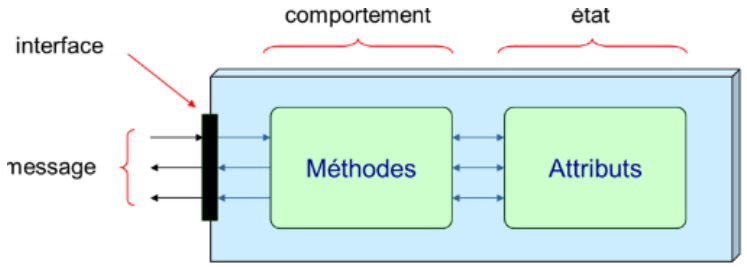
COURS 3

1. Principe d'encapsulation

- **Protéger l'information contenue dans un objet**
  - ✓ Seul l'objet peut modifier ses propres attributs
  - ✓ Masquer les détails de l'implémentation
    - Toutes les méthodes définies du point de vue utilisateur sont publics
    - Les attributs et les autres méthodes sont secrets
- **Un contrat (méthodes publiques) définit ce que fait la classe**
- **L'implémentation des méthodes publiques de ce contrat précise comment elle fait**
- **Les contrats définis dans la classe sont hérités par les sous-classes**



- **Regrouper les données**
  - ✓ Les variables, constantes et objets (instances de classes) caractérisent chaque instance qui sera générée.
- **Définir les services**
  - ✓ Construire les objets (les « constructeurs »)
  - ✓ Modifier les données (les « setters » ou méthodes de modification)
  - ✓ Lire les données (les « getters » ou méthodes d'accès)

2. Niveaux de visibilité des propriétés d'une classe

On distingue 4 niveau de visibilités pour les méthodes les attributs d'instance&class :

visibilité	Public	Protected	Default	Private
la même class	OK	OK	OK	OK
classes de la même package	OK	OK	OK	NO
Une class fille qui se trouve dans le même package	OK	OK	OK	NO
Une class fille qui se trouve dans un autre package	OK	OK	NO	NO
Une class non-fille qui se trouve dans un autre package	OK	NO	NO	NO

- **L'encapsulation** représente un des concepts fondamentaux du monde objet.
  - ✓ Par principe même, on ne peut accéder aux attributs ou opérations d'un objet en dehors de ceux explicitement déclarés publics par la classe.
  - ✓ Ceci garantit l'intégrité des objets en les protégeant de toute modification d'état en dehors d'opérations explicitement définies.

- La visibilité **public** permet à n'importe quel objet d'accéder à la propriété ;

- la visibilité **private** interdit à tout objet instancié à l'extérieur d'accéder à la propriété.

- En règle générale, les constructeurs et les méthodes sont de niveau **public**.

Exercice 1 **:** Définir le diagramme des 2 classes suivantes Point et Segment en respectant et expliquant leurs relations, comme vu dans le Cours2-Java-ING2 UML Diagrammes de cas d'utilisation et de classes, puis implémenter toutes les classes suivantes :

1) Classe *Point* contenant les propriétés suivantes :

- 2 attributs **private** entiers *x* et *y* pour les coordonnées du point
- Constructeur qui initialise les coordonnées *x* et *y* du point avec les paramètres.
- Méthode qui affiche les coordonnées *x* et *y* du point.
- Méthode qui translate les coordonnées *x* et *y* du point, avec 2 paramètres de coordonnées de translation.

2) Classe *Segment* contenant les propriétés suivantes :

- Relation entre 2 points (2 objets de la classe *Point*), avec le même point étant partagé de 0 à plusieurs segments.
- Constructeur qui initialise les 2 points du segment (2 objets de la classe *Point*) avec les 2 points en paramètres.
- Méthode qui affiche les coordonnées des 2 points du segment.
- Méthode qui translate les coordonnées des 2 points du segment, avec 2 points comme paramètres de translation.
- Méthode qui calcule et retourne la longueur réelle d'un segment entre ses 2 points.
  NB : pour rappel, la formule mathématique de la longueur d'un segment entre 2 points (A, B) avec les coordonnées respectives x et y de ces points est la suivante :

  $$\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

  La fonction pour une racine carrée en Java est la fonction **Math.sqrt** dont le prototype de cette fonction est la suivante : *public static double sqrt(double a)*

