



Objets et classes

Principes de l'encapsulation



SOMMAIRE

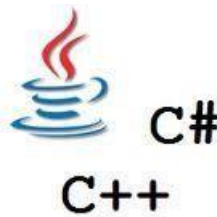
✓	1 - Présentation des quatre étapes de la P.O.O.	3
✓	2 - Présentation du type abstrait <i>Personne</i>	4
✓	3 - Le type <i>Personne</i> en Java	7
✓	4 - La programmation orientée objet	9
✓	5 - La classe : mécanisme du support de l' encapsulation	10
✓	6 - Version 1 de la classe <i>Personne</i>	11
✓	7 - Type d'accès : public ou privé	15
✓	8 - Contrôle des instanciations	17
✓	9 - Version 2 de la classe <i>Personne</i>	19
✓	10 - Récapitulons	20
✓	11 - Types primitifs et types objets	22
✓	12 - Variables et valeurs	23
✓	13 - Variables, valeurs et affectations	25
✓	14 - Pile d'exécution, tas d'allocation	26
✓	15 - Définition des accesseurs	27
✓	16 - Une classe <i>Personne</i> robuste	28
✓	17 - Un ou plusieurs constructeurs	29
✓	18 - Modificateurs static et final	30
✓	19 - Variables et méthodes de classe	31

✓	20 - Récapitulations	33
✓	21 - Version finale de la classe <i>Personne</i>	34
✓	22 - Ecriture et exécution d'un programme <i>Java</i>	36
✓	23 - Une classe usuelle : la classe <i>String</i>	37
✓	24 - Importations	39
✓	25 - TP d'application	41

- 1 - Présentation des quatre étapes de la P.O.O.

✓ Les quatre étapes essentielles apparaissant dans l'activité de la programmation orientée objet sont les suivantes :

1. **Découper l'univers** du problème à implémenter en catégories d'objets : les classes.
2. **Déterminer** les états que peuvent prendre les objets de ces classes.
3. **Identifier** les messages qu'ils pourront recevoir (sous quelles formes)
4. **Définir** la manière et la façon dont ils y réagiront.



- 2 - Présentation du type abstrait *Personne*

Débutons notre apprentissage de la **P.O.O.** en établissant le cahier des charges du type abstrait *Personne*, représentant le concept d'un salarié d'une entreprise.

Spécifions ce cahier des charges simplifié par la description minimale suivante :

1. Une *personne* est décrite par deux informations qui la caractérisent :
 - son **nom**.
 - le **nom de l'entreprise** qui l'emploie.
2. Une *personne* doit être capable de préciser qui elle est, en affichant ses informations, lorsqu'on le lui demande.

On distingue déjà, à la simple lecture de ce cahier des charges, deux types d'éléments liés à une personne :

- ✓ Des caractéristiques physiques (les données) . C'est-à-dire la valeur de ses attributs.
- ✓ Des caractéristiques comportementales. (les actions). C'est-à-dire les ordres auxquels une personne peut répondre lors de sollicitations extérieures. Ici, il s'agit pour une personne, de la capacité à s'afficher.

Comment représenter en **P.O.O.** ce type abstrait *Personne* et ses **deux types de composantes** dans les langages de programmation ?

Nous allons le faire.

1. **D'abord** avec un **langage non objet** : le **C**. Nous y verrons les problèmes associés.
2. **Puis** avec un langage objet : **Java**. Nous y verrons alors comment l'objet peut résoudre les problèmes identifiés lors de l'étape précédente.

Programmation procédurale en langage C

```
# include <stdio.h>
# include <string.h>

struct Personne { // Une personne est décrite par son nom et celui de sa société.
    char nom [30];
    char societe [30];
} ;

// Ce traitement permet d'afficher la description d'une personne
void afficher(struct Personne personne) {


    printf(« Je m'appelle %s\n», personne.nom);
    printf(« Je travaille chez %s\n», personne.societe);
}

void main() {

    // On déclare la variable martin de type struct Personne qui décrit une
    // personne
    struct Personne martin;

    // On initialise cette variable avec ses 2 champs respectifs : nom et societe
    strcpy (martin.nom, "MARTIN");
    strcpy (martin.societe, "JAVA SARL");

    // On affiche la description de martin
    afficher (martin) ;
}
```



Le résultat produit l'affichage suivant :

```
Je m'appelle MARTIN
Je travaille chez JAVA SARL
```

La fonction *afficher* a bien produit les informations de *martin*.

Trois problèmes potentiels liés à l'exemple précédent

1°) Il y a **séparation** - dans 2 entités distinctes - les données (**struct** **Personne**) et les traitements (**afficher**).

Or, un employé est aussi bien caractérisé par ses **données** (**nom** et **societe**) que par ses **comportements** (**afficher**).

2°) Les **contrôles d'intégrité** sont difficiles à établir. (Par exemple : possibilité de déclarer une personne sans l'initialiser, puis utilisation de **afficher** (...)).

