



Réutiliser et spécialiser les classes

- L'héritage -





SOMMAIRE

| | |
|--|----|
| ✓ 1 - Description de la classe <i>Personne</i> . | 1 |
| ✓ 2 - Description de la classe <i>Salarie</i> . | 3 |
| ✓ 3 - La classe <i>Salarie</i> en <i>Java</i> – Version 1. | 4 |
| ✓ 4 - Définition de <i>l'héritage</i> . | 5 |
| ✓ 5 - Règles d'héritage en <i>Java</i> . | 6 |
| ✓ 6 - Les <i>constructeurs</i> dans un contexte d'héritage. | 10 |
| ✓ 7 - Un constructeur efficace pour la classe <i>Salarie</i> . | 12 |
| ✓ 8 - <i>Redéfinition</i> dans une classe dérivée d'une méthode héritée. | 14 |
| ✓ 9 - Rappel : les références <i>this</i> et <i>super</i> | 15 |
| ✓ 10 - Contrôler les accès aux propriétés héritées (variables d'instance et méthodes) | 16 |
| ✓ 11 - Le modificateur d'accès <i>protected</i> | 17 |
| ✓ 12 - Le <i>concept d'héritage</i> et les <i>packages</i> . | 18 |
| ✓ 13 - Package et modificateurs d'accès en <i>Java</i> . | 19 |
| ✓ 14 - Modificateurs d'accès : <i>récapitulatif</i> | 20 |
| ✓ 15 - Accès dans les sous-classes avec <i>protected</i> . | 22 |
| ✓ 16 - Choix de conception : <i>protected</i> ou <i>private</i> ? | 24 |
| ✓ 17 - Version finale de la classe <i>Salarie</i> | 26 |
| ✓ 18 - Redéfinir une méthode : <i>récapitulons</i> . | 28 |

| | |
|--|----|
| ✓ 19 - Le cast induit par l'héritage. | 29 |
| ✓ 20 - Les classes abstraites . | 34 |
| ✓ 21 - Précisions sur les classes abstraites | 42 |
| ✓ 22 - Enregistrer les instances issues de la hiérarchie <i>Vehicule</i> dans une base de données. | 43 |
| ✓ 24 - L'interface <i>Inventoriable</i> | 51 |
| ✓ 25 - Héritage, classes abstraites et interfaces | 53 |



- 1 - Description de la classe *Personne*

Etablissons *le cahier des charges* suivant visant à décrire le type abstrait *Personne* :

- ✓ Une personne a un *nom* patronymique (invariable) devant être stocké en majuscules. Si le nom est inconnu, on adoptera la convention consistant à valoriser ce nom avec la chaîne « *Nom inconnu* ».
- ✓ Une personne a un *âge*. Cet âge sera stocké sous la forme d'un entier compris entre 1 et 120. Si l'âge de la personne est inconnu, on adoptera la convention consistant à valoriser cet âge avec 0 (zéro).
- ✓ Une personne est capable de se décrire lorsque l'on lui demande.

Ecrivons en *Java* une première ébauche de la classe *Personne*, conforme à cette description minimale.

```
public class Personne
```

```
    private String nom;           // Nom de la personne
```

```
    private int age;             // Son âge
```

```
    // Accesseur en écriture
```

```
    private void setAge(int age) {
```

```
        // Ici on fait appel à une méthode filtre qui rejette toute valeur
```

```
        //incorrecte de l'âge
```

```
        // .....
```

```
        this.age = age; // Ici l'âge convient
```

```
    }
```

```
    //-----
```

```
    // Constructeur à deux paramètres
```

```
    //-----
```

```
    public Personne(String nom, int age) {
```

```
        this.nom = nom.toUpperCase();
```

```
        if (age != 0 ) setAge (age);
```

```
    }
```

```
    //-----
```

```
    // Constructeur par défaut rétabli
```

```
    //-----
```

```
    public Personne () {           // On convient qu'une personne
```

```
        nom = « Nom inconnu » ; // dont on ne sait rien aura sa v.i.
```

```
        age = 0 ;                 // nom valorisée avec « Nom
```

```
                                   // inconnu » et sa v.i. age valorisée
```

```
                                   // avec la valeur entière 0.
```

```
    }
```

```
    public void afficher() {
```

```
        System.out.println (« Je m'appelle » + nom);
```

```
        if ( age != 0 )
```

```
            System.out.println (« et j'ai » + age + « ans . ») ;
```

```
    }
```

```
} // class Personne
```

- 2 - Description de la classe *Salarie*

Etablissons maintenant le cahier des charges suivant, visant à décrire le type abstrait *Salarie* :



- ✓ Un salarié **est une** personne employée dans une entreprise.
- ✓ Un salarié **a** donc une entreprise qui l'emploie. Le nom de cette entreprise sera stocké en majuscule. Le nom de cette entreprise peut changer : le salarié peut changer d'employeur.
- ✓ « Un salarié **est une** personne » : la relation « **est une** » traduit ici le fait que la description que nous avons faite pour la classe *Personne* est valable pour la classe *Salarie*.
- ✓ « Un salarié a une entreprise » : la relation traduit ici le fait qu'un salarié possèdera une variable d'instance (**v.i.**) représentant l'entreprise. Cette v.i. (variable d'instance) sera *societe*.

Un salarie est une personne qui travaille dans une société.

1

La classe *Salarie* reprend donc toutes les propriétés de la classe *Personne*....

2

... en les complétant.

- ✓ La relation « **est-un** » traduit la notion objet d'héritage.
- ✓ Héritons alors la classe *Salarie* à partir de la classe *Personne*.

- 3 - La classe *Salarie* en Java – Version 1

```
public class Salarie extends Personne { // extends exprime l'héritage

    private String societe;

    // Accesseur en modification ou mutator
    public void setSociete(String societe) {
        this.societe = societe.toUpperCase();
    }
    // Constructeur
    public Salarie (String societe) {
        setSociete(societe);
    }
} // class Salarie
```

- ✓ La classe *Salarie*, ainsi construite, hérite de tout le potentiel de sa classe mère, c'est-à-dire la classe *Personne*.
- ✓ Toutes les propriétés appartenant à la classe *Personne* sont désormais, par ce mécanisme d'héritage, disponibles dans la classe *Salarie* appelée aussi **classe fille**.

On peut, dès lors, exécuter le code suivant :

```
Salarie dupont = new Salarie(« Java SARL ») ;
dupont.afficher() ;
```

- ✓ Constat : on peut appeler la méthode *afficher()* sur l'objet référencé par *dupont* (de type *Salarie*) bien qu'elle ne fasse pas partie des propriétés de *Salarie* : c'est par le biais de l'héritage que l'on peut utiliser cette méthode qui a été déclarée **public** dans la classe de base *Personne*.

Ce principe est fondamental en P.O.O. et représente la seconde notion caractérisant l'objet.

- 4 - Définition de l'héritage

- ✓ Une **classe** représente une famille d'objets ayant les **mêmes propriétés** (variables d'instances) et les **mêmes méthodes**.
- ✓ L'héritage est un concept objet puissant qui permet de reprendre les caractéristiques d'une classe existante *ClasseMere* - pour les **étendre** - et définir ainsi une nouvelle classe *ClasseFille* qui hérite de *ClasseMere*.
- ✓ La classe *ClasseMere* est aussi appelée « classe de base » et *ClasseFille* : « classe dérivée ».
- ✓ Représentons graphiquement ce phénomène d'héritage :

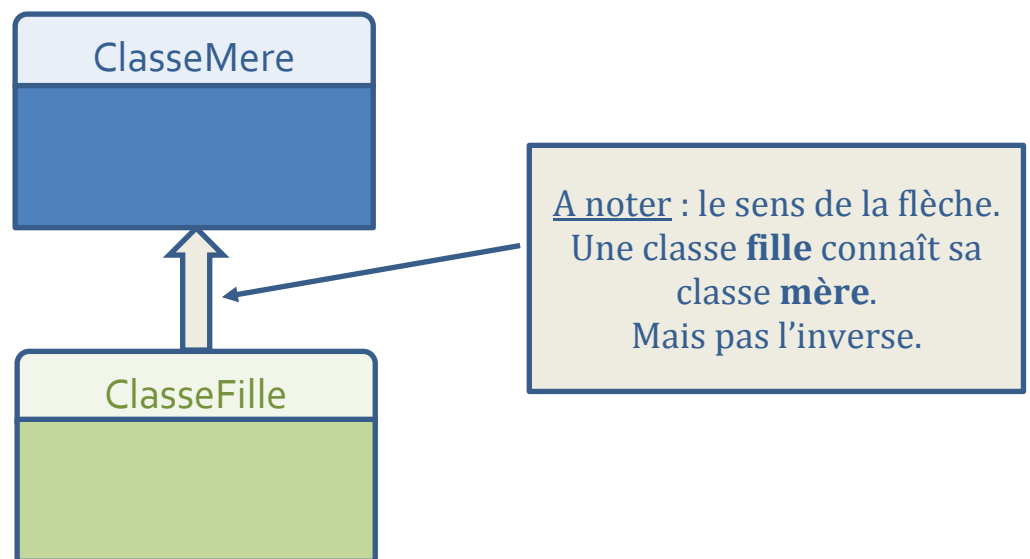


Figure 1 : L'héritage

- 5 - Règles d'héritage en Java

Plusieurs règles d'héritage, spécifiques à Java, caractérisent cette notion :

- ✓ Une classe fille ne peut hériter que d'une seule classe mère. On parle d'héritage simple. (En d'autres termes : pas d'héritage multiple en Java).
- ✓ En revanche, plusieurs classes filles peuvent hériter d'une même classe mère. On retrouve cette règle sous le schéma suivant :

