

Développement Orienté Objets

Généricité et Structures de données

Petru Valicov
petru.valicov@umontpellier.fr

<https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets>

2021-2022



1

Généricité - problématique

- nous avons vu que l'utilisation des "casts" peut engendrer des erreurs d'exécution (non-détectées à la compilation)...
- ... néanmoins garantir la présence d'un certain type est souvent nécessaire :

```
ArrayList listeEntiers = new ArrayList(); // déclarée ainsi, il s'agit d'une liste de Object
listeEntiers.add(56); // emballage automatique en Integer (qui hérite de Object)
Integer x = (Integer) listeEntiers.get(0); // transtypage (cast) obligatoire
```

Comment être sûr que les objets de la liste seront des entiers ?

```
ArrayList listeEntiers = new ArrayList();

for (int i = 0; i < 3; i++){
    double pileOuFace = Math.random();
    if (pileOuFace < 0.5)
        listeEntiers.add("bidule");
    else
        listeEntiers.add(i);
}

Integer x = (Integer) listeEntiers.get(0); // au doigt mouillé ça doit marcher...
```

Solution : restreindre le type des données contenues dans la liste à un seul type (par exemple, ici Integer).

2

Généricité - problématique

- nous avons vu que l'utilisation des "casts" peut engendrer des erreurs d'exécution (non-détectées à la compilation)...
- ... néanmoins garantir la présence d'un certain type est souvent nécessaire :

```
ArrayList<Integer> listeEntiers = new ArrayList<>(); // équivalent à new ArrayList<Integer>()
listeEntiers.add(56);
Integer x = listeEntiers.get(0);
```

```
ArrayList<Integer> listeEntiers = new ArrayList<>(); // équivalent à new ArrayList<Integer>()

for (int i = 0; i < 3; i++){
    double pileOuFace = Math.random();
    if (pileOuFace < 0.5)
        listeEntiers.add("bidule"); // ERREUR de compilation
    else
        listeEntiers.add(i);
}

Integer x = listeEntiers.get(0); // plus besoin de transtyper
```

3

Généricité

La **généricité** permet d'abstraire sur les types.

- Utilisée partout où garantir un type est nécessaire (type de variable/champ, constructeur, ...)
- Participe à la *type safety* (en éliminant les "casts")
- Syntaxe : `NomDeLaClasse<T1,T2,...>`. **Exemples** :

```
public class Box<T> {
    private T item;

    public void add(T item) {
        this.item = item;
    }

    public T get() {
        return item;
    }
}
```

```
public interface List<E> {

    void add(E x);

    void add(int index, E element);

    E remove(E x);

    // d'autres méthodes définies dans l'API
}
```

Ici on dit que Box est une *classe générique* ou *classe paramétrée*.

4

Généricité : exemple

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

```
public class Tool {  
    private int size;  
    private String description  
  
    public Tool(int size, int description){  
        this.size = size;  
        this.description = description;  
    }  
  
    public String toString(){  
        return "outil " + description + " de taille " + size;  
    }  
}
```

```
public class ClasseClient {  
    public static void main(String[] args) {  
        Box<Integer> boite = new Box<>();  
        boite.add(42);  
        System.out.println(boite.get());  
  
        Box<Tool> boite = new Box<>();  
        boite.add(new Tool(5, "marteau"));  
        System.out.println(boite.get());  
    }  
}
```

5

Généricité : comment ça marche ?

- En Java les types génériques existent uniquement à la compilation
- Après vérification, le compilateur va **remplacer** tous les types génériques par leur super-type

```
// code initial écrit par le programmeur  
public class Box<T> {  
    private T item;  
  
    public void add(T item){  
        this.item = item;  
    }  
  
    public T get() {  
        return item;  
    }  
}
```

⇒
compilation

```
// code après la compilation  
public class Box {  
    private Object item;  
  
    public void add(Object item){  
        this.item = item;  
    }  
  
    public Object get() {  
        return item;  
    }  
}
```

Effacement de type (type erasure)

Mécanisme qui supprime les annotations de type d'un programme avant l'exécution.

Attention au diamant <> :

```
Box<Integer> boite = new Box<>(); // instruction non-sûre (et dangereuse) !
```

6

Effacement du type - restrictions

Quelques problèmes se posent :

- Les types paramétrés sont *uniquement* des types objets (pas de int, boolean, etc.)
- À l'intérieur d'une classe/méthode paramétrée par un type **T**, il est impossible d'instancier un objet de type **T**.
- Puisque l'effacement du type a lieu *avant* la production du byte-code, on perd de la flexibilité pour la surcharge :

```
public void trier(List<String> chainesDeCaracteres) {  
    // traitement  
}  
  
public void trier(List<Integer> nombres) {  
    // traitement  
}
```

Le code ci-dessus ne compile pas !

