Global Knowledge®

GKCOB : La CONCEPTION OBJET

PRESENTATION

- Qui suis-je ?

PRESENTATION

- **Qui êtes-vous?**

SOMMAIRE

- **Chapitre 1** : LES CONCEPTS OBJET
- **Chapitre 2 :** DU CONCEPTS AU LANGAGE
- **Chapitre 3** : LIENS ENTRE LES CLASSES

LES CONCEPTS OBJET

PRESENTATION GENERALE

- Du binaire aux langages de quatrième génération, de l'ENIAC aux micro-ordinateurs sous Windows, l'évolution de l'informatique suit un même chemin qui se rapproche de l'utilisateur final pour s'éloigner de l'architecture interne de la machine.
- Loin d'être une révolution, l'orienté objet n'est qu’une étape de cette évolution

PRESENTATION GENERALE

- Celle-ci touche toutefois la plupart des domaines de la technique informatique :
  - la **démarche de projets**, en implémentant un cycle itératif et incrémental ;
  - la **modélisation**, en réconciliant les données et les traitements et en apportant des possibilités de liens supplémentaires entre objets ;
  - le **stockage**, en autorisant la sauvegarde des objets dans des structures adéquates, les bases de données objets (SGBDO) ;
  - la **programmation**, évolution logique et naturelle de la programmation structurée à travers des langages spécialisés.

PRESENTATION GENERALE

- **Aborder l'orienté objet par la phase de programmation est donc un contresens.**
  - Un projet peut être qualifié d'orienté objet lorsque la modélisation de l'application développée s'est attachée à isoler des classes et à les ordonnancer en utilisant des liens d'héritage ou de composition spécifiques au modèle objet, et que l'accent a été porté sur la réutilisabilité des classes conçues.
- Systèmes de gestion de bases de données et langages de programmation ne sont que des outils.

PRESENTATION GENERALE

- Si l'orienté objet fait tout juste son apparition dans le monde de la gestion au début des années 90, porté par la vague déferlante des micro-ordinateurs et de leurs interfaces graphiques, c'est néanmoins loin d'être une technologie naissante :
  - le premier langage objets, SIMULA, date de 1967 ;
  - modélisation et programmation orientées objets ont déjà fait leurs preuves en intelligence artificielle.

LES OBJECTIFS : LA MAITRISE DE LA COMPLEXITE

- La maîtrise de la complexité fait appel a deux notions souvent considérées comme antagonistes, la modularisation et l'abstraction.

- La **modularisation** est le découpage d'un problème général en sous-problèmes, puis en sous-sous-problèmes..., jusqu'à l'obtention d'entités facilement appréhendables et conduisant à une **hiérarchie** des composants. Elle est couramment mise en oeuvre lors de l'analyse top-down d'un programme par exemple.

LES OBJECTIFS : LA MAITRISE DE LA COMPLEXITE

- L'**abstraction** consiste à isoler certaines caractéristiques d'un problème de manière à en construire une image représentative. L'abstraction opère par simplification, généralisation, sélection :
  - la simplification dégrossit un acte en écartant les cas particuliers,
  - la généralisation universalise des éléments simples ou complexes en des propriétés plus générales,
  - la sélection distingue un point de vue qui semble approprié.
- L'orienté objet a pour but de concilier ces deux approches

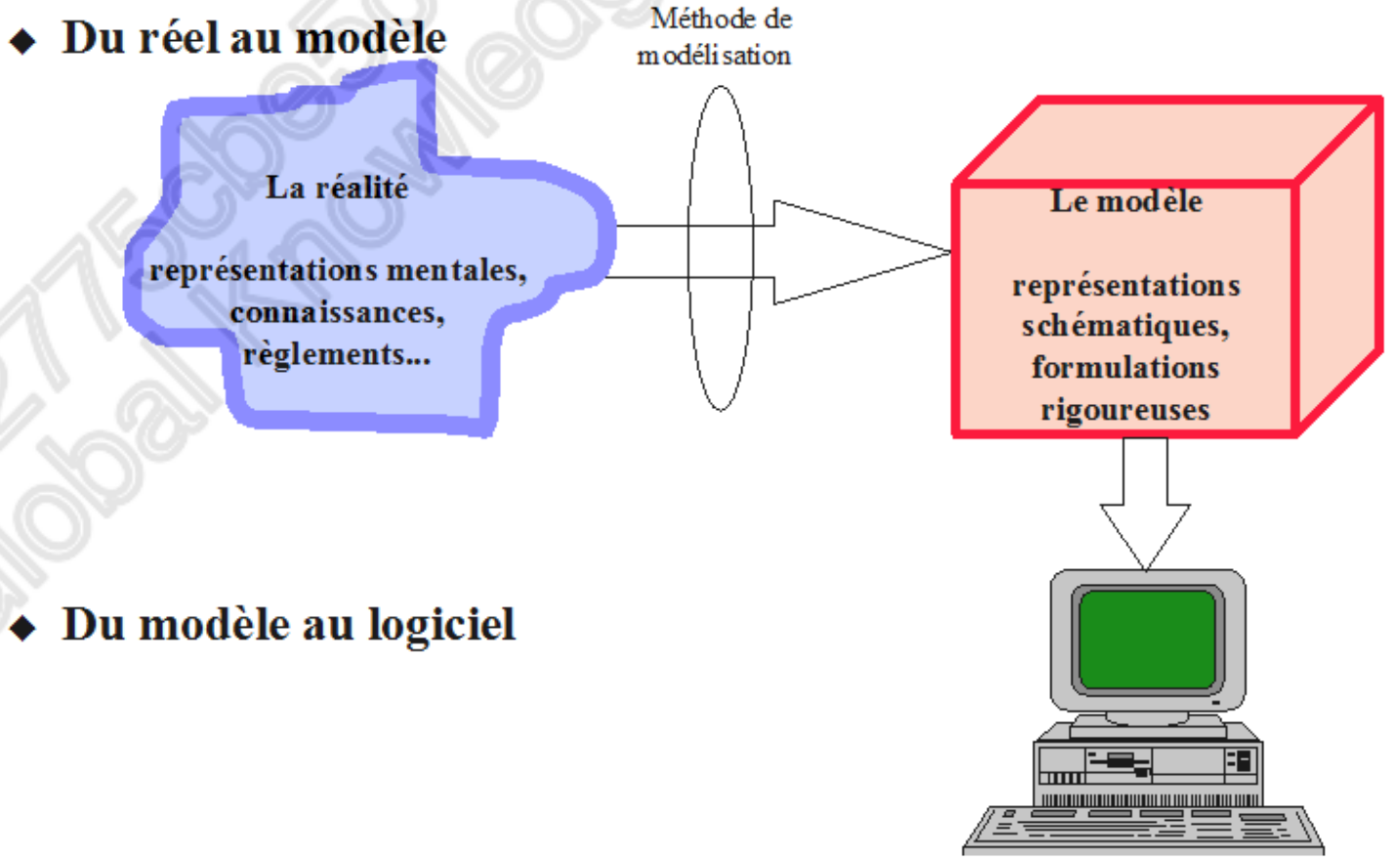
LES OBJECTIFS : LA REUTILISABILITE

- La **réutilisabilité** est une notion ancienne, implémentée jusqu'alors à travers les différentes versions de la programmation structurée.
- Cette implémentation est toutefois extrêmement limitée :
  - elle induit surtout une réutilisation des composants au sein d'un module applicatif, dans le meilleur des cas au sein de tout un applicatif, mais jamais entre plusieurs applicatifs.
  - elle permet principalement une réutilisation de composants basiques, utilitaires de bas niveau tels que les traitements des erreurs graves, les calculs de différence de dates,...
  - elle est mise en oeuvre à travers la modularisation, outil imparfait qui garantit le découpage en composants mais en aucun cas leur aptitude à être réutilisés

UML (UNIFIED MODELING LANGUAGE) : LA MODELISATION

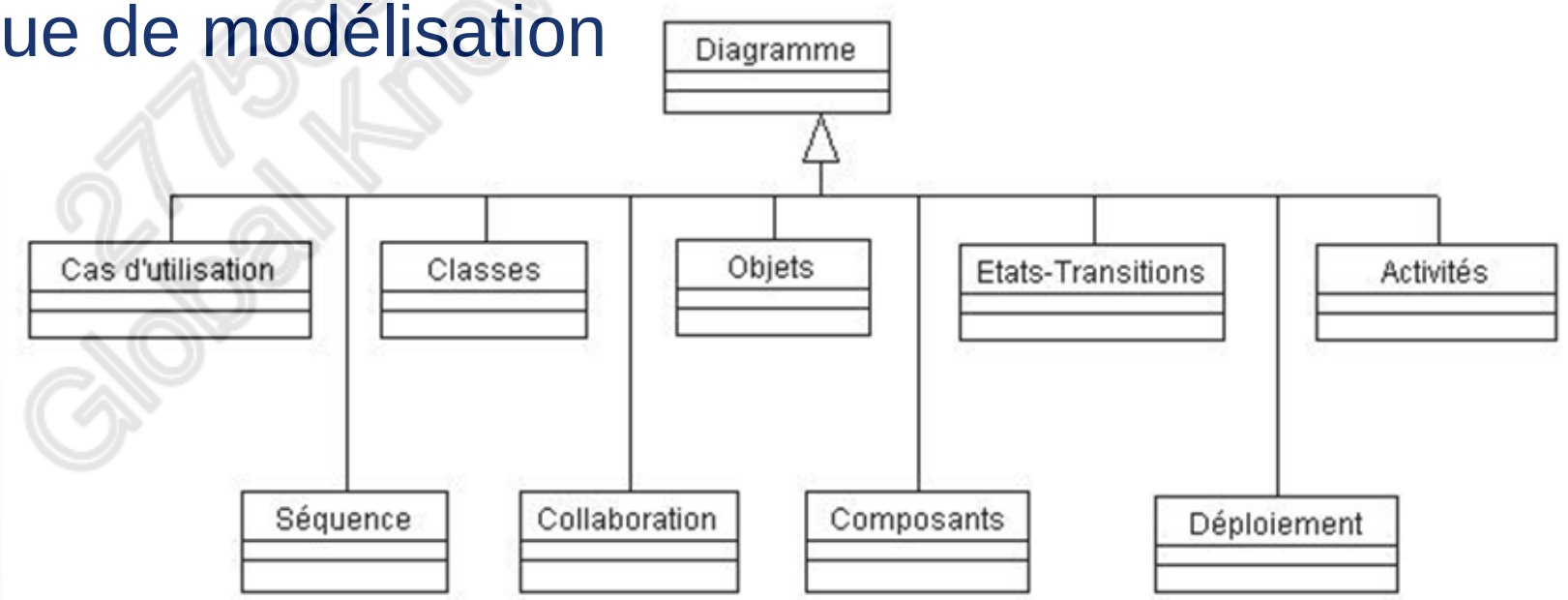
- Un modèle est une représentation simplifiée du réel.

- La modélisation répond à deux objectifs:
  - Communiquer
  - Contrôler



UML (UNIFIED MODELING LANGUAGE) : LA NOTATION UML

- UML est un langage de notation, standardisé par l'OMG (Object Management Group) en 1997.
- UML définit des diagrammes permettant de représenter les différents points de vue de modélisation



LA CLASSE : DEFINITION

- Plus que l'objet, c'est la notion de classe qui caractérise la technologie orientée objet.
- La classe peut-être définie comme un **modèle caractérisant un ensemble d'éléments du système d'information** :
  - **modèle** signifie que la classe n'a aucune existence concrète ; elle ne représente que le moule à partir duquel seront créés les éléments manipulés par le système d'information ;
  - **caractérisant** (au sens de *"définir par un caractère distinctif"*) indique que la classe n'a pas forcément pour vocation de décrire de manière exhaustive les propriétés d'un ensemble d'éléments, mais d'en permettre une abstraction plus ou moins élevée suivant l'avancement du processus de modélisation.

LA CLASSE : DEFINITION

- un **élément du système d'information** peut-être décrit exhaustivement par la connaissance de son **état** et de son **comportement**, ces deux caractéristiques constituant l'**identité** de l'élément :
  - l'**état** d'un élément est représenté par la liste de ses propriétés (habituellement statiques), plus les valeurs courantes de chacune de ces propriétés ;
  - le **comportement** d'un élément est la façon dont celui-ci agit et réagit, en termes de changement de ses états et de circulation de messages.

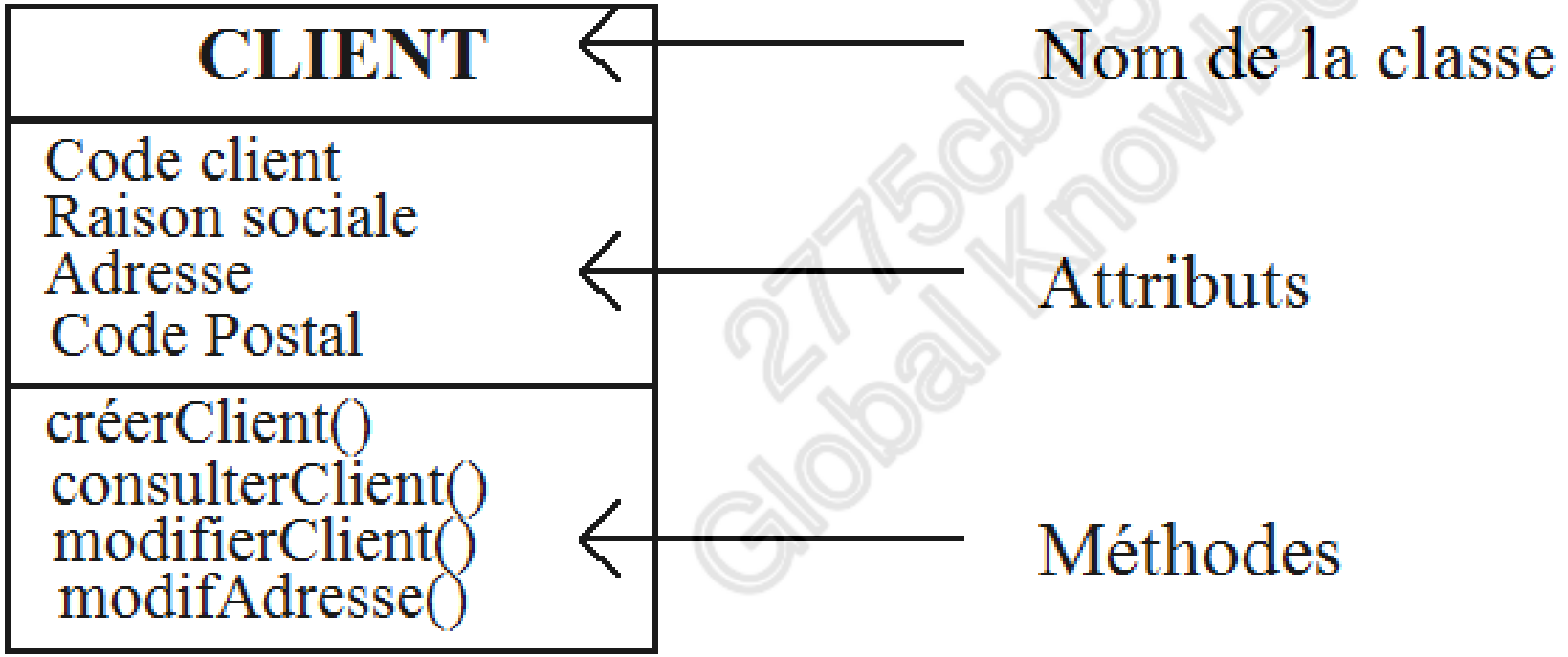
LA CLASSE : DEFINITION

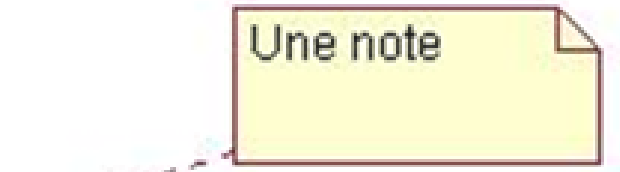
- un **élément du système d'information** peut-être décrit exhaustivement par la connaissance de son **état** et de son **comportement**, ces deux caractéristiques constituant l'**identité** de l'élément :
  - l'**état** d'un élément est représenté par la liste de ses propriétés (habituellement statiques), plus les valeurs courantes de chacune de ces propriétés ;
  - le **comportement** d'un élément est la façon dont celui-ci agit et réagit, en termes de changement de ses états et de circulation de messages.

LA CLASSE : DEFINITION

- La classe déclare donc :
  - la liste des **attributs** (ou des **connaissances**) permettant de connaître précisément l'état des éléments du système d'information qu'elle caractérise ;
  - la liste des **méthodes** permettant de décrire de manière exhaustive le comportement des éléments du système d'information qu'elle caractérise ; les trois principaux types de méthodes sont :
    - les **constructeurs** qui interviennent lors de la création d'un élément ;
    - les **accesseurs** qui permettent d'accéder à l'élément soit en consultation (on parle alors de sélecteur), en mise à jour (modificateur) ou dans un ordre défini (itérateur) ;
    - les **destructeurs** qui interviennent lors de la suppression d'un élément.

