Héritage

Spécialisation généralisation

Modélisation et Programmation par objets 1 – cours 3

Nous introduisons ici la notion d'héritage de classe, permettant de mettre en œuvre la généralisation et la spécialisation de classes. Nous n'aborderons ici que l'héritage simple, puisque c'est le seul que Java permette de mettre en œuvre.

1 Classification d'objets et hiérarchies de classes, héritage de propriétés

La généralisation est un mécanisme qui consiste à réunir des objets possédant des caractéristiques communes dans un nouveau concept plus général. Le processus de généralisation/spécialisation est très naturellement utilisé : l'esprit humain aime à classifier les objets du monde qui l'entoure.

Si l'on prend l'exemple de la classification animale présentée à la figure 1, on classifie les animaux en fonction des caractéristiques qu'ils ont ou qu'ils n'ont pas. Les animaux ont des caractéristiques communes (ici par exemple avoir une tête). On peut ensuite distinguer les vertébrés (qui ont un squelette interne) des arthropodes (qui ont un squelette extérieur). Au plus profond de la classification, on trouve des animaux. Les caractéristiques de chaque animal se trouvent en faisant l'union des caractéristiques introduites tout au long de la branche de l'animal jusqu'à la racine. Ainsi, le chat est un mammifère (il a donc des poils et des mamelles), il a 4 membres, il a un squelette d'os, etc. On peut aussi remarquer que plus on est haut dans la classification, moins on introduit de caractéristiques (plus petite intension [1]), mais plus on décrit un grand nombre d'animaux (plus grande extension). Réciproquement, plus on est bas, plus on a de caractéristiques (plus grande intension) mais moins d'animaux ont de grand nombre de caractéristiques (plus petite extension).

On va reproduire ce genre de classification grâce au mécanisme de généralisation/spécialisation propre aux approches à objets.

1.1 Sous-classage

Une classe permet de décrire un nombre quelconque d'objets qui en seront instances. Toutes ces instances partagent les caractéristiques introduites dans la classe. Il arrive fréquemment que l'on veuille introduire des spécialisations de la classe, permettant de décrire des instances possédant en plus certaines autres caractéristiques.

Regardons sur un exemple. Nos personnages de jeu peuvent appartenir à un clan. Un clan a un nom, un capital (en terme de nombre de pièces), des membres (qui sont des personnages) et un créateur (le personnage qui l'a créé). Les personnages qui le souhaitent peuvent s'inscrire au clan pour en devenir membre. Parmi tous les clans, on voudrait pouvoir également avoir des clans avec une capacité limitée (au delà de cette capacité, le clan n'accepte plus de nouveaux membres) et des clans élitistes (qui n'acceptent parmi leurs membres que des personnages dont le nombre de points de vie et le nombre de pièces sont au delà de seuils fixés).

Pour représenter les différentes sortes de clan, nous les organisons dans une hiérarchie de classes. Nous introduisons 3 classes (Clan, ClanLimiteCapa et ClanElitiste). Les clans avec capacité et les clans élitistes sont des clans particuliers : ils ont les caractéristiques des clans, plus d'autres. Les classes ClanLimiteCapa et ClanElitiste héritent de la classe Clan. On dit que Clan est la **classe mère**, et ClanLimiteCapa et ClanElitiste les classes filles. On dit aussi que Clan est la **super-classe** des classes ClanLimiteCapa et ClanElitiste.

Du point de vue des classes, une classe C_{mere} généralise une autre classe C_{fille} si l'ensemble des objets de C_{fille} est inclus dans l'ensemble des objets de C_{mere}. La relation de spécialisation doit pouvoir se lire : "est un" : un clan élitiste est un clan. Du point de vue des objets (instances des classes), toute instance d'une sous-classe peut jouer le rôle (peut remplacer) d'une instance d'une des super-classes de sa hiérarchie de spécialisation-généralisation.

1.1.1 Notation de l'héritage en UML

En UML, nous représentons la relation d'héritage avec une flèche à tête triangulaire creuse pointant la classe mère. Il est important de bien respecter cette notation car à peu près toutes les formes de flèches ont une sémantique différente. Cela est illustré à la figure 3.

On peut noter que les 2 spécialisations qui sont introduites ici utilisent comme critère de spécialisation la gestion des membres du clan. Il est possible de le mentionner explicitement sur le digramme en UML, nous en reparlerons à la section 6.

1. Ceci n'est pas une faute d'orthographe

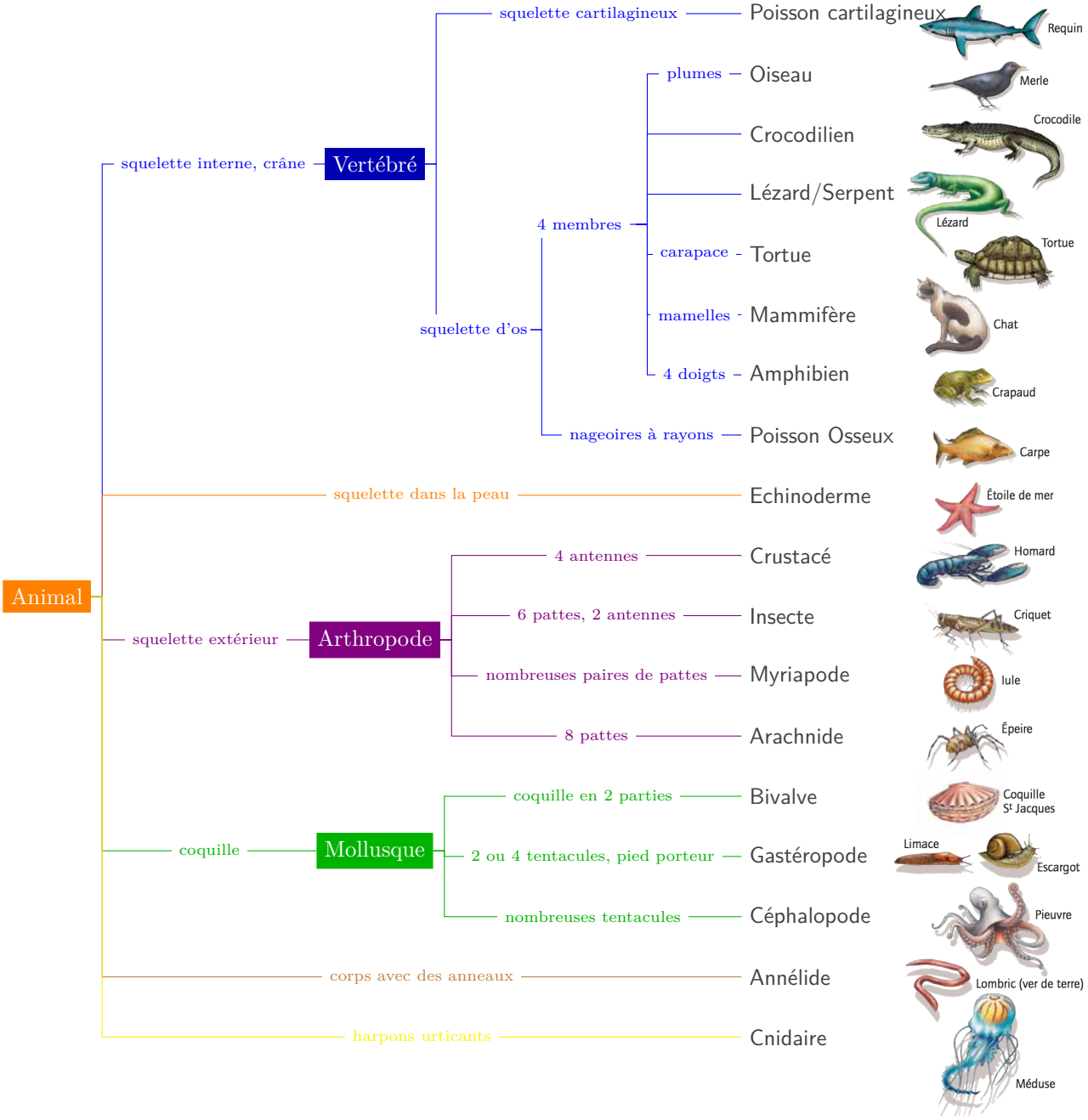

Figure adaptée de : comprendre et enseigner la classification du vivant, Belin, 2004

FIGURE 1 – Classification animale

1.1.2 Notation de l'héritage en Java

En Java, la relation d'héritage se note avec le mot-clef **`extends`** placé dans l'entête des classes filles, comme indiqué au code 1.