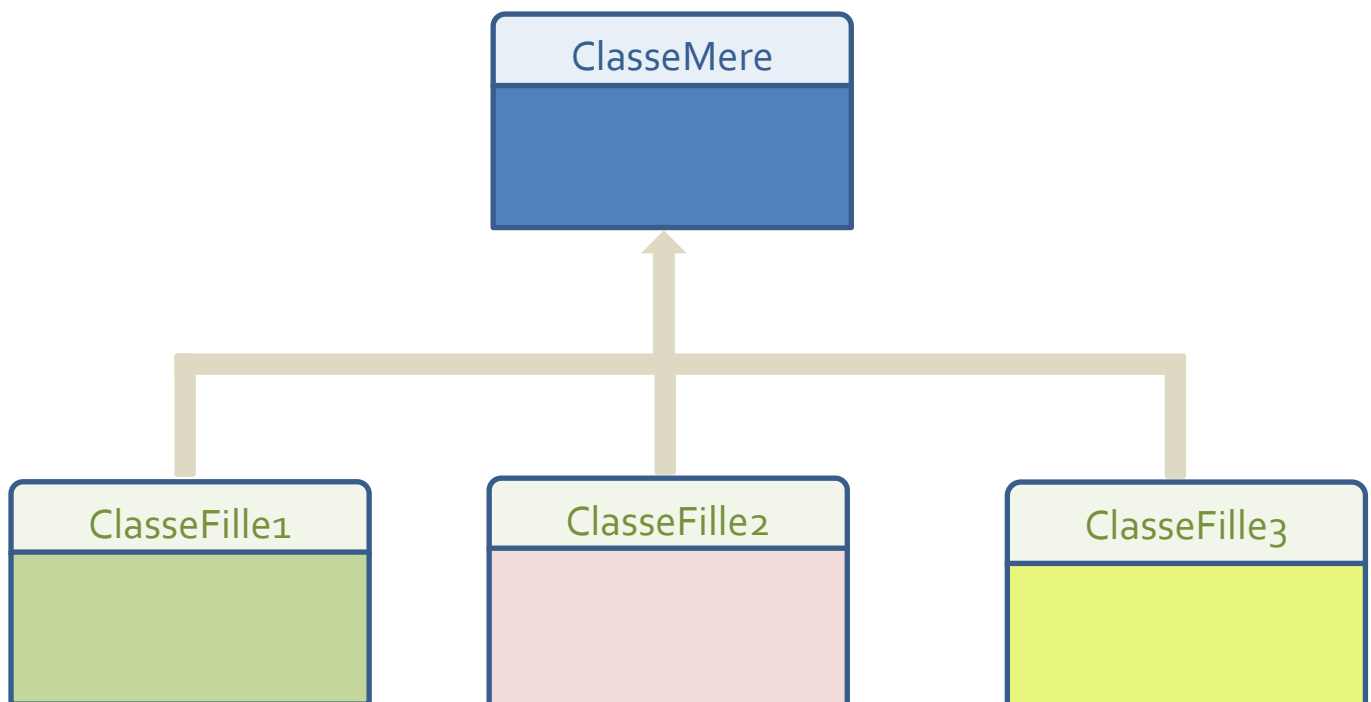


Figure 2 : Plusieurs classes filles pour une seule classe mère.

- ✓ Une classe fille peut elle-même être la classe mère d'une nouvelle classe. On retrouve cette règle sous le schéma suivant :

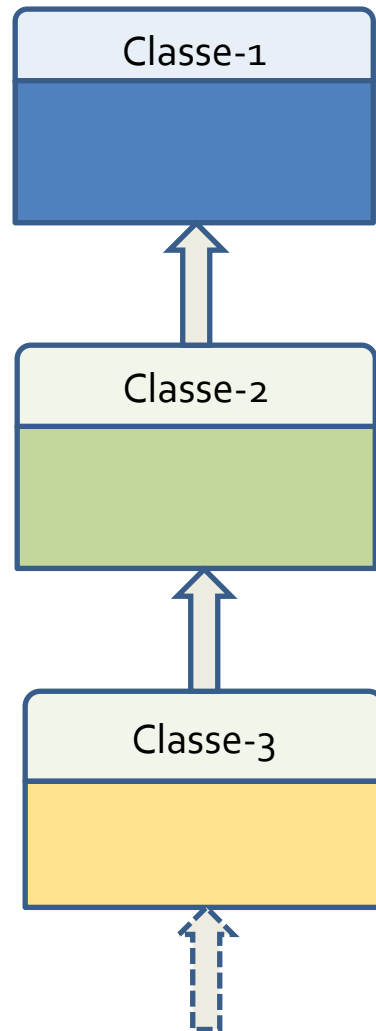


Figure 3 : Une classe fille (Classe-2) peut être à son tour classe mère.

Il n'y a **pas de limites** dans les niveaux. On obtient ainsi **un graphe de classes** liées entre elles par des concepts d'héritage. La relation « **est un** » prend toute sa dimension :

- ✓ un objet construit à partir de la **Classe-1** est de « type **Classe-1** »
- ✓ un objet construit à partir de la **Classe-2** est de « type **Classe-2** » mais aussi de « type **Classe-1** ».
- ✓ un objet construit à partir de la **Classe-3** est de « type **Classe-3** » mais aussi de « type **Classe-2** » et de « type **Classe-1** ».

- ✓ La relation « **est-un** » traduit une **relation hiérarchique** et donc une compatibilité associée entre les instances issues d'un même arbre.
- ✓ A la définition de la classe dérivée ou classe fille, on utilise le mot-clé **extends** pour désigner la classe-mère. Par défaut, si le mot-clé **extends** est omis, la classe hérite de la classe **Object**, classe au sommet de la hiérarchie **Java**.
- ✓ La classe **Object** est la **superclasse** de toutes les classes : en **Java**, la hiérarchie de classes est une arborescence de racine unique.
- ✓ On peut donc affirmer qu'en **Java**, toute instance de classe, quelle que soit sa classe d'appartenance est de type **Object**, en plus d'être du type de la classe avec laquelle elle a été instanciée.

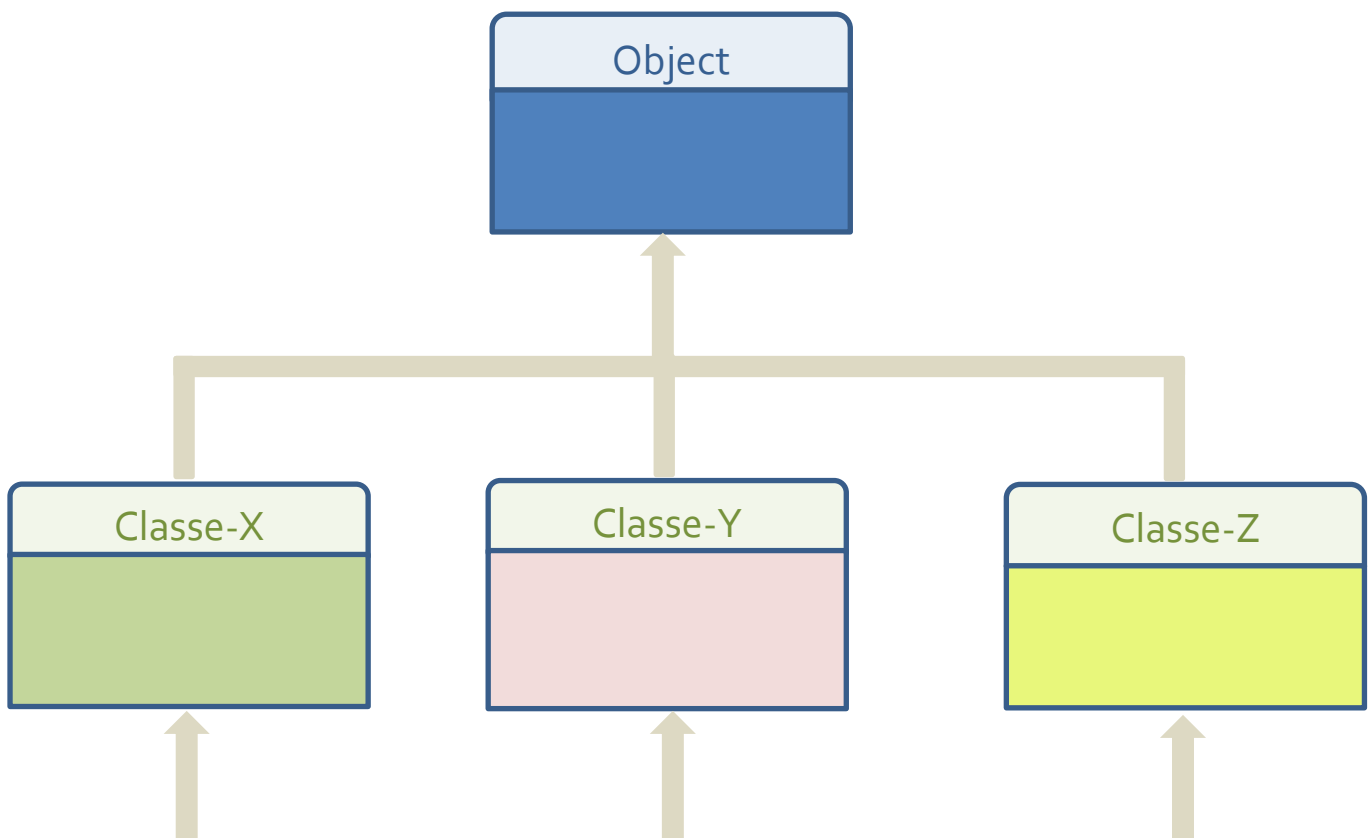


Figure 4 : La hiérarchie **Java** : une arborescence à racine unique.

- ✓ Cette propriété est fondamentale dans la programmation objet en **Java** : là où une instance de type **Object** est attendue, n'importe quelle instance de toute classe peut être fournie.
- ✓ Par exemple : Si dans une méthode, il existe un paramètre de type **Object**, alors n'importe quelle instance pourra satisfaire ce paramètre puisque n'importe quelle instance « est de type » **Object**.

- 6 - Les constructeurs dans un contexte d'héritage

Appel implicite au constructeur de *Personne*

Lorsque l'on exécute la séquence suivante :

```
Salarie durand = new Salarie("Java SARL");  
durand.afficher();
```

l'objet référencé par *durand* se présente comme s'il était une instance de type *Personne* (qu'il est par héritage) car il hérite de la méthode *afficher()*, méthode *public* de *Personne*.

On obtient alors l'affichage suivant :

```
Je m'appelle Nom inconnu
```

Sans l'avoir mentionné dans le constructeur de la classe fille *Salarie*, la variable d'instance *nom* (qui est déclarée dans la classe mère *Personne*) a pris comme valeur "Nom inconnu".

```
public Personne() {  
    nom = "Nom inconnu\n";  
    age = 0;  
}
```

En effet, pour construire l'instance de *Salarie*, le compilateur appelle d'abord le constructeur par défaut de la classe *Personne* (qui existe bien explicitement parmi les deux présents).

✓ Il en est ainsi : lorsque l'on instancie un objet de type classe dérivée, on construit d'abord l'ascendant avant de construire la partie correspondant à la classe de l'objet elle-même.

✓ Chaque constructeur d'un niveau *n* dans une hiérarchie appelle celui immédiatement de la classe mère au-dessus. Ce n'est qu'à la fin de l'exécution du constructeur de niveau *n+1* que l'exécution du constructeur de niveau *n* se poursuit et ainsi de suite.

