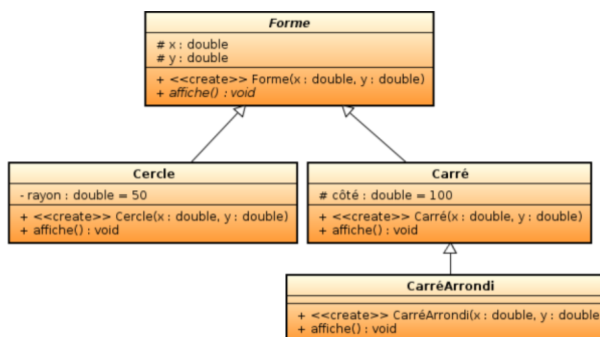


Extrait du référentiel : BTS Systèmes Numériques option A (Informatique et Réseaux)		Niveau(x)
S3. Modélisation S3.4. Spécificités UML	Diagrammes de classes et/ou d'objets	3

Objectifs du cours :

- Le diagramme de classes :
  - rappel sur les classes
  - rôle du diagramme de classes
- Représentation des classes
- Les relations entre classes :
  - la relation de dépendance
  - les associations
  - la relation d'héritage
  - classes concrètes et abstraites

LE DIAGRAMME DE CLASSES

RAPPEL SUR LES CLASSES

Une classe est une **représentation abstraite d'un d'ensemble d'objets**, elle contient les informations nécessaires à la construction de l'objet (c'est-à-dire la définition des attributs et des méthodes).
La classe peut donc être considérée comme le modèle, le moule ou la notice qui va permette la construction d'un objet. Nous pouvons encore parler de type (comme pour une donnée).
On dit également qu'un objet est **l'instance** d'une classe (**la concrétisation d'une classe**).

RÔLE DU DIAGRAMME DE CLASSES

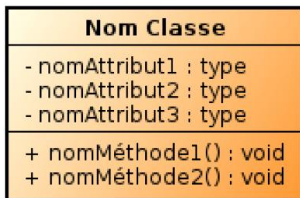
Le diagramme de classes est un diagramme qui permet de représenter :

- **les classes** (attributs + méthodes)
- **les associations** (relations) entre les classes.

Le diagramme de classes est le plus important des diagrammes UML, c'est le seul qui soit obligatoire lors de la modélisation objet d'un système.

REPRÉSENTATION DES CLASSES

Une **classe** est représentée par un **rectangle** (appelé aussi classeur) divisé en 3 compartiments.



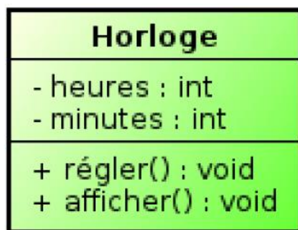
Le premier compartiment contient le nom de la classe qui :
- représente le type d'objet instancié.
- débute par une lettre majuscule.
- il est centré dans le compartiment supérieur de la classe.
- il est écrit en caractère gras.
- il est en italique si la classe est abstraite (IMPOSSIBLE d'instancié un objet).

Le deuxième compartiment contient les attributs.
Le troisième compartiment contient les méthodes.

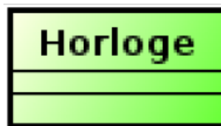
Si la modélisation ne s'intéresse qu'aux relations entre les différentes classe du système (et pas au contenu des classes), les attributs et les méthodes de chaque classe peuvent ne pas être représentés (rien dans le deuxième et troisième compartiment).

Exemples :

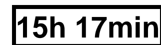
- la classe **Horloge** (ou **Chorloge**)



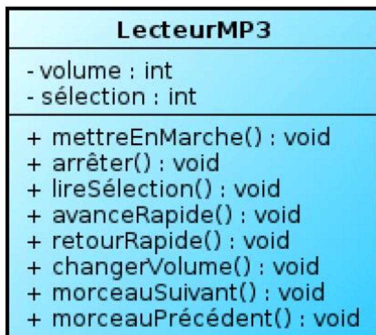
ou :



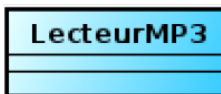
Exemple d'instance de la classe **Horloge** :



- la classe **LecteurMP3** (ou **ClecteurMP3**)



ou :



Exemples d'instance de la classe **LecteurMP3** :

Il est possible de détailler la classe en indiquant :