Une classe n'héritant explicitement d'aucune autre classe hérite implicitement de la classe `Object`. Nous verrons plus loin quelques conséquences de cela.

Comme nous l'avons déjà dit, Java ne permet que l'héritage simple, et ne met pas en place d'héritage multiple qui permettrait à une même classe d'étendre plusieurs classes. D'autres langages à objets (comme Eiffel ou C++) ont fait d'autres choix.

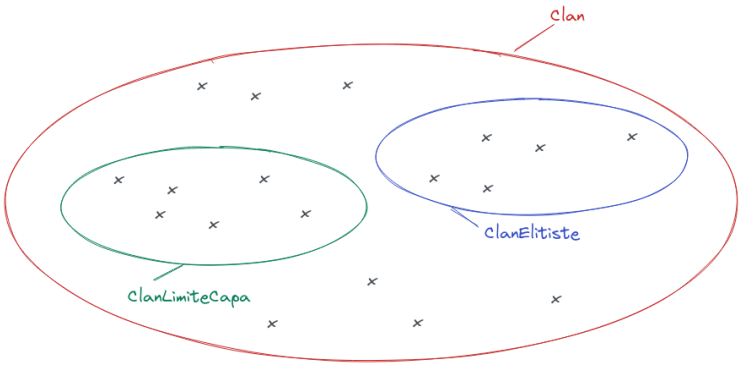

FIGURE 2 – Des clans, des clans avec limite de capacité, des clans élitistes

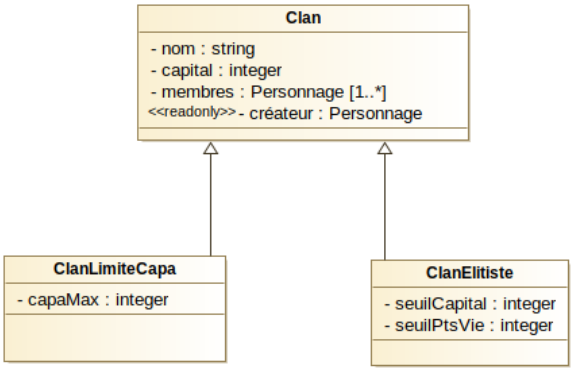

FIGURE 3 – Héritage et attributs

Programme 1 : *Héritage en Java : mot-clef extends*

```
public class Clan {
        private String nom;
        private int capital;
        private final Personnage créateur;
        private Personnage[] membres;
}
...
public class ClanLimiteCapa extends Clan {
        private int capaMax;
}
...
public class ClanElitiste extends Clan {
        private int seuilCapital;
        private int seuilPtsVie;
}
```

On remarque que la classe mère ne porte aucune marque de sous-classage.

L'attribut `private Personnage[] membres` permet de représenter les membres du clan, sous forme de tableau.

1.2 Héritage de propriétés

Un clan avec limite de capacité a un nom, mais on ne fait pas apparaître ce nom dans la classe ClanLimiteCapa : cet attribut est naturellement hérité de la classe Clan. Une instance de ClanLimiteCapa portera donc une valeur pour cet attribut.

Nous illustrons cet héritage de propriétés avec la représentation de 3 clans par un diagramme d'objets. Pour simplifier nous ne représentons pas ici les membres des clans sur le diagramme.

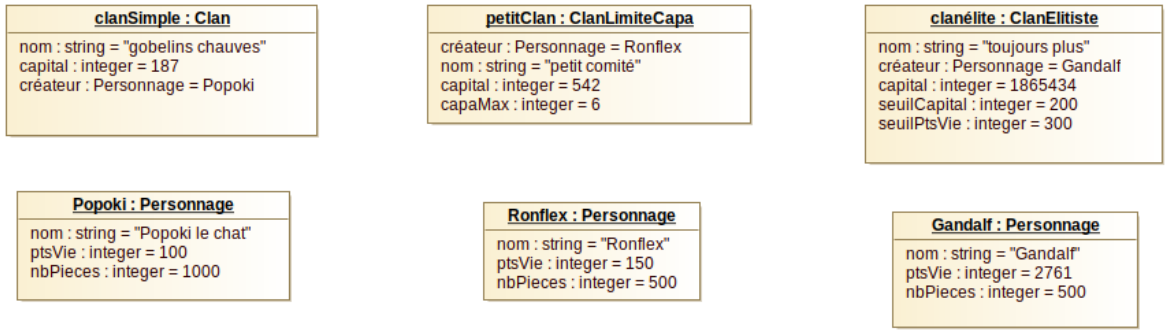

FIGURE 4 – Un diagramme d'objets en présence d'héritage

1.2.1 Héritage et visibilité des attributs

Les instances des sous-classes portent une valeur pour les attributs introduits dans la super-classe, ainsi que nous venons de le voir. Toutefois, si les attributs ont été déclarés (comme c'est l'usage) avec une visibilité privée, alors ces valeurs ne pourront pas directement être manipulées dans les sous-classes, elles n'y seront pas visibles.

Il y a deux façons de résoudre ce problème de visibilité.

Visibilité protégée. La première façon consiste à mettre une visibilité dite protégée aux attributs. En UML, on utilise pour cela le symbole #. Cette visibilité permet de donner accès à cet attribut aux instances de la classe ainsi qu'aux instances des sous-classes. En java, on peut utiliser le mot-clef `protected`. Toutefois, avec cette visibilité, on donne accès aux instances de la même classe, aux instances des sous-classes (comme en UML) mais aussi aux instances de classes du même package. Dans la suite, nous illustrons cette visibilité protégée pour les membres des clans.

Accesseurs et méthodes La seconde façon consiste à laisser la visibilité privée, mais à donner accès aux instances des sous-classes à des méthodes et/ou des accesseurs permettant la manipulation des attributs privés. Dans la suite, nous illustrons cette façon de procéder avec les attributs nom, capital et créateur.

1.2.2 Héritage et constructeurs

Comme toute classe, une sous-classe est munie de constructeurs. Les constructeurs des sous-classes sont chargés d'initialiser tous les attributs pour l'instance qui est construite. Les attributs à initialiser sont de 2 sortes : ils sont soit introduits dans la sous-classe, soit hérités. Dans les 2 cas, ils doivent être initialisés. En général, l'initialisation des attributs hérités se fait par appel du constructeur de la super-classe.

Regardons sur notre exemple :

- On prévoit un unique constructeur de clans prenant en paramètre : le nom du clan, le capital et le créateur du clan (le personnage en ayant demandé la création). On ne prend pas en paramètre de quoi alimenter les membres du clan : initialement, le clan sera créé sans membres, les membres pourront être ajoutés par la suite (le créateur n'est pas membre d'office).
- On prévoit dans la classe clan à capacité limitée un constructeur permettant d'initialiser tous les attributs, introduits ou hérités, et prenant donc en paramètres le nom, le capital et le créateur du clan (comme le constructeur de Clan) mais aussi la capacité maximale du clan.
- De manière similaire, on prévoit dans la classe ClanElitiste un constructeur prenant en paramètres le nom, le capital et le créateur du clan ainsi que les seuils de capital et de points de vie requis pour intégrer le clan.