✓ Ajoutons un affichage de trace de passage dans chaque constructeur comme suit :

```
public Personne() {  
    System.out.println("Constructeur Personne");  
    nom = "Nom inconnu";  
    age = 0;  
}
```

```
public Salarie (String societe) {  
    System.out.println ("Constructeur Salarie");  
    setSociete(societe);  
}
```

✓ Réexécutons la séance suivante :

```
Salarie durand = new Salarie("Java SARL");  
durand.afficher();
```

On obtient l'affichage suivant :

```
Constructeur Personne  
Constructeur Salarie  
Je m'appelle Nom inconnu
```

On distingue bien que le constructeur de la classe de base *Personne* est d'abord appelé – automatiquement par *Java* – avant celui de la classe dérivée *Salarie*.

L'exécution du constructeur de la classe *Personne* a bien valorisé les variables d'instances *nom* et *age* avec les valeurs respectives « Nom inconnu » et 0.

- 7 - Un constructeur efficace pour la classe *Salarie*

Nous l'avons vu : passant par le **constructeur sans argument** de la classe de base *Personne*, la variable d'instance *nom* a pris la valeur « Nom inconnu ».

Comment alors initialiser le nom d'un salarié ? On ne peut pas manipuler directement la variable *nom* dans une méthode de *Salarie*, car elle (*setAge (...)*) est définie **private** dans la classe mère *Personne*.



L'auteur d'une classe dérivée n'a pas plus de privilège que l'instanciateur de la classe de base : il ne peut atteindre les propriétés privées de celle-ci.

En revanche, on peut appeler **explicitement** un constructeur de la classe mère *Personne* dans le constructeur de la classe dérivée *Salarie*, car celui-ci est **public**.

On dispose pour cela en **Java** d'un mot-clé : **super**. Celui-ci s'emploie comme une méthode usuelle. Le nombre et le type des arguments qui interviennent déterminent la signature du constructeur de la classe de base qui sera choisi.

Ici : le constructeur de *Personne* qui attend une *String* et un entier pour le 1^{er} et le 2^{ème} paramètre.

On termine la construction d'un salarié en valorisant la variable d'instance *societe*. On voit que là aussi, on se préoccupe d'abord de la construction de l'ascendant avant de compléter par la partie héritée (ici la v.i. *societe*).

```
public Salarie (String nom, String societe) {  
    super (nom, 0);  
    setSociete(societe);  
}
```

Diagram annotations: A green arrow labeled '1' points from the text 'construction de l'ascendant' to the **super** call. A red arrow labeled '2' points from the text 'partie héritée' to the *setSociete* call.

D'ailleurs, si l'on doit appeler **explicitement** un constructeur de la classe de base dans un constructeur d'une classe dérivée, son appel - grâce à **super** - doit être la **première instruction** dans le code.



Tentez de déplacer l'appel à **super(...)** et vous constaterez que l'éditeur vous indique l'information suivante :

"Call to super must be first statement in constructor"

Instancions maintenant un salarié et appelons la méthode **afficher()**.

```
Salarie dubois = new Salarie ("dubois", "J2EE SA");  
dubois.afficher();
```

Je m'appelle DUBOIS

Cette fois, le constructeur de **Salarie** que nous avons bâti et qui appelle celui de **Personne** permet d'affecter un nom à un salarié. L'affichage précédent l'atteste.



Il serait, naturellement, intéressant d'afficher également le nom de la société ...

C'est l'objet du chapitre suivant.

- 8 - Redéfinition dans une classe dérivée d'une méthode héritée

Si la classe mère *Personne* définit une méthode *m()*, alors une classe fille héritant de *Personne* - comme *Salarie* - peut vouloir légitimement souhaiter un comportement différent pour cette méthode *m()* dont elle hérite :

*La classe **Salarie** redéfinira la méthode *m()*.*

Redéfinissons donc la méthode *afficher()* pour la classe *Salarie*.

```
public class Salarie extends Personne {  
    // .....  
    public void afficher() {  
        // Appelle la méthode afficher() de Personne  
        super.afficher() ;  
        System.out.println(« Je travaille chez » + societe);  
    }  
}
```



La méthode *afficher()* pour la classe *Salarie*



En général, dans le cadre d'une redéfinition d'une méthode *m()* dans une classe dérivée, il est courant d'appeler la méthode *m()* héritée (sauf si l'on veut supprimer le comportement hérité et le remplacer intégralement par le nouveau). Ainsi, le mécanisme de l'héritage facilite la gestion des mises à jour de *m()* de la classe de base.

Ce mécanisme est mis en œuvre par l'utilisation de *super* suivi du nom de la méthode de la classe mère que l'on souhaite atteindre.

✓ L'auteur d'une classe destinée à être dérivée peut vouloir qu'une méthode ne puisse être redéfinie. Il dispose pour cela du modificateur *final*.

✓ *final* empêche donc la redéfinition d'une méthode dans les classes dérivées.

- 9 - Rappel : les références *this* et *super*

Rappelons ici le rôle des mots-clés *this* et *super* qui sont, reprécisons-le, des références sur des instances en *Java*.

- ✓ La référence *this* permet à un objet de se désigner lui-même. On peut utiliser *this* soit *au sein d'un constructeur*, soit *au sein d'une méthode*. Typiquement, les accesseurs en modification ou *mutators* ou encore *setters* utilisent *this* pour qualifier une variable d'instance qui porte le même nom que le paramètre utilisé pour la valoriser.
- ✓ La référence *super* permet à un objet de référencer les variables et les méthodes de sa classe-mère. Cette référence *super* désigne en fait le sous-objet correspondant à la partie héritée.
- ✓ Dans un constructeur, on peut appeler un autre constructeur :
 - ... de la même classe par *this*(...),
 - ... de la classe-mère par *super*(...)
- ✓ Les paramètres intervenant lors de l'appel de *this*(...) ou *super*(...) déterminent lequel de ces constructeurs sera appelé.

- 10 - Contrôler les accès aux **propriétés héritées** (variables d'instance et méthodes)

Accessibilité d'une méthode **private**

On veut maintenant se donner la possibilité de changer l'âge d'un salarié. On définit alors naturellement, pour la classe **Salarie**, la méthode **changerAgeSalarie(...)** qui fait appel à l'accesseur en écriture **setAge** de la classe **Personne** :

```
public void changerAgeSalarie (int age) {  
    setAge(age); // Méthode de la classe Personne  
    // Erreur, car setAge est private dans Personne  
}
```



Le compilateur signale une **erreur de compilation**. En effet, la méthode **setAge** de **Personne** est **private** et donc inaccessible, même pour les sous-classes de **Personne**, en l'occurrence ici : **Salarie**.



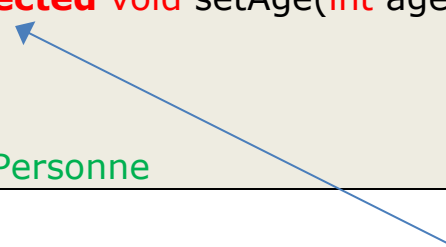
L'auteur d'une classe dérivée n'a pas plus de privilèges concernant les accès des **propriétés privées de la classe de base** que celui qui instancie cette classe de base : il est contraint d'utiliser les accesseurs publics.

Ou bien

- 11 - Le modificateur d'accès *protected*

Modifions la classe de base *Personne* pour lever le problème précédent.

```
public class Personne {    // 2ème version
    protected void setAge(int age) {
        ...
    }
    ...
} // class Personne
```



Nous avons remplacé le mot-clé *private* - qualificateur de *setAge* - par le modificateur *protected*.

- ✓ *protected*, appliqué à une propriété dans une classe de base, permet de conserver une protection semblable à celle assurée par le modificateur *private*, tout en rendant possible l'accès à cette propriété dans les classes *héritées* et les classes du *même package*.

- 12 - Le concept d'héritage et les packages

Rappel : Définition d'un package.

Un **package** est un ensemble de classes, regroupées thématiquement sous un même répertoire, accessible à partir du chemin d'accès aux classes (**classpath**, défini dans l'environnement d'exécution, la **JVM**)

- ✓ Dans les packages, les **classes sont compilées** (fichiers *.class). C'est notamment le cas pour les classes fournies par les distributions de **Java**.

Condition d'accès à une classe définie dans un package

- ✓ Si **monpackage** est un **package Java** dans lequel est définie une classe **ClasseDeMonPackage** alors une classe **MaClasse**, qui ne fait pas partie de **monpackage**, peut utiliser **ClasseDeMonPackage**, à condition que :

1. **ClasseDeMonPackage** soit **publique** dans le **package monpackage** :

```
package monpackage;  
  
public class ClasseDeMonPackage {  
    ...  
}
```

ET QUE

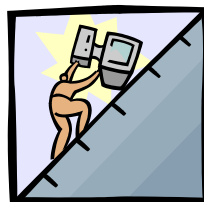
2. **MaClasse** importe la classe **ClasseDeMonPackage** de **monpackage** :

```
import monpackage.ClasseDeMonPackage ;  
public class MaClasse {  
    .....  
    {  
        ClasseDeMonPackage cdmp = new ClasseDeMonPackage() ;  
    }  
}
```

- 13 - Package et modificateurs d'accès en Java

✓ Pour une classe **MaClasse**, appelons « **propriété** » toute **variable d'instance** ou **méthode d'instance de MaClasse**. Etudions pour une propriété de **MaClasse**, les modificateurs d'accès que l'on peut y associer :

1. **private** : accès réservé aux seules méthodes de **MaClasse**. Les utilisateurs de **MaClasse** (ceux qui l'instancient) et les auteurs des classes dérivées de **MaClasse** (ceux qui les définissent) n'ont pas accès à cette propriété **private**.
2. **public** : accès libre à la propriété, pour toute classe qui a accès à MaClasse.
3. **pas de modificateur d'accès** : (on parle d'« accès package ») : la propriété est accessible dans **MaClasse** et aussi pour toutes les autres classes du package où **MaClasse** est définie.
4. **protected** : l'accès à une propriété **protected** est limité aux sous-classes de **MaClasse** et aux classes du même package que MaClasse.



