



# **Introduction à la programmation et au développement d'applications en Java**

**Classes**

# Introduction

- ❑ Classes
- ❑ Utilisation d'une classe
- ❑ Les classes : des types « références »
- ❑ Encapsulation et modificateurs d'accès
- ❑ Retour sur les méthodes
- ❑ *Accessors* et *mutators* (*Getters* et *Setters*)

# Classes en Java

- ❑ Java : langage orienté objets
- ❑ Les objets **encapsulent** les données, les opérations et les sémantiques d'usage
- ❑ Stockage et détails de manipulation sont **cachés**
- ❑ Objectif : séparer
  - ❑ les **services rendus** par l'objet (ce qui intéresse les clients de l'objet, concepteurs d'une application)
  - ❑ de **comment** ils sont rendus (en interne, par les concepteurs de la classe)
- ❑ Classe : fournit une structure pour la **description et la création d'objets**

# Classe : définition

- ❑ Classe : **patron** de création d'un objet
- ❑ Du point de vue des données :
  - ❑ **classe** = formulaire vide
  - ❑ **objet** = formulaire rempli
- ❑ Déclaration : **mot-clé class** suivi du nom de la classe
- ❑ Le **nom du fichier** doit en général être le même que le nom de la classe (avec extension `.java`)
- ❑ Totalité du **contenu entre accolades**

*(fichier Vol.java)*

```
class Vol {
```

```
}
```

# Classe : définition

- ❑ Une classe est donc constituée :
  - ❑ d'un **état** (données)
  - ❑ de **code exécutable** (méthodes)
- ❑ Les **attributs (fields)** stockent l'état
- ❑ Les **méthodes** contiennent le code qui manipule l'état et exécute les opérations

*(fichier Vol.java)*

```
class Vol {  
  
    int nbPassagers;  
    int nbPlaces;  
  
    public ajouterUnPassager() {  
        if (nbPassagers < nbPlaces) {  
            nbPassagers++;  
        }  
    }  
}
```

# Classe : définition

- ❑ Une classe contient également au moins une méthode appelée **constructeur**
- ❑ Méthode spéciale contenant le code appelé au moment de la **création** d'un objet pour **initialiser l'état**
- ❑ Une classe peut contenir plusieurs constructeurs selon les besoins

*(fichier Vol.java)*

```
class Vol {  
  
    int nbPassagers;  
    int nbPlaces;  
  
    Vol() {  
        nbPassagers = 150;  
        nbPlaces = 0;  
    }  
  
    public ajouterUnPassager() {  
        if (nbPassagers < nbPlaces) {  
            nbPassagers++;  
        }  
    }  
}
```

# Classe : utilisation

```
Vol parisGeneve;
```

parisGeneve



# Classe : utilisation

```
Vol parisGeneve;  
parisGeneve = new Vol();
```

Dans le code qui va utiliser la classe, on utilise le mot-clé **new** pour créer une **instance de la classe (= un objet)**

parisGeneve





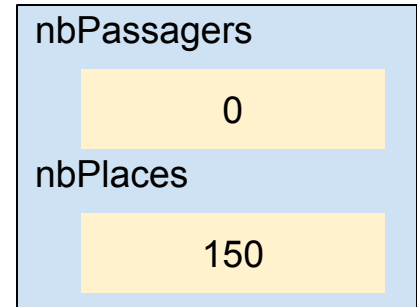
# Classe : utilisation

```
Vol parisGeneve;  
parisGeneve = new Vol();
```

Dans le code qui va utiliser la classe, on utilise le mot-clé **new** pour créer une **instance de la classe (= un objet)**

