

**R1.01**

**INITIATION AU DÉVELOPPEMENT**

---

## **Cours 3 : notion de classe**
















**Hervé Blanchon & Anne Lejeune**

Université Grenoble Alpes

IUT 2 – Département Informatique

# Sommaire

---





-  Programmation orientée objet (POO)
-  Classes déjà rencontrées
  -  Scanner, String
-  Classe et constructeur
-  Classe Covoiturage
  -  définition des attributs et méthodes
-  Principe d'encapsulation
-  Classe Covoiturage
  -  l'encapsulation en pratique (codage de la classe)
-  Notion de classe utilitaire
-  Classe Covoiturage
  -  la classe CovoiturageUtilitaire
-  Classe principale (\_Main)
  -  utilisation de Covoiturage et CovoiturageUtilitaire
-  Conventions de nommage

# PROGRAMMATION ORIENTÉE OBJET

# Motivations et apports de la POO

---

## Objectifs de la POO

-  Améliorer la conception, l'exploitation & la maintenance
-  Programmer par « composants »
-  Faciliter la réutilisation du code
-  Faciliter l'évolution du code (nouvelles fonctionnalités)

## Apports de la POO

-  Objet et Classe
-  Encapsulation
-  *D'autres apports non mentionnés ici seront abordés en R2.01*

# Notions de classe et d'objet

---

 Une classe regroupe des membres


 propriétés ou attributs (des données)


 méthodes (des comportements – manipulations des données)

 fonctions

 procédures

 La notion de classe est une généralisation de la notion de type

 elle ajoute le comportement (méthodes) à la structure de données (attributs)

 Un **objet** est une **instance** (une **réalisation**) d'une classe à laquelle il appartient

 il peut être vu comme une variable initialisée dotée de méthodes (son comportement)

# Définition d'une classe en Java

---



Syntaxe :

```
public class Nom_De_La_Classe {  
  
    // attributs  
  
    // constructeurs  
  
    // méthodes  
  
}
```



Convention de nommage



Le nom d'une classe commence toujours par une lettre majuscule

# CLASSES DÉJÀ RENCONTRÉES

## (Scanner, String)

# La classe Scanner

---


 On a dit qu'un Scanner contenait une bande de caractères qui sont consommés

 cette bande constitue les données du Scanner


 On a présenté comment :

 déclarer et construire un Scanner avec l'instruction :

```
Scanner lecteur = new Scanner(System.in);
```

 *note jusqu'à présent on a dit déclarer et initialiser un Scanner, à partir de maintenant on dira construire au lieu d'initialiser*

 ce qui construit le Scanner c'est : `new Scanner(...)`

 ce qui dit que la bande de caractères est alimentée par les saisies au clavier, c'est : `System.in`


 `lecteur` est un objet de type Scanner (une instance de la classe Scanner) qui a donc des méthodes



# La classe Scanner


 On a présenté comment lire des données de différents types en présentant les instructions associées sans plus d'explications :

nom de la variable lue	Type à lire	<u>instruction</u> sur le Scanner lecteur
unByte	byte	<code>unByte = lecteur.nextByte();</code> <code>lecteur.nextLine(); // vider la bande</code>
unShort	short	<code>unShort = lecteur.nextShort();</code> <code>lecteur.nextLine(); // vider la bande</code>
unInt	int	<code>unInt = lecteur.nextInt();</code> <code>lecteur.nextLine(); // vider la bande</code>
unLong	long	<code>unLong = lecteur.nextLong();</code> <code>lecteur.nextLine(); // vider la bande</code>
unFloat	float	<code>unFloat = lecteur.nextFloat();</code> <code>lecteur.nextLine(); // vider la bande</code>
unDouble	double	<code>unDouble = lecteur.nextDouble();</code> <code>lecteur.nextLine(); // vider la bande</code>
unChar	char	<code>unChar = lecteur.next().charAt(0);</code> <code>lecteur.nextLine(); // vider la bande</code>
unBoolean	boolean	<code>unBoolean = lecteur.nextBoolean();</code> <code>lecteur.nextLine(); // vider la bande</code>
unString	String	<code>unString = lecteur.nextLine();</code>

