Java et Généricité

Cnam Paris
jean-michel Douin
version du 20/10/2010

Notes de cours java : Généricité en Java

Sommaire

Bibliographie et outils utilisés

- Tutoriel sur la généricité en Java, indispensable
  - http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
  - La génèse en http://citeseer.ist.psu.edu/cache/papers/cs/22885/http:zSzzSzcm.bell-labs.comzSzcmzSzcszSzwhozSzwadlerzSzgjzSzDocumentszSzgj-oopsla.pdf/bracha98making.pdf
- MVC et généricité
  - http://www.onjava.com/pub/a/onjava/2004/07/07/genericmvc.html
- Journal of Object Technology http://www.jot.fm
  - http://www.jot.fm/issues/issue_2004_08/column1/column1.pdf
  - http://www.jot.fm/jot/issues/issue_2004_08/column8/index_html
- Variance en POO
  - http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/variance.html
- Diagnosing Java code: Java generics without the pain **part 1..4**
  - http://www-106.ibm.com/developerworks/java/library/j-djc02113.html
- Une liste de références établies par Angelika Langer
  - http://www.langer.camelot.de/Resources/Links/JavaGenerics.htm
  - http://www.ftponline.com/javapro/2004_03/online/jgen_kkreft_03_03_04/

  http://www.ftponline.com/javapro/2004_03/online/j2ee_kkreft_03_10_04/
- Optimizing Web Services Using Java, Part I - Generic Java and Web services
  - http://www.sys-con.com/webservices/article.cfm?id=736&count=5205&tot=4&page=2
- Un quiz après lecture et mises en œuvre, à faire
  - http://www.grayman.de/quiz/java-generics-en.quiz
- Enforce strict type safety with generics (Create a simple event dispatcher using generics)
  - http://www.javaworld.com/javaworld/jw-09-2004/jw-0920-generics.html
- **Cours GLG203/2009 de Louis Dewez**

Généricité : Introduction

- **Motivation**
- **Inférence de type**
  - **aucun « warning » ==> javac -source 1.5**
  - **Principe d 'erasure**
  - **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:**
  - Collection c = new ArrayList();
  - c.add(new Integer(0));
  - String s = (String) c.get(0);
  - idem lors de l 'usage d 'un itérateur, (java.util.Iterator)
  - la méthode next() retourne un Object
  - String s = (String)c.iterator().next();

<T> 5

Généricité / Généralités

- 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 Test<T>{
  private T x;

  public T getX(){ return x;}
}

Test<Integer> t = new Test<Integer>();
Integer i = t.getX();

Test<String> t = new Test<String>();
String i = t.getX();
```

<T> 6

Avant/Après

```
public void avant(){
  Map map = new HashMap();
  map.put("key", new Integer(1));
  Integer i = (Integer) map.get("key");
}

public void après(){
  Map<String,Integer> map = new HashMap<String,Integer>();
  map.put("key", new Integer(1));
  Integer i = map.get("key");
}
```

Nb: byteCode(après()) est égal au byteCode(avant())
Pour se convaincre
http://members.fortunecity.com/neshkov/dj.html

Un premier exemple

```
public class Couple<T>{      // un couple homogène
  private T a;
  private T b;
  public Couple(T a, T b){
    this.a = a;
    this.b = b;
  }

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

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

Premier exemple : utilisation

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

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

Couple<Object> c = new Couple<Object>(new Integer(5), new String("cheval"));
Object o = c.getA();
```

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
```

<T>

9

Mais « Tout » reste-t-il Object ?

```
Couple<Object> c = new Couple<Object>(new Object(), new Object());

Couple<Integer> c1 = new Couple<Integer>(5, 7);
Integer i = c1.getA();


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


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

Notons que

si **c = c1;** était possible

nous pourrions écrire

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

<T>

10

Un constat

Couple<Object> ne peut recevoir qu'un Couple<Object>
Collection<Object> ne peut …. 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 …et rien d'autre

Les mêmes raisons ?