- ❑ **alloue la mémoire** nécessaire pour stocker l'objet

parisGeneve



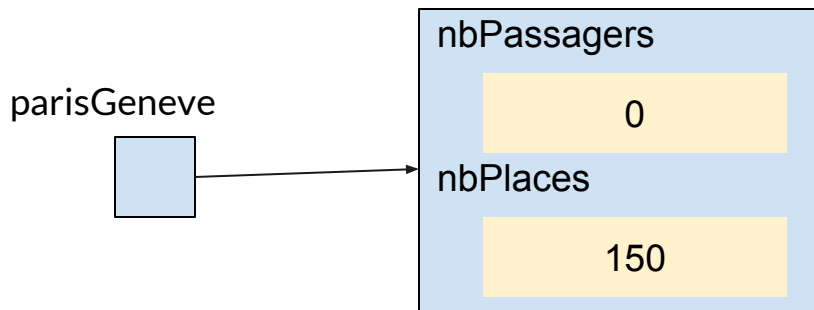
# Classe : utilisation

```
Vol parisGeneve;  
parisGeneve = new Vol();
```

Dans le code qui va utiliser la classe, on utilise le mot-clé **new** pour créer une **instance de la classe (= un objet)**

- ❑ **alloue la mémoire** nécessaire pour stocker l'objet
- ❑ retourne une **référence** qui désigne l'**adresse en mémoire** de l'objet alloué

La variable `parisGeneve` n'est pas l'objet lui-même, mais une **référence** à cet objet



# Classe : utilisation

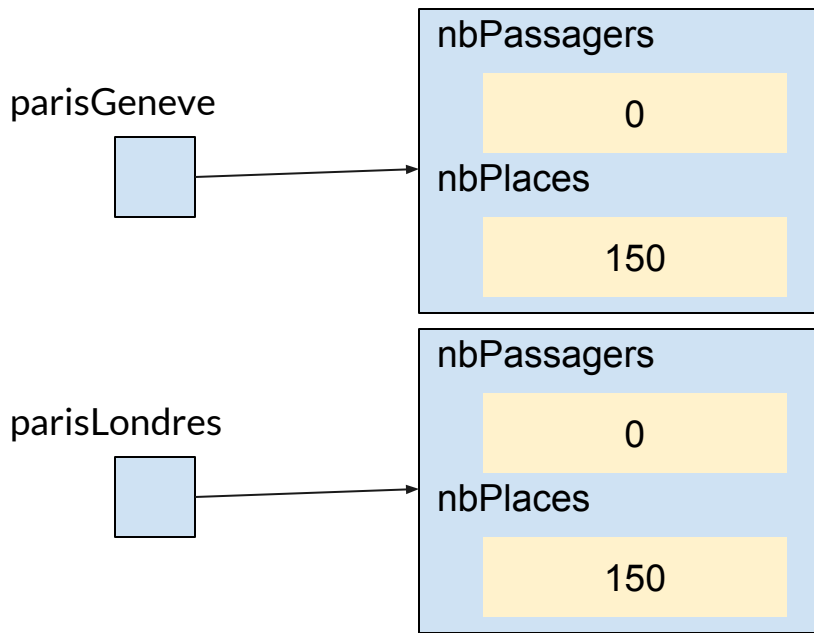
Dans le code qui va utiliser la classe, on utilise le mot-clé **new** pour créer une **instance de la classe (= un objet)**

- ❑ **alloue la mémoire** nécessaire pour stocker l'objet
- ❑ retourne une **référence** qui désigne l'**adresse en mémoire** de l'objet alloué

La variable `parisGeneve` n'est pas l'objet lui-même, mais une **référence** à cet objet

```
Vol parisGeneve;  
parisGeneve = new Vol();
```

```
Vol parisLondres = new Vol();
```