3°) - **Pas de séparation** entre l'interface et l'implémentation du type **Personne**.

- **Pas de restriction d'accès** concernant les valeurs.

- **Pas de possibilité d'établir des contraintes** comme :

« Un employé doit toujours avoir un nom qui ne peut être modifié une fois valorisé ».

Or, parmi les nombreux avantages apportés par la **programmation orientée objet**, on doit au minimum veiller à :

- ✓ la **restriction d'accès** à certains éléments des objets de la classe,
- ✓ la **protection** de ces éléments par des méthodes dans un but de non corruption (**Encapsulation**)
- ✓ la **garantie de la cohérence de ces objets** tout au long de leur vie.

Visualisons le futur type abstrait **Personne** en **Java** que l'on va construire progressivement - avec des explications associées - et qui va remédier aux remarques soulevées précédemment.

- 3 - Le type *Personne* en Java

```
public class Personne {  
    //-----  
    // Les caractéristiques physiques  
    //-----  
    private String nom ;  
    private String societe ;  
  
    //-----  
    // Les caractéristiques comportementales  
    //-----  
  
    // 1- Construit un objet Personne de société inconnue et de nom correspondant  
    // au paramètre nom  
    public Personne (String nom) {  
        this.nom = nom.toUpperCase();  
        societe = "?"; // Par convention, la chaîne de caractères ? stipule que la  
                       // personne n'est pas employée  
    }  
    public void integrerSociete (String entreprise) {  
        societe = entreprise ;  
    }  
  
    // 2 - Affiche les caractéristiques de la personne  
    public void afficher() {  
        System.out.print (« Je m'appelle » + nom) ;  
  
        if (societe.equals (« ? » ))  
            System.out.println (« et je ne suis pas salarié. »);  
        else  
            System.out.println (« et je travaille chez : » societe);  
    }  
} // class Personne
```


Exemples d'utilisation à partir du type *Personne*

Personne p1, p2, p3;

p1 = **new** Personne() ; // Refusé à la compilation

p2 = **new** Personne("Martin"); // OK

p3 = **new** Personne(2); // Refusé à la compilation

p2.nom = "Dupond"; // Refusé à la compilation

System.out.println("Je m'appelle"+ p2.nom); // Refusé à la compilation

p2.afficher();

// affiche « Je m'appelle MARTIN Je ne suis pas salarié. »

p2.integrerSociete("Java SARL"); //OK

p2.afficher();

// affiche « Je m'appelle MARTIN et je travaille chez Java SARL »

Nous allons, dans la suite de ce document, donner un sens à chacun de ces refus.