```
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
```

Et pourtant les tableaux

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

covariance des tableaux

Mais nous avons bien une erreur ici

t[0] = new Boolean(true);
cette affectation déclenche une exception ArrayStoreException

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

<T> 13

Note : Covariance ? les fonctions en Java, depuis 1.5

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


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


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

<T> 14

<?> à la rescousse

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

Alors joker '?'

public static void afficher(java.util.Collection**<?>** c){

- **for(Object o : c)**
- **System.out.println(o);**
- **}**

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

Mais

<T> 15

<?> la contre-partie <?>

? Représente le type inconnu
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)**

c *est uniquement « read-only »* *le compilateur ne peut supposer que o est un bien une sous-type de ? c.f. page suivante*

si cela était possible nous pourrions écrire
Collection<String>cs =
ajouter(cs, new Integer(1));

<T> 16

<?> c.f. 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)

<T> 17

Tout est <?> , le super type

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

Attention, inférence de c en Couple<Object>

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

et non

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

<T> 18

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 (http://fpl.cs.depaul.edu/ajeffrey/se580/pdf/VJGenerics.pdf) et**
- **http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/variance.html**

? extends T

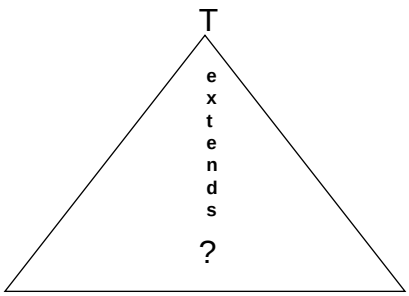
- ? extends T
  - T représente la borne supérieure (super classe)
  - Syntaxe:

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

    Integer est une sous classe de Number alors

    Vector<Integer> est 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>



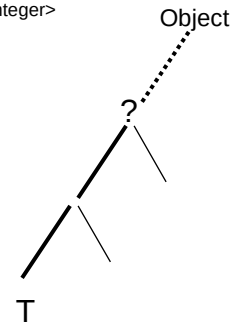
? super T

- ? 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>

<T> 21

Bivariance

- ***Classe C<T> est en bivariance dans T si C<T> est en covariance et en contravariance dans T***

  - ?

    Vector<?> représente la famille des types Vector<T>

    Alors
    Vector<? super Integer> est un sous-type de Vector<?>
    et Vector<? extends Integer> est sous-type de Vector<?>.

<T> 22

Invariance

- ***Classe C<T> invariance dans T si***

  - C<A> est sous-classe de C<B> et seulement si A = B

    - Classe C<T> est un sous type de Classe C<? super T>
    - Classe C<? super T> un sous-type de Classe C<?>

    - Idem pour
      - <? extends T>

Composition : &

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

- **pour l 'exemple : une liste ordonnée, « sérialisable » 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);
 }
```

<T>

25

Un premier résumé

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

<T>

26

Méthodes génériques, syntaxe

```
<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 read_only ….
}

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

<T> 27

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){}
```

<T> 28

Exception et généricité

- **Et si l'exception était générique (enfin, presque)**

```
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
}
```

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{
          throw new ArrayIndexOutOfBoundsException(); // levée pour les tests …
        }

        public Integer depiler() throws ArrayIndexOutOfBoundsException{ return null;}
      };
   }
```

```
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{}
```

<T> 31

Collection, « conteneur »

- **pré-destinées et naturellement générique**
- **voir le paquetage java.util J2SE1.5**
  - **http://java.sun.com/j2se/1.5.0/docs/api/**

<T> 32

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>**

<T> 33

L 'interface Comparable…java.lang

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

<T> 34

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
  }
}
```


Max : Méthode statique et générique

```
public static <T extends Comparable<T>> T max(Collection<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());
```

petites remarques après coup, entre nous

```
public static
  <T extends Comparable<? super Comparable>> T
                                    max(Collection<T> c) {
```

