IUT d’Orsay - DUT Informatique - Jean-Claude MARTIN – JEAN-CLAUDE.MARTIN@U-PSUD.FRProgrammation Orientée Objet (POO)  
COURS 11 : Programmation Générique



# Introduction

* Classes / Méthodes / Interfaces génériques
  + Utilisable avec des objets ou variables de types quelconques
  + Souvent utilisé avec les collections (ex: ArrayList<Integer>)
  + Attention : différent de héritage et polymorphisme (le type peut varier dans une même méthode)
    - Générique : le type est fixé lors de l’appel de la méthode
* Intérêts
  + Renforcer les vérifications de type à la compilation et à l’exécution
  + Éviter les casts

# Objectif de la programmation générique

* Exemple : un objet instance de la classe Box pourrait stocker un Integer et un autre objet instance de cette même classe permettre de récupérer une instance de String

public class Box {

private Object valeur;

public void set(Object valeur) {

this.valeur = valeur; }

public Object get() { return valeur; }  
}

* Inconvénient
  + Si on essaye de stocker la valeur récupérée sans faire de cast

public class Box {

private Object valeur;

public void set(Object valeur) {

this.valeur = valeur; }

public Object get() { return valeur; }

public static void main (String args []) {

Box box\_integer = new Box ();

box\_integer.set(new Integer (12));

Integer i = box\_integer.get();

}

}

* + => message d’erreur :   
    Type mismatch: cannot convert from Object to Integer
  + => Nécessite de faire un cast lorsque l’on veut récupérer la valeur

Integer i = ………………………………...box\_integer.get();

* Objectifs de la programmation générique
  + Réutiliser le même code avec différents types
  + Sécuriser leur utilisation en rendant unique le type des éléments
  + Eviter des casts et permettre donc un programme plus concis et lisible

**Avec générique => pas besoin de cast :-)**

# 

# Classes génériques

Version générique de la classe Box :

* Définition de la classe générique : “T” est un paramètre de type

public class Box<T> {  
 private T t;  
  
 public void set(T t) { this.t = t; }  
 public T get() { return t; }  
}

* Norme : utiliser une seule majuscule pour nommer un paramètre de type : T, S, U, V ...
* Déclaration :

Box<Integer> integerBox;

On passe le type Integer en paramètre à la classe (comme si on passait un paramètre à une méthode.

On parle de *generic type invocation* : on remplace T par le nom d’une classe (ici Integer)

* Instantiation :

Box<Integer> integerBox = new Box<Integer>();

Ou alors avec le “diamant” et une inférence de type

Box<Integer> integerBox = new Box**<>**();

* Le type “brut” : Box

Box rawBox = new Box();

package generique;

public class Box<T> {

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

public static void main (String args []) {

Box<Integer> integerBox = new Box<Integer>();

integerBox.set(new Integer (12));

Integer i = integerBox.get();

// Ou alors avec le “diamant” et une inférence de type

Box<Integer> integerBox2 = new Box<>();

integerBox2.set(new Integer (33));

Integer j = integerBox2.get();

**// Le type “brut” : Box  
 Box rawBox = new Box();  
 rawBox.set(new Integer (35));  
 // Integer k = rawBox.get(); ERREUR DE COMPILATION :   
 // Type mismatch: cannot convert from Object to Integer  
 int k = (Integer) rawBox.get();**

}

}

## 

## Définition d’une classe générique à un seul type

Exemple d’un couple de deux variables d’un même type T

**class Couple<T> {**

**private T x, y ; // les 2 elements du couple seront du même type**

**public Couple (T premier, T second){**

**x = premier ; y = second ;**

**}**

**public T getPremier () {**

**return x ;**

**}**

**public void affiche () {**

**System.out.println ("Couple : 1ere val : " + x + " - 2e val : " + y ) ;**

**}}**

* T peut être remplacé par un type quelconque

## Utilisation de la classe générique

public class Couple1

{

public static void main (String args[])

{

**Integer oi1 = new Integer (3) ;**

**Integer oi2 = new Integer (5) ;**

**Couple <Integer> ci = new Couple (oi1, oi2) ;**

**ci.affiche () ;**

**Couple <Double> cd = new Couple <Double> (2.0, 12.0) ;**

**cd.affiche() ;**

**Double p = cd.getPremier () ;**

**System.out.println ("premier element du couple cd = " + p ) ;**

**EXECUTION :**

**Couple : 1ere val : 3 - 2e val : 5**

**Couple : 1ere val : 2.0 - 2e val : 12.0**

**premier element du couple cd = 2.0**

* ATTENTION : T doit être une classe (ex : Integer et non int)

