La généricité en Java

Jean-Marie Lagniez

LPDIOC : Algorithmique et Programmation
IUT Lens

Département Informatique

Table des matières

Sommaire

Dans ce cours nous allons voir :

- Classes abstraites
- Interfaces
- Templates

Pré-requis :

- Héritage, polymorphisme et création d'objets

Introduction

- Grâce aux cours dispensés pendant la première partie de ce module, vous avez pu vous rendre compte que vos programmes peuvent être **très complexes** et peuvent faire intervenir de **nombreuses classes** qui dépendent les unes des autres

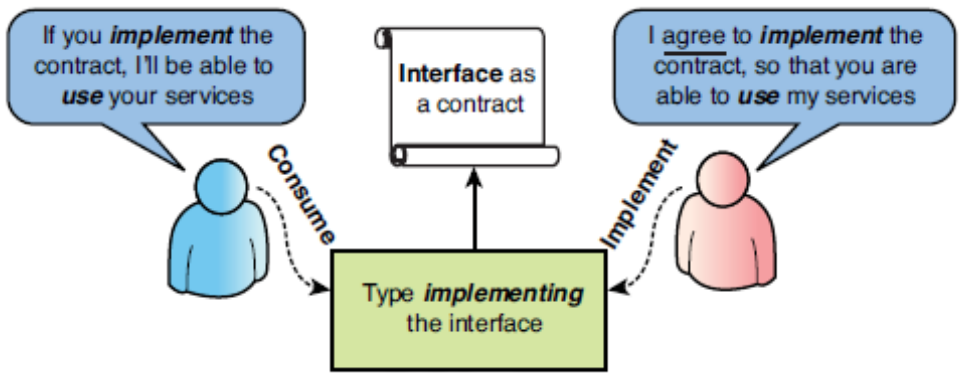
- Afin de bien structurer vos programmes, vous allez devoir faire attention à l'endroit où vous allez définir les méthodes des objets : classe mère ? classe fille ?

- Pour réaliser cela, nous allons utiliser les notions de **classes abstraites**, **d'interfaces** et de **templates**

Table des matières

À quoi cela sert ?

- L'un des atouts majeurs de la programmation orientée objet est la possibilité de **réutiliser** du codes
- Ainsi, on aimerait pouvoir utiliser un objet que nous avons déjà créé pour une nouvelle application
- De plus, on aimerait pouvoir réutiliser ce code **sans savoir comment ce dernier a été conçu**



- L'idée est de convenir d'un contrat de ce que devra implémenter la classe qui utilisera les fonctionalités d'une autre classe : **interface**

Les interfaces en Java

- En pratique **une interface est assez similaire à une classe**
- Seulement, une interface ne peut contenir que :
  - des constantes
  - des signatures de fonctions
  - des méthodes par défaut
  - des méthodes statiques
  - des classes internes
- Le corps d'une méthode est définie uniquement pour les méthodes statiques et par défaut
- Les interfaces ne peuvent pas être instanciées !
- Elles ne peuvent qu'être implémentées ou étendues par d'autres classes

Déclaration

- La déclaration d'une interface est similaire à la déclaration d'une classe
- La différence est qu'on utilise le mot clef **interface**

```
public interface RoulerInterface{
  void rouler();
  void freiner();
}

public interface FairePleinInterface{
  void fairePlein();
}
```

- Notez bien qu'il n'y a **pas d'accolades** après la déclaration de la méthode et qu'on a directement un **point virgule**
- Le mot clef `public` spécifie que l'interface peut être utilisée par l'ensemble des classes de n'importe quel `package`
- Une interface peut déclarer des constantes. Toutes les constantes sont alors implicitement `public`, `static` et `final` et ne **peuvent donc pas être modifiées**

Utilisation

- Pour implémenter une interface il suffit d'utiliser le mot clef **implements**

```
public class Velo implements RoulerInterface{
  private String marque;
  public Velo(String marque){this.marque = marque;}
  void rouler(){System.out.println("Pédaler");}
  void freiner(){
    System.out.println("Arrêter de pédaler");
    System.out.println("Appuyer sur le frein");
  }
}
```

