Quelques classes remarquables de JAVA

Emmanuel ADAM

INSA HdF

La classe Object

Il s'agit de la classe principale, elle contient les fonctions :

`protected Object clone()` qui crée et retourne une copie de l'objet

`boolean equals(Object obj)` qui détermine si obj est égal à l'objet courant.

`String toString()` retourne une chaîne représentant l'objet

`protected void finalize()` appelé par le ramasse miettes s'il n'y a plus de référence à l'objet

`Class getClass()` retourne la classe courante de l'objet

`int hashCode()` retourne une clé pouvant être utilisée pour un tri

`void notify()` réveille un processus en attente sur l'objet

`void notifyAll()` réveille tous les processus en attente

`void wait()` met en pause le processus courant en attendant un réveil

Tous les objets Java héritent donc de ces méthodes

Méthodes Object : clone()

protected Object clone() est donnée aux fonctions qui retourne une copie de l'objet

Syntaxe générale, qui recopie tous les champs :

```
public class Personne implements Cloneable{
 ...
 public Personne clone() {
  Object leClone =null;
  try { leClone =  super.clone();}
  catch(CloneNotSupportedException e) {e.printStackTrace();}
  return (Personne)leClone;
 }
}
```

Bien sûr vous pouvez remplacer ce code par celui de votre choix pour maîtriser ce qui est recopié ou non

Méthodes Object : clone()

Exemple de clonage :

```
Personne p1 = new Personne("Berlin");
// p2 est un clone de p1
Personne p2 = p1.clone();
System.out.println(p1 + "-" +p2);      // Berlin – Berlin
p2.prenom = "Tokio";
System.out.println(p1 + "-" +p2);      // Berlin – Tokio
```

Exemple de classe

```
class Personne
{
 String prenom;    int age = -1;
 // constructeur par défaut
 Personne( ){prenom="";}    // constructeurs avec paramètres
 Personne(String prenom)    { this.prenom = prenom; }
 Personne(String prenom, int age)
  {  this(prenom) ; this.age = age; }


 // surcharge de la méthode héritée de Object
 public String toString() {
   String retour = (prenom!=null?(prenom+ ", « ):"");
   if(age>=0) retour = retour + " age : " + age;
   return retour;
 }
}
```

Exemple de classe
Getter & Setter

Par défaut, il vaut mieux protéger les attributs et les rendre accessibles par des méthodes publiques getXX() & setXX(..). Ce sont les « getters et setters »

```
class Personne
{
   String prenom;   int age = -1;
  ……
  // surcharge de la méthode héritée de Object
   public String getPrenom(){return this.prenom;}
   public void setPrenom(String prenom){this.prenom = prenom;}
   public int getAge(){return this.age;}
   public void setAge(int age){this.age = age;}
}
```

Leurs écritures nécessaires mais laborieuses sont heureusement réalisées à la demande par les principaux IDE (Eclipse, IntelliJ, NetBeans, ..)

Méthodes Object : equals(..)

`boolean equals(Object obj)` détermine si obj est égal à l'objet courant.

`equals` est une fonction qui doit être :

Reflexive : `a.equals(a)`

Symétrique : `a.equals(b) <=> b.equals(a)`

Transitive : `a.equals(b) ET b.equals(c) => a.equals(c)`

Méthodes Object : equals(..)

```
public class Personne {
  String prenom;
  int age ;
    ……
  public boolean equals(Object o) {
    if(this == o) return true;
   //si l'objet passé est différent de null et est de type Personne
    if(o==null || (o.getClass() == Personne.class)) return false;
    //caster l'objet en tant que Personne
    Personne p = (Personne) o;
    //tester l'age en 1er car test plus rapide
    boolean rep = (age == p.age);
     //si le prenom est défini et est égal au prenom de l'autre
         // ou si les deux prenoms sont nuls
    rep = rep && (prenom!=null)?(prenom.equals(p.prenom)):(p.prenom==null);
    return rep;
  }

}
```

Comparer des objets (1/3)

`int compareTo(T other)` compare other à l'objet courant et retourne un entier :

- **< 0** : si this < other (si l'objet courant doit être situé avant l'objet other lors d'un tri)
- **==0** : si this == other
- **> 0** : si this > other (si l'objet courant doit être situé après l'objet other lors d'un tri)

`compareTo` est prédéfinie pour les classes de bases (Double, Integer, …, String)

Comparer des objets (2/3)

```
/**Ajout de la notion de comparaison à la classe Personne */
class Personne implements Cloneable, Comparable<Personne>
{
    String prenom;
    int age;    ...

/**retourne <0 si objet < autre, 0 si objet == autre,
>0 si objet > autre, ici tri par age croissant*/
 public int compareTo(Personne other)
 { int retour = 0;
   // les pointeurs null seront en fin de tableau/liste
   if(other==null) retour = -1;
   // sinon, si je suis plus age que other, j'envoie un nb positif;
   // si je suis plus jeune, j'envoie un nb négatif;
   // si on a le meme age, je retourne 0
   else retour = age - other.age;
  }
  return retour;
 }
}
```

Comparer des objets (3/3)

```
/**tri par nom, prénom puis age croissant*/
 public int compareTo(Personne other)
 { int retour = 0;
   // les pointeurs null seront en fin de tableau/liste
    if(other==null) retour = -1;
    else
    {// on compare les prenoms (méthode existante dans String)
     retour = prenom.compareTo(other.prenom);
    // si prenoms identiques, on compare les ages
     if( retour == 0) retour = age - other.age;
    }
  }
  return retour;
 }
}
```

La classe Objects

Classe contenant des fonctions statiques :

`boolean` **`equals`**`(Object obj1, Object obj2)` qui détermine si obj1 est égal à obj2

`boolean` **`deepEquals`**`(Object obj1, Object obj2)` qui détermine si obj1 et obj2 sont égaux en (si tableaux, application de `equals` à leurs éléments, sinon équivalent à `equals`)