Au lieu de

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

discussion……

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);
```

ArrayList ou le retour de <? extends T>

- **Extrait de java.util**
  - **un des constructeurs possède cette déclaration :**
  - **ArrayList(Collection** <? extends E> **c)**
  - **ou bien la méthode addAll(Collection** <? extends E> **c)**

```
Collection<Number> cc = new ArrayList<Number>(); // correct
Collection<Integer> cc1 = new ArrayList<Integer>(); // correct
cc.addAll(cc1);
```

- **Rappel**

Collection<? extends Number> c = new ArrayList<Integer>(); // correct
c.add(new Integer(3)); *// incorrect, erreur de compilation,*

<? extends T>, <? super T>

```
 public static void section4(){

//
Collection<? extends Number> c = new ArrayList<Integer>();
    c.add(null);
    //c.add(new Integer(5)); erreur de compilation,


//
    Collection<? super Integer> c1 = new ArrayList<Integer>();
    c1.add(null);
    c1.add(new Integer(5));  // ici l 'addition est sûre
  }
```

Tableaux et généricité

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



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
```

<T>

41

Instanceof et cast et plus

- **instanceof**

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

- **(cast)**

```
Collection<String> cs2 = (Collection<String>) cs1;  // errreur pas de sens
```

- **getClass**

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

<T>

42

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)

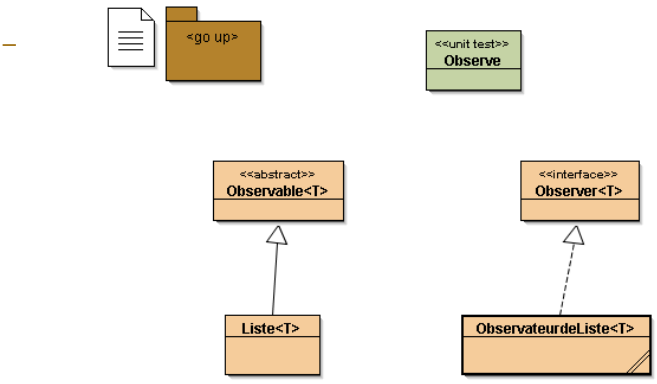
```
public class Couple<T>{
   private static int compte;
   public Couple(){
     compte++;
   }
   public static int leCompte(){ return compte;}
 ….

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

- Que vaut Couple.leCompte(); ?

Pattern et généricité

- **Observateur/observé**
- **MVC**
- **Fabrique**
- **Chaîne de responsabilités**
- **Composite et Visiteur**
- **à compléter (des idées ?)**

Pattern Observateur, un classique



Avant nous avions

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

Observer et Observable, arg est paramétré

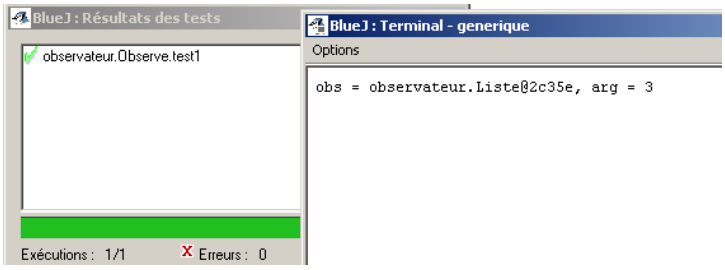
```
public interface Observer<T> {
  public void update(Observable obs, T arg);
    // void update(Observable obs, Object arg)
}


