Programmation Oriente Objet

Les collections et la généricité

Pr Omar EL Midaoui

Plan

- Les collections
- La généricité

Les collections

- Les collections sont des objets permettant de gérer des ensembles d'objets avec éventuellement la possibilité de gérer les doublons, les ordres de tri, etc.
- La version 1 de Java proposait:
  - java.util.Vector, java.util.Stack, java.util.Hashtable
  - Une interface java.util.Enumeration permettant de parcourir ces objets (elle offre deux méthodes : public abstract boolean hasMoreElements() et public abstract Object nextElement())

La classe Vector

- La classe Vector est un " tableau extensible " : on peut y stocker un nombre indéterminé d'objets (le nombre peut augmenter ou diminuer).

Les méthodes de la classes Vector :

Vector()
Vector(int CapacitéInitiale)
Vector(int CapInit, int CapIncrement)
add(Object obj)
addElement(Object obj)
removeElement(Object obj)
firstElement()
lastElement()

isEmpty()
contains(Object obj)
size()
setSize()
indexOf (Object obj)
capacity()
elementAt(int index)

elements()

La classe Vector (2)

Remplissage d'un Vector

```
Vector vect=new Vector ();
for (int i=0; i < 10; i++ )
        vect.addElement("Element de type String numero" + i);
```

Parcours avec elementAt

```
for (int i=0; i < 10; i++ )
        System.out.println(vect.elementAt(i));
```

Parcours avec Enumeration

```
Enumeration enum=vect.elements();
while(enum.hasMoreElements())
        System.out.println(enum.nextElement());
```

Exemple de la collection "stack"

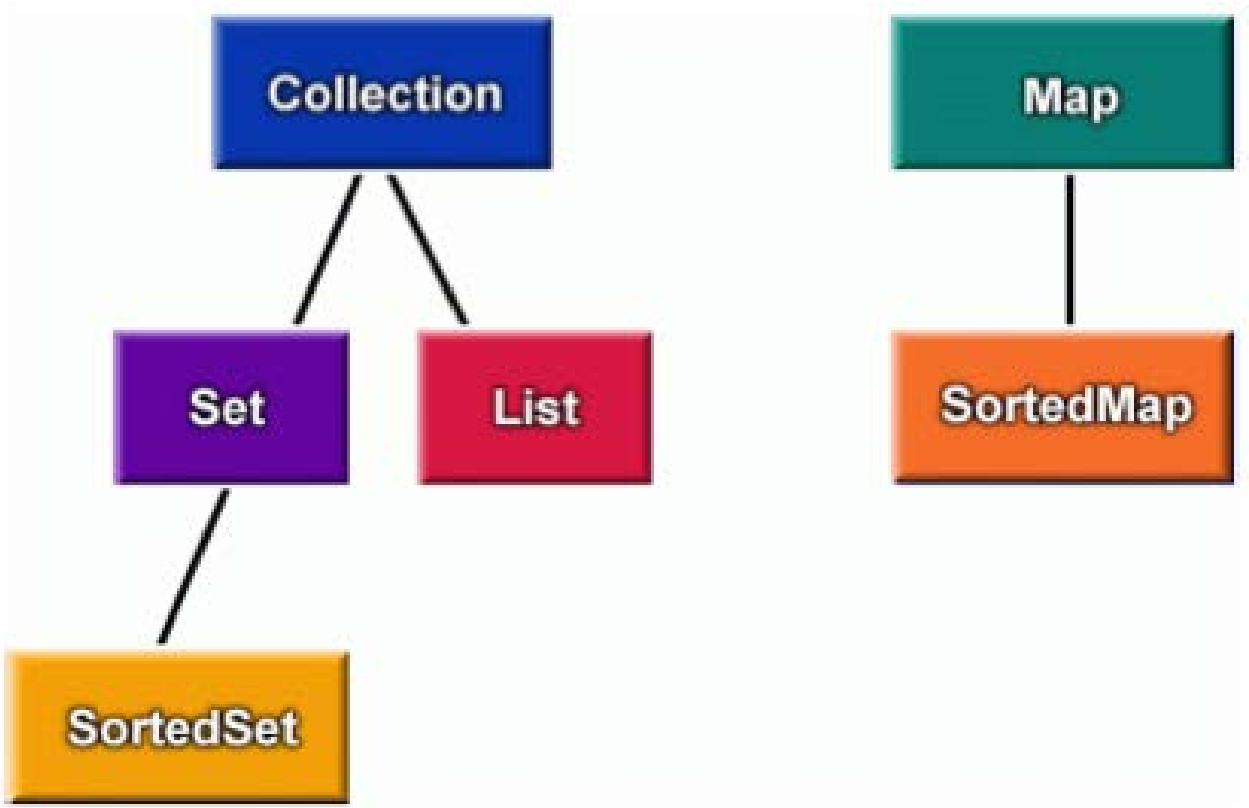
```
package teststack;
import java.util.*
public class ExempleStack
{
Stack pile ;
public ExempleStack ()
{
pile = new Stack () ;
pile.push("Je suis ") ; pile.push("Un exemple ") ; pile.push("de pile")
;
Enumeration enum = pile.elements () ;
while (enum.hasMoreElements()){
System.out.println (enum.nextElement()) ;
}
}
public static void main(String[ ] args){
new ExempleStack () ;
}
}
```