Ces constructeurs sont positionnés dans le modèle UML de la figure 5.

Nous illustrons ces mêmes constructeurs en Java au code 2. On note dans ces constructeurs l'appel au super-constructeur via le mot-clef `super` suivi des paramètres à passer au super-constructeur. Cet appel soit être placé en première instruction du constructeur :

- si l'on le met ailleurs (par exemple en seconde instruction), le code ne compile pas ;
- si l'on omet complètement l'appel au super-constructeur, Java appelle implicitement le super-constructeur non paramétré (`super()`. S'il n'existe pas, une erreur de compilation est levée. C'est pour cela qu'il est conseillé de munir les classes de constructeurs non paramétrés (en plus des constructeurs paramétrés).
- une conséquence du point précédent : il arrive fréquemment qu'un constructeur ne serve que de "courroie de transmission" entre le constructeur de sa sous-classe et celui de sa super-classe, en se contentant d'appeler le constructeur de la super-classe en lui transmettant exactement les paramètres reçus, sans ajouter aucun comportement supplémentaire.

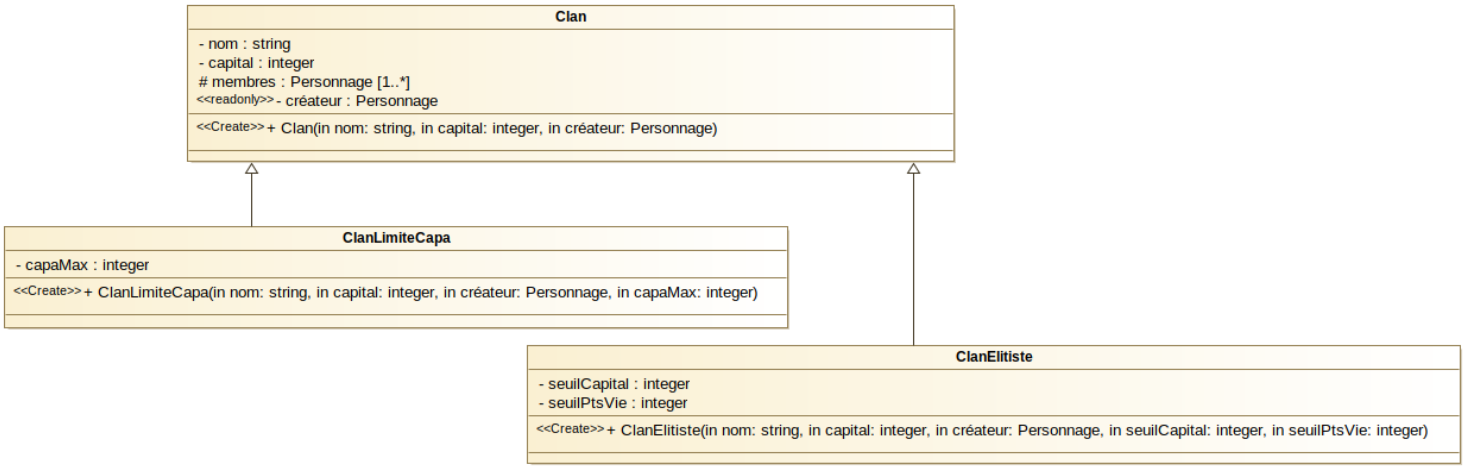

FIGURE 5 – Constructeurs en présence d'héritage

Programme 2 : *Héritage en Java : les constructeurs*

```
public class Clan {
        private String nom;
        private int capital;
        private Personnage créateur;
        protected Personnage[] membres;
        public Clan(String nom, int capital, Personnage créateur) {
                this.nom = nom;
                this.capital = capital;
                this.créateur = créateur;
                membres=new Personnage[100]; // construction du tableau, initialement 100 cases. S'il
                ↪  y a plus de 100 membres, il faudra augmenter cette taille
        }
}
...
public class ClanLimiteCapa extends Clan {
        private int capaMax;
        public ClanLimiteCapa(String nom, int capital, Personnage créateur, int capaMax) {
                super(nom, capital, créateur); // appel du constructeur de la super-classe
                this.capaMax = capaMax;
                membres=new Personnage[capaMax]; // on connait la taille max exacte du tableau
        }
}
...
public class ClanElitiste extends Clan {
        private int seuilCapital;
        private int seuilPtsVie;
        public ClanElitiste(String nom, int capital, Personnage créateur, int seuilCapital, int
        ↪  seuilPtsVie) {
                super(nom, capital, créateur); // appel du constructeur de la super-classe
                this.seuilCapital = seuilCapital;
                this.seuilPtsVie = seuilPtsVie;
        }
}
```

Remarque L'utilisation de `super` dans les constructeurs permet de mieux gérer l'évolution du code, une modification d'un constructeur d'une super-classe provoquant automatiquement la modification des constructeurs des sous-classes faisant appel à

lui par **super**.

Appel des constructeurs Nous illustrons ci-dessous quelques appels aux constructeurs que nous venons de voir.

Programme 3 : *Héritage en Java : appeler les constructeurs*

```
Personnage popoki=new Personnage("Popoki le chat", 100, 1000);
Personnage ronflex=new Personnage("Ronflex", 150, 1500);
Personnage gandalf=new Personnage("Gandalf", 500, 2761);

Clan clanSimple=new Clan("gobelins chauves", 187, popoki);
ClanLimiteCapa petitClan=new ClanLimiteCapa("petit comité", 542, ronflex, 6);
ClanElitiste clanélite=new ClanElitiste("toujours plus", 1865434, gandalf, 200, 300);
```

2 Spécialisation et héritage simple de comportements

Nous avons vu comment l'héritage permettait de spécialiser une classe du point de vue structurel, c'est-à-dire du point de vue de ses attributs. Nous allons maintenons regarder l'héritage comportemental.

En plus d'hériter des attributs de sa classe mère, une classe hérite également de ses comportements, c'est-à-dire de ses méthodes (si elles lui sont accessibles). Cela signifie qu'une instance d'une sous-classe pourra recevoir un appel à une méthode qui est définie dans la classe mère.

Regardons un exemple. Nous introduisons dans la classe Clan deux méthodes :

— taille qui retourne le nombre de membres du clan,

— et capitalTotal qui retourne la somme du capital de chacun des membres du clan avec la capital propre au clan.

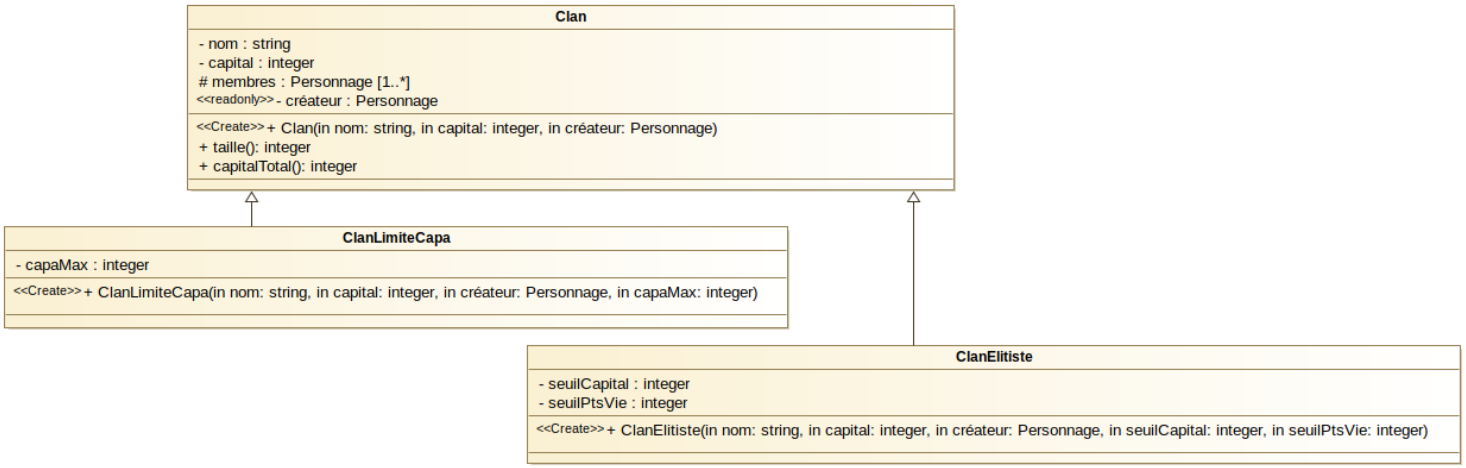

FIGURE 6 – Hiérarchie de classes avec des méthodes dans la super-classe

Sans rien préciser de plus dans les sous-classes de Clan, chaque instance de ClanLimiteCapa et chaque instance de ClanElististe pourra recevoir un appel à ces 2 méthodes, ainsi qu'illustré en Java ci-dessous.