 `nextByte()`, `nextShort()`, ... sont des méthodes, les comportements, proposés par la classe Scanner

# La classe Scanner

---


 Les méthodes `nextXX()` du tableau de la planche précédente sont des fonctions qui retournent un résultat de type `XX`

 la méthode `nextInt()` retourne un `int`

 en Java son entête est le suivant : `public int nextInt()`

 En écrivant `val = lecteur.nextInt();`

 on demande à l'objet `lecteur` d'exécuter sa méthode `nextInt()`

 la méthode `nextInt()` retourne, si possible, un `int` qui est rangé dans la variable `val`

 L'appel d'une méthode se fait donc avec la notation pointée

 **objet • nom\_de\_méthode(...)**

# La classe `String`

---



Nous avons utilisé des variables de type `String`



L'utilisation que nous en avons eu était très simple



déclaration d'une variable de type `String`



```
String uneChaine;
```



initialisation par affectation d'une variable de type `String`



```
uneChaine = "Je suis une chaîne !";
```



initialisation par lecture sur un Scanner d'une variable de type `String`



```
uneChaine = lecteur.nextLine(); // lecteur est un Scanner
```



affichage du contenu d'une variable de type `String`





```
System.out.println(uneChaine);
```









# La classe String

---

 Nous n'avons pas vu les **méthodes** de la classe

-  un extrait est proposé planche suivante
-  un exemple d'usage est proposé ensuite

## Notes

-  la classe **String** propose un constructeur de chaîne vide :
  -  `String sVide = new String();`
-  si on a besoin d'une chaîne pour une saisie, il suffit de la déclarer et de la lire ensuite sur un Scanner
  -  `String uneChaine;`
  -  `uneChaine = lecteur.nextLine(); // lecteur est un scanner`
-  remarquer que toutes les méthodes proposées sont des fonctions dont le résultat...
  -  peut être affiché
  -  peut être affecté à une variable du bon type

# Méthodes de la classe String (extrait)

Type du résultat	Méthode	Description
char	charAt(int index)	Retourne la valeur char à l'index spécifié.
int	compareTo(String anotherStr)	Compare deux chaînes lexicographiquement, sensible à la casse. Le résultat est – négatif si <b>cette chaîne</b> précède strictement anotherStr – positif si <b>cette chaîne</b> suit strictement anotherStr – 0 si <b>cette chaîne</b> est égale à anotherStr
int	compareToIgnoreCase(String anotherStr)	Compare deux chaînes lexicographiquement, en ignorant les différences de casse. Le résultat est – négatif si <b>cette chaîne</b> précède strictement lexicographiquement anotherStr – positif si <b>cette chaîne</b> suit strictement lexicographiquement anotherStr – 0 si <b>cette chaîne</b> est lexicographiquement égale à anotherStr
String	concat(String anotherStr)	Concatène anotherStr à la fin de <b>cette chaîne</b> .
boolean	equals(Object anObject)	Compare <b>cette chaîne</b> à l'objet spécifié, sensible à la casse.
boolean	equalsIgnoreCase(String anotherStr)	Compare <b>cette chaîne</b> à anotherStr, en ignorant les différences de casse.
boolean	isEmpty()	Renvoie true si, et seulement si, length() est 0.