- Par convention, le mot clef **implements** suit le mot clef **extends** si ce dernier est présent

Implémentation de plusieurs interfaces

- Une classe peut implémenter plusieurs interfaces, donc le mot clef **implements** peut être suivi d'une liste d'interfaces séparées par des virgules

```
public class Auto implements RoulerInterface,
    FairePleinInterface{
  private String marque;

  public Auto(String marque){this.marque = marque;}
  void rouler(){System.out.println("Accélérer");}
  void freiner(){
    System.out.println("Arrêter d'accélérer");
    System.out.println("Appuyer sur le frein");
  }

  void fairePlein(){
    System.out.println("Aller à la pompe");
    System.out.println("Mettre de l'essence");
    System.out.println("Payer :D");
  }
}
```

Exercice : trier des figures géométriques (I)

- Considérons la situation où l'on souhaite programmer un logiciel dont le but est de manipuler des objets rectangulaires et que l'on souhaite pouvoir trier ces objets en fonction de l'aire de ces derniers
- Supposons encore que vous travaillez en binôme (Bob et vous) et que vous vous séparez le travail tel que vous implémentez la classe permettant de stocker et trier des éléments, et Bob la classe `Rectangle`
- Maintenant, supposons qu'Alice vous demande aussi de travailler avec elle, mais qu'elle souhaite trier des cercles
- Comment faire pour minimiser le travail que vous aller avoir à réaliser ?

Exercice : trier des figures géométriques (I)

- Considérons la situation où l'on souhaite programmer un logiciel dont le but est de manipuler des objets rectangulaires et que l'on souhaite pouvoir trier ces objets en fonction de l'aire de ces derniers
- Supposons encore que vous travaillez en binôme (Bob et vous) et que vous vous séparez le travail tel que vous implémentez la classe permettant de stocker et trier des éléments, et Bob la classe `Rectangle`
- Maintenant, supposons qu'Alice vous demande aussi de travailler avec elle, mais qu'elle souhaite trier des cercles
- Comment faire pour minimiser le travail que vous aller avoir à réaliser ?
- Leur imposer l'implémentation d'une interface !

Exercice : trier des figures géométriques (II)

```
public interface CompareFigure{
  int estPlusGrandQue(CompareFigure c);
}

public class Rectangle implements CompareFigure{
  private int l, L;

  public Rectangle(int l, int L){
    this.l = l;
    this.L = L;
  }

  public int getl(){return l;};
  public int getL(){return L;};
  public String toString(){return Integer.toString(l * L);}

  public int estPlusGrand(CompareFigure other){
    Rectangle r = (Rectangle) other;

    if((l * L) > (r.getl() * r.getL())) return 1;
    if((l * L) < (r.getl() * r.getL())) return -1;
    return 0;
  }
}
```

Exercice : trier des figures géométriques (III)

```
public class EnsembleFigures{
  private int maxFigure, sizeTab;
  private CompareFigure tabFigure[];

  public EnsembleFigures(int maxFigure){
    this.maxFigure = maxFigure;
    sizeTab = 0;
    tabFigure = new CompareFigure[maxFigure];
  }

  public void add(CompareFigure c){
    if(sizeTab < maxFigure) tabFigure[sizeTab++] = c;
  }

  public void print(){
    for(int i = 0 ; i<sizeTab ; i++)
        System.out.println(tabFigure[i]);
  }

  public void trieFigure(){
    for(int i = 0 ; i<sizeTab ; i++)
    for(int j = i + 1 ; j<sizeTab ; j++)
    if(tabFigure[i].estPlusGrand(tabFigure[j]) > 0){
      CompareFigure tmp = tabFigure[i];
      tabFigure[i] = tabFigure[j];
      tabFigure[j] = tmp;
    }
  }
}
```

Exercice : trier des figures géométriques (IV)

```
class Test
{
  public static void main(String[] args){
    EnsembleFigures e = new EnsembleFigures(10);
    e.add(new Rectangle(3,4));
    e.add(new Rectangle(1,1));
    e.add(new Rectangle(4,4));
    e.trieFigure();
    e.print();
  }
}
```

