1ère année Master GLSD
Département d'Informatique
Université de Biskra
Année 2020/2021

Object Constraint Language (OCL)

Dr. Mohamed Lamine Kerdoudi
Email: l.kerdoudi@univ-biskra.dz

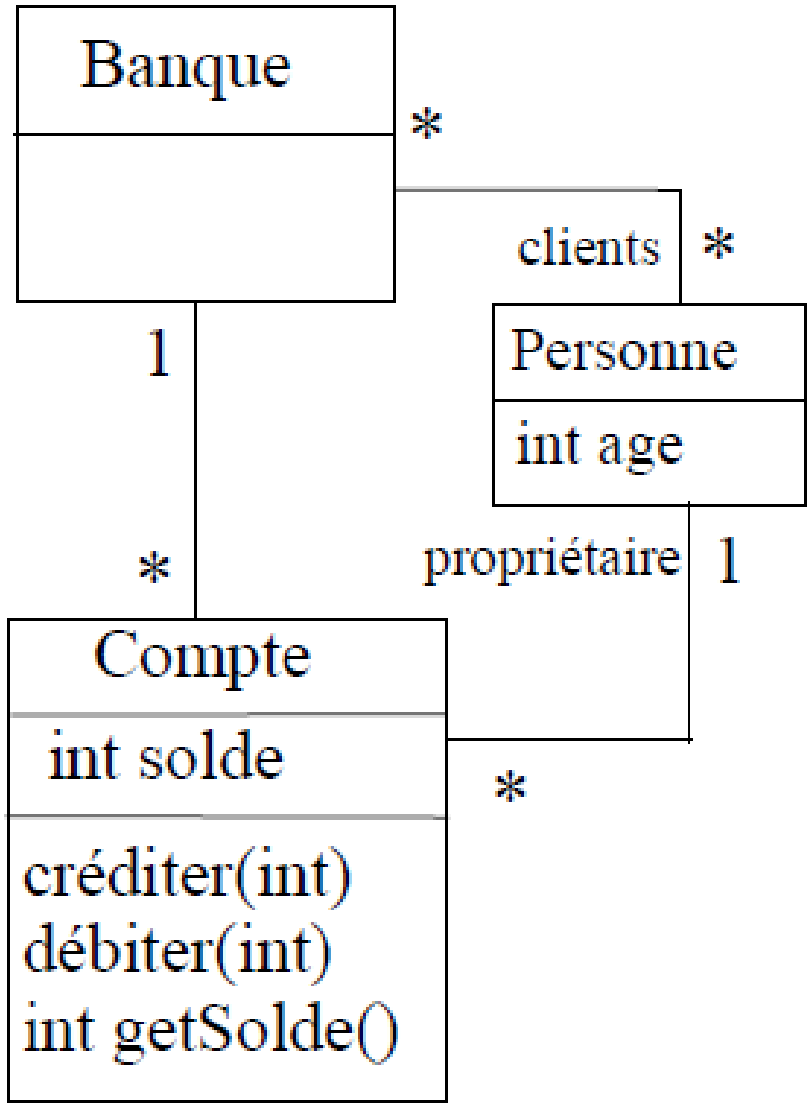
Introduction

- OCL est une norme de l'OMG
  - Version : 2.4 (2014)
  - Peut s'appliquer sur tout type de modèle, indépendant d'un langage de modélisation donné
- OCL est un langage formel, basé sur la logique des prédicats du premier ordre
- Il permet d'annoter les diagrammes UML en permettant notamment l'expression de contraintes.

Exemple d'application bancaire

- **Application bancaire:**
  - Des comptes bancaires
  - Des clients
  - Des banques
- **Spécification**
  - Un compte doit avoir un solde toujours positif
  - Un client peut posséder plusieurs comptes
  - Un client peut être client de plusieurs banques
  - Un client d'une banque possède au moins un compte dans cette banque
  - Une banque gère plusieurs comptes
  - Une banque possède plusieurs clients

Diagramme de classe

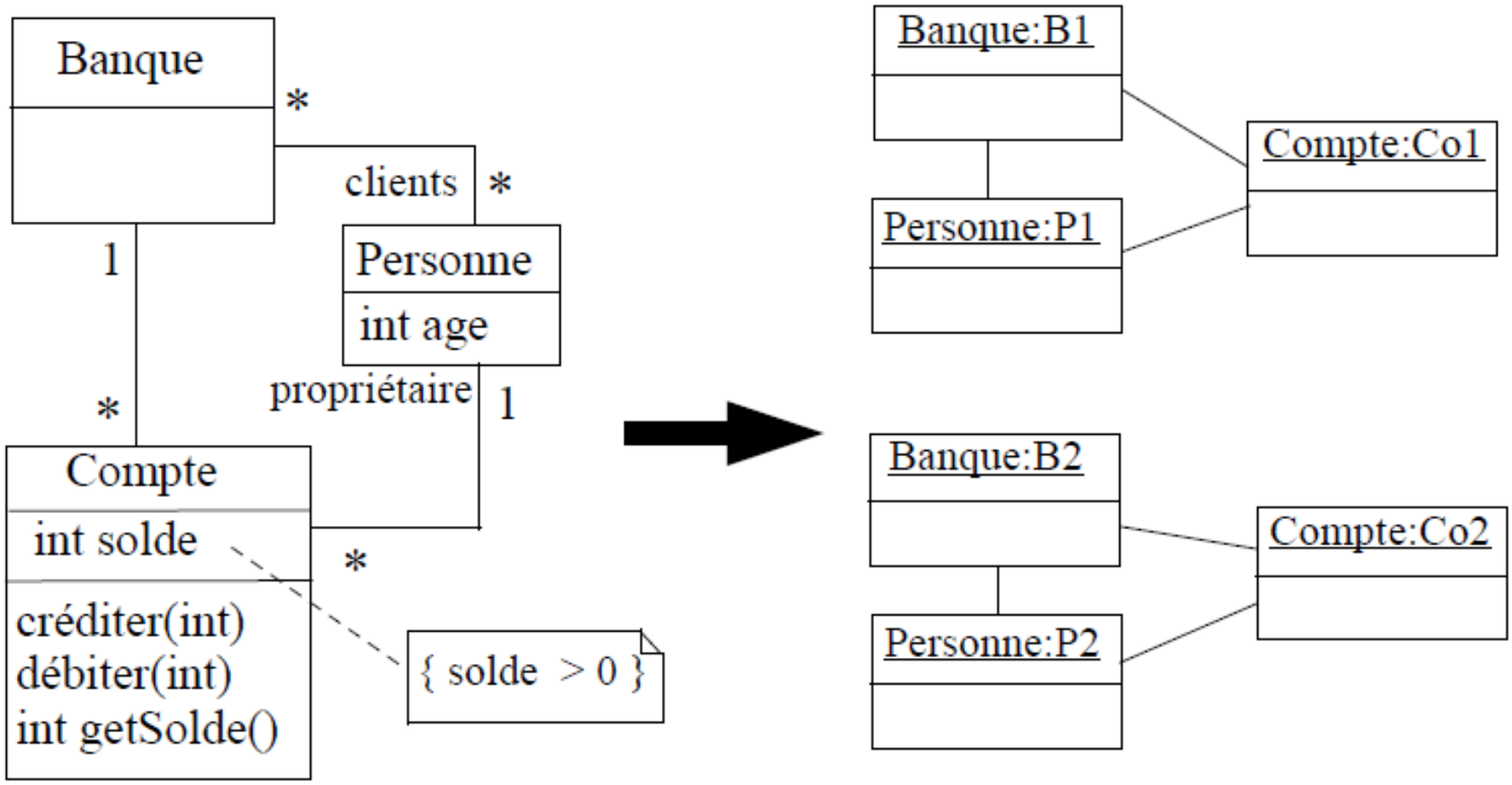


Problème : Manque de précision

- Le diagramme de classe ne permet pas d'exprimer tout ce qui est défini dans la spécification informelle
- **Exemple**
  - Le solde d'un compte doit toujours être positif