## 

## Définition d’une classe générique à deux types

package generique;

class **CoupleDiff <T, U>** {

**private T x; // le premier element du couple**

**private U y; // le second element du couple**

**public CoupleDiff(T premier, U second) {**

**x = premier;**

**y = second;**

**}**

**public T getPremier() {**

**return x;**

**}**

**public void affiche() {**

**System.out.println ("Couple diff 1ere val : " + x + " - 2e val : " + y ) ;**

**}**

**public static void main(String args[]) {**

**Integer oi1 = 3;**

**Double od1 = 2.5;**

**CoupleDiff <Integer, Double> ch1 = new CoupleDiff<Integer, Double>(oi1,**

**od1);**

**ch1.affiche();**

**Integer oi2 = 4;**

**CoupleDiff <Integer, Integer> ch2 = new CoupleDiff<Integer, Integer>(oi1, oi2);**

ch2.affiche();

Integer n = ch1.getPremier();

System.out.println("premier element du couple ch1 = " + n);

}

}

EXECUTION :

**Couple diff 1ere val : 3 - 2e val : 2.5**

**Couple diff 1ere val : 3 - 2e val : 4**

**premier element du couple ch1 = 3**

## 

## “Effacement” du paramètre de type à la compilation et limitations qui en découlent

* Comment le compilateur gère les paramètres de type ?
  + Le type brut (après compilation) d’une classe générique ne mémorise pas le type T qui est remplacé par Object (on parle *d’effacement* du type)
  + Seul le type brut est connu lors de l’exécution
* Cela entraîne plusieurs limitations :
  + Interdit d’instancier un type paramétré : pas de new T()
    - mais on a bien sûr le droit d’appeler le constructeur de la classe générique (ce n’est pas la même chose)
      * new Couple <Double> (...)
  + Interdit de définir une classe générique dérivée de Throwable
  + Un attribut static sera le même pour toutes les instances de la classe, quelque soit la valeur du paramètre de type (Integer, Double, …)
  + Un attribut static ne peut pas être d’un type paramétré
    - pas possible : static T compte
  + Interdit d’allouer un tableau d’un type paramétré
    - pas possible : new T[10]

# 

# Méthodes génériques

* Une méthode générique est une méthode qui retourne un type quelconque ou qui prend en paramètre un type paramétré qui est déclaré avant d’être utilisé
  + cela peut être une méthode static, une méthode d’objet,   
    ou un constructeur de classe générique

public class **Pair**<K, V> {  
  
 private K key;  
 private V value;  
  
 public Pair(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
  
 public void setKey(K key) { this.key = key; }  
 public void setValue(V value) { this.value = value; }  
 public K getKey() { return key; }  
 public V getValue() { return value; }  
}

package generique;

public class **PairMain** {

// Comparer deux paires clés - valeur (ayant chacune un type générique)

public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

public static void main(String args[]) {

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = PairMain.<Integer, String>compare(p1, p2);

// On peut aussi ne pas mettre le type qui sera inféré par le compilateur :

Pair<Integer, String> p3 = new Pair<>(1, "apple");

Pair<Integer, String> p4 = new Pair<>(2, "pear");

boolean same2 = PairMain.compare(p3, p4);

}

}

EXECUTION :

# 

# Héritage et programmation générique

* On peut limiter le type paramétré à des sous-classes d’une classe particulière

static <T extends Number> T hasard (T [] valeurs) {

// instructions

}

* On peut créer une classe dérivée d’une classe générique et ce de différentes manières

Par exemple, soit la classe

class C <T> { …}

On peut définir :

* une sous-classe qui conserve les paramètres de type de la classe de base, sans en ajouter d’autres

class D <T> extends C<T> { …}

D<String> dérive de C<String>

D<Double> dérive de C<Double>

* une sous-classe qui conserve les paramètres de type de la classe de base, et qui en ajoute d’autres

class D <T, U> extends C<T> { …}

D<String, Double> dérive de C<String>

* une sous-classe qui introduit des limitations sur un ou plusieurs des paramètres de type de la classe de base

class D <T extends Number> extends C<T> { …}

=> on peut utiliser D<Double> (qui dérive de C<Double>)

=> on ne peut pas utiliser D<String> (alors qu’on peut utiliser C<String>)

# 

# Références

* Claude Delannoy. Programmer en Java. Editions Eyrolles.
* Tutorial Oracle <http://docs.oracle.com/javase/tutorial/java/generics/index.html>