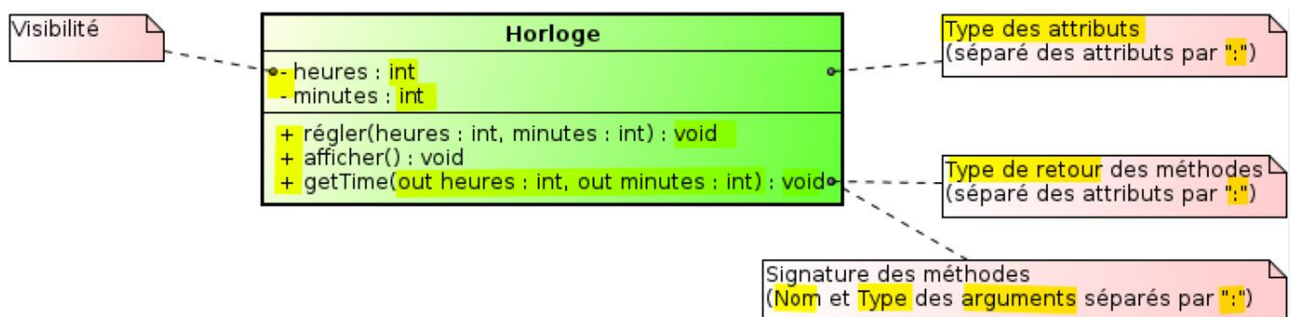
- la **visibilité** (encapsulation) des **méthodes** et des **attributs**.

Type de visibilité	Symbole devant l'attribut et/ou la méthode
public : élément non encapsulé visible par tous.	**+**
private : élément encapsulé visible seulement dans la classe.	**-**
protected : élément encapsulé visible dans la classe et dans les sous-classes.	**#**
package : élément encapsulé visible dans les classes du même paquetage.	**~**

- le type de chaque attribut.
- **la signature de chaque méthode**.
- **le type de valeur retournée par chaque méthode**.

Exemple :

- la classe **Horloge** (ou **Chorloge**)



- la direction des paramètres des méthodes :
Devant le nom du paramètre, il est possible d'indiquer par un mot clé (**in, out, inout**), la direction dans laquelle celui-ci est transmis.

in	La valeur du paramètre est transmise à l'appel de la méthode (par l'appelant de la méthode) et ne peut pas être modifiée (c'est le comportement par défaut si aucune direction n'est spécifiée).
out	La valeur finale du paramètre est transmise au retour de l'appel de la méthode (à l'appelant de la méthode).
inout	La valeur du paramètre est transmise à l'appel et au retour.

Exemple :

Considérons la méthode `régler(heures:int,minutes:int):void` de la classe **Horloge** de l'exemple précédent. L'appelant de la méthode veut affecter les **attributs** `heures` et `minutes` avec des valeurs qu'il va donner.

La direction des **paramètres** sera alors `in:régler(in heures:int,in minutes:int):void`

Considérons maintenant la méthode `getTime(heures:int,minutes:int):void` de la classe **Horloge**. L'appelant de la méthode veut récupérer les valeurs des **attributs** **`heures`** et **`minutes`**.

La direction des paramètres sera alors `out:getTime(out heures:int,out minutes:int):void`

Comme une méthode ne peut retourner qu'une seule valeur, et que dans notre cas nous avons besoin de connaître deux valeurs (les heures et les minutes), le retour se fera par les arguments (en utilisant les références ou les pointeurs en C++).

- valeurs par défauts des attributs et des paramètres des méthodes :
Nous indiquons les valeurs par défauts des attributs lors de leur construction et les valeurs par défauts des paramètres des méthodes s'ils ne sont pas clairement spécifiés lors de l'appel.

Exemple :

```
Horloge
- heures : int = 0
- minutes : int = 0
+ régler( in heures : int = 0, in minutes : int = 0) : void
+ afficher() : void
+ getTime(out heures : int, out minutes : int) : void
```

- attributs et méthodes de classe (ou attributs et méthodes statiques) :
Une classe peut contenir des attributs et des méthodes qui lui sont propres et auxquels nous pouvons accéder sans nécessairement instancier des objets. Un attribut de classe n'appartient pas à un objet en particulier mais à toute la classe (il n'est pas instancié avec l'objet). Un attribut ou une méthode de classe est représenté par un nom souligné.
Cela permet également d'avoir une information commune à tous les objets instanciés.