3) Dans une autre classe, implémenter le **main** qui effectue les instructions suivantes :

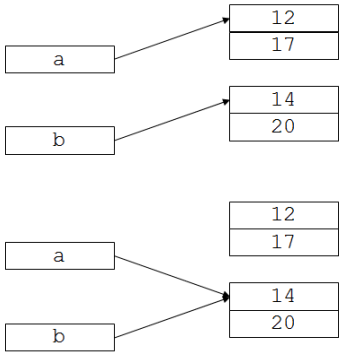
- Instancier les 4 objets suivants de la classe Point avec leurs 2 coordonnées en paramètres : a (12, 17), b (14,20), c (2, 5) et d (-3, 4).
- Instancier un objet s de la classe Segment avec les deux objets a et b précédents de la classe *Point*.
- Appeler la méthode d'affichage pour les 4 objets a, b, c et d précédents de la classe Point.
- Appeler la méthode d'affichage des 2 objets a et b de la classe Point après translation de l'objet s de la classe Segment avec les objets c et d de la classe Point en paramètres.

- Appeler la méthode qui calcule et retourne la longueur réelle du de l'objet s de la classe Segment et affiche cette longueur sous la forme du message suivant : "Longueur du segment =" suivi de la longueur.

Dans la méthode de translation de la classe Segment, pourquoi ne peut-on pas translater directement les 2 points en paramètres du segment ? Quelle solution proposez-vous pour y arriver ? Modifiez votre code pour y arriver et testez-le.

3. Affectation et comparaison d'objets

Soit le schéma d'objets suivant :



Pour comparer l'égalité des variables de deux instances

- ✓ Il faut munir la classe d'une méthode à cet effet
- ✓ La méthode *equals()*, héritée de la classe générique **Object**, détermine si l'objet qui appelle la méthode est égal à l'objet qui est passé en argument

Destruction d'un objet

- ✓ Elle est automatique en Java
- ✓ La destruction automatique nécessite un ramasse-miettes (garbage collector) : l'objet est détruit dès qu'il n'est plus référencé.

Exercice 2 **:** Dans le **main** précédent de l'**exercice 1**, comparer les 2 objets a et b de la classe *Point* entre eux, puis comparer les classes de ces 2 objets entre elles.

Qu'en concluez-vous ? Si maintenant on affecte l'objet b dans l'objet a, que se passe-t-il si on compare les 2 objets entre eux ? Qu'en concluez-vous ? Qu'en est-il si on veut accéder aux anciennes valeurs de l'objet a ? Pourquoi ?

Exercice 3 **:** Reprenons la classe *Point* définie précédemment dans l'**exercices 1**. Dans cette classe, ajoutez les propriétés suivantes :

- Méthode redéfinie *toString()* de la classe générique **Object** dont elle hérite (voir l'exemple sur le site Java | toString() - WayToLearnX) retourne une chaine de caractères (**String**) sous la forme « coordonnées = » + *x* + « , » *y*, mais sans l'afficher.
- Le **main** appelle cette méthode *toString()* et l'affiche avec la méthode d'affichage de la classe *Point*.

4. Deux autres niveaux de visibilité : package et protected

- **package** est celle par défaut (visibilité facultative) en Java : elle permet seulement aux classes membres du package (module regroupant un ensemble de classes) d'accéder à ces propriétés.
- **protected** dans le cas de l'héritage permettent seulement aux classes héritières d'accéder à ces propriétés de la classe mère. Nous verrons cela plus tard.

