



LA BIBLIOTHEQUE

Martine GAUTIER - Université de Lorraine
martine.gautier@univ-lorraine.fr

Structuration de la bibliothèque



Bibliothèque de classes livrée avec le JDK

→ plus d'un millier de classes



Structurée en packages et sous-packages



Packages généraux et packages dédiés à des utilisations spécifiques



Exemples :


→ `java.lang`, `java.util`

→ `java.io`, `javax.swing`, `java.awt`, `java.net`, `java.sql`

La classe `java.lang.String`

- ❑ Une instance de `java.lang.String` est immuable.
 - aucune fonction de transformation
 - un seul objet pour deux chaînes identiques
- ❑ Quelques particularités
 - instantiation avec `" "`
 - surcharge de l'opérateur `+`
- ❑ Plein de fonctions utiles : `charAt`, `indexOf`, `compareTo`, `substring`, ...

```
String jour1 = "Lundi" ;  
String jour2 = "Lundi" ;  
String mois = "Février" ;  
int nj = 21 ; int an = 2016 ;  
String date1 = jour1 + nj ;  
String date2 = jour1 + 21 ;  
String date3 = nj + mois + an ;
```



**L'usage du +
provoque une
instanciation.**

Dans la classe geometrie.Point

```
String toString() {  
    return "<" + getAbscisse() + ", " + getOrdonnee() + ">" ;  
}
```

Dans la classe geometrie.Triangle

```
String toString() {  
    return "<" + getP1() + ", " + getP2() + ", " + getP3() + ">" ;  
}
```

Dans la classe *geometrie.Polygone*

```
private Point[] tp ;

String toString() {

    String res = "" ;

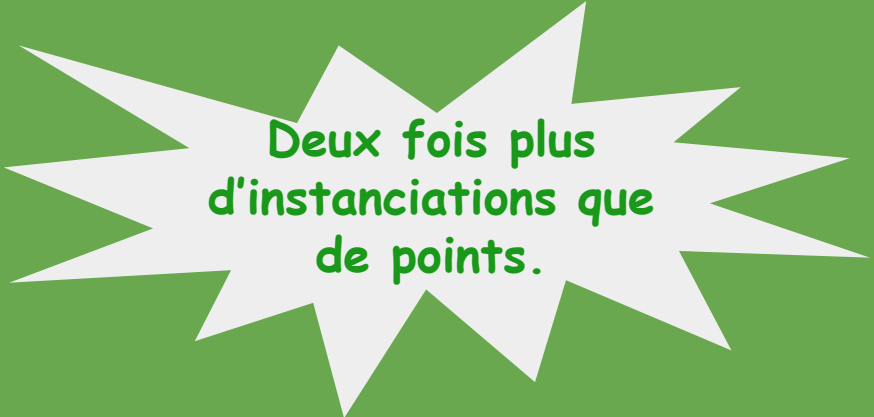
    for (Point p : tp) {

        res = res + " " +p ;

    }

    return res ;

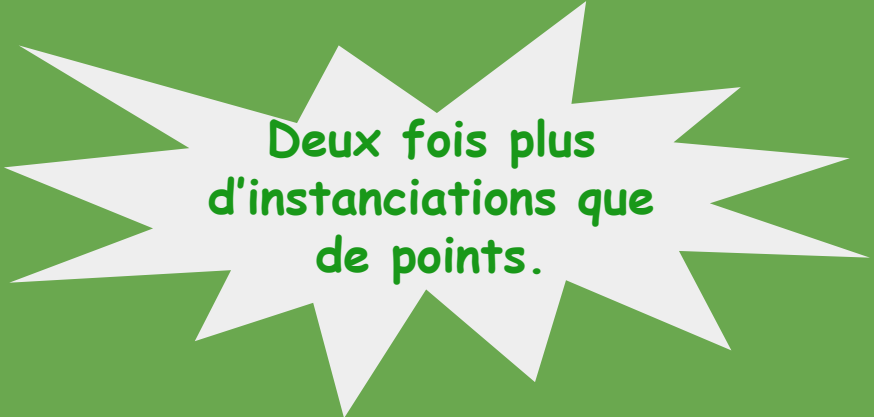
}
```



