Université de Montpellier / Faculté Des Sciences

Modélisation et Programmation par Objets (2e partie)

2019-2020 - HLIN 505

Cours 1
Présentation
Modularité
(classes, classes imbriquées)

Enseignants
Marianne Huchard, Stéphane Bessy, Marie-Laure Mugnier
Clémentine Nebut, Abdelhak-Djamel Seriai

Ce document est un recueil de notes de cours correspondant au premier chapitre du cours. En 2019/2020, il est traité lors d'une séance de TD/TP uniquement.

1 Présentation des concepts développés dans ce module

Ce module vous développera les points suivants :
- Modularité : rappels sur les classes, classes imbriquées (statiques et non statiques)
- Héritage et spécialisation/généralisation : rappels et approfondissements sur le sous-typage et les formes de substituabilité
- Contrats : assertions et exceptions
- Abstraction et modularité : types interfaces, composants en UML, modules (Java 9)
- Eléments de méta-programmation : annotations, introspection
- Généricité
- Manipulation des collections et flux de données : Itérateurs et *streams*
- Diagrammes dynamiques (UML) : séquences, interactions, collaboration, états, activités
- Introduction à la problématique du test de logiciels

Ce cours fait suite à un cours d'initiation à la modélisation et à la programmation par objets qui en a déjà détaillé les grands principes (HLIN406). Pour en dresser à nouveau un tableau à grands traits, rappelons que la modélisation et la programmation par objets cherchent principalement à introduire dans les représentations informatiques (comme des modèles conceptuels ou du code source) une représentation des

concepts du domaine sur lequel porte un logiciel, ou de concepts techniques de l'informatique. Cette approche s'inscrivait à l'origine en rupture avec les méthodes de conception et de développement développées dans les années 70 et antérieures, principalement basées sur la décomposition fonctionnelle, dont il a été démontré que c'était un facteur d'instabilité, car les fonctionnalités d'un logiciel sont souvent plus volatiles que les concepts d'un domaine métier (comme le secteur bancaire, l'agronomie, la construction automobile, etc.). L'objectif de l'approche à objets est de monter en abstraction et d'avoir des représentations relativement stables par rapport à l'évolution des besoins et des technologies. Dans les approches actuelles de développement, le paradigme basé sur les composants, que nous évoquerons en fin du module, vient faire une synthèse des représentations fonctionnelles et par objets, qui sont toutes deux des points de vue pertinents, même s'ils sont en partie orthogonaux.

Dans l'approche par objets, les concepts apparaissent sous deux aspects : un aspect *structurel* qui décrit la forme des objets et un aspect *dynamique* qui décrit leurs comportements. Dans le cadre plus spécifique de la programmation, plutôt que de se trouver face à des appels de fonctions, on va programmer *un monde d'objets s'envoyant des messages*. La notion de *spécialisation/généralisation ou d'héritage* est également le propre des approches à objets auxquelles elle apporte des qualités en terme d'abstraction, de réutilisation, de maintenance et d'évolutivité. Elle sera développée dans le prochain chapitre.

Dans ce chapitre, nous rappelons les deux notions principales des approches à classes, que sont les classes et les instances et leur notation en UML (modélisation) et en Java (programmation) (Section 2) :

— Une classe décrit les aspects structurels et comportementaux d'un ensemble d'instances. Les termes d'instance et d'objets sont synonymes dans le cadre de ce cours.
— Une instance se conforme à une classe.

Puis nous introduisons la notion de classe imbriquée (Section 3).

2 Classes en UML (rappels)

La figure 1, page 4, présente quelques éléments de la modélisation partielle supposée d'un logiciel bancaire. Deux concepts (classes UML, présentées dans la notation en trois compartiments : noms / attributs / opérations) y apparaissent. Le modèle pourrait être représenté avec une association entre les deux classes, mais ce sera revu dans de prochains exemples. Ici nous trouvons :