LA CLASSE : Exemple :

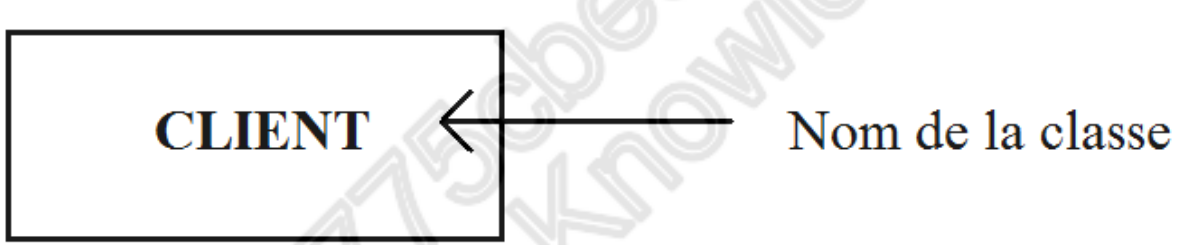




- **NB1** : Le notation adoptée ci-dessus ainsi que dans le reste de ce stage est le formalisme UML (Unified Modeling Language).

LA CLASSE : Exemple :

- **NB2** : On peut utiliser la notation simplifiée suivante :



- **NB3** : En tout état de cause, les attributs et les méthodes figurant sur les schémas de ce support de formation ne prétendent jamais à l'exhaustivité mais ont été choisis en fonction de leur intérêt pédagogique

LA CLASSE : L'ENCAPSULATION

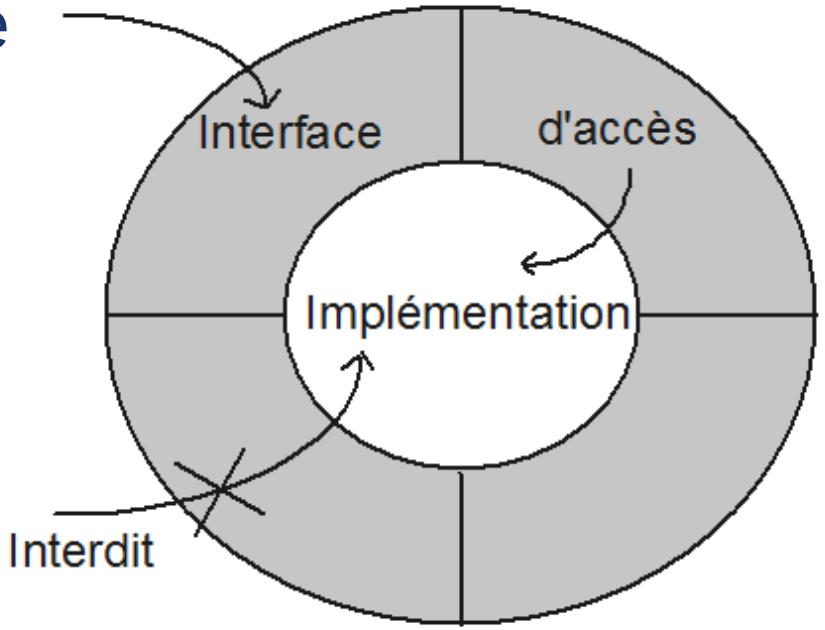
- Pour garantir la cohérence des valeurs d'attributs des occurrences d'une classe, il est intéressant de pouvoir forcer l'utilisateur de l'occurrence (le développeur d'applications) à n'accéder à son état que par l'intermédiaire de ses méthodes.
- Ainsi si l'on veut garantir le respect de la règle de gestion du *Code client* de l'exemple page précédente (incrémentation de 10 en 10 par exemple), on protège cette zone qui devient inaccessible directement depuis l'environnement extérieur à la classe, sa gestion étant prise en charge uniquement par la méthode *Créer client()*.

LA CLASSE : L'ENCAPSULATION

- On fera de même avec les autres attributs de la classe si l'on souhaite s'assurer que pour tout client, les zones *Raison sociale* et *Adresse* ont été valorisées et que la zone *CA annuel total* est initialisée à 0.
- Ce procédé de masquage de tous les attributs d'une classe qui ne seront donc accessibles que par le filtre des méthodes, aussi bien en consultation qu'en mise à jour, est connu sous le nom d'**encapsulation**.

LA CLASSE : L'ENCAPSULATION

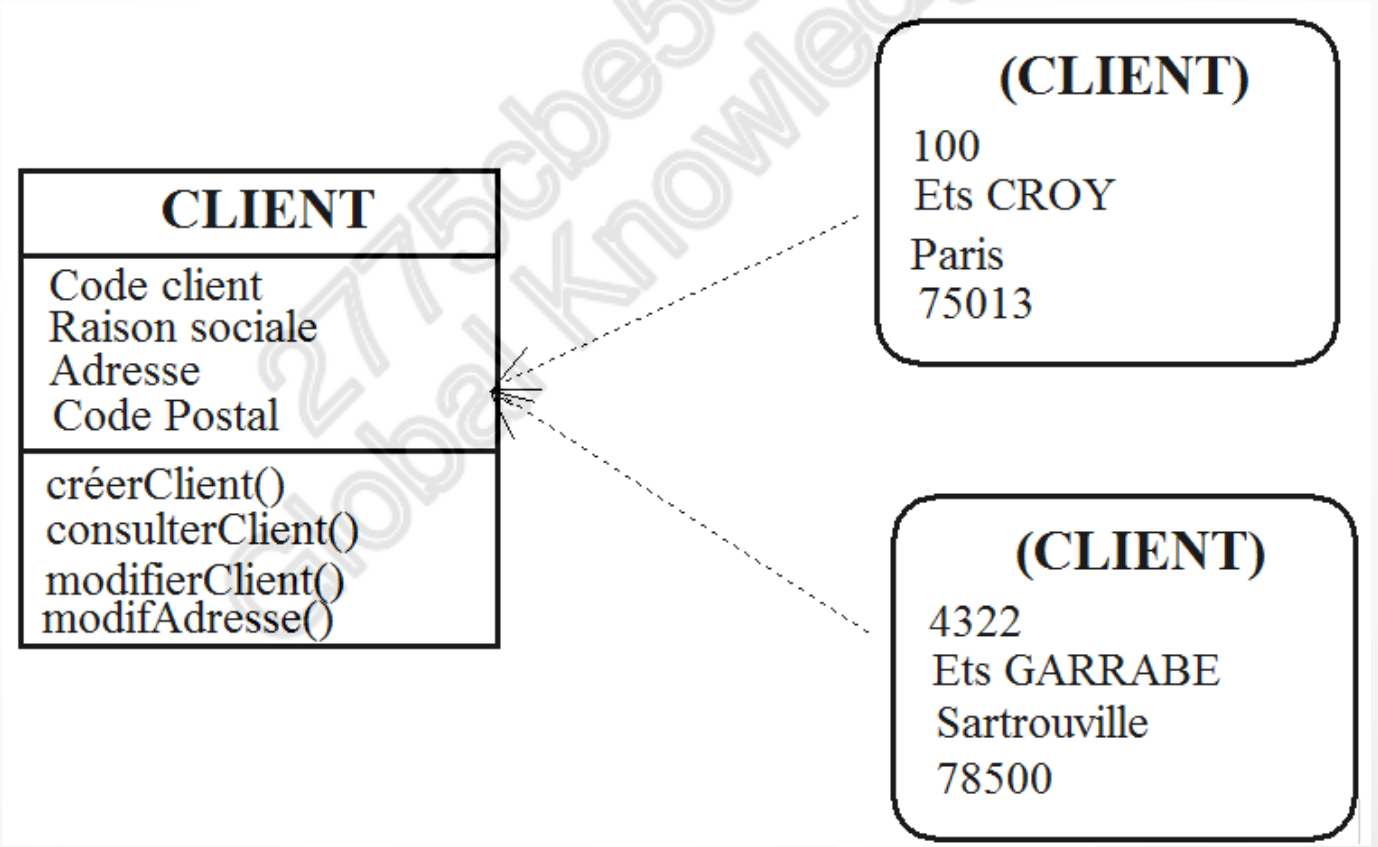
- Loin d'être une caractéristique anodine du modèle objet, l'encapsulation en est un des points forts puisqu'elle induit un couplage faible entre les programmes applicatifs et les éléments du système d'information, ce qui est particulièrement appréciable, entre autres, dans un contexte de maintenance.
- **Remarque :**
  - L'implémentation du modèle objet varie fortement d'un langage de programmation à l'autre. En Java, en C# et en C++ par exemple, l'encapsulation n'est qu'une possibilité offerte alors qu'en Smalltalk elle est incontournable.
  - Quoiqu'il en soit, il est fortement conseillé de la mettre en oeuvre dans tous les contextes



L'OBJET

- L'objet est l'**occurrence ou l'instance d'une classe**.
- Le procédé qui consiste à créer un objet à partir d'une classe est appelé **instanciation**.
- L'instanciation d'un objet est effectuée par l'activation d'un de ses constructeurs. De même, sa suppression ne pourra être obtenue que par l'activation de son destructeur

L'OBJET : Exemple



L'OBJET : Exemple

Voiture

Numéro de série : *Int*
Poids : *double*
Immatriculation : *String*
Kilométrage : *double*

Démarrer ()
Arrêter()
Rouler()

FIAT-UNO-17

233434 : Numéro de série
1500 kg : Poids
8864 YF 17 : Immatriculation
33 000 : kilométrage

Renault-Clio-17

5323454 : Numéro de série
1500 kg : Poids
64 YFT 17 : Immatriculation
23 000 : kilométrage

Peugeot-206-75

3434 : Numéro de série
1700 kg : Poids
8634 YGG 75 : Immatriculation
15 000 : kilométrage

EXERCICE 1 : **DEFINITION DU PRODUIT**

SOMMAIRE

- **Chapitre 1** : LES CONCEPTS OBJET
- **Chapitre 2 : DU CONCEPTS AU LANGAGE**
- **Chapitre 3** : LIENS ENTRE LES CLASSES

DU CONCEPTS AU LANGAGE

LES BASES DU LANGAGE :

LES COMMENTAIRES

- Un commentaire sur une ligne se déclare avec double slash : ***//*** .
- Un commentaire sur plusieurs lignes se déclare en début avec slash étoile : ***/**** et en fin avec étoile slash ****/*** .
- Un commentaire « javadoc », outil qui permet de générer automatiquement la documentation des applications , se déclare en début avec /** et en fin avec */ .

LES BASES DU LANGAGE :

LES EXPRESSIONS

- Une expression se termine toujours par : **;**

LES BLOCS D'INSTRUCTIONS

- Un bloc permet de regrouper plusieurs instructions, il se déclare par **{** et se termine par **}** .
- Toute variable déclarée dans un bloc n'est connue que de ce bloc.

1.4. LES IDENTIFICATEURS

- Attention : java **différencie majuscules et minuscules** :
  - « **Exemple** » et « **exemple** » sont pour lui deux identificateurs différents.

PACKAGES : INTERET

- Un package permet de regrouper des classes et interfaces concernant un même domaine. Il s'agit d'un répertoire qui contient toutes les définitions.
- La création d'un package est liée à un fichier source de définition. Ce source doit contenir en première instruction la déclaration du package auquel il appartient.
- Le code source a une extension *.java*. La compilation donne lieu à une extension *.class*.
- Java propose différents packages.

PACKAGES : DECLARATION

- La déclaration d'un package est la première ligne source d'une classe ou interface.

- Par convention un nom de package débute en minuscule, et ne doit pas contenir de caractère blanc.

```
package nom_du_package
```

REPRESENTATION UML

IMPORTATION

- Par défaut, l'utilisation d'une classe n'est possible qu'à l'intérieur d'un même package. Quand une classe doit utiliser une classe définie dans un autre package, elle est obligée de donner le chemin complet de l'endroit où se trouve la classe ou plus simplement importer le package auquel l'autre classe appartient.
- Il existe deux types d'importation :
  - L'importation de l'ensemble d'un package.
  - L'importation d'un seul élément d'un package.
-

IMPORTATION

```
package nom_du_package1 ;
class ClasseA {…}
class ClasseB {…}
```

```
// Exemple avec importation totale
package nom_du_package2 ;
import nom_du_package1.* ;
class ClasseC {
    ClasseA …
    ClasseB
```

```
// Exemple sans importa
package nom_du_package2
class ClasseC {
    nom_du_package1.Clas
}
```

```
// Exemple avec importation d'une classe
package nom_du_package2 ;
import nom_du_package1.ClasseA ;
class ClasseC {
    ClasseA …
    nom_du_package1.ClasseB
}
```

CLASSES

INTERET :

- Une classe définit un type, utilisable ensuite comme un type de données.

DECLARATION

- La déclaration s'effectue par le mot clé ***class*** et l'on indique le nom de la classe.
- Par convention un nom de classe **débute en majuscule**, et ne doit pas contenir de caractère blanc. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.
- Devant le mot clé, on peut trouver des modificateurs. Derrière on note l'héritage par les mots clés :
  - *extends* pour hériter d'une classe.
  - *implements* pour les interfaces.

CLASSES

```
class ClasseA {…}
```

- ***Exemple avec heritage***

```
class ClasseB extends ClasseA{…}
```

- ***Exemple avec modificateurs***

```
public class ClasseC {…}
```

- ***Syntaxe***

```
[Modificateurs] class NomClasse [extends NomSuperClasse]
[implements Inteface1, Interface2]
{…}
```

MODIFICATEURS

- La visibilité d'une classe est effectuée par les modificateurs.
  - Par **défaut** une classe est visible à l'intérieur d'un même package.
  - **public** : la classe est visible pour les autres packages.
  - **final** : la classe ne pourra pas être super-classe.
  - **abstract** : la classe ne pourra pas être instanciée, elle sert de référence à des sous-classes

METHODES/CONSTRUCTEURS

INTERET

- Une **méthode** définit un traitement à effectuer.
- Le **constructeur** est une méthode particulière de la classe, il permet l'initialisation des variables.

DECLARATION

- La déclaration peut commencer par un modificateur, qui indique le degré de visibilité.
- Une méthode retourne toujours une information ou *void* s'il n'y en a pas.
- Entre parenthèse on indique les paramètres reçus (séparés par des virgules).
- Par convention un nom de méthode débute en **minuscule**, et ne doit pas contenir de caractère blanc.
  - Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.
- Un constructeur **porte le même nom que la classe**, le constructeur se déclare au début, et ne retourne rien à l'appelant donc pas de ***void*** à noter.

METHODES/CONSTRUCTEURS

```
class ClasseA {
  // Constructeur
  ClasseA() {…}

  // Autres méthodes
  void neRetourneRien() {…}
  boolean retourneBooleen() { return false ;}
  void recoitBooleen (boolean a) {…}
}
```

```
// Syntaxe
[Modificateurs] type de retour nomMethode([typeArg1 nomArg1,
typeArg2 nomArg2]) {…}
```

METHODES : SIGNATURE

- Un nom de méthode n’est pas unique.
  - Une même classe peut avoir plusieurs constructeurs et plusieurs fois le même nom de méthode.
- L’association :
  - du nom de méthode
  - du nombre de paramètres reçus et de leur type

  … constitue la **signature** de la méthode. Cette signature est **unique**.
- Quand une classe contient des méthodes ayant le même nom, on parle de **surcharge**.
- On parle de **redéfinition** quand on redéfinit dans une sous-classe une méthode de la classe-mère.

METHODES : MODIFICATEURS

- La visibilité d'une méthode est déterminée par les modificateurs.
- Par défaut une méthode est visible à l'ensemble du package.
  - private : méthode visible par la classe uniquement.
  - public : méthode visible par tous les packages.
  - protected : méthode visible par les sous-classes.
  - static la méthode pourra être invoquée pour la classe elle-même (sans avoir besoin d'instancier un objet)

VARIABLES : DECLARATION

- Chaque variable ou donnée doit être déclarée avant d'être manipulée.
- La déclaration d'une variable doit toujours être précédée du type de variable que l'on veut manipuler.
- Par convention un nom de variable débute en minuscule, et ne doit pas contenir de caractère blanc. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.
- On peut déclarer plusieurs variables en même temps, dans ce cas on les sépare par des virgules.
- On peut initialiser une variable en même temps que sa déclaration.

VARIABLES : Types

Type	
boolean	Booléen valant *true* ou *false*
char	Un seul caractère, à définir avec de simples quotes : 'e'
int	Numérique entier
double	Numérique avec décimales
String	Chaîne de caractères, à définir avec des doubles quotes : "e"

VARIABLES

```
// Déclaration d'une variable
int nomVarEntier;
// Déclaration de plusieurs variables
int nomVarEntier1, nomVarEntier2, nomVarEntier3 ;
// Déclaration et initialisation
int nomVarEntierN = 5 ;
```

- ***Syntaxe :***

[Modificateurs] *type de la variable* nomVariable

VARIABLES : ATTRIBUT D'INSTANCE

- Un attribut d'instance (ou « variable d'instance ») est une donnée définie en dehors de toute méthode. Chaque objet possèdera sa propre valeur : par exemple, le nom d'une personne.
- C'est une variable globale à l'ensemble des méthodes de la classe : toutes peuvent y accéder.

VARIABLES : ATTRIBUT DE CLASSE

- Une classe peut posséder un attribut qui sera partagé par toutes ses instances (par exemple le nombre d'instances créées).
- Une « variable de classe » est créé lors du chargement de la classe en mémoire vive (et non lors de l'instanciation d'objets). Elle doit être créée avec le modificateur *static*.
- Une donnée définie avec le modificateur *final* est une constante. Elle mérite donc d'être également *static*.