Dans un terminal :

```
> javac *.java
> java Test
1
12
16
```

Utiliser une interface comme un type

- Lorsque vous définissez une interface, vous définissez un **nouveau type de donnée**
- Vous pouvez donc utiliser le nom d'une **interface comme** si c'était un **objet quelconque**. Ainsi, si vous définissez une référence sur un type qui est une interface alors l'objet incriminé doit implémenter cette interface
- Par exemple, supposons que l'on souhaite obtenir le plus grand de deux objets qui implémente l'interface `CompareFigure`

```
public Object lePlusGrand(Object a, Object b){
  CompareFigure fa = (CompareFigure) a;
  CompareFigure fb = (CompareFigure) b;
  if(fb.estPlusGrand(fb) > 0) return a;
  else return b;
}
```

Faire évoluer une interface (I)

- Supposons à présent que vous souhaitez modifier pour comparer vos figures selon un second critère

```
public interface CompareFigure{
  int estPlusGrandQue(CompareFigure c);
  int estPlusGrandBis(CompareFigure c);
}
```

- Si vous faite cela vous devrez modifier toutes les classes qui implémente cette interface !
- Si vous voulez ajouter d'autres méthodes à une interface il y a plusieurs approches possibles :
  - utiliser l'héritage
  - utiliser les méthodes avec définition par défaut ou statique

Faire évoluer une interface (II)

- Utiliser l'héritage :

```
public interface CompareFigureBis extends CompareFigure{
  int estPlusGrandBis(CompareFigure c);
}
```

- Méthodes avec définition par défaut :

```
public interface CompareFigure{
  int estPlusGrandQue(CompareFigure c);
  default int estPlusGrandBis(CompareFigure c){
    return 0;
  }
}
```

- Vous pouvez aussi utiliser les méthodes statiques pour faire évoluer vos interfaces

Table des matières

Définition

- Une classe abstraite est pratiquement identique à une classe normale ou a une interface
- Comme pour les interfaces vous ne pouvez pas l'instancier !
- Ainsi, si nous supposons que la classe `A` est abstraite, le code suivant ne compilera pas

```
public class Test{
    public static void main(String[] args){
        A obj = new A(); //Erreur de compilation !
    }
}
```

- Le mécanisme des classes abstraites permet de définir des comportements (méthodes) qui devront être implémentés dans les classes filles, mais **sans implémenter ces comportements**
- Ainsi, on a l'assurance que les classes filles **respecteront le contrat** défini par la classe mère abstraite

Classes abstraites vs. Interfaces

- Les classes abstraites sont assez similaires aux interfaces
- Vous ne pouvez les instancier et elles contiennent à la fois des définitions de méthodes et des méthodes avec leur contenu
- Cependant, avec les classes abstraites vous pouvez déclarer des attributs ou méthodes qui ne sont pas static/final, et définir des méthodes qui sont `public`, `protected` ou `private`
- De plus, vous ne pouvez étendre qu'une seule classe abstraite tandis que vous pouvez implémenter plusieurs interfaces

Exemple de cas d'utilisation

- Considésons la situation où l'on souhaite programmer un logiciel dont le but est de manipuler des objets géométriques
- Supposons encore que l'on souhaite pouvoir afficher à l'écran ces formes géométriques : nous devons donc avoir une méthode `dessiner` implémentée dans chaque objet géométrique
- **Peut-on s'en sortir en utilisant uniquement la notion d'héritage ?**

Exemple de cas d'utilisation

- Considésons la situation où l'on souhaite programmer un logiciel dont le but est de manipuler des objets géométriques
- Supposons encore que l'on souhaite pouvoir afficher à l'écran ces formes géométriques : nous devons donc avoir une méthode `dessiner` implémentée dans chaque objet géométrique
- **Peut-on s'en sortir en utilisant uniquement la notion d'héritage ?**
  - On peut par exemple définir une classe mère `Figure` qui implémente une méthode `dessiner` et les figures géométriques seront des classes filles de cette classe

Exemple de cas d'utilisation

- Considésons la situation où l'on souhaite programmer un logiciel dont le but est de manipuler des objets géométriques
- Supposons encore que l'on souhaite pouvoir afficher à l'écran ces formes géométriques : nous devons donc avoir une méthode `dessiner` implémentée dans chaque objet géométrique
- **Peut-on s'en sortir en utilisant uniquement la notion d'héritage ?**
  - On peut par exemple définir une classe mère `Figure` qui implémente une méthode `dessiner` et les figures géométriques seront des classes filles de cette classe
  - Comment implémenter la méthode `dessiner` dans la classe `Figure` !