— les **comptes bancaires**, décrits par :
  — un numéro, spécifique à chaque compte, qui ne change plus après qu'il ait été attribué au compte ;
  — un titulaire, spécifique à chaque compte, qui est de type client (les classes sont des types dans notre univers) ;
  — un solde, spécifique à chaque compte, qui est un nombre réel ;
  — un plafond, qui est le même pour tous les comptes. Sa valeur, modifiable, étant partagée par tous les comptes, on cherche à ne la stocker qu'une fois ;
  — des opérations de création d'instance (constructeurs), précédées du mot-clef UML <<`create`>> ;
  — des opérations d'accès aux attributs (accesseurs), dont le nom respecte par convention un format particulier (le préfixe `get` pour les accesseurs en lecture et `set` pour les accesseurs en modification/écriture et le suffixe formé avec le nom de l'attribut dont la première lettre a été mise en majuscule) ; suivant la sémantique de la classe, seuls certains accesseurs sont fournis (voir sur l'exemple) ; les accesseurs aux attributs `static` sont souvent `static` eux-mêmes lorsqu'on est sûrs de ne pas avoir à les spécialiser dans une sous-classe ;
  — d'autres opérations spécifiques à une instance, qui sont variées, ici la méthode pour créditer un compte ;

— pour respecter les convention de Java, une opération `toString` qui retourne une chaîne de caractères décrivant une instance. On rappelle qu'elle sera appelée implicitement à chaque fois qu'une expression justifie une transformation de l'objet en chaîne de caractères (par exemple pour une insertion dans un flux, comme un affichage à l'écran) ;
— d'autres opérations globales/générales à la classe (indiquées par `static`), ce sont souvent des opérations utilitaires, comme ici une conversion de monnaie ; dans la pratique courante, elles sont souvent rassemblées dans des classes qui ne contiennent que ce type de méthodes et des constantes (voir la classe `Math` de l'API par exemple).

— les **clients**, décrits par :
  — un nom, spécifique à chaque client ;
  — un portefeuille, spécifique à chaque client, qui est une collection de comptes bancaires ;
  — par souci de simplicité, les opérations de cette classe ne sont pas présentées.

Le bas de la figure 1, page 4 présente l'attribut partagé `plafond` avec sa valeur, deux instances de `CompteBancaire` et une instance de `Client`. Dans les instances, on observe une valuation des attributs spécifiques (non static, c'est-à-dire qui sont des valeurs propres à une instance et non pas des valeurs globales pour le système).

Le code Java partiel correspondant est présenté dans les tables 1, page 3 et 2, page 9 . C'est l'occasion de rappeler certaines règles classiques pour une bonne programmation en Java. Ces rappels apparaissent en partie droite des tables.

TABLE 1 – Classe décrivant partiellement des clients

public class Client {	
	Attributs
private String nom ;	
private ArrayList<CompteBancaire> portefeuille ;	Collection de comptes
}	

3 Les classes imbriquées (*nested classes)*

Les classes internes constituent l'un des modes de structuration du code. Lors du précédent module, nous avons utilisé les paquetages (*package*) qui jouent également ce rôle en permettant le classement thématique des classes, par exemple en Java, on peut trouver un paquetage dédié à des structures de données (`util`), ou un paquetage dédié à l'introspection (`reflect`). Les paquetages peuvent s'organiser dans une hiérarchie d'inclusion des paquetages les uns dans les autres, ce qui régit également le nom des classes. Le nom complet d'une classe est en effet constitué du nom du (des) paquetage(s) la contenant. Ces inclusions s'accompagnent des mécanismes de visibilité que nous reverrons au fur et à mesure du cours et des travaux dirigés et pratiques.

En Java, il existe deux catégories principales de classes internes, dont l'une se subdivise en trois à son tour.

— les classes imbriquées statiques (*static nested classes*)
— les classes imbriquées non statiques
  — classes internes (*inner classes*)