VARIABLES : VARIABLE DE TRAVAIL

- Une méthode peut déclarer une variable, dans ce cas elle reste locale à la méthode et n'est pas visible par d'autres méthodes.
- Une donnée définie avec le modificateur *final* est une constante.

VARIABLES : DECLARATION

- Toute donnée, ou variable doit être déclarée avant d'être manipulée.
- La déclaration d'une donnée doit toujours être précédée du type de donnée que l'on veut manipuler.
- Par convention un nom de donnée débute par une minuscule, et ne doit pas contenir de caractère spécial. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.
- On peut déclarer plusieurs données en même temps, dans ce cas on les sépare par des virgules.
- On peut initialiser une donnée en même temps que sa déclaration.

VARIABLES : DECLARATION

- Exemples déclaration et initialisation de variables

```
// Déclaration d'une variable
int nomVarEntier;
// Déclaration de plusieurs variables
int nomVarEntier1, nomVarEntier2, nomVarEntier3 ;
// Déclaration et initialisation
int nomVarEntiern=5 ;
boolean vrai=true ;
```

- ***Syntaxe***

[Modificateurs] *type de la variable* nomVariable [=valeur];

VARIABLES : MODIFICATEURS

- La visibilité des attributs (d'instance ou de classe) est déterminée par les modificateurs.
- Par défaut un attribut est visible à l'ensemble du package.
  - private : attribut visible par la classe uniquement.
  - public : attribut visible par tous les packages.
  - protected : attribut visible par les sous-classes.
  - final déclaration d'une constante.
  - static attribut partagé par toutes les instances.

  - Remarque : Une variable de travail n'est accessible que dans la méthode où elle a été déclarée. La notion de modificateur ne s'applique donc pas, sauf éventuellement ***final*** s'il s'agit d'une constante.

VARIABLES :

```
class ClasseA
{
  // Variable d'instance
  int nomVarEntier;
  // Constante partagée par toutes les instances
  static final double PI = 3.141 ;
  // Constructeur : initialisation des variables d'instance
  ClasseA()
  {
    nomVarEntier = 10 ;
  }
  // Autres méthodes
  void neRetourneRien()
  {
    // Variable de travail
    boolean nomVarBool = false ;
  }
}
```

OBJETS : CREATION

- La création d'un objet s'effectue avec l'opérateur *new*, puis on utilise le nom du constructeur de la classe que l'on veut instancier.

```
// Instanciation
new ClasseA() ;

ClasseA monObjet = new  ClasseA() ;
```

OBJETS : MANIPULATION

- Pour manipuler cet objet, il doit être memorisé dans une variable. Le type de la variable est la classe.

```
// Définition de la Classe
public class ClasseA
{
  // Constructeur
  public ClasseA {}
  // Méthode
  public int traitement() {return 1
}
```

```
// Définition des variables de travail
ClasseA variable;
int retour;
// Instanciation
variable = new ClasseA() ;
retour = variable.traitement();
ClasseA autre = new ClasseA();
int test = autre.traitement();
```

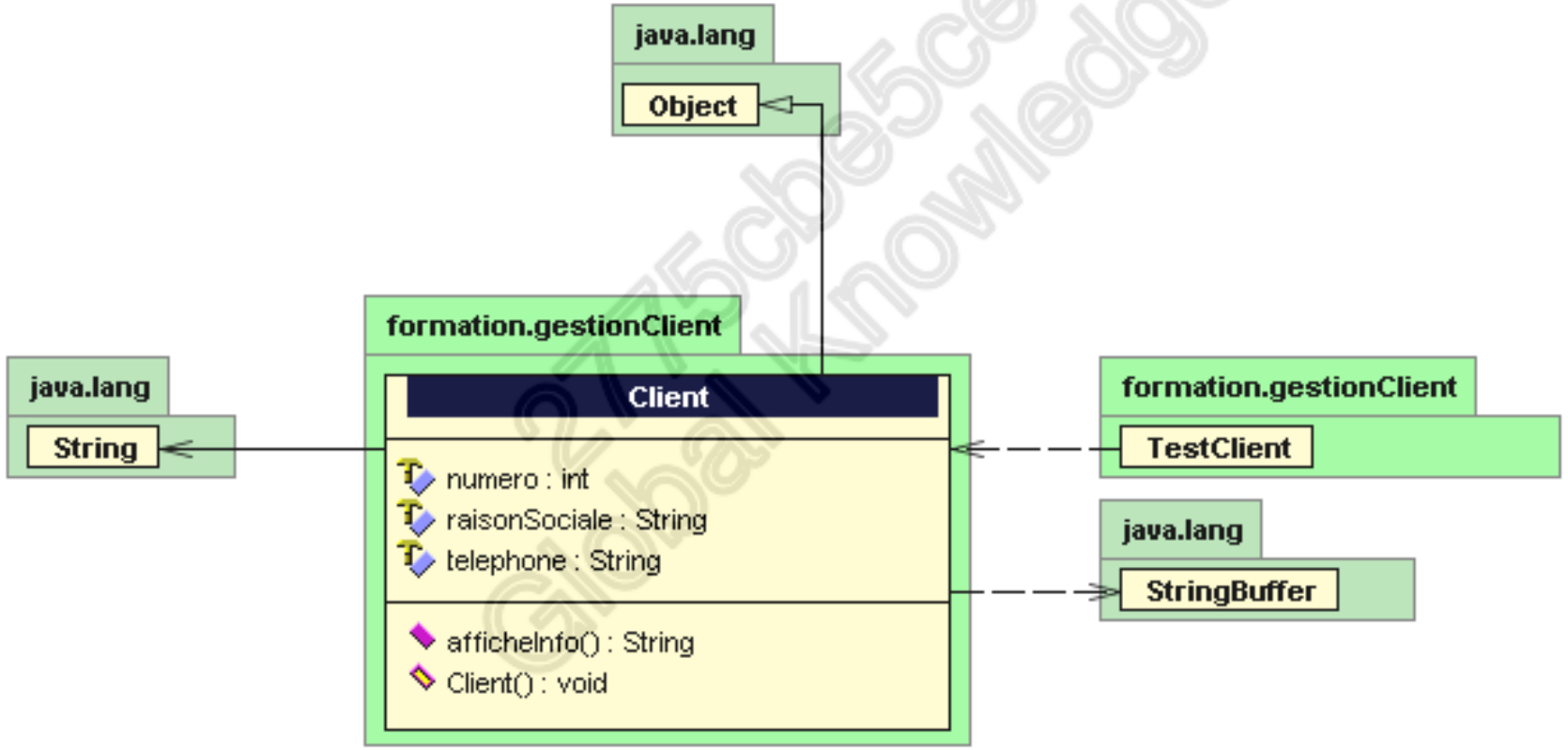
EXERCICE 1 : Definition Produit

- Faire le 1.2

EXEMPLES : DEFINITION DE LA CLASSE CLIENT

- Pour cette classe Client, on mémorise le numéro du client, sa raison sociale et son numéro de téléphone. On définit :
  - Un constructeur, recevant en argument le numéro de client à créer.
  - Un traitement permettant de retourner une chaîne de caractères affichant le numéro et la raison sociale du client, par exemple « le client est : 123 ETABLISSEMENT DUPONT ». L'opérateur de concaténation est le signe +.
    - Cette méthode se nomme `afficheInfo()`.

EXEMPLES : DIAGRAMME UML



EXEMPLES : CODE JAVA

```
package formation.gestionClient ;
public class Client
{
  // Déclaration des variables d'instance
  int numero ;
  String raisonSociale ;
  String telephone ;

  // Constructeur : permet d'initialiser les variables d'instance
  public Client (int num)
  {
    numero = num ;
  }
  // Envoi d'une chaine de caractères affichant le numéro et la raison sociale.
  public String afficheInfo()
  {
     return "le client est : " + numero + " " + raisonSociale ;
  }
}
```

EXEMPLES : DEFINITION DU TEST CLIENT

- Un programme exécutable en java doit posséder une méthode ***main***.
- Cette méthode ne retourne aucun argument et doit être déclarée avec les modificateurs ***public*** et ***static***.
- Elle reçoit en paramètre un tableau de chaîne de caractères.
- Pour afficher des informations sur la console système, on utilise la méthode ***println*** de la classe *System* pour la variable de classe *out*.

EXEMPLES : DEFINITION DU TEST CLIENT

```
class TestClient
{
  // Déclaration du constructeur
  TestClient () {}
  // Programme principal
  public static void main (String args[])
  {
    // création d'un objet client
    Client client1 = new Client (1000);
    // modification de la raison sociale de l'objet client1
    client1.raisonSociale = "Société A";
    // modification du téléphone de l'objet client1
    client1.telephone = "0158225858";
    // affichage à la console système du traitement d'affichage du client1
    System.out.println( "AFFICHAGE " + client1.afficheInfo()) ;
  }
}
```

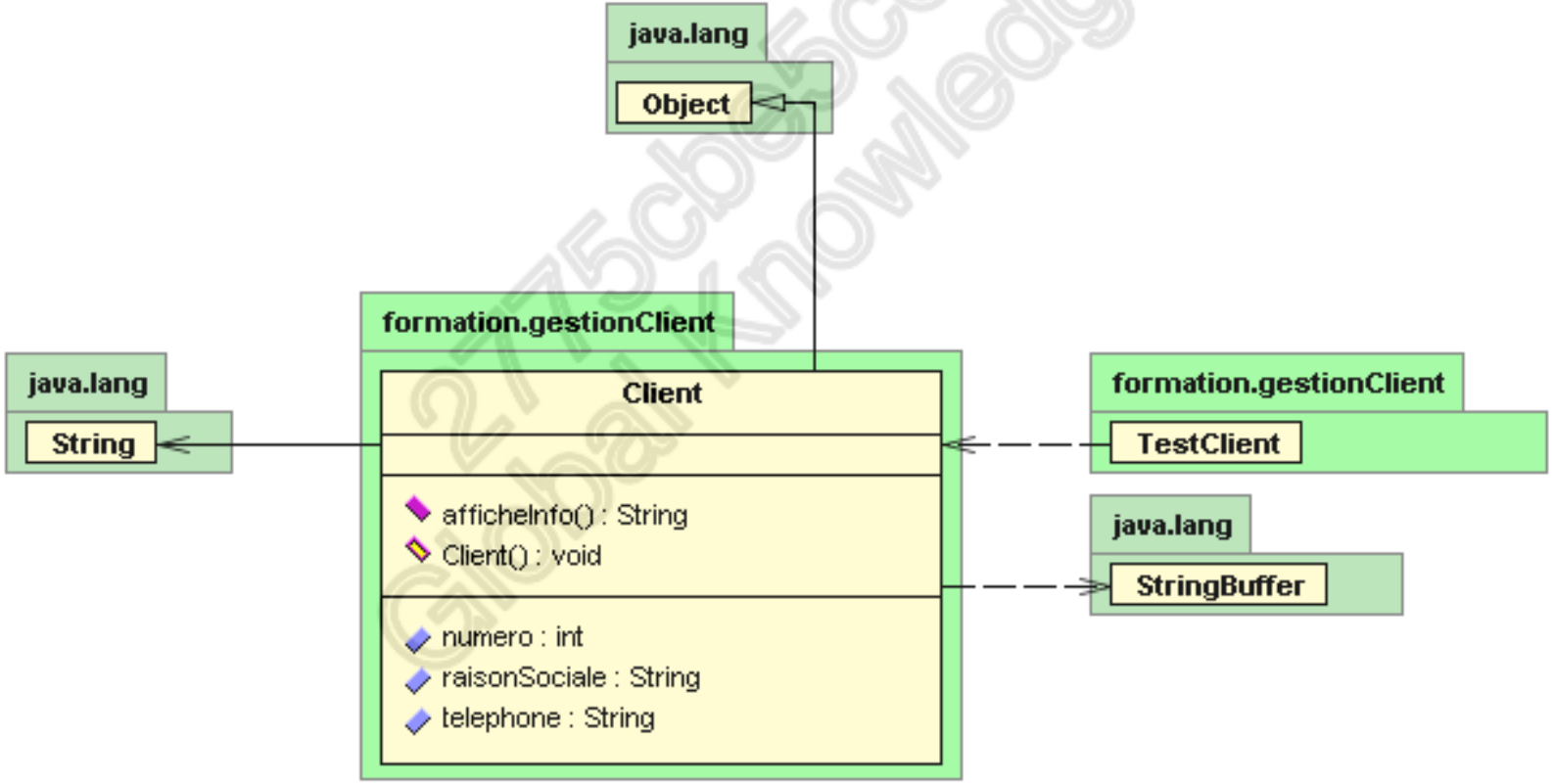
Résultat

```
AFFICHAGE le client est : 1000  Société A
```

ENCAPSULATION : DEFINITION DU CLIENT

- Modification de la classe Client, afin de mettre en oeuvre l'encapsulation.
- On rend les variables d'instance inutilisables, en mettant le modificateur *private*.
- On définit les accesseurs, permettant de récupérer et de mettre à jour les variables d'instances.

ENCAPSULATION : Diagramme UML



ENCAPSULATION : Code java

```
pac
pub
{
   /
   p
   p
   p
   /
   p

   }
   /
   p

   }
   p

   }
```

```
   public String getTelephone() {
      return telephone;
   }
   public void setTelephone(String phone) {
      telephone = phone ;
   }
   public int getNumero() {
      return numero ;
   }
   public void setNumero(int num) {
      numero=num;
   }
   // Envoi d'une chaine de caractères affichant le numéro et la raison sociale.
   public String afficheInfo() {
      return « le client est » + numero + « »+ raisonSociale ;
   }
}
```

```
class TestClient
{
  // Déclaration du constructeur
  TestClient () {}
  // Programme principal
  public static void main (String args[])
  {
    // Test d'un premier objet client
    // Création d'un premier objet client
    Client client1 = new Client (1000);
    // Modification de la raison sociale de l'objet client1
    client1.setRaisonSociale ("Société A");
    // Modification du téléphone de l'objet client1
    client1.setTelephone("0158225858");
    // Affichage à la console système du traitement d'affichage du client1
    System.out.println( "AFFICHAGE " + client1.afficheInfo()) ;
    // Affichage à la console système du téléphone du client1
    System.out.println( "AFFICHAGE " + client1.getTelephone()) ;
    // Test d'un deuxième objet client
    // Création d'un deuxième objet client
    Client client2 = new Client (2000);
    // Modification de la raison sociale de l'objet client2
    client2.setRaisonSociale ("Etablissement D");
    // Affichage à la console système du traitement d'affichage du client2
    System.out.println( "AFFICHAGE " + client2.afficheInfo()) ;
  }
}
```

Résultat obtenu à la console système :

```
AFFICHAGE le client est : 1000  Société A
AFFICHAGE 0158225858
AFFICHAGE le client est : 2000  Etablissement D
```

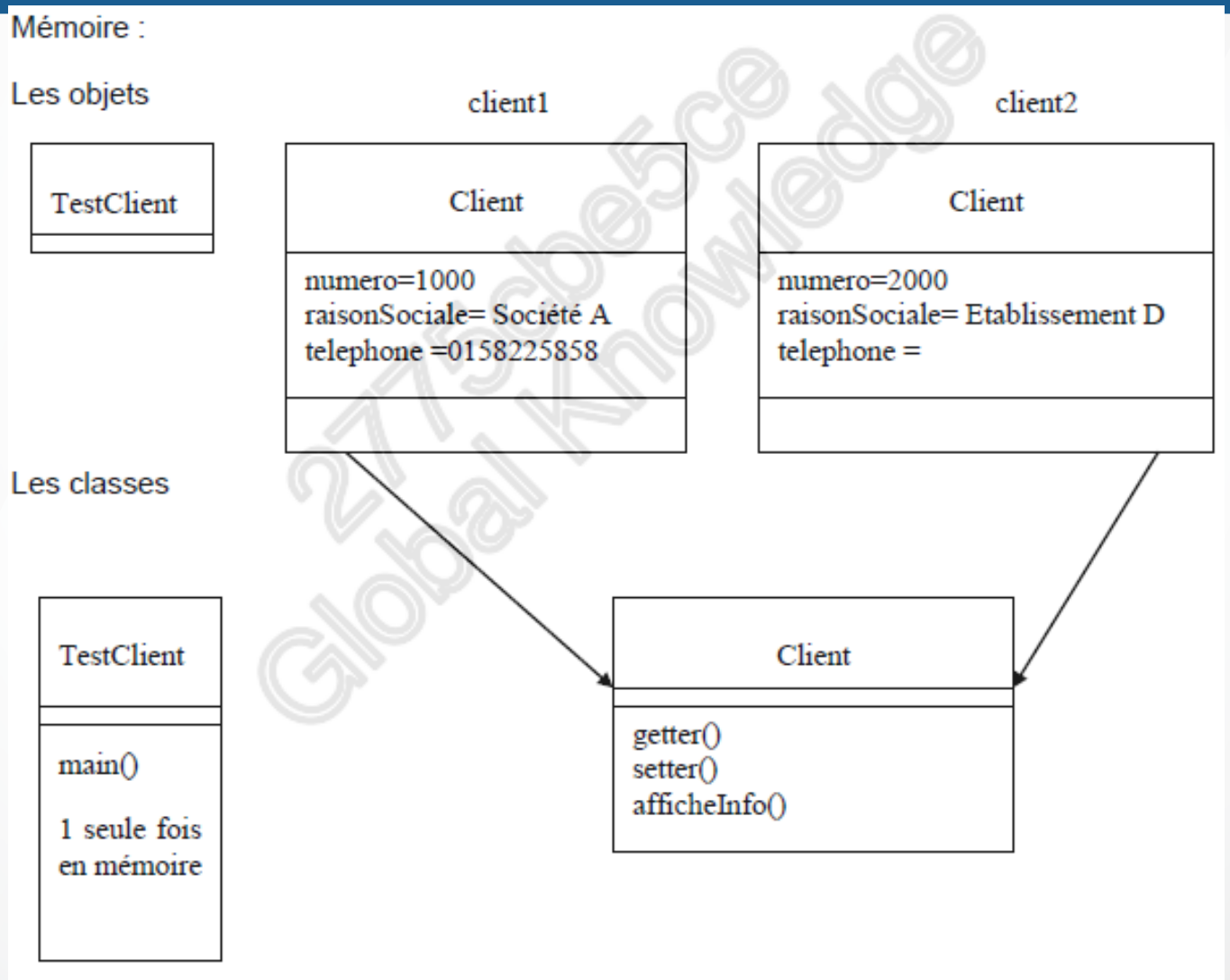
EXERCICE 1 : Definition Produit

- Faire le 1.3 : Code Java Pour la Classe TestProduit

ENCAPSULATION :REPRESENTATION MEMOIRE

- A l'exécution du client test, un objet de type ClientTest a été créée par la machine virtuelle. La méthode *main* de cet objet n'est présente qu'une seule fois en mémoire car elle est déclarée en *static*.
- Par contre ClientTest lance la création de 2 objets client1 et client2, les variables d'instances de ces objets sont présents en mémoire pour chacune des instances, les méthodes sont chargées à partir de la définition de la classe.