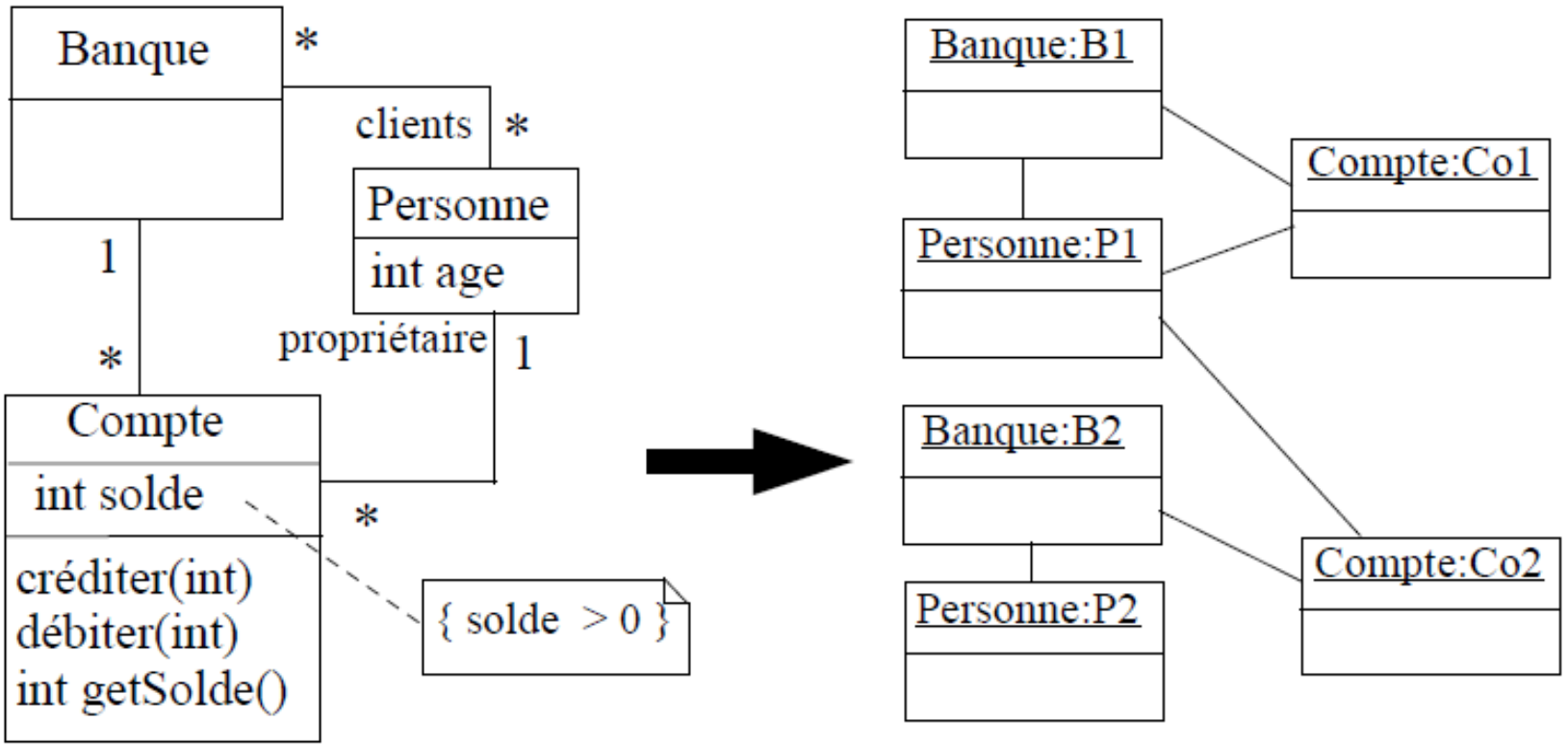
    ➔ Il faut ajouter d'une contrainte sur cet attribut
  - Le diagramme de classe permet-il de détailler toutes les contraintes sur les relations entre les classes ?

Diagramme d'instances 1



- Diagramme d'instances **valides** vis-à-vis du diagramme de classe et de **la spécification attendue**

Diagramme d'instance 2



- Diagramme d'instance **valide** vis-à-vis du **diagramme de classe mais ne respecte pas la spécification attendue**
  - Une personne a un compte dans une banque où elle n'est pas cliente
  - Une personne est cliente d'une banque mais sans y avoir de compte

Diagrammes UML insuffisants

- Pour spécifier complètement une application
  - Diagrammes UML seuls sont généralement insuffisants
  - Nécessité de rajouter des contraintes
- Comment exprimer ces contraintes ?
  - Langue naturelle mais manque de précision, compréhension pouvant être ambiguë
  - Langage formel avec sémantique précise : **par exemple OCL**
- **OCL : Object Constraint Language**
  - Langage de contraintes orienté-objet,
  - **Formel** (mais « simple » à utiliser) avec une syntaxe, une grammaire, une sémantique (manipulable par un outil)
  - S'applique entre autres sur les diagrammes UML

Objectif du langage OCL

- Accompagner les diagrammes UML de descriptions :
  - précises
  - non ambiguës
- Eviter les désavantages des langages formels traditionnels qui sont peu utilisables par les utilisateurs et les concepteurs qui ne sont pas rompus à l'exercice des mathématiques :
  - rester facile à écrire ...
  - et facile à lire
- Dans le cadre de **l'ingénierie des modèles**:
  - la précision du langage OCL est nécessaire pour pouvoir
    - traduire automatiquement les contraintes OCL dans un langage de programmation
    - afin de les vérifier pendant l'exécution d'un programme.

Principe

Notion de contrainte: une contrainte est une **expression à valeur booléenne** (attacher à n'importe quel élément UML).

➔ Elle indique en général une restriction ou donne des informations complémentaires sur un modèle.

- **Langage déclaratif:** les contraintes ne sont pas opérationnelles.
  - On ne décrit pas le comportement à adopter si une contrainte n'est pas respectée.
- **Langage sans effet de bord:** les instances ne sont pas modifiées par les contraintes.
  - Une expression OCL décrit une contrainte à respecter et non pas le « code » d'une méthode

Utilisation

OCL permet principalement d'exprimer **deux types** de contraintes sur l'état d'**un objet** ou d'un **ensemble d'objets**

- Des **invariants** qui doivent être respectés en permanence
- Des **pré** et **post-conditions** pour une opération
  - **Précondition** : doit être vérifiée avant l'exécution
  - **Postcondition** : doit être vérifiée après l'exécution

On peut également utiliser OCL pour exprimer :

- contraintes sur la valeur retournée par une opération ou une méthode
- règles de dérivation des attributs
- expression des gardes (conditions dans les diagrammes dynamiques)

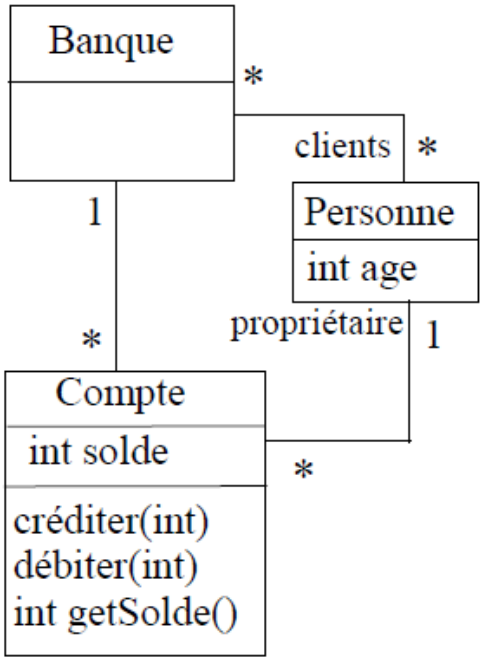
Les principaux concepts d'OCL

Syntaxe d'une contrainte OCL:

context monContexte <**stéréotype**>: Expression de la contrainte

- Une contrainte OCL est liée à un **contexte** qui représente le **type**, **l'opération** ou **l'attribut** auquel la contrainte se rapporte.