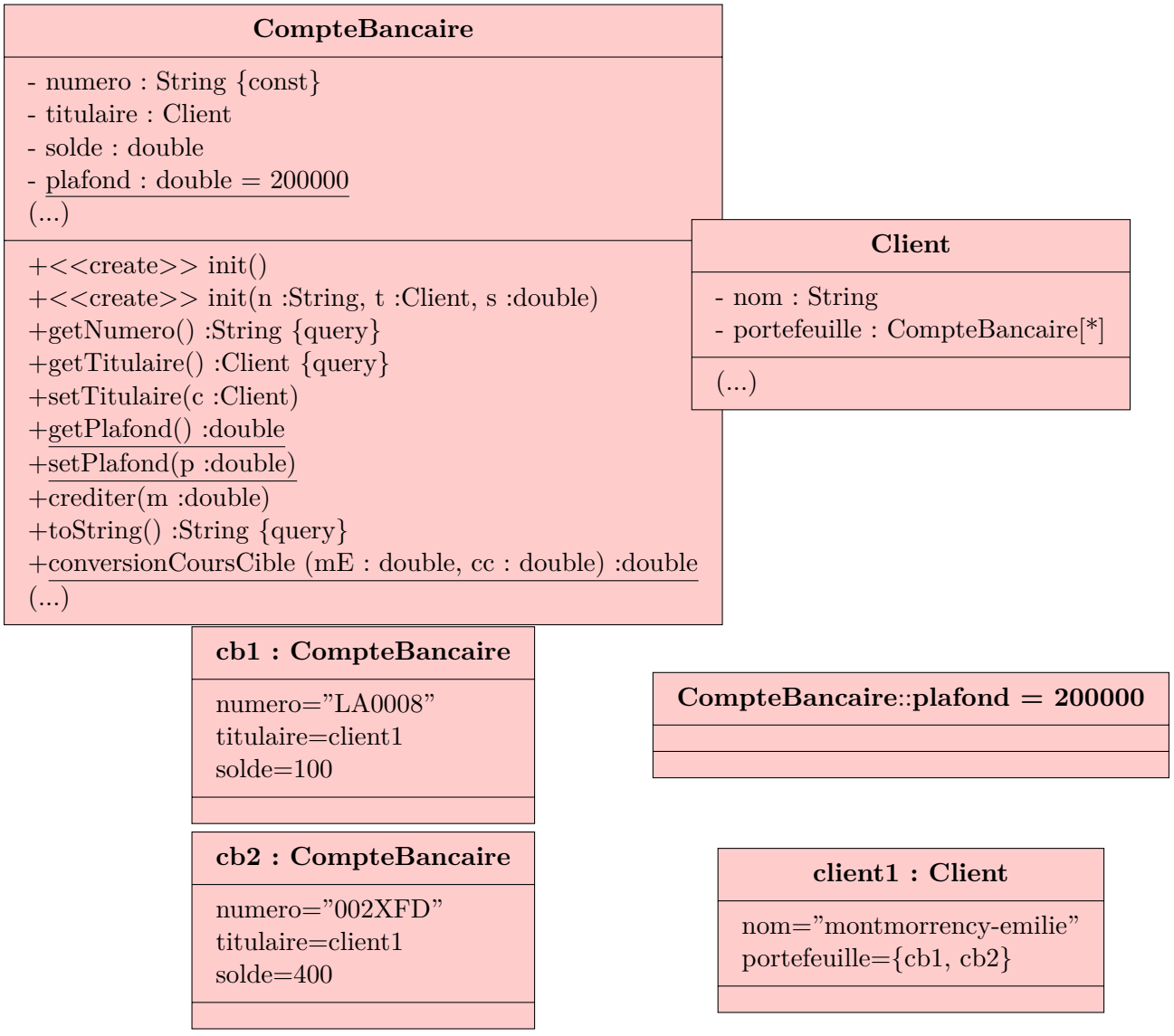

FIGURE 1 – Deux classes `CompteBancaire` et `Client` en UML et quelques instances (vous pouvez ignorer les compartiments vides des boîtes)

— classes locales (*local classes*)
— classes anonymes (*anonymous classes*)

Parmi les usages principaux de ces classes, on peut noter qu'elles servent à :

— grouper des classes qui sont toujours utilisées ensemble, alternativement aux paquetages et souvent en identifiant une classe principale qui contient les autres.
— augmenter l'encapsulation :
  — suivant la visibilité choisie, la classe imbriquée peut être cachée et seulement accessible par la classe englobante.
— faciliter la communication et réduire des temps d'exécution :
  — sous certaines conditions qui seront précisées, la classe imbriquée peut accéder à la partie privée de la classe englobante et inversement. Cela évite des appels de méthodes (notamment d'accesseurs) qui rallongent les temps de calcul, ce qui peut être problématique lors de l'implémentation d'algorithmes coûteux ou appliqués à des données de taille importante.

La figure 3 page 15 propose une synthèse sur les classes imbriquées.

