Introduction à la Programmation Java : généricité, le polymorphisme paramètrique

Frédéric Gava

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours de L3 MIAGE

Plan

Plan

Déroulement du cours

Introduction

Schéma général

La généricité (qui existe depuis les années 70 dans d'autres langages notamment OCaml) en Java est un ajout réalisé sur un langage existant depuis déjà une quinzaine d'années, avec des contraintes de compatibilité ascendante.

On doit aborder la question à partir de deux grands principes : les règles de typage ne sont pas changées, il y a juste de nouveaux types avec de nouvelles règles pour définir la relation de sous-typage. La généricité n'existe pas dans le code compilé, au niveau de la JVM. Cela entraîne quelques restrictions dans son usage.

Intérêt

Avec le polymorphisme objet, on pouvait notamment mettre dans un conteneur différents objets de type différents s'il avait un type père (commun). Mais il était difficile d'exprimer le fait que l'implantation du conteneur ne dépendait pas des objets contenus. Pour cela, il fallait passer soit par un "objet abstrait" (avec les méthodes qui sont utilisées) soit directement par le type Object. La générécité va remédier à ce defaut afin d'exprimer plus concisement ce genre de problèmes (on parle d'expressivité).

Introduction

Schéma général

La généricité (qui existe depuis les années 70 dans d'autres langages notamment OCaml) en Java est un ajout réalisé sur un langage existant depuis déjà une quinzaine d'années, avec des contraintes de compatibilité ascendante. On doit aborder la question à partir de deux grands principes : les règles de typage ne sont pas changées, il y a juste de nouveaux types avec de nouvelles règles pour définir la relation de sous-typage. La généricité n'existe pas dans le code compilé, au niveau de la JVM. Cela entraîne quelques restrictions dans son usage.

Intérêt

Avec le polymorphisme objet, on pouvait notamment mettre dans un conteneur différents objets de type différents s'il avait un type père (commun). Mais il était difficile d'exprimer le fait que l'implantation du conteneur ne dépendait pas des objets contenus. Pour cela, il fallait passer soit par un "objet abstrait" (avec les méthodes qui sont utilisées) soit directement par le type Object. La générécité va remédier à ce defaut afin d'exprimer plus concisement ce genre de problèmes (on parle d'expressivité).

Introduction

Un nouveau type

Voyons un exemple de classe générique. Il s'agit d'une classe Paire qui modélise une paire de deux éléments du même type.

```
Class Paire<T> {
 private T x, y;
 Paire(T x1, T y1) { x=x1; y=y1; }
 T first(){ return x;}
 T second(){return y;}
 void setFirst(T x1) { x=x1; }
 void setSecond(T y1) { y=y1; }
 boolean equals(Paire<T> o) { return o.x.equals(x) && o.y.equals(y); } }
```

Utilisation

```
Paire<String> p = new Paire<String>("Gava", "Frédéric");
```

On voit dans cet exemple apparaître de nouveaux types par rapport à ceux qu'on avait jusqu'ici. Ici, nous avons :

- des variables de type (ex : T, utilisé dans la classe Paire)
- des classes génériques non instanciées (ex : Paire<T> utilisé dans la classe, pour equals). Les paramètres des classes sont encore des variables.
- des classes génériques instanciées (ex : Paire<String> utilisé pour les instances de la classe). Les paramètres des classes sont remplacés par des types.

Il faut savoir comment ces nouveaux types se raccordent aux anciens d'une part, entre eux d'autre part, en ce qui concerne l'héritage et la relation de sous-typage.

Introduction

Un nouveau type

Voyons un exemple de classe générique. Il s'agit d'une classe Paire qui modélise une paire de deux éléments du même type.

```
Class Paire<T> {
 private T x, y;
 Paire(T x1, T y1) { x=x1; y=y1; }
 T first(){ return x;}
 T second(){return y;}
 void setFirst(T x1) { x=x1; }
 void setSecond(T y1) { y=y1; }
 boolean equals(Paire<T> o) { return o.x.equals(x) && o.y.equals(y); } }
```

Utilisation

```
Paire<String> p = new Paire<String>("Gava", "Frédéric");
```

On voit dans cet exemple apparaître de nouveaux types par rapport à ceux qu'on avait jusqu'ici. Ici, nous avons :

- des variables de type (ex : T, utilisé dans la classe Paire)
- des classes génériques non instanciées (ex : Paire<T> utilisé dans la classe, pour equals). Les paramètres des classes sont encore des variables.
- des classes génériques instanciées (ex : Paire<String> utilisé pour les instances de la classe). Les paramètres des classes sont remplacés par des types.