7

Généricité multiple

```
public class Entry<K,V>{  
    private final K key;  
    private final V value;  
  
    public Entry(K k, V v){  
        key = k;  
        value = v;  
    }  
  
    public K getKey(){  
        return key;  
    }  
  
    public V getValue(){  
        return value;  
    }  
  
    public String toString(){  
        return "(" + key + ", "+value+")";  
    }  
}
```

```
Entry<String, String> grade = new Entry<>("Albert", "B");  
Entry<String, Integer> mark = new Entry<>("Albert", 16);  
System.out.println("grade: " + grade);  
System.out.println("mark: " + mark);
```

8

Généricité - imbrication

Le type générique peut être de n'importe quel type objet (y compris de type paramétré) :

```
ArrayList<ArrayList<Integer>> tableau2D = new ArrayList<>();

for (ArrayList<Integer> ligne: tableau2D)
    for (Integer element: ligne)
        System.out.println(element);
```

9

Généricité et l'héritage

Puisque `Object` est la super-classe de toutes les classes, peut-on remplacer la déclaration d'un type générique par `Object` ?

```
ArrayList<Object> liste = new ArrayList<String>(); // ???

liste.add("toto");
liste.add(new Object()); // ???
liste.add(new Etudiant("Fifi"));

String s0 = liste.get(0); // ???
String s1 = liste.get(1); // ???
String s2 = liste.get(2); // ???
```

Quels sont les problèmes dans le programme ci-dessus ?

`ArrayList<Object>` ne peut pas être considérée comme une super-classe de toutes les `ArrayList<Chose>`, où `Chose` est un type donné (Integer, String, Employe, Voiture, etc.)

10

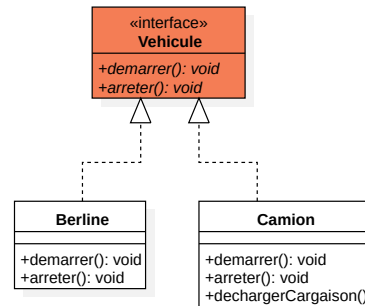
Généricité restreinte

```
public class Garage<T extends Vehicule> {
    private ArrayList<T> vehicules;

    public Garage(ArrayList<T> vehicules) {
        this.vehicules = vehicules;
    }

    public void garer(T vehicule){
        vehicules.add(vehicule);
    }

    public T sortir(){
        return vehicules.remove(0);
    }
}
```



```
public class App {
    public static void main(String[] args) {
        //ArrayList<Camion> vehiculeChantier = new ArrayList<>();
        ArrayList<Berline> vehiculesLuxe = new ArrayList<>();

        // du code manipulant vehiculeLuxe
        Garage<Berline> garageLuxe = new Garage<>(vehiculesLuxe);
        garageLuxe.garer(new Berline());
    }
}
```

Dans cet exemple le mot-clé `extends` signifie "héritage" au sens général, y compris lorsqu'il s'agit d'une interface.

11

Méthodes génériques

Utilisées lorsque la portée du type générique ne sort pas en dehors de la méthode

```
public class App {
    public <U> void fillBoxes(U u, List<Box<U>> boxes) {
        for (Box<U> box : boxes)
            box.add(u);
    }

    public <U extends Employe> U getLePlusRiche(List<U> employes) {
        double max = 0;
        U lePlusRiche = null;
        for (U employe : employes)
            if (max < employe.getSalaireBrut()){
                max = employe.getSalaireBrut();
                lePlusRiche = employe;
            }
        return lePlusRiche;
    }
}
```

12

Généricité - joker (wildcard)

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

    public Entry(K k, V v) {
        key = k;
        value = v;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    public String toString() {
        return "(" + key + ", " + value + ")";
    }
}
```

```
// Un joker peut être utilisé pour plus de flexibilité :
Entry<?, ?> randomNotation = new Entry<String, Double>("Albert", 0.7);
Entry<?, ?> randomNotation = new Entry<String, String>("Donald", "zero");
```

13

Généricité - joker (wildcard)

Rappel : `ArrayList<Object>` n'est pas un super-type de `ArrayList<T>` où T est un type donné.

Comment faire si on souhaite appliquer un traitement général sur des objets des classes paramétrées quelconques (inconnues à l'avance) ?

