

Chapitre 2 : Classes et Objets

- Objets
- Notion de classe
- Encapsulation
- Abstraction
- Modélisation UML
- Constructeurs
- Méthodes
- Visibilité
- Static
- Classes de base en Java
- Tableaux

Les objets

- Rappel
 - Programmation structurée :
 - Des variables
 - Des méthodes
 - Chaque méthode résout une partie du problème
 - Programmation OO :
 - Conception de l'architecture d'une programme à partir des objets (i.e données + fonctionnalités)

Les objets

- Un objet est défini par :
 - Une identité (*nom de la référence*)
 - Un état (*ensemble des attributs*)
 - Un comportement (*ensemble des fonctionnalités*)
- exemple : un téléphone
 - Identité : myPhone, ph2, toto,...
 - état : onOff, brand, price, weight, battery...
 - fonctionnalités :
 - **selecteurs** : getBrand(), getPrice(),...
 - **mutateurs** : turnOn(), turnOff(), phone(...), sms(...),...

Notion de classe

- Qu'est-ce qu'une classe :

On peut voir une classe comme un « moule » qui sert à construire des objets similaires (i.e. mêmes données et mêmes fonctionnalités)

Un objet est **une instance** d'une classe

exemple : Car.java

Encapsulation - Abstraction

- **Encapsulation :**

- Le principe d'encapsulation consiste à regrouper dans une même entité des données et des opérations.

- **Abstraction :**

- Le principe d'abstraction consiste à décrire un objet par des données et des fonctionnalités le décrivant **selon un contexte donné.**

On ne manipule **jamais(*)** directement l'état d'un objet, on passe **toujours** par les méthodes

Visibilité

La visibilité détermine si on a accès à l'élément :
dans la classe, dans les sous-classes, dans le package, ou partout ailleurs

Les visibilitées :


- `private` : seulement dans la classe
- + `public` : partout
- # `protected` : dans la classe, les sous-classes et le package
- package private* (pas de mot clé) : dans la classe et le package

Exemple : Car.java

Les Constructeurs

- permettent de créer des objets (à partir de données)
- portent le même nom que leur classe (commencent pas une majuscule)
- n'ont pas de type de retour

Pas de void ici !



```
public class Car{  
    public Car(...){  
        ...  
    }  
}
```

Les constructeurs

- Quels constructeurs définir ?
 - On peut distinguer deux sortes d'attributs :
 1. Ceux qui sont liés à l'état courant d'un objet (vitesse, niveau d'essence, niveau d'huile...)
 2. Ceux qui caractérisent l'objet dès sa création (couleur, puissance, immatriculation, capacité du réservoir...)
 - Usuellement :
 - Constructeur où tous les attributs ont une valeur par défaut
 - Constructeur où tous les attributs (2) sont indiqués
 - On peut en faire d'autres...

Les constructeurs

- Constructeurs remarquables :
 - Constructeur par défaut (sans paramètre)
`Car ()`
 - Constructeur par copie
`Car (Car c)`

Les constructeurs

- Règles sur les constructeurs :
 - Si aucun constructeur n'est défini dans une classe, alors cette classe possède un constructeur par défaut, défini de manière implicite.
 - **ATTENTION** : Dès qu'un constructeur est défini explicitement, le constructeur par défaut n'est plus défini implicitement (ce qui n'empêche pas de le définir explicitement).

Les constructeurs

- Création d'un objet

- Déclaration

- `Car myCar;`

- Instanciation

- `myCar = new Car();`

- Déclaration + Instanciation

- `Car myCar = new Car();`

exemple : CarConstructors.java

Valeur spéciale **null**

- Tout objet (quelle que soit sa classe) peut être affecté à la valeur `null`
- un objet non instancié vaut `null`

- **ATTENTION :**

```
Car c = null;  
Television tv = null;
```

MAIS on ne peut pas comparer `c` et `tv` car les types ne sont pas compatibles

L'objet courant **this**

- **A quoi ça sert ?**

- Désigner l'objet dans lequel on se trouve
- Passer en paramètre la référence de l'objet courant
- Appeler un constructeur dans un autre
`this(paramètres du constructeur appelé);`
cet appel doit être la première instruction du constructeur

- Note : l'objet courant **this** est une référence "cachée". On ne peut pas affecter une nouvelle valeur à **this**.

