val){ this.valeur=val;}
   public int compareTo(Entier e){
     return this.valeur-e.valeur;
   }
   public String toString(){return Integer.toString(valeur);}
 }

 public void test1(){
   SortedList<Entier> sl = new SortedList<Entier>();
   sl.add(new Entier(5)); sl.add(new Entier(3));
   sl.add(new Entier(8)); sl.add(new Entier(3));

   for( Entier e : sl){
     System.out.println("e : " + e);
    }
   System.out.println("sl : " + sl);
 }
```

Méthodes génériques, syntaxe

T est un paramètre générique de la méthode

```
<T> void ajouter(Collection<T> c,T t){
  c.add(t);
}

<T> void ajouter2(Collection<? extends T> c,T t){
  //c.add(t); // erreur  de compilation ….
}

<T> void ajouter3(Collection<? super T> c,T t){
  c.add(t);
}
```

Contraintes décompilées

```
public <T extends Number> void p(T e){}
```

devient

```
public void p(Number number) {}
```

```
public <T extends Number> void p2(Collection<? super T> c, T t){}
```

devient

```
public void p2(Collection c, Number t){}
```

Un premier résumé

- **List<? extends Number> c = new ArrayList<Integer>(); // Read-only*,**
- ***// c.add(new Integer(3)); // erreur, Aucune connaissance des sous-classes***
- ***Number n = c.get(0);***
- ***// c.add(n); // erreur n n'est pas compatible avec ?***

- **List<? super Integer> c = new ArrayList<Number>(); // Write-only*,**
- ***c.add(new Integer(3)); // ok***
- ***Integer i = c.iterator().next(); // erreur   ? Comme Object***
- ***Object o = c.iterator().next(); // ok***

- **List<?> c = new ArrayList<Integer>();**
- **System.out.println(" c.size() : " + c.size());**
- **c.add(new Integer(5)); // erreur de compil**

- *termes utilisés ici
- https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html

La suite

- <T extends Comparable<T>>
- <T extends Comparable<? super T>>
- …

Les interfaces… de java.util

- Collection<E>
- Comparator<T>
- Enumeration<E>
- Iterator<E>
- List<E>
- ListIterator<E>
- Map<K,V>
- Map.Entry<K,V>
- Queue<E>
- Set<E>
- SortedMap<K,V>
- SortedSet<E>

L 'interface Comparable...java.lang

```
public interface Comparable<T>{
  int compareTo(T o1);
}
```

Exemple : une classe Entier

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

  private int value;
  public Entier (int value) { this.value = value; }
  public int intValue () { return value; }
  public String toString(){
    return Integer.toString(value);
  }

  public int compareTo (Entier o) {
    return this.value - o.value; // enfin lisible
  }
}
```

Méthode statique et générique, un exemple

```
public static <T extends Comparable<T>> T max(Iterable<T> c)
 {

     Iterator<T> it = c.iterator();
     assert c.size() >=1;
     T x = it.next();
     while (it.hasNext()) {
       T y = it.next();
       if (x.compareTo(y) < 0) x = y;
     }
   return x;
 }

java.util.Collection<Entier> coll = new java.util.ArrayList<Entier>();
coll.add(new Entier(3));
coll.add(new Entier(5));
coll.add(new Entier(2));
assertEquals(new Entier(5).toString(),EntierTest.max(coll).toString());
coll.add(new Entier(7));
coll.add(new Entier(6));
assertEquals(new Entier(7).toString(),EntierTest.max(coll).toString());
```

Un autre exemple de méthode générique

Le patron Visiteur<T>

```
public abstract class Visiteur<T>{
  T visite(Nœud nœud);
  T visite(….);
   …
```

```
public class Nœud extends …{

  public <T> T accepter(Visiteur<T> v){
    return v.visite(this);
  }
```

petites remarques après coup, entre nous

```
public static <T extends Comparable<? super Comparable>>
                  T  max(Iterable<T> c) { … }
```

Au lieu de

```
public static <T extends Comparable<T>>
                T max(Iterable<T> c) { … }
```

discussion……

Un autre exemple, extrait du web, isSorted

```
// http://stackoverflow.com/questions/3047051/how-to-determine-if-
   a-list-is-sorted-in-java

public static <T extends Comparable<? super T>>
                              boolean isSorted(Iterable<T> iterable) {

    Iterator<T> iter = iterable.iterator();
    if (!iter.hasNext()) {
        return true;
    }
    T t = iter.next();
    while (iter.hasNext()) {
        T t2 = iter.next();
        if (t.compareTo(t2) > 0) {
            return false;
        }
        t = t2;
    }
    return true;
}
```

petites remarques après coup, tjs entre nous

```
public class Entier implements Comparable<Entier> {
  ...
  public int compareTo (Entier o) {
    return this.value - o.value; // enfin lisible
  }

  public boolean equals(Entier o){ // attention
    return this.value == o.value; // enfin lisible …
  }
}


java.util.Collection<Entier> coll = new java.util.ArrayList<Entier>();
coll.add(new Entier(3));
boolean b = coll.contains(new Entier(3));  // b == false !!!

Object o = new Entier(3);
```

Tableaux et généricité <?>

```
List<Integer>[] t = new ArrayList<Integer>[10]; erreur !

    List<?>[] t = new ArrayList<?>[10]; // ok
    t[0] = new ArrayList<Integer>();
    //t[0].add(5);                      // erreur !
    //Integer i = t[0].get(0); // erreur !
    Object o = t[0].get(0); // ok


// Mais
    List<Integer> l = new ArrayList<Integer>();
    l.add(5);
    t[1] = l;
    Integer i = (Integer)t[1].get(0); // ok
    String s = (String)t[1].get(0); // compilation ok
                                     // ClassCastException
```