```
public class App {
    public void afficher(ArrayList<Object> liste) {
        for (Object element : liste)
            System.out.println(element); // invocation de la méthode toString() de Object (ou de ?)
    }

    public void utiliser() {
        ArrayList<Integer> listeEntiers = new ArrayList<>();
        listeEntiers.add(29);
        afficher(listeEntiers); // ERREUR de compilation
        int entier = listeEntiers.get(0) + 3;
    }

    public void utiliser2() {
        ArrayList<Object> listeEntiersVusCommeDesObject = new ArrayList<>();
        listeEntiersVusCommeDesObject.add(29);
        afficher(listeEntiersVusCommeDesObject); /* fonctionne mais les références dans la liste
                                                    ont Object comme type apparent... */
        int entier = listeEntiersVusCommeDesObject.get(0) + 3; // ERREUR de compilation (besoin du cast)
    }
}
```

14

Généricité - joker (wildcard)

Rappel : `ArrayList<Object>` n'est pas un super-type de `ArrayList<T>` où T est un type donné.

```
public class App {
    public void afficher(ArrayList<?> liste) {
        for (Object element : liste)
            System.out.println(element);
    }

    public void utiliser() {
        ArrayList<Integer> listeEntiers = new ArrayList<>();
        afficher(listeEntiers); // fonctionne
        int entier = listeEntiers.get(0) + 3;

        ArrayList<Object> listeEntiersVusCommeDesObject = new ArrayList<>();
        listeEntiersVusCommeDesObject.add(29);
        afficher(listeEntiersVusCommeDesObject); // fonctionne comme avant
        int entier = (Integer) listeEntiersVusCommeDesObject.get(0) + 3;
    }
}
```

Le joker \approx type inconnu. Noté `?` et utilisé lorsque le code ne dépend pas du type de paramètre.

15

Attention au joker !

```
public class App {
    public void ajouter(ArrayList<Object> liste) {
        liste.add(10); // compile et s'exécute
        liste.add(new Object()); // compile et s'exécute
    }

    public void ajouterJoker(ArrayList<?> liste) {
        liste.add(10); // ERREUR de compilation
        liste.add(new Object()); // ERREUR de compilation
        liste.add(null); // compile et s'exécute
    }
}
```

Plus d'infos sur les pièges et astuces :

- <https://docs.oracle.com/javase/tutorial/java/generics/>
- <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>
- J. Bloesch. *Effective Java, Chapter 5 (3rd Edition)*

16

Joker restreint (borne sup)

`<? extends T>` - défini sur tous les types héritant de T.

```
public double calculerSalaires(ArrayList<? extends Employee> personnel) {  
    double sommeFinale = 0;  
    for (Employee e : personnel) {  
        sommeFinale += e.getSalaireBrut();  
    }  
    return sommeFinale;  
}
```

Attention, pas le droit d'écriture dans la ArrayList dans ce cas :

```
ArrayList<Vendeur> vendeurs = new ArrayList<>();  
ArrayList<? extends Employee> employes = vendeurs;  
employes.add(new Employee()); // ERREUR de compilation - et heureusement !
```

Donc :

```
public void methodeErronée(ArrayList<? extends Employee> personnel) {  
    personnel.add(new Employee()); // ERREUR de compilation  
}
```

17

Joker restreint (borne sup) – un autre exemple

```
public class App {  
  
    public void essayerFonctionnement(ArrayList<? extends Vehicule> liste) {  
        for (Vehicule vehicule : liste) {  
            vehicule.demarrer();  
            vehicule.arreter();  
            System.out.println(vehicule + " essayé, on va l'acheter ! ");  
        }  
    }  
  
    public static void main(String args[]) {  
        ArrayList<Camion> listeCamions = new ArrayList<>();  
        essayerFonctionnement(listeCamions);  
  
        ArrayList<Vehicule> listeVehicules = new ArrayList<>();  
        listeVehicules.add(new Camion());  
        listeVehicules.add(new Berline());  
        essayerFonctionnement(listeVehicules);  
    }  
}
```

18

Joker restreint (borne inf)

`<? super E>` : borne inférieure

- Fonctionne sur tous les types ayant E comme sous-type

```
public class Agregat<E> {  
    ArrayList<E> contenu;  
  
    public Agregat(ArrayList<E> contenu) {  
        this.contenu = contenu;  
    }  
    public void exporter(ArrayList<? super E> destination) {  
        for (E valeur : contenu) {  
            destination.add(valeur);  
        }  
    }  
}
```

```
ArrayList<Integer> liste = new ArrayList<>();  
// ajout de pleins d'éléments dans la liste avec liste.add(...)  
Agregat<Integer> agregat = new Agregat<>(liste);  
ArrayList<Integer> dest = new ArrayList<>();  
agregat.exporter(dest); // on exporte dans une liste de Integer  
ArrayList<Number> dest2 = new ArrayList<>();  
agregat.exporter(dest2); // on exporte dans une liste Number
```

19

Java Collections Framework

- Contient différentes APIs de structures de données classiques (appelées "collections")
- Hiérarchie d'interfaces, d'implémentations et d'algorithmes
- Permet une meilleure interopérabilité avec les autres APIs du langage

Organisation :

- Dans le package `java.util`
- Deux interfaces de base :
 - `java.util.Collection` - structures *itérables*
 - `java.util.Map` - structures de la forme *tableau associatif*
- Une interface pour itérer sur les collections :
 - `java.util.Iterator`

20

Java Collections Framework

- On utilise en général comme type apparent une des interfaces (List, Map,...) et on choisit l'implémentation correspondante
- L'intérêt** : changer plus facilement d'implémentation