**cette chaîne** est l'objet sur lequel on appelle la méthode  
par exemple avec `ch.nomMethode(...)`, **cette chaîne** est `ch`

## Méthodes de la classe String (extrait)

Type du résultat	Méthode	Description
int	length()	Retourne la longueur de <b>cette chaîne</b> .
String	replace(char oldChar, char newChar)	Retourne une chaîne résultant du remplacement de toutes les occurrences de oldChar dans <b>cette chaîne</b> par newChar.
String	stripLeading()	Retourne une chaîne dont la valeur est <b>cette chaîne</b> , privée des espaces* de tête.
String	stripTrailing()	Retourne une chaîne dont la valeur est <b>cette chaîne</b> , privée des espaces* de fin.
String	toLowerCase()	Convertit tous les caractères de <b>cette chaîne</b> en minuscules à l'aide des règles des paramètres régionaux par défaut.
String	toUpperCase()	Convertit tous les caractères de <b>cette chaîne</b> en majuscules à l'aide des règles des paramètres régionaux par défaut.
String	trim()	Retourne une chaîne qui est la valeur de <b>cette chaîne</b> , privée de tous les espaces* de tête et de fin.



**cette chaîne** est l'objet sur lequel on appelle la méthode  
par exemple avec `ch.nomMethode(...)`, **cette chaîne** est `ch`

\* *Un espace est tout caractère dont le code point est inférieur ou égal à « U+0020 » (le caractère espace).*



## Exemple d'utilisation de la classe String

code

```
1 public class TestClasseString {
2
3     public static void main(String[] args) {
4         String sVide = new String(); // construction d'une chaîne vide
5         String s1 = "abc", s2 = "DEF", s3 = "abc";
6         System.out.println("Chaîne sVide : " + sVide);
7         System.out.println("sVide.isEmpty() : " + sVide.isEmpty() + "(true attendu)");
8         System.out.println("Chaîne s1 : " + s1);
9         System.out.println("Chaîne s2 : " + s2);
10        System.out.println("Chaîne s3 : " + s3);
11        System.out.println("s1.equals(s2) retourne = " + s1.equals(s2) + " (false attendu)");
12        System.out.println("s1.equals(s3) retourne = " + s1.equals(s3) + " (true attendu)");
13        System.out.println("s2.toLowerCase() : " + s2.toLowerCase() + ", s2 : " + s2);
14        // remarquer de s2 n'est pas modifiée, s2.toLowerCase() retourne un String
15        System.out.println("s1.concat(s2) : " + s1.concat(s2));
16    }
17
18 }
```

trace

```
6 Chaîne sVide :
7 sVide.isEmpty() : true (true attendu)
8 Chaîne s1 : abc
9 Chaîne s2 : DEF
10 Chaîne s3 : abc
11 s1.equals(s2) retourne = false (false attendu)
12 s1.equals(s3) retourne = true (true attendu)
13 s2.toLowerCase() : def, s2 : DEF
15 s1.concat(s2) : abcDEF
```



## Autre exemple d'utilisation de la classe String

code

```
1 public class MethodesString {
2
3     public static void main(String[] args) {
4         String s1 = "ABC", s2 = "abc", s3 = "DEF";
5         char c = s1.charAt(0);
6         String sConcat = s1.concat(s2);
7         int compareToS1S2 = s1.compareTo(s2);
8         int compareToS2S1 = s2.compareTo(s1);
9         int compareToIgnoreCaseS1S2 = s1.compareToIgnoreCase(s2);
10
11         System.out.println("c = " + c);
12         System.out.println("concaténation de s1 avec s2 : " + sConcat);
13         System.out.println("s1.compareTo(s2) : " + compareToS1S2);
14         System.out.println("s2.compareTo(s1) : " + compareToS2S1);
15         System.out.println("s1.compareToIgnoreCase(s2) : " + compareToIgnoreCaseS1S2);
16     }
17 }
```

trace




```
11 c = A // le premier caractère d'un String est à la position 0
12 concaténation de s1 avec s2 : ABCabc
    // this string est s1, concat() retourne un String
13 s1.compareTo(s2) : -32
    // résultat négatif car s1 est plus petit que s2 lexicographiquement
14 s2.compareTo(s1) : 32
    // résultat positif car s2 est plus grand que s1 lexicographiquement
15 s1.compareToIgnoreCase(s2) : 0
    // résultat nul car en ignorant la casse s1 et s2 sont égales
```