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

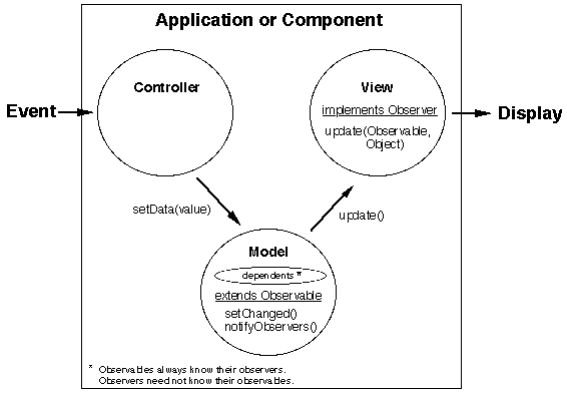
```
public void test1(){
   Liste<Integer> liste1 = new Liste<Integer>();
   ObservateurDeListe<Integer> obs = new ObservateurDeListe<Integer>();
   liste1.addObserver(obs);
   liste1.ajouter(3);

    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>**
  - **(enfin presque new T() ….)**
- **Encore plus générique ?**
- **Et si le Modèle(l'observable) était un Paramètre**

MVC Generic, ModelListener<M>




```
public interface View {
  public <M> void update (M model);
}
```


Model <V extends View>

```
import java.util.Set;
import java.util.HashSet;

public class Model <V extends View>{
  private Set<V> views;

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

  public void addViewToModel( V view){
    this.views.add(view);
  }

  public void notifyy(){
    for( V view : this.views)
      view.update(this);
  }
}
```

Liste2<T> extends Model<View>

```
public class Liste2<T> extends Model<View>{

  public void ajouter(T val){
    // ….
    notifyy();
   }


}
```


Un test, liste2

```
public void test1()
  {
     Liste2<Integer> value1 = new Liste2<Integer>();

     View v1 =
        new ObservateurDeListe2(){
          public void update(Liste2 model){
           System.out.println(model);
          }
        };

     value1.addViewToModel(v1);
     value1.ajouter(3);
  }
```

Discussion

- **Encore plus générique ???**
- **Exercice de style ou**
- **Passage obligatoire, indispensable ?**
- **POO ++**
  - **L'héritage**
  - **+**
  - **Les types paramétrés**

Pattern Factory générique (par définition…)

- **[GoF95] Fabrication, classe créatrice**
- **intention « définir une interface pour la création d 'un objet, mais en laissant à des sous-classes le choix des classes à instancier, la Fabrication permet à une classe de déléguer l 'instanciation à des sous-classes »**
- **Alors naturellement, nous pourrions en déduire**

```
public interface Fabrique<T>{
  public T fabriquer();
}
```

Fabriques d ' ensemble

```
import java.util.Set;
import java.util.HashSet;

public class FabriqueUnEnsemble<E> implements Fabrique<Set<E>>{

  public Set<E> fabriquer(){
    return new HashSet<E>();
  }

}

public class FabriqueUnAutreEnsemble<E> implements Fabrique<Set<E>>{

  public Set<E> fabriquer(){
    return new TreeSet<E>();
  }
}
```


un test pré-fabriqué

```
 public void test1(){

   Fabrique<Set<Integer>> f = new FabriqueUnEnsemble<Integer>();
   Set<Integer> e = f.fabriquer();

   e.add(new Integer(5));
   e.add(new Integer(3));
   e.add(new Integer(5));

     for( Integer i : e)
       System.out.println("i : " + i);

   }
```

Fabrique générique et introspection

- **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)

<T> 59

Classe Class générique

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

```
 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
```

<T> 60

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<…..>()

Pattern Commande

```
public interface Command<T>{
  public void make(T f);
}
```

```
public class ConsoleCommand implements Command<Float>{

  public void make(Float f){
    System.out.println(" f " + f);
  }
}
```

Chaîne de responsabilités