```
List<Integer> liste = new LinkedList<>();

// du code utilisant la liste
int nouveauNombre = 42;
liste.add(nouveauNombre);

int moyenne = 0;
for (Integer e : liste){
    System.out.println("Valeur : " + e);
    moyenne += e;
}
if (liste.size() > 0)
    moyenne /= liste.size();
System.out.println("Moyenne = " + moyenne);
```

On peut appliquer le **principe de substitution** sur la première ligne.

21

API de collections

1. Interface Collection<E> :

- Set - notion d'ensemble mathématique
- List - notion de liste classique
- Queue - stockage temporaire (file FIFO)
- Deque (double ended queue) - file à deux bouts

2. Interface Map<K,V> – généricité sur deux types

Consigne

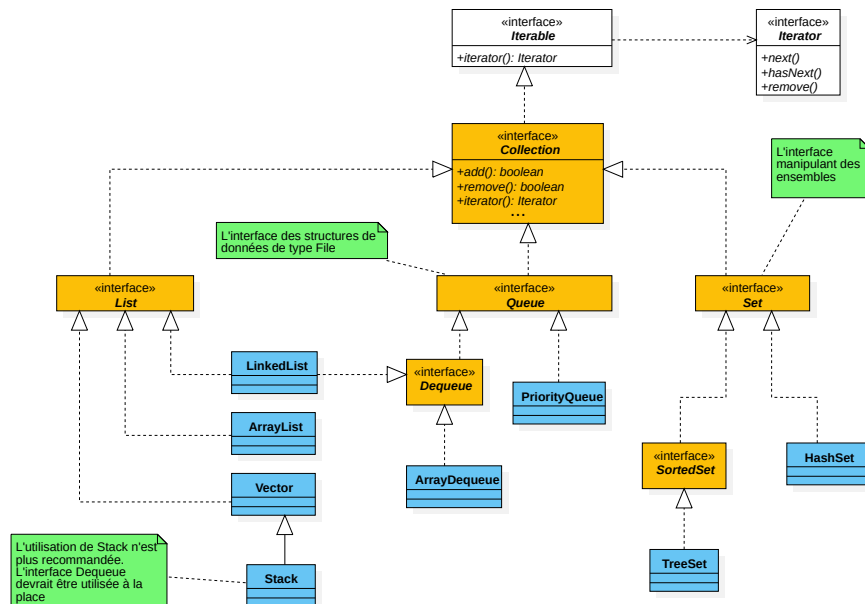
Avant d'aller au prochain TP, consulter la documentation officielle :

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collection.html>

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html>

22

java.util.Collection



23

Interface Collection<E>

```
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element);
boolean remove(Object element);
Iterator<E> iterator();
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
void clear();
Object[] toArray();
```

24

Un moyen simple d'itérer

```
Collection<VotreType> col = new ArrayList<>();
for (VotreType elem : col) {
    System.out.println(elem);
}

List<Integer> liste = new LinkedList<>();
for (Integer elem : liste) {
    System.out.println(elem);
}
```

- Utilise en interne la méthode `iterator()` de `Collection`
- Également appelé *for-each* ou boucle *for renforcée*
- Depuis Java 8 on peut écrire ces instructions de manière "fonctionnelle" (en utilisant les *lambdas expressions*) :

```
List<Integer> liste = new LinkedList<>();
liste.forEach (elem -> System.out.println(elem) );
```

25

Itération sur les collections

```
List<Integer> liste = new ArrayList<>();
for (int i=0; i<liste.size(); i++) {
    liste.add(35); // compile, s'exécute. Mais...
    liste.remove(liste.get(0)); // compile, s'exécute. Mais...
}
```

La boucle *for renforcée* interdit les modifications de la collection :