# La classe String

---

 Dans l'exemple de code précédent

-  String est une classe (un type)
-  sVide, s1, s2, s3 sont des objets (des instances de la classe String)
-  isEmpty(), equals(Object anObject), toLowerCase(), concat(String str) sont des méthodes

# CLASSE ET CONSTRUCTEUR

# Classe et constructeur

---

 Avant de pouvoir utiliser un **objet**, il faut le construire

 chaque classe possède un ou plusieurs **constructeurs**

 un constructeur est une **méthode** publique qui porte le nom de la classe

 pour une classe **MaClasse**, on peut avoir les constructeurs suivants :

 **MaClasse()** // constructeur sans paramètre

 **MaClasse(type nom\_de\_paramètre, ...)** // constructeur avec paramètre(s)

 un constructeur est utilisé avec le mot-clé **new**

 Exemples

 `Scanner lecteur = new Scanner(System.in);`

 `String sVide = new String();`

# UNE PREMIÈRE CLASSE

## Covoiturage

# Besoins applicatif pour du covoiturage

---

 Dans une application de gestion de covoiturage

 les données relatives à un covoiturage sont :

 *une ville de départ*

 *une ville d'arrivée*

 *le nombre de kilomètres à parcourir*

 *le prix au kilomètre que fait payer le chauffeur au passager*

 *la possibilité pour le passager de voyager avec son animal de compagnie ou non*


 les traitements que l'on veut faire sur un covoiturage sont :

 *connaître la ville de départ*

 *connaître la ville d'arrivée*

 *connaître le nombre de kilomètres à parcourir*

 *savoir si le passager peut voyager avec son animal de compagnie*

 *connaître le prix à payer par le passager*

# La classe Covoiturage

## Attributs :

Nom attribut	Type attribut	Commentaire
villeDepart	String	<i>ville de départ du conducteur</i>
villeArrivee	String	<i>ville d'arrivée du conducteur</i>
nombreKm	int	<i>nombre de kilomètres à parcourir</i>
prixAuKm	float	<i>prix du covoiturage au kilomètre</i>
animalAutorise	boolean	<i>le conducteur accepte-il les animaux dans son véhicule ?</i>


## Méthodes :

nom méthode	type résultat	commentaire
getVilleDepart()	String	<i>retourne la valeur de l'attribut villeDepart</i>
getVilleArrivee()	String	<i>retourne la valeur de l'attribut villeArrivee</i>
getNombreKm()	float	<i>retourne la valeur de l'attribut nombreKm</i>
isAnimalAutorise()	boolean	<i>retourne la valeur de l'attribut animalAutorise</i>
getPrixCovoiturage()	float	<i>retourne le prix du Covoiturage (nombreKm x prixAuKm)</i>

# Instance de la classe Covoiturage (un objet)


 Une instance de la classe (un objet) est défini par :

 Un **ÉTAT**

 Représenté (caractérisé) par des données (**attributs**)

 Un **COMPORTEMENT**

 Défini par des procédures et fonctions (**méthodes**) qui modifient les données et envoient des messages à d'autres objets

 Une **IDENTITÉ** (unCovoiturage)

 Permet de distinguer un objet d'un autre objet



# PRINCIPE D'ENCAPSULATION





# Principe d'encapsulation (1/4)

---

*Un utilisateur extérieur ne doit pas modifier directement l'**ÉTAT** d'un objet (les données) et risquer de mettre en péril l'**ÉTAT** et le **COMPORTEMENT** de l'objet !*

 Comment ?