Programme 4 : *Héritage en Java : appeler des méthodes héritées*

```
public class Clan {
        private String nom;
        private int capital;
        private Personnage créateur;
        protected Personnage[] membres;
        public Clan(String nom, int capital, Personnage créateur) {
                this.nom = nom;
                this.capital = capital;
                this.créateur = créateur;
                membres=new Personnage[100]; // construction du tableau, initialement 100 cases. S'il
                 ↪  y a plus de 100 membres, il faudra augmenter cette taille
```

```
        }
        public int taille() {
                int t=0;
                while(t<membres.length&&membres[t]!=null) {
                        t++;
                }
                return t;
        }

        public int capitalTotal() {
                int c=capital;
                for (int i=0;i<taille();i++) {
                        if (membres[i]!=null) {
                                c+=membres[i].getNbPieces();
                        }
                }
                return c;
        }
}
...
public class ClanLimiteCapa extends Clan {
        private int capaMax;
        public ClanLimiteCapa(String nom, int capital, Personnage créateur, int capaMax) {
                super(nom, capital, créateur); // appel du constructeur de la super-classe
                this.capaMax = capaMax;
                membres=new Personnage[capaMax]; // on connait la taille max exacte du tableau
        }
}
...
public class ClanElitiste extends Clan {
        private int seuilCapital;
        private int seuilPtsVie;
        public ClanElitiste(String nom, int capital, Personnage créateur, int seuilCapital, int
        ↪  seuilPtsVie) {
                super(nom, capital, créateur); // appel du constructeur de la super-classe
                this.seuilCapital = seuilCapital;
                this.seuilPtsVie = seuilPtsVie;
        }
}
...
Clan clanSimple=new Clan("gobelins chauves", 187, popoki);
ClanLimiteCapa petitClan=new ClanLimiteCapa("petit comité", 542, ronflex, 6);
ClanElitiste clanélite=new ClanElitiste("toujours plus", 1865434, gandalf, 200, 300);
System.out.println(clanSimple.taille() +" "+clanSimple.capitalTotal()); // 0 187
System.out.println(petitClan.taille() +" "+petitClan.capitalTotal()); // 0 542
System.out.println(clanélite.taille() +" "+clanélite.capitalTotal()); // 0 1865434
```

Evidemment, dans notre exemple, comme tous les clans sont vides, ces méthodes ont un intérêt limité ... Nous allons voir l'ajout de membres dans les clans à la section suivante.

Comme pour les attributs, les méthodes héritées sont accessibles dans les sous-classes si leur visibilité le permet. On peut utiliser la visibilité protégée.

3 Spécialisation et redéfinition de comportements

Nous avons vu à la section précédente qu'une sous-classe héritait naturellement des comportements de la super-classe.

Il est très fréquent que dans une sous-classe, on souhaite modifier le comportement hérité de la super-classe : soit pour le remplacer complètement par un autre comportement (on parle alors de masquage), soit pour lui ajouter un comportement spécifique (on parle alors de spécialisation).

3.1 Spécialisation

Quand le comportement hérité convient, mais qu'on veut y ajouter un comportement spécifique, on le redéfinit dans la sous-classe en le spécialisant.

Par exemple, on s'intéresse pour les clans à une méthode permettant d'inscrire un personnage comme membre du clan :

— Dans la classe Clan, le comportement de cette méthode est d'ajouter le personnage au tableau des membres (en le redimensionnant si besoin).

— Dans la classe ClanLimiteCapa, le comportement hérité doit être modifié : on ajoute effectivement le personnage, mais uniquement si la capacité le permet.

— Dans la classe ClanElitiste, le comportement est là aussi un peu différent : on ajoute le personnage uniquement s'il est suffisamment fort (vis à vis des seuils fixés).

La méthode est alors redéfinie et spécialisée dans chacune des classes filles.

3.2 Masquage

Quand le comportement hérité ne convient pas, on le redéfinit dans la sous-classe.

Par exemple, on s'intéresse pour les clans à une méthode permettant d'éjecter un de ses membres :

— Dans la classe Clan, le comportement de cette méthode est de choisir aléatoirement un des membres et de le supprimer de la liste des membres.

— Dans la classe ClanLimiteCapa, le comportement est différent : on éjecte le dernier membre ayant intégré le clan.

— Dans la classe ClanElitiste, le comportement est là aussi différent : on éjecte le membre dont le cumul des points de vie et des pièces est le plus bas (toujours ce charmant côté élitiste).

La méthode est alors redéfinie et masquée dans chacune des classes filles : la méthode d'une fille masque celui de la mère.