ENCAPSULATION :REPRESENTATION MEMOIRE



ENCAPSULATION :REPRESENTATION MEMOIRE D'UNE VARIABLE DE CLASSE

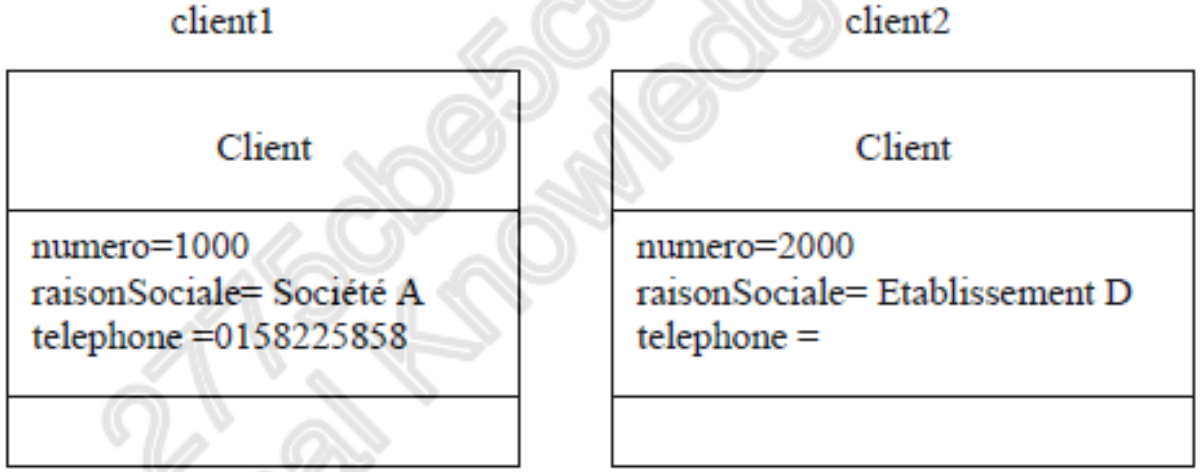
- Dans la définition du client on ajoute une variable de classe :
- Par exemple, un compteur permettant de connaître le nombre d'objets client créé.

```
package formation.gestionClient ;
public class Client
{
  // Déclaration des variables d'instance
  private int numero ;
  private String raisonSociale ;
  private String telephone ;
  private static int cpt ;
  // Constructeur : permet d'initialiser les variables d'instance
  public Client (int num)
  {
    numero = num ;
    cpt = cpt + 1 ; }
}
```

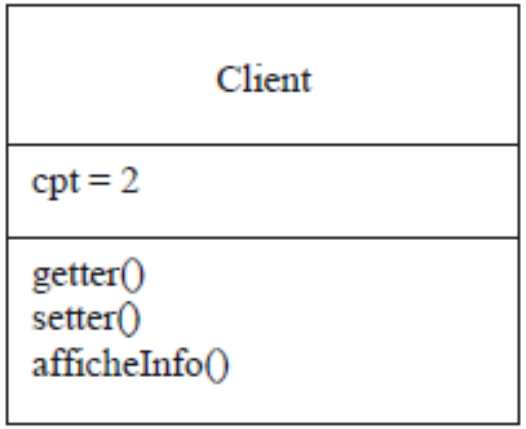
ENCAPSULATION :REPRESENTATION MEMOIRE D'UNE VARIABLE DE CLASSE

Mémoire :

Objet



Classe



SOMMAIRE

- **Chapitre 1** : LES CONCEPTS OBJET
- **Chapitre 2 :** DU CONCEPTS AU LANGAGE
- **Chapitre 3** : **LIENS ENTRE LES CLASSES**

LIENS ENTRE LES CLASSES

LES LIENS ENTRE CLASSES

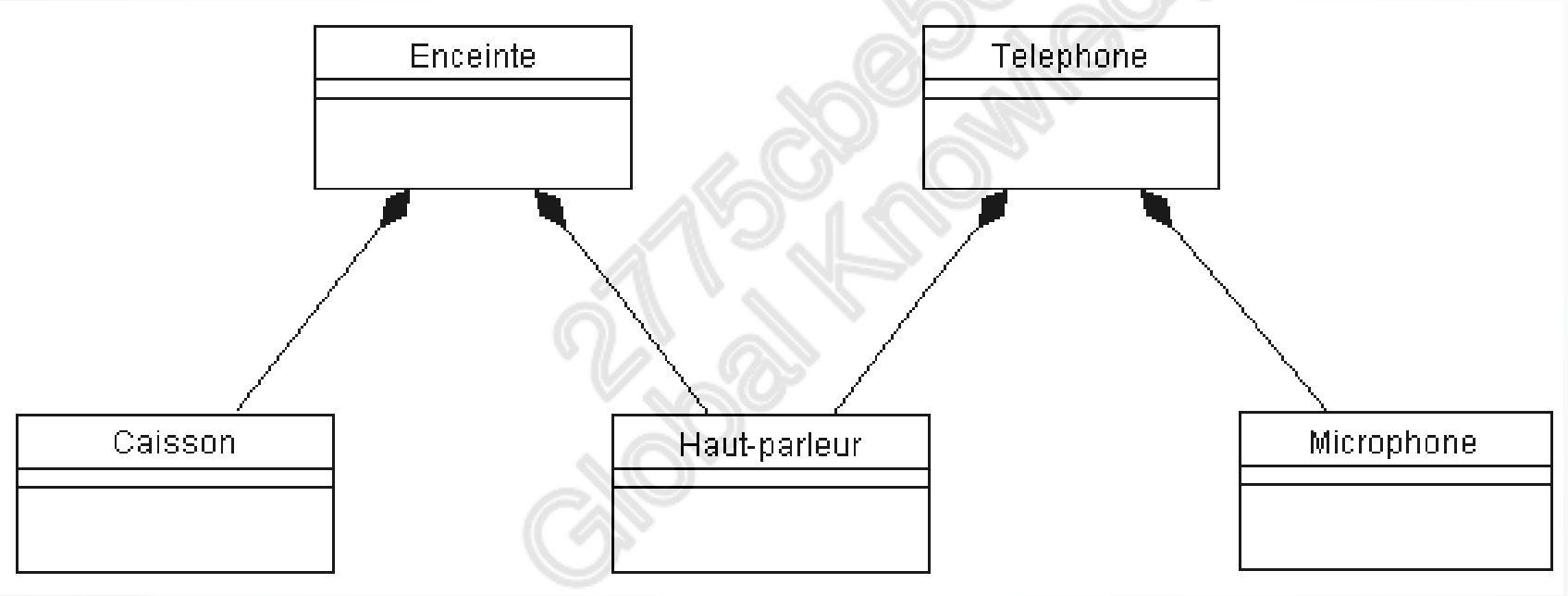
- Un des apports importants de l'orienté objets est la possibilité de définir de nouveaux types de liens entre les classes.
- En plus du lien d'association, déjà présent dans les méthodes d'analyse et de représentation systémique du type de MERISE, ces nouveaux liens sont de deux types :
  - la **généralisation** et la **spécialisation,** qui introduisent la notion de hiérarchie entre les classes,
  - l'**agrégation** et la **composition**, qui permettent de définir une classe comme étant composée de classes plus élémentaires.

AGREGATION ET COMPOSITION : DEFINITION

- L'**agrégation** comme la **composition,** est une forme d'association forte dans laquelle un objet est composé d'objets.
- L'agrégat est un objet composé logiquement d'objets.
- Le composé est physiquement constitué d'objets.

AGREGATION ET COMPOSITION : DEFINITION

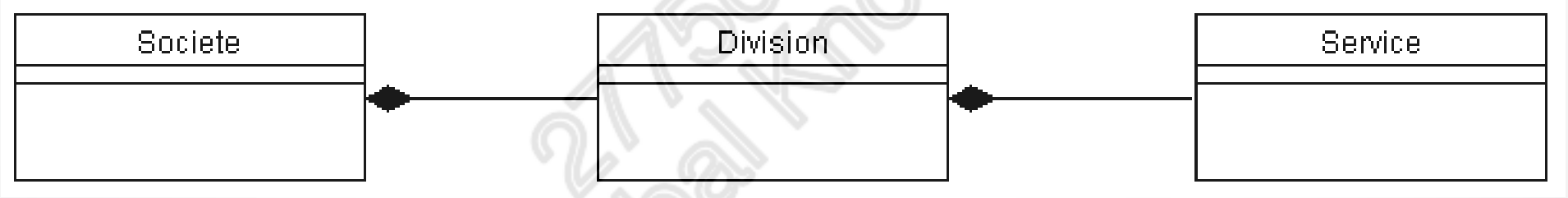
- Exemple :



- Le principal intérêt de ce type de lien est l'élimination de la redondance

AGREGATION ET COMPOSITION : DEFINITION

- Un lien de composition peut être établi entre seulement deux classes :
  - ***Composition d'une société***



- Tout lien de composition doit pouvoir se traduire par la sémantique : **"Est constitué de"**

AGREGATION ET COMPOSITION : DIFFERENCE

Composition

- En UML, le losange représentant le lien de composition est **noir**. Cela permet de traduire que :
  - L'objet agrégé n'existe pas sans l'objet qui l'inclut.
  - La relation est de **1** à 0 ou plusieurs.

- Exemple : un répertoire est constitué de plusieurs fichiers.
  - Le fichier n'appartient qu'à **1** seul répertoire.
  - Si on supprime le répertoire, on supprime tous les fichiers.

AGREGATION ET COMPOSITION : DIFFERENCE

Agrégation

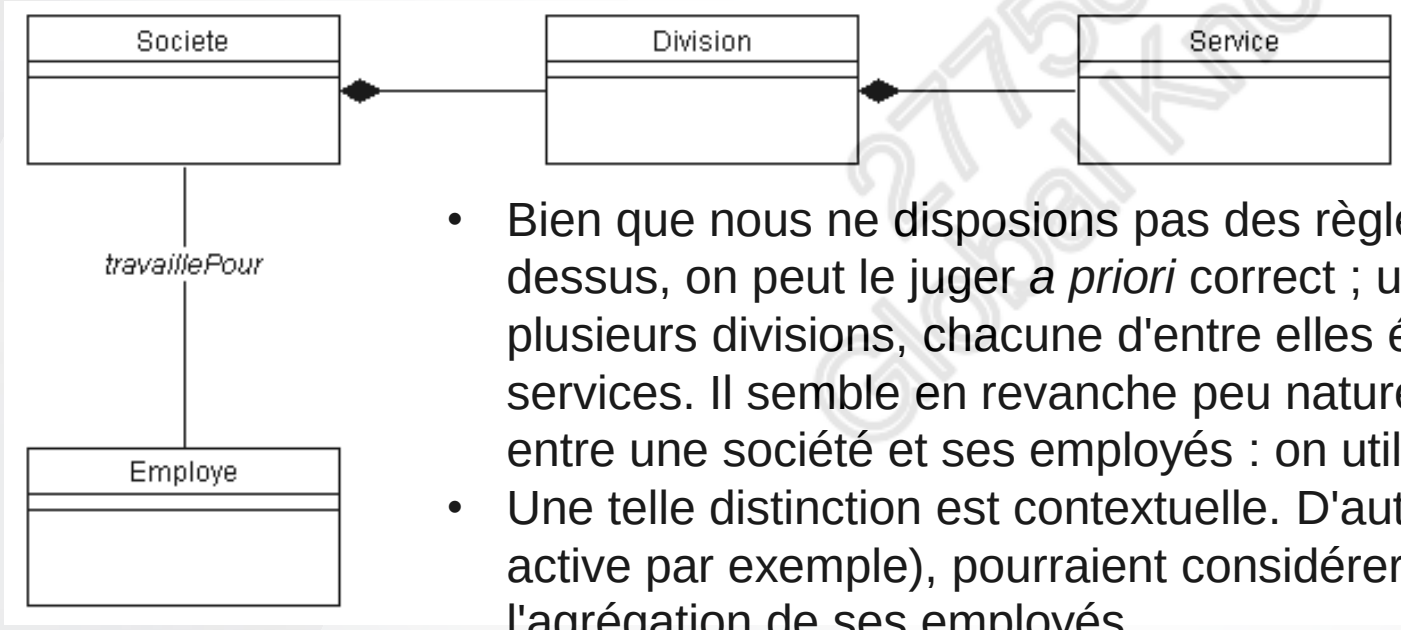
- En UML, le losange représentant le lien d'agrégation est **blanc**. Cela permet de traduire que :
  - L'objet agrégé peut exister indépendamment de l'objet qui l'inclut.
  - La relation est de 0 ou 1 à plusieurs.

- Exemple : une pièce est composée de 0 à plusieurs murs.
  - Le mur mitoyen n'appartient pas qu'à une pièce.
  - Si on supprime la pièce, on ne supprime pas les murs.

- Tout lien d'agrégation doit pouvoir se traduire par la sémantique : **"Est composé de"**

AGREGATION ET ASSOCIATION : DIFFERENCE

- Le lien d'agrégation est reconnu par les AGL (Atelier de Génie Logiciel) pour la génération de code.
- ***Exemple : Agrégation et association pour une société***



- Bien que nous ne disposions pas des règles de gestion ayant conduit au modèle ci-dessus, on peut le juger *a priori* correct ; une société est le résultat de l'agrégation de plusieurs divisions, chacune d'entre elles étant le résultat de l'agrégation de plusieurs services. Il semble en revanche peu naturel de tenter d'établir le même type de lien entre une société et ses employés : on utilisera donc ici une association ordinaire.
- Une telle distinction est contextuelle. D'autres modèles (statistiques sur la population active par exemple), pourraient considérer une société comme n'étant que le résultat de l'agrégation de ses employés.

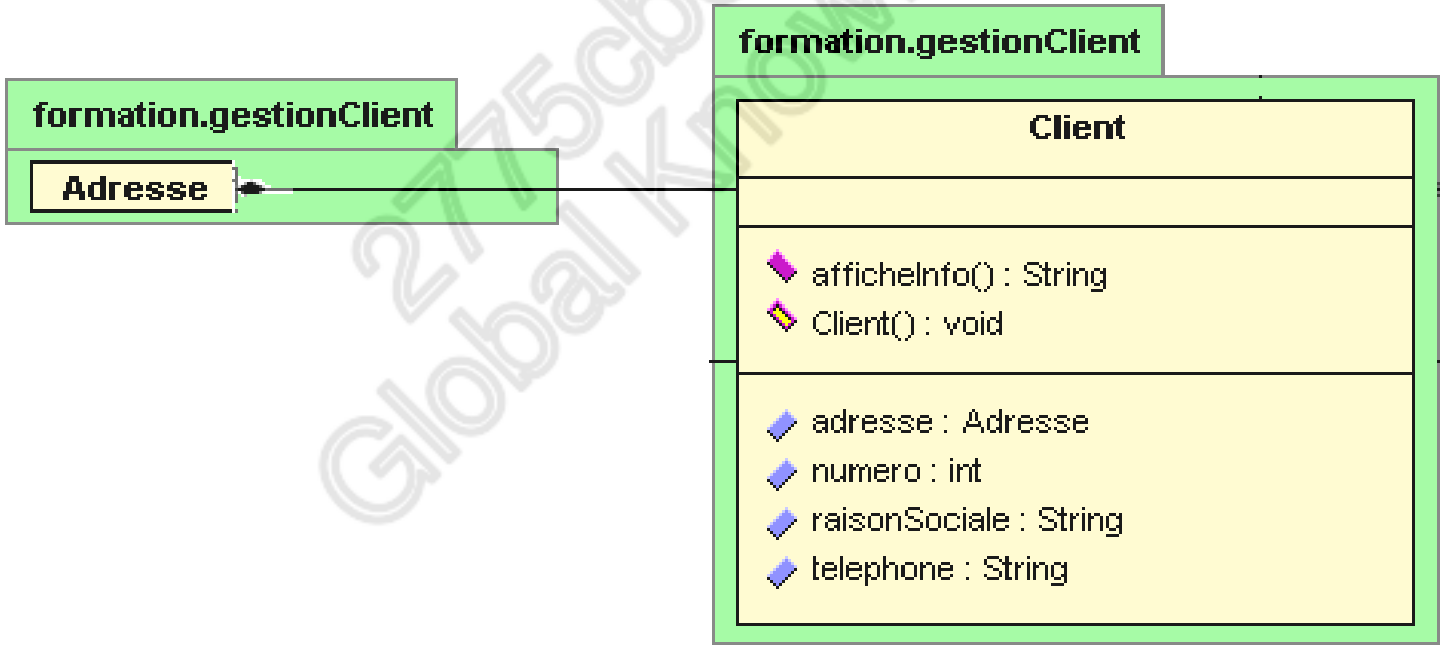
DU CONCEPT AGREGATION / COMPOSITION AU CODE

EXPLICATIONS

- Dans l'exemple Client précédent, on décide de mémoriser l'adresse du client. On peut donc créer des variables d'instance supplémentaires pour la rue, la ville, le code postal…
- On préfère créer une classe Adresse définissant l'ensemble des informations d'une adresse.
- Ainsi notre classe Client sera composée d'une adresse.