# Classe : utilisation

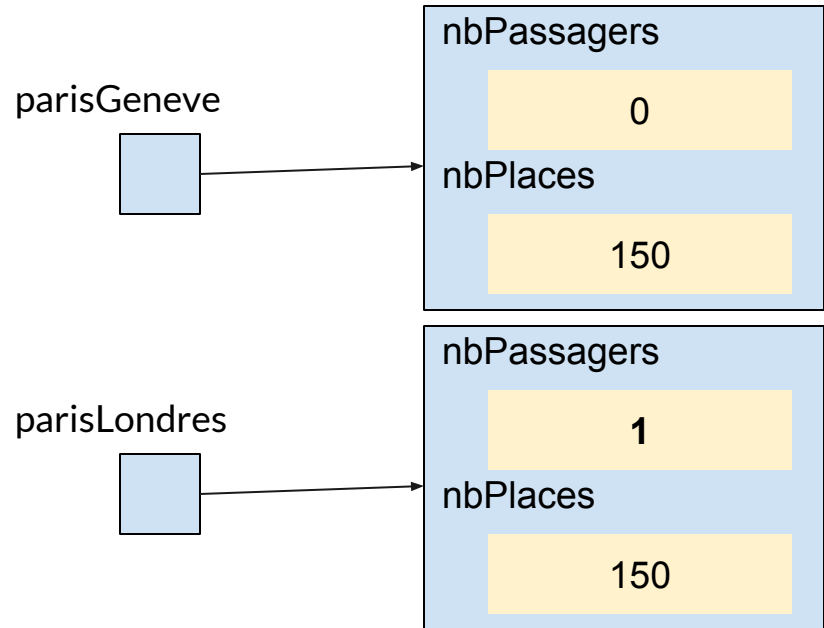
Dans le code qui va utiliser la classe, on utilise le mot-clé **new** pour créer une **instance de la classe (= un objet)**

- ❑ **alloue la mémoire** nécessaire pour stocker l'objet
- ❑ retourne une **référence** qui désigne l'**adresse en mémoire** de l'objet alloué

La variable `parisGeneve` n'est pas l'objet lui-même, mais une **référence** à cet objet

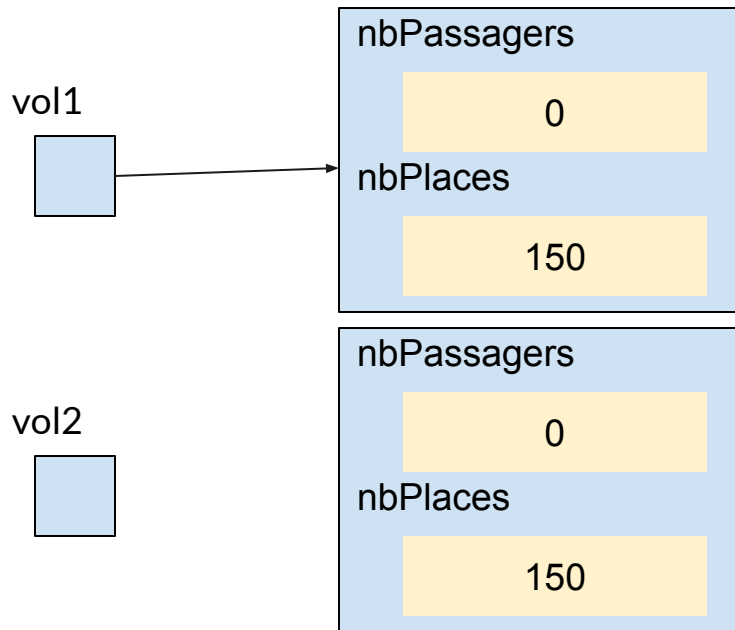
```
Vol parisGeneve;  
parisGeneve = new Vol();
```

```
Vol parisLondres = new Vol();  
parisLondres.ajouterUnPassager();
```



# Les classes : des types *références*

```
Vol vol1 = new Vol();  
Vol vol2 = new Vol();  
  
vol2.ajouterUnPassager();  
System.out.println(vol1.nbPassagers); //  
0  
System.out.println(vol2.nbPassagers); //  
1  
  
vol2 = vol1;  
System.out.println(vol2.nbPassagers); //  
0  
  
vol1.ajouterPassager();  
vol1.ajouterPassager();
```



# Encapsulation : modificateurs d'accès

- ❑ La représentation interne d'un objet est cachée  
(au maximum, et dans l'idéal complètement)
- ❑ Ce concept s'appelle l'**encapsulation**
- ❑ En Java, on utilise des **modificateurs d'accès** pour atteindre ce but :
  - ❑ *aucun* (modificateur d'accès par défaut)
  - ❑ `public`
  - ❑ `private`