-  **Protéger les données** contenues dans un objet
-  **Proposer des méthodes pour manipuler les données** d'un objet qui assurent la validité/cohérence des données

# Principe d'encapsulation (2/4)


---

 Comment **protéger les données** contenues dans un objet ?

les **attributs** seront **privés**, c'est-à-dire consultables ou modifiables uniquement par des méthodes de la classe de l'objet

 Comment **proposer des méthodes pour manipuler** (consulter, modifier) **les données** d'un objet ?


les **méthodes** de la classe de l'objet **utilisables par un utilisateur de la classe** seront **publiques**

 **Conséquence** : la classe devra fournir des méthodes publiques pour consulter (*accesseurs* ou *getters*) et/ou modifier (*mutateurs* ou *setters*) les attributs quand c'est nécessaire en assurant validité/cohérence des données


# Principe d'encapsulation (3/4)

---

 Cohérence des données d'un objet ?

 les méthodes qui modifient des attributs doivent garantir la **validité/cohérence de l'objet** (la cohérence des valeurs données aux attributs)

 Les **ATTRIBUTS** d'une classe sont tous **PRIVÉS** !

 Les méthodes d'une classe sont-elles toutes publiques ?



 sont **PUBLIQUES** les **MÉTHODES UTILISABLES PAR UN UTILISATEUR** de la classe (*il a le droit de les utiliser*)

 sont **PRIVÉES** les **MÉTHODES DE « SERVICE »** aux méthodes publiques **NON UTILISABLES PAR UN UTILISATEUR** de la classe (*il n'a pas le droit de les utiliser*)

# Principe d'encapsulation (4/4)

---

## PRIVÉ vs PUBLIC

-  Les **membres** privés (**private**) ne sont visibles qu'à l'intérieur de la classe (par les méthodes de la classe donc)
-  Les **membres** publics (**public**) sont visibles par toutes les parties de programmes (par l'utilisateur de la classe ou par d'autres classes par exemple)

 **Note** : le terme **membre** désigne ici un attribut ou une méthode.

# PRINCIPE D'ENCAPSULATION APPLIQUÉ À LA CLASSE Covoiturage

# Application à la classe Covoiturage



Attributs et méthodes (constructeur traité plus loin)

```
public class Covoiturage {  
    // attributs privés (encapsulation)  
    private String villeDepart;  
    private String villeArrivee;  
    private int nombreKm;  
    private float prixAuKm;  
    private boolean animalAutorise;  
  
    // constructeurs  
    ... // voir planche suivante  
  
    // méthodes  
    public String getVilleDepart() {  
        return villeDepart;  
    }  
    public String getVilleArrivee() {  
        return villeArrivee;  
    }  
    public int getNombreKm() {  
        return nombreKm;  
    }  
    public boolean isAnimalAutorise() {  
        return animalAutorise;  
    }  
  
    /**  
     * calcul du prix de ce Covoiturage  
     * @return nombreKm * prixAuKm  
     */  
    public float getPrixCovoiturage() {  
        return nombreKm * prixAuKm;  
    }  
}
```

définition de la classe  
**Covoiturage**

les attributs sont colorisés par l'IDE  
– on sait ainsi lorsqu'on les utilise  
dans le code

documentation de la méthode  
– pour l'écrire positionner le curseur  
au dessus et saisir la séquence  
« /\*\*← », et compléter le canevas  
généré par l'IDE

# Application à la classe Covoiturage

## Constructeur



```
public class Covoiturage {
```

```
// attributs privés (encapsulation)
```

```
private String villeDepart;  
private String villeArrivee;  
private int nombreKm;  
private float prixAuKm;  
private boolean animalAutorise;
```

les attributs sont colorisés par l'IDE  
– on sait ainsi lorsqu'on les utilise  
dans le code

```
// constructeurs
```

```
/*
```

Ici, un seul constructeur avec 5 paramètres pour initialiser chacun des cinq attributs.

Un constructeur doit normalement initialiser tous les attributs.

Un attribut non initialisé explicitement à la valeur par défaut de son type.

Noter que les noms des paramètres formels (ici soulignés) sont les mêmes que les noms des attributs, on désambiguïse en utilisant la notation **this**.