Il faut savoir comment ces nouveaux types se raccordent aux anciens d'une part, entre eux d'autre part, en ce qui concerne l'héritage et la relation de sous-typage.

Introduction

Nouvelle forme d'héritage

L'héritage en classe générique est toujours possible. Par exemple, une classe des triplet pourrait étendre celle des paires : **class** Triplet<T> **extends** Paire<T> ...
Java protège avec une analyse statique, du certaine forme de sous-typage même si c'est parfois frustant. Exemple :

```
Paire<Integer> unePaire = new Paire<Integer>(12,45);
Paire<Object> laMeme = unePaire; // erreur à la compilation même si Integer hérite d'Object
unePaire.setFirst("archi");
// cas pathologique qui empêche la généralisation du sous-typage en cas de généricité
Paire<T> p = new Paire<String>("arty","olm"); //erreur à la compilation
```

Mais pas complêtement impossible !

```
// sous-typage possible
T val;
Paire<T> unePaire=new Triplet<T>(val,val,val);
Paire<String> unePaire=new Triplet<String>("un","deux","trois"); //idem

// transtypage (cast) possible
unePaire.equals( (Paire<String>) new Triplet<String>("u","deu","troi"));
```

Introduction

Nouvelle forme d'héritage

L'héritage en classe générique est toujours possible. Par exemple, une classe des triplet pourrait étendre celle des paires : **class** Triplet<T> **extends** Paire<T> ...
Java protège avec une analyse statique, du certaine forme de sous-typage même si c'est parfois frustant. Exemple :

```
Paire<Integer> unePaire = new Paire<Integer>(12,45);
Paire<Object> laMeme = unePaire; // erreur à la compilation même si Integer hérite d'Object
unePaire.setFirst("archi");
// cas pathologique qui empêche la généralisation du sous-typage en cas de généricité
Paire<T> p = new Paire<String>("arty","olm"); //erreur à la compilation
```

Mais pas complêtement impossible !

```
// sous-typage possible
T val;
Paire<T> unePaire=new Triplet<T>(val,val,val);
Paire<String> unePaire=new Triplet<String>("un","deux","trois"); //idem

// transtypage (cast) possible
unePaire.equals( (Paire<String>) new Triplet<String>("u","deu","troi"));
```

Introduction

Restriction des constructions dynamiques

La JVM n'a pas été modifiée par la généricité : une classe paramétrée est compilée en une classe non paramétrée.

Les variables de types sont oubliées après la compilation. Certaines constructions qui utilisent les types à l'exécution (comme certains transtypages) ne sont donc pas autorisées avec des types paramétrés.

1- la création d'objet

Un code du genre **new** T() n'est pas permis.

2- un transtypage vers une sous-classe

Exemple :

```
// on suppose des classes CompteRemunere et CompteSecurise qui héritent de Compte
Compte c = new CompteRemunere();
CompteSecurise c2 = (CompteSecurise) c;
// Exception java.lang.ClassCastException à l'exécution
```

Une vérification de type est faite à l'éxécution pour vérifier que le type d'instance est bien sous-type du type imposé par le transtypage. Comme les types paramétrés n'existent pas à l'exécution, on ne peut pas faire de transtypage vers une sous-classe. En revanche, dans l'autre sens, de la sous-classe vers la super-classe (classe mère), le contrôle se fait à la compilation, il n'a pas de problème. Exemple :

```
// erreur à la compilation
class ConstructionDynamique<T> { Triplet<T> convert(Object o) { return (Triplet<T>) o; } }
```

Introduction

Restriction des constructions dynamiques

La JVM n'a pas été modifiée par la généricité : une classe paramétrée est compilée en une classe non paramétrée.

Les variables de types sont oubliées après la compilation. Certaines constructions qui utilisent les types à l'exécution

(comme certains transtypages) ne sont donc pas autorisées avec des types paramétrés.