Instanceof et cast et plus

- **instanceof**

```
Collection<? extends Number> c = new ArrayList<Integer>();
    assert c instanceof ArrayList<Integer>; // erreur, et pas de sens
```

- **(cast)**

```
Collection<String> cs2 = (Collection<String>) c;  // ok, mais
Collection<String> cs2 = (Collection) c;  // ok
```

- **getClass()**

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
assert l1.getClass() == l2.getClass();          //   l 'assertion est vraie
```

Surcharge … achtung*

```
public class SurchargeQuiNEnEstPas{
    public void p( Collection<Integer> c){ }
    public void p( Collection<String> c){ }
}

public class C<T> {
  public T id(T t) {;}
}
public interface I<E> {
     E id(E e);
}

public class D extends C<String> implements I<Integer>{

  public Integer id(Integer i){ return i;}
}// erreur name clash, id(T) in C and id(E) have the same erasure
```

*extrait de http://www.inf.fu-berlin.de/lehre/WS03/OOPS/Generizitaet.ppt

Inspiré du Quiz (voir en annexe)

```
public class Couple<T>{
   private static int compte;
   public Couple(){
     compte++;
   }
   public static int leCompte(){ return compte;}
 ….
```

```
Couple<Integer> c = new Couple<Integer>();
Couple<String> c = new Couple<String>();
```

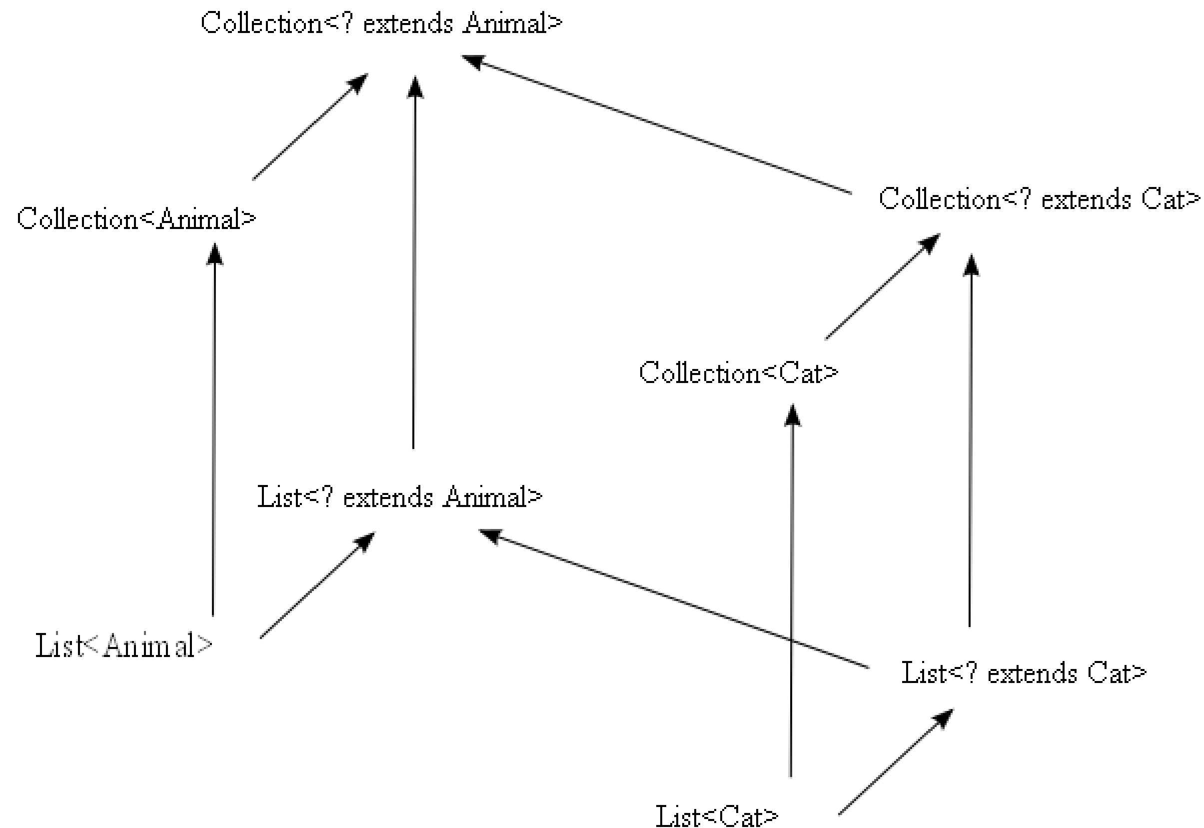
- Que vaut Couple.leCompte(); ?

Conclusion

- **Difficile de s'en passer !**

- **Avantages :**
  - **Lisibilité des programmes,**
  - **Documentation ...**
  - **Vers une qualité des sources ?**
  - **Fiabilité attendue …**

« wikipedia »covariance et contravariance



- http://upload.wikimedia.org/wikipedia/commons/3/3e/Java_wildcard_subtyping.svg
- http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

En java…