3.3 Redéfinition : spécialisation et masquage sur notre exemple

Nous aboutissons donc au diagramme de classe présenté à la figure 7

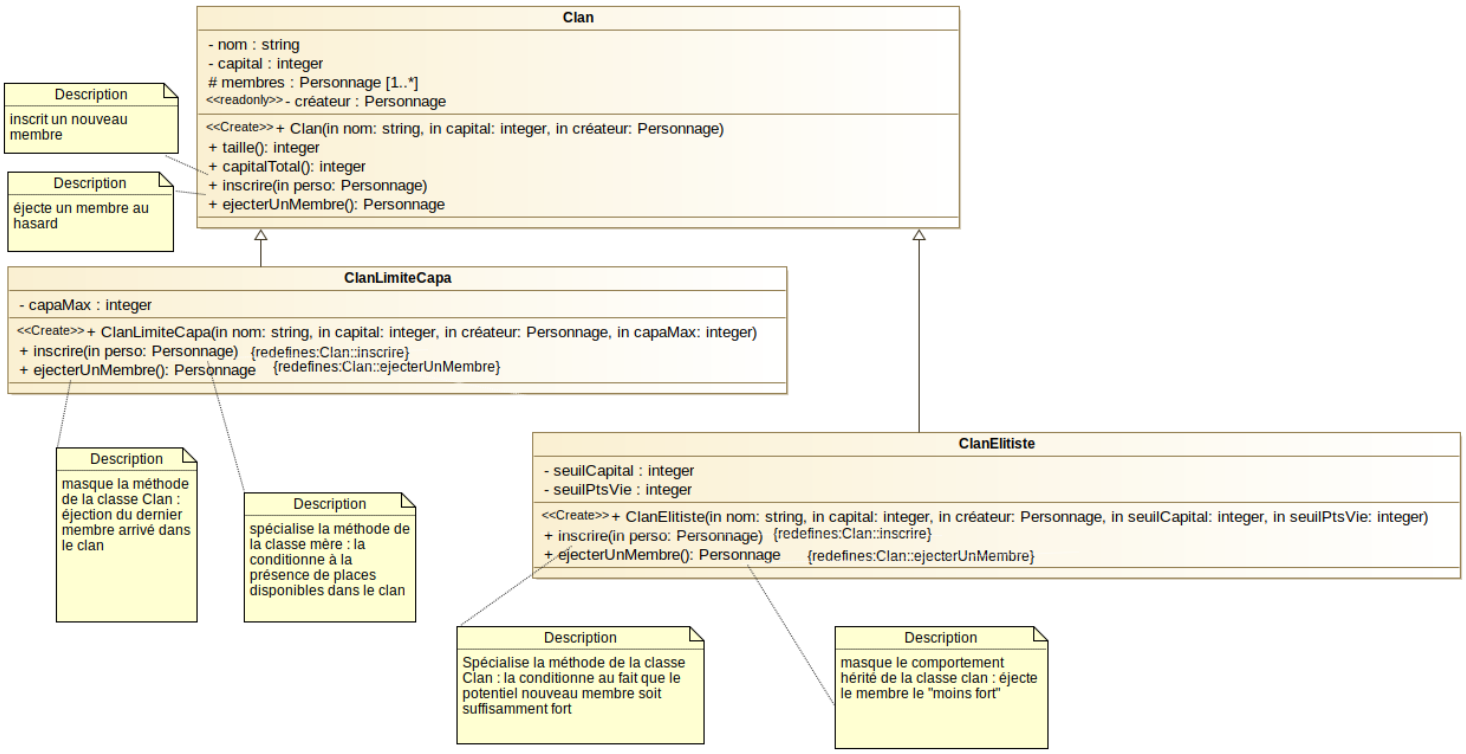

FIGURE 7 – Redéfinition : masquage et spécialisation

Programme 5 : *Redéfinition de méthodes en Java : classe Clan*

```
public class Clan {
        private String nom;
        private int capital;
        private Personnage créateur;
        protected Personnage[] membres;
        ...
        public void ajoutMembre(Personnage p) {
                int taille=taille();
                if (membres.length>taille) { // il y a de la place dans le tableau
                        membres[taille]=p; //il faudrait vérifier que p n'est pas déjà dans le clan
                } else {
                        // il faut agrandir le tableau. On ne le fait pas ici ...
                }
        }
        public Personnage ejecterUnMembre() {
                Personnage resultat=null;
                int taille=taille();
                if (taille()!=0) {
                        Random rand=new Random(); // java.util.Random
                        int i=rand.nextInt(taille);
                        resultat=membres[i];
                        membres[i]=null;
                        // décalage à gauche
                        for (int j=i+1; j<taille;j++) {
                                membres[j-1]=membres[j];
                                membres[j]=null;
                        }
                }
                return resultat;
        }
        ...
}
```

Programme 6 : *Redéfinition de méthodes en Java : classe ClanLimiteCapa*

```
public class ClanLimiteCapa extends Clan {
        private int capaMax;

        ...

        public void ajoutMembre(Personnage p) {
                System.out.println(taille());
                if(taille()<capaMax) {
                        super.ajoutMembre(p);
                }else {
                        System.out.println("capacité max atteinte");
                }
        }

        public Personnage ejecterUnMembre() {
                int taille=taille();
                Personnage result=null;
                if (taille()!=0) {
                result=membres[taille-1];
                membres[taille-1]=null;
                }
```

```
                return result;
        }
}
```

Programme 7 : *Redéfinition de méthodes en Java : classe ClanElitiste*

```
public class ClanElitiste extends Clan {
        private int seuilCapital;
        private int seuilPtsVie;

        ...
        public void ajoutMembre(Personnage p) {
                if(p.getNbPieces()>=seuilCapital&&p.getPtsVie()>=seuilPtsVie) {
                        super.ajoutMembre(p);
                }else {
                        System.out.println("pas assez fort ...");
                }
        }
        private Personnage plusFaibleMembre() {
                Personnage pmin=null;
                int taille=taille();
                if (taille!=0) {
                        pmin=membres[0];
                        for (int i=1;i<taille;i++) {
                                if (membres[i].getNbPieces()+membres[i].getPtsVie()<
                                pmin.getNbPieces()+pmin.getPtsVie()) {
                                        pmin=membres[i];
                                }
                        }
                }
                return pmin;
        }

        public Personnage ejecterUnMembre() {
                Personnage eject=plusFaibleMembre();
                if (eject!=null) {
                        int taille=taille();
                        boolean trouve=false;
                        for (int i=0;i<taille;i++) {
                                if (trouve) {
                                        membres[i-1]=membres[i];
                                }else {
                                        if (membres[i].equals(eject)) {
                                                trouve=true;
                                                membres[i]=null;
                                        }
                                }
                        }
                }
                return eject;
        }
}
```

Lors d'une spécialisation en Java, on fait appel à la méthode de la super-classe en utilisant le mot-clef `super`. Par exemple, dans la méthode ajoutMembre de la classe ClanElitiste, on trouve l'appel : `super.ajoutMembre(p);`. En effet, on souhaite appeler la méthode ajoutMembre de la classe Clan, celle que l'on est en train de spécialiser. Si l'on omet le mot-clef `super`, on appelle la méthode ajoutMembre de la classe ClanElitiste, ce qui mène à une récursivité infinie ...

L'appel à `super.ajoutMembre(p)` permet de ne pas replacer dans la méthode fille le code de la méthode mère, et permet une évolution simplifiée du code. Par exemple en cas d'évolution de la façon d'ajouter un membre dans un clan (par exemple en

vérifiant que p n'est pas déjà présent dans le clan), cette évolution ne sera répercutée qu'une fois, dans la méthode ajoutMembre de la classe Clan.

3.4 Les règles de Java régissant la redéfinition ; redéfinition vs surcharge

3.4.1 Redéfinition en Java

Tous les langages à objets ne font pas les mêmes choix quant aux règles régissant la redéfinition. Nous donnons ici celles de Java.

Pour qu'une méthode d'une sous-classe soit une redéfinition d'une méthode d'une super-classe :

— les 2 méthodes doivent porter le même nom ;
— la liste des paramètres des 2 méthodes doit être inchangée (hormis éventuellement sur le nom des paramètres), on parle de redéfinition invariante des paramètres ;
— le type de retour de la méthode de la sous-classe doit être le même que celui de la méthode de la super-classe, ou une spécialisation de ce type, on parle de redéfinition covariante du type de retour [2] ;
— la visibilité doit être la même pour les 2 méthodes, ou éventuellement être élargie dans la méthode de la sous-classe.

Nous omettons ici la règle sur les exceptions puisque nous ne les avons pas encore abordées.

Au moment de l'exécution (et non pas de la compilation), les langages à objets utilisent un mécanisme d'héritage (ou résolution de messages) qui consiste à résoudre dynamiquement l'envoi de message sur les objets (instances), c'est-à-dire à trouver et exécuter le code le plus spécifique correspondant au message.

3.4.2 Redéfinition vs surcharge en Java

Il est à noter qu'en Java, si une méthode d'une sous-classe porte le même nom qu'une méthode d'une super-classe, mais ne peut pas en être considérée comme une redéfinition (par exemple à cause de variations dans les types des paramètres), alors elle est considérée comme une surcharge de la méthode de la super-classe. Les surcharges sont, elles, résolues statiquement. Nous ne détaillerons pas ici finement les conséquences de cette différence entre surcharge et redéfinition.

3.4.3 Surcharge

La surcharge est plus généralement la possibilité laissée (ou pas) par un langage d'avoir plusieurs éléments de même nom. Tous les langages ne le permettent pas, Java le permet et l'encourage. Ainsi, nous avons vu que nous avions fréquemment plusieurs constructeurs dans une même classe. Tous ces constructeurs portent le même nom : on parle de surcharge de constructeurs. Il en est de même pour les opérations : plusieurs opérations peuvent porter le même nom tant qu'il est possible de les distinguer à la compilation : il doit être possible de déterminer dès la compilation laquelle des surcharges est appelée.