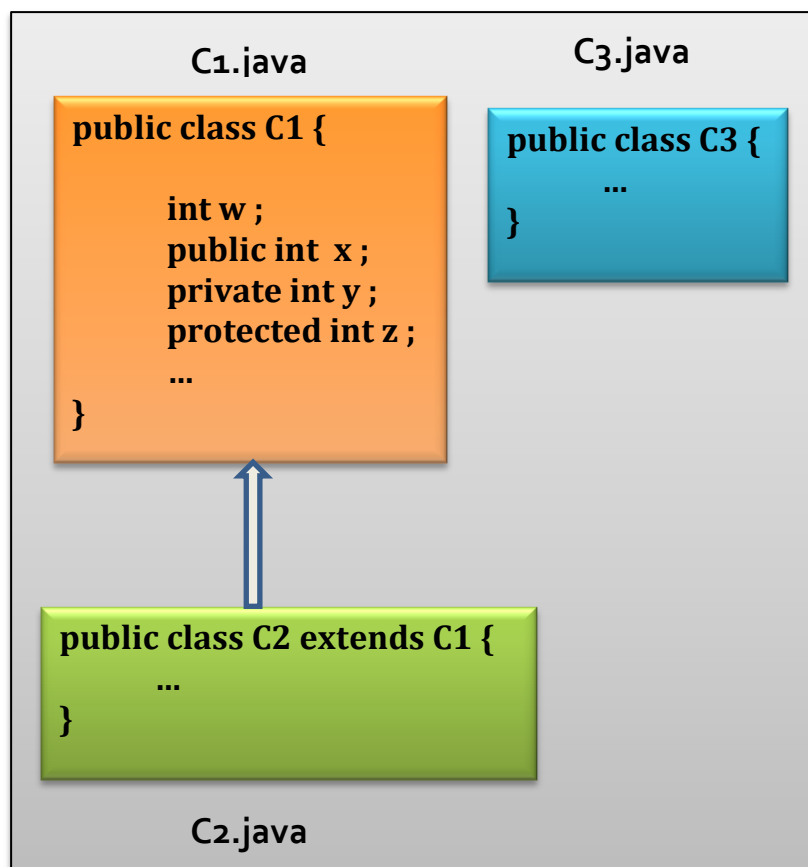
- 14 - Modificateurs d'accès : récapitulatif

Nous allons récapituler les divers modificateurs d'accès et vérifier leur impact respectif dans les contextes d'héritage et de packages.

Dans l'exemple ci-dessous :

- ✓ **C1**, **C2** et **C3** sont 3 classes publiques appartenant à *monpackage*.
- ✓ Dans *monpackage*, la classe **C2** hérite de **C1**.
- ✓ Les classes **C4** et **C5** importent le package *monpackage* mais n'appartiennent pas à ce package.
- ✓ La classe **C4** hérite de **C1**.

monpackage



C4.java

```
import monpackage.*;

public class C4 extends C1 {
    ...
}
```


C5.java

```
import monpackage.*;

public class C5 {
    ...
}
```


Autorisations d'accès aux variables d'instances selon la classe et le modificateur d'accès :

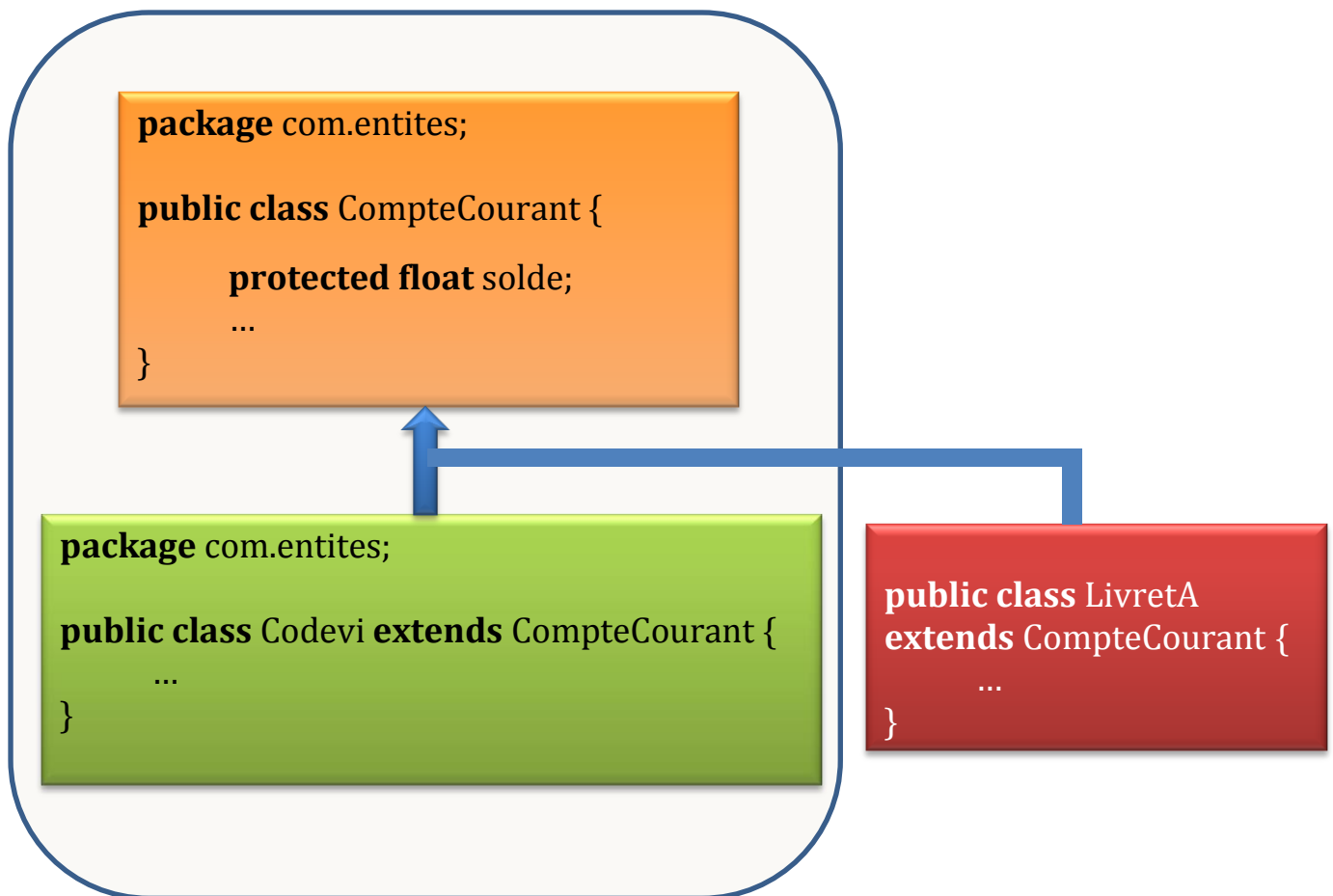
L'examen du schéma des classes ci-dessus nous amène au tableau suivant :

| Accès à  | accès package w | public x | private y | protected z |
|--|-----------------------|-------------|--------------|----------------|
| C1 | oui | oui | oui | oui |
| C2 | oui | oui | non | oui |
| C3 | oui | oui | non | oui |
| C4 | non | oui | non | oui |
| C5 | non | oui | non | non |

- 15 - Accès dans les sous-classes avec *protected*

Si une classe *ClasseFille* hérite de la classe *ClasseMere*, et si *propriete* est une propriété qualifiée *protected* dans *ClasseMere*, toute méthode de *ClasseFille* n'a accès à *propriete* que pour des objets de la classe *ClasseFille*.

Exemple



La classe *CompteCourant* est définie dans le package *com.entites* :

```
package com.entites ;
public class CompteCourant {
    protected float solde;
    public void crediter(float somme) {
        solde += somme;
    }
}
```

La classe **Codevi** hérite de **CompteCourant**. Elle est aussi définie dans le package **com.entites**.

```
package com.entites ;  
public class Codevi extends CompteCourant {  
    ... etc.  
}
```

La classe **LivretA** hérite également de **CompteCourant**. Elle n'est pas définie dans le package **com.entites**.

Elle définit une méthode **tester(...)** qui admet trois paramètres :

- ✓ un objet **compteCourant** de type **CompteCourant**,
- ✓ un objet **livretA** de type **LivretA** et
- ✓ un objet **codevi** de type **Codevi** :

```
import com.entites.*;  
  
public class LivretA extends CompteCourant {  
    public void tester (CompteCourant compteCourant,  
                        LivretA livretA,  
                        Codevi codevi) {  
        solde = livretA.solde;           // OK  
        solde = compteCourant.solde;     // Erreur de compilation  
        solde = codevi.solde;           // Erreur de compilation  
    }  
} // class LivretA
```

- 16 - Choix de conception : *protected* ou *private* ?

Choisir le modificateur d'accès adéquat

Imaginons la situation de programmation objet classique suivante :

- ✓ La classe *MaClasse* doit être implémentée avec les contraintes d'implémentation suivantes :
 - *MaClasse* possède une variable d'instance *vi* de type *TypeVi*, qui doit rester inaccessible aux utilisateurs de la classe – c'est-à-dire aux instanciateurs.
 - *MaClasse* est appelée à être dérivée pour générer des classes filles qui doivent avoir accès à *vi*.
- ✓ Deux possibilités permettent de satisfaire ces contraintes types :

1°) Utilisation du modificateur *protected* dans *MaClasse*

```
public class MaClasse {  
    // vi visible uniquement dans le package de MaClasse et les sous-classes de  
    // MaClasse  
    protected TypeVi vi ;  
    ...  
} // class MaClasse
```

2°) Maintien de private et utilisation d'accesseurs

```
public class MaClasse {  
  
    // vi , private, n'est pas accessible directement hors de MaClasse  
    private TypeVi vi;  
  
    private final TypeVi validerVi(TypeVi valeurVi) {  
        // Contrôle de la validité pour vi  
        // Test de validité : valeurVi est-elle une valeur acceptable pour vi ?  
        // ....  
        return valeurVi;  
    }  
    protected TypeVi getVi() { // Accesseur en consultation  
        return vi;  
    }  
    protected void setVi(TypeVi v) { // Accesseur en modification  
        this.vi = validerVi( v );  
    }  
} // class MaClasse
```

La deuxième solution est **plus contraignante** pour le concepteur de la classe. Mais elle présente l'avantage d'apporter un **vrai contrôle de validité** sur les variables d'instances, et assure ainsi une **plus grande sûreté** dans l'utilisation de la classe.

Cette technique contribue ainsi à **l'intégrité des données** à travers les instances manipulées.

- 17 - Version finale de la classe *Salarie*

```
public class Salarie extends Personne {

    private String societe; // Employeur du salarié

    protected void setSociete(String entreprise) {
        societe = entreprise.toUpperCase (); // La vi societe est stockée en majuscule
    }

    // Deux constructeurs pour Salarie
    public Salarie (String nom, String societe) {
        super(nom , 0); // Appel du constructeur Personne ( String , int )
        setSociete(societe);
    }

    public Salarie (String nom, String societe, int age) {
        super(nom, age); // Appel du constructeur Personne ( String , int )
        setSociete(societe);
    }

    public void afficher() {
        super.afficher(); // Pour appeler la méthode d'affichage de Personne
        System.out.println(" Je travaille chez " + societe);
        System.out.println("-----");
    }

    public static void main (String args [ ]) {
        Salarie salarie ;
        salarie = new Salarie (" Meunier ","Java SARL");
        salarie.afficher() ;
        // salarie.age = 77;
        // Erreur de compilation, age est inaccessible
        salarie.setAge(77);
        salarie.setSociete("JDK SA ") ;
        salarie.afficher() ;
    } // main
} // class Salarie
```



A noter qu'il est tout à fait possible de faire figurer la méthode principale *main()* qui est une méthode classe (*static* donc) au sein de la classe *Salarie*. Et c'est dans cette méthode *main()* qui se font les instantiations.

