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;
  }

}
```

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
```

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, ….

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 String

Constante

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

Conversion

Conversion :

- ***primitive*** : `String ch = String.valueOf(valeur)`
  -> *`String ch = String.valueOf(5.2)`*
- ***objet*** :
  - Explicite :
    - `String ch = String.valueOf(objet)` ou
    - `String ch = objet.toString()`
  - Implicite :
    - `String ch = objet.toString()`

  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

Affichage

Afficher des valeurs dans une chaîne :

- **Concaténation**

```
int i = 4;
double pi = 3.141592654
Personne p = new Personne("alfred");
System.out.println("voici un entier : " + i + ", un reel : " + pi +
", et une personne " + p);
```

- **Formatage**

```
System.out.println(String.format("voici un entier : %d, un reel :
%f, et une personne %s",i,pi,p);
```

Mise en forme possible des réels :

```
System.out.println(String.format("voici un entier : %d, un reel :
%.2f, et une personne %s",i,pi,p);
```

—> voici un entier : 4, un reel : 3,14, et une personne anais

La classe String

Bloc de texte (Java 15)

Depuis Java 15, possibilité de définir des blocs de texte :

```
String ch = """
voici un entier : %d,
un reel : %.2f,
   et une personne %s """.formatted(i,pi,p);

System.out.println(ch);

->
voici un entier : 4,
un reel : 3,14,
   et une personne anais
```

La classe String

Découpe

String permet de tronçonner une chaîne :

```
String ch = "une,suite,de,,valeurs";

String[]mots1 = ch.split(",");
   ->["une", "suite", "de", , "valeurs"]

String[]mots2 = ch.split("n",3);
   ->["une", "suite", "de,,valeurs"]
```

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();
    }
}
```

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)

Elle est nommée dans l'interface **Comparable**
Il est nécessaire d'implémenter cette interface

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;
 }
}
```

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));
```