Les modificateurs de visibilité en Java

Visibilité des champs :

	Classe A	Classe B	Classe C	Classe D	Classe E
privateField	v				
packageField	v	v	v		
protectedField	v	v	v	v	
publicField	v	v	v	v	v

5. Les accesseurs aux attributs non visibles

- Les accesseurs sont des méthodes permettant d'accéder aux attributs non visibles à partir d'une classe extérieure à ces attributs. Cet accès peut se faire :
  - ✓ Soit en lecture grâce à une fonction « getter » retournant la valeur de l'attribut non visible : à ne faire que si on autorise de lire la valeur de l'attribut non visible ;
  - ✓ Soit en écriture grâce à une procédure **void** « setter » permettant de modifier la valeur de l'attribut non visible : à ne faire que si on autorise de modifier la valeur de l'attribut non visible.

- Un attribut non visible avec un accesseur en lecture et un en écriture revient à le rendre **public** : dans ce cas, autant qu'il soit public et éviter inutilement ces 2 accesseurs.

Exercice 4 **:** Définir le diagramme des classes suivantes, en respectant leurs 3 packages suivants et les relations citées :

> Dans un package transport, définir la classe *Train* suivante qui définit les propriétés suivantes :

- 2 attributs public entiers : *numero* et *vitesse* (numéro du train).
- Constructeur qui initialise le *numero* et la *vitesse* du train avec leurs paramètres.
- Méthode *augmenter* qui augmente la vitesse de celle indiquée en paramètre.
- Méthode qui affiche le *numero* et la *vitesse* du train.

> Dans un package individu, définir les 3 classes suivantes :

1) Classe *Personne* avec les propriétés suivantes :
- Attribut *nom* **String** protected.
- Constructeur qui initialise le *nom* avec celui en paramètre.
- Méthode qui affiche le *nom* de la personne.

2) Classe *Passager* qui hérite de la classe *Personne* avec les propriétés suivantes :
- Relation avec l'objet de la classe *Train* que le passager réserve.
- Attribut *reservation* entier privé (numéro de réservation du train).

- Constructeur qui initialise son *nom* par héritage de la classe *Personne*, sa *reservation* et son *Train* en paramètres.
- Méthode qui affiche le *nom*, *reservation* du passager et son *numero* de *Train*.

3) Classe *Conducteur* qui hérite de la classe *Personne* avec les propriétés suivantes :

- Relation avec l'objet de la classe *Train* que le conducteur conduit.
- Attribut *salaire* réel privé (salaire du conducteur).
- Constructeur qui initialise son *nom* par héritage de la classe *Personne*, son *salaire* et son *Train* en paramètres.
- Méthode qui affiche le *nom*, *numero* du conducteur et son *numero* de *Train*.

> Dans un package conduite, définir une seule classe contenant le **main** avec les instructions suivantes :

- Instancier un objet de la classe *Train* du package transport avec une vitesse et un numéro de train en paramètres.
- Instancier un tableau de 3 objets de la classe *Passager* du package individu dont leur nom, leur réservation sont en paramètres, choisis en libre arbitre, ainsi que l'objet instancié de la classe *Train*.
- Instancier un tableau de 2 objets de la classe *Conducteur* du package individu dont leur nom, leur salaire sont en paramètres, choisis en libre arbitre, ainsi que l'objet instancié de la classe *Train*.
- Appeler la méthode qui affiche le numéro et vitesse de l'objet instancié de la classe *Train*.
- Appeler la méthode qui affiche les attributs du tableau des 3 objets instanciés de la classe *Passager* : leur nom, réservation et numéro de train.
- Appeler la méthode qui affiche les attributs du tableau des 2 objets instanciés de la classe *Conducteur* : leur nom, salaire et numéro de train.

NB : En respectant votre diagramme de classes avec les packages et leur.s relation.s, implémenter en Java les classes ci-dessus.

Que se passe-t-il dans le **main** du package conduite si on essaie de modifier l'attribut *salaire* de la classe *Conducteur* qui se trouve dans la classe *Conducteur* du package individu ? De même si on essaie d'afficher l'attribut *nom* de la classe Passager, héritant de la classe Personne, du package individu ? Quelle solution préconisez-vous pour remédier à ces problèmes ?