Les affichages produits par l'exécution précédente sont :

```
run:
Je m'appelle MEUNIER
Je travaille chez JAVA SARL
-----
Je m'appelle MEUNIER
et j'ai 77 ans.
Je travaille chez JDK SA
-----
BUILD SUCCESSFUL (total time: 0 seconds)
```

- 18 - Redéfinir une méthode : récapitulons

Modificateurs d'accès

Il est possible d'élargir l'accès aux méthodes et aux variables d'instance à travers l'héritage :

- ✓ Une méthode **private** n'est pas accessible par les sous-classes et ne peut **donc pas être redéfinie** (au sens objet du terme).
- ✓ Une méthode **protected** peut être redéfinie **protected** ou **public** (élargissement de l'accès).
- ✓ Une méthode **public** ne peut être redéfinie que **public**.
- ✓ Une méthode dite "**package**" (sans modificateur d'accès) ne peut pas être redéfinie **private**.

Redéfinitions multiples

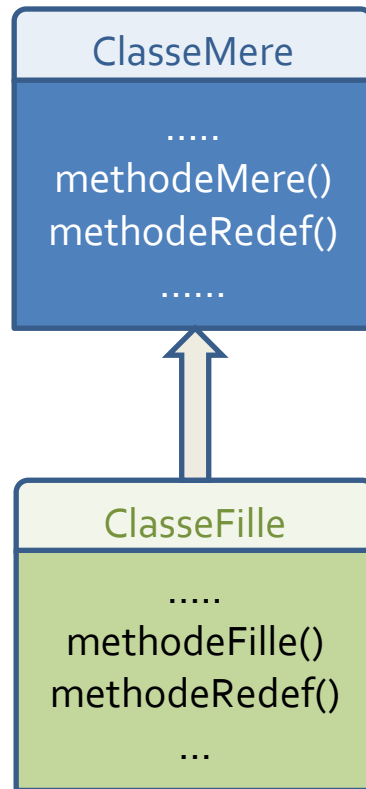
Quand une méthode *maMethode()* est redéfinie plusieurs fois, on ne peut, dans une redéfinition, accéder qu'à la redéfinition immédiatement supérieure (via **super**).

En d'autres termes, il n'est pas possible d'enchaîner les :

super(...).super(...)....

- 19 - Le **cast** induit par l'héritage

Reprenons la hiérarchie suivante, déjà étudiée :



La classe **ClasseFille**, qui hérite de la classe **ClasseMere**, récupère **par héritage** les propriétés de cette dernière tout en ajoutant des propriétés en plus.

Constats :



ClasseMere possède deux méthodes : *methodeMere()* et *methodeRedef()*.

ClasseFille possède également deux méthodes : *methodeFille()* mais aussi *methodeRedef()* qui fait donc l'objet d'une ... redéfinition.



En **Programmation Orientée Objet**, une instance d'une classe fille, appliquant donc la relation « est un » relativement à sa classe mère, peut tout à fait être **considérée comme une instance de la classe mère**.

On peut donc écrire en **Java** :

```
ClasseMere objMere = new ClasseFille() ; // OK
```

Ce qui peut se traduire par : là où une instance de la classe **ClasseMere** est attendue, une instance de la classe **ClasseFille** peut être fournie.

C'est bien ce qui se passe ici : **objMere**, référence sur la classe **ClasseMere** est valorisée avec le résultat d'une instanciation de la classe fille **ClasseFille**.

La compatibilité est **valide**.

En revanche, la **réciproque n'est pas vraie**:

```
ClasseFille objFille = new ClasseMere() ; // Erreur de compilation
```

On dit que :

L'héritage définit un **cast implicite** de **ClasseFille** vers **ClasseMere**

✓ Etudions l'exécution de :

```
ClasseMere objMere = new ClasseFille();  
objMere.methodeMere();  
objMere.methodeRedef(); // Quelle méthode va être appelée ?
```

La question qui vient immédiatement à l'esprit est la suivante :

Lors de l'appel à `objMere.methodeRedef()` : quelle méthode va être sollicitée ? En effet, `methodeRedef()` est redéfinie dans `ClasseFille` et existe donc en 2 exemplaires.

`objMere` étant de type `ClasseMere`, on peut légitimement penser que c'est `methodeRedef()` de `ClasseMere` qui va être appelée.

Mais on peut aussi avoir le raisonnement suivant : `objMere` est certes de type `ClasseMere` mais, en tant que référence, est en train de désigner une instance de `ClasseFille` qui possède elle aussi `methodeRedef()`.

Alors ?

Alors, `Java` considère toujours qu'il doit tenir compte de l'objet désigné par la référence : en conséquence, c'est bien la méthode de `ClasseFille` qui va être appelée. Ce comportement est logique : il est raisonnable que le traitement que l'on souhaite voir s'exécuter soit bien celui de l'objet désigné.



Pour les habitués de C++ : en `Java`, les fonctions virtuelles sont automatiquement implémentées.

Mais :


```
ClasseMere objMere = new ClasseFille() ;  
objMere.methodeFille() ; // Erreur de compilation  
( ( ClasseFille ) objMere ).methodeFille() ; // OK, le cast « rassure » le  
// compilateur
```

- Les seules méthodes que l'on peut appeler directement à partir d'une référence en **Java** sont les méthodes de la classe du type de cette référence.

Donc ici, les seules méthodes appelables directement sont celles de **ClasseMere**. Parmi ces méthodes, certaines peuvent avoir été redéfinies, auquel cas, ce sont **les méthodes redéfinies qui seront appelées** si l'on pointe sur une instance de la classe dérivée.

Cette propriété est essentielle en P.O.O. . On peut même affirmer qu'elle est déterminante.



 TP à réaliser : Nous vous invitons à écrire un projet **Java SE** mettant en évidence ce principe absolument capital pour la suite de votre parcours.

- 20 - Les classes abstraites

Trois classes pour modéliser un parc de véhicules de location.

Etude de cas préalable :

- ✓ Une société de location « **Loc-auto** » souhaite informatiser son parc de véhicules selon les spécifications suivantes :
- ✓ **Loc-auto** propose trois types de véhicules à la location : **Citadine**, **Familiale** et **Utilitaire**.



- ✓ Chaque **citadine** est caractérisée par une **marque**, un **nom**, une **autonomie** et un **identifiant** (qui sera son immatriculation).



- ✓ Chaque **familiale** est caractérisée par une **marque**, un **nom**, un **identifiant**, un **nombre de passagers maxi**.



- ✓ Chaque **utilitaire** est caractérisé par une **marque**, un **nom**, un **identifiant** et un **poids total en charge**.

- ✓ Chaque véhicule doit faire l'objet d'une **révision périodique** à un kilométrage et une fréquence **spécifiques selon sa catégorie**.

Problème :

- ✓ On souhaite construire un **modèle de classes** représentant chaque type de véhicule. La consigne pour une bonne modélisation étant de trouver les **propriétés communes** entre classes et celles qui caractérisent, au contraire, spécifiquement chacune d'elles.
- ✓ Voyons comment gérer les **points communs**, mais aussi les **différences** entre ces 3 types de véhicules.

Quelles sont les propriétés communes ?

- ✓ A la lecture du cahier des charges, on distingue 3 propriétés communes à chaque type de véhicule : *marque*, *nom* et *identifiant*.
- ✓ On retrouvera donc ces propriétés en tant que variables d'instances pour chaque objet créé, issu des trois classes.
- ✓ La classe *Citadine* possède une propriété supplémentaire : l'*autonomie*.
- ✓ La classe *Familiale* possède une propriété supplémentaire : le *nombre de passagers maxi*.
- ✓ La classe *Utilitaire* possède une propriété supplémentaire : le *poids total en charge*.
- ✓ Concernant les méthodes maintenant :
 - on va doter chaque classe d'une méthode de description *toString()* qui va varier d'un véhicule à l'autre mais qui s'appuiera sur une partie commune aux 3 classes.
 - chaque classe possédera sa propre méthode *planifierRevision()* : les 3 méthodes porteront le même nom dans chaque classe mais auront *une forme* et un *comportement distincts* les unes des autres.



L'idée généraliste de la modélisation est de mettre en facteur et de remonter **les informations communes** le plus haut possible dans une hiérarchie.

C'est ainsi que l'on va créer une classe : la classe *Vehicule* dans laquelle on va retrouver toutes les variables d'instances communes aux 3 types de véhicules à louer. Cette classe sert à factoriser du contenu (les **variables d'instances**) et la façon de les valoriser : le constructeur. Une méthode de description initiale sera aussi présente dans cette classe.

Mais cette classe **ne peut donner lieu à une instantiation** : on peut louer **concrètement** une citadine, une familiale, ou encore un utilitaire mais pas un véhicule : ce concept de véhicule est abstrait et pas **assez précis pour qu'il soit éligible à un phénomène d'instanciation**.

La classe *Vehicule* est **abstraite**

✓ Construisons cette classe particulière *Vehicule* :

```
public abstract class Vehicule {  
  
    protected String marque;  
    protected String nom;  
    protected String identifiant;  
  
    public Vehicule (String marque, String nom, String identifiant) {  
        this.marque = marque;  
        this.nom = nom;  
        this.identifiant = identifiant ;  
    }  
    public String toString() {  
        return marque + "-" + nom + "-" + identifiant;  
    }  
    public abstract void planifierRevision();  
} // class Vehicule
```


Afin de bien appréhender ce concept et ceux qui vont suivre, veuillez construire, au fur et à mesure des explications, le TP suivant que vous allez enrichir, petit à petit, pour aboutir à une solution complète et parfaitement maîtrisée.



TP d'application : Construisez au sein d'un projet Java SE le projet *LocAuto* pour reproduire l'étude de cas actuelle qui traite des **classes abstraites** et qui sera bientôt complétée par la notion d'interface en Java.

✓ Réalisez l'étape 1 du TP *LocAuto*, dans le support « *Exercices Java – Série 1* »

Remarques diverses et variées résultant de la lecture du code de la classe abstraite *Vehicule* :

1. La classe est qualifiée d'**abstract**. Conséquence immédiate : elle n'est pas instanciable. C'est-à-dire qu'on ne peut pas exécuter la séquence suivante :

```
Vehicule vehicule = new Vehicule(.....) ;
```

En clair, on ne peut pas instancier un « *Vehicule* » au sens classe du terme.

