

Responsable de l'UE : Christophe Marsala
(email: Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)



Cours 1 – 12 septembre 2022

INFORMATIONS

- Sur le site Moodle de l'UE LU2IN002-S1-22 :
<https://moodle-sciences-22.sorbonne-universite.fr/course/view.php?id=2369>
 - slides des cours
 - version PDF du poly TD/TME (ne pas imprimer !!)
 - le poly TD/TME sera distribué au premier TD
 - PDF contenant des exercices de TME supplémentaires pour les rapides
 - quizzes d'auto-évaluation en ligne
 - forum pour poser des questions
 - rendu des devoirs (TME, projet)

PROGRAMME DU JOUR

- 1 Introduction
- 2 Premier programme
- 3 Concepts de base de POO
- 4 Guide de survie en Java

FONCTIONNEMENT

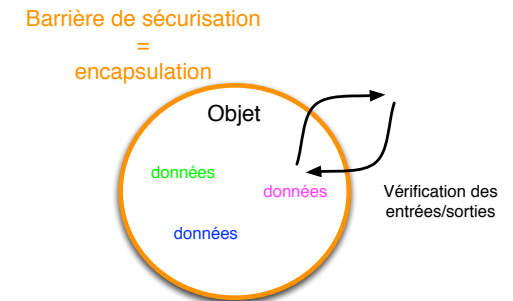
- Organisation de l'UE
 - 1h45 Cours
 - 1h45 TD : connaître le cours n avant d'aller en TD n
 - 1h45 TME (en binôme sur machine)
- **Evaluation** (dans l'état actuel de la situation)
 - contrôle de mi-semestre = 25%
 - TME solo (en général, pendant la séance de TME9) = 15%
 - mini-projet (rendu & soutenance lors du dernier TME) = 10%
 - examen final (janvier) = 50 % de la note finale

Rem : en général, aucun document n'est autorisé lors des épreuves

Pourquoi faire de la Programmation Orientée Objet (POO) ?

- Pour développer des systèmes complexes... *Sans se planter*

- **diviser** le système complexe en une multitude de systèmes simples : **les objets**
- **sécuriser** l'accès aux données sensibles



- [corollaire] Travailler à plusieurs... *Avec un minimum de bugs*
 - toujours penser son programme pour les autres : **sécuriser, simplifier, compartimenter**
 - double vision : client/fournisseur

LE LANGAGE JAVA

JAVA : LE CHOIX D'UNE ARCHITECTURE DYNAMIQUE

De nombreux langages sont orientés objets, Java en est un exemple classique

Le langage **Java**

- langage moderne qui puise son inspiration de sources diverses
 - syntaxe très proche du C/C++
 - architecture dynamique avec un **compilateur** et une machine virtuelle appelée **JVM** (*Java Virtual Machine*) capable d'exécuter le code Java compilé
- évolution régulière
 - composants additionnels et fonctionnalités
 - la syntaxe n'a, elle, que peu évolué
- très utilisé dans l'industrie
 - API (*Application Programming Interface*) de programmation pour de multiples applications

- Langage orienté objet, syntaxe simple, traits impératifs
- Langage robuste et sûr
- Pour créer des applications indépendantes de la machine
- Langage compilé pour créer des applications performantes

- 1 Introduction
- 2 Premier programme
- 3 Concepts de base de POO
- 4 Guide de survie en Java

UNE CLASSE, UN MAIN : SYNTAXE

La syntaxe des **signatures de classe** et de la **signature du main** est à **apprendre par cœur** (explications dans les cours suivants).