`boolean` **`compare`**`(T obj1, T obj2, Comparator<T> comp)` qui compare obj1 par rapport à obj2 (de type T) selon le comparateur fourni

`String` **`toString`**`(Object o)` retourne une chaîne représentant l'objet o

`String` **`toString`**`(Object o, String siNull)` retourne une chaîne représentant l'objet, ou la chaîne *`siNull`* si l'objet est nul

`boolean` **`isNull`**`(Object o)` retourne si o est une référence nulle

`boolean` **`isNonNull`**`(Object o)` retourne si o est une référence non nulle

`void` **`requireNonNull`**`(Object o, String msg)` déclenche une erreur msg si o est une référence nulle

`T` **`requireNonNullElse`**`(T o, T other)` retourne l'objet o si non nul, sinon l'objet other

Tous les objets Java héritent donc de ces méthodes

Utilisation de Objects

```
int[] tab1 = {1,2,3,4,5,6};
int[] tab2 = {1,2,3,4,5,6};
System.out.println(Objects.equals(tab1, tab2));
// -> false
System.out.println(Objects.deepEquals(tab1, tab2));
// -> true

Personne p1 = new Personne("anna");
Personne p2 = null;
Personne remplacant = new Personne("individu");

Personne p = Objects.requireNonNullElse(p1, remplacant);
System.out.println(p);
// -> anna

p = Objects.requireNonNullElse(p2, remplacant);
System.out.println(p);
// -> individu
```

Types Génériques (1/2)

Utilisation de caractères remplaçant un type défini à l'exécution.

Exemple avec les méthodes :

```
//affiche les objets d'un tableau de …
<T>void affiche(T[] tab)
{
  for (Object o : tab) System.out.println(o.toString());
}
```

On indique ici, avant sa description, que la méthode utilise un type générique nommé ici T

Types Génériques (2/2)

Possibilité d'utiliser plusieurs types génériques :

```
//T et V sont des types génériques
<T, V>void afficheEtVal(T[] tab, V v)
{
  for (Object o : tab) System.out.println(o);
  System.out.println("valeur = " + v.toString());
}
```

Possibilité de préciser le type :

```
//T doit être un type comparable
<T extends Comparable<T>> void compare(T a, T b)
{
  int comp = a.compareTo(b);
  if(comp<0) System.out.println(a + " est plus petit que " + b);
  if(comp==0) System.out.println(a + " est égal à " + b);
  if(comp>0) System.out.println(a + " est plus grand que " + b);
}
```

Classe Génériques (1/2)

Exemple de classe utilisant deux types génériques pour la définition d'un couple :

```
public class Couple<T1, T2> {
  T1 v1;
  T2 v2;
  Couple(){}
  Couple(T1 v1, T2 v2){this.v1 = v1; this.v2 = v2;}
  public T1 getV1(){return v1;}
  public T2 getV2(){return v2;}
  public void setV1(T1 v1){this.v1 = v1;}
  public void setV2(T2 v2){this.v2 = v2;}
 public String toString() {
   return("(" + v1.toString() + "||" + v2.toString() + ")"); }
}
```

Classe Génériques (2/2)

Utilisation d'une classe utilisant des listes génériques :

```
Personne p = new Personne();
Hamster h = new Hamster();
Couple<Personne, Hamster> couple = new  Couple<>(p,h);
System.out.println(couple);

Ecriture simplifiée possible depuis java 1.10 :
var couple = new  Couple<>(p,h);
```

La classe System (1/2)

Gestion du système :

- `static PrintStream` **`err`** : sortie d'erreur standard
- `static InputStream` **`in`** : entrée standard
- `static PrintStream` **`out`** : sortie standard
- `static void` **`arraycopy`**`(...)` : copie de tableaux
- `static long` **`currentTimeMillis`**`()` : temps courant en millisecondes
- `static long` **`nanoTime`**`()` : temps courant en nanoseconde
- `static void` **`exit`**`(int status)` : sortie de programme
- `static void` **`gc`**`()` : lance le ramasse-miettes
- `static void` **`load`**`(String fichier)` : charge le code en tant que librairie dynamique

La classe System (2/2)

Gestion des propriétés du système :

- `static String` **`getProperty`**`(String key)`:
  retourne la valeur de la propriété spécifiée par la clé

- `static Properties` **`getProperties`**`()`:
  retourne les propriétés du système

- `static String` **`setProperty`**`(String key, String value)` :
  affecte une nouvelle valeur à une propriété

Exemples de propriétés : *user.home* : répertoire de l'utilisateur, *java.class.path* : valeur du classpath, symbole séparateur de ligne, de fichiers, de répertoire, répertoire temporaire, langage, ….

La classe String (1/3)

Les chaînes sont constantes, leurs valeurs ne peuvent être changées après leurs créations.

StringBuffer et StringBuilder(non synchronisée) permettent l'utilisation de chaînes "dynamiques".

Construction : `String str = "abc";` **est équivalent à** `String str = new String("abc");`

La classe ***String*** comporte des méthodes d'accès aux caractères, de comparaisons, de recherche, d'extraction, de copie, de conversion minuscules/majuscule, ...

La classe String (2/3)

L'opérateur `(+)` est surchargé pour permettre la concaténation

Toute conversion se fait de manière automatique en faisant appel à la méthode **`toString()`** héritée de la classe **`Object`**.

Par défaut, **`toString()`** retourne le nom de la classe de l'objet et son adresse en mémoire virtuelle : `td1.Personne@3d4eac69`

Il est possible de surcharger cette méthode dans les nouvelles classes créées.