1- la création d'objet

Un code du genre **new** T() n'est pas permis.

2- un transtypage vers une sous-classe

Exemple :

```
// on suppose des classes CompteRemunere et CompteSecurise qui héritent de Compte
Compte c = new CompteRemunere();
CompteSecurise c2 = (CompteSecurise) c;
// Exception java.lang.ClassCastException à l'exécution
```

Une vérification de type est faite à l'éxécution pour vérifier que le type d'instance est bien sous-type du type imposé par le transtypage. Comme les types paramétrés n'existent pas à l'exécution, on ne peut pas faire de transtypage vers une sous-classe. En revanche, dans l'autre sens, de la sous-classe vers la super-classe (classe mère), le contrôle se fait à la compilation, il n'a pas de problème. Exemple :

```
// erreur à la compilation
class ConstructionDynamique<T> { Triplet<T> convert(Object o) { return (Triplet<T>) o; } }
```

Introduction

Restriction des constructions dynamiques

La JVM n'a pas été modifiée par la généricité : une classe paramétrée est compilée en une classe non paramétrée.

Les variables de types sont oubliées après la compilation. Certaines constructions qui utilisent les types à l'exécution

(comme certains transtypages) ne sont donc pas autorisées avec des types paramétrés.

1- la création d'objet

Un code du genre **new** T() n'est pas permis.

2- un transtypage vers une sous-classe

Exemple :
// on suppose des classes CompteRemunere et CompteSecurise qui héritent de Compte
Compte c = **new** CompteRemunere();
CompteSecurise c2 = (CompteSecurise) c;
// Exception java.lang.ClassCastException à l'exécution

Une vérification de type est faite à l'éxécution pour vérifier que le type d'instance est bien sous-type du type imposé par le transtypage. Comme les types paramétrés n'existent pas à l'exécution, on ne peut pas faire de transtypage vers une sous-classe. En revanche, dans l'autre sens, de la sous-classe vers la super-classe (classe mère), le contrôle se fait à la compilation, il n'a pas de problème. Exemple :
// erreur à la compilation
class ConstructionDynamique<T> { Triplet<T> convert(Object o) { **return** (Triplet<T>) o; } }

Introduction

3- Vérification dynamique

La vérification de type d'instance dynamique au moyen de l'opérateur **instanceof** :

```
//erreur à la compilation
class ConstructionDynamique {
 boolean estPaireInteger(Object o) { return o instanceof Paire<Integer>; } }
```

Traduction

D'ordinaire

La généricité a pour seul effet de changer des types pour les variables et méthodes d'une classe. Ces types sont connus une fois la substitution des paramètres effectuée, c'est à dire dans le code qui utilise la classe paramétrée.
Prenons comme exemple le code généré les paires :

```
class Paire {
 Object x, y;
 Paire(Object x1, Object x2) { x=x1;y=y2;}
 Object first() { return x; }
 boolean equals(Paire o) { return o.x.equal(x) && o.y.equals(y) } }

 Paire p = new Paire("Gava","Frédéric");
 System.out.println((String) p.x); // au lieu de p.x !!!
```

Cas particuliers

Dans certains cas, liés à l'héritage et à la redéfinition, la traduction est un iota plus complexe et fait intervenir une "méthode-pont" qui transtype les paramètres :

```
class Compte { int solde; }
interface Test<T> { public boolean propriete (T x); }
class DansLeRouge implements Test<Compte> {
public boolean propriete(Compte c){return c.solde<0; } }
donne
interface Test { public boolean propriete (Object x); }
class DansLeRouge implements Test {
 public boolean propriete (Compte c) { return c.solde<0; }
 public boolean propriete (Object x) {return this.propriete((Compte) x);} }
```

Traduction

D'ordinaire

La généricité a pour seul effet de changer des types pour les variables et méthodes d'une classe. Ces types sont connus une fois la substitution des paramètres effectuée, c'est à dire dans le code qui utilise la classe paramétrée.
Prenons comme exemple le code généré les paires :

```
class Paire {
 Object x, y;
 Paire(Object x1, Object x2) { x=x1;y=y2;}
 Object first() { return x; }
 boolean equals(Paire o) { return o.x.equal(x) && o.y.equals(y) } }

 Paire p = new Paire("Gava","Frédéric");
 System.out.println((String) p.x); // au lieu de p.x !!!
```