- 1 Signature d'une classe
 - **public**
 - **class**
 - suivi du nom de la classe
- 2 Signature d'un main
 - **public static void**
 - **main**
 - toujours le même argument **String[] args**
- 3 Puis les instructions dans le main
 - par exemple, affichage de la chaîne **"Bonjour !"**

```

1 // dans le fichier MonPremierProgramme.java
2 public class MonPremierProgramme{
3     public static void main(String[] args) {
4         System.out.println("Bonjour !");
5     }
6 }
```

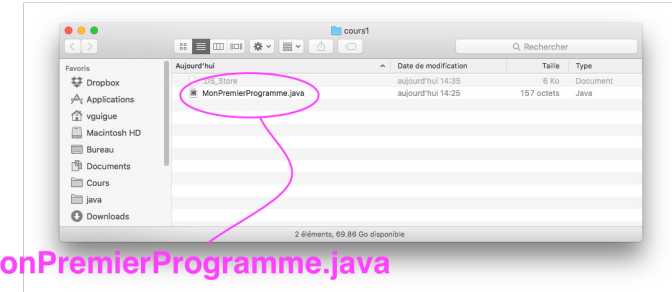
UNE CLASSE, UN MAIN()

- 1 En Java, tout code doit être écrit dans une classe
 - règle : les **noms de classe commencent par une majuscule**

```

public class MonPremierProgramme{
    // code
}
```

- 2 1 classe est mise dans 1 fichier du même nom que la classe



- 3 **Programme** = ensemble de classes avec un (ou plusieurs) **point d'entrée (=main)**
 - ce programme est exécutable après compilation

COMPILATION/EXÉCUTION

- Pré-requis :
 - JDK (Java Dev. Kit) installé sur la machine
 - être dans le bon répertoire (!)
- 1 Compilation
 - vérification de la syntaxe, droits d'accès...
 - création d'un exécutable en **bytecode** (code de la JVM) : **MonPremierProgramme.class**

```
» javac MonPremierProgramme.java
```

- 2 Exécution par la JVM
 - ici, exécution dans la console

```
» java MonPremierProgramme (pas d'extension)
```

⇒ résultat :

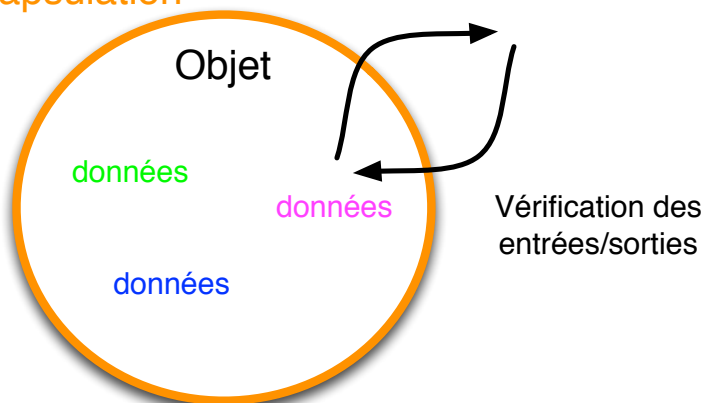
```
» Bonjour !
```

- 1 Introduction
- 2 Premier programme
- 3 Concepts de base de POO
- 4 Guide de survie en Java

- Diviser un programme complexe en **objets**
 - objet autonome : **réutilisable** dans plusieurs projets
 - Vecteurs, Personne, DisplaySimulation,...
 - objet **sécurisé** : garantie de bon usage par d'autre
 - un objet intègre des **données** et des **méthodes** pour les manipuler proprement,
 - les transactions bancaires sont journalisées, les éléments d'une simulation physique ne se téléportent pas...
 - objet **simple et intuitif**
- Enjeux
 - **réfléchir en amont** au découpage et à la sécurisation
 - **documenter** le code (en premier lieu, respecter les conventions pour faciliter la compréhension)

APPROCHE ORIENTÉE-OBJET

Barrière de sécurisation
=
encapsulation



Barrière de sécurisation

SYNTAXE : SIGNATURE ET ATTRIBUTS

- 1 Fichier : **1 classe** correspond à **1 fichier**
 - nom de la classe + **.java** = nom du fichier
 - marquer l'encapsulation, faciliter la ré-utilisation
 - **le nom de classe commence par une majuscule**
- 2 Déclaration des **attributs** / **champs** / **variables d'instance**
 - répondre à : **De quoi est composé notre objet ?**
 - les attributs sont presque toujours **private** (cf plus loin)
 - **nom des attributs en minuscules**
- 3 Définir des **méthodes** (\simeq fonctions)
 - comment construire un objet ?
 - quelles opérations effectuer sur l'objet ?
 - **nom des méthodes en minuscules**

```
1 // Fichier Point.java
2 public class Point{ // classe publique
3     private double x,y; // attributs privés
```