```
public class Main{

  public static void a1(Collection<? extends Animal> collection){}
  public static void a2(Collection<Animal> collection){}
  public static void la1(List<? extends Animal> collection){}
  public static void la2(List<Animal> collection){}

  public static void c1(Collection<? extends Cat> collection){}
  public static void c2(Collection<Cat> collection){}
  public static void lc1(List<? extends Cat> collection){}
  public static void lc2(List<Cat> collection){}

  public static void testSubTyping(){
    List<Animal> listA = null;
    a1(listA);
    a2(listA);
    la1(listA);
    la2(listA);
    List<Cat> listC = null;
    c1(listC);
    c2(listC);
    lc1(listC);
    lc2(listC);

    a1(listC);
    la1(listC);
    //la2(listC); erreur, normale ici …
```

Annexe 1

Quiz: Java 1.5 Generics

One of the new features of Java 1.5 are the Generics. They allow us to abstract over types and help us to write type safe code. The generics are in some aspects similar to the templates in C++, but there are also significant differences.

You can learn more about the generics in this tutorial. Also you can have a look at this collection of links about Java 1.5 Generics

Have you read it already? Then you can test your knowledge here.

Lets go

- **http://www.grayman.de/quiz/java-generics-en.quiz**

Le quiz en capture d'écran, http://www.grayman.de/quiz/java-generics-en.quiz

Quiz: Java 1.5 Generics: Question No. 1/14

With generics the compiler has more information about the types of the objects, so explicit casts don't have to be used and the compiler can produce type safe code.

What implications have the generics for the runtime performance of the program which uses them?

Choose the correct answer:

a) ○ With the generics the compiler can optimize the code for used types. This and the omission of the casts are the reasons why the code compiled with the generics is **quicker** than the one compiled without.

b) ○ The usage of generics has **no implications** for the runtime performance of the compiled programs.

c) ○ The improved flexibility and type safety means that the compiler has to generate concrete implementation from the generic template for each used type. This means that applications start **a bit slower**.

Answer

Quiz: Java 1.5 Generics: Question No. 2/14

As an example for a generic class we will use a very simple container. A **Basket** can contain only one element.

Here the source code:

```
public class Basket<E> {
        private E element;

        public void setElement(E x) {
                element = x;
        }

        public E getElement() {
                return element;
        }
}
```

We will store fruits in the baskets:

```
class Fruit {
}

class Apple extends Fruit {
}

class Orange extends Fruit {
}
```

What would Java 1.5 do with the following source code?

```
Basket<Fruit> basket = new Basket<Fruit>(); // 1
basket.setElement(new Apple()); // 2
Apple apple = basket.getElement(); // 3
```

Choose the correct answer:

a) ○ The source code is OK. Neither the compiler will complain, nor an exception during the runtime will be thrown.
b) ○ Compile error in the line 2.
c) ○ Compile error in the line 3.

Quiz: Java 1.5 Generics: Question No. 3/14

Let's stay with our baskets. What do you think about the following source code?

```
Basket<Fruit> basket = new Basket<Fruit>();
basket.setElement(new Apple());
Orange orange = (Orange) basket.getElement();
```

Choose the correct answer:

a) ◯ The source code is OK. Neither the compiler will complain, nor an exception during the runtime will be thrown.
b) ◯ Compile error in the line 2.
c) ◯ Compile error in the line 3.
d) ◯ A **ClassCastException** will be thrown in the line 3.

Answer

Quiz: Java 1.5 Generics: Question No. 4/14

Which ones of the following lines can be compiled without an error?

Check all correct options:

a) ☐ `Basket b = new Basket();`

b) ☐ `Basket b1 = new Basket<Fruit>();`

c) ☐ `Basket<Fruit> b2 = new Basket<Fruit>();`

d) ☐ `Basket<Apple> b3 = new Basket<Fruit>();`

e) ☐ `Basket<Fruit> b4 = new Basket<Apple>();`

f) ☐ `Basket<?> b5 = new Basket<Apple>();`

g) ☐ `Basket<Apple> b6 = new Basket<?>();`

Answer

Quiz: Java 1.5 Generics: Question No. 5/14

Let's have a look at the typeless baskets and the ones where the type is an unbounded wildcards.

```
// Source A
Basket<?> b5 = new Basket<Apple>();
b5.setElement(new Apple());
Apple apple = (Apple) b5.getElement();
```

```
// Source B
Basket b = new Basket();
b.setElement(new Apple());
Apple apple = (Apple) b.getElement();
```

```
// Source C
Basket b1 = new Basket<Orange>();
b1.setElement(new Apple());
Apple apple = (Apple) b1.getElement();
```

Which of the following statements are true?

Check all correct options:

a) ☐ Source A cannot be compiled

b) ☐ Source B will be compiled with warning(s). No exception will be thrown during the runtime.

c) ☐ Source C will be compiled with warning(s). A **ClassCastException** exception will be thrown during the runtime.

Answer

Quiz: Java 1.5 Generics: Question No. 6/14

And what aobout this one?