Je suis
Un exemple
de pile

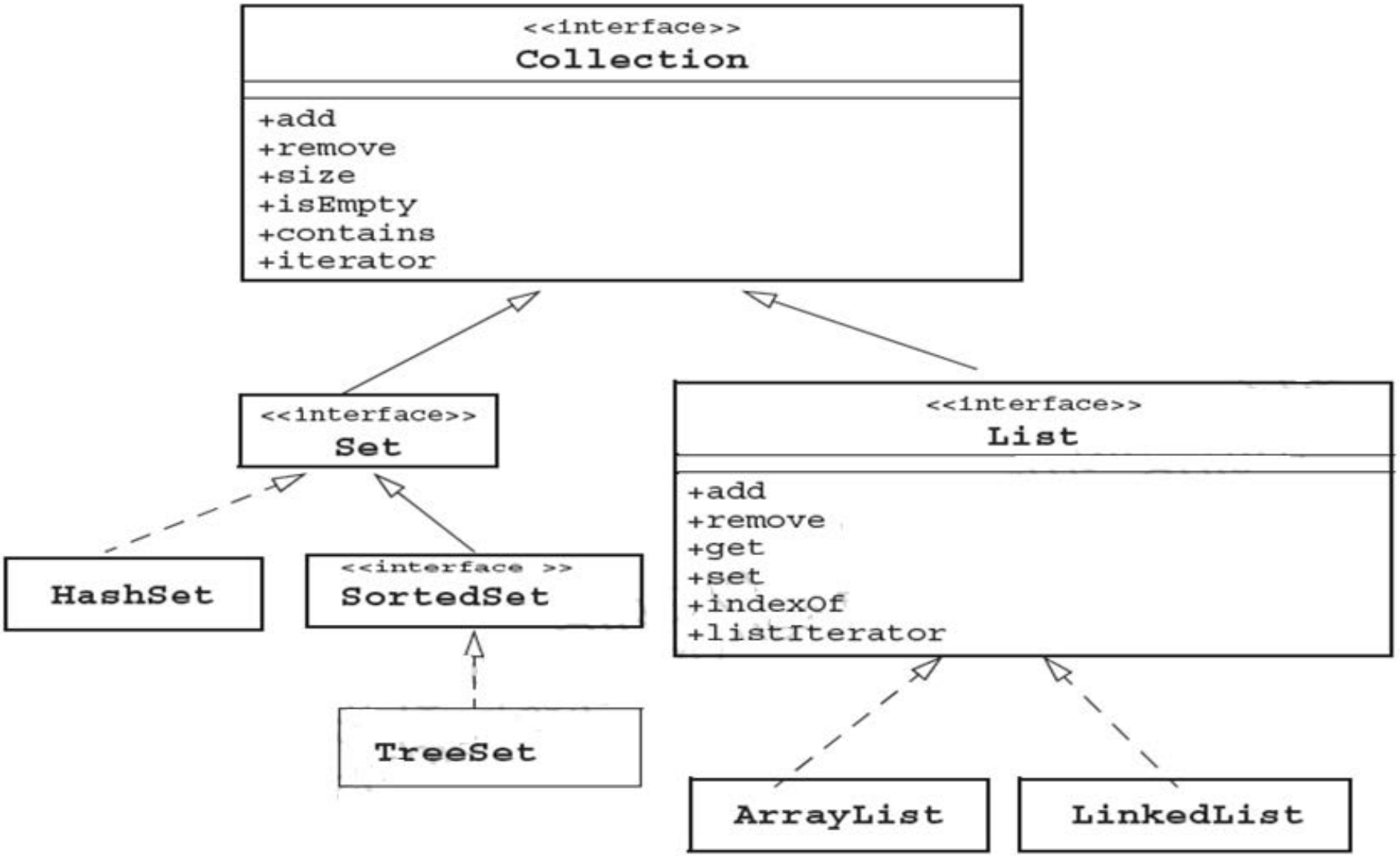
Interfaces de collections

Collection à partir de java 2

Réparties en deux groupes :



Collection à partir de java 2



Collection à partir de java 2

- **Collection :** interface qui est implémentée par la plupart des objets qui gèrent des collections.
- **Map :** interface qui définit des méthodes pour des objets qui gèrent des collections sous la forme **clé/valeur**
- **Set :** interface pour des objets qui n'autorisent pas la gestion des doublons dans l'ensemble
- **List :** interface pour des objets qui autorisent la gestion des doublons et un accès direct à un élément
- **SortedSet :** interface qui étend l'interface Set et permet d'ordonner l'ensemble
- **SortedMap :** interface qui étend l'interface Map et permet d'ordonner l'ensemble

Implémentation des interfaces

Interface	Implémentation
Set	HashSet
SortedSet	TreeSet
List	ArrayList, LinkedList, Vector
Map	HashMap, Hashtable
SortedMap	TreeMap

La classe ArrayList

- Une instance de la classe ArrayList est une sorte de tableau qui peut contenir un nombre quelconque d'instances d'une classe quelconque.
- Les emplacements sont indexés par des nombres entiers (à partir de 0).
- Les constructeurs :
  - ArrayList()
  - ArrayList(int taille initiale) : peut être utile si on connaît la taille finale ou initiale

La classe ArrayList<E> (2)

Les méthodes principales de ArrayList:

- boolean **add(E elt)** : Ajoute un élément elt en fin de liste
- void **add(int i, E elt)** : Insère un élément elt dans la liste, à l'indice i
- boolean **addAll(Collection<? extends E> c)** : copie une collection en fin de liste
- boolean **contains(Object obj)** : returne True si obj est dans la liste, false sinon
- E **get(int indice)** : Retourne l'élément stocké à l'indice i
- int **indexOf(Object obj)**: Retourne l'indice de l?élément obj
- E **remove(int indice)** : Supprime l'élément d'indice i
- void **removeRange(int i, int j)** : Supprime les éléments de l'intervalle d'indices [i,j[
- void **clear()** : Supprime tous les éléments de la liste
- int **size()** : Retourne le nombre d'éléments de la liste

La classe ArrayList (3)

TestArrayList.java

```
public class TestArrayList {
public static void main (String [] args) { ArrayList<Object> L =
new ArrayList<Object>( ); L.add( "Un" ); L.add( 2 ); L.add(
"Lundi" ); L.add( new Etudiant("Ali","Mesnaoui","FSR" ); for
(Object o : L ) { System.out.println( o );} }
}
```

Interface Iterator

Testiterator.java

```
public class TestArrayList {
public static void main (String [] args) { ArrayList<Object> L =
new ArrayList<Object>(100); L.add( "Un" ); L.add( 2 ); L.add(
"Lundi" ); L.add( new Etudiant("Ali","Mesnaoui","FSR" );
Iterator iter = L.iterator()
while ( iter.hasNext())
{ System.out.println(iter.next()); }
}
```