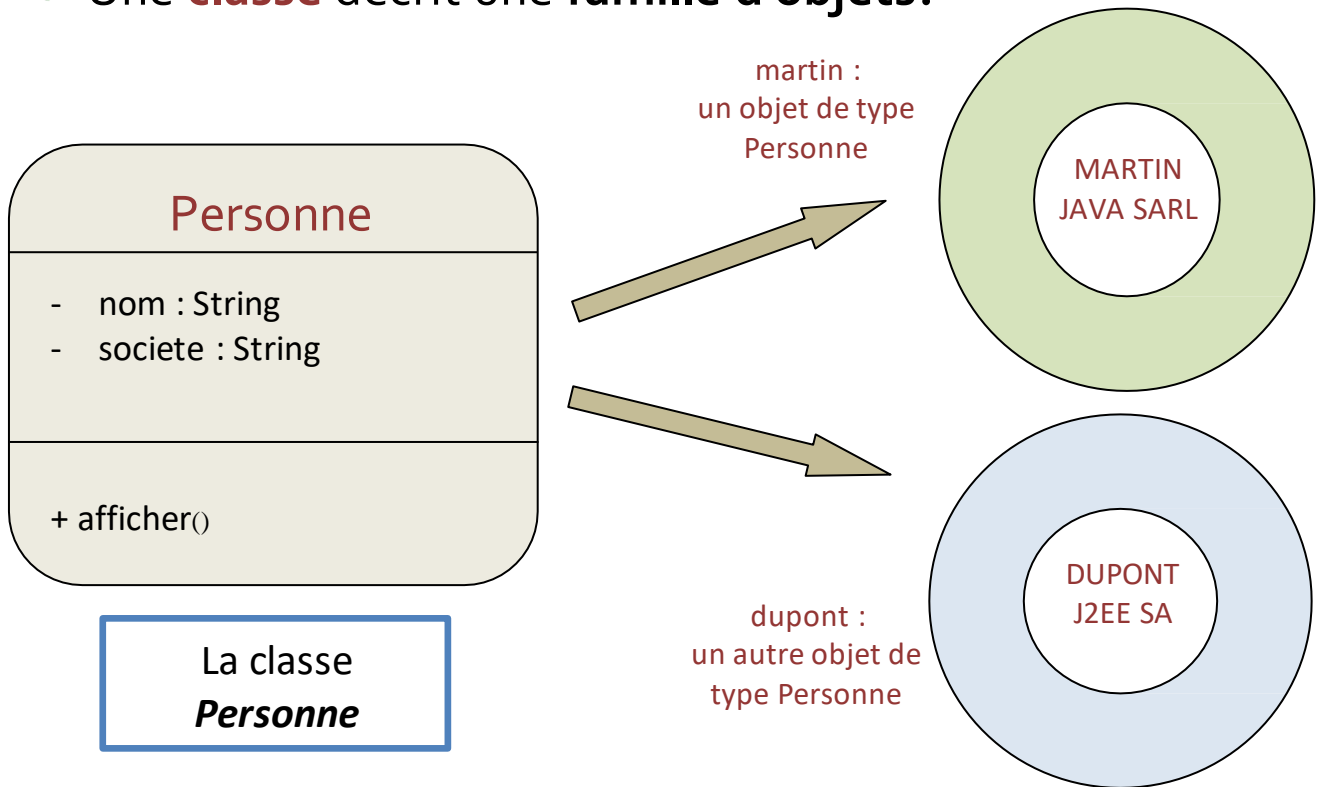
- 4 - La Programmation Orientée Objet

Vocabulaire et définitions

- ✓ En P.O.O., un objet est caractérisé par ses **informations** et ses **comportements** : en programmation, on dira qu'un objet encapsule données et traitements.
- ✓ Un objet reçoit des **messages** qui déclenchent ses **comportements**. (Un objet client envoie des messages aux objets serveurs).
- ✓ Toute **communication** dans une application objet s'effectue par le biais de **messages**.
- ✓ La programmation orientée objet (P.O.O.) apporte de la ^{Sureté}~~sécurité~~ par rapport à la programmation procédurale classique puisque l'objet contrôle chacun des comportements qui lui sont propres.

- 5 - La classe : mécanisme du support de l'encapsulation

✓ Une **classe** décrit une **famille d'objets**.



Un peu de vocabulaire ...

- ✓ Les objets appartenant à une même classe *MaClasse* sont appelés les instances de *MaClasse*.
- ✓ Les données d'un objet sont ses variables d'instance ou attributs d'instance.
- ✓ Les comportements communs à toutes les instances sont représentés par des méthodes d'instance.

- 6 - Version 1 de la classe *Personne*

Pour le cahier des charges N°1...

- ✓ Une personne est décrite par deux informations :
 - son **nom**, enregistré en **majuscules**.
 - la **société** qui l'emploie.
- ✓ Une personne doit être capable de s'identifier, en affichant les informations qui la caractérisent :

... on peut définir :

```
class Personne {  
  
    // Deux variables d'instance (pour l'instant non protégées)  
    public String nom;  
    public String societe;  
  
    // Une méthode  
    public void afficher() {  
        System.out.println ("Je m'appelle " + nom);  
        System.out.println ("Je travaille chez " + societe);  
    }  
} // class Personne, version provisoire
```

Quelques problèmes résident dans le code ci-dessus :

1. On peut **instancier** (c'est-à-dire créer) des objets ***Personne*** indéterminés :

```
Personne p = new Personne() ;
```

2. On peut modifier, accidentellement ou non, les variables d'instance d'un objet ***Personne***, une fois instancié.

```
p.nom = "durand" ; // nom en minuscules
```

3. **On ne peut pas appliquer** des règles de gestion comme par exemple, respecter la règle suivante :

« Une *Personne* doit toujours avoir un nom qui ne peut être modifié; elle peut, en revanche, changer de société »

Tous ces problèmes, qui visent à **apporter de la sécurité** et garantir la **cohérence** des données, vont être petit à petit résolus, grâce aux mécanismes de la **P.O.O.** que nous allons mettre en œuvre.



Une nouveau cahier des charges pour le type abstrait *Personne*

Complétons et précisons les règles liées aux objets de type *Personne*

- ✓ 1 - Une personne est décrite par deux informations : son **nom** et la **société** qui l'emploie.
- ✓ 2 - Une personne a **toujours** un nom. Ce nom doit être invariable. C'est une chaîne de caractères, exprimés en **majuscules**.
- ✓ 3 - Une personne n'est **pas nécessairement associée** à une société : elle peut être considérée comme un indépendant ou comme une personne sans activité.
- ✓ 4 - Une personne doit pouvoir indiquer, lors de l'affichage, si elle est, ou non, salariée (c'est à dire si sa société est identifiée et différente de notre convention « ? »).
- ✓ 5 - Lorsqu'une personne est associée à une société, le nom de cette société ne peut excéder **30 caractères**, dans laquelle les lettres sont en majuscules.
- ✓ 6 - Une personne **peut changer de société** , ou perdre toute association à une société.
- ✓ 7 - Enfin, une personne est capable de **s'identifier**, en affichant les informations qui la caractérisent.

Toutes ces **nouvelles contraintes** vont être mises en œuvre grâce aux concepts de la **P.O.O.**

Une règle d'or, absolue :

En **Programmation Orientée Objet**, l'accès aux informations doit être contrôlé.