Comment construire un Point ? Utiliser un **constructeur**

⚠ Attention à considérer le problème des 2 points de vues

Fournisseur

- **Besoin** : 2 coordonnées fournies en argument
- **Action** : initialisation des valeurs des attributs

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4     // constructeur
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

Client

ie : utilisateur d'une classe Point existante...

- **Besoin** : créer une instance dans la mémoire avec les bonnes valeurs

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // appel du constructeur
7         // avec des valeurs choisies
8         Point p = new Point(2., 3.1);
9     }
```

Un constructeur porte le nom de la classe et il ne rend rien !

Point p1 ;

p1 est une **variable** de type Point (**aucun objet n'est créé**)

p1 = **new** Point(1,2);

La variable p1 **référence** un **objet** / une **instance** de la classe Point

Représentation mémoire correspondante (résultat)



- représenter les classes sans les méthodes
- valeurs des attributs prises pour l'objet correspondant
- types et noms des variables
- liens de référencement

CRÉER UN OBJET : INSTANCIATION

Coté fournisseur :

mise en route de l'objet

Constructeur = contrat d'initialisation des attributs

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

Coté client :

création d'une instance de classe

Instanciation = création d'une zone mémoire réservée à l'objet

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // appel du constructeur
7         // avec des valeurs choisies
8         Point p1 = new Point(1., 2.);
9     }
```

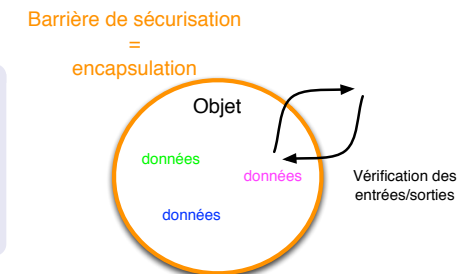


La variable p1, de type Point, **référence** une instance de la classe Point dont les attributs ont pour valeur 1 et 2.

SYNTAXE : MÉTHODES

Comment manipuler un Point ?

- 1 **définir des méthodes**
eg : accéder aux attributs (en lecture)
- 2 **les invoquer depuis l'extérieur**



Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4     public Point(double x2, double y2){
5         x = x2;
6         y = y2;
7     }
8
9     public double getX(){
10         return x;
11     }
12 }
13 }
```

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8
9         double px = p.getX();
10
11         System.out.println(
12             "coord_x_y: "+px);
13     }
14 }
```

AUTORISATIONS D'ACCÈS

- **public** : accessible / visible depuis l'extérieur de l'objet (eg : un main, un autre objet...)
- **private** : protégé / invisible depuis l'extérieur de l'objet
- Les constructeurs sont en général **public**
 - ils ont vocation à être appelés depuis l'extérieur
- Les attributs sont en général **private**
 - ils sont protégés et non accessibles depuis l'extérieur