**Deux fois plus
d'instanciations que
de points.**

Dans la classe *geometrie.Polygone*

```
private Point[] tp ;  
String toString() {  
    String res = "" ;  
    for (Point p : tp) {  
        res = res + " " + p ;  
    }  
    return res ;  
}
```



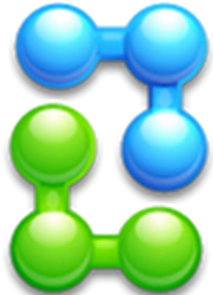
Deux fois plus
d'instanciations que
de points.


```
String date1 = "Lundi 1er février" ;  
String date2 = "Mardi 2 février" ;  
char carac = date1.charAt(12) ;  
int index = date1.indexOf("1") ;  
int cd = date1.compareTo(date2) ;    // ordre lexicographique  
String ssc = date2.substring(6, date2.length()) ;  
boolean eq = date1.equals(date2);    // comparaison des caractères
```

**en cas de débordement
IndexOutOfBoundsException**

La classe `java.lang.StringBuilder`

- ❑ Une instance de `java.lang.StringBuilder` peut être modifiée
 - fonctions de transformations permettant d'ajouter, enlever, modifier des caractères
- ❑ Représentation mémoire différente de celle de `String`
 - que dit [artEoz](#) ?



Dans la classe *geometrie.Polygone*

```
private Point[] tp ;
```

```
String toString() {
```

```
    int capacite = this.getNbPoints()*10 ;
```

```
    StringBuilder res = new StringBuilder(capacite) ;
```

```
    for (Point p : tp) {
```

```
        res.append(" ").append(p) ;
```

```
    }
```

```
    return res.toString() ;
```

**Fixer la capacité
au mieux !**

**Une seule
instanciation**

La classe `java.lang.Math`

- ❑ Cette classe n'a pas de constructeur ; elle ne contient que des fonctions statiques.

```
public static double sqrt(double a)
```

- ❑ Appel d'une fonction statique

```
double rc = Math.sqrt(144.) ;
```

- ❑ Utilisation de la notation pointée, mais rien à voir avec un appel de fonction avec receveur, ce qui peut être perturbant.