Si la classe **Personne** définit deux variables d'instance **nom** et **societe**, le développeur qui utilise (après l'avoir créé) un objet **Personne** ne doit pas pouvoir affecter **sans contrôle** une nouvelle valeur à l'une ou l'autre de ces variables d'instances (v.i.)

C'est la garantie de la cohérence de l'objet.

- 7 - Type d'accès public ou privé ?

Implémentation nouvelle de la classe *Personne*

```
class Personne {  
  
    private String nom;  
    private String societe;  
  
    public void afficher() {  
        System.out.println(«Je m'appelle » + nom);  
        System.out.println(«Je travaille chez » + societe );  
    }  
} // class Personne, version provisoire
```

Les variables d'instance étant maintenant *private*, si on écrit :

```
Personne p = new Personne();  
p.nom = «durand» ; // Impossible d'accéder au champ (private)
```

La compilation refuse la seconde instruction : *p.nom*, privée, est **inaccessible** pour l'utilisateur de la classe *Personne*.

Accès *public* ou *private* ?

public	Les variables et méthodes de la classe <i>MaClasse</i> , définies avec le modificateur public sont accessibles partout où <i>MaClasse</i> est instanciable.
private	Les variables et méthodes d'une classe <i>MaClasse</i> , définies avec private ne sont accessibles <u>que dans la classe</u> <i>MaClasse</i> .

Les deux acteurs de la programmation objet

✓ Le concepteur de la classe *MaClasse* :

Celui qui définit *MaClasse* : dans les méthodes de *MaClasse*, il a **directement** accès aux variables d'instance et aux méthodes **privées**.

✓ Un utilisateur de la classe *MaClasse* :

Celui qui instancie *MaClasse* : il n'a directement **accès** qu'aux variables et méthodes **publiques** de *MaClasse*.

- 8 - Contrôle des instantiations

✓ L'instanciation par défaut

Avec la définition de la classe **Personne** de la page précédente, si nous exécutons :

```
Personne p=new Personne() ;  
p.afficher() ;
```

nous voyons s'afficher : Je m'appelle **null**
 Je travaille chez **null**

- ✓ Quand le concepteur de la classe n'a pas spécifié de mécanisme d'instanciation explicite, **Java** en fournit un par défaut, appelé constructeur par défaut.
- ✓ Le **constructeur** est une méthode particulière en ce sens où elle porte le même nom que la classe à laquelle elle appartient.
- ✓ Le **constructeur par défaut** initialise chaque variable d'instance **vi** avec une valeur par défaut :
 - **null** si **vi** est de type objet,
 - valeur par défaut du type **type** si **vi** appartient à un **type** primitif.

Définir un constructeur

```
class Personne {  
    private String nom;  
    private String societe;  
  
    public Personne (String patronyme) {  
        nom = patronyme.toUpperCase();  
        // Cahier des charges, règle 2  
    }  
    //.... cf page précédente  
    // ....  
} // class Personne
```

Avec le constructeur *Personne (String)*, on peut instancier un objet *Personne* dont le nom est conforme aux spécifications évoquées :

```
Personne p = new Personne(«durand») ;
```

... et on ne peut plus instancier d'objet de nom indéterminé :

```
Personne p = new Personne() ;  
// Erreur de compilation : il n'y a plus de constructeur par défaut !!
```

NB : En **Java**, tous les objets instanciés sont créés dans le tas.

- 9 - Version 2 de la classe *Personne*

```
class Personne {  
  
    private String nom;  
  
    // Convention : l'absence de société sera matérialisée par '?'  
    private String societe;  
  
    // Construit un objet Personne de nom invariable et de societe inconnue  
    public Personne ( String nom ) {  
  
        // Qualification avec this pour distinguer la v.i. du paramètre  
        this.nom = nom.toUpperCase();  
        societe = new String(«?»);  
    }  
    public void afficher() {  
        System.out.println ( "Je m'appelle " + nom );  
        if ( societe.equals( «?» ) )  
            System.out.println( "Je ne suis pas salarié" );  
        else  
            System.out.println ( "Je travaille chez " + societe );  
    }  
} // class Personne, version 2
```

Utilisation de *this*

- Dans un **constructeur**, le mot-clef **this** désigne l'objet en phase de construction.
- Dans une méthode, ce mot-clef désigne l'objet qui traite le message :

```
// Une autre méthode de la classe Personne  
public void affecter( Personne personne ) {  
  
    this.societe = personne.societe;  
    // Ici this est facultatif  
}
```