```
List<Integer> liste = new ArrayList<>();
for (Integer element : liste) {
    liste.add(35); // une "exception" (ici "erreur")
                  // de type java.util.ConcurrentModificationException
                  // va être levée à l'exécution
}
```

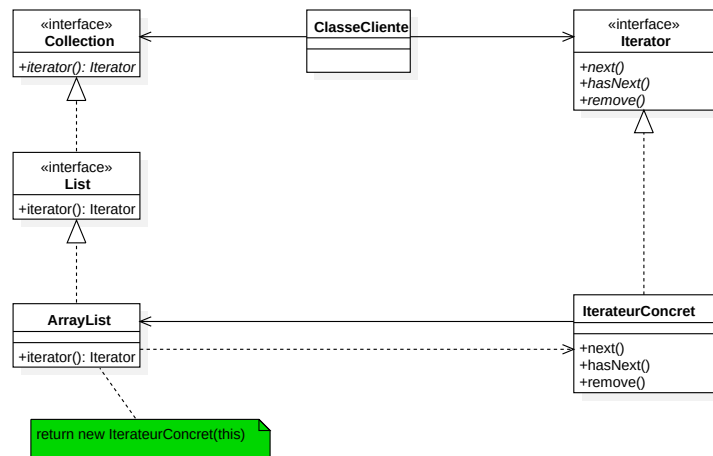
Conseil

Évitez d'utiliser le `for` classique pour parcourir et/ou modifier une collection.

26

Itération : l'interface `java.util.Iterator`

Objectif : fournir un moyen d'accès séquentiel aux éléments d'une agrégation d'objets, sans exposer la représentation interne de celle-ci



L'utilisateur (client) n'est pas concerné par la manière dont les objets sont gérés/organisés dans la collection.

27

Écrire son propre itérateur : exemple

```
public class Entreprise {
    private String nom;
    private List<Personne> employees;
    public Entreprise(String nom) {
        this.nom = nom; employees = new LinkedList<>();
    }

    public void embaucher(Personne p) { employees.add(p); }

    public Iterator<Personne> getIterator() {
        return new IterateurConcret(employees);
    }
}
```

```
public class Personne {
    private String nom, prenom;
    public Personne(String nom, String prenom) {
        this.nom = nom; this.prenom = prenom;
    }

    public String toString() {
        return "nom=" + nom + ", " + "prenom=" + prenom + " ";
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Entreprise maBoite = new Entreprise("Poire Mordue");
        maBoite.embaucher(new Personne("Bougeret", "Marin"));
        maBoite.embaucher(new Personne("Poupet", "Victor"));
        maBoite.embaucher(new Personne("Valicov", "Petru"));

        Iterator<Personne> it = maBoite.getIterator();
        while(it.hasNext())
            System.out.println(it.next());
    }
}
```

```
// interface définie dans java.util
public interface Iterator<T> {

    // test de fin
    boolean hasNext();

    //retourner courant + passer au suivant
    T next();

    /* méthodes par défaut (implémentation optionnelle) */
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }

    default void forEachRemaining(/* ... */) { /* ... */ }
}
```

```
public class IterateurConcret implements Iterator<Personne> {

    private int position = 0;
    private List<Personne> lesEmployes;

    public IterateurConcret(List<Personne> employees) {
        lesEmployes = employees;
    }

    public boolean hasNext() {
        if (position >= lesEmployes.size())
            return false;
        return true;
    }

    public Personne next() {
        Personne p = lesEmployes.get(position);
        position++;
        return p;
    }
}
```

28

Itération et suppression des éléments

Pour supprimer des éléments, il faut utiliser `iterator()` manuellement.

```
// création d'une collection avec des objets String
Collection<String> tasDeGugusses = new ArrayList<>();
tasDeGugusses.add(...); // ajout d'une référence vers un objet String
tasDeGugusses.add(...); // ajout d'une référence vers un objet String

// itération sur la collection pour supprimer certains éléments
for (Iterator<String> it = tasDeGugusses.iterator(); it.hasNext();) {
    // ici l'itérateur it est instancié de manière anonyme (à travers une classe "jetable")

    String bonhomme = it.next();
    if (bonhomme.equals("Toto"))
        it.remove(); // Supprime la référence bonhomme de tasDeGugusses
}
```

Il est également possible d'utiliser une méthode par défaut de l'interface `Collection` avec une lambda expression :

```
tasDeGugusses.removeIf(bonhomme -> bonhomme.equals("Toto"));
```

Plus d'infos sur les lambdas leur syntaxe sur la doc officielle :

<https://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html>

29

Interface Set

- Modélise les opérations ensemblistes (au sens mathématique) :
 - Unicité des éléments \approx `add(element)` ne change pas la collection et retourne `false` si l'élément est déjà présent
 - Comparaison des ensembles (avec `equals(Object o)`)
 - Les opérations mathématiques avec les ensembles : `addAll(..)`, `containsAll(..)`, `removeAll(..)`, `retainAll(..)`
- Trois implémentations :
 - `HashSet` - *ordre aléatoire*
⇒ performant - utilise `hashCode()` des objets qu'il contient
 - `TreeSet` - *triée* en fonction des valeurs de la collection
⇒ plus lourd à manipuler à cause de l'ordre maintenu
 - `LinkedHashSet` - *ordre d'insertion*
⇒ presque aussi performant que `HashSet`
- L'ordre des éléments est celui de parcours avec *for-each*.

30

Interface Set - exemples

Que font-ils ?