Classements des éléments

Classements des éléments

- Lorsqu'on crée une classe, la JVM n'a aucun moyen de déterminer l'ordre envisagé pour les objets de cette classe.
- L'implémentation de l'interface Comparable permet d'ordonner les objets d'une classe.
- Les interfaces Comparable (compareTo) et Comparator (compare) sont utiles pour classer les collections. Ils permettent de comparer les éléments d'une collection.
- L'interface Comparable fait partie du package java.Lang.
- Les classes Byte, Long, String, Date, Float... implémentent l'interface Comparable.
- La méthode statique sort de **la classe Collections** permet de trier une liste passée en argument.
- Pour permettre de comparer les objets d'une classe, cette dernière doit implémenter la méthode compareTo de l'interface Comparable.

Classements des éléments

redéfinition de compareTo

```
import java.util.* ;
class Etudiant implements Comparable <Etudiant> {
private int code; private String nom;
public Etudiant(int a, String b){code=a; nom=b;}
public String toString() {return "Code:"+code+" Nom: "+nom;}
public int compareTo(Etudiant e)
{
if (this.code == e.code) return 0; // égaux
if (this.code > e.code) return 1; // this est supérieur à e
return -1; // this est inférieur à e
}
}
```

Classements des éléments

Test de compareTo

```
public class TestComparable {
public static void main(String [ ] args)
{ Etudiant e1 = new Etudiant(350, "Amine");
Etudiant e2 = new Etudiant(300, "Khalid");
if(e1.compareTo(e2) == 0)
System.out.println( "Egaux " );
else
if(e1.compareTo(e2) == 1)
System.out.println( "e1 > e2" );
else
System.out.println( "e1 < e2" ); }
}
```

Classements des éléments

Utilisation de Collections.sort

```
public class TestComparable {
public static void main(String [ ] args)
{ List <Etudiant> A = new ArrayList <Etudiant>();
A.add(new Etudiant(500, "Amina"));
A.add( new Etudiant(300, "Omar"));
A.add( new Etudiant(400, "Houda"));
System.out.println("Avant le tri: "+ A );
Collections.sort(A);
System.out.println("Après le tri: "+ A );
}
}
```

Résultat:

Avant le tri: [500 Amina, 300 Omar, 400 Houda]

Après le tri: [300 omar, 400 Houda, 500 Amina]

La classe Hashtable

La classe Hashtable

- La classe Hashtable implémante l'interface Map (carte) : une carte enregistre des paires clé/valeur (objets)
- La recherche d'une valeur se fait donc à partir de la clé (et non selon un index comme dans le cas d'une table)
- Le dictionnaire est un exemple de carte : les clés correspondent aux mots et les objets associés aux significations
- Seuls les objets définissant les méthodes hashCode() et equals peuvent figurer comme clé dans une carte.
- pour chacun des clés , Hashtable (table de hachage) utilise les fonctions de hachage pour calculer un nombre entier, appelé **code de hachage**.

La classe Hashtable (2)

Les méthodes principales :

- **put(Object Cle, Object Valeur)** : Ajouter un élément
- **get(Object Cle)** : Retrouver un objet (get retourne null si aucune paire ne se trouve avec une telle clé)
- **remove(Object Cle)** : supprime l'entrée correspondant à la clé

Remarque :

- Les clés doivent être uniques.
- Si on fait deux appels à put avec la même clé, la seconde valeur remplacera la première

La classe Hashtable (3)

TestHashtable.java

```
class TestHashtable {
public static void main (String [] args) {
Hashtable rect = new Hashtable();
rect.put ("petit", new Rectangle(0,0,5,5));
rect.put ("moyen", new Rectangle(10,10,15,15));
rect.put ("grand", new Rectangle(20,20,25,25));
Rectangle r = (Rectangle) rect.get("moyen");
//Besoin de sous-casting, car get(..) retourne un object
}
}
```