Exemple:



context Compte

inv: solde > 0

Contexte et Stéréotype

context monContexte <**stéréotype**>: Expression de la contrainte

- **context** Compte
- L'expression OCL s'applique à la classe **Compte**, c'est-à-dire à: **toutes les instances de cette classe**

Le **stéréotype** peut prendre les valeurs suivantes :

- **inv** invariant de classe
- **pre** précondition
- **post** postcondition
- **body** indique le résultat d'une opération **query**
- **init** indique la valeur initiale d'un attribut
- **derive** indique la valeur dérivée d'un attribut

Invariants

context monContexte <**stéréotype**>: Expression de la contrainte

- Un **invariant** exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence

Exemple

context Compte

inv: solde > 0

Pour toutes les instances de la classe **Compte**, l'attribut **solde** doit toujours être **positif**

Pré et post-conditions

context monContexte **<stéréotype>:** Expression de la contrainte

Pour spécifier une opération on a besoin de (**pre** et **post**) :

- **Pré-condition** : état qui doit être respecté avant l'appel de l'opération
- **Post-condition** : état qui doit être respecté après l'appel de l'opération
  - Dans la **post-condition**, deux éléments particuliers sont utilisables
    - Pseudo-attribut **result** : référence la valeur retournée par l'opération
    - **mon_attribut@pre** : référence la valeur de **mon_attribut** avant l'appel de l'opération
- **Syntaxe pour préciser la signature de l'opération en OCL:**

  context ma_classe::mon_op(liste_param) : type_retour

Exemple 1

context Compte::débiter(somme : Integer)
pre: somme > 0
post: solde = solde**@pre** – somme

- La somme à débiter doit être positive pour que l'appel de l'opération soit valide
- **Après** l'exécution de l'opération, l'attribut **solde** doit avoir pour valeur sa valeur avant l'appel à laquelle a été soustrait la somme passée en paramètre

Exemple 2

context Compte::getSolde() : Integer
post: **result** = solde

- Le résultat retourné doit être le solde courant

❑ Attention

- On ne décrit pas comment l'opération est réalisée mais des contraintes sur l'état avant et après son exécution

Exemple 3

context Personne **inv**:
(age <= 140) and (age >=0) - - l'âge est compris entre 0 et 140 ans

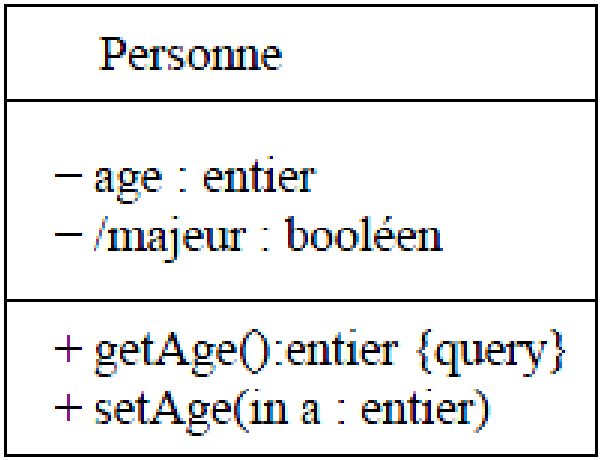
On peut placer des commentaires dans les expressions OCL, ils débutent par deux tirets et se prolongent jusqu'à la fin de la ligne.

context Personne::setAge(a :entier)
pre: (a <= 140) and (a >=0) and (a >= age)
post: age = a - - on peut écrire également a=age

context Personne::getAge() :entier
body: age -- résultat d'une opération **query**

context Personne::**age** :entier
init: 0 -- valeur initiale de **age** doit être 0

context Personne::**majeur** :booléen
derive: age>=18



Nommage de la contrainte, Accès aux objets, navigation

- **Exemple de nommage de la contrainte:**

context Personne **inv** **ageBorné** :
(age <= 140) and (age >=0) - - l'âge ne peut dépasser 140 ans

- Utilisation du mot-clef **self** pour désigner l'objet:

➔ offre la navigation (accès) à l'attribut.

Exemple :

context Personne **inv**:
(**self**.age <= 140) and (**self**.age >=0)

- Utilisation d'un **nom d'instance formel** :

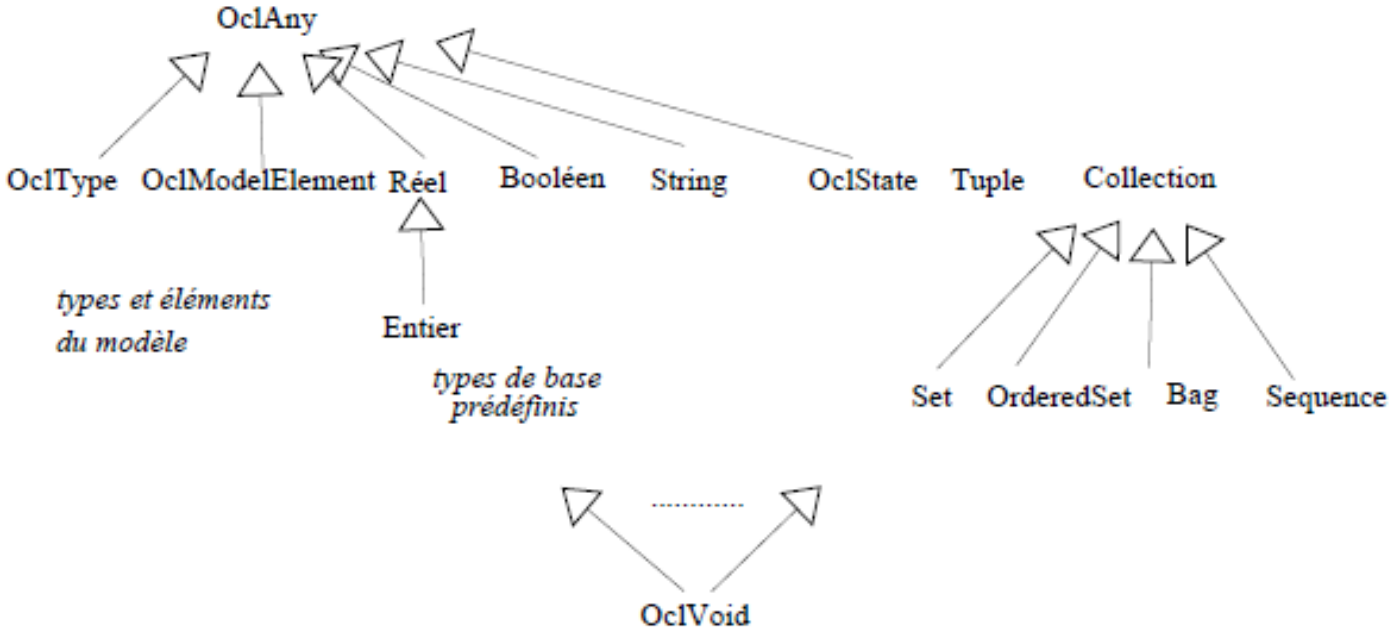
context **p** :Personne **inv**:

(**p**.age <= 140) and (**p**.age >=0)

Question 1

- Ajoutez un attribut **mère** de type **Personne** dans la classe Personne.
- Ecrivez une contrainte précisant que la mère d'une personne ne peut être cette personne elle-même et que l'âge de la mère doit être supérieur à celui de la personne.