Cas particuliers

Dans certains cas, liés à l'héritage et à la redéfinition, la traduction est un iota plus complexe et fait intervenir une "méthode-pont" qui transtype les paramètres :

```
class Compte { int solde; }
interface Test<T> { public boolean propriete (T x); }
class DansLeRouge implements Test<Compte> {
public boolean propriete(Compte c){return c.solde<0; } }
```

donne

```
interface Test { public boolean propriete (Object x); }
class DansLeRouge implements Test {
 public boolean propriete (Compte c) { return c.solde<0; }
 public boolean propriete (Object x) {return this.propriete((Compte) x);} }
```

Traduction

D'ordinaire

La généricité a pour seul effet de changer des types pour les variables et méthodes d'une classe. Ces types sont connus une fois la substitution des paramètres effectuée, c'est à dire dans le code qui utilise la classe paramétrée.
Prenons comme exemple le code généré les paires :

```
class Paire {
 Object x, y;
 Paire(Object x1, Object x2) { x=x1;y=y2;}
 Object first() { return x; }
 boolean equals(Paire o) { return o.x.equal(x) && o.y.equals(y) } }

 Paire p = new Paire("Gava","Frédéric");
 System.out.println((String) p.x); // au lieu de p.x !!!
```

Cas particuliers

Dans certains cas, liés à l'héritage et à la redéfinition, la traduction est un iota plus complexe et fait intervenir une "méthode-pont" qui transtype les paramètres :

```
class Compte { int solde; }
interface Test<T> { public boolean propriete (T x); }
class DansLeRouge implements Test<Compte> {
public boolean propriete(Compte c){return c.solde<0; } }
```

donne

```
interface Test { public boolean propriete (Object x); }
class DansLeRouge implements Test {
 public boolean propriete (Compte c) { return c.solde<0; }
 public boolean propriete (Object x) {return this.propriete((Compte) x);} }
```

Déroulement du cours

Exemple sans généricité

Jurassic Park

```
Class Cell {
 private Object val;
 Cell (Object o) { val=o;}
 void set(Object o) { val=o;}
 Object get(){return val;}
 }
```

Et pour l'utiliser :

```
Cell c=new Cell(new Compte("Martin"));
((Compte) c.get()).depotEuro(50);
```

La famille Pierreafeu

```
import java.util.Vector;
    Vector v=new Vector(12);
    v.add(new Compte("Martin"));
    v.add(new Integer(5));
    System.out.println(v.toString); //erreur à l'exécution
    ((Compte) v.elementAt(0)).depotEuro(50);
    ((Compte) v.elementAt(1)).depotEuro(100); //erreur à l'exécution
```

Exemple sans généricité

Jurassic Park

```
Class Cell {
 private Object val;
 Cell (Object o) { val=o;}
 void set(Object o) { val=o;}
 Object get(){return val;}
 }
```

Et pour l'utiliser :

```
Cell c=new Cell(new Compte("Martin"));
((Compte) c.get()).depotEuro(50);
```

La famille Pierreafeu

```
import java.util.Vector;
    Vector v=new Vector(12);
    v.add(new Compte("Martin"));
    v.add(new Integer(5));
    System.out.println(v.toString); //erreur à l'exécution
    ((Compte) v.elementAt(0)).depotEuro(50);
    ((Compte) v.elementAt(1)).depotEuro(100); //erreur à l'exécution
```

Exemple avec la généricité

Star-Treck

```
class Cell<T> {
 T val;
 Cell(T x) {val=x;}
 void set(T x){val=x;}
 T get() {return val;}
```

Et pour l'utiliser :

```
Cell<Compte> c = new Cell<Compte>(new Compte("Martin"));
c.get().depotEuro(50);
```

StarWar

```
import java.util.Vector;
Vector<Compte> v = new Vector<Compte>(12);
v.add(new Compte("Martin");
v.add(new Integer(5)); //erreur à la compilation
v.elementAt(0).depotEuro(50);
```

Exemple avec la généricité

Star-Treck

```
class Cell<T> {
 T val;
 Cell(T x) {val=x;}
 void set(T x){val=x;}
 T get() {return val;}
```