3.1 Les classes imbriquées statiques (*static nested classes*)

Les classes imbriquées statiques respectent les principes généraux suivants :

— Elles sont utilisées principalement comme un moyen de ranger les classes les unes dans les autres pour des raisons thématiques.
— Il n'y a pas de lien particulier entre une instance de la classe imbriquée et une instance de la classe englobante.
— À la visibilité près, la classe imbriquée pourrait être du même niveau que la classe englobante.

Nous prenons comme cas d'étude la description d'une liste chaînée. Une mise en œuvre habituelle avec une approche par objets consiste à définir une classe `Liste` et une classe `Cellule`. Une liste est principalement décrite par un attribut `premier` qui référence la première cellule de la liste. Une cellule est principalement décrite par les attributs `valeur` (contenant la valeur stockée dans la cellule) et `suivante` (référençant la cellule suivante). Nous considérons un modèle dans lequel les cellules peuvent être partagées entre les listes (ce type de modèle permet d'économiser de la mémoire dans certaines applications), dont un exemple est donné par la figure 2, page 5. Dans ce type de modèle de mise en œuvre des listes, il n'y a pas de lien privilégié entre une cellule et une liste puisque les cellules sont partagées. Il se prête donc bien à une illustration de l'intérêt des classes imbriquées statiques.

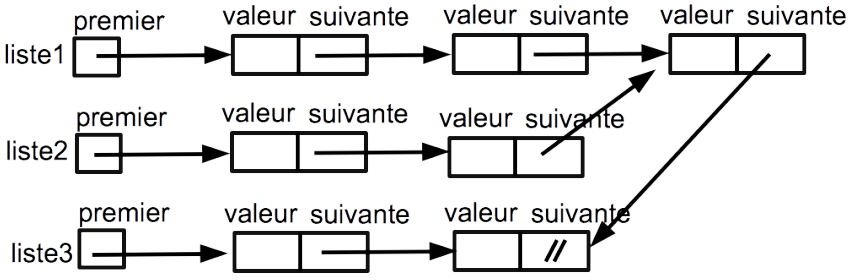

FIGURE 2 – Liste chaînée et cellules, modèle avec partage des cellules entre les listes

La table 3, page 10 détaille des extraits de la classe statique (`Cellule`) imbriquée dans la classe `Liste`. On peut y observer les phénomènes suivants :

— La classe englobante peut accéder aux attributs de la classe imbriquée, même s'ils sont privés.
— La classe imbriquée peut accéder aux attributs **statiques** de la classe englobante, même s'ils sont privés. Elle ne peut accéder aux autres attributs, quelle que soit leur visibilité, que sur une instance créée explicitement.
— Pour créer une instance de la classe imbriquée, on utilise le plus souvent le nom de la classe englobante suivi du '.' suivi du nom de la classe imbriquée.

Dans l'API Java, on trouve des exemples de classes imbriquées statiques dans la description des collections (paquetage `java.util`), comme les classes décrivant les entrées d'un dictionnaire associatif (`map`), incluses dans la classe abstraite décrivant les `map`s.

```
public abstract class AbstractMap<K,V> {
    public static class AbstractMap.SimpleEntry<K,V> {
    // .....
    }
    public static class AbstractMap.SimpleImmutableEntry<K,V> {
    // .....
    }
...
}
```

3.2 Les classes imbriquées non statiques de type classe interne (*inner class*)

Une classe interne s'utilise lorsque chacune de ses instances est associée à une instance de la classe englobante. Ces instances associées partagent une certaine "intimité" avec l'instance englobante.

Nous donnerons un exemple avec la structuration d'une classe `Personne`. Une personne peut être décrite par des informations telles que :

— nom, prénom,
— pays, ville, numéro de rue, nom de rue, code de postal,
— année de naissance.

Cette liste d'attributs possible fait apparaître des groupes, par exemple le deuxième item est un ensemble d'informations relatives à l'adresse d'une personne. Une bonne approche consiste à ajouter une classe `AdressePersonne` rassemblant ces informations qui ont une cohérence entre elles. Cette classe `AdressePersonne` pourrait être du même niveau que la classe `Personne`, mais si dans la modélisation considérée une adresse n'existe qu'en tant qu'adresse d'une personne particulière, utiliser une classe imbriquée non statique (imbriquer `AdressePersonne` dans `Personne`) est une bonne approche. L'instance d'adresse a dans ce cas un lien privilégié implicite vers l'instance de personne (dite englobante).