```
public abstract class Handler<T>{
  protected Handler<T> successor = null;

  public Handler(){ this.successor = null;}
  public Handler(Handler<T> successor){ this.successor = successor;}
  public void setSuccessor(Handler<T> successor){ this.successor=successor;}
  public Handler<T> getSuccessor(){return this.successor;}
  public boolean <T extends Number> handleRequest(T value){
    if(successor != null) return successor.handleRequest(value);
    else return false;
  }
}

public class TraceHandler extends Handler<Float>{

  public TraceHandler(Handler<Float> successor){super(successor);}

  public boolean handleRequest(Float value){
    System.out.println("value : " + value);
    return super.handleRequest(value);
  }}
```


Chaîne de responsabilités(2)

```
public abstract class Handler<T>{
  protected Handler<T> successor = null;
  public Handler(){ this.successor = null;}
  public Handler(Handler<T> successor){ this.successor = successor;}
  public void setSuccessor(Handler<T> successor){this.successor = successor;}
  public Handler<T> getSuccessor(){return this.successor;}
  public <T extends Socket> boolean handleRequest(T value){
    if ( successor == null )  return false;
    return successor.handleRequest(value);
  }
}

public class TraceHandler extends Handler<Socket>{
  public TraceHandler(Handler<Socket> successor){super(successor); }
  public <T extends Socket> boolean handleRequest(T s){
    Calendar c = Calendar.getInstance();
    DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT,Locale.FRANCE);
    DateFormat dt = DateFormat.getTimeInstance(DateFormat.SHORT,Locale.FRANCE);
    String date = df.format(c.getTime()) + "-" + dt.format(c.getTime());
    System.out.print("[" + s.getInetAddress().getHostName() + "]- » + date + "-");
    return super.handleRequest(s);
  }
}
```

Autres Patterns

- **Composite et Visiteur**

- **class Component{**
- ** public <T> accept(Visitor<T> v);**
- **}**

- **public Visitor<T>{**
- ** public T visitLeaf(Leaf l);**
- **}**

Conclusion

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

- **Un constat :**
  - **il suffit de parcourir la documentation de Sun (1.5)**

- **Améliorations :**
  - **de la lisibilité des programmes ...**
  - **de leur qualité, fiabilité, ...**

.à voir avec ESC/java et l'opérateur <:

Annexe

Quiz: Java 1.5 Generics

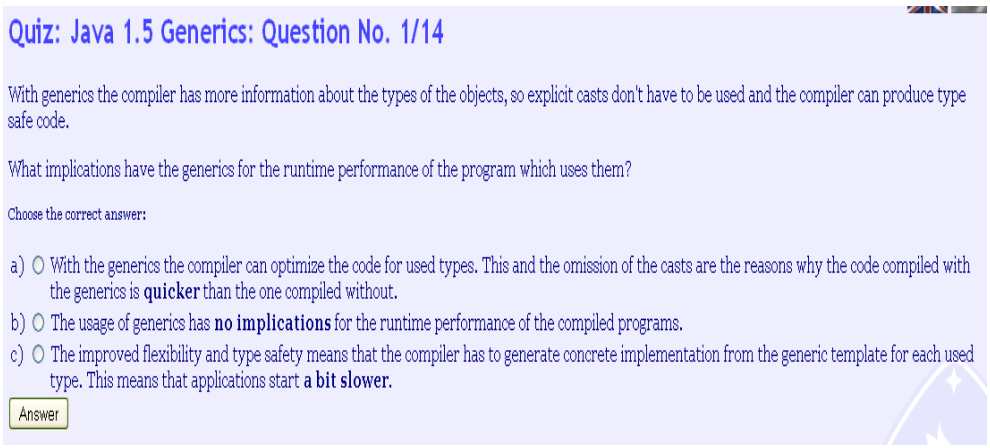
One of the new features of Java 1.5 are the Generics. They allow us to abstract over types and help us to write type safe code. The generics are in some aspects similar to the templates in C++, but there are also significant differences.

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. 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

<T> 77

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

<T> 78

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

<T>

79

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

<T>

80

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

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 {e}
- 12/14 {b}
- 13/14 {b}
- 14/14 {c,d}