Cf paragraphe sur les `static`, à la fin de ce chapitre

Le package java.util

❏ Classes utilitaires

- Classe `java.util.GregorianCalendar` : calendrier grégorien
- Classe `java.util.Random` : générateur aléatoire
- Classe `java.util.Formatter` : formatteur de chaînes
- Classe `java.util.Scanner` : analyseur lexical
- Classe `java.util.Arrays` : fonctions sur les tableaux (tri, ...)

```
// Copier
```

```
int[] te = {1, 3, 2, 0, 7} ;
```

```
int[] teb = Arrays.copyOf(te, 3) ; // extraire les 3 premiers
```

```
// Trier
```

```
int[] ttrie = Arrays.sort(te) ;
```

```
// Remplir
```

```
Point[] tp = new Point[5];
```

```
Arrays.fill(tp, new Point(1., 2.)) ;
```

```
// Comparer

int[] te1 = {1, 3, 2, 0, 7} ;

int[] te2 = {1, 3, 2, 0, 7} ;

int[] te3 = t1 ;

// equals compare les adresses

boolean eq12 = te1.equals(te2);    // false

boolean eq13 = te1.equals(te3);    // true

// equals compare les valeurs

boolean eq12b = Arrays.equals(te1, te2);    // true
```



```
// Comparer

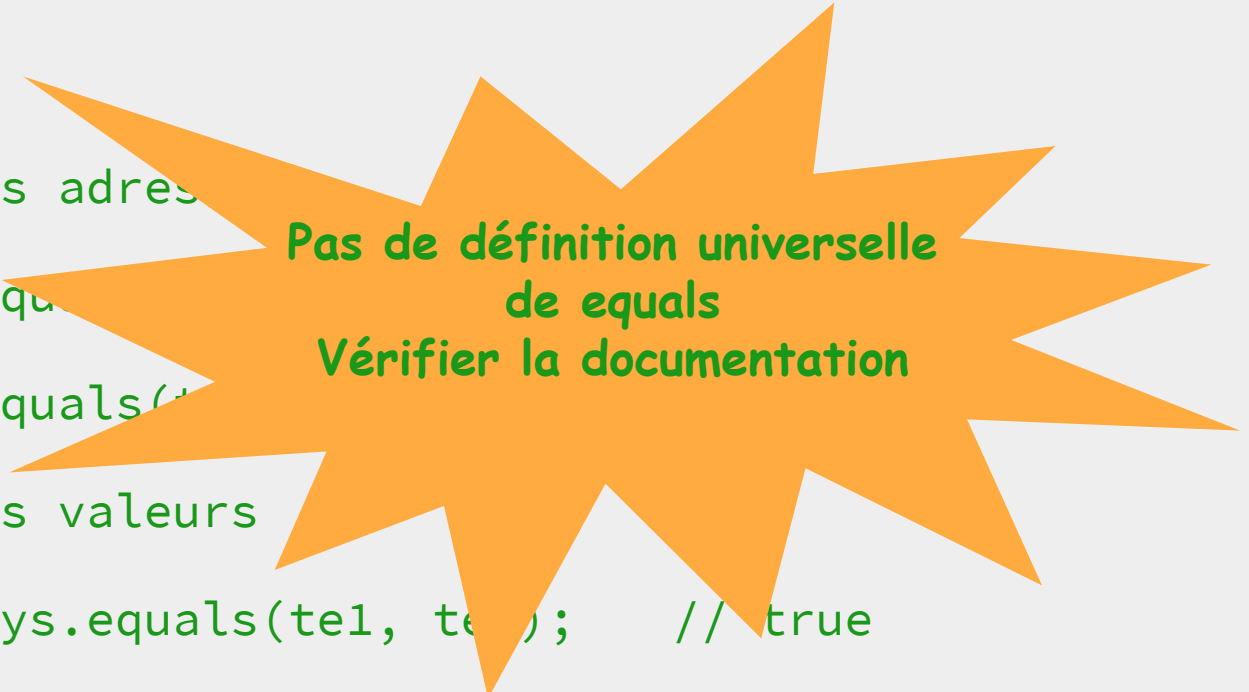
int[] te1 = {1, 3, 2, 0, 7} ;

int[] te2 = {1, 3, 2, 0, 7} ;

int[] te3 = t1 ;

// equals compare les adresses
boolean eq12 = te1.equals(te2); // true
boolean eq13 = te1.equals(te3); // true

// equals compare les valeurs
boolean eq12b = Arrays.equals(te1, te2); // true
```

A large, multi-pointed orange starburst graphic with a black outline, positioned in the lower right quadrant of the image. It contains two lines of bold black text.

**Pas de définition universelle
de equals**
Vérifier la documentation

Le package java.util

❏ Classes de gestion de collections

- Classe `java.util.ArrayList` : tableau dynamique
- Classe `java.util.LinkedList` : liste doublement chaînée
- Classe `java.util.Stack` : pile
- Classe `java.util.HashMap` : table
- Classe `java.util.Collections` : fonctions sur les collections (tri, ...)

❏ Classes génériques

- le type des éléments est quelconque
- il est fixé lors de la déclaration


```
ArrayList als = new ArrayList() ;  
  
als.add("Neige");  
  
als.add("Glace");  
  
int e = als.indexOf("Neige") ;  
String ch = als.get(0) ;
```



Non générique, à
bannir


```
Stack<Point> ps = new Stack<>() ;
```

```
ps.push(new Point(6., 9.)) ;
```

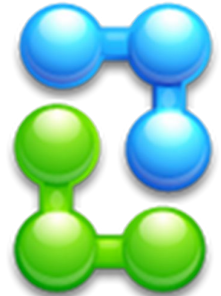
```
ps.push(new Point(6., 9.)) ;
```

```
Point p = ps.peek() ;
```

```
ps.pop() ;
```

```
HashMap<String, Point> hm = new HashMap<>() ;  
hm.add("Point 1", new Point(6., 9.)) ;  
hm.add("Point 2", new Point(61., -9.)) ;  
hm.add("Point 3", new Point(2., -1.)) ;  
Point p = hm.get("Point 2") ;
```


Représentation mémoire des collections



Parcourir une collection

- ❑ Itération simple sur le rang des éléments

```
for (int k = 0 ; k < col.size() ; k++) { ... e = col.get(k); .... }
```