DU CONCEPT AGREGATION / COMPOSITION AU CODE

DIAGRAMME UML



DU CONCEPT AGREGATION / CODE JA

```
package formation.gestionClient ;
public class Client {
  // Déclaration des variables d'instance
  private int numero ;
  private String raisonSociale ;
  private String telephone ;
  private Adresse adresse;
  // Constructeur : permet d'initialiser les variables d'instance
  public Client (int num) {
    numero = num ;
    adresse = new Adresse() ;
  }
  //Accesseurs
  public Adresse getAdresse () {
    return adresse;
  }
  public void setAdresse(Adresse adr) {
    adresse = adr;
  }
  …/… (suite du code précédemment écrit)
}
```

CODE JAVA DE LA CLASSE ADRESSE

```
package formation.gestionClient;
public class Adresse
{
  private int numero ;
  private String rue;
  private String ville;
  private int codePostal;
  // Constructeur
  public Adresse(){ }
  //Accesseurs
  public int getCodePostal() {
    return codePostal;
  }
  public int getNumero() {
    return numero;
  }
  public String getRue() {
    return rue;
  }
  public String getVille() {
    return ville;
  }
  public void setCodePostal(int code) {
    codePostal = code;
  }
  public void setNumero(int num) {
    numero = num;
  }
  public void setRue(String ru) {
    rue = ru;
  }
  public void setVille(String vil) {
    ville = vil;
  }
}
```

REFERENCE A L'OBJET

- Dans la classe Adresse, nous créons une méthode qui permet d'afficher l'adresse, sous forme de chaîne de caractères.
- Il faut donc utiliser l'ensemble des méthodes *get*.
- Cependant, dans la définition de la méthode, l'objet de type Adresse n'existe pas.
  - On utilise donc un pseudo code permettant de référencer l'objet en cours. Ce pseudo code est ***this***.

REFERENCE A L'OBJET

```
package formation.gestionClient;
public class Adresse
{
  private int numero;
  private String rue;
  private String ville;
  private int codePostal;
  // Constructeur
  public Adresse() {}

  public String afficheAdresse() {
    return this.getNumero() + " " + this.getRue()+ " " + this.getCodePostal()+
                              " " + this.getVille();
  }
  // Suite des méthodes get et set
  ….
}
```

REFERENCE A L'OBJET

- Par ailleurs, lorsque l'on écrit les setters, on passe en paramètre une variable dont le nom est identique à la variable d'instance.
- Etant donné que l'on ne peut pas écrire :

```
public void setRue(String rue) {
    rue = rue;
}
```

- On fait référence à la variable d'instance de l'objet en cours :

```
public void setRue(String rue) {
    this.rue = rue;
}
```

REFERENCE A L'OBJET

- La classe Adresse

```
package formation.gestionClie
public class Adresse {
  private int numero ;
  private String rue;
  private String ville;
  private int codePostal;
  // Constructeur
  public Adresse() {}
  // Affichage de l'adresse s
  public String afficheAdresse
    return this.getNumero() +
                            +
  }
  // Définition des getters
  public int getCodePostal()
    return codePostal;   }
  public int getNumero() {
    return numero;   }
  public String getRue() {
    return rue;
  }
  public String getVille() {
    return ville;
  }
  // Définition des setters
  public void setCodePostal(int codePostal) {
    this.codePostal = codePostal;
  }
  public void setNumero(int numero) {
    this.numero = numero;
  }
  public void setRue(String rue) {
    this.rue = rue;
  }
  public void setVille(String ville) {
    this.ville = ville;
  }
}
```

CODE JAVA DU TRAITEMENT MANIPULANT LA CLASSE CLIENT

- Le client Test manipule donc les nouvelles méthodes :

```
package formation.gestionClient;
public class TestClient
{
  // Déclaration
  TestClient() {
  // Programme p
  public static
  {
    // Création
    Client clien
    // Modificat
    client2.setR
    // Affichage
    System.out.p
    // Récupération de l'adresse du client
    Adresse adr = client2.getAdresse();
    // Mise à jour des valeurs
    adr.setNumero(100);
    adr.setRue("rue de la justice");
    adr.setCodePostal(91230);
    adr.setVille("Montgeron");
    // Affichage de l'adresse
    System.out.println("Adresse du client " + adr.afficheAdresse());
  }
}
```

Résultat obtenu à la console système :

```
AFFICHAGE le client est : 2000  Etablissement D
Adresse du client 100 rue de la justice 91230 Montgeron
```

EXERCICE 2 : **AGREGATION OU COMPOSITION**

LA GENERALISATION, LA SPECIALISATION

- Le processus classique de modélisation des données et des traitements conduit à introduire des redondances volontairement ou involontairement.
- Ainsi pour un assureur, modéliser un contrat d'assurance automobile et un contrat d'assurance habitation induira une double définition du numéro de contrat, du nom de l'assuré, de son adresse, des informations concernant son état civil...
- De même un banquier, pour qui un compte chèque de particulier et un compte épargne logement sont deux entités de gestion différentes, sera vraisemblablement amené à introduire des redondances dans les traitements de mouvement sur les comptes, de consultation du solde,...

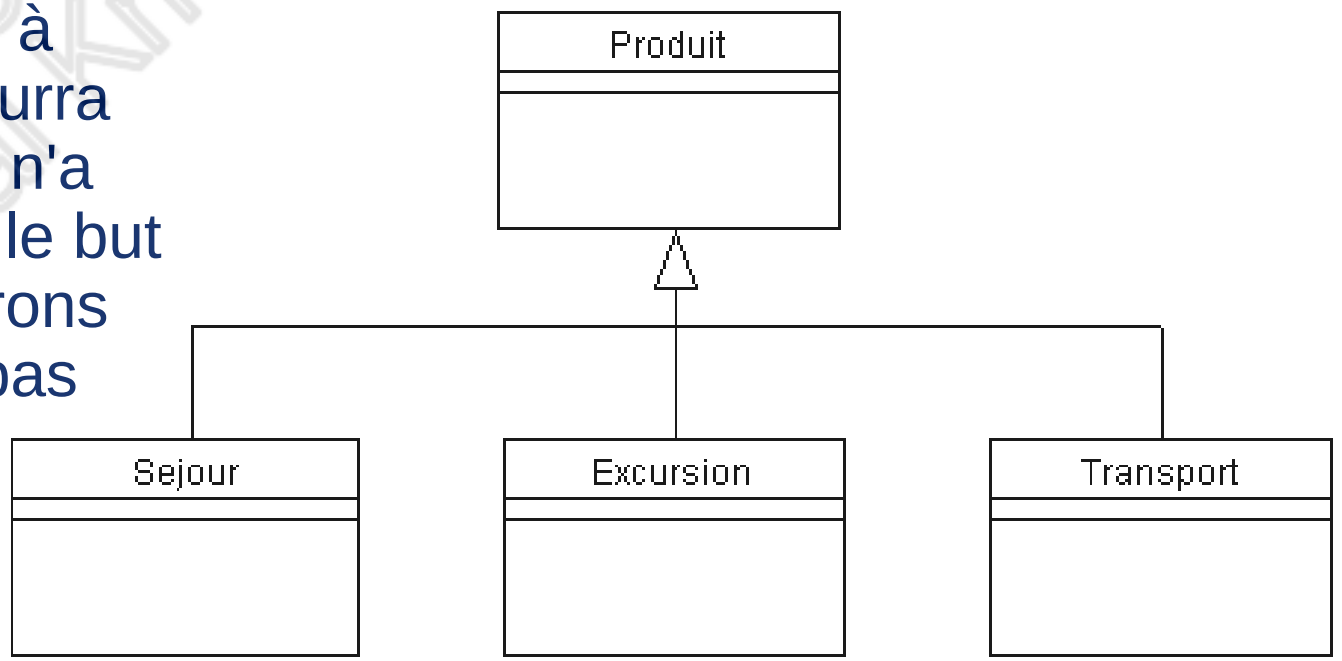
LA GENERALISATION, LA SPECIALISATION

- Une telle redondance peut paraître anodine au moment du développement des applicatifs et se révèler particulièrement coûteuse lors d'une maintenance. Quelle est la répercussion d'une modification des règles de gestion des mouvements de crédit de comptes bancaires pour le banquier précédent ?
- On peut donc admettre que deux classes de gestion, dont la distinction est justifiée, peuvent néanmoins présenter un certain nombre de similitudes, qu'il peut être intéressant de factoriser afin d'éliminer tout risque de redondance et de faciliter les maintenances ultérieures.
- Ce processus est connu sous le nom de **généralisation** et se traduit par l'établissement d'un lien hiérarchique entre une classe, dénommée **sur-classe** ou **classe-mère**, et sa **sous-classe** ou **classe-fille**.

LA GENERALISATION, LA SPECIALISATION

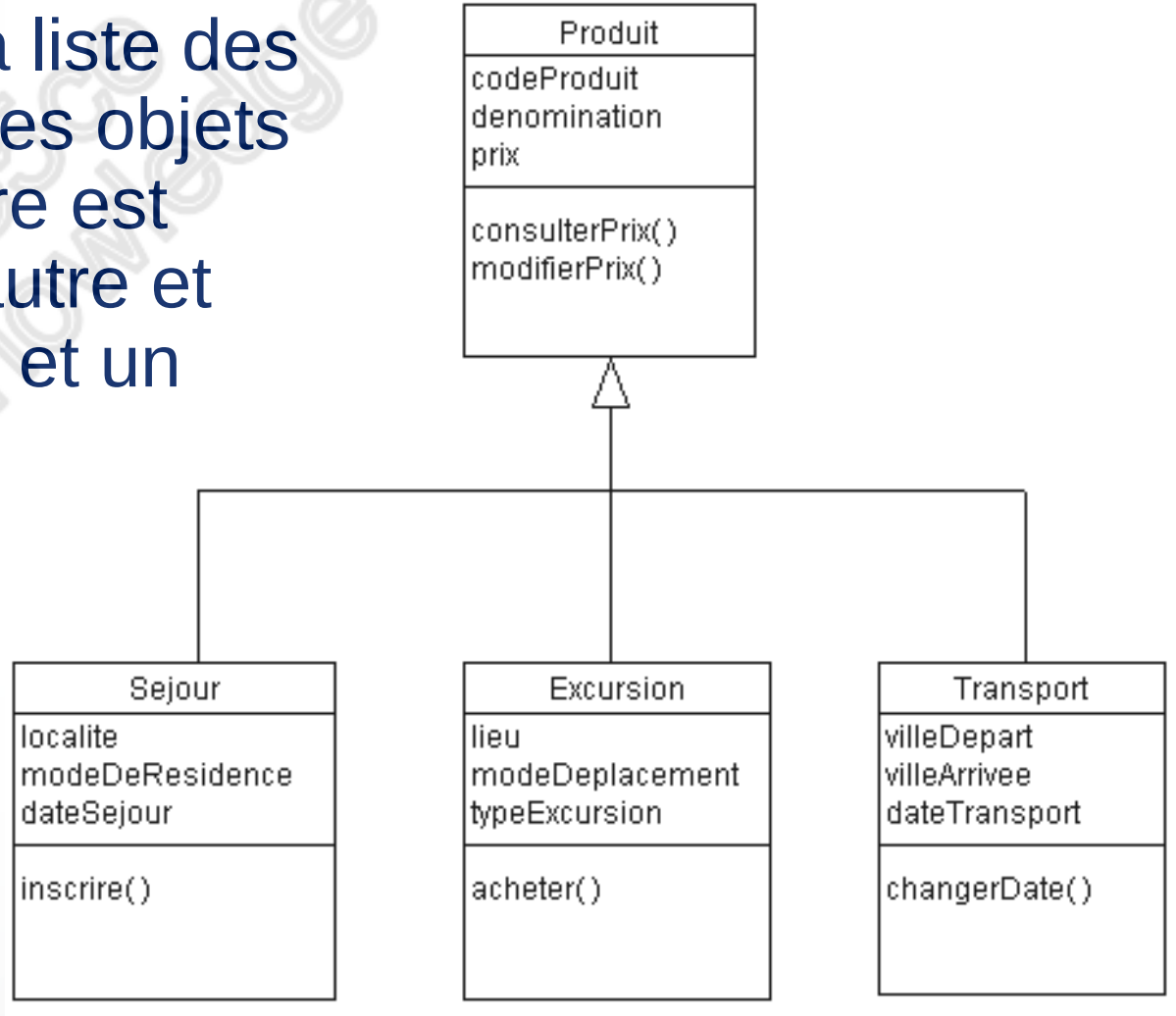
Exemple :

- **Produits commercialisés par une agence de voyage :**
  - Cet exemple introduit la notion de **classe abstraite**, la classe *Produit*, à partir de laquelle aucun objet ne pourra être instancié. Une classe abstraite n'a donc qu'un rôle de fédérateur dans le but d'éliminer la redondance. Nous verrons plus loin qu'une classe-mère n'est pas forcément une classe abstraite.



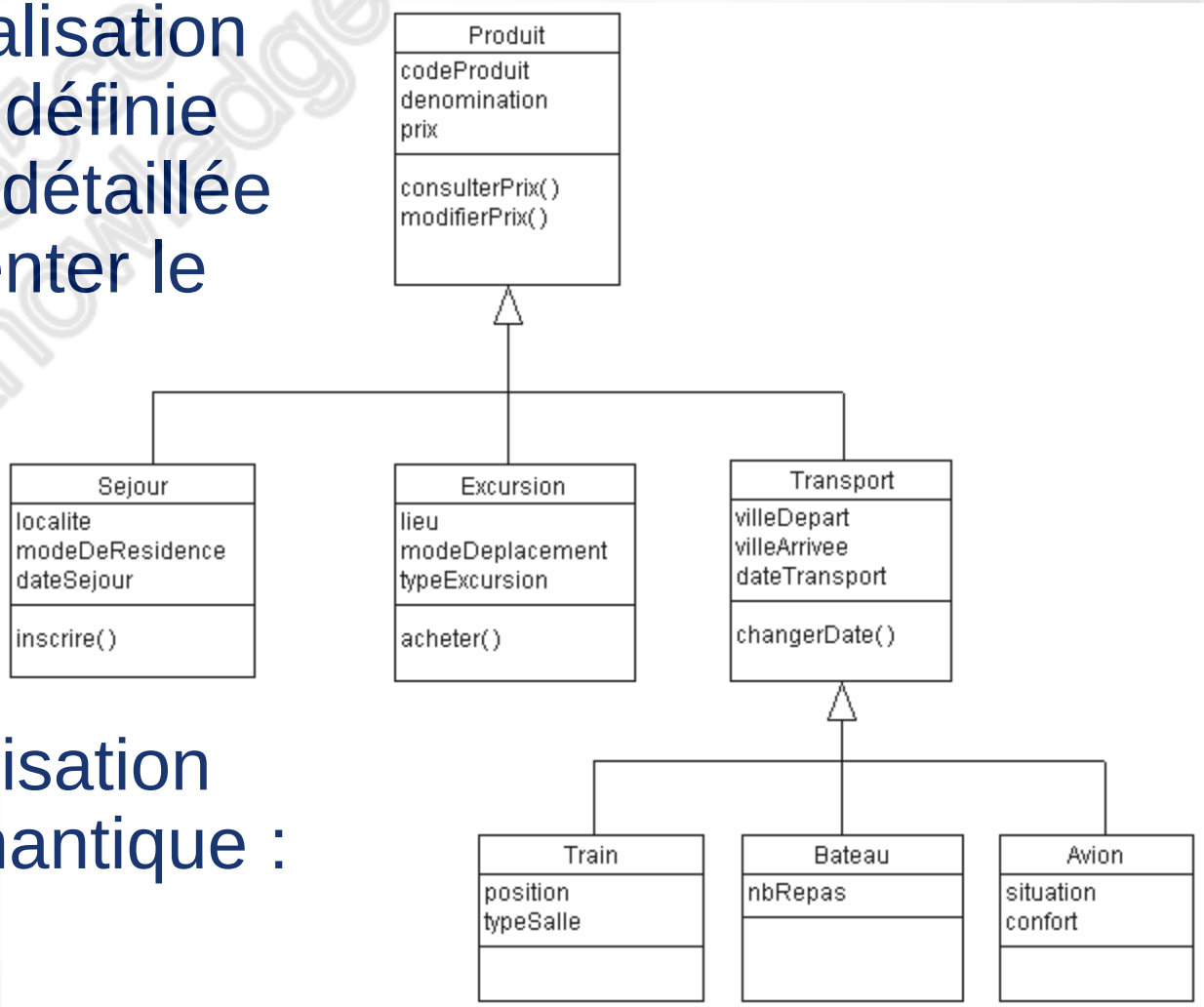
LA GENERALISATION, LA SPECIALISATION

- La classe a pour but de déclarer la liste des attributs et la liste des méthodes des objets qu'elle caractérise.Une classe-mère est une classe au même titre qu'une autre et doit donc elle aussi décrire un état et un comportement.
- Exemple :



LA GENERALISATION, LA SPECIALISATION

- Le processus inverse de la généralisation est la **spécialisation**. Une classe définie de manière trop globale peut être détaillée si les règles de gestion à implémenter le justifient.
- ***Exemple :***



- Tout lien de généralisation/spécialisation doit pouvoir se traduire par la sémantique : **"Est une sorte de"**

LA GENERALISATION, LA SPECIALISATION

- Dans l'exemple précédent, même si les attributs déclarés au niveau de la classe *Séjour* sont uniquement *Localité*, *Mode de résidence* et *Dates*, tout objet instancié à partir de cette classe peut accéder aux attributs *Code*, *Dénomination*, *Prix* (de la classe mère ou sur-classe), *Localité*, *Mode de résidence* et *Dates (de la classe fille)*.
- Autrement dit, **l'état d'un objet est décrit à partir de la valeur des attributs de la classe à partir de laquelle il a été instancié et de toutes ses sur-classes**.