Exemple de cas d'utilisation

- Considésons la situation où l'on souhaite programmer un logiciel dont le but est de manipuler des objets géométriques
- Supposons encore que l'on souhaite pouvoir afficher à l'écran ces formes géométriques : nous devons donc avoir une méthode `dessiner` implémentée dans chaque objet géométrique
- **Peut-on s'en sortir en utilisant uniquement la notion d'héritage ?**
  - On peut par exemple définir une classe mère `Figure` qui implémente une méthode `dessiner` et les figures géométriques seront des classes filles de cette classe
  - Comment implémenter la méthode `dessiner` dans la classe `Figure` !
  - On peut aussi vouloir implémenter la méthode `dessiner` uniquement dans les classes représentant les objets géométriques

Exemple de cas d'utilisation

- Considésons la situation où l'on souhaite programmer un logiciel dont le but est de manipuler des objets géométriques
- Supposons encore que l'on souhaite pouvoir afficher à l'écran ces formes géométriques : nous devons donc avoir une méthode `dessiner` implémentée dans chaque objet géométrique
- **Peut-on s'en sortir en utilisant uniquement la notion d'héritage ?**
  - On peut par exemple définir une classe mère `Figure` qui implémente une méthode `dessiner` et les figures géométriques seront des classes filles de cette classe
  - Comment implémenter la méthode `dessiner` dans la classe `Figure` !
  - On peut aussi vouloir implémenter la méthode `dessiner` uniquement dans les classes représentant les objets géométriques
  - Cependant on ne peut pas s'assurer que cela sera fait systématiquement !

Méthodes abstraites

- Une méthode abstraite est définie uniquement par son intitulé **sans code**

```
abstract class figure{
  protected double longueur;

  astract void computeLongueur;
}
```

- Dans les classes dérivées, la méthode abstraite doit être décrite
- **Remarque :**
  - Une classe devient abstraite dès lors **qu'au moins une méthode est abstraite** et cela doit être précisé explicitement dans la déclaration de la classe avec le mot clef **abstract**
  - Une classe abstraite peut contenir des méthodes concrètes
  - Une classe peut être déclarée abstraite sans contenir de méthode abstraite