Exemple :

Soit la classe **Gasoil** qui représente le gasoil fourni par une pompe de station service.

```
Gasoil
- quantité : double {quantité<100}
- prixParLitre : double = 1.23          (souligné)
+ setQuantité(quantité : double) : void
+ getQuantité() : double {query}
+ fixePrix(parLitre : double) : void    (souligné)
```

L'attribut prixParLitre appartient à la classe Gasoil, il ne fait pas parti des objets instanciés, mais chaque objet individuel peut y accéder.

La méthode fixePrix() permet de modifier la valeur de l'attribut de classe prixParLitre (c'est le seul autorisé).

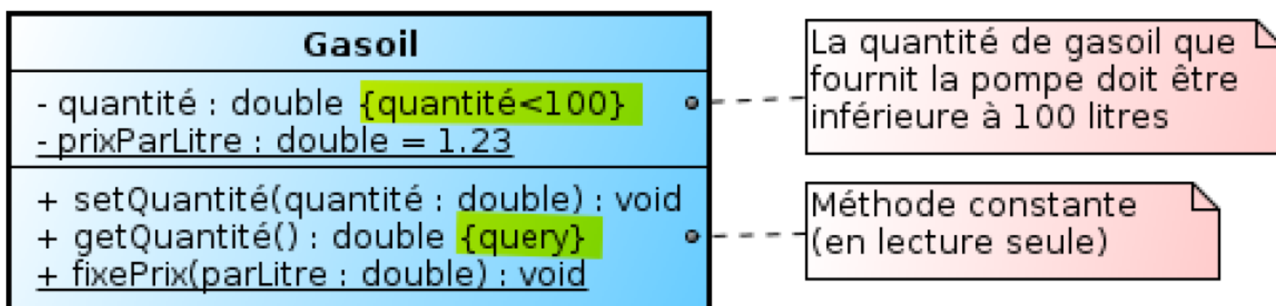
- les contraintes :
Une contrainte est une condition écrite **entre 2 accolades** elle peut être exprimée dans :
un langage naturel (description textuelle) ;
un langage formel (**C++, java, OCL…**).

OCL (Object Contraint Language) est un langage spécialement conçu pour exprimer des contraintes.

Voici quelques contraintes qui peuvent être utiles :

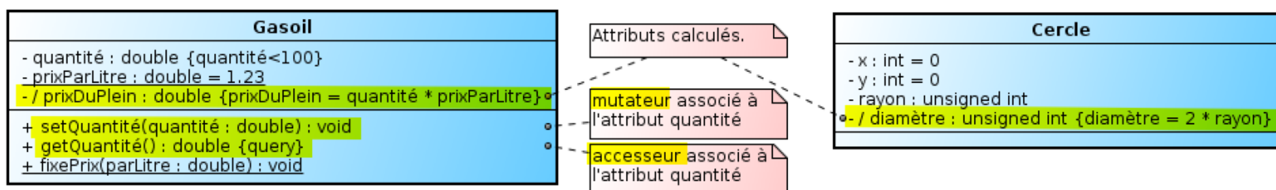
`{readOnly}` : si une telle contrainte est appliquée à un attribut, alors la valeur de celui-ci ne peut plus être modifiée une fois la valeur initiale fixée (équivalent à un attribut constant).
`{query}` : une méthode peut être déclarée comme étant de type requête (**query**) si le code implémentant celle-ci ne modifie nullement l'état de l'objet, donc aucun de ses attributs.
`{ordered} {list}` : lorsqu'une multiplicité supérieure à 1 est précisée, il est possible d'ajouter une contrainte pour préciser si les valeurs sont ordonnées **{ordered}** ou pas **{list}**. Très souvent, dans ce dernier cas, nous ne précisons même pas cette deuxième contrainte, c'est le mode par défaut.
`{unique}` : on demande cette fois-ci qu'il n'y ait aucun doublon dans les valeurs de la dans les valeurs de la collection.
`{not null}` : l'attribut doit à tout prix être initialisé (utile dans le cas des pointeurs).

Exemple :



- les attributs calculés (dérivé) :
Une classe peut avoir des attributs calculés, c'est-à-dire que leurs valeurs sont proposées au travers d'une fonction utilisant les autres attributs précédemment exprimés. Un tel attribut possède un nom précédé du signe « **/** » et suivi d'une contrainte permettant de le calculer.