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

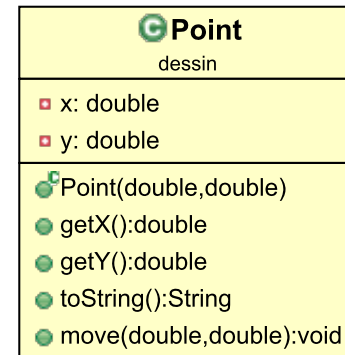
Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // opération autorisée
7         Point p = new Point(2., 3.1);
8
9         // opération impossible :
10        // ERREUR DE COMPILATION
11        double d = p.x;
12    }
13 }
```

REPRÉSENTATION UML

On ne programme pas pour soi-même... mais pour les autres :

- Respecter les **codes syntaxiques** : majuscules, minuscules...
- Donner des **noms explicites** (classes, méthodes, attributs)
- Développer une **documentation** du code (cf cours javadoc)
- ... Et proposer une **vision synthétique** d'un ensemble de classes : ⇒ **UML : Unified Modeling Language**

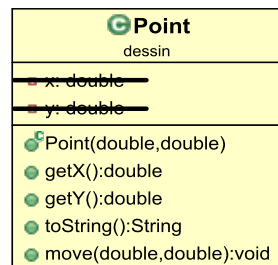


- nom de la classe
 - attributs
 - méthodes (et constructeurs)
- + code pour visualiser **public/private**
+ liens entre classes pour les dépendances (cf cours sur composition)
Ne pas confondre : représentation des classes et représentation des objets en mémoire

UML CLIENT vs FOURNISSEUR

Plusieurs types de diagrammes pour plusieurs usages :

- Vue **fournisseur** : représentation complète
- Vue **client** : représentation public uniquement



Idee :

Le code doit être pensé pour les autres :

- tous les noms doivent être aussi clairs que possible
- un diagramme plus limité est plus facile à lire

REPRÉSENTATION UML (SUITE)

Deux manières de voir l'UML :

- 1 Outil pour une **visualisation** globale d'un code complexe
- 2 Outil de **conception** / développement indépendant du langage

Dans le cadre de LU2IN002 : **seulement l'approche 1**

Limites de l'UML :

- Vision architecte...
- Mais pas d'analyse de l'exécution du code

REFLEXION SUR LA SYNTAXE OBJET

Exemple type : addition de 2 Point

Réfléchir à la signature d'une méthode `add` permettant d'additionner 2 instances de `Point` (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des attributs des opérandes)

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8         Point p2 = new Point(0.5, 1);
9
10        Point p3 = p.add(p2);
11    }
12 }
```

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4     public Point(double x2, double y2){
5         x = x2;
6         y = y2;
7     }
8
9     public Point add(Point p){
10        return new Point(x+p.x, y+p.y);
11    }
12 }
```

Syntaxe objet = il faut penser objet... **Pas évident au début !**

SYNTAXE : REDÉFINITION DE MÉTHODES STANDARDS

- Des **méthodes standards** **existent** et sont **utilisables** pour tous les objets (cf cours héritage)...
 - mais avec un comportement pas toujours satisfaisant**
- Exemple : conversion d'un objet en chaîne de caractères :
`public String toString()`

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8
9         String str = p.toString();
10
11        System.out.println("p:" + str);
12    }
13 }
```

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     ...
4     public String toString(){
5         // redéfinition
6         return "[" + x + ", " + y + "];"
7     }
8 }
```

» p: Point@8764152

» p: [2, 3.1]

COMPILATION/EXÉCUTION

Nous avons vu précédemment comment compiler et exécuter **UNE** classe... **Comment faire maintenant qu'il y en a plusieurs ?**

```
» javac Point.java
» javac TestPoint.java
» java TestPoint
```

Syntaxe réduite :

```
» javac Point.java TestPoint.java OU javac *.java
» java TestPoint
```

Remarque : il peut y avoir **plusieurs main**
(mais pas plus de 1 par classe)

- Compilation de tous les main d'un coup
- Execution d'un seul (appel à la classe correspondante)

PLAN DU COURS

- 1 Introduction
- 2 Premier programme
- 3 Concepts de base de POO
- 4 Guide de survie en Java

TYPES DES VARIABLES DE BASE EN JAVA

- Entier, réel, booléen, caractère : ces types sont disponibles de base en Java avec les opérateurs les plus courants.

`int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`

⚠ La plupart des types et syntaxes associées sont comparables au C/C++... **Sauf le booléen.**

Le booléen vaut `true/false` et n'est pas convertible en entier

- Déclaration

```
1 int i; // déclaration de i
2 System.out.println(i); // => 0
3 double d = 2.6;
4 boolean b = true; // ou false
5 char c = 'a';