```
Basket b = new Basket(); // 1
Basket<Apple> bA = b; // 2
Basket<Orange> bO = b; // 3
bA.setElement(new Apple()); // 4
Orange orange = bO.getElement(); // 5
```

Choose the correct answer:

a) ○ The lines 2 and 3 will cause a compile error.
b) ○ The line 4 will cause a compile error.
c) ○ The line 5 will cause a compile error because a cast is missing
d) ○ The source code will be compiled with warning(s). During the runtime a **ClassCastException** will be thrown in the line 5.
e) ○ The soure code will be compiled with warning(s). No exception will be thrown during the runtime.

Answer

Quiz: Java 1.5 Generics: Question No. 7/14

In our rich class hierarchy the class **Apple** has following subclasses:

```
class GoldenDelicious extends Apple {}
class Jonagold extends Apple {}
```

Our fruit processing application contains an utility class which can decide, whether an apple is ripe:

```
class FruitHelper {
        public static boolean isRipe(Apple apple) {
                ...
        }
}
```

In the class **FruitHelper** we want to implement a method which can look into any basket which can contain apples only and decide, whether the apple in the basket is ripe of not. Here the body of the method:

```
{
        Apple apple = basket.getElement(); // 1
        return isRipe(apple); // 2
}
```

What should the signature of the method look like:

Choose the correct answer:

a) `public static boolean isRipeInBasket(Basket basket)`

b) `public static boolean isRipeInBasket(Basket<Apple> basket`

c) `public static boolean isRipeInBasket(Basket<?> basket)`

d) `public static boolean isRipeInBasket(Basket<? extends Apple> basket)`

e) `public static <A extends Apple> boolean isRipeInBasket(Basket<A> basket)`

f) `public static <A> boolean isRipeInBasket(Basket<A extends Apple> basket)`

g) `public static boolean isRipeInBasket(Basket<T super Apple> Basket)`

Quiz: Java 1.5 Generics: Question No. 8/14

Now we want to implement a method which inserts only ripe apples into the basket. Here the method's body:

```
{
        if (isRipe(apple)) { // 1
                basket.setElement(apple); // 2
        }
}
```

Which of these signatures should we use?

Choose the correct answer:

a) ◯ `public static void insertRipe(Apple apple, Basket<Apple> basket)`

b) ◯ `public static void insertRipe(Apple apple, Basket<? extends Apple> basket)`

c) ◯ `public static void insertRipe(Apple apple, Basket<? super Apple> basket)`

d) ◯ `public static <A extends Apple> void insertRipe(A apple, Basket<? super A> basket)`

e) ◯ `public static <A super Apple> void insertRipe(A apple, Basket<? extends A> basket)`

Answer

Quiz: Java 1.5 Generics: Question No. 9/14

We could acquire some expertise in the orangeology and now we can decide whether an orange is ripe or not - and this in pure Java. Now we want to extend the class **FruitHelper**.

Here is our updated source code :

```
class FruitHelper {
        public static boolean isRipe(Apple apple) {
                ... // censored to protect our know-how
        }

        public static boolean isRipe(Orange orange) {
                ... // censored to protect our know-how
        }

        public static boolean isRipeInBasket(Basket<? extends Apple> basket) {
                Apple apple = basket.getElement();
                return isRipe(apple);
        }

        public static boolean isRipeInBasket(Basket<? extends Orange> basket) {
                Orange orange = basket.getElement();
                return isRipe(orange);
        }
}
```

Ist this source code OK?

Choose the correct answer:

a) ○ Yes. The source code is OK.

b) ○ No. The source code cannot be compiled.

Answer

Quiz: Java 1.5 Generics: Question No. 10/14

What about the following source code. Can it be compiled?

```
class FruitHelper {
        public static boolean isRipe(Apple apple) {
                ... // censored to protect our know-how
        }

        public static boolean isRipe(Orange orange) {
                ... // censored to protect our know-how
        }

        public static <A extends Apple>
        void insertRipe(A a, Basket<? super A> b)
        {
                if (isRipe(a)) {
                        b.setElement(a);
                }
        }

        public static <G extends Orange>
        void insertRipe(G g, Basket<? super G> b)
        {
                if (isRipe(g)) {
                        b.setElement(g);
                }
        }
}
```

Choose the correct answer:

a) ◯ Yes. The source code is OK.

b) ◯ No. The source code cannot be compiled.

Answer

Quiz: Java 1.5 Generics: Question No. 11/14

The accounting departement needs to know, how many baskets we produce. So we've changed the class **Basket**:

```
public class Basket<E> {

  ...

        private static int theCount = 0;
        public static int count() {
                return theCount;
        }

        Basket() {
                ++theCount;
        }

        ...
}
```

What output would be produced by the following source code?

```
public static void main(String[] args) {
        Basket<Apple> bA = new Basket<Apple>();
        Basket<Orange> bG = new Basket<Orange>();
        System.out.println(bA.count());
}
```

Choose the correct answer:

a) ◯ 1
b) ◯ 2
c) ◯ Compile error

Answer

Quiz: Java 1.5 Generics: Question No. 12/14

What abut the following source code?

```
Basket<Orange> bG = new Basket<Orange>(); // 1
Basket b = bG; // 2
Basket<Apple> bA = (Basket<Apple>)b; // 3
bA.setElement(new Apple()); // 4
Orange g = bG.getElement(); // 5
```

Choose the correct answer:

a) ◯ No compile error, no exception during the runtime
b) ◯ Compile error in the line 3
c) ◯ **ClassCastException** the line 3
d) ◯ Compile warning in the line 3, **ClassCastException** in the line 4
e) ◯ Compile warning in the line 3, **ClassCastException** in the line 5

Answer

Quiz: Java 1.5 Generics: Question No. 13/14

And what about this one?