Les Méthodes

Une méthode est définie par :

- un nom,
- un type de retour (éventuellement void),
- des paramètres d'entrée (éventuellement aucun).

Un paramètre peut être :

- De type primitif
- Une référence d'objet typée par n'importe quelle classe

Appel de méthode :

```
anInstance.aMethod(...);
```

Les Méthodes

- En Java, tous les paramètres sont passés **par valeur**

Les fonctions agissent **sur des copies** des paramètres

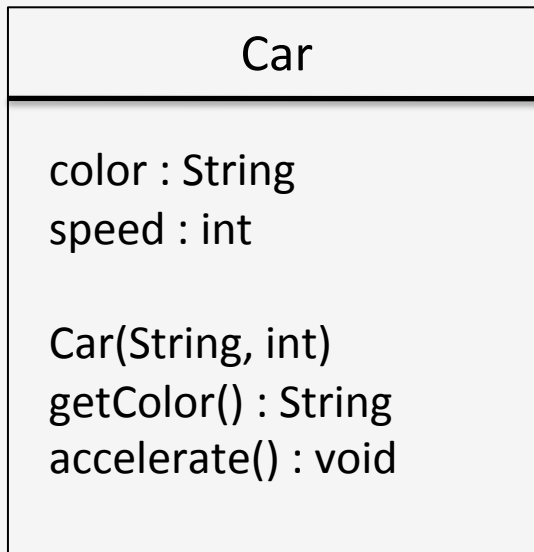
- types primitifs
- Objets

```
public f(MyClass obj){  
    // ici on peut agir sur l'objet référencé par obj,  
    // mais on travaille avec une copie de la référence  
    // obj  
}
```

Modélisation UML

Unified Modeling Language:

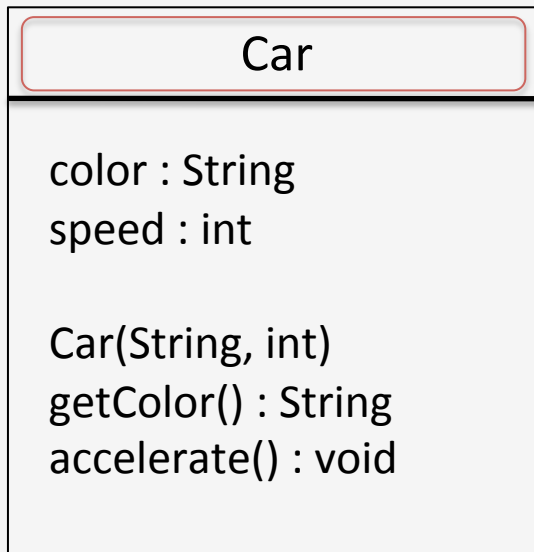
Langage graphique de modélisation de programmes OO.



Modélisation UML

Unified Modeling Language:

Langage graphique de modélisation de programmes OO.