- ❑ L'utilisation du rang pour accéder à un élément n'est guère efficace en cas de rangement contigu

tableau, ArrayList,

et totalement inefficace voire impossible pour des rangements non contigus

LinkedList, HashMap

- ❑ Itération *foreach*

```
for (Element e : col) { ... }
```



```
HashMap<String, Point> hp ;
```

```
for (Point s : hp.keySet()) { System.out.println(key); }
```

ou bien

```
keys.forEach(key -> System.out.println(key));
```



On parcourt l'ensemble
des clés.

```
HashMap<String, Point> hp ;
```

```
Collection<Double> values = hp.values();  
values.forEach(value -> System.out.println(value));
```

On parcourt l'ensemble
des valeurs.

```
hp.forEach((key, value) -> {  
    System.out.print("key: "+ key);  
    System.out.println(", Value: "+ value);  
});
```

On parcourt les couples
clé/valeur.

Comparaison des efficacités

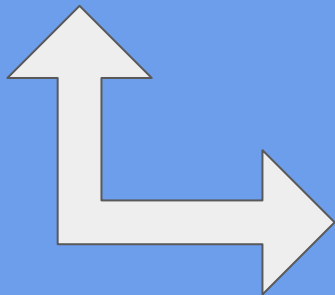
Les classes enveloppes

- ❑ Impossible d'utiliser un type primitif pour créer une collection
 - `ArrayList<int>` : refusé par le compilateur
- ❑ Les classes enveloppes portent bien leur nom ...
 - `Integer` : enveloppe de `int`
 - `Double` : enveloppe de `double`
- ❑ La conversion entre type primitif et classe enveloppe est implicite.



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

```
ali.add(new Integer(43)) ;  
ali.add(new Integer(-89)) ;  
int e = ali.get(0).intValue() ;
```



```
ali.add(43) ;  
ali.add(-89) ;  
int e = ali.get(0) ;
```

Les fonctions déclarées *static*

❑ Fonctions *static* en l'absence d'objet

-> Pour déclarer la fonction `main` qui ne peut pas s'appliquer sur un objet

-> Pour déclarer une fonction qui opère sur des types primitifs (cf. classe `Maths`)

❑ Fonctions *static* pour construire des bibliothèques d'algorithmes applicables sur des types de données variés

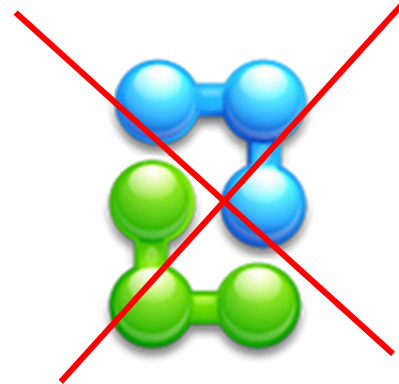
-> cf classe `Collections`

❑ ATTENTION à l'abus de fonctions *static*

-> Toute fonction appelée par une fonction *static* doit être déclarée *static*

Les attributs déclarés *static*

- ❑ **Attribut static** : commun à toutes les instances d'une classe
 - > Pour définir une constante : codification de l'océan et des icebergs, couleurs, ...
 - > Pour partager une donnée : compteur du nombre d'instances
- ❑ **ATTENTION à l'abus d'attributs static**
 - > Seule une fonction `static` peut accéder à un champ `static`



```
public class Ocean {  
  
    private static int OCEAN = 0 ;  
  
    private static int GLACE = 1 ;  
  
    public int[][] getColors() {  
  
        ...    res[i][j] = GLACE ; ....  
  
    }  
  
}
```

Règle de style : les
champs static de
type int s'écrivent en
majuscules

La codification est
définie une bonne fois
pour toutes, ce qui
peuvent éviter des
erreurs.

```
public class Point {
```

```
    private static int CPT = 0 ;
```

```
    private double abscisse, ordonnee ;
```

```
    public Point(double a, double o) {
```

```
        this.abscisse = a ; this.ordonnee = o ;
```

```
        CPT++ ;
```

```
    }
```

```
    public static int getCount() {
```

```
        return CPT;
```

```
    }
```

```
}
```



Le champ CPT est
partagé par toutes les
instances

```
Point p1 = new Point(4., 7.) ;  
Point p2 = new Point(8., 1.) ;  
int cpt = Point.getCount() ;
```