Noter que l'attribut est bien colorisé après **this**.

```
*/
```

```
public Covoiturage(String villeDepart, String villeArrivee, int nombreKm,  
                    float prixAuKm, boolean animalAutorise) {
```

```
    this.villeDepart = villeDepart;
```

```
    this.villeArrivee = villeArrivee;
```

```
    this.nombreKm = nombreKm;
```

```
    this.prixAuKm = prixAuKm;
```

```
    this.animalAutorise = animalAutorise;
```

```
}
```

```
// méthodes
```

```
... // voir planche précédente
```

```
}
```

# Paramètres formels des constructeurs

Il est habituel de nommer les **paramètres formels** d'un constructeur en utilisant les noms des **attributs**

Dans le corps du constructeur, il faut donc distinguer l'usage des paramètres formels de celui des attributs

l'usage des attributs se marque préfixant l'attribut avec **this.**

**this** est « cet objet »

Exemple

```
this.villeDepart = villeDepart ;
```

attribut








paramètre formel



# NOTION DE CLASSE UTILITAIRE


# Pourquoi ?

---

-  Une classe `MaClasseUtilitaire` vise à définir des services pour la classe `MaClasse`
-  Un service est un traitement qui ne peut ou ne doit pas être réalisé sous forme de méthode de `MaClasse`
  -  les entrées/sorties sont dépendantes de l'application qui utilise la `MaClasse`
    -  dans les applications que l'on développe en R1.01, on utilise le terminal, les entrées/sorties se font donc au terminal
    -  vous ferez plus tard des applications avec interface graphique, les entrées/sorties se feront dans des fenêtres graphiques
  -  la production de données pour une application
  -  la sauvegarde et la récupération des données d'une application

# Comment ?

---

 Dans une classe utilitaire on va définir des fonctions et des procédures en utilisant le mot-clé **static**

 Fonction de classe utilitaire

```
public static type_retourné nom_Fonction(...) {...}
```

 Procédure de classe utilitaire

```
public static void nom_Procédure(...) {...}
```

# Utilisation des `static` d'une classe utilitaire

---

*Soit `MaClasseUtilitaire` définissant des procédures et fonctions `static`*



Pour utiliser les fonctions et procédures `static` de `MaClasseUtilitaire` dans une autre classe, il conviendra de préfixer leur appel par :

**`MaClasseUtilitaire.`**



## Exemple d'utilisation d'une **static** de classe utilitaire

### Classe utilitaire avec une fonction static

```
import java.util.Scanner;

public class MaClasseUtilitaire {
    public static int saisirUnEntierEntre0et10() {
        Scanner lecteur = new Scanner(System.in);
        int unEntier;
        System.out.print("Saisir un entier entre 0 et 10 : ");
        unEntier = lecteur.nextInt();
        lecteur.nextLine();
        while (unEntier < 0 | unEntier > 10) {
            System.out.println("Saisie erronée, recommencer...");
            System.out.print("Saisir un entier entre 0 et 10 : ");
            unEntier = lecteur.nextInt();
            lecteur.nextLine();
        }
        return unEntier;
    }
}
```

### Trace classe main

Saisir un entier entre 0 et 10 : **15**  
Saisie erronée, recommencer..  
Saisir un entier entre 0 et 10 : **12**  
Saisie erronée, recommencer..  
Saisir un entier entre 0 et 10 : **5**  
Vous avez saisi : 5

### Classe main utilisant la fonction static de la classe utilitaire

```
public class TestClasseUtilitaire_Main {
    public static void main(String[] args) {
        int i = MaClasseUtilitaire.saisirUnEntierEntre0et10();
        System.out.println("Vous avez saisi : " + i);
    }
}
```


# CLASSE UTILITAIRE DE LA CLASSE

## Covoiturage



# Les fonctionnalités attendues

---

## Attendus

-  Créer un Covoiturage par lecture des attributs
-  Afficher les attributs d'un Covoiturage