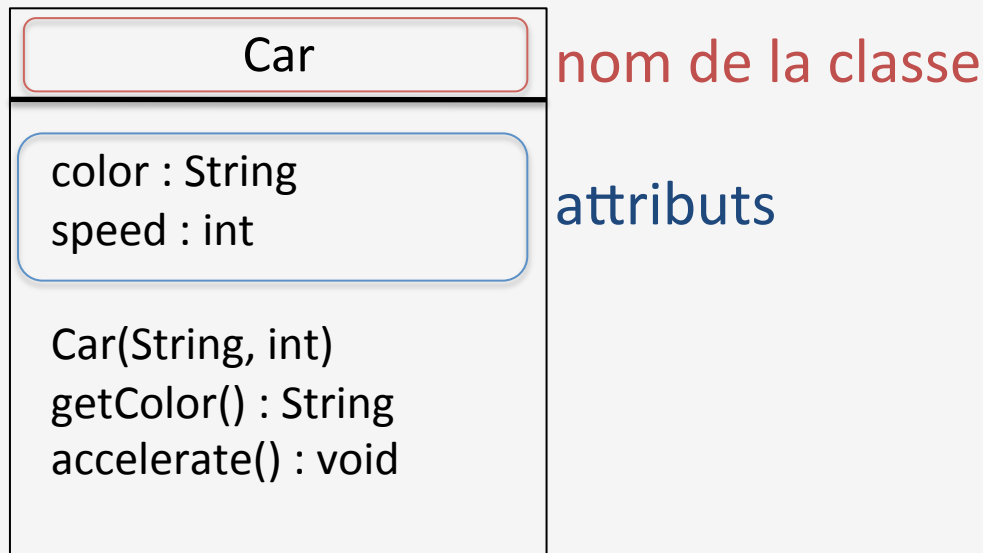


nom de la classe

Modélisation UML

Unified Modeling Language:

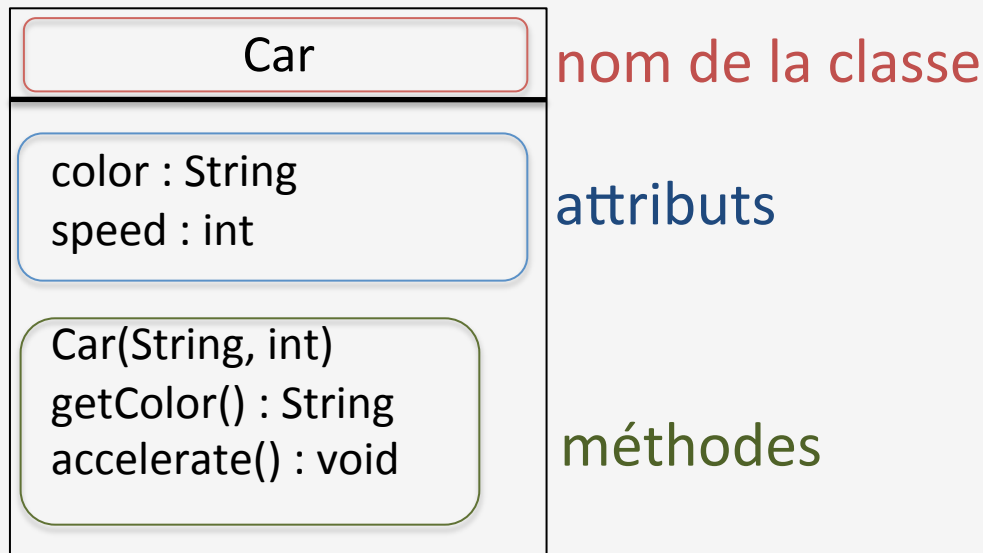
Langage graphique de modélisation de programmes OO.



Modélisation UML

Unified Modeling Language:

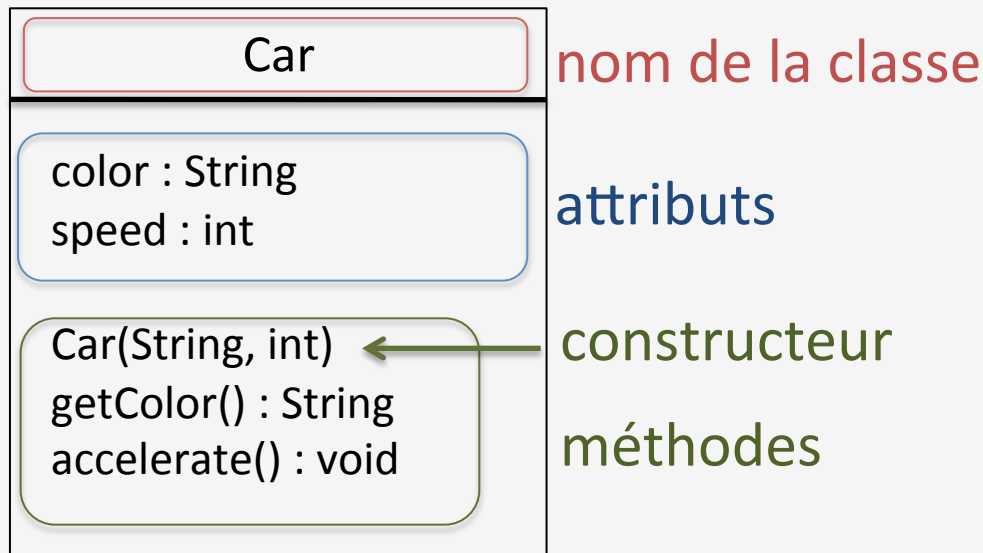
Langage graphique de modélisation de programmes OO.



Modélisation UML

Unified Modeling Language:

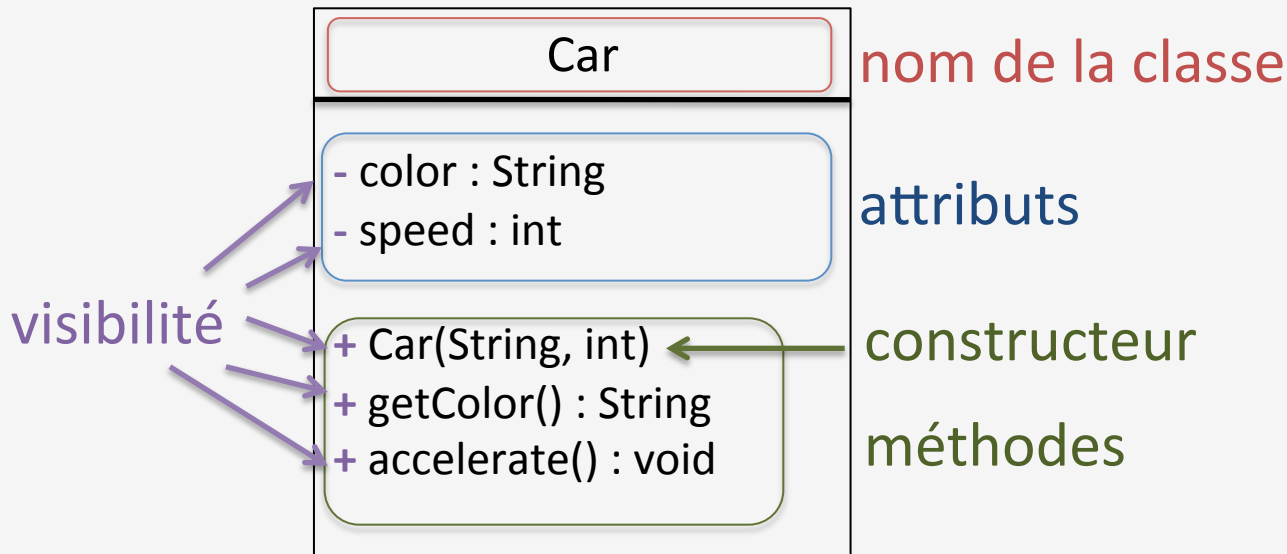
Langage graphique de modélisation de programmes OO.



Modélisation UML

Unified Modeling Language:

Langage graphique de modélisation de programmes OO.