Hiérarchie des types en OCL



1. Types de base

Les types de base prédéfinis sont les suivants.

- Entier (Integer)
- Réel (Real)
- String
- Booléen (Boolean)

2. Quelques types spéciaux

- OclModelElement (énumération des éléments du modèle)
- OclType (énumération des types du modèle)
- OclAny (tout type autre que Tuple et Collection)
- OclState (pour les diagrammes d'états)
- OclVoid sous-type de tous les types

Types de base en OCL

- **Entier**

  opérateurs = <> + - * / abs div mod max min < > <= >=

  L'opérateur - est unaire ou binaire

- **Réel**

  opérateurs = <> + - * / abs floor round max min < > <= >=

  L'opérateur - est unaire ou binaire

- **String**

  opérateurs = **size**() **concat**(String) **toUpper**() **toLower**()

  substring(Entier, Entier)

  Les chaînes de caractères constantes s'écrivent entre deux simples quotes : 'voici une chaîne'

- **Booléen**

  opérateurs = or xor and not

  b1 **implies** b2

  if b **then** expression1 **else** expression2 **endif**

Types de base en OCL

- Quelque exemples d'utilisation des **opérateurs booléens**:

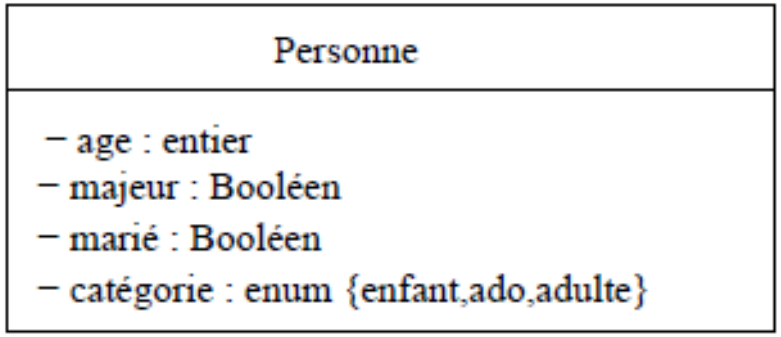
context Personne **inv**:

marié implies majeur

context Personne **inv**:

if age >=18 **then** majeur=vrai

else majeur=faux **endif**



- Cette dernière contrainte peut être **plus concise :**

context Personne **inv**:

majeur = age >=18

- Nous l'avons même déjà écrite avec **derive**:

context Personne::majeur :booléen

derive: age>=18

Types énumérés

- Leurs valeurs apparaissent précédées de **#**
- Par exemple, pour donner un invariant supplémentaire pour la figure précédente, nous pourrions écrire :

context Personne **inv**:

if age <=12 **then** catégorie =**#**enfant

else if age <=18 **then** catégorie =**#**ado

else catégorie=**#**adulte

endif

endif

Question 2

- Peut-on écrire cet invariant avec **derive? Comment**?

Types des modèles

- Les types des modèles utilisables en OCL sont les « classificateurs »:
  - les classes,
  - les interfaces,
  - les associations.
- On peut écrire en particulier des expressions impliquant des objets dont les types sont reliés par une relation de spécialisation, grâce aux **opérateurs** suivants:

❑ **oclAsType(t)** (conversion **ascendante** ou **descendante** de **type** vers **t**);

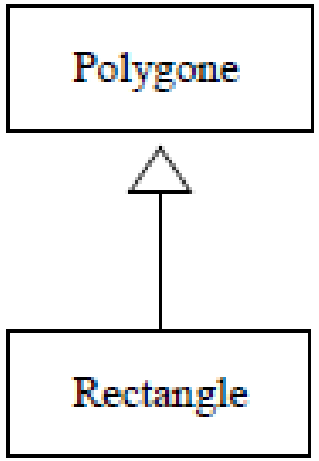
- ➢ la conversion ascendante sert pour l'accès à une **propriété redéfinie** ;
- ➢ la conversion descendante sert pour l'accès à **une nouvelle propriété**.

❑ **oclIsTypeOf(t)** retourne **true** si type de **self** est le même comme **t**

❑ **oclIsKindOf(t)** (retourne **true** si le type de **self** correspond au type ou supertype de **t**)

Types des modèles

- Dans le cadre de la figure suivante, nous pourrions écrire par exemple les expressions (r est supposé désigner une instance de Rectangle) :



- **p.oclAsType**(Rectangle)
- **r.oclIsTypeOf**(Rectangle) (vrai)
- **r.oclIsKindOf**(Polygone) (vrai)

p: Polygone

r: Rectangle

Question 3

- En supposant l'existence d'un attribut hauteur dans la classe Rectangle et d'une méthode hauteur():Réel dans Polygone,
- Ecrivez une contrainte dans Polygone disant que le résultat de hauteur():Réel vaut hauteur pour les polygones qui sont des **rectangles**, sinon **0**.
- Remarquez que c'est un exemple de mauvaise conception objet mais une excellente illustration des opérateurs présentés ci-dessus !

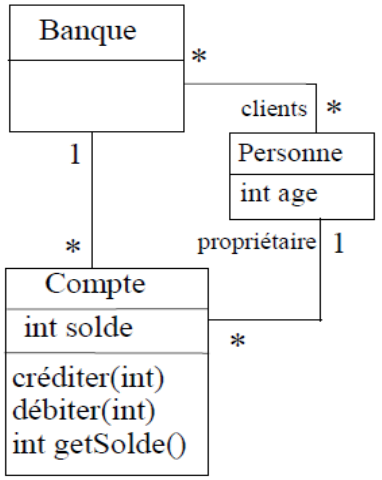
Navigation dans les modèles

- Dans une contrainte OCL associée à un objet, on peut
  - Accéder à l'état interne de cet objet (ses attributs)
  - Naviguer dans le diagramme : accéder de manière transitive à tous les objets (et leur état) avec qui il est en relation
- Nommage des éléments pour y accéder
  - Attributs ou paramètres d'une opération : utilise leur nom directement
  - Objet(s) en association : on utilise au choix:
    - Le nom de la classe associée (avec la première lettre en minuscule), à condition qu'il n'y ait pas ambiguïté
    - Le nom de l'association si elle nommée
    - Le nom du rôle d'association du coté de la classe vers laquelle on navigue s'il est nommé
- La navigation retourne
  - Si **cardinalité de 1** pour une association : **un objet**
  - Si **cardinalité > 1** : une **collection** d'objets