La classe String (3/3)

String permet de tronçonner une chaîne :

```
String ch = "une chaîne à tronçonner »;

String[]mots1 = ch.split(" ");

String[]mots2 = ch.split("n");

String[]mots3 = ch.split("n",3);
```

```
["une";"chaîne";"à";"tronçonner"]
["u";"e chaî";"e à tro";"ço";"";"er"]
["u";"e chaî";"e à tronçonner"]
```

Les classes StringBuffer et StringBuilder

A utiliser si besoin de chaînes dynamiques (modifications, concaténation, …)
StringBuilder n'est pas synchronisée au contraire de StringBuffer

```
int taille = 60000;
long momentDebut = System.currentTimeMillis();
//creation d'une chaine ch = 0, 1, …., 59999
String ch = new String();
for(int i=0; i<taille; i++)  ch += i + ", ";
System.out.println(ch);
long momentFin  = System.currentTimeMillis();
System.out.println("temps écoulé = "+ (momentFin – momentDebut));
                                  ->  3356 ms = 3 secondes
momentDebut = System.currentTimeMillis();
StringBuilder sb = new StringBuilder();
String sep = ", ";
for(int i=0; i<taille; i++) sb.append(i).append(sep);
System.out.println(sb);
momentFin  = System.currentTimeMillis();
System.out.println("temps écoulé = "+ (momentFin – momentDebut));
                                          ->  0 ms  !!!!
```

Retour sur Exemple de classe (amélioration de toString)

```
class Personne
{
    …

   // amélioration de toString()
    public String toString()
    {
     StringBuilder retour = new StringBuilder();
     if (prenom != null) retour.append(prenom).append(", ");
     if (age>0) retour.append("age = ").append(age);
     return retour.toString();
    }
}
```

Des classes très utiles (import java.util.*)

Interfaces : **Collection**, **Comparator**, *Enumeration*, EventListener, *Iterator*, **List**, ListIterator, Map, Map.Entry,Observer, **Set**, SortedMap, SortedSet

Classes : AbstractCollection, AbstractList, AbstractMap, AbstractSequentialList, AbstractSet, **ArrayList**, **Arrays**, BitSet, Calendar, **Collections**, Date, Dictionary, EventObject, GregorianCalendar, HashMap, HashSet, **Hashtable**, LinkedList, ListResourceBundle, Locale, **Observable**, Properties, PropertyPermission, PropertyResourceBundle, **Random**, ResourceBundle, SimpleTimeZone, Stack, StringTokenizer, TimeZone, TreeMap, **TreeSet**, Vector, WeakHashMap

La classe Arrays

Cette classe contient des méthodes statiques pour la gestion de tableaux :
(remarque, les fonctions présentées existent pour tous les types)

`static int` **`binarySearch`**`(int []tab, int valeur)` retourne l'index de la valeur, -1 si introuvable

`static boolean` **`equals`** `(boolean []tab1, boolean []tab2)` teste l'égalité de deux tableaux

`static boolean` **`deepEquals`** `(Object []tab1, Object []tab2)` *teste l'égalité de deux tableaux récursivement (`deepEquals` effectue un appel à elle même si tab1 et tab2 contiennent des tableaux)*

`static void` **`fill`** `(double []tab, double valeur)` remplit le tableau avec la valeur

`static void` **`copyOf`** `(long []tab, int taille)` retourne une copie du tableau tab

La classe Arrays

Exemple d'autres méthodes statiques pour la gestion de tableaux :

`static void` **`sort`**`(long []tab)` : *trie le tableau*

`static void` **`sort`**`(Object [] tab)` : *trie le tableau si les objets contenus sont comparables*

`static String` **`toString`**`(int []tab)` : *retourne un chaine contenant l'ensembles des valeurs entièresfonctionne pour toutes primitives et également les objets*

`static String` **`deepToString`**`(Object []tab)` : *retourne un chaine résultant de la concaténation des appels à deepToString() pour chaque élément*

`static <T> List<T>` **`asList`**`(T... tab)` : *retourne un liste dynamique à partir du tableau tab (T remplace tout type d'objets)*

La classe Arrays, ajouts depuis 1.8

`static void parallelSort(int []tab)` utilise le parallélisme pour le tableau qui est découpé en parties assez courtes pour être triées par un 'sort classique'

Ex : 1 millions d'entiers triés en 65 ms en //, contre 150 ms

10 millions d'entiers triés en 440 ms en //, contre 1000 ms

`static boolean setAll(double[] array, IntFunction<? extends T> generator)` utilise un **générateur** prenant l'index d'un élément du tableau et retournant la valeur correspondante.

```
double[] tab = new double[6];
Arrays.setAll(tab, i -> (2d*i));
System.out.println(Arrays.toString(tab))
                                    ⇒ [0.0,2.0,4.0,6.0,8.0,10.0]
```

La classe Arrays, ajouts depuis 1.8

`static <T> void sort(T[] a, Comparator<? super T> c)` *: trie les éléments du tableau en utilisant le comparateur passé en paramètre*

Comme pour la fonction compareTo, le comparateur prend 2 entrées a et b et retourne <0 si doit être placé avant b dans le tri, 0 si a et b se valent et >0 si a doit être placé après b dans le tri:

```
Personne[] tab = new Personne[taille];
Random hasard = new Random();
Arrays.setAll(tab,  i-> new Personne("p"+hasard.nextInt(taille)));
System.out.println(Arrays.deepToString(tab));
//tri par prénom en ordre décroissant
Arrays.sort(tab, (p1, p2)-> (-p1.prenom.compareTo(p2.prenom)));
System.out.println(Arrays.deepToString(tab));
```

La classe Arrays, ajouts depuis 1.8

`static <T> void` **`parallelSetAll`**`(T[] array,` **`IntFunction`**`<? extends T> generator)` **:** *initialise en parallèle les éléments du tableau à l'aide d'un générateur*

`static <T> void` **`parralelSort`**`(T[] a,` **`Comparator`**`<? super T> c)` *: trie en parallèle les éléments du tableau en utilisant le comparateur passé en paramètre*