Mise en place d'un protocole pour un calcul d'aire

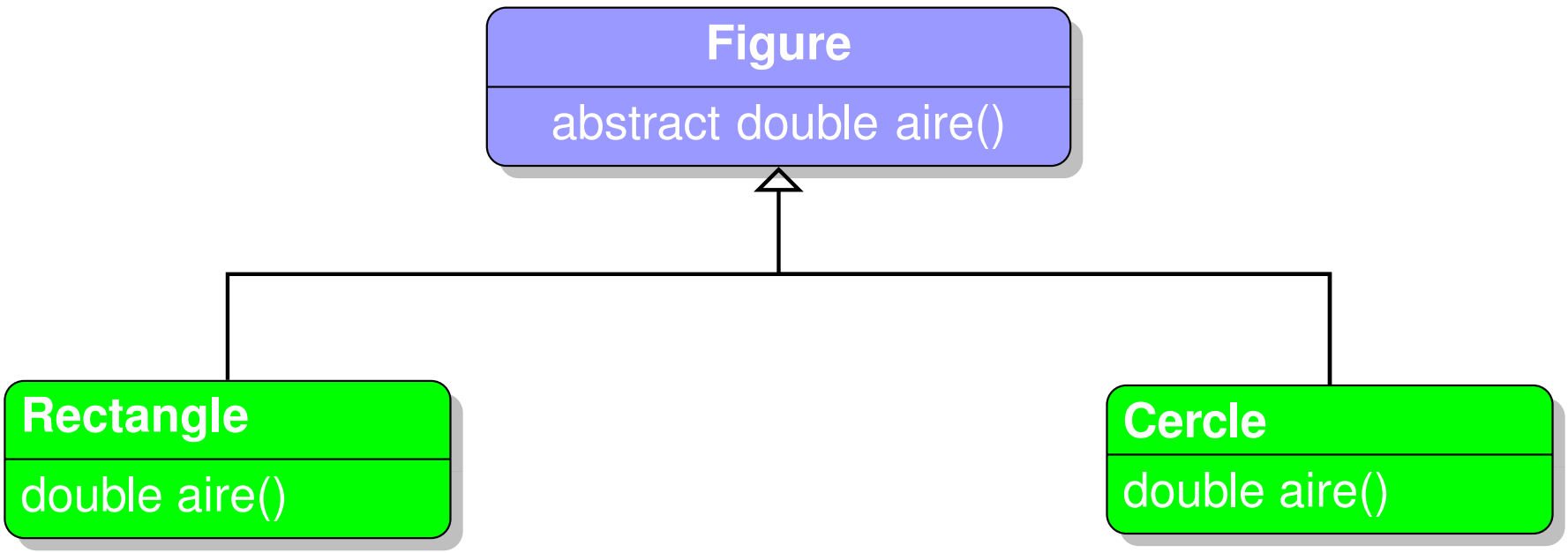
- **Objectif** :
  - un dessin est un ensemble de figure quelconques
  - l'aire d'un dessin c'est la somme de l'aire de ses figures

Mise en place d'un protocole pour un calcul d'aire

- **Objectif** :
  - un dessin est un ensemble de figure quelconques
  - l'aire d'un dessin c'est la somme de l'aire de ses figures
- **Solution** :
  - faire du polymorphisme sur une méthode `aire()`
  - toutes les classes doivent disposer de cette méthode

Mise en place d'un protocole pour un calcul d'aire

- **Objectif** :
  - un dessin est un ensemble de figure quelconques
  - l'aire d'un dessin c'est la somme de l'aire de ses figures
- **Solution** :
  - faire du polymorphisme sur une méthode `aire()`
  - toutes les classes doivent disposer de cette méthode



Exemple : déclaration de la classe abstraite

```
public abstract class Figure {
  private String nom;

  // Bien qu'une classe abstraite ne peut pas être
      instanciée,
  // elle peut avoir un constructeur :
  public Figure(String nom) {
    this.nom=nom;
  }

  // Nous définissons la méthode abstraite :
  public abstract double aire();

  // Cependant, toutes les méthodes ne sont pas
      forcément abstraites :
  public String toString() {
     return "Je suis un(e) " + this.nom;
  }
}
```

Exemple : implémentation d'un cercle

```
public class Cercle extends Figure {
  private double rayon;

  public Cercle(double rayon) {
    super("cercle");
    this.rayon = rayon;
  }

  public double aire() {
    return Double.PI * this.rayon * this.rayon;
  }
}
```