# Modificateurs d'accès basiques

Modificateur	Visibilité	Utilisable sur classes	Utilisable sur membres (attributs et méthodes)
<i>aucun</i>	seulement dans son propre package ( <b>package private</b> )	oui	oui
<b>public</b>	partout	oui	oui
<b>private</b>	seulement dans sa propre classe	non	oui

oui dans les classes internes



# Appliquer les modificateurs d'accès

```
class Vol {  
    int nbPassagers ;  
    int nbPlaces;  
  
    Vol() {  
        nbPlaces = 150;  
        nbPassagers = 0;  
    }  
  
    void ajouterUnPassager() {  
        if (nbPassagers < nbPlaces)  
            nbPassagers++;  
        else  
            gererTropNombreux();  
    }  
  
    void gererTropNombreux() { ... }  
}
```



# Appliquer les modificateurs d'accès

```
Vol parisBerlin;
```

```
class Vol {  
    int nbPassagers ;  
    int nbPlaces;  
  
    Vol() {  
        nbPlaces = 150;  
        nbPassagers = 0;  
    }  
  
    void ajouterUnPassager() {  
        if (nbPassagers < nbPlaces)  
            nbPassagers++;  
        else  
            gererTropNombreux();  
    }  
  
    void gererTropNombreux() { ... }  
}
```

# Appliquer les modificateurs d'accès

```
Vol parisBerlin;
```

```
public class Vol {  
    int nbPassagers ;  
    int nbPlaces;  
  
    Vol() {  
        nbPlaces = 150;  
        nbPassagers = 0;  
    }  
  
    void ajouterUnPassager() {  
        if (nbPassagers < nbPlaces)  
            nbPassagers++;  
        else  
            gererTropNombreux();  
    }  
  
    void gererTropNombreux() { ... }  
}
```

# Appliquer les modificateurs d'accès

```
Vol parisBerlin = new Vol();

public class Vol {
    int nbPassagers ;
    int nbPlaces;

    public Vol() {
        nbPlaces = 150;
        nbPassagers = 0;
    }

    void ajouterUnPassager() {
        if (nbPassagers < nbPlaces)
            nbPassagers++;
        else
            gererTropNombreux();
    }

    void gererTropNombreux() { ... }
}
```

# Appliquer les modificateurs d'accès

```
Vol parisBerlin = new  
Vol();
```

```
public class Vol {  
    private int nbPassagers;  
    private int nbPlaces;  
  
    public Vol() {  
        nbPlaces = 150;  
        nbPassagers = 0;  
    }  
  
    void ajouterUnPassager() {  
        if (nbPassagers < nbPlaces)  
            nbPassagers++;  
        else  
            gererTropNombreux();  
    }  
  
    void gererTropNombreux() { ... }  
}
```

# Appliquer les modificateurs d'accès

```
Vol parisBerlin = new Vol();  
  
System.out.println(  
    parisBerlin.nbPassagers);  
  
public class Vol {  
    private int nbPassagers;  
    private int nbPlaces;  
  
    public Vol() {  
        nbPlaces = 150;  
        nbPassagers = 0;  
    }  
  
    void ajouterUnPassager() {  
        if (nbPassagers < nbPlaces)  
            nbPassagers++;  
        else  
            gererTropNombreux();  
    }  
  
    void gererTropNombreux() { ... }  
}
```

# Appliquer les modificateurs d'accès

```
Vol parisBerlin = new Vol();  
  
System.out.println(  
    parisBerlin.nbPassagers);  
  
public class Vol {  
    private int nbPassagers;  
    private int nbPlaces;  
  
    public Vol() {  
        nbPlaces = 150;  
        nbPassagers = 0;  
    }  
  
    void ajouterUnPassager() {  
        if (nbPassagers < nbPlaces)  
            nbPassagers++;  
        else  
            gererTropNombreux();  
    }  
  
    void gererTropNombreux() { ... }  
}
```

# Appliquer les modificateurs d'accès

```
Vol parisBerlin = new Vol();  
parisBerlin.ajouterUnPassager();  
  
public class Vol {  
    private int nbPassagers ;  
    private int nbPlaces;  
  
    public Vol() {  
        nbPlaces = 150;  
        nbPassagers = 0;  
    }  
  
    public void ajouterUnPassager() {  
        if (nbPassagers < nbPlaces)  
            nbPassagers++;  
        else  
            gererTropNombreux();  
    }  
  
    void gererTropNombreux() { ... }  
}
```