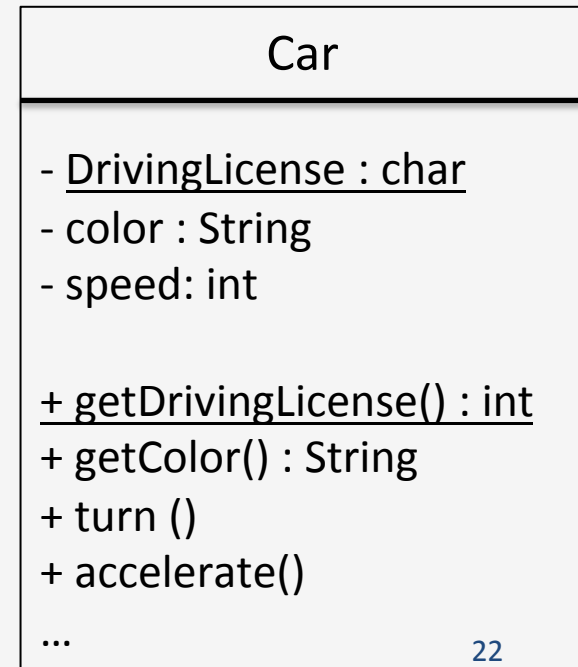


Le mot clé static

Une méthode ou un attribut **static** est **relatif à la classe**, et non à une instance particulière.

```
private static char DrivingLicense='B';  
public static int getDrivingLicense();
```

- Notation UML : souligné



Le mot clé static

Accès à un attribut/méthode static :

`ClassName.staticElement`

exemples :

`Car.getDrivingLicense()`

`Car.drivingLicense`

Notes :

- On ne peut pas utiliser l'objet courant `this` dans une méthode static (ça n'a pas de sens)
- On peut accéder à un attribut/méthode static via une instance de la classe.
- On peut utiliser un attribut ou une méthode static sans même avoir instancié d'objet de la classe.

Le mot clé static

- bloc d'initialisation static

Si l'initialisation d'une variable static ne peut pas se faire avec une simple affectation, on peut utiliser un bloc static.

```
private static Type myStaticInstance;  
static{  
    ... // bloc d'instructions  
}
```


Cycle de vie d'un objet

- Appel à un constructeur par l'opérateur `new` `ClassName(parameters)`
- Si un objet n'est plus référencé, alors le destructeur `finalize()` est appelé par la JVM juste avant de libérer l'espace mémoire.
- Le destructeur `finalize()` n'est jamais appelé explicitement.

Cycle de vie d'un objet

- La gestion de la mémoire s'effectue par le **garbage collector** (ramasse miette) qui est lancé régulièrement et automatiquement par la JVM.
- On peut suggérer à la JVM le lancer le GC par la commande `System.gc()`
- **A retenir** : En java, la JVM s'occupe de la libération de la mémoire. Lorsqu'un objet n'est plus référencé, on considèrera que la mémoire est libérée.

Réalisation d'un programme OO

1. On conçoit (en UML)
2. On implante

Réalisation d'un programme OO

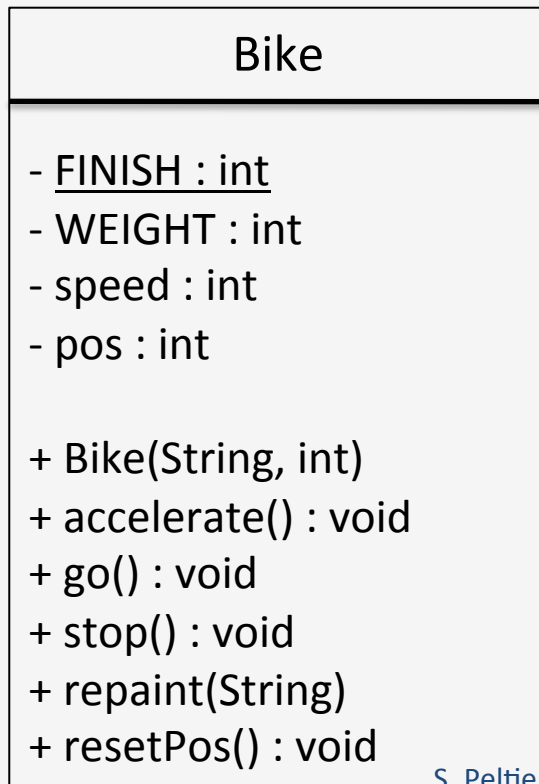
1. On conçoit (en UML)
2. On implante

On souhaite réaliser jeu de course de vélos en ligne droite. Chacun ayant un poids et une position définis à sa création. Un vélo peut accélérer, avancer et s'arrêter et repositionné au départ. Tous les vélos connaissent la position de la ligne d'arrivée.