Exemple : implémentation d'un rectangle

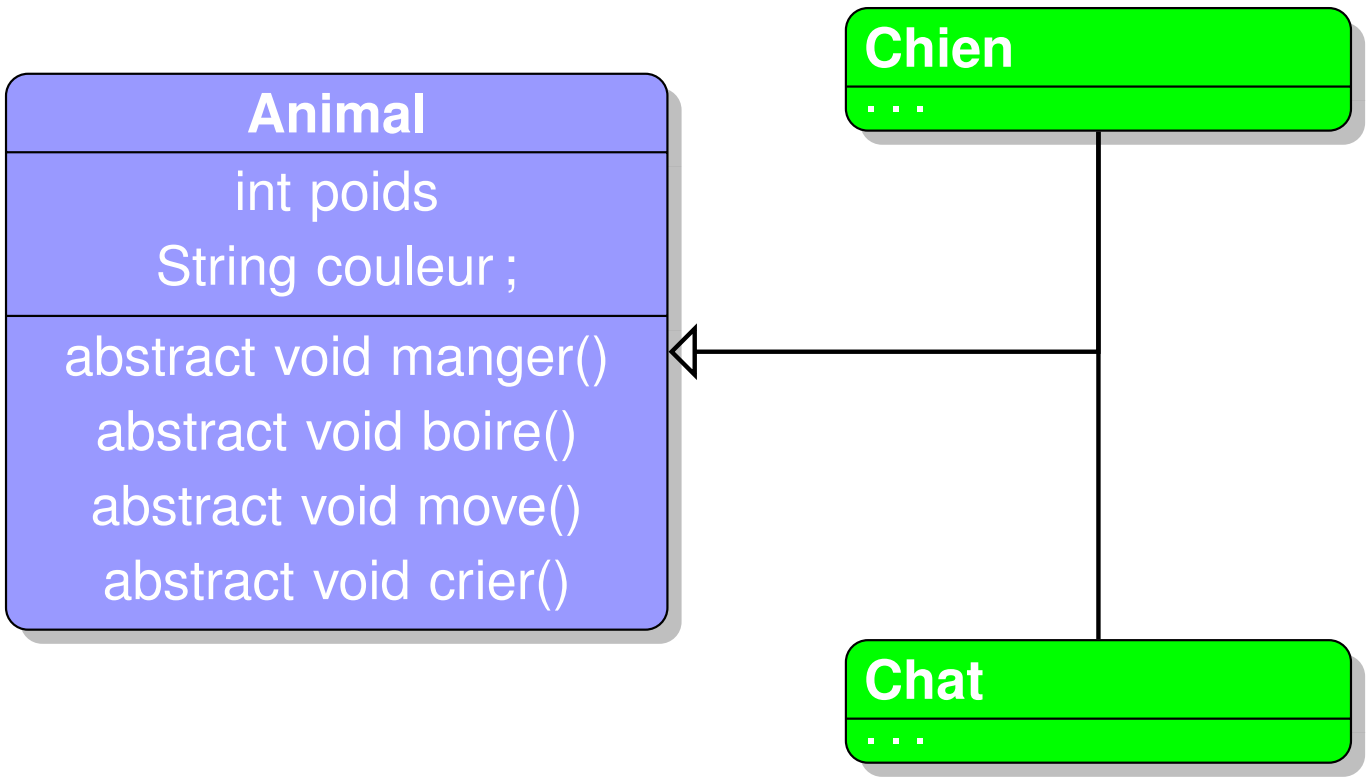
```
public class Rectangle extends Figure {
  private double largeur;
  private double longueur;

  public Rectangle(double largeur,double longueur) {
    super("rectangle");
    this.largeur = largeur;
    this.longueur = longueur;
  }

  public double aire(){
    return this.largeur * this.longueur;
  }
}
```

Exercice

- Considérons que l'on souhaite gérer diférents types d'animaux (chiens, chats, tigres, . . .)
- Les animaux ont de nombreux points communs : une couleur, un poids, un cri, une façon de se déplacer, ils mangent et boivent quelque chose
- Dessinez le diagramme de classe et proposez une implémentation

Exercice

- Considérons que l'on souhaite gérer diférents types d'animaux (chiens, chats, tigres, . . .)
- Les animaux ont de nombreux points communs : une couleur, un poids, un cri, une façon de se déplacer, ils mangent et boivent quelque chose
- Dessinez le diagramme de classe et proposez une implémentation



La Ferme des animaux (I)

```
public abstract class Animal {
  private int poids;
  private String couleur;
  private String nom;

  public Animal(String nom) { this.nom=nom; }

  public abstract void boire();
  public abstract void manger();
  public abstract void move();
  public abstract void crier();

  public String toString() {
    return "Je suis un animal";
  }
}
```