La classe Arrays

```
Random hasard = new Random();
int[] tab = new int[taille];
Arrays.setAll(tab, i -> hasard.nextInt(taille));

if(taille<101)
 System.out.println("tab="+ Arrays.toString(tab));

long momentDebut = System.currentTimeMillis();
java.util.Arrays.sort(tab);
long momentFin = System.currentTimeMillis();

System.out.println("tri en : " + (momentFin -
momentDebut) + " millisecondes");

if(taille<101)
System.out.println("tab="+ Arrays.toString(tab));
```

Si taille = 100==>

```
[82, 9, 63, 49, 13, 89, 89, 15, 36, 65, 98,
41, 23, 96, 84, 4, 86, 27, 98, 41, 79, 64,
99, 50, 61, 33, 72, 19, 48, 71, 70, 20, 14,
34, 99, 44, 12, 92, 15, 59, 20, 72, 81, 39,
32, 61, 70, 85, 65, 11, 18, 76, 56, 14, 39,
28, 98, 90, 61, 8, 60, 1, 53, 32, 29, 80, 55,
13, 10, 37, 53, 84, 2, 57, 90, 73, 7, 52, 27,
41, 88, 17, 25, 21, 45, 44, 76, 57, 57, 4,
35, 40, 72, 51, 85, 49, 34, 35, 9, 74]

[1, 2, 4, 4, 7, 8, 9, 9, 10, 11, 12, 13, 13,
14, 14, 15, 15, 17, 18, 19, 20, 20, 21, 23,
25, 27, 27, 28, 29, 32, 32, 33, 34, 34, 35,
35, 36, 37, 39, 39, 40, 41, 41, 41, 44, 44,
45, 48, 49, 49, 50, 51, 52, 53, 53, 55, 56,
57, 57, 57, 59, 60, 61, 61, 61, 63, 64, 65,
65, 70, 70, 71, 72, 72, 72, 73, 74, 76, 76,
79, 80, 81, 82, 84, 84, 85, 85, 86, 88, 89,
89, 90, 90, 92, 96, 98, 98, 98, 99, 99]

tri effectue en 0 millisecondes
```

Programmation fonctionnelle (depuis java 8)
Consumer, Supplier, Function

Java 8 introduit la programmation fonctionnelle et la notation lambda. Parmi les interfaces ajoutées :

- **Function<T,R>** pour une fonction qui accepte un paramètre de type T et retourne un résultat de type R
- **BiFunction<T,U,R>** pour une fonction qui accepte un paramètre de type T, un paramètre de type U et retourne un résultat de type R
- **Consumer<T>** pour une fonction qui accepte un paramètre de type T et ne retourne rien
- **Supplier<R>** pour une fonction qui produit un résultat de type R
- **Predicat<T>** pour une fonction qui accepte un paramètre de type T et retourne un booléen
- **Comparator<T, T>** pour une fonction qui accepte deux paramètres en entrée et retourne un entier négatif, positif ou nul selon la comparaison entre les paramètres

Programmation fonctionnelle (depuis java 8)
Function

Une objet de type **Function<T,R>** est une fonction prenant une valeur de type T en entrée et produisant une valeur de type R. Cette objet possède la fonction apply().

Fonction prenant un int et retournant un Double entre 0 et cet entier (utilisation ici de IntFunction) :

```
//notation 1
  IntFunction<Double> f = (int i) ->
              {double r = Math.random() * i;  return r;};

//notation 2 : les types d'entrée et de sortie étant connus, on ne les indique pas
  IntFunction<Double> f = (i) -> {return Math.random() * i;};

//notation 3 : le bloc retourne forcément un double, on peut ici retirer le mot clé return
  IntFunction<Double> f = (i -> (Math.random() * i));

 //Exemple d'utilisation :
    Double tirage = f.apply(10);
    Double[] tab = new Double[taille];
    Arrays.setAll(tab, f);

//Exemple définissant la fonction lors du passage en paramètre :
  Arrays.setAll(tab, i -> (Math.random() * 10));
```

Programmation fonctionnelle (depuis java 8)
BiFunction

Une objet de type **BiFunction<T,U,R>** est une fonction prenant deux valeurs en entrée, la première de type T, la seconde de type U, et produisant une valeur de type R. Cette objet possède la fonction apply().

Fonction prenant en paramètre trois tableaux et une BiFunction:

```
void mapTab(int[]tab1, int[]tab2, double[]tab3, BiFunction<Integer,Integer,Double>f)
{
  int nb = Math.min(tab1.length,Math.min(tab2.length,tab3.length));
  for(int i=0; i<nb; i++)
    tab3[i] = f.apply(tab1[i], tab2[i]);
}
```

Utilisation:

```
int[]tabA = {2,5,4,3,1,7,8};
int[]tabB = {5,4,8, 2};
double[]tabC = new double[10];
// on souhaite que c contienne les valeurs de tabA multipliées par celles de tabB
mapTab(tabA, tabB, tabC, (a,b)->(double)a*b);

System.out.println(Arrays.toString(tabC));
```

Programmation fonctionnelle (depuis java 8)
Consumer

Un Consumer ne produit rien. Mais il peut modifier l'objet auquel il s'applique

Affichage d'un tableau de réels en passant par son flux:

```
//notation
    Arrays.stream(tab).forEach(d-> System.out.printf("%.3f ; ", d ));
```

Veillissement de 10 ans d'un tableau de personnes:

```
//notation
    Arrays.stream(tab).forEach(p-> p.age+=10);
```

Programmation fonctionnelle (depuis java 8)

Predicat

Un Predicat, unaire, vérifie si un objet vérifie une condition.