6. Surcharge (redéfinition) des méthodes

➢ **Plusieurs méthodes peuvent porter le même nom mais pas la même signature (prototype)**

- ✓ On distingue leur signature par le nombre, le type et l'ordre de leurs paramètres (arguments)
- ✓ La signature de chaque méthode doit être unique
- ✓ Quand plusieurs méthodes portent le même nom, Java sélectionne la bonne depuis les paramètres fournis à l'appel

Exemple: Les méthodes nommées *deplace* ci-dessous peuvent toutes exister dans la même classe Point vue précédemment.

```
public void translater (int dx, int dy) { x += dx; y += dy; }
public void translater(int dx) { x += dx; }
public void tranlater(short dx) { x += dx; }
```

Exercice 5 **:** Reprenons la classe *Point* définie précédemment dans l'**exercice 1**. Dans cette classe, ajoutez les propriétés suivantes :

- Plusieurs méthodes redéfinies permettant de translater, puis d'afficher un point, dans la classe *Point* avec des paramètres de types différents, notamment quand le nombre de paramètres est différent.
- Le **main** appelle ces différentes méthodes de translation et d'affichage d'un obde la classe *Point* et l'impact que cela pourrait provoquer sur la méthode de translation de la classe *Segment*.

7. Surcharge (redéfinition) de constructeurs

Les constructeurs peuvent accepter des paramètres et être surchargés

- ✓ Cela permet de fournir plusieurs façons d'initialiser les attributs d'un objet
- ✓ Plusieurs constructeurs peuvent être définit pour la même classe tant qu'ils ont chacun une signature unique
- ✓ Java choisit quel constructeur à utiliser d'après les paramètres fournis à **new.**

Exercice 6 **:** Définir une classe *Arbre* avec les propriétés suivantes :

- Un attribut privé entier : la hauteur d'un arbre
- Un premier constructeur par défaut (sans paramètre) qui initialise la hauteur de l'arbre avec une valeur par défaut.
- Un second constructeur surchargé (redéfini) qui initialise la hauteur de l'arbre avec son paramètre.

- Méthode qui affiche la hauteur de l'arbre.
- Le **main** qui effectue les instructions suivantes :
  - Instancier un premier objet de la classe Arbre avec le premier constructeur par défaut.
  - Appeler la méthode d'affichage de ce premier objet.
  - Instancier un second objet de la classe Arbre avec le second constructeur avec en paramètre une hauteur.
  - Appeler la méthode d'affichage de ce second objet.

8. L'autoréférence à un objet : **this**

- Le mot-clef **this** est utilisé depuis un constructeur ou une méthode d'instance pour accéder directement aux membres définis dans le contexte de la classe
  - ✓ Cela permet au constructeur ou à la méthode de :
    - Renvoyer une référence à l'objet qui le contient
    - Contourner l'éclipse d'un attribut par une variable locale du même nom.
- Le mot-clef **this** n'a aucune signification dans une méthode de classe **static**

Exercice 7 **:** Reprenons la classe *Point* définie précédemment dans l'**exercice 1** en y ajoutant les propriétés suivantes :

- Ajouter comme attribut couleur entier **private**, en plus des 2 coordonnées
- Surcharger les constructeurs suivants :
  - Le premier constructeur par défaut (sans paramètre) initialise les attributs avec des valeurs par défaut (de votre choix).
  - Le second constructeur a les coordonnées en paramètres : il appelle le premier constructeur et initialise les coordonnées en attributs avec les paramètres de même nom.
  - Le troisième constructeur a les coordonnées et la couleur en paramètres : il appelle le second constructeur et initialise les attributs avec les paramètres de même nom.
  - Le dernier constructeur par copie a un objet de la classe *Point* en paramètre : il appelle le troisième constructeur et initialise les attributs de l'objet référencé par copie des attributs de l'objet en paramètre.
- Méthode qui change la couleur avec celle en paramètre et de même nom.
- Méthode qui retourne l'objet référencé de la classe *Point*.
- Compléter le **main** pour tester tous ces constructeurs et ces méthodes.