Exemple de navigation dans les modèles

Exemple 1: Dans contexte de la classe **Compte**

solde, banque , propriétaire , banque.clients, banque.clients.age



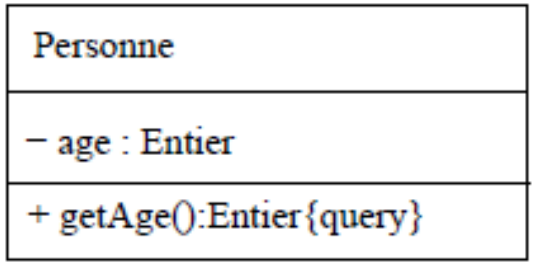
- solde : attribut référencé directement
- banque : objet de la classe Banque (référence via le nom de la classe) associé au compte
- propriétaire : objet de la classe Personne (référence via le nom de rôle d'association) associée au compte
- banque.clients : ensemble des clients de la banque associée au compte (référence par transitivité)
- banque.clients.age : ensemble des âges de tous les clients de la banque associée au compte
- Le propriétaire d'un compte doit avoir plus de 18 ans

  context Compte

  inv: self.propriétaire.age >= 18

Exemple de navigation dans les modèles

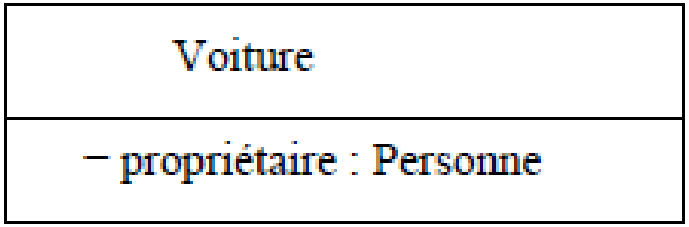
Exemple 2: Dans le contexte de classe **Voiture**,



1. Accès aux attributs

context Voiture **inv** propriétaireMajeur **:**

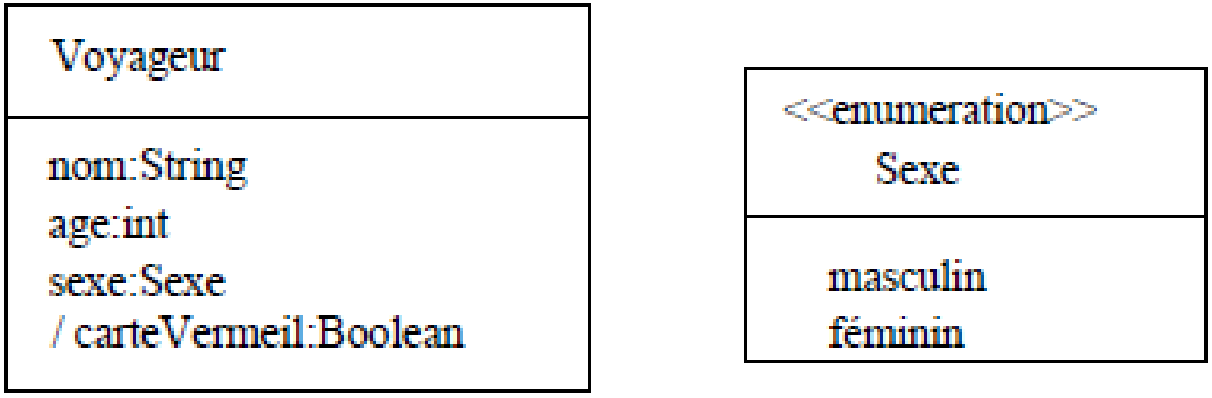
self.propriétaire.age >= 18



Exemple de navigation dans les modèles

Question 4

- Ecrivez pour le diagramme suivant la contrainte qui caractérise l'attribut dérivé carteVermeil.
- Un voyageur a droit à la carte vermeil si c'est une femme de plus de 60 ans ou un homme de plus de 65 ans.



Exemple de navigation dans les modèles

Exemple 3: Dans le contexte de classe **Voiture**,

2. Accès aux opérations query

Il se fait avec une notation identique (utilisation du '.').

Pour la classe Voiture, on peut ainsi réécrire la contrainte propriétaireMajeur.

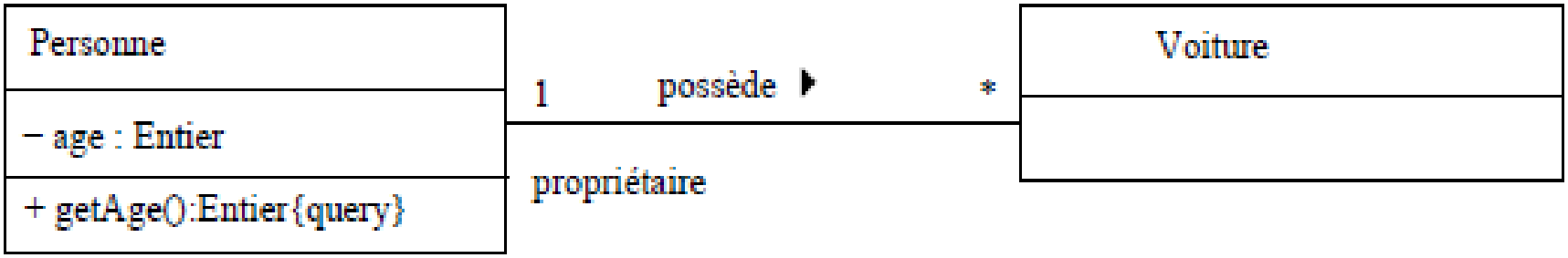
context Voiture **inv**:

self.propriétaire.getAge() >= 18

Exemple de navigation dans les modèles

3. Accès aux extrémités d'associations

❑ **Exemple 4 :** Pour la classe Voiture de la figure suivantes, on peut ainsi réécrire la contrainte propriétaireMajeur.



❑ **context** Voiture **inv:**

self.**propriétaire**.age >= 18

❑ **context** Voiture **inv:**

self.**personne**.age >= 18

Question 5

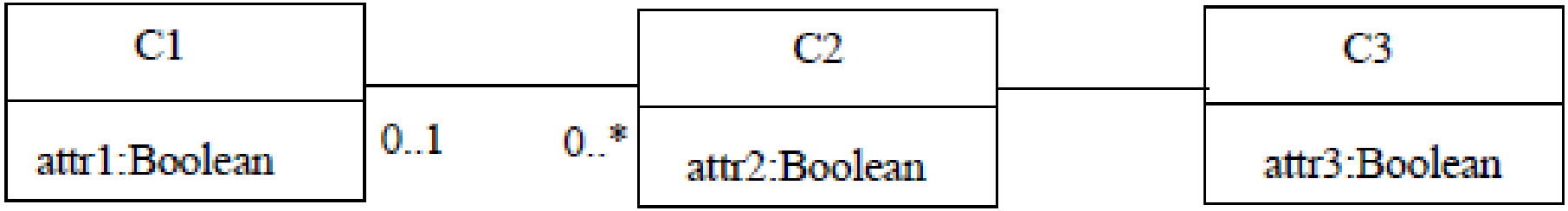
Soient deux contraintes associées au diagramme suivant:

- **context** C1 **inv**:

  c2.attr2=c2.c3.attr3

- **context** C2 **inv**:

  attr2=c3.attr3



Pensez-vous que les deux contraintes soient équivalentes ?