Le singleton

- ❑ Contexte : un objet est utilisé dans bon nombre de fonctions dans des classes différentes
 - ❑ Exemple : constitution d'un journal qui trace certains appels de fonction
- ❑ Solution : passer cet objet en paramètre des fonctions
 - ❑ fastidieux
- ❑ Alternative : définir un singleton
 - ❑ = instance unique d'une classe, accessible dans n'importe classe de l'application.

```
public class Journal {  
    private static Journal instance = new Journal() ;  
    public static Journal getInstance() { return instance; }  
    private List<String> traces ;  
    private Journal() { this.traces = new ArrayList<>(); }  
    public void add(String trace) {this.traces.add(trace); }  
}
```

```
public class Journal {  
    private static Journal instance = new Journal() ;  
    public static Journal getInstance() { return instance; }  
    private List<String> traces ;  
    private Journal() { this.add(""); }  
    public void add(String trace); }  
}
```

Instance unique

```
public class Journal {  
    private static Journal instance = new Journal() ;  
    public static Journal getInstance() { return instance; }  
    private List<String> traces ;  
    private Journal() { this.traces = new ArrayList<>(); }  
    public void add(String trace) { traces.add(trace); }  
}
```

Fonction de consultation
de cette instance unique

```
public class Journal {  
    private static Journal instance = new Journal() ;  
    public static Journal getInstance() { return instance; }  
    private List<String> traces ;  
    private Journal() { this.add("Initialisation"); }  
    public void add(String trace) { traces.add(trace); }  
}
```

Garantit que cette
classe ne sera jamais
instanciée ailleurs qu'ici


```
public class Journal {  
    ...  
    public static Journal getInstance() { return instance; }  
    public void add(String trace) {this.traces.add(trace); }  
}
```

```
Journal j = Journal.getInstance() ;  
j.add("Création du compte ") + noCompte;
```

```
Journal j = Journal.getInstance() ;  
j.add("Suppression du compte ") + noCompte;
```

```
public class Journal {
```

```
...
```

```
public static Journal getInstance() { return instance; }
```

```
public void add(String s) { add(s+trace); }
```

```
}
```

Un appel à `getInstance` peut
être écrit n'importe où.
Chaque appel de `getInstance`
fournit toujours le même
objet.

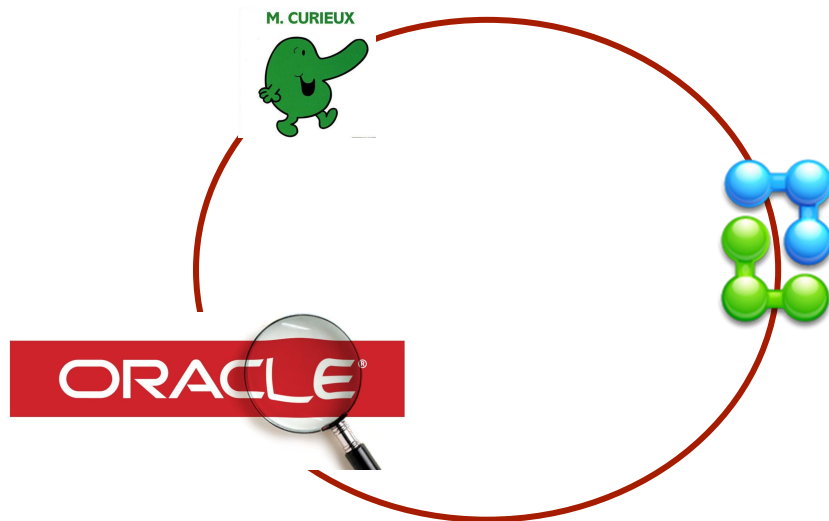
```
Journal j = Journal.getInstance();
```

```
j.add("Création du compte");
```

```
Journal j = Journal.getInstance() ;
```

```
j.add("Suppression du compte "+noCompte);
```

Pour conclure sur la bibliothèque



Pour conclure sur la bibliothèque



Plusieurs milliers de classes dans Java 11

M. CURIEUX



Faire preuve de curiosité



ORACLE®

<https://docs.oracle.com/javase/11/docs/>



Utiliser [artEoz](#) pour comprendre ...