4 Classes et méthodes abstraites

Dans une hiérarchie de classes, plus une classe occupe un rang élevé, plus elle est générale donc plus elle est abstraite. On peut donc envisager de l'abstraire complètement en lui ôtant d'une part le rôle de génitrice (elle ne sera pas autorisée à créer des instances) et en lui permettant d'autre part de factoriser des structures et comportements (sans savoir exactement comment les faire) uniquement pour rendre service à sa sous-hiérarchie.

— Une classe abstraite est une classe qui ne peut pas être instanciée.
— Une méthode abstraite est une méthode dont on ne connaît pas le comportement à l'endroit où on la définit. Elle ne possède pas de corps.
— Une méthode abstraite est placée dans une classe qui doit être également abstraite .
— En revanche, une classe abstraite peut ne pas posséder de méthode abstraite.
— Une classe concrète sous-classe d'une classe abstraite doit obligatoirement redéfinir (de manière concrète) les méthodes abstraites héritées.
— Une classe abstraite sous-classe d'une classe abstraite peut ne pas redéfinir les méthodes abstraites héritées.

Illustrons ces notions sur notre exemple. Nous voulons disposer de 2 autres sortes de clans : les clans avec impôt forfaitaire (les membres du clan se voient tous imposés de la même somme), et les clans avec impôt proportionnel (les membres du clan se voient imposés d'une somme proportionnelle à leur nombre de pièces). Ces deux classes hériteront d'une classe ClanAvecImposition, qui permettra de factoriser ce qui est commun entre les 2 sortes de clans avec imposition. Toutefois, ClanAvecImposition ne sera pas instanciable, ce sera une classe abstraite.

4.1 Notation des classes et méthodes abstraites en UML

Pour indiquer qu'une classe ou une méthode est abstraite, en UML on appose la mention <<abstract>> ou on écrit leur nom en italique.

2. Notons que la redéfinition covariante du type de retour est arrivée en Java à sa version 5)

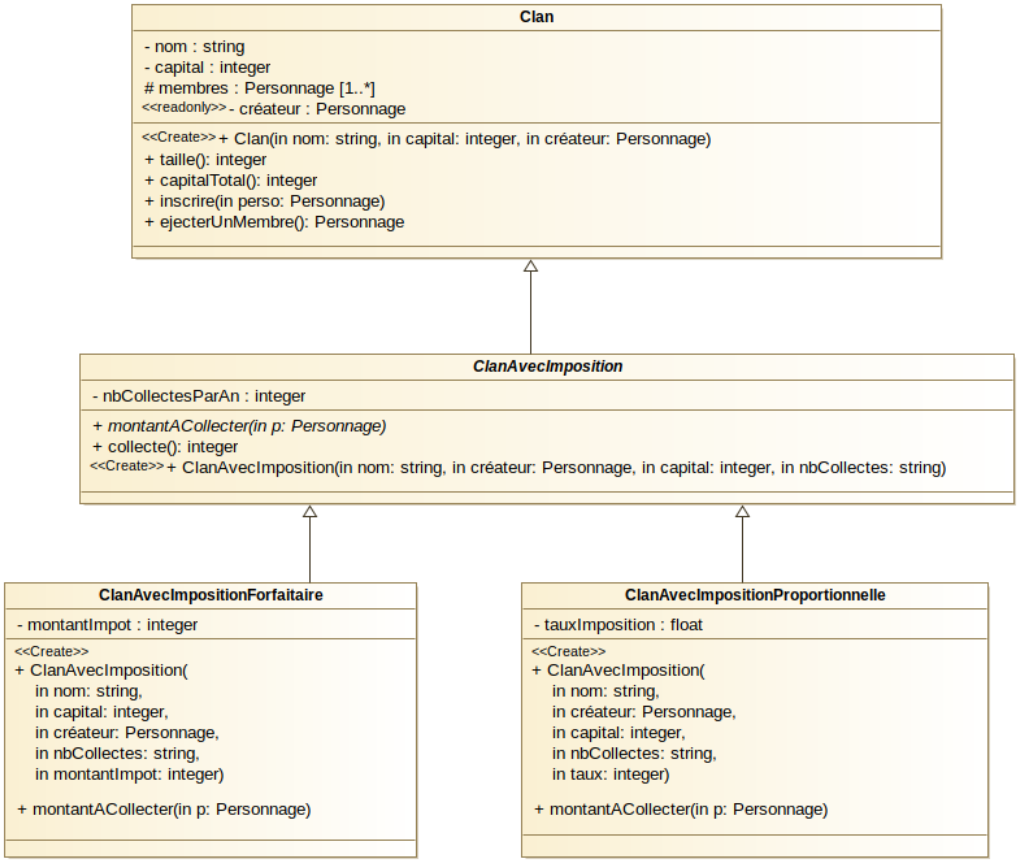

FIGURE 8 – Classes et méthodes abstraites

Ici la méthode montantACollecter (abstraite dans la classe ClanAvecImposition et concrète dans les classes filles) est en charge de calculer pour un personnage donné le montant qui est à collecter (le personnage est supposé appartenir au clan). Ce calcul ne peut pas être effectuer dans la classe ClanAvecImposition, puisqu'on n'y connaît pas le mécanisme d'imposition utilisé.

La méthode collecte est en charge de faire la collecte de l'impôt auprès de tous les membres du clan (elle retourne le montant total collecté). Cette méthode est concrète : elle utilisera la méthode montantACollecter pour calculer le montant à prélever sur chaque membre, puis prélèvera ce montant.

4.2 Classes et méthodes abstraites en Java

Programme 8 : *Classe et méthode abstraites en Java : classe ClanAvecImposition*

```
public abstract class ClanAvecImposition extends Clan {
        private int nbCollectesParAn;

        public ClanAvecImposition(String nom, int capital, Personnage créateur, int nbCollectesParAn)
        ↪  {
                super(nom, capital, créateur);
                this.nbCollectesParAn = nbCollectesParAn;
        }

        public abstract int montantACollecter(Personnage p); // méthode abstraite, pas de corps

        public int collecte() {
                int taille=taille();
                int totalCollecté=0;
                for (int i=0; i<taille;i++) {
                        int prelevement=montantACollecter(membres[i]);
```

```
                        membres[i].paiement(prelevement); // paiement est une méthode de Personnage
                        totalCollecté+=prelevement;
                }
                augmentationCapital(totalCollecté); // augmentationCapital est une méthode de Clan
                return totalCollecté;
        }
}
```

Programme 9 : *Classe et méthode abstraites en Java : classe ClanAvecImpositionProportionnelle*

```
public class ClanAvecImpositionProportionnelle extends ClanAvecImposition {
        private float tauxImposition;

        public ClanAvecImpositionProportionnelle(String nom, int capital, Personnage créateur, int
        ↪  nbCollectesParAn,
                        float tauxImposition) {
                super(nom, capital, créateur, nbCollectesParAn);
                this.tauxImposition = tauxImposition;
        }

        public int montantACollecter(Personnage p) {
                int montantImposable=p.getNbPieces();
                return Math.round(montantImposable*tauxImposition);// round est une méthode static de
                ↪  la classe Math qui arrondit à l'entier le plus proche
        }
}
```

Programme 10 : *Classe et méthode abstraites en Java : classe ClanAvecImpositionForfaitaire*

```
public class ClanAvecImpositionForfaitaire extends ClanAvecImposition {
        private int montantImpot;

        public ClanAvecImpositionForfaitaire(String nom, int capital, Personnage créateur, int
        ↪  nbCollectesParAn,
                        int montantImpot) {
                super(nom, capital, créateur, nbCollectesParAn);
                this.montantImpot = montantImpot;
        }

        public int montantACollecter(Personnage p) {
                // le prélèvement est plafonné par le nombre de pièces du personnage
                return Integer.min(montantImpot, p.getNbPieces());// min est une méthode static de la
                ↪  classe enveloppe Integer
        }
}
```

L'intérêt de définir une méthode abstraite est double : permettre au développeur de ne pas oublier de redéfinir une méthode qui a été définie `abstract` au niveau d'une des super-classes ; et permettre de mettre en œuvre le polymorphisme.