LA GENERALISATION, LA SPECIALISATION

- Ainsi l'état d'un objet instancié à partir de la classe *Bateau* est décrit par les valeurs des attributs *Code*, *Dénomination*, *Prix*, *Ville départ*, *Ville arrivée*, *Date*, *Nb de repas*.
- Il en va de même pour les méthodes. Un objet instancié à partir de la classe *Excursion* peut accéder aux méthodes *Consulter prix()*, *Modifier prix()*, et *Acheter()*.
- Autrement dit, **le comportement d'un objet est constitué des comportements de la classe à partir de laquelle il a été instancié ainsi que de ceux de toutes ses sur-classes**.

- Cette capacité d'une classe à bénéficier de la description d'une classe-mère se nomme l'**héritage**.

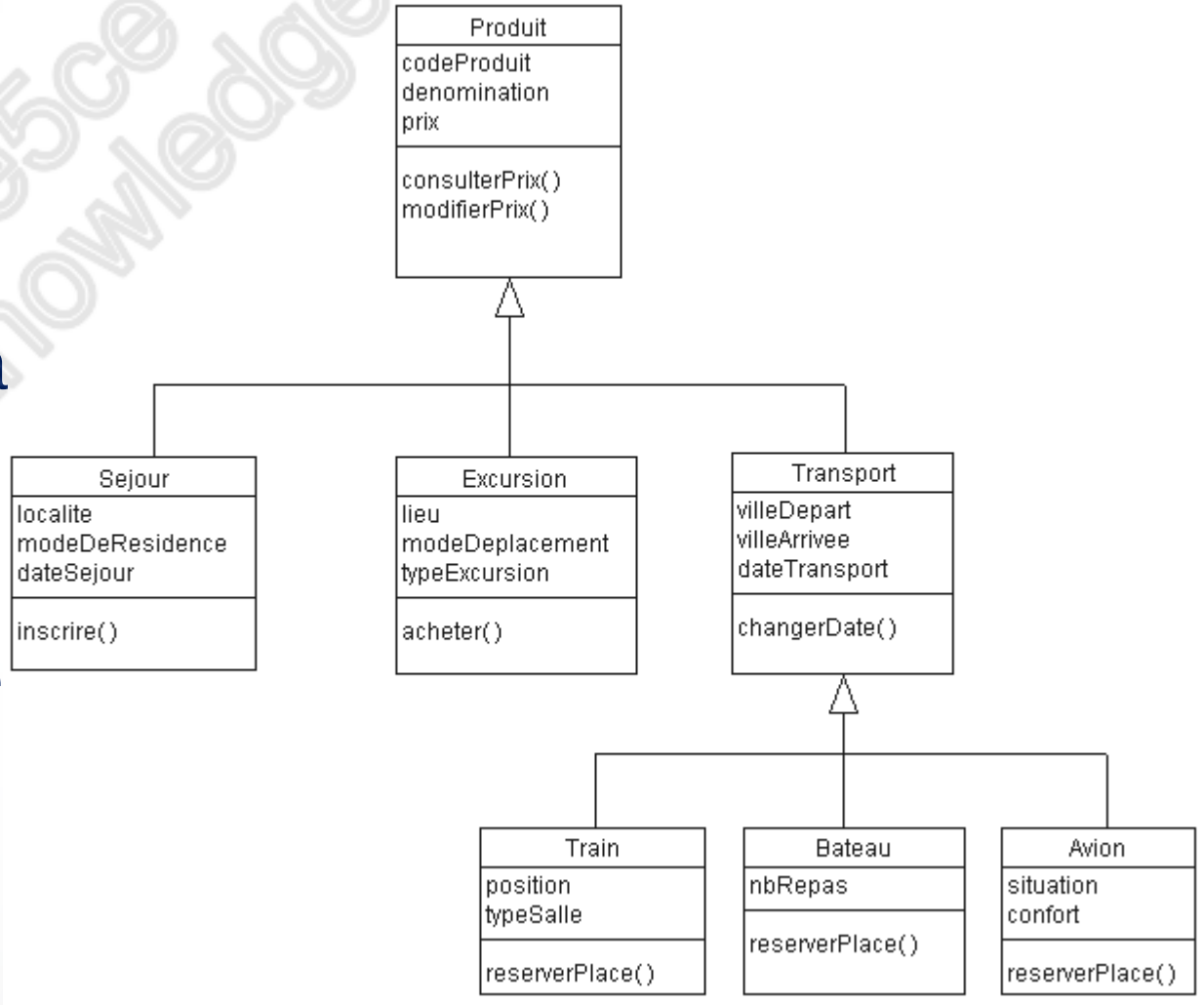
LE POLYMORPHISME

- Le modèle de l'exemple précédent ne permet pas de réserver une place dans un moyen de transport, la méthode adéquate n'ayant pas été déclarée.
- Coder la méthode *Réserver place()* dans la classe *Transport* n'est pas satisfaisant puisque celle-ci n'a pas accès aux attributs spécialisés tels que *Couloir/fenêtre*, *Nb de repas,*... De plus, réserver une place dans un train suppose de se mettre en relation avec la SNCF alors que la réservation d'un billet d'avion peut d'abord nécessiter l'entrée dans une application de choix de la compagnie aérienne.
- La méthode *Réserver place()*, même si elle désigne globalement une réalité similaire dans les trois cas de l'exemple, recouvre en pratique trois types d'action (et donc de codage) complètement distincts.

LE POLYMORPHISME

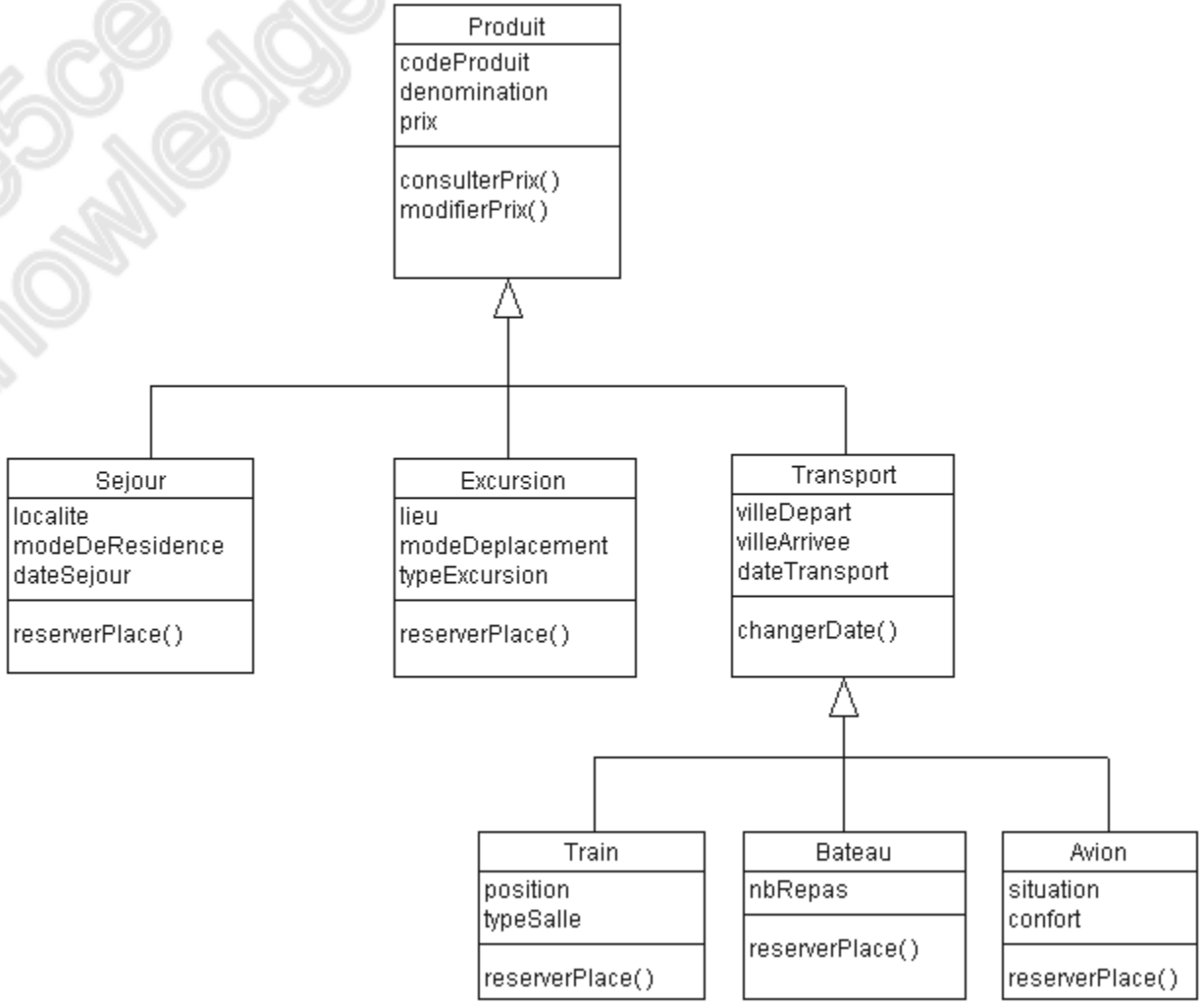
- La solution retenue est donc celle-ci :

- Cette possibilité est particulièrement puissante. Lorsque le développeur d'applications aura besoin d'activer la méthode qui réserve une place dans un transport, il pourra ignorer totalement le type de l'objet pour lequel il active la méthode, le routage étant résolu par l'implémentation



LE POLYMORPHISME

- On appelle **polymorphisme**, la caractéristique d'une méthode à recouvrir des réalités différentes de manière transparente pour l'utilisateur des classes.
- Le modèle précédent peut d'ailleurs être rendu encore plus efficace en rebaptisant la méthode *Inscrire()* de la classe *Séjour* et la méthode *Acheter()* de la classe *Excursion*



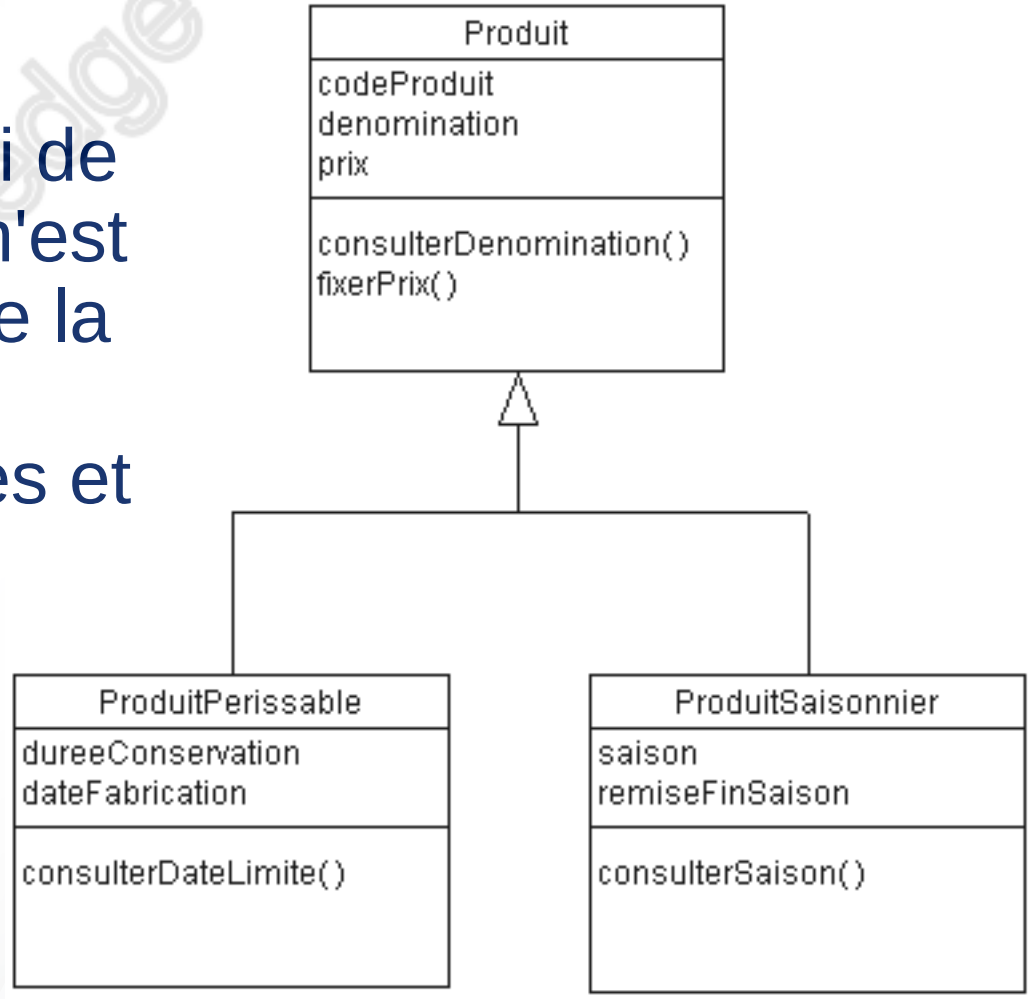
LA REDEFINITION

- ***Exemple : Grossiste multi-domaines***
- Un grossiste gère trois types de produits :
  - des produits périssables (produits laitiers, viandes...),
  - des produits saisonniers (bonnets, écharpes, parapluies...),
  - des produits ni périssables, ni saisonniers (fournitures de bureau...).
- L'étude des règles de gestion permet de modéliser les classes *Produit saisonnier* et *Produit périssable* comme étant sous-classes de la classe *Produit* :

LA REDEFINITION

Remarque :

- Dans cet exemple, contrairement à celui de l'agence de voyages, la classe *Produit* n'est pas une classe abstraite. Elle représente la quasi-totalité des produits gérés par le grossiste, hormis les produits périssables et les produits saisonniers