Verifier si toutes les personnes d'un tableau sont majeures :

```
//notation
    boolean majeures = Arrays.stream(tab).allMatch(p-> (p.age>=18));
```

Verifier s'il existe au moins une personne de plus de 10 ans

```
//notation
    Arrays.stream(tab).anyMatch(p-> (p.age>=10));
```

Programmation fonctionnelle (depuis java 8)

Comparator (1/3)

Un Comparator représente une fonction binaire qui, à partir de deux éléments a et b d'un même type retourne un entier :

- Négatif si a<b (si a doit être classé avant b)
- Positif si a>b (si a doit être classé après b)
- Nul si a = b (aucune préférence sur le classement de a par rapport à b)

Trier un tableau de personnes par age décroissant :

```
//notation
    Arrays.sort(tab, (p1, p2) -> (p2.age-p1.age));
```

Programmation fonctionnelle (depuis java 8)

Comparator *(2/3)*

L'interface **Comparator** est une **BiFunction** permet la combinaison de comparaisons

Trier un tableau de personnes par prénom :

```
//notation
    Comparator<Personne> comp1 = Comparator.comparing(p->p.getPrenom());
    Arrays.sort(tab, comp1);
```

Trier un tableau de personnes par age décroissant :

```
//notation
    Comparator<Personne> comp2 = Comparator.comparingInt(p->p.getAge());
    Arrays.sort(tab, comp2.reversed());
```

Trier un tableau de personnes par prénom décroissant, puis par age :

```
//notation
    Arrays.sort(tab, comp1.reversed().thenComparing(comp2));
```

Programmation fonctionnelle (depuis java 8)

Comparator *(3/3)*

L'utilisation d'une variable est inutile dans la notation, on la simplifie ainsi :

Trier un tableau de personnes par prénom décroissant, puis par age:

```
//notation
    Comparator<Personne> comp1 =
        Comparator.comparing(Personne::getPrenom);

    Comparator<Personne> comp2 =
        Comparator.comparingInt(Personne::getAge);

    Arrays.sort(tab, comp1.reversed().thenComparing(comp2));

//notation en ligne
    Arrays.sort(tab, Comparator.comparing(Personne::getPrenom).reversed().
                thenComparing(Comparator.comparingInt(Personne::getAge)));
```

Interface Collection (1/2)

Une Collection est une suite d'éléments, ordonnés ou non, uniques ou non.
La Collection est généralement utilisé pour des ensembles de taille variables
Toute Collection possède ces méthodes :

`boolean` **`add`**`(E e)` : ajoute un élément dans la collection
`boolean` **`addAll`**`(Collection<? extends E> c)` : ajoute une collection dans la collection
`void` **`clear`**`()` : retire tous les éléments de la collection
`boolean` **`contains`**`(Object o)` : retourne si un objet existe ou non dans la collection
`boolean` **`containsAll`**`(Collection<?> c)` : retourne si une collection appartient à la collection
`boolean` **`isEmpty`**`()` : retourne vrai si la collection est vide
`boolean` **`remove`**`(Object o)` : retire un objet de la collections si présent
`boolean` **`removeAll`**`(Collection<?> c)` : retire les éléments d'une collection présent dans la collection
`int` **`size`**`()` : retourne la taille de la collection
`Object[]` **`toArray`**`()` : Retourne un tableau contenant les éléments de la collection
`<T> T[]` **`toArray`**`(T[] tab)` : Retourne un tableau contenant les éléments de la collection

Interface Collection (2/2)

Depuis Java 8, quelques méthodes sont ajoutées :

`default Stream<E> parallelStream()` : retourne si possible un flux parallèle

`default boolean removeIf(Predicate<? super E> filter)` : retire les éléments qui satisfont le prédicat unaire

`boolean retainAll(Collection<?> c)` : ne garde que les éléments qui satisfont le prédicat unaire

`default Stream<E> stream()` : Retourne un flux séquentiel à partir de la collection

default void **forEach**(Consumer<? super T> action) : effectue une action de type Consumer sur chaque élément de la liste

Interface List (1/2)

Un objet de type List est une Collection d'éléments **ordonnés**, uniques ou non.

Toute List possède en plus méthodes :

`boolean` **`add`**`(int index, E e)` : **insère** l'élément e dans la position index

`boolean` **`addAll`**`(int index, Collection<? extends E> c)` : **insère** la collection c dans la collection à partir de l'index

`int` **`indexOf`** `(Object o)` : retourne l'index de l'élément dans la liste (-1 si non présent)

`int` **`lastIndexOf`** `(Object o)` : retourne la dernière position de l'élément dans la liste (-1 si non présent)

`boolean` **`remove`**`(int index)` : retire l'élément à l'index donné

`boolean` **`set`**`(int index, E e)` : **remplace** l'élément dans la position index par e

`boolean` **`subList`**`(int from, int to)` : retourne une vue de la liste entre les index from et to exclusif

Interface List (2/2)

Depuis Java 8, quelques méthodes sont ajoutées :

`void sort(Comparator<? super E> c)`: trie les éléments de la liste selon le comparator
`default void replaceAll(UnaryOperator<E> operator)` : remplace chaque élément de la liste en appliquant l'opérateur unaire

`static <E> List<E> copyOf(Collection<? extends E> coll)` : retourne copie **immuable** d'une collection

`static <E> List<E> of(E e1)` : retourne une liste **immuable** contenant un élément
`static <E> List<E> of(E... elements)` : retourne une liste **immuable** contenant les éléments

*Remarque : il n'est pas possible d'ajouter, de retirer des objets d'une liste **immuable** , ni de remplacer ses objects par d'autres.*
(par contre, les caractéristiques accessibles et non constantes des objets peuvent être modifiées)

Classe ArrayList (1/4)

La classe **`ArrayList`** est une implémentation de l'interface List.
Elle permet de stocker, trier, modifier des objets dans un ensemble de longueur variable. Non synchronisée, elle est majoritairement utilisée.