Exemple :



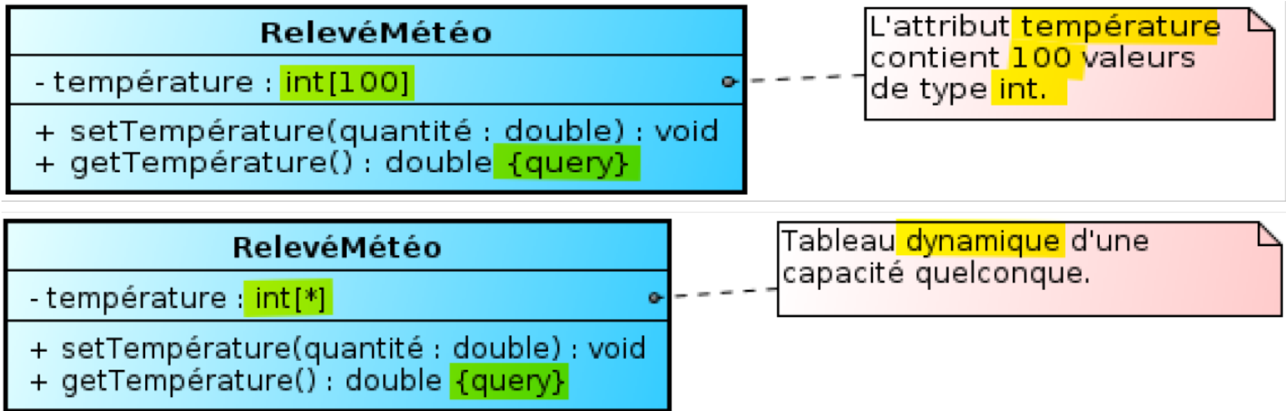
Les **mutateurs** et les **accesseurs** sont des méthodes particulières qui permettent respectivement de modifier ou de consulter le contenu d'un attribut spécifique dans la classe.

Cardinalité	Signification
0..1	Zéro ou une fois
1..1 (ou 1)	Une et une seule fois
0..* (ou *)	De zéro à plusieurs fois
1..*	De une à plusieurs fois
m..n	Entre **m** et **n** fois
n..n (ou n)	**n** fois

La multiplicité (ou cardinalité) :
La multiplicité indique le nombre de valeur que l'attribut peut contenir (l'attribut est souvent un tableau de valeurs, statique ou dynamique). La multiplicité se note entre crochets après le type de valeur que contient l'attribut.

Exemple :

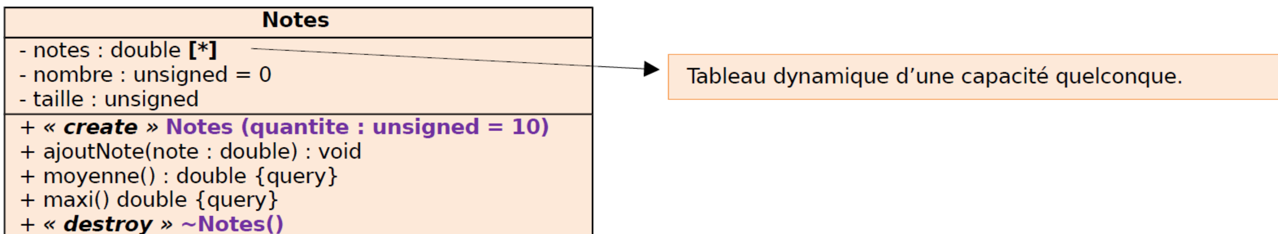
Une station météo doit relever la température à intervalle de temps régulier. Elle doit pouvoir stocker 100 relevés.



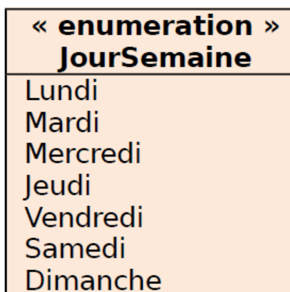
- constructeur et destructeur :
Les stéréotypes peuvent être utilisés pour identifier des opérations particulières comme les constructeurs (stéréotype « `create` ») et le destructeur (stéréotype « `destroy` »).

Exemple :

Il est possible de mettre en place une gestion de notes, avec le calcul de la moyenne, de la valeur maxi, etc. au travers d'une classe adaptée nommée **Notes**.



- les énumérations :
Une énumération est un type possédant un nombre fini et arbitraires de valeurs possibles, construite sur mesure par le développeur, pour typer des variables bien particulières, comme le représentation des jours de la semaine ou des mois de l'année.