Et pour l'utiliser :

```
Cell<Compte> c = new Cell<Compte>(new Compte("Martin"));
c.get().depotEuro(50);
```

StarWar

```
import java.util.Vector;
Vector<Compte> v = new Vector<Compte>(12);
v.add(new Compte("Martin");
v.add(new Integer(5)); //erreur à la compilation
v.elementAt(0).depotEuro(50);
```

Paramètre générique et héritage

Exemple

```
class Cell3<T extends Compte> {
 T val;
 Cell3(T x){val=x;}
 void set(T x){val=x;}
 T get(){return val;}
 void affiche(){val.print();} // on suppose Compte doté d'une méthode print
```

Utilisation

```
Cell3<CompteSecurise> c; // on suppose CompteSecurise extends Compte
c=new Cell3<CompteSecurise>(new CompteSecurise("Martin"));
c.get().depotEuro(50);
c.get().print();
```

Paramètre générique et héritage

Exemple

```
class Cell3<T extends Compte> {
 T val;
 Cell3(T x){val=x;}
 void set(T x){val=x;}
 T get(){return val;}
 void affiche(){val.print();} // on suppose Compte doté d'une méthode print
```

Utilisation

```
Cell3<CompteSecurise> c; // on suppose CompteSecurise extends Compte
c=new Cell3<CompteSecurise>(new CompteSecurise("Martin"));
c.get().depotEuro(50);
c.get().print();
```

Paramètre générique et héritage

Exemple d'héritage de type générique

```
class Cell4<T> {
 T val;
 Cell3(T x){val=x;}
 void set(T x){val=x;}
 T get(){return val;} }

class Couple<X,Y> extends Cell4<X> {
 Y second;
 Couple(X x, Y y) { super(x); second=y;}
 void set2(Y y) {second =y;}
 Y get2() { return second;} }
```

Utilisation

```
Couple<Integer, Boolean> var;
var = new Couple<Integer, Boolean>(new Integer(5), new Boolean(true));
System.out.println(var.get().intValue());
System.out.println(var.get2().booleanValue());
```

Paramètre générique et héritage

Exemple d'héritage de type générique

```
class Cell4<T> {
 T val;
 Cell3(T x){val=x;}
 void set(T x){val=x;}
 T get(){return val;} }

class Couple<X,Y> extends Cell4<X> {
 Y second;
 Couple(X x, Y y) { super(x); second=y;}
 void set2(Y y) {second =y;}
 Y get2() { return second;} }
```

Utilisation

```
Couple<Integer, Boolean> var;
var = new Couple<Integer, Boolean>(new Integer(5), new Boolean(true));
System.out.println(var.get().intValue());
System.out.println(var.get2().booleanValue());
```

Paramètre générique et sous-typage

Exemple

```
Compte c = new CompteSecurise("Martin");
Cell2<Compte> ce = new Cell2<CompteSecurise>(new CompteSecurise("Jean"));
//erreur à la compilation car sous−typage interdit avec généricité !
```

Une WildCard qui ne va pas

```
class QuiVaPas {
 static boolean equals(Cell2<Object> c1, Cell2<Object> c2) {return c1.get() == c2.get();}
 public static void main(String[] args) {
  Cell2<Compte> boite1 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite2 = new Cell2<Compte>(new Compte("Jean"));
  System.out.println(QuiVaPas.equals(boite1,boite2));
  //erreur à la compilation, methode equals(Cell2<Compte>, Cell2<Compte>) unknow
```

Une WildCard

```
class Wildcard {
 static boolean equals(Cell2<?>c1, Cell2<?> c2) { return c1.get() == c2.get();}
 public static void main(String[] args) {
  Cell2<Compte> boite1 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite2 = new Cell2<Compte>(new Compte("Jean"));
  System.out.println(Wildcard.equals(boite1,boite2));
  Cell2<Compte> boite3 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite4 = new Cell2<Integer>(new Integer(4));
  System.out.println(Wildcard.equals(boite3,boite4)); // pas de pb car égalité structurelle...
```

Paramètre générique et sous-typage

Exemple

```
Compte c = new CompteSecurise("Martin");
Cell2<Compte> ce = new Cell2<CompteSecurise>(new CompteSecurise("Jean"));
//erreur à la compilation car sous−typage interdit avec généricité !
```