Une instance de la classe interne peut accéder aux attributs et aux méthodes de son instance associée de la classe englobante quelle que soit leur visibilité. Le même type d'accès peut se faire également en sens inverse. Par conséquent, une stratégie usuelle de création des instances consistera à créer tout d'abord une instance de la classe englobante, puis la (ou les) instance(s) de la classe interne. En programmant, on doit veiller à la logique des créations, en particulier à ce qu'une instance englobante référence sa (ses) propre(s) instance(s) interne(s) et ne référence pas une instance interne à une autre instance englobante.

Une classe interne ne peut pas contenir :

— ni d'attribut `static` (sauf s'il est en plus `final`),
— ni de méthode `static`.

La table 4, page 11 présente un exemple où la classe `Personne` contient la classe interne `AdressePersonne`. Comme dit plus haut, une instance d'adresse est donc naturellement associée à une seule instance de personne (implicite) dans cette modélisation. Une personne pourrait avoir plusieurs adresses associées dans un développement de l'exemple non présenté ici, mais qui peut être réalisé à titre d'exercice d'application du cours.

3.3 Les classes imbriquées non statiques de type classe locale (*local class*)

Une classe locale est une classe interne créée dans un bloc (méthode, corps d'une itération, etc.) et dont la portée sera restreinte à ce bloc. Elle permet de structurer ce bloc en lui ajoutant des déclarations de types qui ne sont pas utiles en d'autres endroits du programme. C'est un procédé qui va permettre d'expliciter de la connaissance métier enfouie dans le code.

Si on considère le code suivant qui permet de saisir les éléments d'un numéro de téléphone dans un flot et de retourner une chaîne construite avec un numéro "numérotable" sur un téléphone :

```
...
public String saisieNumeroTelephone(Scanner sc){

System.out.println("indicatif ?");
String i = sc.next();
System.out.println("numéro local ?");
String j = sc.next();
return "+"+i+j;
}
```

Ce code contient de l'information du domaine non explicitée qui est que :

- le domaine dispose d'un concept de "Numéro de Téléphone" (qui devrait être une classe, par exemple nommée `NumTel`) ;
- les objets de cette classe `NumTel` peuvent être décrits par :
  - un numéro local (information spécifique à chaque numéro de téléphone) ;
  - un indicatif (pour simplifier, information spécifique à chaque numéro de téléphone, quoique partagée entre pays, mais on n'approfondira pas ici pour rester focalisé sur la problématique des classes imbriquées) ;
  - un préfixe (ou code) "+" partagé, global à tous les numéros de téléphone.

Cette information sur le domaine va être explicitée grâce à l'ajout d'une classe locale au bloc (là encore, si cette classe semble devoir exister au-delà du bloc, on pourra la mettre à un niveau d'imbrication autre dans un paquetage). Dans l'exemple présenté (table 5, page 12), dans une classe `Autorisation`, ajoutée pour avoir un code complet, la méthode de construction du numéro de téléphone `saisieNumTel` déclare la classe locale non statique `NumTel` décrivant le format d'un numéro et une méthode de saisie.

La classe locale non statique ne peut pas comporter de visibilité ; elle peut seulement être abstraite ou `final` (pour une classe, cela signifie qu'elle est non spécialisable, c'est-à-dire qu'elle ne peut pas avoir de sous-classe).

Le bloc englobant a accès aux attributs et méthodes de la classe locale, même s'ils sont privés.

La classe locale a accès aux paramètres et aux variables du bloc, à condition qu'ils soient spécifiés `final` ou non modifiés dans les faits, et comme toute classe interne, à son instance englobante implicite.

3.4 Les classes imbriquées non statiques de type classe anonyme (*anonymous class*)

Les classes anonymes sont des variantes des classes locales, qui ne possèdent pas de nom. Ce sont des expressions. Elles se construisent sur la base d'une classe support (ou d'un type interface) préalablement déclarée, et qui peut être abstraite.

Dans l'exemple précédent, si on ne désire pas donner un nom à la classe `NumTel`, on peut la déclarer de manière anonyme (table 6, page 13). Par souci de clarté du code, on commence par définir une classe abstraite `Saisissable` qui représente les objets disposant d'une méthode de saisie sur un `Scanner`. Puis on retrouve la classe `Autorisation`, dont la méthode de saisie contiendra l'expression de définition et d'instanciation simultanée de la classe interne anonyme. Les tables 6, page 13) et 7, page 14) montrent deux implémentations possibles. L'implémentation proposée dans la table 7, page 14 utilise la classe `Object` comme classe de base (ou super-classe) pour la classe anonyme, et met `code` dans la classe anonyme comme une information globale (`static`), ce qui est recommandé ici.