2. Les 3 variables d'instance *marque*, *nom* et *identifiant* se retrouvent bien factorisées dans cette **classe abstraite**.
3. La méthode de description *toString()* permet d'afficher ces 3 informations communes aux 3 types de véhicules.
4. La classe *Vehicule* est dotée d'un constructeur qui prend 3 arguments, précisément ceux dont on a besoin pour valoriser ces **v.i.**
5. Question taraudante : comment se fait-il qu'une classe qui n'est pas instanciable soit munie d'un constructeur ? Hum ? Bizarre ...

Tentez d'y réfléchir et d'y apporter une réponse.

6. Le dernier point remarquable concerne :

```
public abstract void planifierRevision() ;
```

On y voit une méthode *planifierRevision()* qui n'est que déclarée : elle ne possède pas de paire { ... } mais uniquement un point-virgule ;

De plus, elle est elle aussi qualifiée d'**abstract**. Que cela signifie-t-il ?

Réponse : cela implique que toute sous-classe qui va hériter de cette classe abstraite *Vehicule* va devoir de façon **contraignante et obligatoire** définir (c'est-à-dire fournir une **définition** avec { ... }) pour cette méthode .

Si elle ne le fait pas, elle ne pourra être instanciée à son tour et obligera son concepteur à la qualifier aussi d'**abstract** pour ne pas s'arrêter au stade de la compilation.

Hériter d'une classe abstraite

Rappels :

- ✓ La classe *Vehicule* n'est pas instanciable.
- ✓ Elle **déclare** la méthode **abstraite** *planifierRevision()* : toute sous-classe de *Vehicule* devra définir cette méthode pour être instanciable.
- ✓ Définissons maintenant les classes concrètes *Citadine*, *Familiale* et *Utilitaire*.

```
public class Citadine extends Vehicule {  
  
    private int autonomie ;  
  
    public Citadine ((String marque, String nom, String identifiant, int  
        autonomie) {  
        // Sollicitation du constructeur de la classe abstraite  
        // Vehicule dont le rôle se limite à valoriser les v.i.  
        super(marque, nom, identifiant) ;  
  
        this.autonomie = autonomie;  
    }  
    public String toString() {  
        return super.toString() + "-"+ "- Mon autonomie est de " +  
            autonomie + " kms";  
    }  
    public void planifierRevision() {  
        // Définition obligatoire de la méthode abstraite  
        System.out.println("Planifier révision pour Citadine");  
    }  
} // class Citadine
```

Commentaires sur la classe *Citadine* :

- ✓ Son constructeur fait appel au constructeur de la classe abstraite *Vehicule* afin de lui confier la valorisation des *v.i.* et de les centraliser en un seul endroit.
On voit donc qu'un **constructeur d'une classe abstraite** peut être sollicité **comme n'importe quelle méthode** même si cette classe n'est pas instanciable !

- ✓ L'instruction :

```
super(marque, nom, identifiant) ;
```

- ✓ ... est bien la 1ere instruction du constructeur de la classe *Citadine*. Une fois exécuté, on termine en valorisant la *v.i. autonomie*, spécifique à *Citadine*.
- ✓ La méthode de description *toString()* s'appuie sur *toString()* (*super.toString()*) de *Vehicule* puis la complète pour afficher l'autonomie.
- ✓ Nous avons eu l'**obligation** de **respecter l'imposition** exprimée dans la classe abstraite *Vehicule* : celle qui consiste à définir *planifierRevision()* qui est déclarée **abstract** dans cette classe.



Tentez dans le TP précédemment débuté de commenter la définition de cette méthode pour constater le phénomène.



- ✓ Réalisez l'étape 2 du TP *LocAuto*, dans le support « *Exercices Java – Série 1* »

Rappel : lorsqu'une référence sur une instance *ref* est fournie dans un ordre d'affichage `System.out.println(ref)`, Java déclenche **automatiquement** la méthode `toString()`, spécifique au type de l'objet si elle a été redéfinie ; ce qui est le cas pour *Citadine* .



✓ Dans le TP « *LocAuto* », réalisez les étapes 3 puis 4.

- 21 - Précisions sur les classes abstraites

Une classe abstraite est définie grâce au modificateur **abstract**. Elle peut contenir deux types de méthodes :

- ✓ des méthodes abstraites : elles sont déclarées avec le modificateur **abstract** mais ne sont pas définies : elles jouent le rôle d'un **outil de spécification**, en forçant toute sous-classe instanciable à **implémenter le comportement correspondant**.
- ✓ des méthodes non abstraites, complètement définies, qui représentent soit des comportements par défaut pour l'héritage, soit un comportement commun hérité.

Classes abstraites et méthodes abstraites

- ✓ Une classe est **abstraite** si elle est définie avec le modificateur **abstract**. Elle peut déclarer **0 (zéro)** ou **plusieurs méthodes abstraites**.
- ✓ Une méthode **abstraite** est déclarée (et non définie) avec le modificateur **abstract**.
- ✓ Une classe **MaClasse** qui hérite d'une classe abstraite **ClasseAbstraite** doit implémenter toutes les méthodes déclarées **abstract**, sauf si elle est elle-même **abstraite**.
- ✓ Si **MaClasse** n'est pas définie **abstract**, et si toutes les méthodes abstraites héritées de **ClasseAbstraite** ne sont pas définies dans **MaClasse**, la compilation de **MaClasse** provoque une erreur.

- 22 - Enregistrer les instances issues de la hiérarchie *Vehicule* dans une base de données.

Poursuite de l'étude de cas :

Nous souhaitons pouvoir enregistrer chaque véhicule proposé à la location dans une base de données. On va créer pour cela une classe *ParcVehicules* au sein de laquelle une méthode nommée *enregistrer* va permettre de déclencher des *ordres SQL* visant à stocker chaque type de véhicule et ses variables d'instances associées.

La classe *ParcVehicules*

- ✓ Cette classe sert à enregistrer dans une base de données les objets concrets représentant des véhicules. Elle définit donc une méthode *enregistrer* qui prend en argument l'objet nommé logiquement *vehicule* à enregistrer et dont le code pourrait correspond à :

```
public static void enregistrer (... vehicule) {  
    stockerDansLaBase (vehicule.getIdentifiant() + vehicule.getInfosCompletes());  
}
```

- ✓ La méthode *enregistrer* reçoit donc une référence nommée *vehicule*. Nous allons débattre dans quelques instants concernant le type auquel elle sera rattachée. Elle appelle une méthode nommée *stockerDansLaBase* à qui on transmet le résultat de l'appel de 2 méthodes appliquées sur *vehicule* : *getIdentifiant()* et *getInfosCompletes()*.
- ✓ Ces méthodes, avec un peu d'imagination, renvoient pour la première l'identifiant du véhicule et pour la seconde, l'ensemble de toutes les *v.i.* qui le caractérisent.

- ✓ Réfléchissons sur le type d'appartenance du paramètre *vehicule*.
- ✓ La référence *vehicule*, si elle représente un véhicule, doit répondre aux messages *getIdentifiant()* et *getInfosCompletes()*.
- ✓ On peut envisager de définir ces méthodes spécifiquement dans les classes *Citadine* et *Familiale* et *Utilitaire* en imposant leur présence par :

```
public abstract String getIdentifiant()
```

et

```
public abstract String getInfosCompletes()
```

dans *Vehicule* comme nous l'avons fait avec *planifierRevision()* et définir le type de *vehicule* dans *enregistrer* comme étant *Vehicule*.

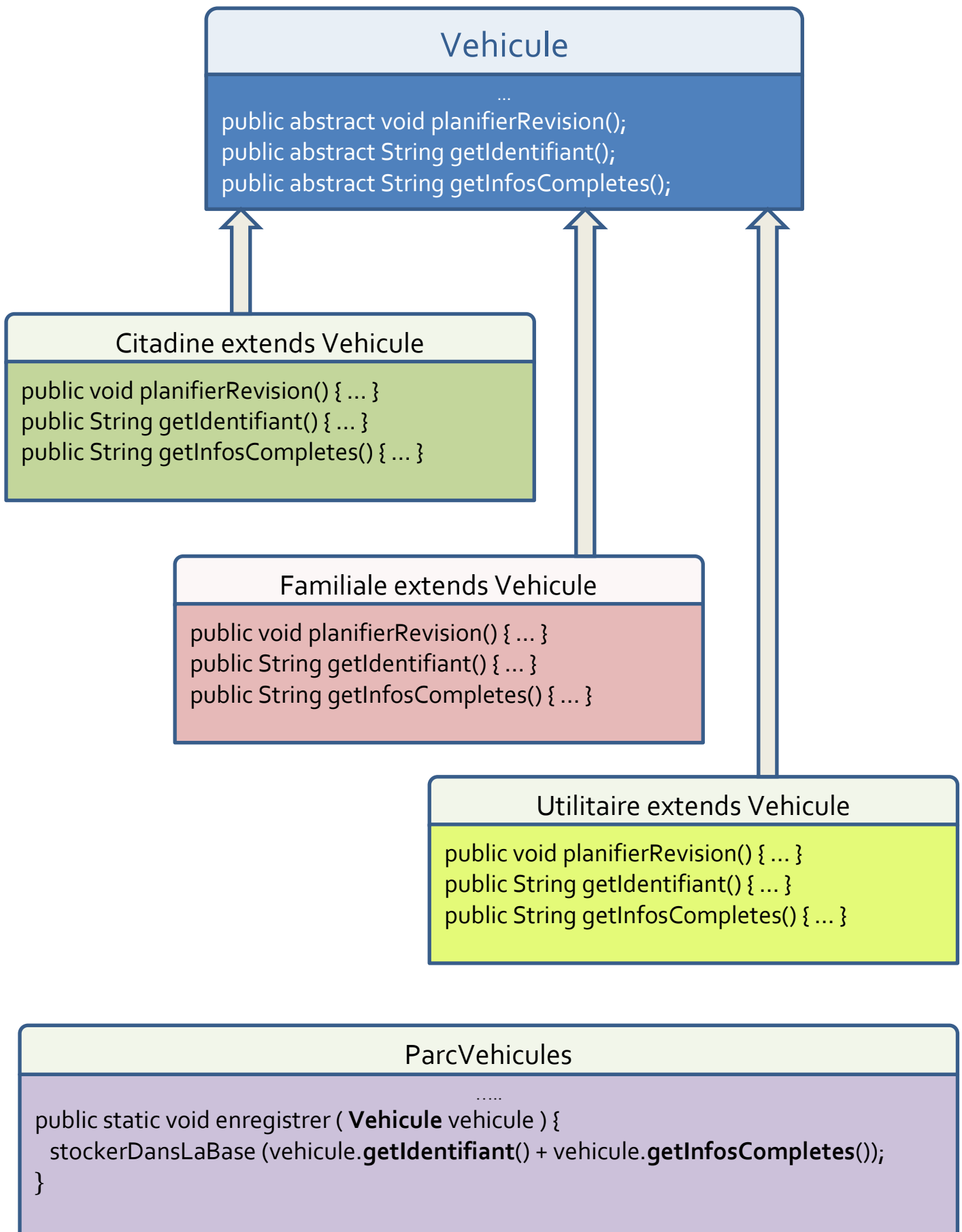
```
public static void enregistrer (Vehicule vehicule) {  
    stockerDansLaBase (vehicule.getIdentifiant() + vehicule.getInfosCompletes());  
}
```

```
public class Vehicule {  
    ....  
    ....  
    public abstract void planifierRevision();  
    // Penser à ajouter ces 2 méthodes abstract pour obliger à les définir ...  
    public abstract String getIdentifiant();  
    public abstract String getInfosCompletes();  
}
```

Cette configuration fonctionne parfaitement : *vehicule* peut recevoir n'importe quelle instance issue d'une classe concrète qui hérite de la classe abstraite *Vehicule*.