- 10 - Récapitulons

- ✓ Le constructeur par défaut est encore appelé constructeur sans paramètres ou bien encore constructeur implicite.
- ✓ Un constructeur est une méthode particulière - en général *public* - et dont l'identificateur est le même que celui de la classe et qui est toujours définie sans type de renvoi (même pas *void*).
- ✓ Dès qu'un constructeur explicite est défini, le constructeur par défaut n'est plus disponible, sauf si le développeur le rétablit en définissant explicitement un constructeur sans paramètres.
- ✓ Le constructeur garantit que l'objet va être dans un état initial satisfaisant dès sa création, puisqu'il permet la valorisation des variables d'instance de tout objet.
- ✓ Un objet est responsable des tâches qu'il exécute, et lui seul.
- ✓ Grâce à l'encapsulation, on ne permet pas l'accès direct aux champs d'un objet (qui peut être vu comme une boîte noire).
- ✓ Les objets possédant les mêmes propriétés (données et comportements) sont décrits par une même classe.
- ✓ En Programmation Orientée Objets (P.O.O.) , on cherche à bâtir des objets (et donc des classes) spécialisés en cherchant à favoriser l'utilisation de classes déjà existantes.
- ✓ On veillera à ce qu'une classe soit la plus indépendante possible, tout en évitant qu'elle n'implémente trop de fonctionnalités. La conception, le débogage et la réutilisabilité s'en trouvant nettement plus favorisés.

En P.O.O., trois caractéristiques essentielles :

1. Quel doit être le comportement de l'objet (déterminé par les messages qu'il sait exécuter) ?
2. Quel est l'état d'un objet (aspect courant – modifiable par les messages).
3. Quelle est l'identité de l'objet (Deux objets peuvent posséder les mêmes propriétés et être différents).
4. L'état d'un objet influence son comportement et vice-versa.

- 11 - Types primitifs et type objet

Deux catégories de variables existent en Java :

Dans la séquence de code suivante :

```
Personne personne;  
int age;  
String prenom = "Albert";
```

- la variable *personne* est une référence non initialisée sur le type objet *Personne*,
- la variable *age* est une variable de type primitif (*int*),
- la variable *prenom* est une référence sur le type objet *String*, initialisée avec l'objet *String* "Albert".

Les types primitifs en Java

- Entiers avec signe :
byte (8 bits), *short* (16 bits), *int* (32 bits), *long* (64 bits).
- Réels représentés en virgule flottante :
float (32 bits), *double* (64 bits)
- Caractères : *char* (16 bits, *Unicode*)
- Valeurs logiques :
boolean, deux valeurs *true* et *false*.

- 12 - Variables et valeurs

1°) En **Java**, une variable de type primitif contient sa valeur :

```
int rayon ; // Déclaration d'une variable de type int
```

rayon ? ?

```
rayon = 5 ; // On lui affecte la valeur 5
```

rayon 5

L'emplacement mémoire de la variable contient la valeur associée à la variable.

2°) En **Java**, une variable de type objet désigne sa valeur

```
String nom ; // On déclare un objet de type String
```

nom ? ?

```
// Construction d'un nouvel objet String référencé par nom  
nom = new String ("Dupond");
```

nom → « Dupond »

3°) L'emplacement mémoire de la variable contient une référence sur l'objet associé à la variable.

4°) La valeur par défaut d'une référence est *null*, quel que soit le type objet associé.

Valeurs par défaut

- ✓ Une **variable d'instance** non initialisée par un constructeur explicite est **toujours initialisée** par **Java** avec **la valeur par défaut de son type**.
- ✓ Toutes les autres variables ne **sont** pas initialisées par défaut : elles doivent **obligatoirement être initialisées** par le développeur **avant la première utilisation de leur valeur**.

- 13 - Variables, valeurs et affectations

Affectation de variables de **types primitifs**

L'exécution de la séquence suivante :

```
int x, y ;  
x = 1234 ;  
y = x ;  
x = x + 66 ;  
System.out.println( « x vaut » + x ) ;  
System.out.println( « y vaut » + y ) ;
```

affichera :
x vaut 1300
y vaut 1234

Les valeurs de x et y sont **distinctes**.

Affectation d'une **référence**

Supposons que la classe **Personne** dispose de la méthode **integrerSociete** qui permet d'affecter une valeur à la variable d'instance **societe**.

Alors l'exécution de la séquence :

```
Personne dupont, martin;  
dupont = new Personne (« Dupont »);  
dupont.afficher() ;  
martin = dupont;  
martin.integrerSociete (« Java SARL »);  
dupont. afficher() ;
```

affichera :
Je m'appelle DUPONT
Je ne suis pas salarié
Je m'appelle DUPONT
Je travaille chez Java SARL

Les **références** *dupont* et *martin* désignent le **même objet**.

- 14 - Pile d'exécution, tas d'allocation

Pendant l'exécution d'une application, **Java** gère une pile d'exécution : chaque méthode appelée ajoute à cette pile son environnement, c'est-à-dire ses variables locales et ses paramètres.

Plus généralement, l'exécution d'un bloc de code empile un environnement, qui est ensuite dépilé quand l'exécution du bloc est terminée.

Une variable de pile (variable locale ou paramètre) peut être de type primitif ou de type objet. Elle est détruite quand son environnement est dépilé.