Une WildCard qui ne va pas

```
class QuiVaPas {
 static boolean equals(Cell2<Object> c1, Cell2<Object> c2) {return c1.get() == c2.get();}
 public static void main(String[] args) {
  Cell2<Compte> boite1 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite2 = new Cell2<Compte>(new Compte("Jean"));
  System.out.println(QuiVaPas.equals(boite1,boite2));
  //erreur à la compilation, methode equals(Cell2<Compte>, Cell2<Compte>) unknow
```

Une WildCard

```
class Wildcard {
 static boolean equals(Cell2<?>c1, Cell2<?> c2) { return c1.get() == c2.get();}
 public static void main(String[] args) {
  Cell2<Compte> boite1 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite2 = new Cell2<Compte>(new Compte("Jean"));
  System.out.println(Wildcard.equals(boite1,boite2));
  Cell2<Compte> boite3 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite4 = new Cell2<Integer>(new Integer(4));
  System.out.println(Wildcard.equals(boite3,boite4)); // pas de pb car égalité structurelle...
```

Paramètre générique et sous-typage

Exemple

```
Compte c = new CompteSecurise("Martin");
Cell2<Compte> ce = new Cell2<CompteSecurise>(new CompteSecurise("Jean"));
//erreur à la compilation car sous-typage interdit avec généricité !
```

Une WildCard qui ne va pas

```
class QuiVaPas {
 static boolean equals(Cell2<Object> c1, Cell2<Object> c2) {return c1.get() == c2.get();}
 public static void main(String[] args) {
  Cell2<Compte> boite1 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite2 = new Cell2<Compte>(new Compte("Jean"));
  System.out.println(QuiVaPas.equals(boite1,boite2));
  //erreur à la compilation, methode equals(Cell2<Compte>, Cell2<Compte>) unknow
```

Une WildCard

```
class Wildcard {
 static boolean equals(Cell2<?>c1, Cell2<?> c2) { return c1.get() == c2.get();}
 public static void main(String[] args) {
  Cell2<Compte> boite1 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite2 = new Cell2<Compte>(new Compte("Jean"));
  System.out.println(Wildcard.equals(boite1,boite2));
  Cell2<Compte> boite3 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite4 = new Cell2<Integer>(new Integer(4));
  System.out.println(Wildcard.equals(boite3,boite4)); // pas de pb car égalité structurelle...
```

Méthode générique

Exemple

```
static void pasBonEchange(Cell2<?>c1, Cell2<?>c2) {
 Cell2<?> tampon = new Cell2<?>(c1.get());
 //erreur à la compilation car Cell2<?> <> Cell2<T>
 c1.set(c2.get());
 c2.set(tampon.get()); }

static <T> void bonEchange(Cell2<T> c1, Cell2<T> c2) {
 Cell2<T> tampon = new Cell2<T>(c1.get());
 c1.set(c2.get());
 c2.set(tampon.get()); }

  Cell2<Compte> boite1 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite2 = new Cell2<Compte>(new Compte("Jean"));
  bonEchange(boite1, boite2);

  Cell2<Compte> boite3 = new Cell2<Compte>(new Compte("Martin"));
  Cell2<Compte> boite4 = new Cell2<Integer>(new Integer(4));
  bonEchange(boite3,boite4);
  //erreur à la compilation car Integer<>Compte
```

Généricité et interfaces

Exemple d'une double extentions

```
interface faiseur { void fait(); }
interface mangeur { void mange(); }
class Cell1<T extends faiseur> {
 T val;
 void set(T x){val=x;}
 void T get(){return val;} }

class Cell2<T extends faiseur & mangeur>{
 T val;
 void set(T x){val=x;}
 void T get(){return val;} }
```

Exemple d'une double implémentation

```
class QuiVa implements faiseur, mangeur {
 public void fait() {}
 public void mange() {} }

class Exemple2{
 public static void main(String[] argv) {
  QuiVa qv = new QuiVa();
  Cell1<QuiVa> c1 = new Cell1<QuiVa>();
  Cell2<QuiVa> c2 = new Cell2<QuiVa>();
  c1.set(qv); c2.set(qv);
  c1.get().fait(); c2.get().fait();
  c2.get().mange(); }}
```