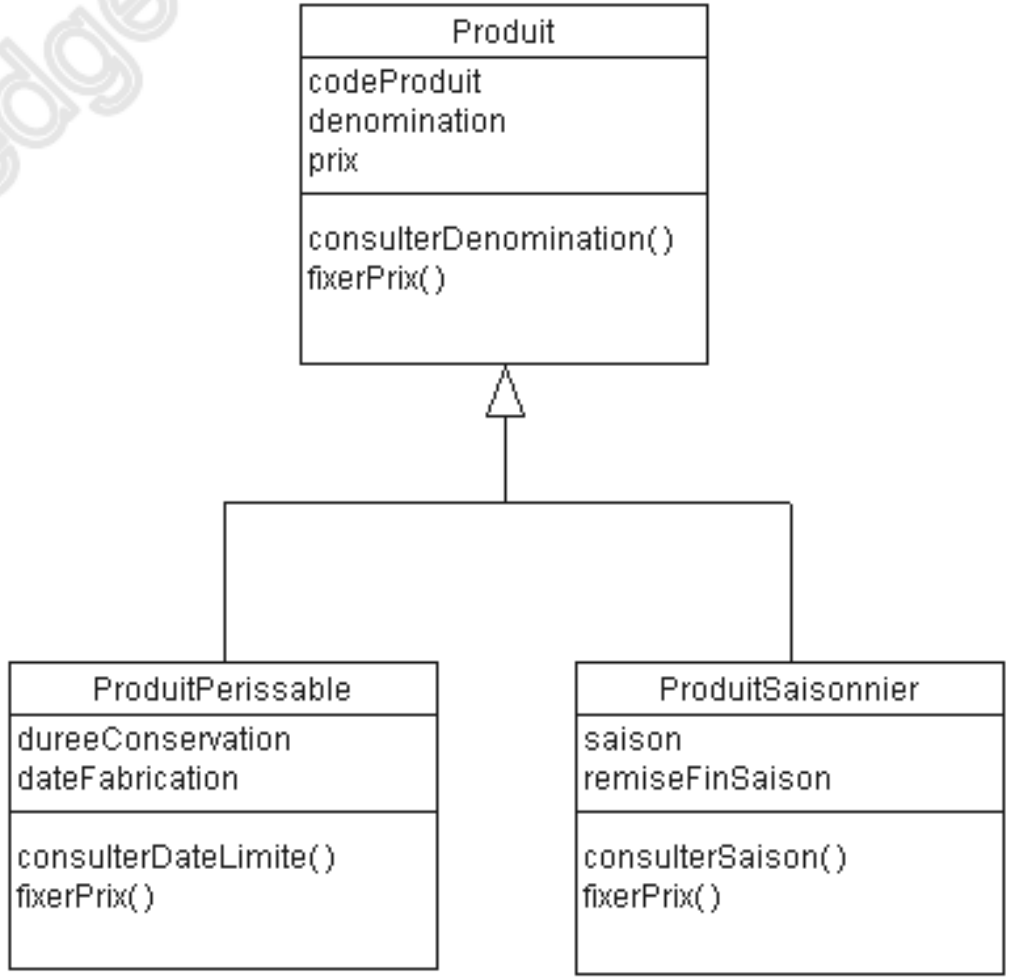


LA REDEFINITION

- Le grossiste peut souhaiter affiner les règles de gestion régissant le prix d'un produit en effectuant une remise sur les produits périssables qui atteignent leur date limite de vente et en soldant les produits saisonniers proportionnellement à l'approche de la fin de saison.
- Une fois encore, c'est le polymorphisme qui va permettre de répondre à ce nouveau besoin, la méthode *Fixer prix()* ayant des comportements différents en fonction de l'objet sur lequel elle s'applique, et ce de manière complètement transparente pour le développeur d'applications :

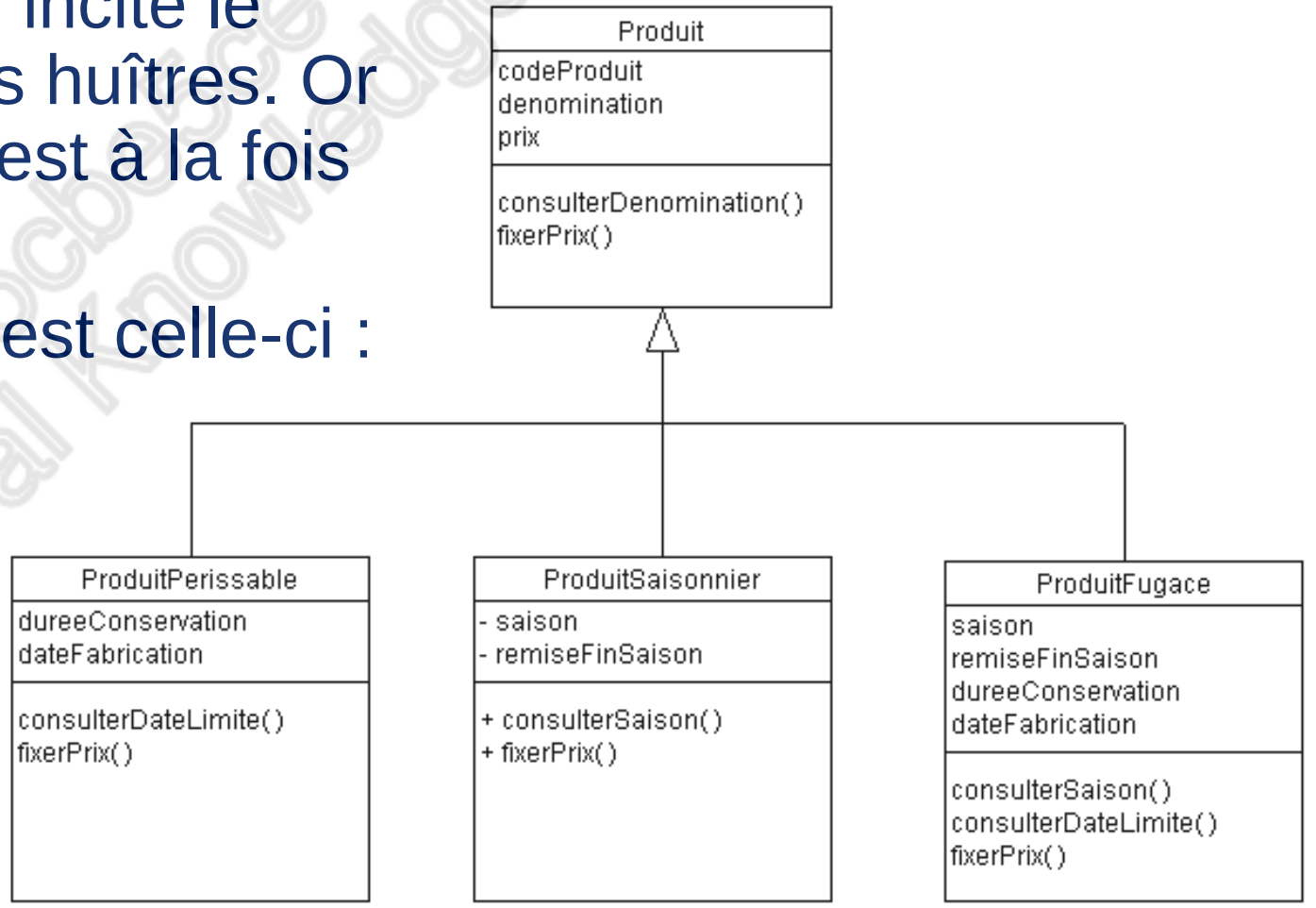
LA REDEFINITION

- Le mécanisme d'héritage s'effectue naturellement de haut en bas. La classe *Produit périssable* hérite des attributs *Code*, *Dénomination*, *Prix* et déclare les attributs *Durée conservation* et *Date fabrication*. De même, elle hérite des méthodes *Consulter dénom.()* et *Fixer prix()* de la classe *Produit*, mais cette dernière est annulée et remplacée par la méthode de même nom déclarée dans la classe même.
- On dit qu'il y a **redéfinition** d'une méthode lorsqu'une méthode héritée d'une classe mère est annulée et remplacée par une autre méthode d'une sous-classe.



L'HERITAGE MULTIPLE

- Une opportunité économique incite le grossiste à se lancer dans les huîtres. Or une huître est un produit qui est à la fois saisonnier et périssable...
- L'erreur à ne pas commettre est celle-ci :



Une telle modélisation, même si elle résout le problème de la méthode *Fixer prix()*, génère une redondance inacceptable.

L’HERITAGE MULTIPLE

- Le modèle objet autorise l'**héritage multiple**, capacité dont dispose une sous-classe d'hériter de plusieurs classes mères **de même niveau** :
- La méthode f*ixerPrix()* peut être soit surchargée au niveau de la classe *Produit fugace*, soit choisie statiquement ou dynamiquement entre les méthodes de même nom des classes *Produit périssable* et *Produit saisonnier*.
- Cette ambiguïté sur la méthode fixerPrix() est inaceptable pour certains langages orienté objet, qui vont donc interdire l’héritage multiple !

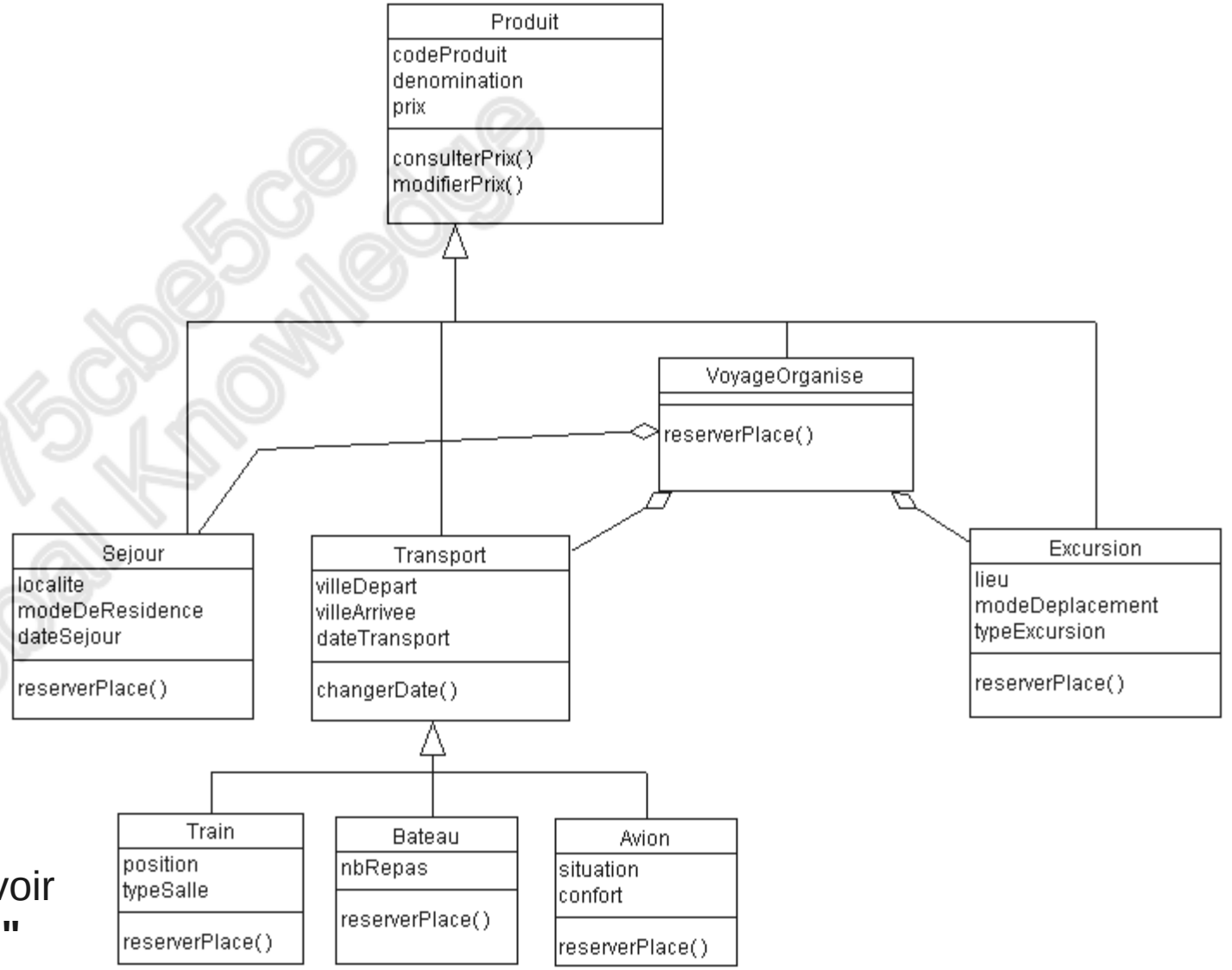
Attention !...

- L’héritage multiple est en fait un lien assez rare, qu’il ne faut pas confondre avec le lien de composition.

RECAPITULATIF DES LIENS

Exemple : Agence de voyage

- En plus de commercialiser des séjours, des excursions et des transports, l'agence de voyage propose à ses clients des voyages organisés (un transport + un séjour + une ou plusieurs excursions) :



Tout lien d'agrégation doit pouvoir se traduire par la sémantique : **"Est composé de"**

Tout lien de généralisation/spécialisation doit pouvoir se traduire par la sémantique : **"Est une sorte de"**

RECAPITULATIF DES LIENS

- Il n'y a pas, à proprement parler, de mécanisme d'héritage dans une relation d'agrégation. Toutefois, la classe *Voyage organisé* étant composée logiquement des classes *Séjour*, *Transport* et *Excursion*, elle inclut tous les attributs et toutes les méthodes de ces classes.
- En cas de conflit, par exemple sur la méthode r*éserver place()* ci-dessus, les traditionnelles règles de masquage des langages de programmation s'appliquent, chacune des méthodes restant toutefois accessible à condition de la préfixer correctement.

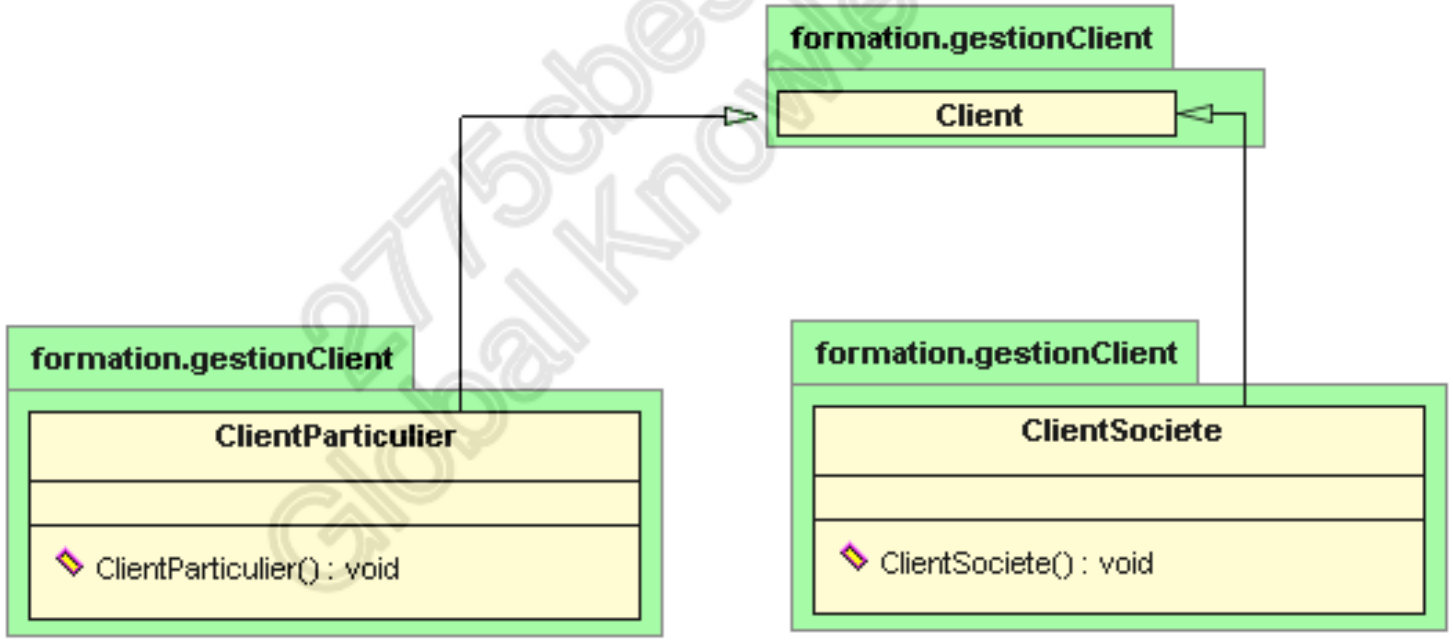
DES CONCEPTS AU CODE ; HERITAGE SIMPLE

EXPLICATIONS

- Dans l'exemple Client précédent, on décide de gérer des clients particuliers, afin de connaître leurs enfants et de gérer des clients « société » pour connaître le nombre d'employés. Ces deux types de clients sont des clients, ils vont donc hériter de la classe Client.

DES CONCEPTS AU CODE ; HERITAGE SIMPLE

DIAGRAMME UML



DES CONCEPTS AU CODE : HERITAGE SIMPLE

CODE JAVA

```
package formation.gestionClient;
public class ClientSociete extends Client
{
  // Définition de la variable d'instance spécialisant l'information
  private int nombreEmployes;
  // Définition des getters et des setters
  public int getNombreEmployes()
  {
     return nombreEmployes;
  }
  public void setNombreEmployes(int nombreEmployes)
  {
     this.nombreEmployes = nombreEmployes;
  }
}
```

DES CONCEPTS AU CODE : REDÉFINITION

EXPLICATIONS :

- On désire redéfinir la méthode « AfficheInfo() » de la classe mère. Dans la classe fille, la méthode affichera le nombre de salariés.

DES CONCEPTS AU CODE : REDÉFINITION

CODES JAVA : Code de la classe ClientSociete

```
package formation.gestionClient;
public class ClientSociete extends Client
{
  private int nombreEmployes;
  // Constructeur
  public ClientSociete(int num)
  {
     // code du constructeur (vo
     …
  }
  public int getNombreEmployes() {
     return nombreEmployes;
  }
  public void setNombreEmployes(int nombreEmployes) {
     this.nombreEmployes = nombreEmployes;
  }
```

```
  // Envoi d'une chaine de caractères affichant
  //le nombre de salariés.
  public String afficheInfo()
  {
     return "Il y a : " +  this.getNombreEmployes()
                          + " employés";
  }
}
```

DES CONCEPTS AU CODE : REDÉFINITION

```
package formation.gestionClient;
public class TestClientSociete
{
  public TestClientSociete()
  {
  }
  public static void main(String[] args)
  {
    ClientSociete client3 = new ClientSociete(3000);
    client3.setRaisonSociale("Societe Durand");
    client3.setNombreEmployes(30);
    System.out.println(client3.afficheInfo());
  }
}
```

Résultat obtenu à la console système :

```
Il y a : 30 employés
```

DES CONCEPTS AU CODE : REFERENCE A L'OBJET EN COURS

Dans une méthode :

- En fait, on souhaite récupérer le code existant à la définition de la classe mère et ajouter le nombre d'employés. Pour eviter la réécriture de code, on fait référence au code de la classe mère en spécifiant que l'on applique la méthode de la classe mère par le pseudo code « *super* ».