Les classes internes anonymes sont très utilisées lorsque l'on fait de la programmation graphique événementielle, afin de définir des objets temporaires (comme un objet bouton ou un objet menu ayant des particularités dont il est le seul à disposer, ce qui ne justifie pas de créer une classe). A part ces cas très particuliers, il est souvent préférable de nommer les classes.

TABLE 2 – Classe décrivant partiellement des comptes bancaires

public class CompteBancaire {	**Entête de la classe**
	Attributs privés
private final String numero ;	numero : Constant, spécifique
private Client titulaire ;	titulaire : Spécifique
private double solde ;	solde : Spécifique
private static double plafond = 200000 ;	plafond : Partagé, global
// *un constructeur porte le nom de sa classe*	**Constructeurs publics**
public CompteBancaire()	Toujours un constructeur
{ this.numero = "compte par defaut" ; }	sans paramètres,
public CompteBancaire(String num, Client tit, double solde) {	et un ou plusieurs constructeurs avec
this.numero = num ;	des paramètres suivant la sémantique
this.setTitulaire(tit) ;	de construction.
this.solde = solde ;	ici, on crée un compte après son client
}	`this` désigne l'instance receveur
	Accesseurs suivant sémantique
public Client getTitulaire() { return titulaire ; }	Nom codifié
public void setTitulaire(Client titulaire)	
{ this.titulaire = titulaire ; }	
public double getSolde() {return solde ; }	Pas de setSolde, la méthode `créditer`
	contrôle la modification du solde
public static double getPlafond() { return plafond ; }	Accesseurs static
public static void setPlafond(double plafond) {	quand l'attribut est `static`
CompteBancaire.plafond = plafond ;	
}	
public String getNumero() { return numero ; }	Pas de setNumero car numéro est `final`
	Autres méthodes
public void crediter(double montant){	on modifie le solde
if (montant >=0) this.solde += montant ;	par des méthodes
else System.out.println("Erreur") ;	qui effectuent des contrôles
}	
public String toString()	Méthode conventionnelle `toString`
{	Transformation d'une instance
return this.numero+" "+this.titulaire +" "+this.solde ;	en chaîne de caractères
}	
public static double conversionCoursCible	
(double montantEuros, double coursCible){	méthode utilitaire
return montantEuros*coursCible ;	non spécifique à un compte bancaire
}	
public static void main(String[] a){	**Programme principal**
Client cl1 = new Client() ;	avec des créations d'instances
CompteBancaire cb1 =	
new CompteBancaire("LA0008",cl1,100) ;	
CompteBancaire cb2 =	
= new CompteBancaire("002XFD",cl1,400) ;	
System.out.println(cb1) ;	
}	
}	

TABLE 3 – La classe Cellule est une classe imbriquée statique de la classe Liste

public class Liste { private Cellule premier ; private static int nbListes ; (...) private static **class Cellule {** **private int valeur ;** **private Cellule suivante ;** (...) **public String toString(){** Liste l=new Liste() ; **return ""+nbListes** ~~//+premier ;~~ +" "+l.premier ; **} // fin classe Cellule** public Liste() {} public void ajouteTete(int v){ Liste.Cellule c = new Liste.Cellule() ; c.valeur=v ; c.suivante=premier ; premier=c ; } public static void main(String[] a){ Liste l = new Liste() ; } } // fin classe Liste	`Liste` est la classe englobante Elle a ses propres attributs, `premier` et `nbListes` (ce dernier est static) La class imbriquée `Cellule` est statique Elle peut avoir différentes visibilités, ici `private` `Cellule` a ses propres attributs `valeur` et `suivante` `Cellule` a sa propre méthode `toString` On peut accéder aux membres static de la classe englobante (comme `nbListes`). Mais pas à `premier` qui n'est pas static sauf sur une instance de `Liste` créée explicitement Constructeur de `Liste` Méthode d'ajout dans une `Liste` Création d'instance de la classe imbriquée La classe englobante peut accéder aux attributs privés de la classe imbriquée