5 Polymorphisme

Au sens le plus général, le polymorphisme [3] est la capacité à se présenter sous différentes formes. Dans les approches à objets, le polymorphisme est utilisé en relation avec les méthodes : des méthodes de mêmes noms peuvent avoir des comportements différents, ou les variables : une variable d'un certain type peut référencer un objet de ce type ou d'un de ses sous-types.

5.1 Affectation polymorphe

Une référence sur un objet d'une sous-classe peut toujours être implicitement convertie en une référence sur un objet de la super-classe. Dans nos exemples de clans, on peut par exemple "ranger" un objet de type ClanElitiste dans une variable typée

3. Ceux qui ont fait du grec ancien reconnaîtront les racines grecques : poly (plusieurs) et morphe (forme) ; normalement ceux qui n'ont pas fait de grec ancien devraient s'y retrouver aussi ...

par Clan.

Programme 11 : *Affectation polymorphe en Java*

```
Clan clanSimple=new Clan("gobelins chauves", 187, popoki);
Clan petitClan=new ClanLimiteCapa("petit comité", 542, ronflex, 6);
Clan clanélite=new ClanElitiste("toujours plus", 1865434, gandalf, 200, 300);
Clan c1=petitClan;
//ClanElitiste c2=clan; // non : mauvais sens
// ClanElitiste c3=petitClan; // non : classes soeurs
```

L'opération inverse (cast-down), par exemple ici quelque chose comme `ClanElitiste c2=clan` est possible mais uniquement avec un changement de type (cast) explicite ; ceci est à réaliser avec précaution et uniquement en cas de nécessité absolue.

Programme 12 : *Down-cast en Java*

```
Clan clanSimple=new Clan("gobelins chauves", 187, popoki);
ClanElitiste c2=(ClanElitiste) clan;
```

Pour vérifier qu'une instance appartient bien à une classe précise d'une hiérarchie, on peut utiliser l'opérateur `instanceof`.

Programme 13 : *Test de type en Java*

```
Clan clanSimple=new Clan("gobelins chauves", 187, popoki);
Clan petitClan=new ClanLimiteCapa("petit comité", 542, ronflex, 6);
Clan clanélite=new ClanElitiste("toujours plus", 1865434, gandalf, 200, 300);

System.out.println(clanSimple instanceof Clan); // true
System.out.println(petitClan instanceof Clan); // true
System.out.println(petitClan instanceof ClanLimiteCapa); // true
System.out.println(clanSimple instanceof ClanLimiteCapa); // false
```

5.2 Polymorphisme des opérations

Le fait que l'on puisse définir dans plusieurs classes des méthodes de même nom revient donc à désigner plusieurs formes de traitement derrière ces méthodes. Le code de la méthode qui sera réellement exécuté n'est donc pas figé, un appel de message autorisé à la compilation donnera des résultats différents à l'exécution car le langage retrouvera selon l'objet et la classe à laquelle il doit son existence le code à exécuter (on parle de `liaison dynamique`). La capacité pour un message de correspondre à plusieurs formes de traitements est appelé `polymorphisme`. Nous n'abordons ici que le polymorphisme de redéfinition et pas le polymorphisme de surcharge.

Le polymorphisme de redéfinition fonctionne grâce à la liaison dynamique qui cherche à l'exécution la méthode la plus spécifique pour résoudre un envoi de message. Cette méthode plus spécifique se trouve soit dans la classe de l'instance (classe déterminée par le type dynamique suivant le `new`), soit dans une super-classe, en remontant à partir de la classe de l'instance et en utilisant la première méthode trouvée répondant au message.

Programme 14 : *Polymorphisme de redéfinition en Java*

```
Personnage popoki=new Personnage("Popoki le chat", 100, 1000);
Personnage ronflex=new Personnage("Ronflex", 150, 1500);

Clan clanSimple=new Clan("gobelins chauves", 187, popoki);
Clan petitClan=new ClanLimiteCapa("petit comité", 542, ronflex, 6);

clanSimple.ajoutMembre(popoki); // méthode ajoutMembre de Clan
petitClan.ajoutMembre(ronflex); // méthode ajoutMembre de ClanLimiteCapa (qui elle même appelle
↪  ajoutMembre de Clan)
int t=petitClan.taille();// méthode taille de Clan
```

Dans l'exemple précédent, intéressons-nous à la résolution des méthodes appelées au niveau des lignes : `clanSimple.ajoutMembre(popoki);` ; `petitClan.ajoutMembre(ronflex);` et `petitClan.taille();`.

Lors de la compilation, le compilateur s'assure que tous les appels pourront bien être résolus à l'exécution. Ainsi, le compilateur s'assure qu'il sera bien possible d'appeler une méthode ajoutMembre avec pour receveur clanSimple. Pour cela, le compilateur

inspecte le type statique de clanSimple, c'est-à-dire celui utilisé lors de la déclaration, soit ici : Clan. Le compilateur regarde alors s'il existe une méthode ajoutMembre dans la classe Clan, qui correspond du point de vue de la signature (ici : pas de type de retour, et un seul paramètre de type Personnage). Puisque cette méthode existe, le compilateur est satisfait. Toutefois, la liaison entre le site d'appel et la méthode appelée n'est pas réalisée à la compilation, cette liaison n'aura lieu qu'à l'exécution, d'où le terme de liaison dynamique. Le compilateur procède de la même façon pour la ligne `petitClan.ajoutMembre(ronflex);`. Il inspecte le type statique de petitClan, c'est un Clan, et puisqu'il existe une méthode ajoutMembre satisfaisante dans la classe Clan, le compilateur est satisfait. Il en est de même pour l'instruction suivante. La compilation peut déclencher une erreur. Par exemple, imaginons que la classe ClanLimiteCapa contienne une méthode de signature : `public void setCapaMax(int capa)`. Cette méthode n'existe pas dans la superclasse Clan. La compilation de la ligne : `petitClan.setCapaMax(13);` provoquera une erreur de compilation, car le type statique de petitClan est Clan, et Clan ne possède pas de méthode setCapaMax, le compilateur ne peut donc pas apporter de garantie de résolution de cette méthode à l'exécution, et provoque donc une erreur.

Ensuite, lors de l'exécution, il y aura résolution des méthodes à appeler. Cette résolution ne provoquera pas d'erreur, elle est garantie par la phase de compilation. Ainsi, l'interpréteur va chercher à exécuter la ligne `clanSimple.ajoutMembre(popoki);`. Pour cela, il doit déterminer quelle méthode ajoutMembre est à appeler. C'est cette fois-ci le type dynamique de clanSimple qui est pris en compte, c'est-à-dire celui qui a été utilisé lors de la construction de l'objet clanSimple. Ici clanSimple a été construit avec l'appel `new Clan("gobelins chauves", 187, popoki)` donc son type dynamique est Clan. Une méthode ajoutMembre est trouvée dans la classe Clan, et c'est cette méthode qui sera donc exécutée. L'interpréteur peut alors s'intéresser à l'instruction suivante : `petitClan.ajoutMembre(ronflex);`. Le type dynamique de petitClan est ClanLimiteCapa, donc la méthode ajoutMembre est cherchée dans la classe ClanLimiteCapa. Elle y est trouvée, donc c'est cette méthode qui sera exécutée. Regardons maintenant l'instruction suivante : `int t=petitClan.taille();`. Encore une fois, l'interpréteur étudie le type dynamique de petitClan, ce type est ClanLimiteCapa. La méthode taille est donc cherchée dans la classe ClanLimiteCapa. Elle n'y est pas trouvée. Elle est ensuite cherchée dans la super-classe de ClanLimiteCapa (i.e. Clan) et y est trouvée, c'est cette méthode qui sera alors exécutée. C'est ce mécanisme de recherche de méthode à partir de la classe correspondant au type dynamique de l'objet qui garantit que lors d'une liaison dynamique la méthode appelée est la plus spécifique.