```
Collection<TypeG> col1 = new ArrayList<>();
//ici du code pour ajouter des éléments dans col1
Collection<TypeG> col2 = new HashSet<>(col1);
```

```
import java.util.Set;
import java.util.TreeSet;

public class TestSet {

    public static void main(String[] args) {
        Set<String> ensemble = new TreeSet<>();

        for (String a : args)
            if (!ensemble.add(a))
                System.out.println("Mot " + a);

        System.out.println(ensemble.size() + " mots");
        System.out.println("La liste des mots : " + ensemble);
    }
}
```

31

Interface List<E>

- Collection ordonnée
- Méthodes d'accès à un élément avec son index :
`get(int i)` et `set(int i, E element)`
- Recherche l'index d'un élément : `indexOf`, `lastIndexOf`
- Sous-liste : `List<E> subList(int from, int to)`
- Méthodes d'ajout et de suppression :
 - `boolean add(E element)`
 - `void add(int index, E element)`
 - `E remove(int index)`
 - `boolean addAll(int index, Collection<? extends E> c)`

32

Interface List<E>

Trois principales implémentations :

- **LinkedList** - liste chaînée (avec accès séquentiel)
 - suppression ou insertion rapide
 - accès coûteux
- **ArrayList** - tableau dynamique
 - suppression ou insertion coûteuse
 - accès rapide
- **Vector** - tableau dynamique, supporte le multithreading
 - très lourd et coûteux à l'utilisation
 - la plupart du temps son utilisation est déconseillée

33

Itération sur List

Un itérateur spécifique **ListIterator** (implémente **Iterator**) :

- parcours à l'envers : **hasPrevious()**, **previous()**
- accès à l'index : **nextIndex()**, **previousIndex()**
- modification de la liste (à utiliser avec précaution) :
add(E e), **remove()**, **set(E e)**

```
List<Integer> liste = new LinkedList<>();
ListIterator<Integer> monIterateur = liste.listIterator();

while(monIterateur.hasNext()){
    System.out.println(monIterateur.next());
}

while(monIterateur.hasPrevious()){
    System.out.println(monIterateur.previous());
}
```

34

La classe Stack - exemple de conception erronée

Étend la classe **Vector**

- Principe de pile (ou LIFO)
- Quatre opérations de base :
empty(), **peek()**, **push(E e)**, **pop()**
- Opération supplémentaire : **search(Object o)**

Quelques problèmes avec **Stack** :

- N'est pas une interface
- Hérite de **Vector** - donc par défaut **c'est un tableau !**
- Les opérations de pile ne sont pas optimales

À éviter (utiliser plutôt Deque)

35

Interface Queue<E>

- Stockage temporaire d'éléments en attente de traitement
- On ne choisit pas l'endroit d'insertion
- Uniquement le premier élément (la tête) de la queue est accessible
- Différentes implémentations
 - **LinkedList** : queue FIFO
 - **PriorityQueue** : les éléments sont traités par ordre de priorité
 - **Deque** : une queue utilisable dans les deux sens (LIFO)
- Les méthodes ont deux versions :

add(e)	remove()	element()	lèvent une exception si problème
offer(e)	poll()	peek()	retournent une valeur spéciale

36

Interface Queue : exemple

```
import java.util.PriorityQueue;

public class TestQueue {

    public static void main(String[] args) {

        // Déclaration d'un file de priorité
        PriorityQueue<Integer> maFile = new PriorityQueue<>();

        maFile.add(200);
        maFile.add(400);
        maFile.add(900);
        maFile.add(100);