Exemple de navigation dans les modèles

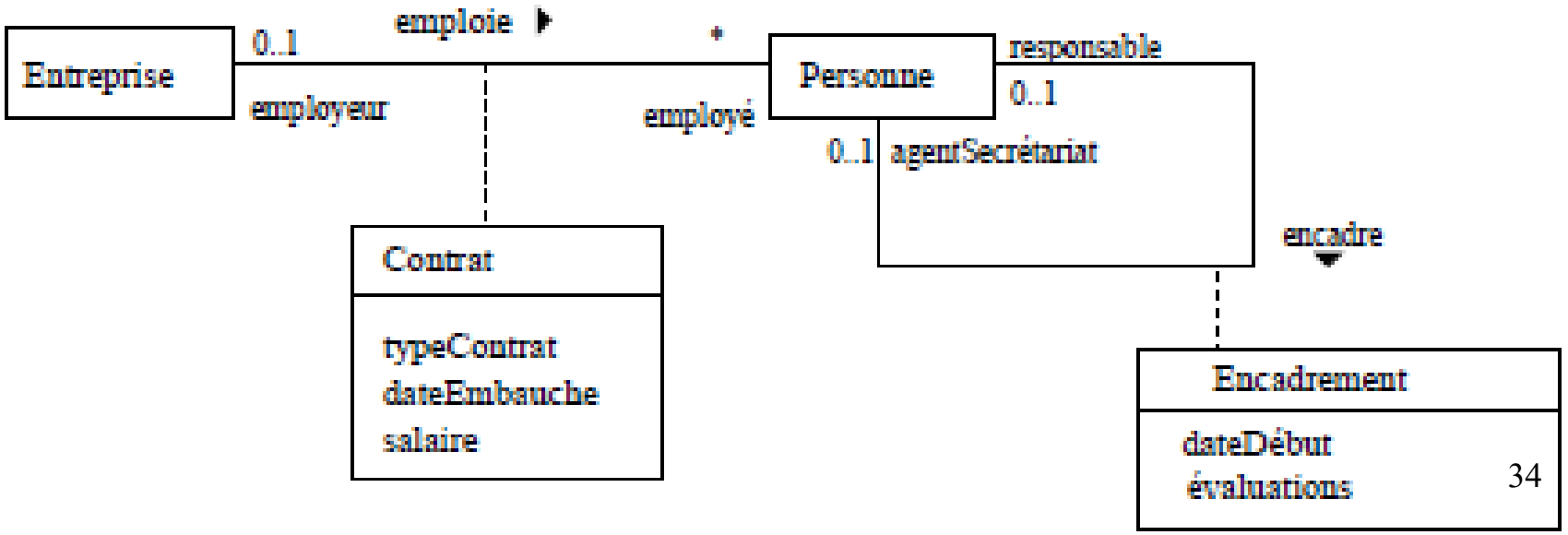
4. Navigation vers les classes association

Pour naviguer vers une classe association, on utilise le nom de la classe (en mettant le premier caractère en minuscule).

Exemple 5

context p :Personne **inv**:

p.contrat.salaire >= 0



Exemple de navigation dans les modèles

- On peut également préciser le nom de rôle opposé (c'est même obligatoire pour une association réflexive) :

context p :Personne **inv**:

p.contrat[employeur].salaire >= 0

- Pour naviguer depuis une classe association, on utilise les **noms des rôles** ou **de classes.**

- La navigation depuis une classe association vers un rôle ne peut donner **qu'un objet.**

- **context c :Contrat inv**:

  c.employé.age >= 16

- Dans cet exemple, **c** est un lien, qui associe par définition une seule entreprise à un seul employé.

4. Navigation vers les classes association

Question 6

Dans la même figure précédente, exprimez par une contrainte :

- le fait que le salaire d'un agent de secrétariat est inférieur à celui de son responsable ;
- un agent de secrétariat a un type de contrat "agentAdministratif" ;
- un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers) ;
- écrire la dernière contrainte dans le contexte **Personne**.

Eléments du langage

1. Définition de variables et d'opérations

a) Définition de variables

On utilise pour cela la syntaxe :

let variable : type = expression1 **in** expression2

Eléments du langage

Exemple: avec l'attribut dérivé **impot** dans Personne, on pourrait écrire :

context Personne **inv**:

let montantImposable : Réel = contrat.salaire*0.8 **in**

if (montantImposable >= 100000) **then** impot = montantImposable*45/100

else if (montantImposable >= 50000) **then**

impot = montantImposable*30/100

else impot = montantImposable*10/100

endif

endif

- Là encore, il serait plus correct de vérifier tout d'abord que **contrat** est valué.

- Si on veut définir une variable utilisable dans plusieurs contraintes de la classe, on peut utiliser la construction **def**.

context Personne
def: montantImposable : Réel = contrat.salaire*0.8

Eléments du langage

b) Définition d'opérations:

- Lorsqu'il est utile de définir de **nouvelles opérations**, on peut procéder avec la même construction **def**.

context Personne

def: ageCorrect(a :Réel) :Booléen = a>=0 and a<=140

- Une telle définition permet de réécrire la pré-condition de l'opération

setAge et **l'invariant** sur l'âge d'une personne de manière plus courte.

context Personne **inv**:

ageCorrect(age) - - l'âge ne peut depasser 140 ans

context Personne::**setAge(a** :entier)

pre: ageCorrect(a) and (a >= age)

Eléments du langage

2. Retour de méthode

- L'objet retourné par une opération est désigné par **result**.

context Personne::getAge()

post: **result**=age

- Lorsque la **postcondition** se résume à décrire la valeur du résultat c'est donc équivalent à l'utilisation de **body**.

- Lorsqu'un objet est créé dans une méthode (l'objet n'existait pas au moment des **préconditions**) on peut le savoir grâce à **l'opération oclIsNew** que l'on peut utiliser dans une **postcondition**.

context Personne::donneNaissance() :Personne

post: **result.oclIsNew**() and **result**.age=0

Eléments du langage

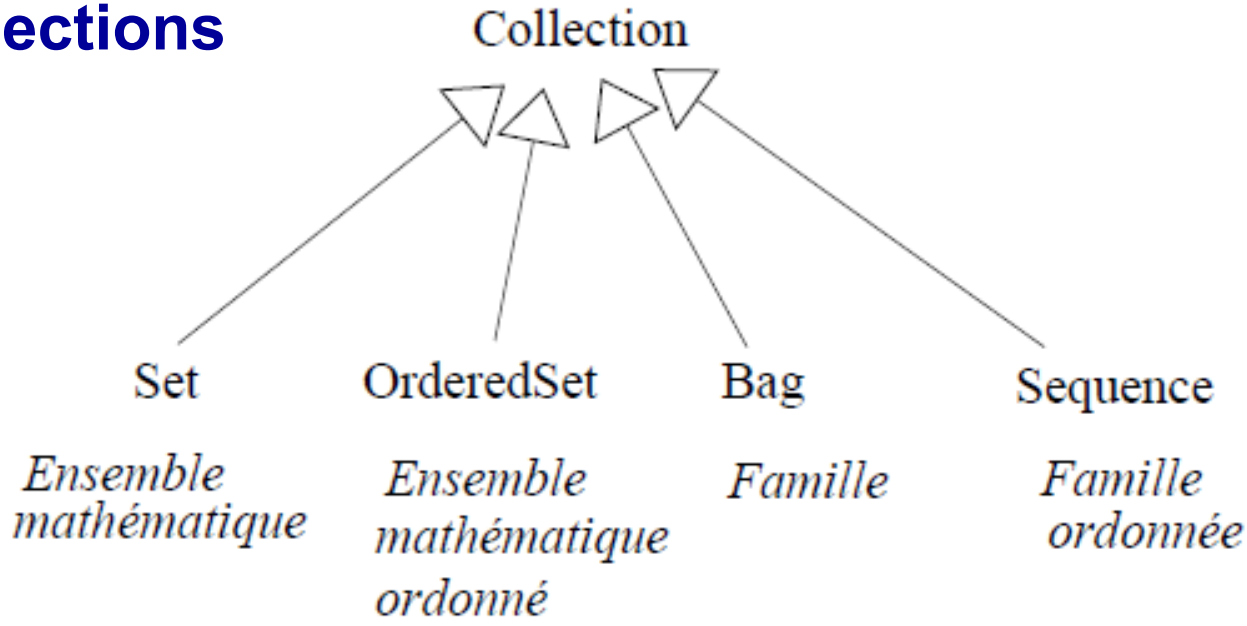
2. Retour de méthode

Question 7

- Proposez une classe **Etudiant**, disposant de 3 **notes** et munie d'une opération **mention** qui retourne la mention de l'étudiant sous forme d'une chaîne de caractères.
- Ecrivez les contraintes et en particulier utilisez le **let** pour écrire la **postcondition** de **mention**.