Réalisation d'un programme OO

1. On conçoit (en UML)
2. On implante

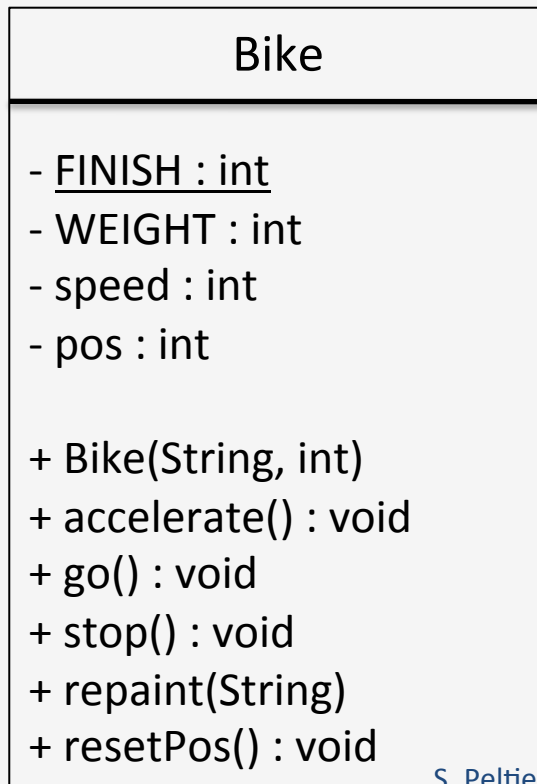
On souhaite réaliser jeu de course de vélos en ligne droite. Chacun ayant un poids et une position définis à sa création. Un vélo peut accélérer, avancer et s'arrêter et repositionné au départ. Tous les vélos connaissent la position de la ligne d'arrivée.



Réalisation d'un programme OO

1. On conçoit (en UML)
2. On implante

On souhaite réaliser jeu de course de vélos en ligne droite. Chacun ayant un poids et une position définis à sa création. Un vélo peut accélérer, avancer et s'arrêter et repositionné au départ. Tous les vélos connaissent la position de la ligne d'arrivée.



```
public class Bike {
    public final static int FINISH = 10;
    private final int WEIGHT ;
    private int speed;
    private int pos;

    public Bike(String col, int wei){
        this.WEIGHT = wei;
        this.speed = 0;
        this.pos = 0;
    }
    ...
}
```

Classes de base en Java

- Chaque type primitif a son analogue objet, muni de méthodes utiles
 - boolean -> Boolean
 - byte, short, int, long -> Byte, Short, Integer, LongInteger
 - Constantes de classes
MIN_VALUE ET MAX_VALUE
 - Conversion à partir de String :

```
String s = « 123 »;  
Integer i = Integer.parseInt(s);
```

Classes de base en Java

- Chaque type primitif a son analogue objet, muni de méthodes utiles
 - `char` -> `Character`
`isDigit()`, `isJavaLetter()`, `isLowerCase()`,
`toUpperCase()`...
 - `float`, `double` -> `Float`, `Double`
 - Valeurs spécifiques
`NaN` ATTENTION : `NaN != NaN`
`NEGATIVE_INFINITY`, `POSITIVE_INFINITY`
Le zéro négatif

exemple : API Java

Classes de base en Java

- La classe String
 - C'est une classe **non mutable**, qui n'a pas d'analogue primitif
 - Concaténation : +
`System.out.println("blabla" + 12 + ' ' + (2>3));`
 - méthode `length()` pour récupérer la longueur
 - pour tester si deux chaînes sont égales `s1.equals(s2)`
 - NB. Java effectue des optimisations mémoire, et dans certains cas, on a
`s1.equals(s2)` est équivalent à `s1==s2`.