```
Basket<Orange> bG = new Basket<Orange>(); // 1
Basket<? extends Fruit> b = bG; // 2
if (b instanceof Basket<Apple>) { // 3
        Basket<Apple> bA = (Basket<Apple>) b; // 4
        bA.setElement(new Apple()); // 5
} // 6
Orange g = bG.getElement(); // 7
```

Choose the correct answer:

a) ○ No compiler error, no exception
b) ○ Compiler error in the line 3
c) ○ Compiler warning in the lines 3 and 4. An exception will be thrown in the line 7.

Answer

Quiz: Java 1.5 Generics: Question No. 14/14

The last question is about arrays and generics. Which of the following lines can be compiled?

Check all correct options:

a) ☐ `Basket<Apple>[] b = new Basket<Apple>[10];`

b) ☐ `Basket<?>[] b = new Basket<Apple>[10];`

c) ☐ `Basket<?>[] b = new Basket<?>[10];`

d) ☐ `public <T> T[] test() {return null;}`

e) ☐ `public <T> T[] test() {return new T[10];}`

Answer

À l'origine http://www.grayman.de/quiz/java-generics-en.quiz

- 1/14 {b}
- 2/14 {c}
- 3/14 {d}
- 4/14 {a,b,c,f}
- 5/14 {a,b}
- 6/14 {d}
- 7/14 {d,e}
- 8/14 {d}
- 9/14 {b}
- 10/14 {a}
- 11/14 {b}
- 12/14 {e}
- 13/14 {b}
- 14/14 {c,d}

Annexe 2, les Exceptions sont génériques…

- **Utile / inutile …**

```
public interface PileI<T, PV extends Exception,
                          PP extends Exception>{

  public void empiler(T elt) throws PP; // PP comme PilePleine
  public T depiler() throws PV;         // PV comme PileVide
}
```

- **Le type de l'exception est décidée par l'utilisateur**

Exception et généricité

à l 'aide de deux implémentations concrètes

PileVideException et PilePleineException

```
public class Pile<T>
  implements PileI<T, PileVideException , PilePleineException >{

  private Object[] zone;
  private int index;
  public Pile(int taille){…}

  public void empiler( T elt) throws PilePleineException {
    if(estPleine()) throw new PilePleineException();
```

Exception et généricité suite ...

```
PileI<Integer,ArrayIndexOutOfBoundsException,ArrayIndexOutOfBoundsException>

   p1 = new PileI<Integer,ArrayIndexOutOfBoundsException,ArrayIndexOutOfBoundsException>(){

         public  void empiler(Integer elt) throws ArrayIndexOutOfBoundsException{
           // …
         }

         public Integer depiler() throws ArrayIndexOutOfBoundsException{
          // …
         }
       };
   }
```

```
try{
  p1.empiler(3);
}catch(Exception e){
  // traitement
}
```

l 'utilisateur a donc le choix du type de l'exception à propager

Exceptions et généricité : un résumé

- Une interface

```
public interface I< T, E extends Throwable>{
     public void testNull(T t) throws E;
}
```

- Une implémentation

```
public class Essai<T extends Number> implements I<T, NullPointerException>{

   public void testNull(T t) throws NullPointerException{
     if(t==null) throw new NullPointerException();
    }
}
```

- Une autre implémentation

```
I<? extends Float,NullException> i = new I<Float,NullException>(){
          public void testNull(Float t) throws NullException{
           if(t==null) throw new NullException();
          }
      };

avec public class NullException extends Exception{}
```

Annexe 3 Le patron fabrique

- **Discussion de l'usage de l'introspection au sein du patron fabrique**

- *Utile/inutile …*

Essai d'une fabrique générique

- **La classe Class est générique** mais

```
public class Fabrication<T> {
  public T fabriquer(){
    return T.class.newInstance(); // erreur de compil
    return new T();               // erreur de compil
  }
}
```

- **Mais …. avons nous ?**

public T getInstance(Class<T>, int id)

Classe Class générique

- `Class<?> cl1 = Integer.class;`
- `Class<? extends Number> cl = Integer.class;   // ok`

```
 public  < T extends Number> T[] toArray(int n, Class<T> type){
     T[] res = (T[])Array.newInstance(type,n);
     return res;
    }

Integer[] t = toArray(4, Integer.class);   // satisfaisant
```

La Fabrication revisitée

```
public class Fabrication<T> {

  public T fabriquer(Class<T> type) throws Exception{
    return type.newInstance();
  }
}
```

- Usage :