## Réalisation

-  nous n'allons pas mettre ces méthodes dans la classe Covoiturage
-  on va utiliser une classe CovoiturageUtilitaire qui va permettre la mise en place de ces services

# Squelette de la classe CovoiturageUtilitaire

```
public class CovoiturageUtilitaire {  
    /**  
     * saisie interactive des attributs d'un Covoiturage  
     * @return le Covoiturage saisi  
     */  
    public static Covoiturage saisirCovoiturage() { ...}  
  
    /**  
     * affichage d'un Covoiturage (ville de départ, ville d'arrivée,  
     * nombre de kilomètres, prix, voyage avec animal  
     * @param unCovoit à afficher  
     */  
    public static void afficherCovoiturage(Covoiturage unCovoit) {...}  
}
```



Noter que la fonction et la procédure sont



**public static**



voir planches suivantes pour le code de  
saisirCovoiturage et afficherCovoiturage



# Saisie d'un Covoiturage

```
/**
 * saisie interactive des attributs d'un Covoiturage
 * on considère que toutes les saisies sont correctes
 * @return le Covoiturage saisi
 */
public static Covoiturage saisirCovoiturage() {
    System.out.println("Saisie des informations d'un covoiturage :");
    Scanner lecteur = new Scanner(System.in);
    System.out.print(" - ville de départ : ");
    String villeDepart = lecteur.nextLine();
    System.out.print(" - ville d'arrivée : ");
    String villeArrivee = lecteur.nextLine();
    System.out.print(" - nombre de kilomètres du parcours (un réel positif) : ");
    int nombreKm = lecteur.nextInt();
    lecteur.nextLine();
    System.out.print(" - prix au kilomètre du parcours (un réel positif) : ");
    float prixAuKm = lecteur.nextFloat();
    lecteur.nextLine();
    System.out.print(" - animal de compagnie autorisé (true/false) : ");
    boolean animalAutorise = lecteur.nextBoolean();

    // construire et retourner un Covoiturage
    return new Covoiturage(villeDepart, villeArrivee,
                           nombreKm, prixAuKm, animalAutorise);
}
```

# Affichage d'un Covoiturage

```
/**
 * affichage d'un Covoiturage (ville de départ, ville d'arrivée,
 * nombre de kilomètres, prix, voyage avec animal
 * @param unCovoit à afficher
 */
public static void afficherCovoiturage(Covoiturage unCovoit) {
    System.out.println("Informations sur un covoiturage :");
    System.out.println(" - ville de départ : " + unCovoit.getVilleDepart());
    System.out.println(" - ville d'arrivée : " + unCovoit.getVilleArrivee());
    System.out.println(" - nombre de kilomètres de l'itinéraire : " + unCovoit.getNombreKm());
    System.out.println(" - prix du covoiturage : " + unCovoit.getPrixCovoiturage());
    if (unCovoit.isAnimalAutorise()) {
        System.out.println((" - voyage possible avec un animal de compagnie"));
    } else {
        System.out.println((" - voyage impossible avec un animal de compagnie"));
    }
}
```



Noter l'utilisation des getters et de la méthode `isAnimalAutorise` du paramètre formel `unCovoit`