Un objet n'est jamais enregistré dans la pile. Il est construit dans un espace d'allocation dynamique, le tas d'allocation (*heap*), géré par le processeur **Java**.

Un objet ne peut être détruit tant qu'il existe au moins une référence sur lui. C'est le ramasse-miettes (**garbage collector**) de la machine virtuelle Java (JVM) qui récupérera l'espace que cet objet occupait : le développeur n'a pas à se soucier de cette opération.

Exemple d'exécution

```
{ // Bloc N°1
    Personne personne1 = null;
    int age= 54;

    { // Bloc N°2
        Personne personne2 = new Personne (« Dupont »);
        personne2.afficher();
        personne1 = personne2;
    } // fin bloc N°2
    personne1.afficher() ; // personne1 désigne bien l'instance
                          // (unique) créée dans le bloc N°2
} // fin bloc N°1
```

affichera :

Je m'appelle DUPONT
Je ne suis pas salarié

- 15 - Définissons des accesseurs

✓ Comment changer de *société* ?

Nous n'avons pour l'instant **aucun moyen** d'associer une société à une personne puisque la variable d'instance *societe* est **privée**.

Il faut donc définir de nouvelles **méthodes publiques** dans la classe qui permettront de **modifier** et **d'afficher** la société d'une personne.

✓ Créer deux méthodes supplémentaires

```
// Accès en consultation
public String lireSociete() {
    return societe;
}

// Accès en modification
public void changerSociete(String entreprise) {
    // Attention au cahier des charges !!
    societe = entreprise;
}
```

Les méthodes *lireSociete* et *changerSociete* sont respectivement des **accesseurs** en **lecture** (*accessor*) et en **écriture** (*mutator*) de la variable d'instance *societe*.

Intérêt des accesseurs

- Gérer l'accessibilité des variables **privées**

On peut **limiter** l'accès aux données à la lecture (avec uniquement un **accesseur en consultation**) ou **étendre** l'accès en lecture/écriture (avec deux accesseurs en **consultation** et en **modification**).

- Gérer l'intégrité des données

L'accesseur en modification comporte **souvent du code de contrôle** qui permet de valider la nouvelle valeur de la variable.

- 16- Une classe *Personne* robuste

Améliorons notre classe *Personne* :

```
public String lireNom() { // Accès en consultation
    return nom;
}
public String lireSociete() { // Accès en consultation
    return societe ;
}
public void quitterSociete() {
    if (societe.equals(" ? »)) { // La personne n'est pas rattachée à une société
        afficher ();           // on décide d'arrêter l'application
        System.out.println ("Impossible de quitter la société") ;
        System.exit(1); // Arrêt de l'exécution, code erreur 1
    }
    societe = " ? » ; // Ici, il y a bien une société à quitter, on applique la convention
}

// Méthode-filtre : renvoie le paramètre nomSociete s'il représente un nom de société
// acceptable selon les règles établies
private String validerSociete(String nomSociete ) {

    if (nomSociete .length() > 30 || nomSociete .equals("?")) {
        // En Java, // représente l'opérateur logique OU
        System.out.println (« Classe Personne, société incorrecte : »+ nomSociete );
        System.exit(2); // Arrêt exécution, code erreur 2
    }
    // Ici, on est sûr que nomSociete est valide : on le retourne
    return nomSociete ;
}
public void affecterSociete(String entreprise) {
    // Avant d'aller dans une société, il faut avoir quitté la précédente
    if ( ! societe.equals(" ? ») ) {
        afficher();
        System.out.println ("Erreur : 1- quitterSociete , puis 2-affecterSociete") ;
        System.exit(1);
    }
    societe = validerSociete( entreprise ).toUpperCase();
}
```

- 17 - Un ou plusieurs constructeurs ?

Revenons sur le premier constructeur

```
public Personne (String nom) {  
    // Construit 1 objet Personne de nom invariable et de societe inconnue  
    this.nom = nom.toUpperCase() ;  
    societe = new String(«?») ;  
}
```

Pour instancier l'individu *Dupont*, de la société **Java SARL**, il faut exécuter :

```
Personne dupont = new Personne(« Dupont ») ;  
dupont.affecterSociete («Java SARL») ;
```

On peut légitimement souhaiter faire cette instanciation en une seule opération.

Un deuxième constructeur

```
public Personne (String nom, String entreprise) {  
    // Construit un objet Personne de nom fixe et de societe connue  
    this.nom = nom.toUpperCase();  
    societe = validerSociete(entreprise).toUpperCase() ;  
}
```

Notion de signature

Pour le compilateur, il n'y a pas d'ambiguïté entre les deux constructeurs. Ainsi, l'exécution de :

```
new Personne(«Dupont», «Java SARL»);
```

fait appel au **deuxième constructeur** car celui-ci utilise deux chaînes en paramètre.

Plus généralement, la **signature** d'un constructeur ou d'une méthode comprend son **identificateur**, la **liste** et les **types** de ses **paramètres**.