TABLE 4 – La classe Adresse est une classe interne de la classe Personne

```
public class Personne {
  private String nom;
  private AdressePersonne ad;

  public Personne(String nom, String ville,
                  String pays) {
    this.nom = nom;
    ad = this.new AdressePersonne(ville, pays);
  }

  public String toString(){
    return nom+" habite "+ad;


    //return nom+" habite "+ad.ville;
  }

  private class AdressePersonne{
    private String ville="là", pays="ailleurs";
    private static int nbAdresses=0;
    private static final String entete="Adresse";
    public AdressePersonne(String ville,
                           String pays)
    {this.ville = ville; this.pays = pays;}
    public String toString(){
      return ville+" "+pays;

      //return Personne.this.nom+this.ville+...;
      //return nom+this.ville+...;


      //return this.nom;
    }
  }

  public static void main(String[] args) {
    Personne j =
        new Personne ("Jacky","Boston","USA");
    System.out.println(j);
  }
}
```

référence vers une instance de la classe interne

création d'une instance interne

code standard avec partage des responsabilités sur `toString` (on appelle `toString` sur `ad` plutôt que sur ses attributs)

exemple d'accès à un attribut de l'instance interne si nécessaire

on ne peut pas avoir une variable `static`

on peut avoir une variable `static` ET `final`

code standard avec partage des responsabilités sur `toString`

accès à un attribut de l'instance englobante

accès à un attribut de l'instance englobante (alternative montrant que l'instance englobante est implicite)

mais `this.nom` provoque une erreur !

TABLE 5 – La classe NumTel est une classe locale de la méthode saisieNumTel

public class Autorisation { private int num ; public Autorisation() {} public String saisieNumTel(Scanner sc) { System.out.println(”saisie num tel”) ; String numComplet ; String code=”+” ; **class NumTel{** **private String indicatif ;** **private String numerolocal ;** **public String saisie(Scanner sc){** **System.out.println(”saisie indicatif”) ;** **indicatif = sc.next() ;** **System.out.println(”saisie numero”) ;** **numerolocal = sc.next() ;** //~~**numComplet = indicatif+” ”+... ;**~~ **return code+indicatif+” ”+numerolocal ;** **//+Autorisation.this.num ;** **//+num ;** **}** **}** // fin classe NumTel NumTel n = new NumTel() ; numComplet = n.saisie(sc) ; return numComplet ; }//fin saisieNumTels public static void main(String[] args) { Scanner sc = new Scanner(System.in) ; Autorisation a = new Autorisation() ; System.out.println(a.saisieNumTel(sc)) ; } }	 pas de visibilité indiquée on ne peut affecter une valeur à numComplet ici on peut utiliser la variable locale `code` accès possible à l’instance implicite idem (comme dans une classe interne) création d’un objet de la classe locale

TABLE 6 – Classe anonyme dans la méthode saisieNumTel

public class Saisissable{ public String saisie(Scanner sc){ System.out.println("On peut me saisir !") ; return "" ; } } public class AutorisationAnonyme { public String saisieNumTel(Scanner sc) { System.out.println("saisie num tel") ; String code="+" ; Saisissable n = **new Saisissable(){** **private String indicatif ;** **private String numerolocal ;** **public String saisie(Scanner sc){** **System.out.println("saisie indicatif") ;** **indicatif = sc.next() ;** **System.out.println("saisie numero") ;** **numerolocal = sc.next() ;** **return code+indicatif+" "+numerolocal ;** **}** **} ; // fin classe anonyme** return n.saisie(sc) ; }//fin saisieNumTels public static void main(String[] args) { Scanner sc = new Scanner(System.in) ; AutorisationAnonyme a = new AutorisationAnonyme() ; System.out.println(a.saisieNumTel(sc)) ; } }	définition de la classe support mais ici la classe `Object` aurait pu servir de support également (voir listing suivant) définition de la classe anonyme par spécialisation de `Saisissable` et instanciation en même temps

TABLE 7 – Classe anonyme dans la méthode `saisieNumTel`, basée sur `Object`

```
public class AutorisationAnonyme {

   public String saisieNumTel(Scanner sc) {
      System.out.println("saisie num tel");
      return new Object(){
        private String indicatif;
        private String numerolocal;
        private final static String code="+";
        public String saisie(Scanner sc){
         System.out.println("saisie indicatif");
         indicatif = sc.next();
         System.out.println("saisie numero");
         numerolocal = sc.next();
         return code+indicatif+" "+numerolocal;
        }
      }.saisie(sc);
   }//fin saisieNumTel

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      AutorisationAnonyme a = new AutorisationAnonyme();
      System.out.println(a.saisieNumTel(sc));
   }
}
```