# Appliquer les modificateurs d'accès

```
Vol parisBerlin = new Vol();
parisBerlin.ajouterUnPassager()
;
parisBerlin.gererTropNombreux() ;

public class Vol {
    private int nbPassagers ;
    private int nbPlaces;

    public Vol() {
        nbPlaces = 150;
        nbPassagers = 0;
    }

    public void ajouterUnPassager() {
        if (nbPassagers < nbPlaces)
            nbPassagers++;
        else
            gererTropNombreux();
    }

    private void gererTropNombreux() { ... }
}
```



# Appliquer les modificateurs d'accès

```
Vol parisBerlin = new Vol();
parisBerlin.ajouterUnPassager();
;
parisBerlin.gererTropNombreux();

public class Vol {
    private int nbPassagers ;
    private int nbPlaces;


    public Vol() {
        nbPlaces = 150;
        nbPassagers = 0;
    }

    public void ajouterUnPassager() {
        if (nbPassagers < nbPlaces)
            nbPassagers++;
        else
            gererTropNombreux();
    }

    private void gererTropNombreux() { ... }
}
```

# Appliquer les modificateurs d'accès

- ❑ Les classes publiques doivent avoir le même nom que le fichier dans lequel elles sont définies
- ❑ Ce n'est pas nécessaire pour les classes *package private* (sans modificateur d'accès)

```
public class Vol {  Vol.java
    private int nbPassagers ;
    private int nbPlaces;
```

```
    public Vol() {
        nbPlaces = 150;
        nbPassagers = 0;
    }
```

```
    public void ajouterUnPassager() {
        if (nbPassagers < nbPlaces)
            nbPassagers++;
        else
            gererTropNombreux();
    }
```

```
    private void gererTropNombreux() { ... }
}
```

# Nommage des classes

- ❑ Convention de nommage : `PascalCase`
  - ❑ chaque mot commence par une majuscule
  - ❑ tout le reste en minuscules
- ❑ ex. : `CompteBanque`, `Personne`, `PanierDeCommande`
- ❑ Utilisez des **noms communs** descriptifs
- ❑ Évitez les abbréviations, sauf si celles-ci sont plus courantes dans le domaine concerné que le nom complet (ex. : URL)

# Méthodes dans le contexte OO

- ❑ Code exécutable qui manipule l'état et exécute les opérations
- ❑ **Nom**
  - ❑ idem que variables : `camelCase` (première lettre en minuscule)
  - ❑ utilisez un **verbe** ou décrivez une **action** (`afficherTableau`, `toString...`)
- ❑ **Type de retour** (`void` quand pas de valeur de retour)
- ❑ Liste de **paramètres typés** (peut être vide)
- ❑ Corps de la méthode entre **accolades**

`afficherSomme`

# Méthodes dans le contexte OO

- ❑ Code exécutable qui manipule l'état et exécute les opérations
- ❑ **Nom**
  - ❑ idem que variables : `camelCase` (première lettre en minuscule)
  - ❑ utilisez un **verbe** ou décrivez une **action** (`afficherTableau`, `toString...`)
- ❑ **Type de retour** (`void` quand pas de valeur de retour)
- ❑ Liste de **paramètres typés** (peut être vide)
- ❑ Corps de la méthode entre **accolades**

```
void afficherSomme
```

# Méthodes dans le contexte OO

- ❑ Code exécutable qui manipule l'état et exécute les opérations
- ❑ **Nom**
  - ❑ idem que variables : `camelCase` (première lettre en minuscule)
  - ❑ utilisez un **verbe** ou décrivez une **action** (`afficherTableau`, `toString...`)
- ❑ **Type de retour** (`void` quand pas de valeur de retour)
- ❑ Liste de **paramètres typés** (peut être vide)
- ❑ Corps de la méthode entre **accolades**

```
void afficherSomme(double x, double y, int nbAffichages)
```