Le compilateur détermine la **méthode à exécuter** en **fonction** de sa **signature**. Des **méthodes différentes** peuvent porter le même nom, à partir du moment où **leur signature respective diffère**.

- 18 - Modificateurs *static* et *final*

✓ Améliorons la gestion des personnes sans société

```
class Personne {  
    private String pasDeSociete = «?» ;  
    private String nom;  
    private String societe;  
    ... // etc.  
}
```

Quels reproches
peut-on faire à cette
implémentation ?



✓ Une variable de classe constante

```
class Personne {  
    private static final String PAS_DE_SOCIETE = «?» ;  
    private String nom;  
    private String societe;  
    ... // etc .  
}
```

final : ce modificateur impose que la déclaration comporte une valeur d'initialisation et empêche toute affectation ultérieure d'une autre valeur. Il permet de définir une information constante.

Par ailleurs, lorsque l'on déclare un élément ***final***, le compilateur est à même d'optimiser le code afin d'améliorer sa vitesse d'exécution.

static : ce modificateur indique que toutes les instances de la classe se partageront un exemplaire unique : une variable de classe. Les éléments ***static*** d'une classe existent quel que soit le nombre d'instances (même 0).

- 19 - Variables et méthodes de classe

Définition

On peut parfois souhaiter disposer de données communes et accessibles à toutes les instances d'une même classe.

Une variable **permanente** et **unique** pour toutes les instances d'une même classe s'appelle une variable de classe.

Une méthode de classe représente un comportement **associé à la classe** elle-même et non pas à une instance particulière de cette classe.

En **Java**, une variable de classe ou une méthode de classe, est définie avec le modificateur **static**.

```
class Personne {  
  
    // Variables de classe  
    //-----  
    private static final String PAS_DE_SOCIETE = «?» ;  
    private static String fichier;  
  
    // .... etc.  
    // Méthodes de classe  
    //-----  
    public static void choisirFichier() {  
        fichier = ...  
        // ... etc  
    }  
    ....  
}
```


✓ Exemple d'utilisation

La classe **System** fournit une **variable de classe** publique, **out**, que l'on exploite dans l'instruction suivante :

```
System.out.println(« Utilisation du flux de sortie »);
```

```
// Elle fournit également une méthode de classe publique : exit :  
System.exit(0) ;
```

- 20 - Récapitulations

Définir une classe

- Variables d'instances généralement **privées**.
- Instanciación :
 - ◆ Pas de constructeur : **Initialisation par défaut**.
 - ◆ Un ou plusieurs constructeurs **explicites** : plus de constructeur par défaut (sauf si un constructeur est défini sans arguments).
 - ◆ Comme pour toutes les méthodes, il peut **exister plusieurs constructeurs**. Il s'agit là à la technique de **surcharge**.
- Méthode d'instance
 - ◆ L'objet est un **paramètre implicite** de la méthode, accessible si nécessaire via la notation **this**.
- Variables et méthodes de classe
 - ◆ définies avec le modificateur **static**.
 - ◆ une méthode de classe ne dispose pas de la référence **this**.

Conventions d'écriture

Dans la pratique professionnelle, on utilise des règles de nommage consistant à utiliser des **verbes à l'infinitif** pour les méthodes :

"quitterSociete(...)" , "afficher()" , lireNom() ,

De même, pour les **accesseurs**, on adopte les conventions suivantes : les préfixes **get** et **set** : **getNom** au lieu de *lireNom*, **setSociete** au lieu de *integrerSociete*.

Cette convention est celle attendue par la technologie des **JavaBeans**, pour retrouver dynamiquement ces accesseurs et valoriser les **v.i.s.**

- 21 - Version finale de la classe *Personne*

```
class Personne {  
  
    // La variable de classe matérialisant le non rattachement à une  
    // société selon notre convention .  
    private static final String PAS_DE_SOCIETE = "?";  
  
    // Les variables d'instance de type String  
    private String nom;  
    private String societe;  
  
    private String validerSociete(String entreprise) {  
        // .....  
    }  
  
    //-----  
    // Deux constructeurs pour instancier  
    //-----  
    public Personne (String nom) {  
        // Construit un objet Personne de societe inconnue  
        this.nom = nom.toUpperCase() ;  
        societe = PAS_DE_SOCIETE;  
    }  
    public Personne (String nom, String societe) {  
        // Construit un objet Personne de nom et societe connus  
        // ...  
    }  
    //-----  
    // Accesseurs en consultation  
    //-----  
    public String getNom() {  
        // ...  
    }  
    public String getSociete() {  
        // ...  
    }  
    // .....
```

```

//-----
// Accesseur en modification
//-----
public void setSociete(String entreprise) {
    // ...
}

public boolean etreSalarie() {
    return ! societe.equals(PAS_DE_SOCIETE);
}

public void quitterSociete() {
    // ...
}

//-----
// Afficher les caractéristiques d'un objet
//-----
public void afficher () {
    System.out.println (« Je m'appelle » + nom) ;

    if ( ! etreSalarie() )
        System.out.println (« Je ne suis pas employé
d'une entreprise ») ;
    else
        System.out.println (« Je travaille chez » +
societe) ;
}

} // class Personne, dernière version

```

- 22 - Ecriture et exécution d'un programme Java

Deux types d'applications

- les applications **autonomes** (***stand-alone***), exécutées via un appel au système d'exploitation.
- les ***applets***, exécutées sous le contrôle d'un **navigateur Web**, ou par l'intermédiaire de ***l'appletViewer***, les ***servlets*** exécutées par un serveur Web.