```
Integer i = new Fabrication<Integer>().fabriquer(Integer.class);
Number n = new Fabrication<Integer>().fabriquer(Integer.class);
```

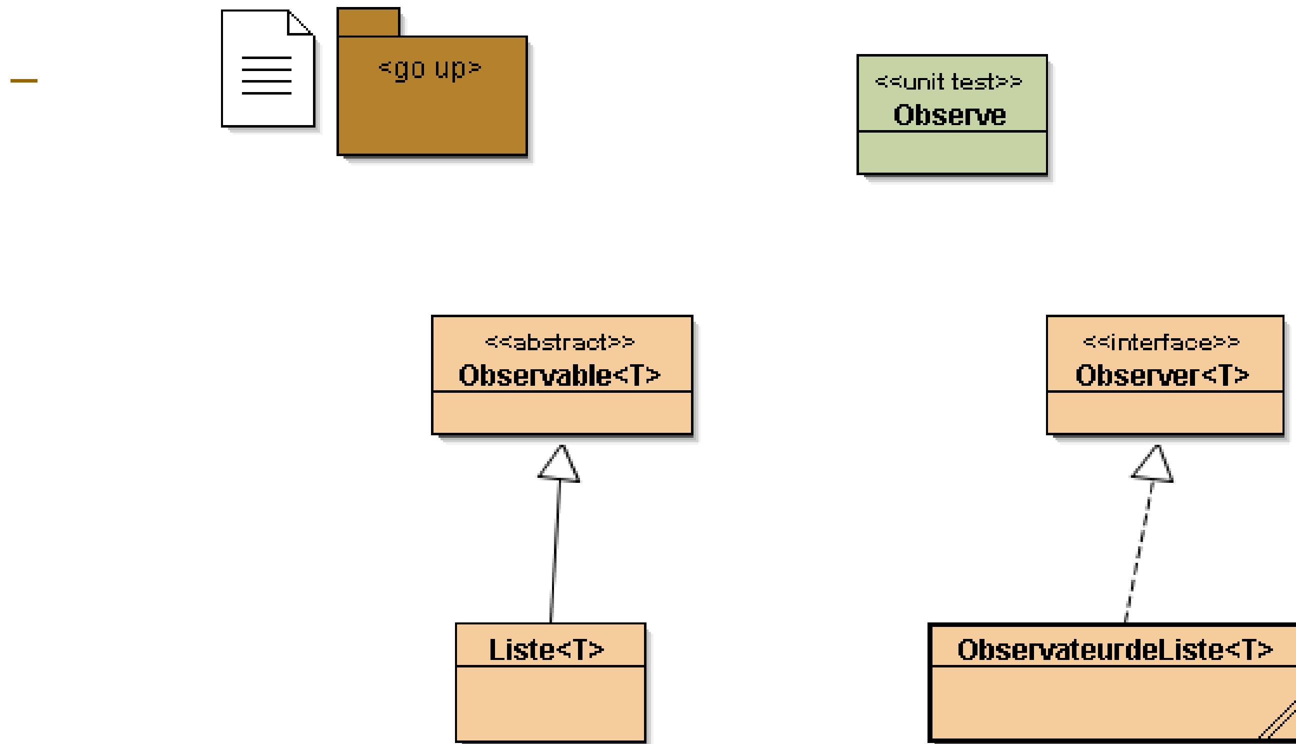
Exercice : Set<Integer> = new Fabrication<…..>()

Utile/inutile …

Pattern et généricité

- **Première tentative**
- **Observateur/observé**
- **<M,V,C> ?**

Pattern Observateur, un classique



Avant nous avions

```
public interface Observer {
  void update(Observable obs, Object arg)
}
```

Observer et Observable, arg est paramétré

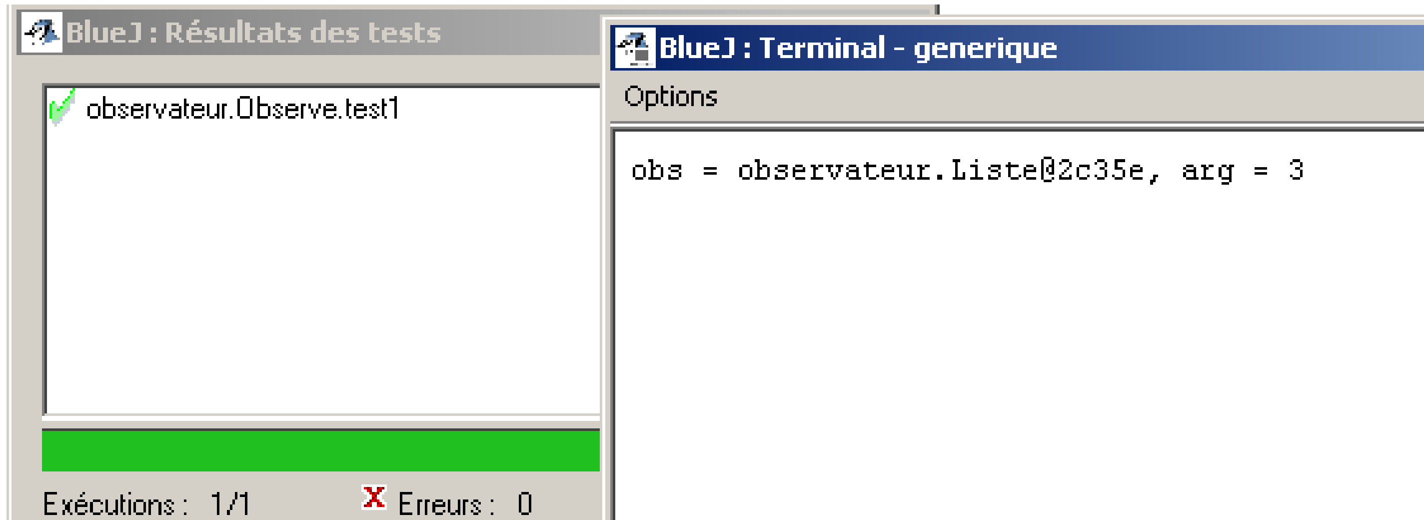
```
public interface Observer<T> {
  public void update(Observable obs, T arg);
    // void update(Observable obs, Object arg)
}


public abstract class Observable<T>{
  private List<Observer<T>> list;

  public Observable(){ list = new ArrayList<Observer<T>>();}

  public void addObserver(Observer<T> obs){
    list.add(obs);
  }

  public void notifyObservers(T arg){
    for (Observer<T>  obs : list) {
       obs.update(this, arg);
    }
  }
}
```

Une instance possible : une liste<T>