1 // opérations de base: + - / * ...
2 int j = i+2;
3 int k = 1/2; // = 0 Attention a la division entiere
```

OPÉRATEURS CLASSIQUES (PAR ORDRE DE PRIORITÉ)

opérateurs postfixés	[] . expr++ expr--
opérateurs unaires	++expr --expr +expr -expr ~ !
création ou cast	new (type) expr
opérateurs multiplicatifs	* / %
opérateurs additifs	+ -
décalages	<< >> >>>
opérateurs relationnels	< > <= >=
opérateurs d'égalité	== !=
et bit à bit	&
ou exclusif bit à bit	^
ou (inclusif) bit à bit	
et logique	&&
ou logique	
opérateur conditionnel	? :
affectations	= += -= *= /= %= &= ^= = <=> >>= >>>=

CONVERSIONS ENTRE TYPES

Java, un langage typé

Les types sont très importants en Java : le compilateur vérifie toujours les types des différentes variables

- Certaines conversions sont **implicites**

```
1 double d = 42; double d2 = i; // avec i un int existant
```

Tout type de base peut se convertir en `String`

```
1 String s = "mon_message"+1.5+" "+d;
```

- Certaines conversions doivent être données **explicitement**

```
1 int i = (int) 2.4;
```

Perte d'information liée à la conversion ; Java ne tolère pas la conversion implicitement, il faut que le programmeur la demande explicitement (pour être sûr que la perte d'information est souhaitée).

- Conversions **impossibles**

```
1 int i = (int) true; // conversion impossible des booléens
```

CONDITIONNELLES

- Syntaxe de l'**alternative**

```
1 int i=11;
2 if (i > 38) {
3     // code à effectuer dans ce cas
4 } else { // le else est facultatif
5     // Code à effectuer sinon
6 }
```

- En cas de clauses multiples :

```
1 switch(i){
2 case 1:
3     // Code à effectuer si i == 1
4     break; // sinon le reste du code est AUSSI effectué
5 case 2: //
6     // Code à effectuer si i == 2
7     break;
8 default : // Si on n'est passé nulle part ailleurs
9 }
```


Même définition des boucles qu'en C/C++

- Syntaxes : 2 options (principales)

Pour i allant de 0 à 9, faire...

```
1 int i;
2 for(i=0; i<10; i++) { // i prend les valeurs 0 à 9
3                     // ==> 10 itérations
4 // code à effectuer 10 fois
5 }
```

Tant que i inférieur à 10, faire...

```
1 int i = 0;
2 while(i<10) { // i prend les valeurs 0 à 9 =
3             // 10 itérations
4 // code à effectuer 10 fois
5 i++; // ne pas oublier, sinon boucle infinie !
6 }
```

- D'autres syntaxes sont possibles : `do...while` etc...

Trois types d'interruptions de boucles

- `return` : l'interruption **la plus forte**. Retour anticipé de la fonction (**sort de la fonction**, pas seulement de la boucle).

```
1 // le modulo par 5 peut-il retourner un entier >=5?
2 public void maFonction(){
3     for(int i=0; i<10; i++){
4         if(i%5>4){
5             System.out.println("C'est très étrange");
6             return;
7         }
8     }
9     System.out.println("L'opération modulo 5 retourne "+
10                        "toujours un entier inférieur à 5");
11 }
```

INTERRUPTIONS DE FONCTIONS/BOUCLES (2/3)

INTERRUPTIONS DE FONCTIONS/BOUCLES (3/3)

3 types d'interruptions de boucles

- `return`
- `break` : sortie anticipée de la boucle