Structure d'une application stand-alone

Exemple: Le programme **Personne.java**

```
class Personne { // Définition des variables
```

```
    // Définition des méthodes
```

```
    ...
```

```
    public static void main(String args[]) {
```

```
        // Méthode principale appelée au début de l'exécution
```

```
    }
```

```
} // ... class Personne
```

Projet : Personne

Personne.java

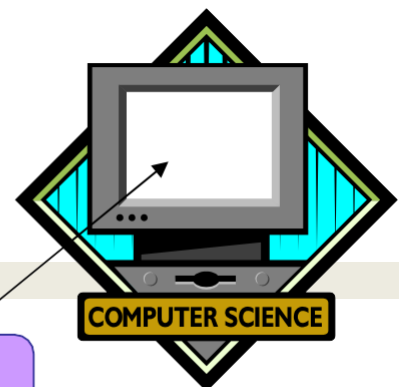
```
class Personne  
{...};
```

Personne.class

bytecode

COMPILATION

EXECUTION



- 23 - Une classe usuelle: la classe *String*

- ✓ La classe *String* permet la manipulation de chaînes caractères.
- ✓ Les objets *String* sont immuables.

Déclaration de chaînes de caractères

```
String uneChaine, uneAutre;  
  
uneChaine = « Voici une chaîne de caractères »;  
uneAutre = « en voilà une autre ... »;  
String encoreUneAutre = new String (« Je suis une chaîne »);
```

Longueur et accès aux caractères

La méthode *length()* fournit la longueur d'une chaîne de caractères. La méthode *charAt()* renvoie un caractère de rang donné.

```
String uneChaine = « Voici une chaîne de caractères »;  
int longueur= uneChaine.length() ;  
char caractere = uneChaine.charAt( 0 ); // caractere prend la valeur 'V'
```

Comparaison

La méthode *equals* permet de comparer deux chaînes.

```
String chaine1 = « Voici une chaîne »;  
String chaine2= « et une autre »;  
if ( chaine2.equals( chaine1 ) ) [ ... ] else [ ... ]
```

Autres méthodes

```
String chaine1 = « Voici une chaîne »;  
String chaine2 = chaine1.substring(2,5);
```

```
// Extraction de sous-chaîne : chaine2 vaut "ici"  
// ch.substring( deb, fin+1) pour obtenir les caractères de rang ' deb à fin'
```

```
String chaine3 = chaine1.trim().toUpperCase();  
// Ecrémage puis majuscule : chaine3 vaut "VOICI UNE CHAINE »
```

```
String chaine4 = chaine1.replace( 'i', ' ?' );  
// Remplacement des occurrences d'un caractère : chaine4 vaut « "Vo?c? une chaîne »
```

```
int rangDeH = chaine1.indexOf( 'h' );  
// rangDeH vaut 11, ( -1 si la lettre était absente de chaîne1 )
```

```
String chaine5 = String.valueOf( 20.6 );  
// Représentation d'une valeur numérique sous forme de chaîne ( méthode de classe )
```

-24 -Importations

Les packages

Des librairies de classes appelées **packages** standard peuvent être utilisées afin d'accéder à des classes d'utilité générale :

► ***java.lang*** : *String, StringBuffer, Math, System ...*

Ce package est importé par défaut.

► ***java.util, java.net, java.awt***, qui doivent être importés explicitement.

Instruction d'importation

```
import java.util.Date ; // Permet l'utilisation de la classe Date .  
import java.util.* ;  
import java.io.* ;  
  
// Permet l'utilisation de toutes les classes des 2 packages util et io.
```

Si une classe n'est pas trouvée, elle sera recherchée dans les **packages importés**.

Les instructions d'importation doivent se trouver **en tête du fichier**.

L'accès aux classes peut se faire **sans importation** mais au prix d'une **plus grande complexité d'écriture** :

```
java.util.Date aujourd'hui = new java.util.Date() ;
```

au lieu de :

```
import java.util.*;  
....  
Date uneDate = new Date() ;
```

- 25 - TP D'APPLICATION

Afin de mettre en œuvre tous ces concepts, rendez-vous dans la batterie d'exercices « *Exercices Java Série 1* » afin de réaliser le TP nommé *PrincipesEncapsulation* dans lequel vous allez revoir et expérimenter l'ensemble des notions de ce support.