# Méthodes dans le contexte OO

- ❑ Code exécutable qui manipule l'état et exécute les opérations
- ❑ **Nom**
  - ❑ idem que variables : `camelCase` (première lettre en minuscule)
  - ❑ utilisez un **verbe** ou décrivez une **action** (`afficherTableau`, `toString...`)
- ❑ **Type de retour** (`void` quand pas de valeur de retour)
- ❑ Liste de **paramètres typés** (peut être vide)
- ❑ Corps de la méthode entre **accolades**

```
void afficherSomme(double x, double y, int nbAffichages) {  
  
    }  
}
```

# Méthodes dans le contexte OO

- ❑ Code exécutable qui manipule l'état et exécute les opérations
- ❑ **Nom**
  - ❑ idem que variables : `camelCase` (première lettre en minuscule)
  - ❑ utilisez un **verbe** ou décrivez une **action** (`afficherTableau`, `toString...`)
- ❑ **Type de retour** (`void` quand pas de valeur de retour)
- ❑ Liste de **paramètres typés** (peut être vide)
- ❑ Corps de la méthode entre **accolades**

```
void afficherSomme(double x, double y, int nbAffichages) {  
    double somme = x + y;  
    for (int i = 0; i < nbAffichages; i++) {  
        System.out.println(somme);  
    }  
}
```



# Méthodes dans le contexte OO

```
void afficherSomme(double x, double y, int nbAffichages) {  
    double somme = x + y;  
    for (int i = 0; i < nbAffichages; i++) {  
        System.out.println(somme);  
    }  
}
```

# Méthodes dans le contexte OO

```
public class MaClasse {  
  
    void afficherSomme(double x, double y, int nbAffichages) {  
        double somme = x + y;  
        for (int i = 0; i < nbAffichages; i++) {  
            System.out.println(somme);  
        }  
    }  
}
```

# Méthodes dans le contexte OO

```
public class MaClasse {  
  
    void afficherSomme(double x, double y, int nbAffichages) {  
        double somme = x + y;  
        for (int i = 0; i < nbAffichages; i++) {  
            System.out.println(somme);  
        }  
    }  
}
```

```
MaClasse maClasse = new MaClasse();  
maClasse.afficherSomme(1.0, 2.3, 3);
```

```
> 3.3  
> 3.3  
> 3.3
```

# Sortie de méthode

- ❑ Quand une méthode est terminée, le contrôle retourne à l'appelant
- ❑ Trois façons de sortir d'une méthode :
  - ❑ Fin de la méthode (accolade fermante atteinte)
  - ❑ Instruction `return` rencontrée
  - ❑ Erreur se produit pendant l'exécution de la méthode

```
void afficherSomme(double x, double y, int nbAffichages) {  
    if (nbAffichages < 1) {  
        return;  
    }  
    double somme = x + y;  
    for (int i = 0; i < nbAffichages; i++) {  
        System.out.println(somme);  
    }  
}
```

# Valeur de retour d'une méthode

Quand une méthode n'a pas un type **void**, elle **doit** retourner une valeur unique :

- ❑ valeur primitive
- ❑ référence (y compris vers tableau)

# Valeur de retour d'une méthode

Quand une méthode n'a pas un type **void**, elle **doit** retourner une valeur unique :

- ❑ valeur primitive
- ❑ référence (y compris vers tableau)

```
public class Vol {  
    private int nbPassagers ;  
    private int nbPlaces;  
  
    public boolean aDeLaPlacePour(Vol autre) {  
        int total = nbPassagers  
                    + autre.nbPassagers;  
        if (total <= nbPlaces)  
            return true;  
        else  
            return false;  
    }  
}
```

# Valeur de retour d'une méthode

Quand une méthode n'a pas un type **void**, elle **doit** retourner une valeur unique :

- ❑ valeur primitive
- ❑ référence (y compris vers tableau)

```
public class Vol {  
    private int nbPassagers ;  
    private int nbPlaces;  
  
    public boolean aDeLaPlacePour(Vol autre) {  
        int total = nbPassagers  
                    + autre.nbPassagers;  
        return total <= nbPlaces;  
    }  
}
```

# Valeur de retour d'une méthode

Quand une méthode n'a pas un type **void**, elle **doit** retourner une valeur unique :