Collections

1. Types de collections



- Cinq types de collections existant en OCL :

➢ **Collection** est un type abstrait

➢ **Set** correspond à la notion mathématique d'ensemble

➢ **OrderedSet** correspond à la notion mathématique d'ensemble ordonné

➢ **Bag** correspond à la notion mathématique de famille (un élément peut y apparaître plusieurs fois)

➢ **Sequence** correspond à la notion mathématique de famille, et les éléments sont, de plus, ordonnés.

Collections

❑ On peut définir des collections par des littéraux de la manière suivante :

- **Set** { 2, 4, 6, 8 }
- **OrderedSet** { 2, 4, 6, 8 }
- **Bag** { 2, 4, 4, 6, 6, 8 }
- **Sequence** { 1,2,3,4,7,9}
- **Sequence** { 1..10 } spécification d'un intervalle d'entiers

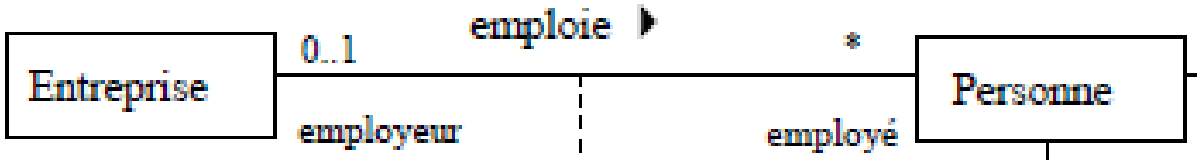
Type de Collection	Ordonnée	Unique
Sequence	true	false
OrderedSet	true	true
Bag	false	false
Set	false	true

❑ Les collections peuvent avoir plusieurs niveaux, il est possible de définir des collections de collections, par exemple :
Set { 2, 4, **Set** {6, 8 }}

Les collections comme résultats de navigation

- Une navigation à partir d'un seul objet peut donnée une collection:
- On obtient un **Set** en naviguant le long d'une association ordinaire

  (**Mais**, elle ne mentionne pas **bag** ou **seq** à l'une de ses extrémités).
- Par exemple, pour le diagramme de l'entreprise, l'expression **self.employé** dans le **contexte Entreprise** est un **Set**.

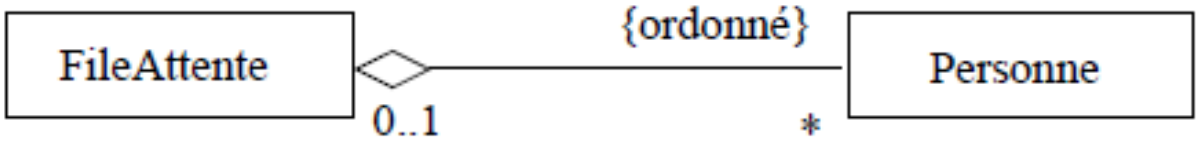


- La exemple suivant illustre une navigation permettant d'obtenir un
  - ➔ **Set** : c'est le résultat de l'expression **self.occurrence.mot** dans le **contexte Texte**.
  - ➔ **Bag** : c'est le résultat de l'expression **self.occurrence** dans le **contexte Texte**.

Les collections comme résultats de navigation

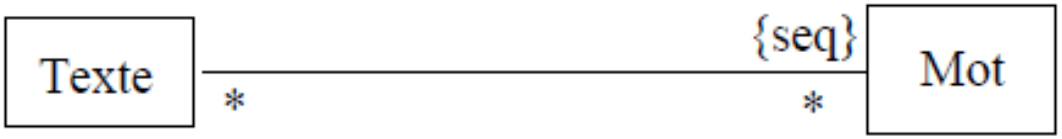
- **OrderedSet** On peut obtenir un ensemble ordonné en naviguant vers une extrémité d'association munie de la contrainte ordonné,

➔ par exemple avec l'expression **self.personne** dans le contexte **FileAttente**.



- **Sequence** On peut obtenir une séquence en naviguant vers une extrémité d'association munie de la contrainte **seq**,

➔ par exemple avec l'expression **self.mot** dans le contexte **Texte** de la figure suivante.



Question 8: Qu'obtient-on en accédant à un attribut multi-valué ?

Opérations de Collection

1. Remarques générales

- Dans la suite, nous considérons des collections d'éléments de type **T**.
- **Appel:** Les opérations sur une collection sont généralement mentionnées avec **->**
- **Itérateurs:** Les **opérations** qui prennent une **expression** comme paramètre peuvent déclarer optionnellement un **itérateur**,

➔ Par exemple toute opération de la forme **operation(expression)** existe également sous deux formes plus complexes.

operation(v | expression-contenant-v)
operation(v : T | expression-contenant-v)

➔ Ou bien simplement
operation(v)

2. Opérations sur tous les types de collections

- **isEmpty() :Boolean**
- **notEmpty() :Boolean**

Permettent de tester si la collection est vide ou non. Pour revenir sur l'exemple des employés

context p :Personne **inv**:

p.agentSecretariat->**notEmpty**() **implies**
p.agentSecrétariat.contrat.dateEmbauche

<= p.encadrement[agentSecrétariat].dateDebut

- **size() :Entier** Donne la taille (nombre d'éléments).
- **count(unObjet :T) :Entier** Donne le nombre d'occurrences de : **unObjet**.
- **sum() :T** Addition de tous les éléments de la collection.

context e :Entreprise

def: ageMoyen : Real = e.employé.age->**sum**() / e.employé.age->**size**()

context e :Etudiant

def: moyenne : Real = e.notes->**sum**() / e.notes->**size**()

❑ includes(obj :T) :Boolean

- Retourne **true** si et seulement si **obj** est un élément de la collection, et **false** sinon
- Toujours pour la classe **Etudiant** généralisée, disposant d'un attribut **notes**:Real[*],
- on peut définir l'opération **estEliminé():Boolean.**
- Un étudiant est éliminé s'il a obtenu un zéro.

context e :Etudiant

def: estEliminé() : Boolean = e.notes->includes(0)

❑ excludes(obj :T) :Boolean

- Retourne **true** si et seulement si **obj** n'est pas un élément de la collection, et **false** sinon
- **Sequence{2.3}-> excludes(2.1)** **true**
- **Sequence{2.0}->excludes(2)** **false**

- **includesAll(c :Collection(T)) :Boolean**

➢ Retourne **true** si et seulement si la collection **contient tous les éléments de c. (false sinon)**

- **excludesAll(c :Collection(T)) :Boolean**

- Retourne **true** si et seulement si la collection **ne contient aucun** des éléments de c. (**false sinon**)

➢ **Sequence{2.3,5.2,'a',3,null}->excludesAll(**Set{4,null}**)** **false**

➢ **Sequence{2.3,5.2,'a',3}->excludesAll(**Set{4,null}**)** **true**

- **Sequence{2.3,5.2,'a',3,4,null}->includesAll(**Set{4,null}**)** **true**

- **exists(uneExpression :Boolean) true** si et seulement si **au moins un élément** de la collection satisfait **uneExpression**.