```
public class Liste<T> extends Observable<T>{

  public void ajouter(T t){
    //....
    notifyObservers(t);
  }
}



public class ObservateurDeListe<T> implements Observer<T>{

  public void update(Observable obs, T arg){
    System.out.println("obs = " + obs + ", arg = " + arg);
  }

}
```

<T>est et <T>race

```
public void test1(){
   Liste<Integer> liste1 = new Liste<Integer>();
   ObservateurDeListe<Integer> obs = new ObservateurDeListe<Integer>();
   liste1.addObserver(obs);
   liste1.ajouter(3);

    ObservateurDeListe<String> obs1 = new ObservateurDeListe<String>();
   // liste1.addObserver(obs1); engendre bien une erreur de compilation

   }
```

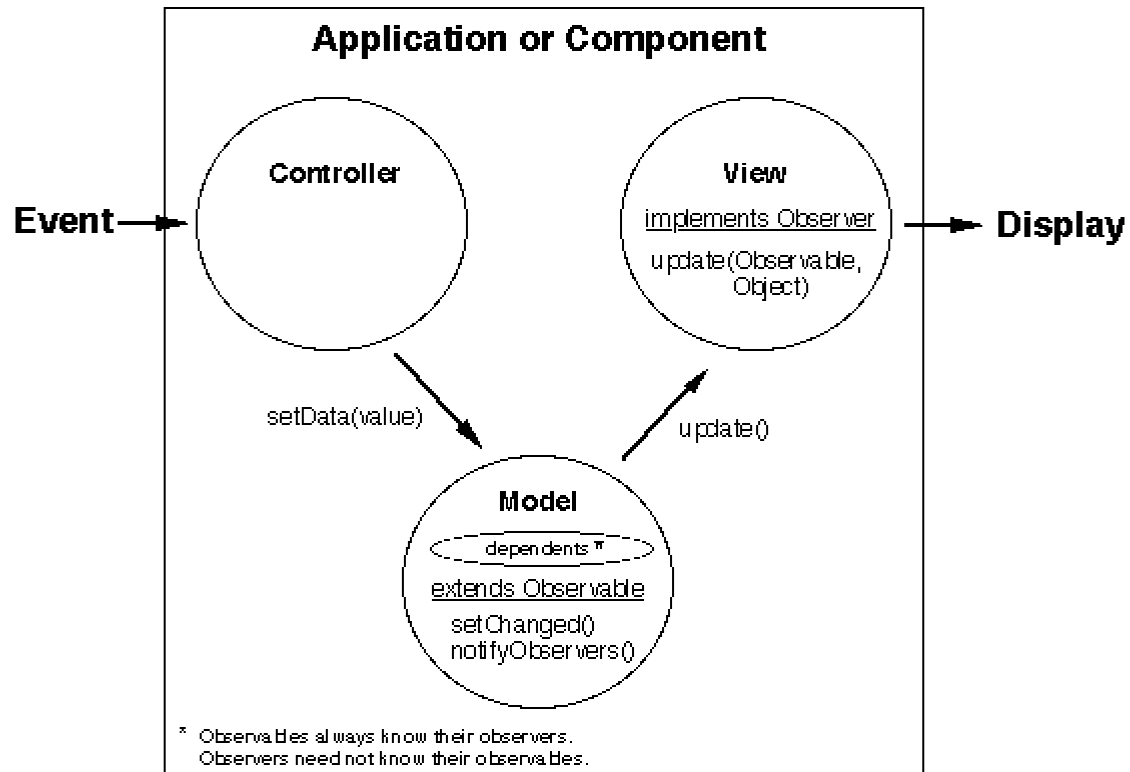


Discussion

- **En lieu et place d'Object , facile et sécurisant**
  - **Object o devient T t avec <T>**

- **Encore plus générique ?**
  - **Découpage selon M, V ,C**
  - **utile/inutile …**

MVC Generic, <M,V,C> ?

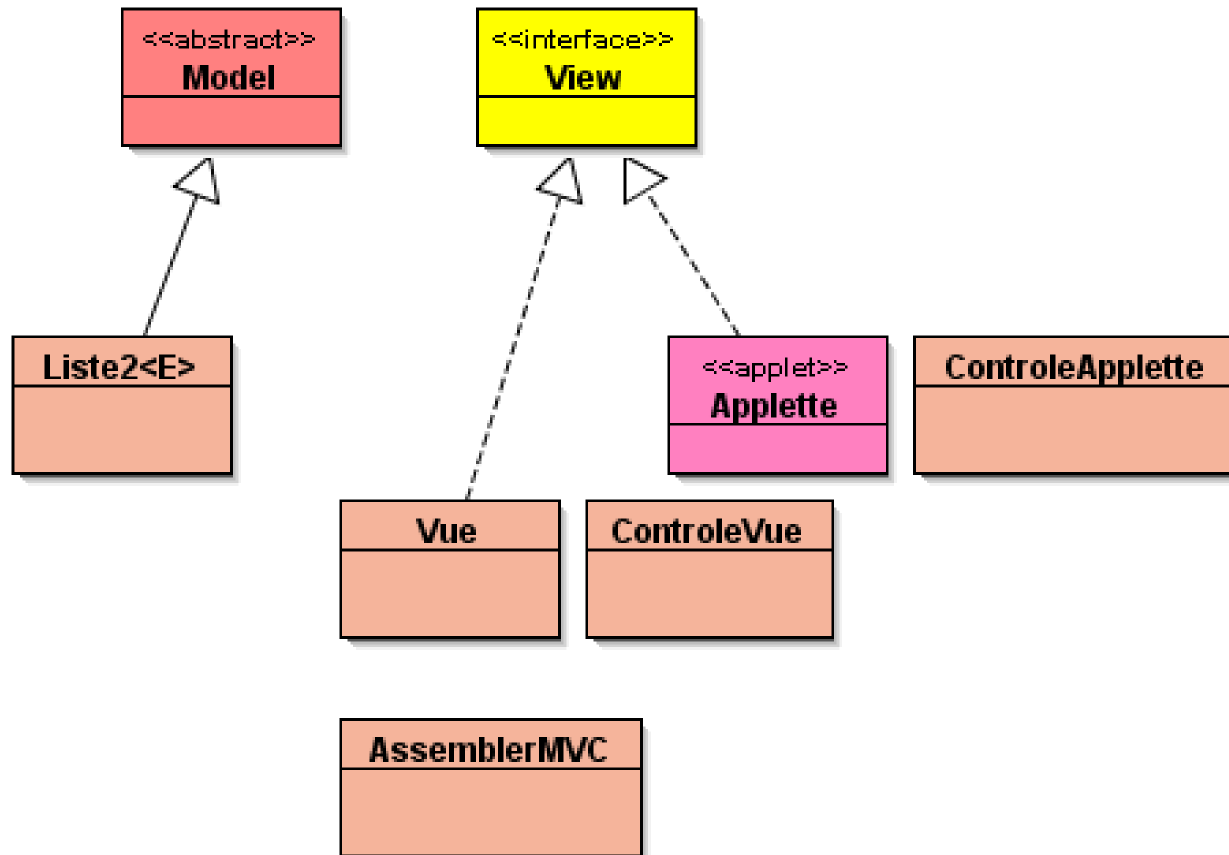