- ❑ valeur primitive
- ❑ référence (y compris vers tableau)

```
public class Vol {  
    private int nbPassagers ;  
    private int nbPlaces;  
  
    public boolean aDeLaPlacePour(Vol autre) {  
        int total = nbPassagers  
                    + autre.nbPassagers;  
        return total <= nbPlaces;  
    }  
  
    public Vol nouveauEnCombinantAvec(Vol autre){  
        Vol nouveau = new Vol();  
        nouveau.nbPlaces = nbPlaces;  
        nouveau.nbPassagers = nbPassagers  
                                + autre.nbPassagers;  
        return nouveau;  
    }  
}
```



# Valeur de retour d'une méthode

```
Vol vol1 = new Vol();  
Vol vol2 = new Vol();  
// Ajout de passagers...
```

```
Vol vol3;  
if (vol1.aDeLaPlacePour(vol2))  
{  
    vol3 =  
  
vol1.creerEnCombinant(vol2);  
}
```

```
public class Vol {  
    private int nbPassagers;  
    private int nbPlaces;  
  
    public boolean aDeLaPlacePour(Vol autre) {  
        int total = nbPassagers  
                    + autre.nbPassagers;  
        return total <= nbPlaces;  
    }  
  
    public Vol creerEnCombinant(Vol autre) {  
        Vol nouveau = new Vol();  
        nouveau.nbPlaces = nbPlaces;  
        nouveau.nbPassagers = nbPassagers  
                                + autre.nbPassagers;  
        return nouveau;  
    }  
}
```

# Références spéciales : `this` et `null`

Java fournit des références spéciales ayant une signification prédéfinie :

- ❑ `this` : référence à l'objet courant
  - ❑ éliminer les ambiguïtés
  - ❑ utiliser l'objet courant dans un appel de méthode

# Références spéciales : `this` et `null`

Java fournit des références spéciales ayant une signification prédéfinie :

- ❑ `this` : référence à l'objet courant
  - ❑ éliminer les ambiguïtés
  - ❑ utiliser l'objet courant dans un appel de méthode

```
public class Vol {  
  
    private int nbPassagers ;  
    private int nbPlaces;  
  
    public boolean aDeLaPlacePour(Vol autre) {  
        int total = nbPassagers  
                    + autre.nbPassagers;  
        return total <= nbPlaces;  
    }  
}
```

# Références spéciales : `this` et `null`

Java fournit des références spéciales ayant une signification prédéfinie :

- ❑ `this` : référence à l'objet courant
  - ❑ éliminer les ambiguïtés
  - ❑ utiliser l'objet courant dans un appel de méthode

```
public class Vol {  
  
    private int nbPassagers ;  
    private int nbPlaces;  
  
    public boolean aDeLaPlacePour(Vol autre) {  
        int total = this.nbPassagers  
                    + autre.nbPassagers;  
        return total <= nbPlaces;  
    }  
}
```

# Références spéciales : `this` et `null`

Java fournit des références spéciales ayant une signification prédéfinie :

- ❑ `null` (littéral)
  - ❑ « aucun objet »
  - ❑ peut être affecté à n'importe quelle variable de type référence

# Références spéciales : `this` et `null`

Java fournit des références spéciales ayant une signification prédéfinie :

- ❑ `null` (littéral)
  - ❑ « aucun objet »
  - ❑ peut être affecté à n'importe quelle variable de type référence

```
Vol vol1 = new Vol();  
Vol vol2 = new Vol();  
// Ajout de passagers...
```

```
Vol vol3;  
if (vol1.aDeLaPlacePour(vol2)) {  
    vol3 = vol1.creerEnCombinant(vol2);  
}
```

```
// ici vol3 peut ne pas avoir été initialisée  
// => Erreur de compilation si tentative  
// d'utiliser vol3
```

# Références spéciales : `this` et `null`

Java fournit des références spéciales ayant une signification prédéfinie :