La Ferme des animaux (II)

```
public class Chien extends Animal{
  public Chien(String nom, int poids, String couleur) {
    super(nom);
    this.poids = poids;
    this.couleur = couleur;
  }

  public void boire(){
    System.out.println("Le chien " + nom + " boit");
  }

  public void manger() {...}
  public void move() {...}
  public void crier() {...}

  public String toString() {
    return "Je suis un chien";
  }
}
```

La Ferme des animaux (III)

```
public class Test{
  public static void main(String args[]){
    Animal loup = new Loup();
    Animal chien = new Chien();
    loup.manger();
    chien.crier();
  }
}
```

Table des matières

Un pas de plus vers la généricité

- En Java il est possible de paramétrer vos classes et interfaces
- Cette notion a été empruntée au **langage C++**
- Cela permet de spécialiser les classes qui gèrent les collections en fonction de la classe des objets que ces collections servent à manipuler
- Supposons que l'on souhaite gérer une liste de chaîne de caractères

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- Il est nécessaire de faire un `cast` lorsqu'on accède à un élément
- De plus, on n'est pas sûr que les éléments de cette liste seront des `String`. C'est dans ce contexte qu'on utilise les classes paramétrées

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // pas de cast !
```

Pourquoi utiliser des classes génériques

- Une classe ou une interface générique est **paramétrisé suivant un type**
- Considérons l'exemple suivant :

```
public class Box {
  private Object object;

  public void set(Object o){this.object = o;}
  public Object get(){return object;}
}
```

- Puisque la méthode accepte et retourne un `Object`, vous êtes libre de fournir l'objet que vous souhaitez
- Il est impossible de s'assurer du type des objets !
- **Cependant, on aimerait pouvoir créer des boîtes d'un certain type**
- C'est en cela que va servir de paramétrer la classe

Définir une classe générique

- Une classe générique est construite suivant le format suivant :

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

- La section permettant de paramétrer la classe est délimitée par des chevrons <>
- $T1, T2, \ldots, Tn$ donne les types des paramètres
- Si nous souhaitons une classe `Box` qui contient des éléments particuliers (qu'on ne connaîtra qu'à l'instantiation de la classe) nous pouvons faire comme suit :

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

Convention de nommage

- Par convention les noms des types de paramètres sont représentés par des lettres majuscules

- Les types les plus couramment utilisés sont :
  - E : pour des éléments (liste, collection, . . .)
  - K : pour des clefs
  - N : pour des nombres
  - V : pour des valeurs
  - T : pour des types
  - S, U, V, . . . : pour les types suivants

- Sans cette convention il devient difficile de s'y retrouver dans le code

Invoquer et instancier un type générique

- **Afin de référencer un objet générique vous devez fournir les types entre chevrons**

```
Box<Integer> integerBox;
```

- Vous pouvez imaginer que vous invoquer une méthode et que le type que vous passez est un paramètre
- Le code précédent ne fait que déclarer un objet générique
- **Pour instancier un objet générique il suffit de fournir le type lors de l'appel du constructeur**
- Cela se fait de la même manière que pour la déclaration, c'est-à-dire en passant le paramètre entre chevrons

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

- Depuis Java 7, il est possible dès lors qu'il n'y a pas d'ambiguïté d'omettre le type dans l'appel du constructeur

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

Plusieurs paramètres

- Comme énoncé en préambule, il est possible d'utiliser plusieurs paramètres

```
public class Paire<K, V> {
  private K key;
  private V value;

  public Pair(K key, V value) {
    this.key = key;
    this.value = value;
  }

  public K getKey(){ return key;}
  public V getValue(){ return value;}
}
```

- L'instanciation est similaire au cas où il n'y a qu'un paramètre

```
Paire<String, Integer> p1=new Paire<String, Integer>("Pair",8);
Paire<String, String> p2 = new Paire<>("hello", "world");
```

- Il est aussi possible d'utiliser un type paramétré comme paramètre

```
Paire<String, Box<Integer>> p1=new Paire<>("Pair",new
    Box<Integer>(8));
```

Méthodes génériques

- **Les méthodes génériques** sont des méthodes qui introduisent leurs propres types
- C'est assez similaire à déclarer une classe générique
- Cependant, le scope des paramètres est **local à la méthode**