Exemples d'union et d'intersection (1/2)….

```
Deux listes immuables d'entiers
 List<Integer>  liste1 = List.of(1, 2, 3, 4, 5);
 List<Integer>  liste2 = List.of(4, 5, 6, 7, 8);
Une liste modifiable, contenant une copie de la liste 1
 ArrayList<Integer>  liste3 = new ArrayList<>(liste1);
Ajout de toute la liste 2
 liste3.addAll(liste2);
 System.out.println("union : " + liste3); ->[1, 2, 3, 4, 5, 4, 5, 6, 7, 8]
```

Classe ArrayList (2/4)

Exemples d'union et d'intersection (2/2)....

```
Une liste modifiable, contenant une copie de la liste 1
 liste3 = new ArrayList<>(liste1);
Retrait des éléments communs à la liste 2
 liste3.removeAll(liste2);
Ajout de toute la liste 2
 liste3.addAll(liste2);
 System.out.println("union disjointe : "+ liste3);->[1, 2, 3, 4, 5, 6, 7, 8]
Une liste modifiable, contenant une copie de la liste 1
 liste3 = new ArrayList<>(liste1);
Ne retenir que les éléments communs à la liste 2
 liste3.retainAll(liste2);
 System.out.println("intersection : " + liste3);->[4, 5]
```

Classe ArrayList (3/4)

```
Random hasard= new Random();
ArrayList<Personne> liste = new ArrayList<>();
for (int i = 0; i < 10; i++)
  liste.add(new Personne("p" + i, hasard.nextInt(100)));

liste.forEach(p->System.out.println(p));
```

Raccourcis de notation pour l'affichage, comme p est passé sans modification :

```
liste.forEach(System.out::println);
```

Ne garder que les personnes majeures

```
liste.removeIf(p->p.age<18);

liste.forEach(System.out::println);
```

Classe ArrayList (4/4)
COPIE DE LISTE

Attention à la copie de références et non d'objets !

```
List<Personne> liste1 = new ArrayList<>(List.of(new
Personne("anton", 17), new Personne("paula", 19));
List<Personne> liste2 = new ArrayList<>(liste1);
liste2.get(0).age = 33;
System.out.println(liste1.get(0));    ——> anton, age = 33
```

```
List<Personne> liste2 = new ArrayList<>(List.copyOf(liste1));
liste2.get(0).age = 45;
System.out.println(liste1.get(0));    ——> anton, age = 45
```

```
List<Personne> liste2 = new ArrayList<>();
for(Personne p:liste1) liste2.add(p.clone());
liste2.get(0).age = 27;
System.out.println(liste1.get(0));    ——> anton, age = 17
```

LinkedList

La classe **`LinkedList`** permet l'utilisation en tant que FIFO (First In First Out) ou LIFO (Last In First Out) :

`push(objet)` : ajoute un objet sur la pile ;

`pop(), pollLast()` : retirent l'objet du dessus de la pile;

`remove(), poll(), pollFirst()` : retirent l'objet en bas de la pile;

`peek()` : renvoie une référence à l'objet du bas de la pile sans le retirer ;

`peekLast()` : renvoie une référence à l'objet du haut de la pile sans le retirer ;

`isEmpty()` : teste si la pile est vide.

TreeSet

La classe `TreeSet` définit un ensemble ne pouvant contenir qu'un exemplaire d'un objet.

Quelques méthodes :

`add(objet)` qui n'ajoute l'objet que s'il n'est pas déjà présent

`contains(objet)` ;

`remove(objet)` ;

`ceiling(objet)` : retourne le plus petit élément de l'ensemble plus grand ou égal à l'objet

`floor(objet)` : retourne le plus grand élément de l'ensemble plus petit ou égal à l'objet

Classe Collection**S**

Cette classe contient des méthodes statiques, dont :

`boolean` **`addAll`**`(Collection<E> c, E… tab)` : ajoute le contenu du tableau dans la collection c

`int` **`binarySearch`** `(List<T> list, T key)` : recherche la clé dans une liste d'objets comparables, retourne son index

`int` **`binarySearch`** `(List<T> list,` **`Comparator`**`<T> c, T key)` : recherche la clé dans une liste d'objets à l'aide du comparator c, retourne son index

`int` **`sort`** `(List<T> list)` : trie une liste d'objets comparables

`int` **`sort`** `(List<T> list,` **`Comparator`**`<T> c)` : trie une liste d'objets à l'aide du comparator c

`T` **`max`**`(List<T> list)` : retourne l'élément le plus grand d'une liste d'objets comparables

`T` **`max`**`(List<T> list,` **`Comparator`**`<T> c)` : retourne l'élément le plus grand d'une liste d'objets en se basant sur le Comparateur c

`T` **`min`**`(List<T> list)` : retourne l'élément le plus petit d'une liste d'objets comparables

`T` **`min`**`(List<T> list,` **`Comparator`**`<T> c)` : retourne l'élément le plus petit d'une liste d'objets en se basant sur le Comparateur c

`void` **`reverse`**`(List<?> l)` : renverse les éléments de la liste l

`void` **`rotate`**`(List<?> l, int dist)` : effectue une rotation des éléments de la liste l, sur une distance dist

`void` **`shuffle`**`(List<?> l)` : mélange les éléments de la liste l

Des méthodes permettent également la création de liste synchronisées et de liste immuables

Classe Collection**S**

Trouver la personne la plus jeune et la personne la plus âgée :

```
Random hasard= new Random();
ArrayList<Personne> liste = new ArrayList<>();
for (int i = 0; i < 10; i++)
  liste.add(new Personne("p" + i, hasard.nextInt(100)));

Personne jeune = Collections.min(liste,
                          Comparator.comparingInt(Personne::getAge));
System.out.println("-personne la plus jeune : " +  jeune);

Personne agee = Collections.max(liste,
                           Comparator.comparingInt(Personne::getAge));
System.out.println("-personne la plus agee : " +  agee);
```

Interface Map (1/2)

Une Map lie une clé à exactement une valeur. Une clé ne peut être dupliquée dans une Map.
Toute Map possède, entres autres, ces méthodes :