Généricité

Collection et java 1.5

- Jusqu'à la version 1.4, on stockait et récupérait des "Object" d'une collection.
- **Exemple:**

```
ArrayList liste = new ArrayList ()
liste.add (new Maclasse ()) ;
Maclasse obj = (Maclasse) liste.get (0) ;
```

- Depuis la version 1.5, il est recommandé de spécifier la nature des objets stockés.
- **Exemple:**

```
ArrayList<Maclasse> liste = new
ArrayList<Maclasse> () ;
liste.add (new Maclasse ())
Maclasse obj = liste.get (0) ;
```

Les types génériques

- Les types génériques permettent de spécifier le type d'objets que l'on va placer dans une collection d'objets (List, Vector ou autre)
- Avantages:
  - meilleure lisibilité: on connaît à la lecture du programme quel type d'objets seront placés dans la collection.
  - La vérification peut être faiteà la compilation.
  - Le cast pour récupérer un objet de la collection est devenu implicite (sans cette fonctionnalité, il fallait faire un cast explicite, sachant que celui-ci peut échouer mais cela n'était détectable qu'à l'exécution)
- La syntaxe pour utiliser les types génériques utilise les symboles $<$ et $>$.

Exemple de type générique

```
import java.util.* ;
public class TestGenerique{
public static void main(String[ ] args) { new TestGenerique () ; }
public TestGenerique (){
String chaine,str ;
boolean bFinBoucle = false ;
List<String> liste = new ArrayList () ;
Scanner clavier = new Scanner (System.in) ;
while (bFinBoucle == false){
chaine = clavier.nextLine() ;
if (chaine.equalsIgnoreCase("quit") == false)
liste.add (chaine) ; // on ne peut stocker que des Strings
else bFinBoucle = true ;
}
for (Iterator<String> iter = liste.iterator (); iter.hasNext () ;)
{
str = iter.next () ; // Pas de cast ici
System.out.println (str) ;
}
}
}
```

Définition

- Une méthode peut être paramétrée avec des valeurs.
- La généricité permet de paramétrer du code (classe) avec des types de données.
- **Exemple :**

  class ArrayList<T>

- dont le code est paramétré par un type T
- Pour l'utiliser il faut passer un type en argument :
  new ArrayList<Employe>()

Pourquoi la généricité

- Une collection d'objets ne contient le plus souvent qu'un seul type d'objet : liste d'employés, liste de livres, liste de chaînes de caractères, etc.
- Mais sans la généricité les éléments des collections doivent être déclarés de type Object.
- Sans la généricité, il est impossible d'indiquer qu'une collection ne contient qu'un seul type d'objet, comme on le fait avec les tableaux (par exemple String[ ])

Exemple de code non générique

```
ArrayList employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
.  .  .  // On ajoute d'autres employes
for(int i = 0; i < employes.size(); i++) {
System.out.println(((Employe)employes.get(i)).getNom());
}
```

- Remarque que le casting est nécessaire pour pouvoir utiliser getNom().

```
// On ajoute un livre au milieu des employes
Livre livre = new Livre(...);
employes.add(livre);

for (int i = 0; i < employes.size(); i++) {
System.out.println(((Employe) employes.get(i)).getNom());
}
```

Conséquences

- Il est impossible d'indiquer qu'une liste ne contient que des instances d'un certain type.
- Certaines erreurs ne peuvent être repérées qu'à l'exécution et pas à la compilation.
- Il faut sans arrêt caster les éléments pour pouvoir utiliser les méthodes qui n'étaient pas dans `Object`.
- Si un objet contenu dans les collections n'est pas du type attendu, on a une erreur à l'exécution mais pas à la compilation.

Collections à type particulier

- Si on veut éviter ces problèmes (éviter les casts et vérifier les types à la compilation), il faut
  - écrire un code différent pour chaque collection d'un type particulier
  - Par exemple, il faut écrire une classe ListInteger et une classe ListEmploye.
  - Ce qui revient à écrire plusieurs fois la même chose, en changeant seulement les types.
  - Ou alors,écrire des codes génériques.