```
1 // 6 fait-il partie des multiples de 2?
2 public void maFonction(){
3     boolean found = true;
4     for(int i=0; i<10; i++){
5         if(i * 2 == 6){
6             found = true;
7             break; // pas besoin d'aller plus loin
8         }
9     }
10    if(found)
11        System.out.println("6 est un des multiples de 2");
12 }
```

3 types d'interruptions de boucles

- `return`
- `break`
- `continue` : passer à l'itération suivante

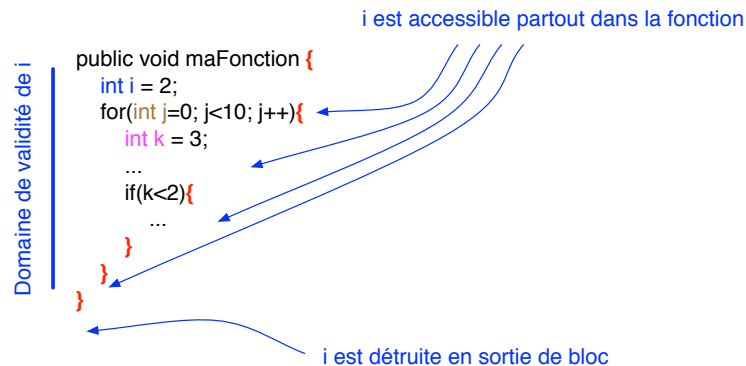
```
1 // afficher 3./i pour i variant de -10 à 10
2 // il faut penser à sauter le cas 0 qui provoque un problème
3 public void maFonction(){
4     for(int i=-10; i<=10; i++){ // -10 et 10 inclus
5         if(i == 0)
6             continue;
7         System.out.println("3./"+i+"="+ (3./i) );
8     }
9 }
```

Ces instructions rendent le code plus lisible en limitant notamment le nombre de blocs imbriqués.

Logique de bloc

- une fonction est un bloc,
- une boucle ou une conditionnelle forme également un bloc,
- les blocs sont repérés par des accolades : `{...}`

Les variables **déclarées** dans un bloc sont détruites en sortant du bloc.



STRING (SUITE)

2 choses à retenir sur les String

- 1 Les chaînes sont immutables : modifier une chaîne existante est impossible, il faut créer une nouvelle chaîne qui est une modification de l'ancienne. Cela rend la classe peu efficace dans certains cas... Et il faut alors se tourner vers des objets plus évolués (`StringBuffer` notamment)
- 2 Ne pas utiliser `==` avec les `String` mais toujours la méthode `.equals`. Les deux versions compilent mais la première donnera régulièrement des résultats faux (que nous expliquerons plus tard).

```

1 String s1 = "Leia";
2 String s2 = "Luke";
3 if( s1.equals(s2) )
4     System.out.println("Les chaînes sont identiques");
5 else
6     System.out.println("Les chaînes sont différentes");

```

Gestion des chaînes de caractères

`String` n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à Java et son caractère immuable la rapproche très nettement d'un type de base.

```

1 String s = "Luke"; // création d'une chaîne de caractères
2 s = s + "est le frère de Leia";
3 System.out.println(s); // affichage de s dans la console

```

⚠ Ne pas confondre l'objet `String` et l'affichage dans la console.

Les possibilités sont nombreuses : extraction de sous-chaînes (`substring`), division en plusieurs chaînes (`split`), recherche de caractères, construction de nouvelles chaînes à partir d'expressions régulières (`replace`)... Toute la documentation sur : <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

EXEMPLES DE FORMATAGES

`String.format(String format, Object... args)`

```

1 double[] tab = new double[10];
2 for(int i=0; i<10; i++)
3     tab[i] = Math.random()*1000;
4 for(int i=0; i<10; i++)
5     System.out.println(tab[i]);

```

```

1 510.4306229034564
2 775.6503067597263
3 15.528224029893511
4 ...

```

```

1 for(int i=0; i<10; i++)
2     System.out.println(
3         String.format("%12f", tab[i]));
4 ...

```

```

1      510,430623
2      775,650307
3      15,528224
4 ...

```

```

1 for(int i=0; i<10; i++)
2     System.out.println(
3         String.format("%10.3f", tab[i]));
4 ...

```

```

1      510,431
2      775,650
3      15,528
4 ...

```

```

1 for(int i=0; i<10; i++)
2     System.out.println(
3         String.format("%010.3f", tab[i]));
4 ...

```

```

1 000510,431
2 000775,650
3 000015,528
4 ...

```

Ca marche aussi avec les entiers (`%d`) et les `String` (`%s`)