`void` **`clear`**`()` : retire tous les éléments de la map
`boolean` **`containsKey`**`(Object key)` : retourne si une clé existe ou non dans la collection
`boolean` **`containsValue`**`(Object v)` : retourne si une valeur existe ou non dans la collection
`boolean` **`get`**`(Object key)` : retourne la valeur associée à la clé, ou null
`boolean` **`isEmpty`**`()` : retourne vrai si la map est vide
`V` **`put`**`(K key, V value)` : lie une valeur à une clé, retourne la valeur précédemment associée
`void` **`putAll`**`(Map<K,V> map)` : copie la map passée en paramètre à la map en cours
`V` **`remove`**`(Object key)` : retire une clé et son association. retourne la valeur associée
`boolean` **`remove`**`(Object key, Object value)` : retire l'association, si elle existe
`V` **`replace`**`(K key, V value)` : lie une nouvelle valeur à une clé *existante*, retourne la valeur précédemment associée
`int` **`size`**`()` : retourne la taille de la collection
`Collection<V>` **`values`**`()` : Retourne les valeurs de la map
`Set<K>` **`keySet`**`()` : Retourne les clés de la map dans un ensemble

Interface Map (2/2)

Quelques ajouts depuis Java 8

`V` **`compute`**`(K key, BiFunction<K,V, V> remappingFunction)` : applique une fonction `<K,V> => V` , basé sur la clé et la valeur précédemment associé, pour créer une nouvelle valeur et la lier à la clé

`V` **`computeIfPresent`**`(K key, BiFunction<K,V, V> remappingFunction)` : applique une fonction `<K,V> => V` , basé sur la clé et la valeur précédemment associé, pour créer une nouvelle valeur et la lier à la clé, *seulement si la clé était présente*

`cle`

`V` **`computeIfAbsent`**`(K key, Function<K,V> mappingFunction)` : applique une fonction `K => V` , basé sur la clé pour créer une valeur et la lier à la clé, *si la clé était absente de la map*

`void` **`forEach`**`(BiConsumer<K, V> action)` : applique une action à chaque couple clé-valeur

`V` **`merge`**`(K key, V newValue, BiFunction< V, V, V> remappingFunction)` : si la clef n'est pas associée à une valeur, associe la clé à la nouvelle valeur; sinon associe la clé au résultat d'une fonction basée sur la précédente valeur et la valeur transmise

Classe HashMap & Hashtable (1/5)

La classe **HashMap** (non synchronisée) est une implementation d'une **Map**. La classe **Hashtable** est quasi identique mais synchronisée.

Ces classes implémentent les fonctions de l'interface Map.

Exemple d'utilisation : association login - nom complet

```
HashMap<String, String> users = new HashMap<>();
users.put("al1", "alain");
users.put("krol", "carole");
users.put("yan", "yanice");
String key = "krol";
String val = users.get(key);
if(val!=null)
  System.out.println("valeur associee à " + key + "=" + val);
```

```
----> valeur associée à krol=carole
```

Classe HashMap (2/5)

```
//exemple d'utilisation de forEach :
users.forEach((k,v) ->
                System.out.println("valeur associée à " + k + "=" + v));
—> valeur associée à krol=carole
—> valeur associée à all=alain
—> valeur associée à yan=yanice

key = "yan";
val = "yann";
String avant = users.put(key, val);
System.out.println(key + " est associée à " + users.get(key));
System.out.println("avant, "+ key + " était associée à " + avant);
——> yan est associée à yann
——> avant, yan était associée à yanice
```

Classe HashMap (3/5)

Exemple d'utilisation : gestion de voyages

```
public class Voyage {
      String depart;
      String arrivee;
      double cout;

// generation automatique des constructeurs et des getters&setters par l'IDE :
      public Voyage() { super(); }
      public Voyage(String depart, String arrivee, int duree, double cout) {
            super();
            this.depart = depart; this.arrivee = arrivee; this.cout = cout;
      }
      public String getDepart() { return depart; }
      public String getArrivee() { return arrivee; }
      public double getCout() { return cout; }
      public void setDepart(String depart) { this.depart = depart; }
      public void setArrivee(String arrivee) { this.arrivee = arrivee; }
      public void setCout(double cout) { this.cout = cout;}
// redefinition de toString() :
      public String toString()
      {return new StringBuilder("[").append(depart).append("-").append(arrivee).
                  append(« =").append(cout).append("]").toString();} }
}
```

Classe HashMap (4/5)

On souhaite utiliser une map 'ville départ', 'liste de voyages à partir de cette ville'

```
HashMap<String, List<Voyage>> offres = new HashMap<>();
```

Ajout d'un voyage à partir de 'A', menant à 'B', pour un coût de 12.4 €

```
offres.put("A", new Voyage("A", "B", 12.2));
```

N'est pas possible !! La valeur doit être une liste de voyages, pas un simple voyage

création d'une fonction d'ajout d'un voyage utilisant la programmation fonctionnelle :

```
void mapAddVoyages(HashMap<String, List<Voyage>> offres, Voyage vge)
{
    offres.compute(vge.getDepart(), (k,v)->{
                    // si aucune liste n'est associée à k (vge.depart), en créer une
                    if(v==null) {v= new ArrayList<>();}
                    // ajouter le voyage à la liste présente ou créée
                    v.add(vge);
                    // la retourner comme nouvelle valeur
                    return v;});
}
```

Classe HashMap (5/5)

Création et stockage de voyages

```
HashMap<String, List<Voyage>> offres = new HashMap<>();
mapAddVoyages(offres, new Voyage("A", "B", 12.2));
mapAddVoyages(offres, new Voyage("B", "C", 2.5));
mapAddVoyages(offres, new Voyage("A", "C", 20));
```

recherche du voyage le moins cher partant de A

```
List<Voyage> fromA = offres.get("A");
Voyage moinsCher = Collections.min(fromA, (v1, v2)
                                -> Double.compare(v1.getCout(),  v2.getCout()));
```

recherche du voyage le plus cher partant de A, utilisation de notation par référence

```
Voyage plusCher =
      Collections.max(fromA, Comparator.comparingDouble(Voyage::getCout));
```

Classe Stream (1/4)

Classe représentant un flux de données, spécifiquement dédiée à la programmation fonctionnelle. Un flux s'épuise en le parcourant.