Généricité

- La généricité permet de paramétrer une classe ou interface avec un ou plusieurs types de données.
- On peut par exemple donner en paramètre le type des éléments d'un ArrayList : ArrayList<**E**>
- **E** est un paramétre qui sera remplacé par un argument de type pour typer des expressions ou créer des objets : ArrayList<Integer> l = new ArrayList<Integer>();

Extraits de la classe ArrayList

```
public class ArrayList<E> extends abstractList<E>
{
public ArrayList() // pas ArrayList<E>() !
public boolean add(E element)
public E get(int index)
public E set(int index, E element)
}
```

Utilisation d'un ArrayList paramétré

```
ArrayList<Employe> employes = new ArrayList<Employe>();

Employe e = new Employe("Dupond");
employes.add(e);
...  // On ajoute d'autres employes

for (int i = 0; i < employes.size(); i++)
{
System.out.println(employes.get(i).getNom());
}


//Erreur détectée à la compilation :
// Ajoute un livre au milieu des employes
Livre livre = new Livre(...);
employes.add(livre); // Erreur de compilation
```

Quelques définitions

- Type générique : une classe ou une interface paramétrée par une section de paramètres de la forme <T1, T2, . . ., Tn>.
- Les Ti sont les paramètres de type formels. Ils représentent des types inconnus au moment de la compilation du type générique.
- On place la liste des paramètres à la suite du nom de la classe ou de l'interface :
  List<E>
  Map<K,V>.

Où peuvent apparaître les paramètres de type ?

- Dans le code du type générique, les paramètres de type formels peuvent être utilisés comme les autres types pour **déclarer des variables, des paramètres, des tableaux ou des types retour de méthodes** :
  - `E element;`
  - `E[ ] elements;`

Où peuvent apparaître les paramètres de type ? (2)

A ne pas faire!!!

- Ils ne peuvent être utilisés pour créer des objets ou des tableaux, ni comme super-type d'un autre type :
  - pas de `new E()`
  - pas de `new E[10]`
  - pas de `class C extends E`
- On ne peut utiliser un paramètre de type à droite de `instanceof` : pas de x instanceof E

Les classes génériques - Exemple

```
class MaclasseGenerique<T1,T2>{
private T1 param1 ;
private T2 param2 ;
public MaclasseGenerique (T1 param1,T2 param2)
{
this.param1 = param1 ;
this.param2 = param2 ;
}
public T1 getParam1 () { return param1 ; }
public T2 getParam2 () { return param2 ; }
}
public class TestclasseGenerique{
public static void main(String[ ] args)
{
MaclasseGenerique<String,Integer> a=new
MaclasseGenerique<String,Integer> ("Dupont",33) ;
System.out.println("Nom :"+a.param1+", Age :"+a.param2 }
}
```

Types des arguments de type

- Les arguments de type peuvent être des classes, même abstraites, ou des interfaces ;
- **par exemple :** `new ArrayList<Comparable>`
- Ils peuvent même être des paramètres de type formels ;
- **par exemple :**

```
public class C<E>
{
...
f = new ArrayList<E>();
...
}
```

- Un argument de type ne peut pas être un type primitif.

Arguments de type abstraits

- On peut utiliser un argument de type abstrait (classe abstraite, interface) pour instancier une classe générique ;
- si `EmployeI` est une interface, **on peut écrire :**

```
List<EmployeI> l = new ArrayList<EmployeI>();
```

- évidemment, il faudra remplir cette liste avec des employés concrêts (les classes qui implémente l'interface EmployeI).

Création d'une instance de classe paramétrée

- Au moment de la création de l'objet paramétrée, on indique le type de la collection en donnant un argument de type pour chaque paramètre de type formel

```
ArrayList<String> liste = new
                          ArrayList<String>();

Map<String,Integer> map = new
                           Map<String,Integer>();
```

Utilisation des types génériques

- On peut utiliser les types génériques comme sur-type (classe mère ou interface).
- **Exemple :**

```
public class C<P> extends M<P>
```

- Sinon, on ne peut utiliser un paramètre de type que dans une classe générique paramétrée par ce paramètre de type.