        while (!maFile.isEmpty()) {
            System.out.println(maFile.remove());
        }
    }
}
```

À l'affichage, les éléments seront ordonnées en fonction de leur priorité : ici en ordre croissant défini par défaut pour les entiers

37

Comment ordonner les objets ?

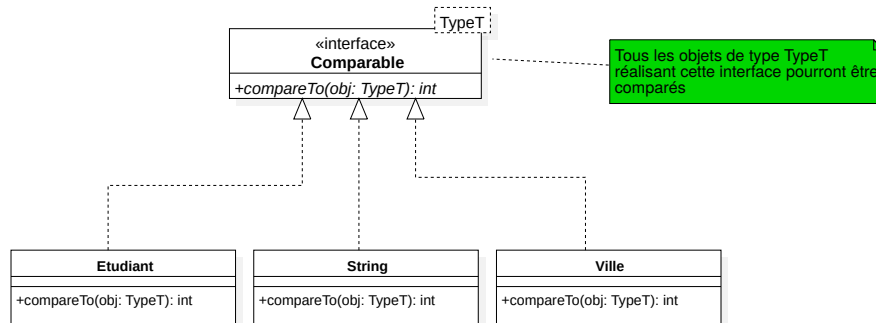
- Certaines implémentations de collections ont un ordre à respecter : TreeSet, PriorityQueue, TreeMap.
- L'ordre est défini par l'interface Comparable
- Les classes Java standard ont déjà un ordre naturel (Integer, String)

Pour utiliser l'interface générique Comparable<TypeT> :

1. implémenter l'interface :
public class MaClasse implements Comparable<MaClasse>
2. définir la méthode `int compareTo(MaClasse o)`
3. redéfinir `boolean equals(Object o)` et `int hashCode()`

38

Exemple d'interface : java.lang.Comparable



1. La méthode abstraite `compareTo()` retourne une valeur entière, censée être :
 - positive si l'objet comparable est supérieur à l'objet obj
 - 0 (zéro) si l'objet comparable est égal à l'objet obj
 - négative si l'objet comparable est inférieur à l'objet obj
2. Cette méthode abstraite **doit être implémentée** dans chaque réalisation de Comparable.

39

Interface Comparable : illustration

```
// L'interface telle que définie dans
// l'API Java. À implémenter par toute
// classe T destinée à être "comparable"
public interface Comparable<T> {
    public int compareTo(T o);
}
```

```
import java.util.PriorityQueue;

public class App {
    public static void main(String[] args) {
        Ville montp = new Ville("Montpellier", 288600);
        Ville sete = new Ville("Sète", 43858);
        Ville paris = new Ville("Paris", 2175601);

        PriorityQueue<Ville> file = new PriorityQueue<>();
        file.add(montp);
        file.add(paris);
        file.add(sete);

        // défilement de la PriorityQueue
        while(!file.isEmpty()) {
            Ville courant = file.poll();
            System.out.println(courant);
        }
        // affiche: Sète, Montpellier, Paris
    }
}
```

```
public class Ville implements Comparable<Ville> {
    private String nom;
    private int population;

    public Ville(String nom, int population) {
        this.nom = nom;
        this.population = population;
    }

    public String toString() {
        return "" + nom;
    }

    public int compareTo(Ville ville) {
        return population - ville.population;
    }

    // equals doit être cohérent avec compareTo...
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Ville ville = (Ville) o;
        return population == ville.population;
    }

    // hashCode doit être cohérent avec equals
    public int hashCode() {
        return population;
    }
}
```

40

Interface Comparator

- Imaginons qu'on ne souhaite pas modifier les objets.
- exemple* : d'abord ordonner les villes suivant leur code postal et plus tard suivant leur nom.
- L'interface Comparator permet de définir un ordre sur les objets sans que cet ordre devienne une propriété de l'objet.

```
// L'interface telle que définie dans l'API Java. À implémenter par toute
// classe définissant un ordre de comparaison sur des objets de type T
public interface Comparator<T>{
    public int compare(T o1, T o2);
}
```

La méthode abstraite compare() retourne une valeur entière :

- positive si o1 est supérieur à o2
- 0 (zéro) si o1 est égal à o2
- négative si o1 est inférieur à o2

41

Interface Comparator - exemple

```
public class OrdreSelonCP implements Comparator<Ville> {
    public int compare(Ville v1, Ville v2) {
        //comparaison en fonction du code postal
        return v1.getCodePostal() - v2.getCodePostal();
    }
}
```

```
public class OrdreSelonNom implements Comparator<Ville> {
    public int compare(Ville v1, Ville v2) {
        //comparaison en fonction du nom
        return v1.getNom().compareTo(v2.getNom());
    }
}
```

```
public class Ville {
    private String nom;
    private int codePostal;

    public Ville(String nom, int cp) {
        this.nom = nom; codePostal = cp;
    }

    public int getCodePostal(){ return codePostal;}

    public String getNom() { return nom; }

    public String toString() { return " " + nom; }
}
```

```
import java.util.TreeSet;
public class TestComparator {
    public static void main(String[] args) {
        Ville montpellier = new Ville("Montpellier", 34000);
        Ville lyon = new Ville("Lyon", 69000);

        TreeSet<Ville> ensemble = new TreeSet<>(new OrdreSelonCP());
        //TreeSet<Ville> ensemble = new TreeSet<>(new OrdreSelonNom());
        ensemble.add(lyon);
        ensemble.add(montpellier);

        for(Ville v : ensemble)
            System.out.println(v);
    }
}
```

42

Comparator et Comparable - remarques

Les deux interfaces permettent de comparer des objets, mais :

- Comparable** est une propriété de l'objet : les classes *deviennent comparables*... et le seront à tout jamais
- Comparator** est un outil de comparaison : plus souple car on n'impose pas de comportement aux objets à comparer.

Il est très facile d'obtenir un ordre *décroissant* :