Attention, ce que nous venons de voir fonctionne pour les redéfinitions de méthodes, mais pas pour les surcharges. En effet, le choix entre 2 surcharges est lui réalisé statiquement, c'est-à-dire à la compilation, en prenant en compte les types statiques.

6 Discriminants et contraintes

Les relations de spécialisation/généralisation peuvent être décrites avec plus de précision par deux sortes de description UML : les contraintes et les discriminants.

Discriminant Les discriminants sont les critères utilisés pour classer les objets dans des sous-classes. Ils étiquettent ainsi les relations de spécialisation et doivent correspondre à une classe ou une énumération du modèle.

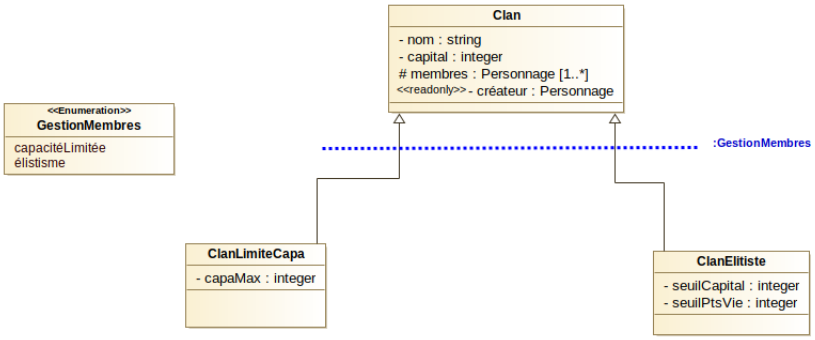

FIGURE 9 – Discriminants

Contraintes Les contraintes décrivent la relation entre un ensemble de sous-classes et leur super-classe en considérant le point de vue des extensions (ensemble d'instances des classes).

Il en existe quatre :

- `incomplete` (`incomplet`) : l'union des extensions des sous-classes est strictement incluse dans l'extension de la super-classe ; par exemple il existe des clans qui ne sont ni à limite de capacité, ni élitistes ;
- `complete` (`complet`) : l'union des extensions des sous-classes est égale à l'extension de la super-classe ;
- `disjoint` : les extensions des sous-classes sont d'intersection vide ; par exemple aucun clan n'est à limite de capacité et élitiste ;
- `overlapping` (`chevauchement`) : les extensions des sous-classes se rencontrent.

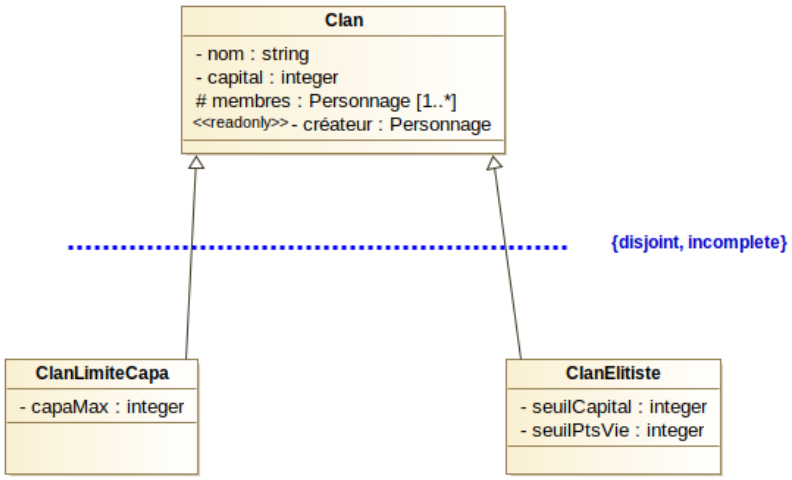

FIGURE 10 – Contraintes

7 Java : la classe Object

La classe Object est la racine implicite de toute hiérarchie de classe en Java. Cette classe contient plusieurs méthodes, et nous avons déjà évoqué deux d'entre elles, nous y revenons ici.

La méthode toString donne une représentation d'objet sous forme de chaîne de caractères. Cette méthode est appelée implicitement dès qu'un objet doit être converti en chaîne de caractère, notamment si l'on affiche un objet. La méthode toString est documentée ainsi dans la documentation Java :

> Returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.
>
> The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of :
>
> ```
> getClass().getName() + '@' + Integer.toHexString(hashCode())
> ```

Quand on ne redéfinit pas dans une classe la méthode toString, c'est donc celle définie dans Object qui est utilisée. Elle affiche le nom de la classe (préfixé par ses éventuels packages), suivi d'un @ suivi du hashCode de l'objet en hexadécimal. Le hashCode est calculé avec la méthode hashCode de la classe Object, nous ne la détaillerons pas ici.

Même si la méthode toString de la classe Object retourne une chaîne compréhensible par un programmeur (le programmeur peur regarder si les codes de 2 objets sont identiques ou différents par exemple), il est conseillé (comme indiqué ci-dessus) de redéfinir la méthode toString dans chacune des classes, pour avoir une chaîne décrivant bien l'objet (pas avec un hashCode mais des valeurs d'attributs par exemple).

La méthode equals permet de comparer un objet à un autre pris en paramètre, pour déterminer s'ils sont égaux. La méthode equals est documentée ainsi dans la documentation Java :

> ```
> public boolean equals(Object obj)
> ```
>
> Indicates whether some other object is "equal to" this one.
>
> The equals method implements an equivalence relation on non-null object references :
>
> — It is reflexive : for any non-null reference value x, x.equals(x) should return true.
> — It is symmetric : for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
> — It is transitive : for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
> — It is consistent : for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
> — For any non-null reference value x, x.equals(null) should return false.

> The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).
>
> Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

Comme on le voit, la méthode equals implémente globalement de l'égalité de référence. Si l'on souhaite une égalité plus sémantique (une égalité de valeur par exemple), il faut redéfinir la méthode equals. Comme indiqué, il faut en général dans ce cas implémenter aussi la méthode hashCode.

La classe String implémente une méthode equals documentée ainsi :

> Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

La méthode equals de la classe String fait donc de l'égalité de valeur.

8 En bref

- Le mécanisme d'héritage permet de mettre en place des relations de spécialisation/généralisation sur les classes d'objets.
- Quand une classe fille hérite d'une classe mère, elle hérite :
  - de ses attributs, qu'elle peut utiliser si leur visibilité le lui permet
  - de ses opérations. Lors de l'héritage d'opération, la classe fille peut décider de masquer le comportement hérité (en remplaçant la méthode héritée par une méthode ayant un autre comportement), de spécialiser le comportement hérité (en remplaçant la méthode héritée par une méthode qui appelle a méthode méthode, et ajoute un comportement supplémentaire), ou encore de garder telle quelle l'opération (en ne faisant rien !).
- En présence de polymorphisme, la résolution entre plusieurs redéfinitions de méthodes est réalisée à l'exécution : c'est la liaison dynamique.
- Grâce à la mise de en place de la relation de spécialisation/généralisation, on peut éviter des duplications de code : duplication d'attributs ou de comportements.
- Une classe mère peut être abstraite si on ne souhaite pas qu'elle puisse être instanciée directement ou si elle possède au moins une méthode abstraite (dont le comportement ne peut pas être décrit à ce niveau d'abstraction).