Comparaison de variables

- Types primitifs
v1 == v2 retourne vrai si v1 et v2 ont la même valeur
- Pour les objets c'est pareil, mais **ATTENTION** : v1 et v2 sont des références. Donc v1 == v2 si et seulement si v1 et v2 référencent le même objet.

Si on souhaite comparer les attributs des objets, on peut :

- Définir une méthode ad hoc
`public boolean myEqualsMethod(MyClass x)`
- Redéfinir la méthode (on verra ça plus tard)
`@Override`
`public boolean equals(Object o)`

Comparaison de variables

- Types primitifs
v1 == v2 retourne vrai si v1 et v2 ont la même valeur
- Pour les objets c'est pareil, mais **ATTENTION** : v1 et v2 sont des références. Donc v1 == v2 si et seulement si v1 et v2 référencent le même objet.

Si on souhaite comparer les attributs des objets, on peut :

- Définir une méthode ad hoc
public boolean myEqualsMethod(MyClass x)
- Redéfinir la méthode (on verra ça plus tard)
@Override
public boolean equals(Object o)

*Ça aussi,
on verra plus tard*

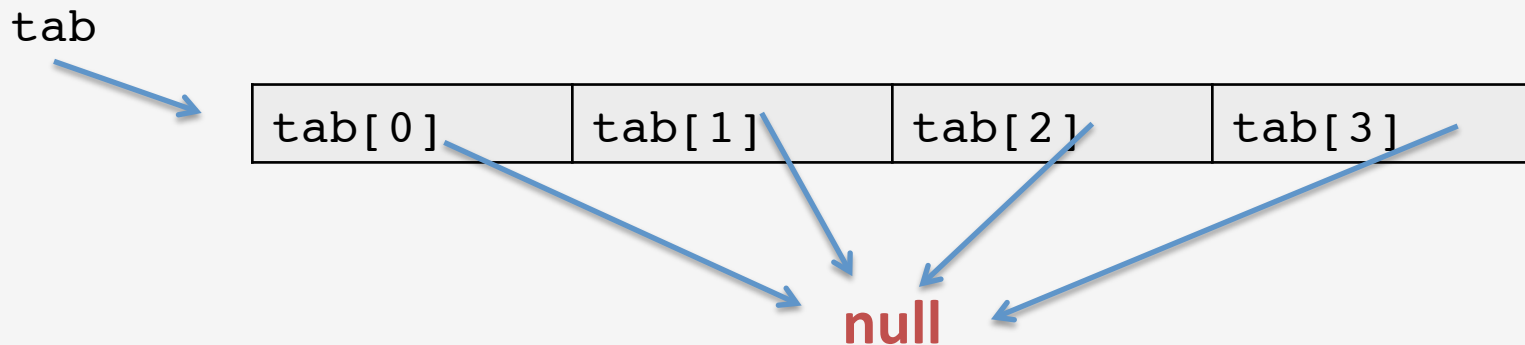
Les tableaux d'objets

- Même chose que pour les tableaux de types primitifs :

```
Type[] tab = new Type[size];
```

– ATTENTION :

Le tableau est instancié, mais chaque case du tableau contient une référence qui n'a pas été instanciée.



Affichage

- **Affichage à l'écran**

`System.out.print()` affichage sans retour à la ligne

`System.out.println()` affichage avec retour à la ligne

- **Différentes sorties possibles**

`out` : sortie standard

`err` : sortie en cas d'erreur

- **Ce que l'on peut afficher**

String, nombres, booléens, caractères...

Affichage

- **Affichage d'objets :**

```
System.out.print(myObject);
```

```
affichage : package.Classe@adresse
```

- **Redéfinir l'affichage :**

```
@Override  
public String toString() {  
...  
}
```