```
public class OrdreSelonNom implements Comparator<Ville>{
    public int compare(Ville v1, Ville v2) {
        // ordre décroissant suivant les noms
        return v2.getNom().compareTo(v1.getNom());
    }
}
```

```
TreeSet<Ville> ensemble;
ensemble = new TreeSet<>(new OrdreSelonNom());
ensemble.add(lyon);
ensemble.add(montpellier);

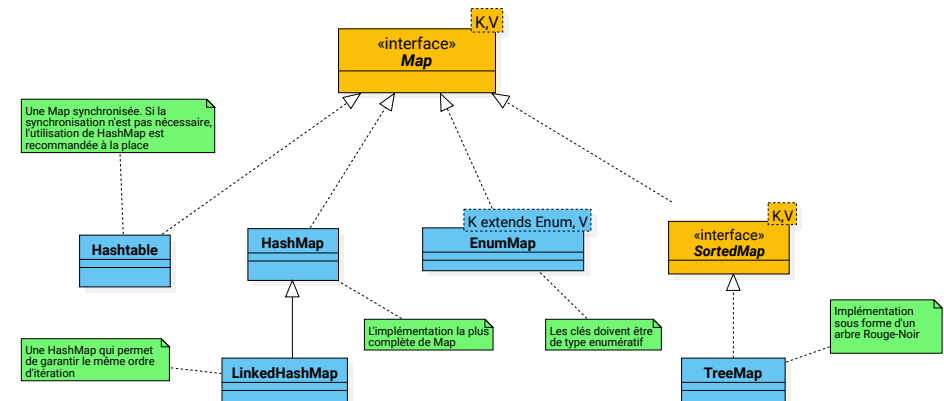
for(Ville v : ensemble)
    System.out.println(v);
```

L'ordre naturel (imposé avec Comparable)) **doit être cohérent** avec boolean equals(Object o).

43

Interface java.util.Map

- Fait correspondre une clef unique de type arbitraire à une valeur
- La clef et la valeur sont de types génériques



Une Map c'est comme un tableau sauf qu'indexé par des objets.

44

Interface Map

```
import java.util.HashMap;
import java.util.Map;

public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> villes = new HashMap<>();

        villes.put("Montpellier", 34);
        villes.put("Sète", 34);
        villes.put("Aix", 13);
        villes.put("Marseille", 13);
        villes.put("Paris", 75);

        String clef = "Montpellier";
        System.out.println("Dép. " + villes.get(clef)); // affiche Dép. 34

        villes.remove(clef);
        System.out.println("Dép. " + villes.get(clef)); // affiche Dép. null

        System.out.println(villes.containsKey("Marseille")); // affiche true
        System.out.println(villes.containsValue(34)); // affiche true
    }
}
```

45

Interface Map

- `get(clef)`, `remove(clef)` - null si pas présent
- `put(clef, valeur)` - efface la valeur existante
- `containsKey(clef)`, `containsValue(val)`
- `size()`, `isEmpty()`, `clear()`
- `Set<K> keySet()`, `Collection<V> values()`
- `putAll(Map<? extends K, ? extends V> m)`

Il est possible d'associer plusieurs valeurs à une clef :

`Map<E, List<E2>>`

46

Interface Map

Mêmes types d'implémentations que pour Set :

- `HashMap` - performance mais aucun garanti d'ordre
- `TreeMap` - les clés sont triées
- `LinkedHashMap` - une `HashMap` mais où l'ordre d'insertion des clés est préservé

Itération sur une Map :

- avec un *for-each* explicite

```
for (Map.Entry<TClef, TVal> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ":" + entry.getValue());
}
```

- avec une expression lambda

```
map.forEach((k, v) -> System.out.println((k + ":" + v)));
```

Map n'implémente pas `Iterable` (contrairement à `Collection`).

47

Algorithmes

- Définis dans la classe `java.util.Collections` (**et pas** `java.util.Collection`).
- La classe contient uniquement des méthodes statiques qui manipulent des collections (des objets de type `Collection`) :
 - `Collections.reverse(List l)`
 - `Collections.swap(List l, int index1, int index2)`
 - `Collections.rotate(List list, distance)`
 - `Collections.shuffle(List l)`
 - `Collections.addAll(Collection c, elements)`
 - `Collections.sort(List l)`
 - `synchronizedCollection(Collection c)`
 - etc...*

Ne pas confondre `Collections` avec `Collection` !!!

48