`boolean allMatch(Predicate<T> predicate)` : retourne vrai si *tous* les éléments du flux correspondent au prédicat
`boolean anyMatch(Predicate<T> predicate)` : retourne vrai si *au moins un* élément du flux correspondent au prédicat
`boolean noneMatch(Predicate<T> predicate)` : retourne vrai si *aucun* élément du flux ne correspond au prédicat
`Stream<T> filer(Predicate<T> predicate)` : retourne un flux dont les éléments respectent le prédicat

`long count()` : retrouve le nombre d'éléments dans le flux

`Stream<T> distinct()` : retrouve un flux sans éléments dupliqués

`void forEach(Consumer<T> action)` : applique une action à chaque élément du flux

`Stream<R> map(Function<T, R> mapper)` : retourne un flux de type R en appliquant une fonction à chaque objet
`DoubleStream mapToDouble(ToDoubleFunction<T> mapper)` : retourne un flux de doubles en appliquant une fonction à chaque objet
`IntStream mapToDouble(ToIntFunction<T> mapper)` : retourne un flux d'entiers en appliquant une fonction à chaque objet

`Optional<T> max(Comparator<T> comparator)` : retourne, possiblement, l'objet le plus grand du flux selon le comparator
`Optional<T> min(Comparator<T> comparator)` : retourne, possiblement, l'objet le plus petit du flux selon le comparator

Classe Stream (2/4)

Exemples d'utilisation

```
List<Personne> liste = List.of(new Personne("anton",15),new Personne("yvan",18),new
Personne(« paula",20));

Stream<Personne> flux = liste.stream();
IntStream fluxAge = flux.mapToInt(Personne::getAge);
OptionalInt vieilAge = fluxAge.max();
if(vieilAge.isPresent())
   System.out.println("-age le plus avancé = " + vieilAge.getAsInt());

//ATTENTION RECREER LE FLUX !!!
flux = liste.stream();
Optional<Personne> jeune =  flux.min(Comparator.comparingInt(Personne::getAge));
if(jeune.isPresent())
   System.out.println("-personne majeure la plus jeune : " + jeune.get());

//ATTENTION RECREER LE FLUX !!!
flux = liste.stream();
Stream<Personne> adultes =  flux.filter(p->p.getAge()>=18);
System.out.println("-personne adultes : ");
adultes.forEach(System.out::println);
```

Classe Stream (3/4)

`<R,A> R collect(Collector<? super T,A,R> collector)` : retourne le contenu du stream dans une collection

`void sorted()`: trie le flux
`void sorted(Comparator<T> comp)`: trie le flux selon le comparateur

`<A> A[] toArray(IntFunction<A[]> generator)`: retourne un tableau à partir du flux

`static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)` : retourne un flux infini, à partir de seed, en appliquant l'opérateur f à chaque étape

`Stream<T> limit(long maxSize)` : borne la taille d'un flux

```
//Créer un flux avec 10 nombres impairs
Stream<Integer> nbImpairs = Stream.iterate(1, n -> n + 2).limit(10);
nbImpairs.forEach(System.out::println);
```

Classes IntStream, DoubleStream, Collectors

Les classes **IntStream**, **DoubleStream**, **LongStream**, … sont des classes de flux possédant des méthodes retournant la somme et la moyenne des valeurs

La classe **Collectors** permet la réduction de flux, vers une Collection, un nombre (somme, moyenne), une Map

Exemples d'utilisation

```
List<Personne> liste = List.of(new Personne("anton",15),new Personne("yvan",18),new
Personne("paula",20));
Stream<Personne> flux = liste.stream();
List<String> prenoms =  flux.map(Personne::getPrenom).Collectors.toList() );

//ATTENTION RECREER LE FLUX !!!
flux = liste.stream();
Stream<Personne> adoStream =  flux.filter(p->(p.getAge()>=12 && p.getAge()<18));
Personne[] tabAdos = adoStream.toArray(Personne[]::new);

//ATTENTION RECREER LE FLUX !!!
flux = liste.stream();
double ageMoyen = flux.collect(Collectors.averagingInt(Personne::getAge));
System.out.printf("age moyen = %.1f", ageMoyen);
```

Classe Stream (3.9/4)

Exemples d'utilisation

On ajoute la notion de Region à la classe Personne, en modifiant le constructeur, avec :

```
  public enum Region
  { ARA,BFC,BRE,CVL,COR,GES,HDF,IDF, NOR,NAQ,OCC,PDL,PAC,GP,GF,MQ,RE,YT; }

List<Personne> liste = List.of(new Personne("anton",15,Region.HDF),
new Personne("yvan",18, Region.RE),new Personne("paula",20, Region.HDF));

Stream<Personne> flux = liste.stream();
//grouper par région dans une Map :
Map<Region, List<Personne>> parRegion =
                flux.collect(Collectors.groupingBy(Personne::getRegion));

parRegion.forEach((r,l) -> System.out.println(r + "="+ l.toString()));
```

RE=[yvan age = 18]
HDF=[anton, age = 15, paula, age = 20]

Classe Stream (3.95/4)

Exemples d'utilisation

```
//ATTENTION RECREER LE FLUX !!!
flux = liste.stream();
//grouper selon prédicat dans une Map :
Map<Boolean, List<Personne>> parStatut =
            flux.collect(Collectors.partitioningBy(p -> p.getAge() >= 18));
```

---

false=[anton, age = 15]
true=[yvan, age = 18, paula, age = 20]

Classe Stream (4/4)

COPIE DE LISTE

Retour sur la copie de liste d'objets clonables :

```
List<Personne> liste1 = new ArrayList<>(List.of(new
Personne("anton", 17), new Personne("paula", 19));

List<Personne> liste2 =
liste1.stream().map(Personne::clone).collect(Collectors.toList());

liste2.get(0).age = 33;
System.out.println(liste1.get(0));     ——> anton, age = 17
```