DES CONCEPTS AU CODE : REFERENCE A L'OBJET EN COURS

```
…/…
// Envoi d'une chaine de caractères affichant le nombre de
// salariés en plus des informations
// données dans la classe mère….
public String afficheInfo()
{
   String retour;
   retour = super.afficheInfo();
   return retour + " Il y a : " + this.getNombreEmployes() +
                   " employés";
}
…/…
```

Le code du client test reste inchangé mais lorsque l'on exécute le test on obtient :

```
le client est : 3000  Societe Durand Il y a : 30 employés
```

DES CONCEPTS AU CODE : REFERENCE A L'OBJET EN COURS

Dans le constructeur

- On n'hérite pas des constructeurs, cependant on veut utiliser le constructeur de la classe mère. L'appel d'un constructeur est particulier. On fait référence au constructeur de la classe mère dans le constructeur de la classe fille en utilisant aussi le pseudo code super.

DES CONCEPTS AU CODE : REFERENCE A L'OBJET EN COURS

```
package formation.gestionClient;
public class ClientSociete extends Client
{
  private int nombreEmployes;
  // Constructeur
  public ClientSociete(int num) {
     super(num) ;
  }
  public int getNombreEmployes() {
    return nombreEmployes;
  }
  public void setNombreEmployes(int nombreEmployes) {
    this.nombreEmployes = nombreEmployes;
  }
  // Envoi d'une chaine de caractères affichant le nombre de salariés.
  public String afficheInfo() {
    return "Il y a : " + this.getNombreEmployes() + " employés";
  }
}
```

EXERCICE 3 : **HERITAGE SIMPLE**

DES CONCEPTS AU CODE : POLYMORPHISME

EXPLICATIONS

- La redéfinition est un polymorphisme.
- On enregistre dans un tableau de type Client, des clients, des sociétés, des particuliers.
- On effectue une boucle afin d'afficher les informations du client quelqu'il soit.
- On va donc avoir un polymorphisme, car la méthode afficheInfo() va s'appliquer de manière différente en fonction du type de client.
- Le polymorphisme prend sa puissance quand on parle d'ensemble d'objets, de type indentique (ici le type Client), mais dont les instances sont différentes (ici ClientParticulier, ClientSociete et aussi Client). On ne souhaite plus voir l'objet dans sa définition spécifique mais dans sa définition générale.

DES CONCEPTS AU CODE : POLYMORPHISME

```
package formation.gesti
public class TestDiffer
{
  public TestDifferentC
  public static void ma
    // Création d'un ob
    Client client1 = ne
    // Modification de
    client1.setRaisonSo
    // Création d'un ob
    Client client2 = ne
    // Modification de
    client2.setRaisonSo
    // Création d'un ob
    ClientSociete clien
    client3.setRaisonSo
    client3.setNombreEm
```

```
    // Création d'un objet client particulier
    ClientParticulier client4 = new   ClientParticulier(4000);
    client4.setRaisonSociale("Particulier Durand");
    client4.setNombreEnfants(2);
    // Enregistrement des clients crées dans un tableau
    Client clients[] = new Client[4];
    clients[0] = client1;
    clients[1] = client2;
    clients[2] = client3;
    clients[3] = client4;
    // Polymorphisme
    for (int i = 0; i < clients.length; i++) {
      System.out.println("AFFICHAGE élement " + i + " " +
      clients[i].afficheInfo());
    }
  }
}
```

DES CONCEPTS AU CODE : POLYMORPHISME

- Résultat obtenu à la console système :

```
AFFICHAGE élement 0 le client est : 1000  Client A
AFFICHAGE élement 1 le client est : 2000  Client B
AFFICHAGE élement 2 le client est : 3000  Societe Durand Il y a : 30 employés
AFFICHAGE élement 3 le client est : 4000  Particulier Durand Il a : 2 enfants
```

EXERCICE 4 : **POLYMORPHISME**

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

INTERET

- En Java, l'héritage multiple n'existe pas. Les interfaces sont la solution pour cette représentation.
- Une **interface** est similaire à une **classe abstraite**, c'est donc juste une déclaration de **méthodes abstraites** et de **variables statiques** et **finales** (des constantes), pas une définition.
- Une interface n'est donc pas instanciable.
- Une interface comme une classe définit un type, utilisable ensuite comme un type de données.

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

DECLARATION

- La déclaration s'effectue par le mot clé « **interface** » et l'on indique le nom de l'interface
- Par convention un nom d'interface débute en majuscule, et termine par « able ». Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.
- Devant le mot clé, on peut trouver des modificateurs. Derrière on note l'héritage par le mot clé « **extends** ». Une interface peut hériter de plusieurs interfaces (séparées par des virgules).

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

```
interface Interfacable {
   …
}

// Exemple avec héritage
interface InterfacableA extends Interfacable , InterfacableC {…}

// Exemple avec modificateurs
public interface InterfacableC{…}


Syntaxe
[Modificateurs] interface Nom extends Inteface1, Interface2 {…}
```

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

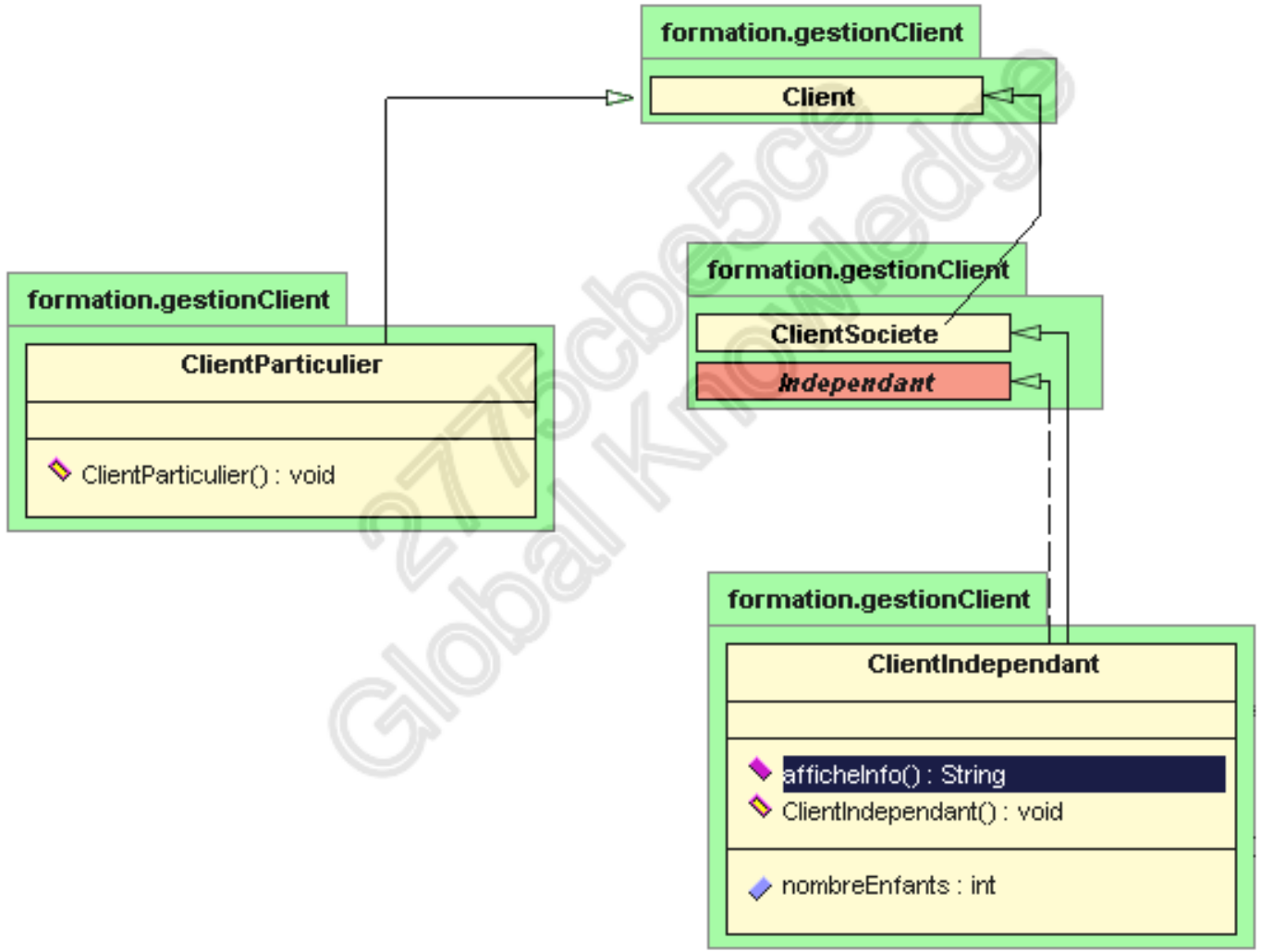
- **MODIFICATEURS :**

- Par défaut une interface n'est visible que pour les classes définies à l'intérieur de son package.
  - public : l'interface est visible pour les classes des autres packages.
  - abstract : l'interface est par définition abstraite.

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

EXEMPLES

- Nous désirons gérer des indépendants, il s'agit de client société et de client particulier. L'héritage multiple n'existant pas en java, on va donc hériter de la classe ClientSociete et se créer une interface Independant que l'on implémentera dans la classe ClientIndependant. Cette interface contient des déclarations de afficheInfo(), getNombreEnfants, setNombreEnfants.



DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

Code java de l'interface

```
package formation.gestionClient;
public interface Independant
{
   public String afficheInfo();
   public int getNombreEnfants();
   public void setNombreEnfants(int nombreEnfants);
}
```

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

UTILISATION D'UNE INTERFACE DANS UNE CLASSE

- Une classe qui implémente une ou plusieurs interfaces s'engage à définir toutes les méthodes déclarées dans les interfaces.
- Une classe qui n'implémente pas l'ensemble des méthodes de l'interface devient abstraite

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

Code java

```
package formation.gestionClient;
public class ClientIndependant extends ClientSociete implements Independant
{
  private int nombreEnfants;
  public ClientIndependant(int num) {super(num); }
  // Envoi d'une chaine de caractères affichant le nombre de salariés.
  public String afficheInfo() {
    String retour;
    retour = super.afficheInfo() + " Affichage info supplementaires " +
    this.getNombreEnfants() ;
    if (this.getNombreEnfants() > 1) {return retour + " enfants";}
    else {return retour + " enfant";} }
    public int getNombreEnfants() { return nombreEnfants;}
    public void setNombreEnfants(int nombreEnfants) {
      this.nombreEnfants = nombreEnfants; }
}
```

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

Code du client test

- Dans notre client test précédent on crée une instance de ClientIndependant, on l'ajoute dans notre tableau de client pour afficher les informations

```
package forma
public class
{
  public Tes
  public sta
  {
    // Créat
    Client c
    // Modif
    client1.
    // Créat
    ClientSo
    client3.
    client3.
```

```
    // Création d'un objet client particulier
    ClientParticulier client4 = new ClientParticulier(4000);
    client4.setRaisonSociale("Particulier Durand");
    client4.setNombreEnfants(2);
    // Création d'un objet client indépendant
    ClientIndependant client5 = new ClientIndependant(5000);
    client5.setRaisonSociale("Indépendant Martin");
    client5.s
    client5.s
    // Enregi
    Client cl
    clients[0
    clients[1
    clients[2
    clients[3
```

```
    // Polymorphisme
    for (int i = 0; i < clients.length; i++)
    {
      System.out.println("AFFICHAGE élement " + i + " " +
                                    clients[i].afficheInfo());
    }
  }
}
```

DES CONCEPTS AU CODE : HERITAGE MULTIPLE : INTERFACES

- Résultat obtenu à la console système

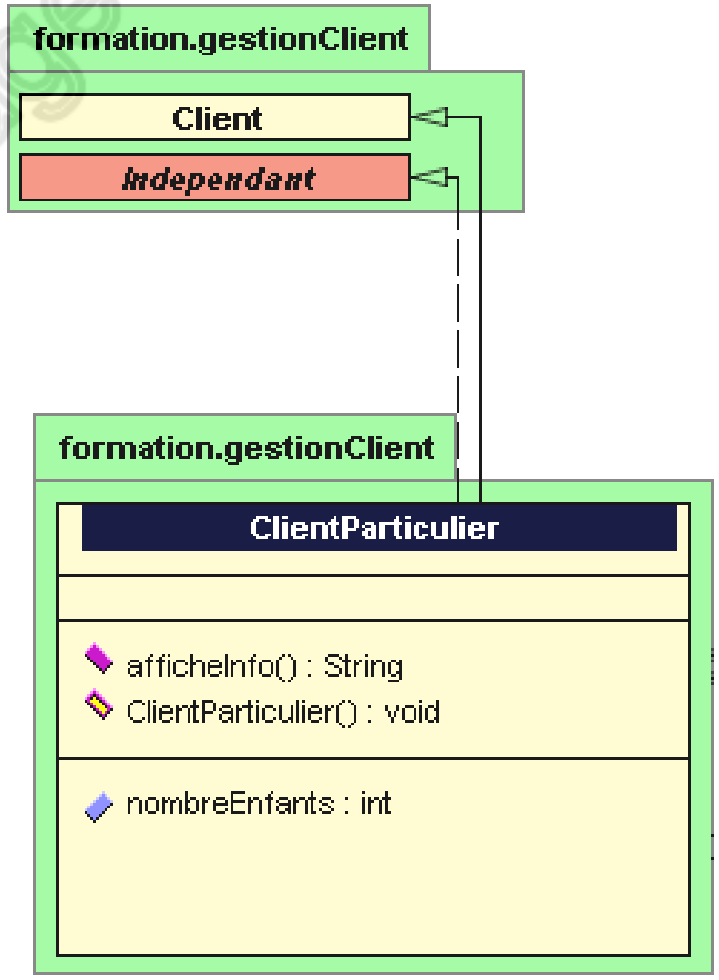
```
AFFICHAGE élement 0 le client est : 1000  Client A
AFFICHAGE élement 1 le client est : 2000  Client B
AFFICHAGE élement 2 le client est : 3000  Societe Durand Il y a : 30 employés
AFFICHAGE élement 3 le client est : 4000  Particulier Durand Il a : 2 enfants
AFFICHAGE élement 4 le client est : 5000  Indépendant Martin Il y a : 1 employé Affichage info supplementaires 0 enfant
```

DES CONCEPTS AU CODE : UTILISATION D'UNE INTERFACE COMME TYPE DE BASE

- Il est possible de déclarer des objets de type interface.
- Ces objets ne sont pas instanciables mais ils référenceront un objet qui implémente l'interface et donc qui comprendra les méthodes.
- Ceci est pratique pour gérer des objets de classes différentes, mais qui réagissent identiquement.
- On modifie donc la classe ClientParticulier afin qu'elle implémente l'interface Independant.

DES CONCEPTS AU CODE : UTILISATION D'UNE INTERFACE COMME TYPE DE BASE

Diagramme UML



D
D

```
package formation.gestionClient;
public class ClientParticulier extends Client implements Independant
{
  private int nombreEnfants;
  public ClientParticulier(int num){
     super(num);
  }
  // Envoi d'une chaine de caractères affichant le nombre d'enfants.
  public String afficheInfo() {
     String retour;
     retour = super.afficheInfo() + " Il a : " + this.getNombreEnfants() ;
     if (this.getNombreEnfants() > 1)
       {return retour + " enfants";}
     Else
       {return retour + " enfant";}
  }
  public int getNombreEnfants() {
    return nombreEnfants;
  }
  public void setNombreEnfants(int nombreEnfants) {
    this.nombreEnfants = nombreEnfants;
  }
}
```

DES CO
D'UNE I

- On crée
objets d

```
package formation.gestionClient;
public class TestEnfants {
  public TestEnfants() {}
  public static void main(String[] args)
  {
      // Création d'un objet client particulier
      ClientParticulier client4 = new ClientParticulier(4000);
      client4.setRaisonSociale("Particulier Durand");
      client4.setNombreEnfants(2);
      // Création d'un objet client indépendant
      ClientIndependant client5 = new ClientIndependant(5000);
      client5.setRaisonSociale("Indépendant Martin");
      client5.setNombreEnfants(0);
      client5.setNombreEmploye(5);
      // Enregistrement des clients crées dans un tableau
      Independant clients[] = new Independant[2];
      clients[0] = client4;
      clients[1] = client5;
      // Polymorphisme
      or (int i = 0; i < clients.length; i++) {
       System.out.println("AFFICHAGE élement " + i + " "
    clients[i].getNombreEnfants());
      }
  }
}
```

Résultat obtenu à la console système

```
AFFICHAGE élement 0 2
AFFICHAGE élement 1 0
```

EXERCICE 5 : **HERITAGE MULTIPLE**

QUESTIONS