On est certain que ces instances répondent parfaitement aux méthodes *getIdentifiant()* et *getInfosCompletes()* puisque la présence de **abstract** dans *Vehicule* a obligé de les définir dans les sous-classes.

Hiérarchie de l'étude précédente



✓ Poursuivons notre étude :

Le loueur de véhicules souhaite maintenant que les garages où sont stockés les véhicules **soient également répertoriés** dans la base de données pour gérer ses locaux professionnels.

Ces garages seront représentés par une classe **Garage** dotée des **v.i. surface, nbr_vehicule_stockables, nbr_etages**. Les instances de cette classe **Garage** ne peuvent pas être transmises à la méthode

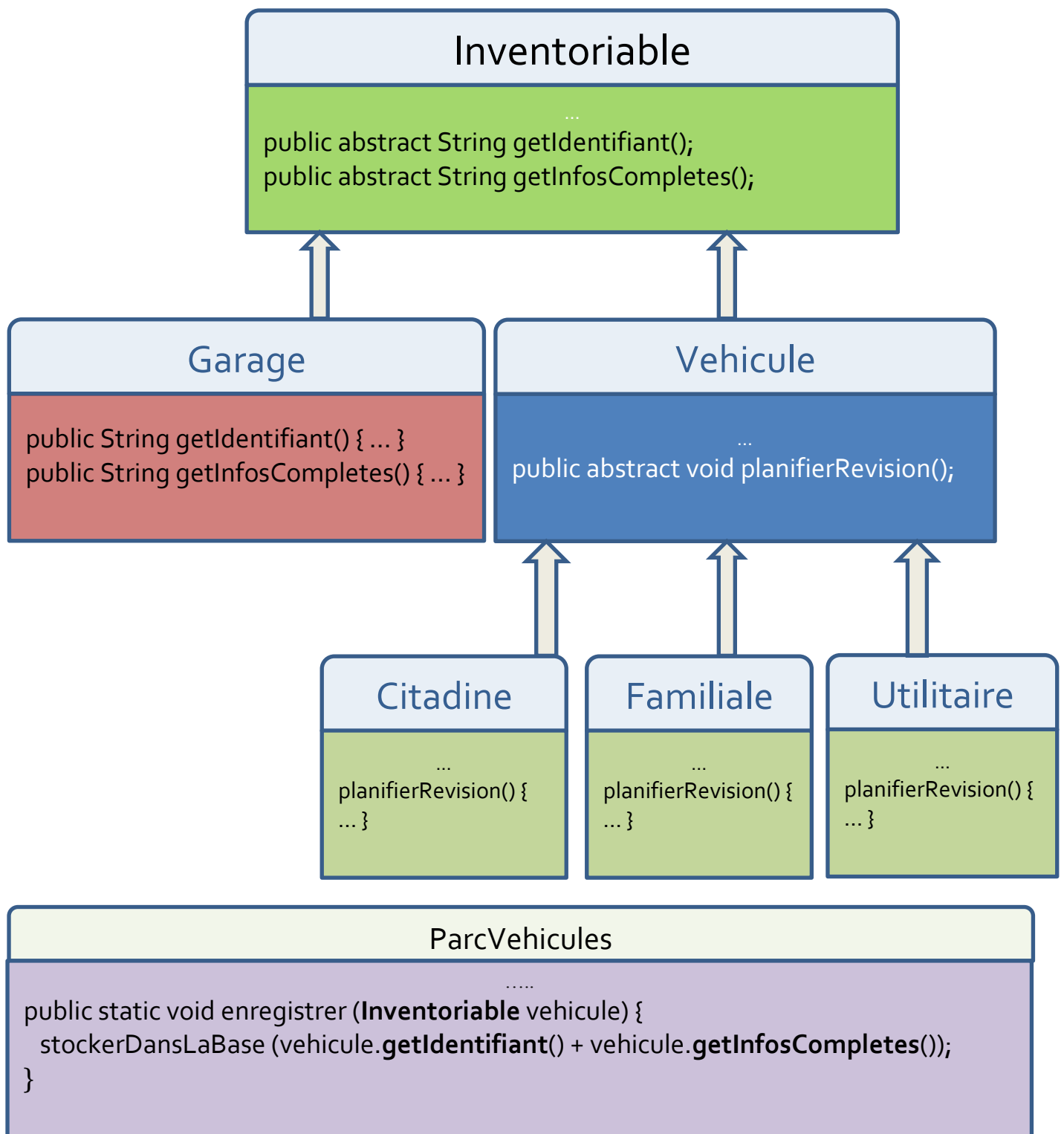
enregistrer(Vehicule vehicule) car on ne peut pas raisonnablement hériter de la classe **Vehicule** pour générer la classe **Garage** !! En d'autres termes : un garage n'est pas un véhicule. Or, la relation d'héritage traduit bien la notion « **est-un** ».

Ce serait un défaut conceptuel de faire hériter la classe **Garage** de **Vehicule**.

On va donc générer une classe d'un niveau supérieur nommée **Inventoriable** dont vont hériter les classes **Vehicule** et **Garage**.

Un garage et un véhicule (et ses classes dérivées bien sûr) sont bien, par héritage, des « **Inventoriable** » au sens objet.

Ce qui donne :



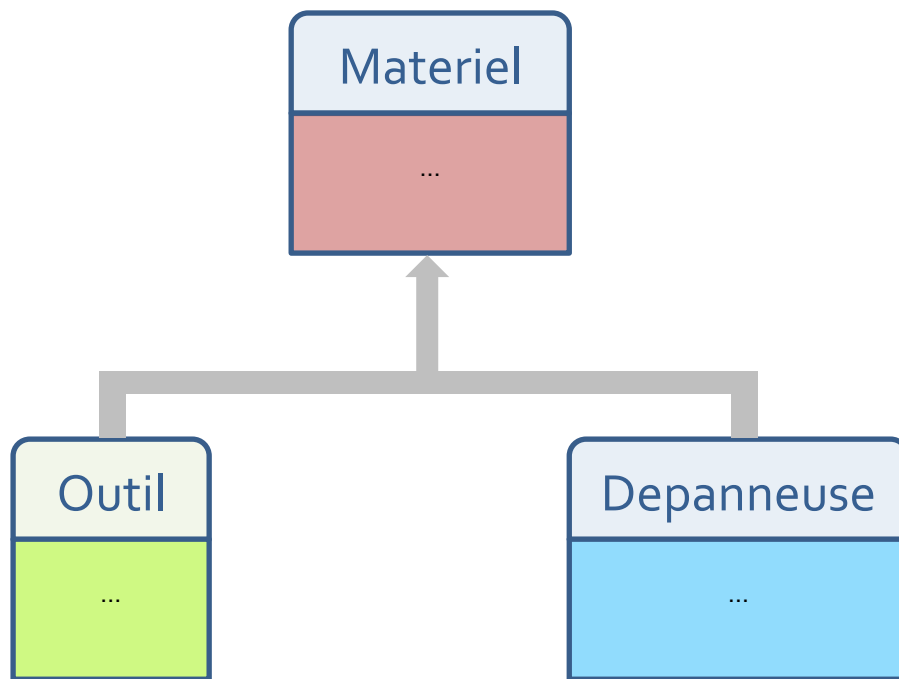
- 23 – Le code de la hiérarchie de classes

```
public abstract class Inventoriable {  
    public abstract String getIdentifiant();  
    public abstract String getInfosCompletes();  
}  
  
abstract class Vehicule extends Inventoriable {  
    // ...  
}  
  
class Garage extends Inventoriable {  
    // ...  
}
```

Les classes **Vehicule** et **Garage** héritant maintenant de **Inventoriable**, on peut changer le type (et le nom) du paramètre de la méthode **enregistrer** de **ParcVehicules**.

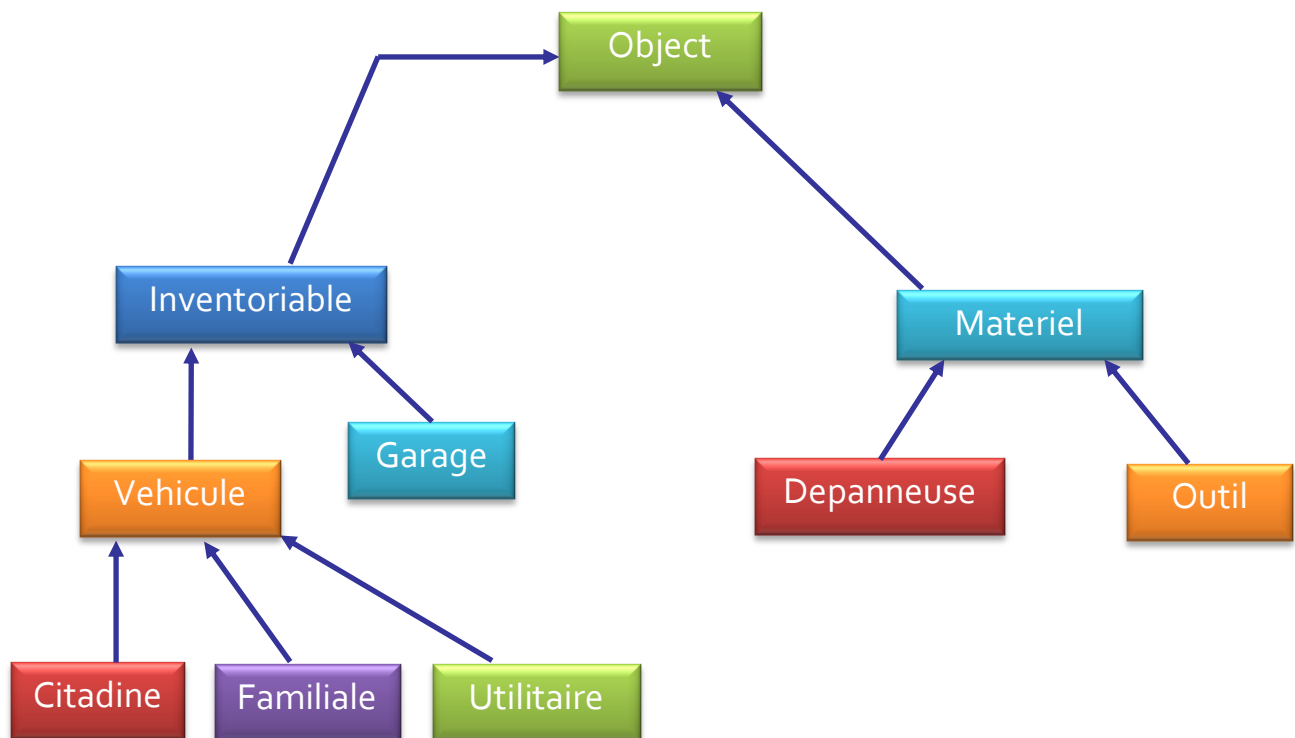
```
public void enregistrer (Inventoriable inventoriable) {  
    ..  
}
```