```
public interface View {
  public <M extends Model> void update (M modèle);
}
```

Model

```
public abstract class Model {
  private Set<View> views;

  public Model(){
    this.views = new HashSet<View>();
  }

  public <V extends View> void addViewToModel( V view){
    this.views.add(view);
  }

  public void notifyViews(){
    for( View view : this.views)
      view.update(this);
  }
}
```

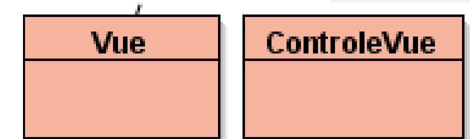
Architecture du déjà vu



- Liste2<E> est un « Model »
- Vue et Applette sont des « View »
- ControleVue et ControleApplette sont les contrôles
  - associés respectivement à la Vue et à l'Applette
- AssemblerMVC assemble un modèle, deux vues et deux contrôles

Vue et son contrôle

```
public class Vue implements View{
  public Vue(Model m){
    m.addViewToModel(this);
  }
  public  <M extends Model> void update(M model){
    System.out.println("notification par : " + model);
  }
}

public class ControleVue{

  private Liste2<Integer>  modele;
  private Vue   vue;

  public ControleVue( Liste2<Integer> modele, Vue vue){
    this.modele = modele;
    this.vue    = vue;
  }

   public void entrer(int i){
     modele.ajouter(i);
   }
}
```

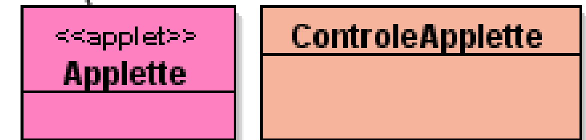


Applette et ...

```
public class Applette extends JApplet implements View{
  private JTextField donnee;
  private JButton add;
  private JLabel liste;

  public Applette(Model m){m.addViewToModel(this);}

  public void init(){
    this.donnee = new JTextField(6);
    this.add    = new JButton("ajouter");
    this.liste  = new JLabel("");
    this.getContentPane().setLayout(new FlowLayout());
    this.getContentPane().add(donnee);
    this.getContentPane().add(add);
    this.getContentPane().add(liste);
  }

  public  <M extends Model> void update(M model){
    liste.setText(model.toString());
  }

  public JButton getBoutonAjouter(){return add;}

  public int getDonnee(){return Integer.parseInt(donnee.getText());}}
```

… son contrôle

```
public class ControleApplette implements ActionListener{

  private Liste2<Integer>  modele;
  private Applette   vue;

  public ControleApplette( Liste2<Integer> modele, Applette vue){
    this.modele = modele;
    this.vue    = vue;
    vue.getBoutonAjouter().addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae){
    modele.ajouter(vue.getDonnee());
  }
}
```



Un Assembleur …

```
public class AssemblerMVC{
  public static void main(String[] args){

    Liste2<Integer> liste = new Liste2<Integer>(); // M

    Vue  vue1 = new Vue(liste);    // V
    ControleVue controle1 = new ControleVue(liste, vue1); // C

    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Applette vue2 = new Applette(liste); //cf. slide 25, cours 4-2
    frame.getContentPane().add(vue2);
    frame.setSize(300, 100);
    vue2.init();
    vue2.start();
    frame.setVisible(true);
    ControleApplette control = new ControleApplette(liste, vue2);
    controle1.entrer(33);
  }
}
```

Discussion

- **<M,V,C>**
  - **Encore plus générique ???**
  - **http://www.onjava.com/pub/a/onjava/2004/07/07/genericmvc.html**
  - **Exercice de style ?**
    - **utile/inutile**