Méthode générique

- Comme une classe ou une interface, une méthode (ou un constructeur) peut être paramétrée par un ou plusieurs types.

- Une méthode générique peut être incluse dans une classe non générique, ou dans une classe générique (si elle utilise un paramètre autre que les paramètres de type formels de la classe).

Syntaxe

- Une liste de paramètres apparaît dans l'en-tête de la méthode (juste avant le type de retour) pour indiquer que la méthode ou le constructeur dépend de types non connus au moment de l'écriture de la méthode : `<T1, T2, ... > ... m(...)`

- Exemple de l'interface Collection<E> :

```
public abstract <T> T[ ] toArray(T[ ] a)
```

Instanciation d'une méthode générique

- On peut appeler une méthode paramétrée en la préfixant par le (ou les) type qui doit remplacer le paramètre de type : `NomObjet.<String> NomMethode()`.

- Mais le plus souvent le compilateur peut faire une inférence de type ("deviner" le type) d'après le contexte d'appel de la méthode.

Inférence de type

- On appelle alors la méthode paramétrée sans la préfixer par une argument de type :

```
                    ArrayList<Personne> liste;
.   .   .
Employe[ ] res = liste.toArray(new Employe[0]);
// Inutile de préfixer par le type :
// liste.<Employe> toArray(...);
```

Inférence de type (2)

- Parfois, c'est un peu plus complexe pour le compilateur :

```
public <T> T choose(T a, T b) { ... }
...
Number n = choose(new Integer(0), new
Double(0.0));
```

- Le compilateur infère Number qui est la plus proche super-classe de `Integer` et `Double`.

Sous-typage et généricité

- Puisque que `ArrayList<E>` implémente `List<E>`, `ArrayList<E>` est un sous-type de `List<E>`.
- Ceci est vrai pour tous les types paramétrés construits à partir de ces 2 types.
- Par exemple, `ArrayList<Personne>` est un sous-type de `List<Personne>` ou `ArrayList<Integer>` est un sous-type de `List<Integer>`.

Sous-typage pour les types paramétrés

- **Mais attention :** si B hérite de A, les classes ArrayList<B> et ArrayList<A> n'ont aucun lien de sous-typage entre elles.

- **Exemple :** ArrayList<Integer> n'est pas un sous-type de ArrayList<Number>! ! (Number est la classe mère de Integer).

Raison

- Si ArrayList<B>était un sous-type de ArrayList<A>, le code suivant compilerait :
  // ce code ne compile pas!!
  ```
  ArrayList<A> la = new ArrayList<B>;
  la.add(new A());
  ```
- Quel serait le problème ?
- Ce code autoriserait l'ajout dans un ArrayList<B> un élément qui n'est pas un B !

Conséquences

- Une méthode `sort(ArrayList<Number> aln)` qui trie un `ArrayList<Number>` ne peut être appliquée à un `ArrayList<Integer>`.

- De même, l'instruction suivante est interdite :

```
ArrayList<Number> l = new ArrayList<Integer>();
```

Exemple de problème

```
public static void printElements(ArrayList<Number>
liste) {
for(Number n :  liste) {
System.out.println(n.intValue());
}
}
ArrayList<Double> liste = new ArrayList<Double>();
...
printElements(liste); // Ne compile pas !
```

Types Joker

- Un type Joker sans limite: <?>
- Un type Joker limité aux sous-classes d'une classe <? extends Animal> ou aux classes qui implémentent une interface <? extends Comparable>
- Un type Joker limité aux classes ayant une certaine sous-classe: <? super Integer>
- Exemple d'un attribut : ArrayList <? extends Vehicule> liste;

Méthodes ayant un nombre variable de paramètres de même type

Exemple.java

```
public class Exemple.java {
static void ecrireLesMots(String ... mots) {
for (String mot : mots) // mots est envisagé comme un tableau
System.out.print(mot + " ");
}
public static void main (String [] args) {
ecrireMots(" bonjour ", " le ", " monde ");
}
}
```