➢ **Exemple**: Dans toute entreprise, il y a au moins une personne dont le contrat est de type "agent administratif".

➢ **Sequence{2.3,5.2}->exists(self>3)** **true**

➢ **context** e :Entreprise **inv**:

e.contrat->**exists**(c :Contrat | c.typeContrat='agent administratif')

❑ **forAll(uneExpression :Boolean) :Boolean** retourne **true** si et seulement si **uneExpression** est **vraie** pour tous les éléments de la collection.

➢ **Sequence{2.3, 5.2 }->forAll(self>3) false**

➢ Dans une entreprise, une contrainte est que tous les employés aient plus de 16 ans.

context entreprise :Entreprise **inv**:

entreprise.employe->**forAll**(emp :Personne | emp.age >= 16)

❑ Cette opération a une **variante** permettant d'itérer avec plusieurs itérateurs sur la même collection. On peut ainsi parcourir des ensembles produits de la même collection.

forAll(t1,t2:T | expression-contenant-t1-et-t2)

❑ En supposant que la classe **Personne** dispose d'un attribut **nom**, on peut admettre que deux employés différents n'ont jamais le même nom.

context e :Entreprise **inv**:

e.employe->forAll(**e1,e2 :Personne** | e1 <> e2 implies e1.nom <> e2.nom)

- **iterate(i :T ; acc :Type = uneExpression | ExpressionAvec_i_et_acc) :**

➢ Cette opération permet de généraliser et d'écrire la plupart des autres.

- **i** est un itérateur sur la collection
- **acc** est un accumulateur initialisé avec **uneExpression**

➢ L'expression **ExpressionAvec_i_et_acc** est évaluée pour chaque **i** et **son résultat est affecté dans** **acc**. Le résultat de **iterate** est **acc**.

- **Exemple** : l'opération **masseSalariale** d'une entreprise.

context Entreprise::masseSalariale() :Real **post**:

result = employé->**iterate**(**p :Personne ; ms :Real=0 |**

ms+p.contrat.salaire)

Question 8 Ecrire size et forAll dans le contexte de la classe Collection.

❑ **sortedBy(uneExpression) :Sequence** Retourne une séquence contenant les éléments de la collection triés par ordre croissant suivant le critère **uneExpression**.

➔ L'évaluation de **uneExpression** doit donc supporter l'**opération <**

- self.employees->**sortedBy**(age)

- **Question 9 Ecrire la post condition d'une méthode salariésTriés dans le contexte de la classe Entreprise. Cette méthode retourne les employés ordonnés suivant leur salaire.**

❑ **any(uneExpression :OclExpression) :T** Retourne n'importe quel élément de la collection qui vérifie ***uneExpression.***

❑ ***one(uneExpression :OclExpression) :Booléen*** Vrai si et seulement si un et un seul élément de la collection vérifie ***uneExpression.***

❑ **select**, retourne une collection avec tous les éléments de **self** qui **valident** l'expression **expr**.

➔n'est pas définie dans **Collection**, mais seulement dans les types concrets avec des signatures spécifiques:

- ❑ **select(expr:BooleanExpression): Set(T)**
- ❑ **select(expr:BooleanExpression): OrderedSet(T)**
- ❑ **select(expr:BooleanExpression): Bag(T)**
- ❑ **select(expr:BooleanExpression): Sequence(T)**

- Dans un tel cas, nous présentons une généralisation de ces opérations:

select(**expr :BooleanExpression**) : Collection(T)

reject(**expr :BooleanExpression**) : Collection(T)

- Retourne une collection du même type construite par sélection des éléments **vérifiant** (resp. ne vérifiant pas) **expr**.

➔ L'expression représentant dans le contexte d'une **entreprise** les employés âgés de **plus de 60 ans** serait la suivante :

self.employé->**select**(**p:Personne|p.age>=60)**

- **collect(expr :OCLExpression) : Collection(T)**

  Retourne une collection composée des résultats successifs de l'application de **expr** à chaque élément de la collection.

- Sequence{'first', 'second'}->**collect**(toUpper())

  Sequence{'FIRST', 'SECOND'}

- Pour définir la collection des dates de naissance des employés d'une société, il faut écrire, dans le contexte de la classe Entreprise
- **self.employee->collect(birthdate)**

  On peut l'ecrire comme suit :

  self.employee.birthdate

  ➔ Toutes les dates de naissance des employés.

- La manière standard d'atteindre les mots d'un texte s'écrit ainsi dans le contexte d'un texte (Figure précédente) :

  self.occurrence->collect(mot)
  self.occurrence->collect(o | o.mot)
  self.occurrence->collect(o:Occurrence | o.mot)

- Comme nous l'avons évoqué précédemment, cette notation admet un raccourci: *self.occurrence.mot*

- **including(unObjet :T) :Collection(T)**

  excluding(unObjet :T) :Collection(T)

Retourne une collection résultant de l'ajout (resp. du retrait) de ***unObjet*** à la collection.

- On peut ainsi décrire la post-condition d'une opération **embauche(p:Personne**) de la classe Entreprise.

context Entreprise ::embauche(p :Personne) **post**:

self.employé **=** self.employé@pre->**including(p)**

3. Opérations sur tous les OrderedSet et Sequence

- **first**() : le premier élément de la collection
- **last**() : le dernier élément de la collection
- **at(index** : Integer) : l'élément de la collection se trouvant en position *index*
- **indexOf(elt**) : la position de l'élément *elt dans la collection*
- **append(elt**) : la collection augmentée de l'élément *elt placé à la fin*
- **prepend(elt**) : la collection augmentée de l'élément *elt placé au début*
- **insertAt(index** : **Integer**, **elt**) : la collection augmentée de l'élément *elt* placé à la position *index*
- **reverse**() : la même collection mais avec les éléments inversés en position
- **subOrderedSet(lower** : **Integer**, **upper** : **Integer**) : l'ordered set contenant les éléments de la position *lower à upper à partir d'un* ordered set
- **subSequence(lower** : **Integer**, **upper** : **Integer**) : la séquence contenant les elements de la position *lower à upper à partir d'une séquence*

4. Opérations de conversion

- Chaque sorte de collection (set, orderedSet, sequence, bag) peut être transformée dans n'importe quelle autre sorte.

- Par exemple un **Bag** peut être transformé en :

  - **sequence** par l'opération **asSequence():Sequence(T)** ; l'ordre résultant est déterminé

  - **ensemble** par l'opération **asSet():Set(T) ;** les doublons sont éliminés

  - **ensemble ordonné** par l'opération **asOrderedSet():OrderedSet(T) ;** les doublons sont éliminés ; l'ordre résultant est indéterminé.

- A titre d'illustration, si nous voulons exprimer le fait que le nombre de mots différents d'un texte est plus grand que deux (figure précédente) :

context Texte **inv**:

self.occurrence->**collect**(mot)->**asSet**()->size() >= 2

Questions ?