```
public class Util {
  public <K, V> boolean compare(Pair<K, V> p1, Pair<K, V>
      p2) {
    return p1.getKey().equals(p2.getKey()) &&
    p1.getValue().equals(p2.getValue());
  }
}

// dans un main
Util u = new Util();
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = u.<Integer, String> compare(p1, p2);
```

- Il est aussi possible des déclarer des méthodes paramétrés `static`
- Il est possible d'omettre les paramètres lorsqu'il n'y a pas d'ambiguïté

Limiter le type des paramètres des méthodes

- Il existe de nombreuses situations où les types des **objets que l'on souhaite utiliser comme paramètre sont limités**
- Par exemple, on peut vouloir que les paramètres donnés sont des sous-classes d'un autre type
- Pour déclarer une méthode paramétré qui force cette propriété, il suffit d'utiliser le mot clef `extends`

```
class Test
{
  public <T extends Integer> void test(T v){
    System.out.println("Integer " + v);
  }

  public static void main(String[] args){
    Test t = new Test();
    t.test(3);
    t.test("toto"); // erreur à la compilation
  }
}
```

Limiter le type des paramètres des classes

- De la même manière que pour les méthodes, **il est possible de limiter le type des paramètres possibles pour les classes**
- Cela se fait comme pour les méthodes en utilisant le mot clef `extends`

```
public class NaturalNumber<T extends Integer> {
  private T n;

  public NaturalNumber(T n)  { this.n = n; }

  public boolean isPaire() {
    return n.intValue() % 2 == 0;
  }
}
```

- Il est possible **d'étendre de plusieurs types** de la manière suivante :

```
<T extends B1 & B2 & B3>
```

- Cette notation fonctionne aussi pour les méthodes !

Généricité, héritage et sous-types

- Comme vous le savez déjà Java est capable de faire du transtypage
- Ainsi il est possible d'utiliser un type plus spécifique lors de l'instanciation d'une méthode

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

- Cependant, cela ne fonction pas lorsque l'on considère l'objet lui même
- Considérons la situation suivante :

```
public void boxTest(Box<Number> n) { /* ... */ }
```

- Le code suivant ne fonctionnera pas :

```
Box<Integer> bi = new Box<>(3);
boxTest(bi); // erreur !
```

Wildcard

- Soit la classe `Forme` qui définie la méthode `draw` et supposons que les classes `Rect` et `Ovale` étendent de cette classe
- Supposons que l'on souhaite dessiner tous les objets de type `Forme` compris dans une collection

```
public static void drawAll(Collection<Forme> formes) {
  for (Forme s : formes) s.draw();
}
```

- Cette méthode fonctionne très bien si nous l'appliquons avec une collection d'objets de type `Forme`
- Cependant, si on se rappel du slide précédent, on voit très bien qu'on ne peut pas utiliser cette méthodes avec une collection d'objets `Rect`
- Pour palier ce problème il est possible d'utiliser le *wildcard* `?`

Wildcard en pratique

- L'utilisation du `wildcard ?` se fait simplement de la manière suivante

```
public static void drawAll(Collection<?> formes) {
  for (Forme s : formes) s.draw();
}
```

- Cependant, dans le contexte qui nous intéresse nous aimerions nous assurer que les objets de la collection sont bien de type `Forme`
- Pour cela, il est possible d'utiliser le mot clef `extends`

```
public static void drawAll(Collection<? extends Forme>
    formes) {
  for (Forme s : formes) s.draw();
}
```

- Ainsi, nous sommes sûr que les objets de notre collection implémenteront la méthode `draw`

Exercice

- Définir une classe `Pile` générique

Exercice

- Définir une classe `Pile` générique

```
public classe Pile<T>
{
  private T tab[];
  private int sz;

  public Pile(){tab = new T[1000];}

  public boolean estVide(){return sz == 0;}

  // /!\ taille du tableau tab
  public void empile(T a){
    tab[sz++] = a;
  }

  public void depile(){
    sz--;
  }

  // /!\ la pile est peut être vide
  public T sommet(){
    return tab[sz-1];
  }
}
```