✓ On veut aussi **répertorier dans l'inventaire** les dépanneuses servant à rapatrier les différents véhicules loués tombés en panne lors de leur utilisation par les clients. Les dépanneuses appartiennent à une hiérarchie déjà constituée selon le schéma suivant :



Materiel est la classe de base de ***Outil*** (ceux liés à la mécanique qui permettent la révision des véhicules) et de ***Depanneuse***.

On a, finalement, la hiérarchie suivante :



La hiérarchie complète de notre étude

Selon la hiérarchie précédente, si l'on veut pouvoir enregistrer les dépanneuses, il suffit de faire hériter la classe **Depanneuse** de la classe **Inventoriable**.

Les instances de **Depanneuse** seraient ainsi des « **Inventoriable** » et toute instance de **Depanneuse** pourraient convenir pour la méthode **enregistrer** qui attend, rappelons-le, un paramètre de type **Inventoriable**. Mais :

Depanneuse hérite déjà de la classe **Materiel** : elle ne peut, de fait, hériter en plus de **Inventoriable** : **Java** interdit tout héritage multiple comme cela a été indiqué en début de ce cours.

Ceci met en évidence un nouveau concept en **Java** : on va faire de **Inventoriable** non plus une classe **abstract** mais une **interface** dont on va implémenter les méthodes.

- 24 - L'interface *Inventoriable*

Définir une interface

On définit les comportements liés à l'enregistrement dans l'interface *Inventoriable*.

On transforme la classe abstraite *Inventoriable* en **interface**.

✓ Une classe ne peut hériter de plusieurs classes mais elle peut implémenter une, voire plusieurs interfaces en complément de l'héritage.

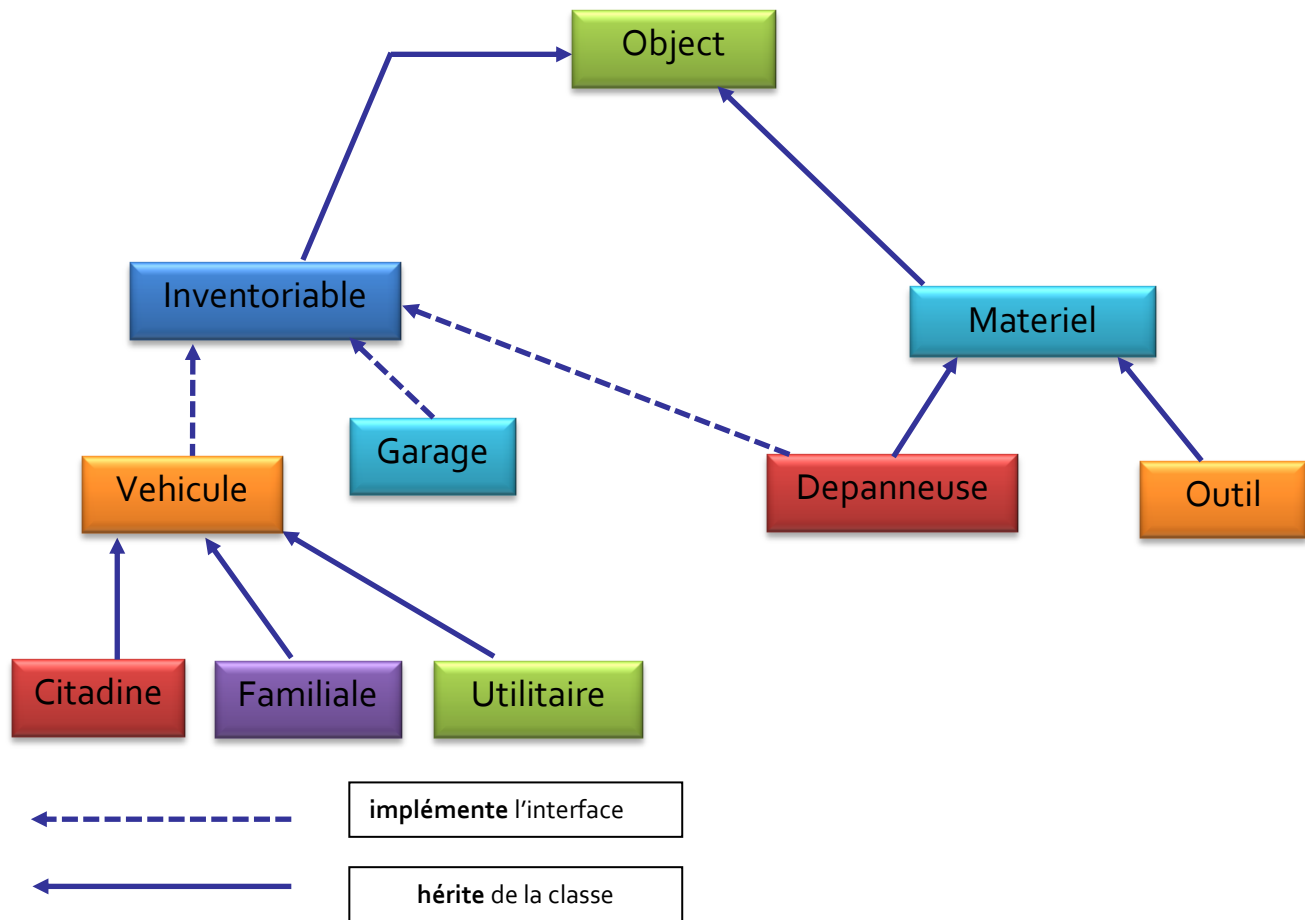
✓ Une interface est L'outil de spécification par excellence en Java.

```
public interface Inventoriable {  
    public String getIdentifiant();  
    public String getInfosCompletes();  
} // interface Inventoriable  
  
class Depanneuse extends Materiel implements Inventoriable {  
    // .... etc  
    public String getIdentifiant() { return identifiant ; }  
    public String getInfosCompletes() { ... }  
} // class Depanneuse
```

✓ Une **interface** est un outil de spécification dans lequel les méthodes sont implicitement **abstract**. Toute classe qui se déclare candidate à implémenter une **interface** s'engage à respecter le contrat qui consiste à donner un sens aux méthodes et donc les définir.

✓ Toute classe ne peut hériter que d'une seule classe mais peut implémenter autant d'interfaces qu'elle le souhaite.

Une nouvelle hiérarchie



- 25 - Héritage, classes abstraites et interfaces

Les interfaces

✓ Une **interface** est une classe abstraite particulière. Elle ne définit aucune variable d'instance et toutes ses méthodes doivent être abstraites. Une interface ne fait que représenter, sans les implémenter, **un ensemble de comportements**.

Elle a uniquement un pouvoir de spécification.

On définit une **interface** en utilisant le mot clé **interface**.

Implémentation d'une interface

Une classe peut implémenter une ou plusieurs interfaces.

```
class Airbus implements Aeronef, Materiel { ... }
```

Une **interface** peut hériter de **plusieurs interfaces**. Elle rassemble alors les **spécifications des interfaces mères** :

```
interface Avion extends Aeronef, Materiel
```

On peut alors définir la classe **AirBus** comme suit :

```
class Airbus implements Avion
```

Interfaces ou classes abstraites ?

- ✓ En **Java**, il n'y a pas d'héritage multiple : une classe fille n'hérite que d'une classe mère (**abstraite** ou non).
- ✓ Une extension à l'héritage simple tel qu'il est pratiqué en Smalltalk est fournie par les interfaces :
- ✓ Une classe peut hériter d'une seule classe-mère et d'une ou plusieurs **interfaces**, dont elle **fournit une implémentation**.

Utilisation des interfaces

- ✓ Les **interfaces** n'ont pas seulement un rôle de spécification, elles permettent d'avoir une « vue » particulière sur un objet : un objet peut être casté dans le type d'une **interface** qu'il implémente.
- ✓ Plusieurs objets implémentant la même interface pourront être considérés de façon équivalente à travers cette interface.

```
public interface Aeronef {  
    public String vitesseDeCroisiere();  
}
```

```
class Airbus implements Aeronef, Materiel {  
    public String vitesseDeCroisiere () {  
        // etc...  
        return .... ;  
    }  
}
```

```
class Ariane implements Aeronef, Fusee {  
    public String vitesseDeCroisiere () {  
        // etc...  
        return " ....";  
    }  
    ....  
}
```

On pourra alors passer indifféremment un objet **AirBus** ou un objet **Ariane** comme paramètre dans la méthode suivante :

```
public void afficheEnginAeronef(Aeronef aeronef) {  
    System.out.println("Type : " + aeronef.getClass().getName());  
    System.out.println("Vitesse de croisière : " + aeronef.vitesseDeCroisiere());  
}
```



L'expression appliquée à une référence `getClass().getName()` renvoie le nom du type pleinement qualifié de l'instance désignée.

Copyright

➤ **Chef de projet (responsable du produit de formation)**

PERRACHON Chantal, DIIP Neuilly-sur-Marne

➤ **Ont participé à la conception**

COULARD Michel, CFPA Evry Ris Orangis

➤ **Réalisation technique**

COULARD Michel, CFPA Evry Ris Orangis

➤ **Crédit photographique/illustration**

Sans objet

➤ **Reproduction interdite / Edition 2013**

AFPA Février 2014

Association nationale pour la Formation Professionnelle des Adultes

13 place du Général de Gaulle – 93108 Montreuil Cedex

www.afpa.fr