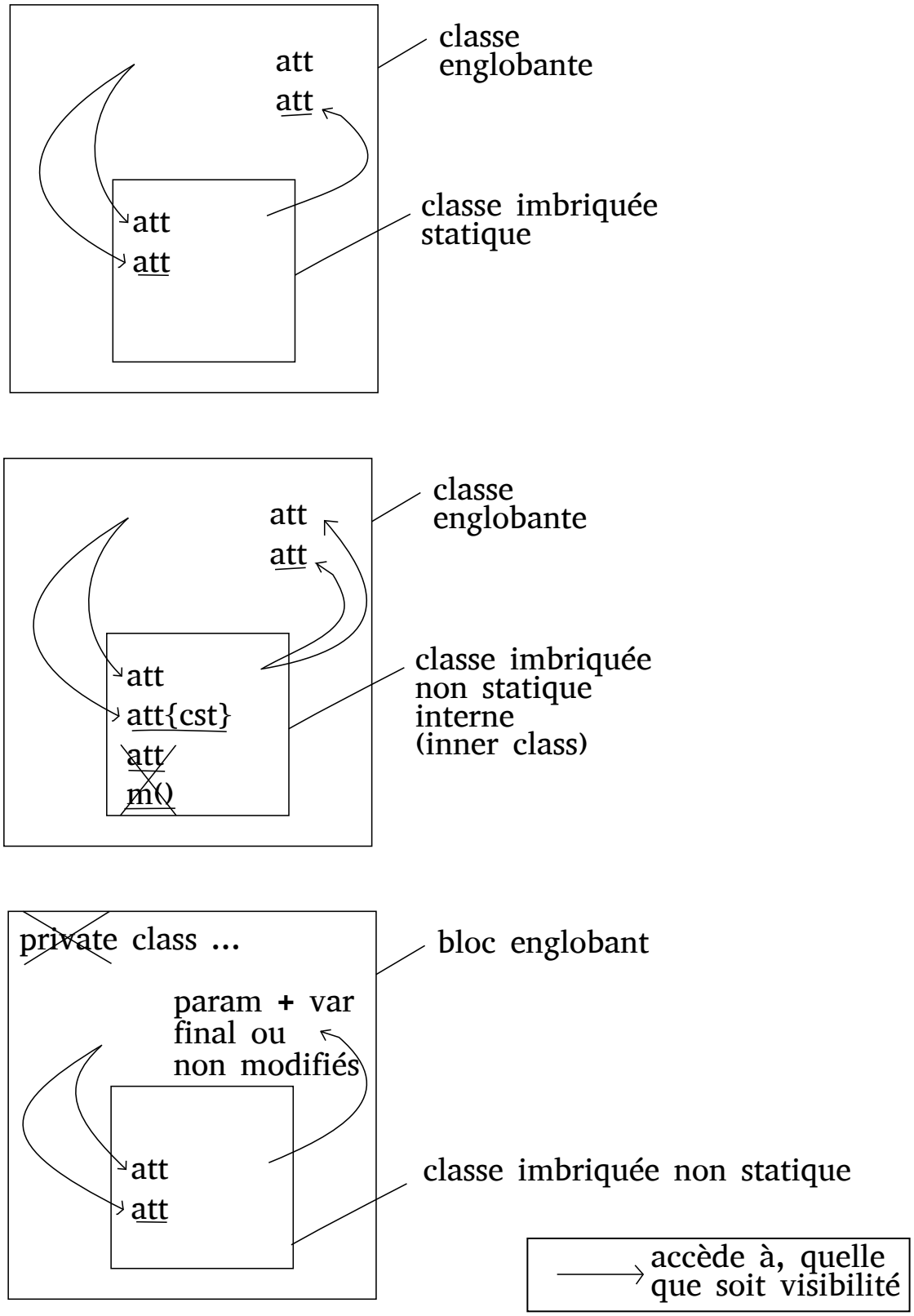

FIGURE 3 – Vue synthétique des classes imbriquées