Généricité et interfaces

Exemple d'une double extentions

```
interface faiseur { void fait(); }
interface mangeur { void mange(); }
class Cell1<T extends faiseur> {
 T val;
 void set(T x){val=x;}
 void T get(){return val;} }

class Cell2<T extends faiseur & mangeur>{
 T val;
 void set(T x){val=x;}
 void T get(){return val;} }
```

Exemple d'une double implémentation

```
class QuiVa implements faiseur, mangeur {
 public void fait() {}
 public void mange() {} }

class Exemple2{
 public static void main(String[] argv) {
  QuiVa qv = new QuiVa();
  Cell1<QuiVa> c1 = new Cell1<QuiVa>();
  Cell2<QuiVa> c2 = new Cell2<QuiVa>();
  c1.set(qv); c2.set(qv);
  c1.get().fait(); c2.get().fait();
  c2.get().mange(); }}
```

Généricité et interfaces

Exemple d'un double paramétrage

```
interface DomaineACle<T> {
 T getCle();
 void print(); }

class SddGen <Y, X extends DomaineACle<Y>> {
 X[] tab;
 int nb=0;
 SddGen(X[] t) { tab = t;}
 void ajouter(X o) { tab[nb]=o; nb++; }
 X retrouver(Y cle){
  for (int i=0; i<nb; i++) if (tab[i].getCle().equals(cle)) return tab[i];
  return null; }
 int nombreElem() { return nb; }
 void voir() { for (int i=0; i<nb; i++) tab[i].print(); }
 }
```

Généricité et interfaces

Exemple d'une class abstraite

```
abstract class Animal implements DomaineACle<Integer> {
 String nom;
 Integer tatouage;
 String cri;
 Animal (String st, Integer tat) { nom=st;tatouage = tat;}
 public Integer getCle() { return tatouage; }
 public void print() { System.out.println(nom+" "+tatouage);
 boolean repond(String n){return n==nom;}
 void crie() { System.out.print(cri); }
 abstract boolean court_t_y();
 abstract boolean volde_t_y();
 abstract boolean nage_t_y(); }
```

Généricité et interfaces

Exemple d'un premier héritage

```
class Chat extends Animal {
 Chien(String nom, Integer tatouage) {
 super(nom, tatouage);
 cri="miaooouuuuuu"; }
boolean court_t_y() { return true; }
boolean vole_t_y() { return true; }
boolean nage_t_y() { return true; }
boolean mord_t_y() { return true; }
}
```

Exemple d'un autre héritage

```
class Corbeau exends Animal {
 Corbeau(String nom, Integer tatouage) { super(nom,tatouage); cri="croa-croa"};
 boolean court_t_y() { return true; }
 boolean vole_t_y() { return true; }
 boolean nage_t_y() { return false; }
 void chante(int nb) { for (int i=0; i<nb; i++) { this.crie(); }
 }
```

Généricité et interfaces

Exemple d'un premier héritage

```
class Chat extends Animal {
 Chien(String nom, Integer tatouage) {
 super(nom, tatouage);
 cri="miaooouuuuuu"; }
boolean court_t_y() { return true; }
boolean vole_t_y() { return true; }
boolean nage_t_y() { return true; }
boolean mord_t_y() { return true; }
}
```

Exemple d'un autre héritage

```
class Corbeau exends Animal {
 Corbeau(String nom, Integer tatouage) { super(nom,tatouage); cri="croa-croa"};
 boolean court_t_y() { return true; }
 boolean vole_t_y() { return true; }
 boolean nage_t_y() { return false; }
 void chante(int nb) { for (int i=0; i<nb; i++) { this.crie(); }
 }
```

Généricité et interfaces

Utilisation

```
Chat Lulu = new Chat("Lulu", new Integer(17));
SddGen<Integer, Animal> menagerie = new Sdd<Integer, Animal>(new Animal[50]);
menagerie.ajouter(Lulu);
for (int i=0; i<30; i++) {
 Corbeau piaf = new Corbeau ("corbeau"+i,new Integer(20+i));
 menagerie.ajouter(piaf); }
menagerie.voir();
Animal animal25= menagerie.retrouver(new Integer(25));
animal25.crie();
}
```

Bonnes révisions !