- ❑ `null` (littéral)
  - ❑ « aucun objet »
  - ❑ peut être affecté à n'importe quelle variable de type référence

```
Vol vol1 = new Vol();  
Vol vol2 = new Vol();  
// Ajout de passagers...
```

```
Vol vol3 = null;  
if (vol1.aDeLaPlacePour(vol2)) {  
    vol3 = vol1.creerEnCombinant(vol2);  
}  
// ici vol3 est initialisé,  
// que la branche 'if' ait été exécutée ou non  
// Il faut toujours tester une référence  
// potentiellement null avant de l'utiliser  
if (vol3 != null) { ... }
```

# Encapsulation des champs

- ❑ La plupart du temps, un champ (attribut) d'une classe **ne devrait pas être directement accessible** en dehors de la classe
- ❑ Les champs font partie de la « couche bas niveau » de l'implémentation de la classe, ils ont une **probabilité de changement** potentiellement importante
- ❑ L'encapsulation des champs aide donc à **cacher les détails d'implémentation**
- ❑ Si on a vraiment besoin d'accéder à ces champs, il faut les « protéger » au maximum en n'offrant l'accès que par l'intermédiaire de **méthodes dédiées**



# Accessors et mutators (getters/setters)

- ❑ *accessor/mutator pattern* pour contrôler l'accès à un champ

```
public class Vol {  
  
    private int nbPassagers ;  
    private int nbPlaces;  
    // autres membres non repris ici  
  
}
```

# Accessors et mutators (getters/setters)

- ❑ *accessor/mutator pattern* pour

contrôler l'accès à un champ

- ❑ *accessor* pour lire la valeur du

champ

- ❑ `getNomChamp()`

- ❑ souvent appelé *getter*

```
public class Vol {  
  
    private int nbPassagers ;  
    private int nbPlaces;  
    // autres membres non repris ici  
  
    public int getNbPlaces() {  
        return nbPlaces;  
    }  
}
```

# Accessors et mutators (getters/setters)

- ❑ *accessor/mutator pattern* pour contrôler l'accès à un champ
  - ❑ *mutator* pour **modifier** la valeur du champ
  - ❑ **set**NomChamp (**valeur**)
  - ❑ souvent appelé *setter*
- ```
public class Vol {  
    private int nbPassagers ;  
    private int nbPlaces;  
    // autres membres non repris ici  
  
    public int getNbPlaces() {  
        return nbPlaces;  
    }  
  
    public void setNbPlaces(int nbPlaces) {  
        nbPlaces = nbPlaces;  
    }  
}
```

# Accessors et mutators (getters/setters)

- ❑ *accessor/mutator pattern* pour contrôler l'accès à un champ
  - ❑ *mutator* pour **modifier** la valeur du champ
  - ❑ **set**NomChamp (**valeur**)
  - ❑ souvent appelé *setter*
- ```
public class Vol {  
    private int nbPassagers ;  
    private int nbPlaces;  
    // autres membres non repris ici  
  
    public int getNbPlaces() {  
        return nbPlaces;  
    }  
  
    public void setNbPlaces(int nbPlaces) {  
        this.nbPlaces = nbPlaces;  
    }  
}
```

# *Accessors et mutators (getters/setters)*

```
Vol lilleParis = new Vol();  
lilleParis.setNbPlaces(180);  
  
System.out.println(  
  
lilleParis.getNbPlaces());
```

```
public class Vol {  
  
    private int nbPassagers ;  
    private int nbPlaces;  
    // autres membres non repris ici  
  
    public int getNbPlaces() {  
        return nbPlaces;  
    }  
  
    public void setNbPlaces(int nbPlaces) {  
        this.nbPlaces = nbPlaces;  
    }  
}
```

# Résumé

- ❑ Classe = *template* (patron) pour objets
  - ❑ déclarée avec le mot-clé `class`
  - ❑ les instances (objets) sont allouées par le mot-clé `new`
- ❑ Classe = type référence (la variable ne désigne pas directement l'objet, mais une référence à cet objet, copie de référence ≠ copie d'objet)
- ❑ Modificateurs d'accès pour contrôler l'encapsulation
- ❑ Méthodes pour manipuler l'état et exécuter des opérations (`return` pour contrôler la sortie de méthode en retournant éventuellement une valeur)
- ❑ Les champs mémorisent l'état de l'objet, il ne faut les exposer que si nécessaire