# CLASSE PRINCIPALE UTILISANT Covoiturage ET CovouturageUtilitaire

# Classe Covoiturage\_Main

code de la classe	<pre>public class Covoiturage_Main {     public static void main(String[] args) {         // déclaration d'une variable de type Covoiturage         Covoiturage <u>unCovoiturage</u>;         // initialisation de la variable avec la fonction de saisie         // de la classe CovoiturageUtilitaire         <u>unCovoiturage</u> = CovoiturageUtilitaire.saisirCovoiturage();         // affichage de la variable avec la procédure d'affichage         // de la classe CovoiturageUtilitaire         CovoiturageUtilitaire.afficherCovoiturage(<u>unCovoiturage</u>);     } }</pre>
trace	<p>Saisie des informations d'un covoiturage :</p> <ul style="list-style-type: none"><li>- ville de départ : <b>Paris</b></li><li>- ville d'arrivée : <b>Brest</b></li><li>- nombre de kilomètres du parcours (un réel positif) : <b>591</b></li><li>- prix au kilomètre du parcours (un réel positif) : <b>0,128</b></li><li>- animal de compagnie autorisé (true/false) : <b>false</b></li></ul> <p>Informations sur un covoiturage :</p> <ul style="list-style-type: none"><li>- ville de départ : Paris</li><li>- ville d'arrivée : Brest</li><li>- nombre de kilomètres de l'itinéraire : 591</li><li>- prix du covoiturage : 75.648</li><li>- voyage impossible avec un animal de compagnie</li></ul>

# CONVENTIONS DE NOMMAGE

# Conventions de nommage

---

## Vocabulaire

### Notation chameau (camel case)

#### Exemple

 NotationChameauHaute

(UpperCamelCase)

 notationChameauBasse

(lowerCamelCase)

### Notation serpent (snake case)

#### Exemple

 notation\_serpent

(snake\_case)

 NOTATION\_SERPENT\_HAUTE

(UPPER\_SNAKE\_CASE)

## Usage en Java

 variable et méthodes : lowerCamelCase

 classe : UpperCamelCase

 constante : UPPER\_SNAKE\_CASE

# Conventions de nommage proposées par Oracle

Quoi ?	Règle de nommage	Exemples
Package	<p>Le préfixe d'un nom de package unique est toujours écrit en lettres ASCII minuscules et doit être l'un des noms de domaine de premier niveau, actuellement com, edu, gov, mil, net, org ou l'un des codes anglais à deux lettres identifiant les pays comme spécifié dans la norme ISO 3166, 1981.</p> <p>Les composants suivants du nom du package varient en fonction des conventions d'affectation de noms internes d'une organisation. Ces conventions peuvent spécifier que certains composants de nom d'annuaire soient des noms de division, de service, de projet, d'ordinateur ou de connexion.</p>	<pre>com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese</pre>
Classe	<p>Les noms de classes doivent être des noms, en casse mixte avec la première lettre de chaque mot interne en majuscules. Essayez de garder vos noms de classe simples et descriptifs. Utilisez des mots entiers - évitez les acronymes et les abréviations (sauf si l'abréviation est beaucoup plus largement utilisée que la forme longue, telle que URL ou HTML).</p>	<pre>class Point; class PointCardinal;</pre>
Méthode	<p>Les méthodes doivent être des verbes, en casse mixte avec la première lettre minuscule, avec la première lettre de chaque mot interne en majuscules.</p>	<pre>getPrix(); calculerPrix(); afficherPoint(); saisirPoint();</pre>
Variables	<p>Les noms de variables sont en casse mixte avec une première lettre minuscule. Les mots internes commencent par des majuscules. Les noms de variables ne doivent pas commencer par des caractères de soulignement _ ou de signe dollar \$, même si les deux sont autorisés.</p> <p>Les noms de variables doivent être courts mais significatifs. Le choix d'un nom de variable doit être mnémonique, c'est-à-dire conçu pour indiquer à l'observateur occasionnel l'intention de son utilisation. Les noms de variables à un caractère doivent être évités, à l'exception des variables temporaires « jetables ». Les noms communs des variables temporaires sont i, j, k, m et n pour les entiers ; c, d et e pour les caractères.</p>	<pre>int i; char c; float laLargeur;</pre>
Constantes	<p>Les noms des constantes doivent être tous en majuscules avec des mots séparés par des traits de soulignement (« _ »).</p>	<pre>final int MIN = 4; final int MAX = 999; final int TEMP_CANICULE = 38;</pre>