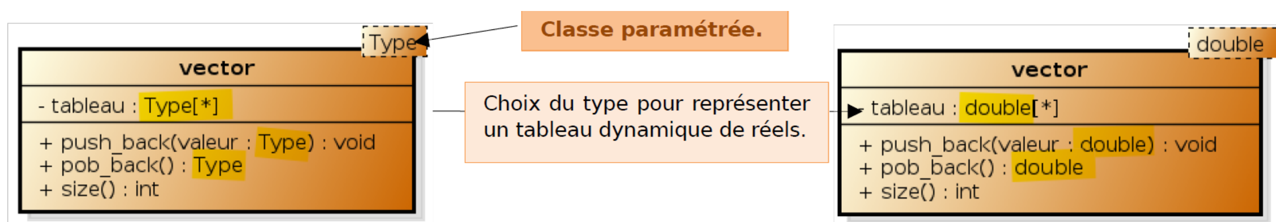


En UML, une énumération ne se définit pas par une classe, mais par un **classeur stéréotypé** « enumeration ». Il s'agit d'un type de données, possédant un nom, et utilisé pour énumérer un ensemble de littéraux correspondant à toutes les valeurs possibles que peut prendre une expression de ce type.

- les modèles de classe :
Les modèles (**templates**) sont une fonctionnalité avancée de l'orientée objet. Un modèle est une classe paramétrée qui permet ainsi de choisir le type des attributs au besoin suivant le paramètre précisé, dans le coin supérieur droit dans un rectangle dont les côtés sont en pointillés.

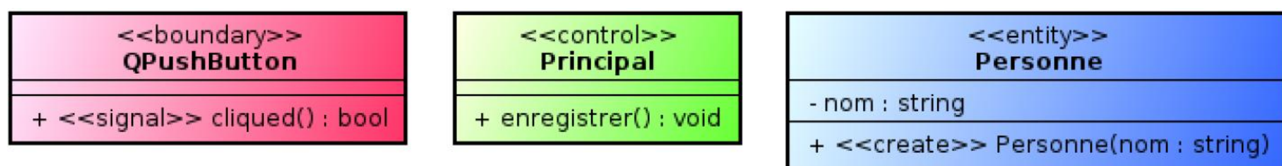
Exemple :

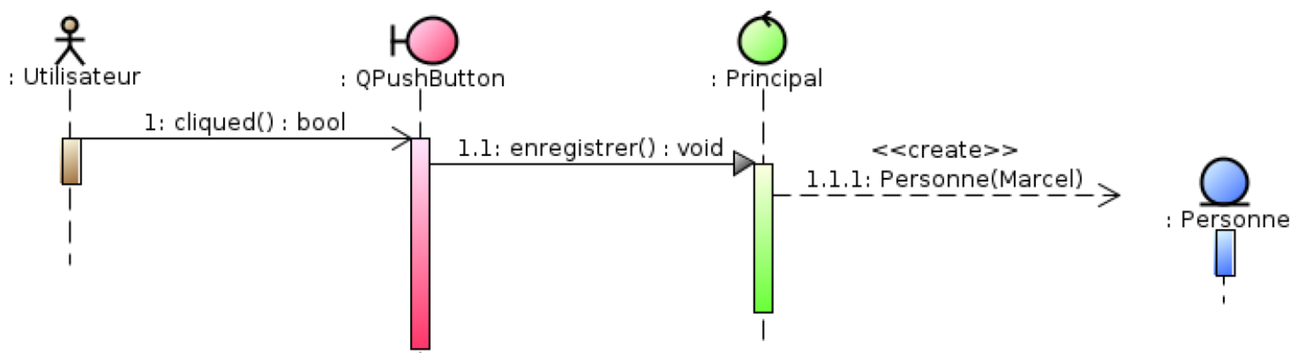
Ces modèles de classes sont particulièrement utiles pour toutes les collections qui stockent des valeurs d'un même type, soit sous forme de tableaux dynamiques ou de listes. La classe **vector**, issue de la STL en est un parfait exemple.



- stéréotypes de Jacobson :
À l'intérieur d'un système, il existe très souvent des classes qui possèdent un rôle bien particulier qui serait intéressant de visualiser d'une façon non équivoque dans votre diagramme de séquence.
C'est le cas notamment :
pour les classes qui représentent des composants de **l'IHM** ;
pour la classe qui contrôle globalement le système avec la prise en compte de la **gestion événementielle** ;
pour les classes qui implémentent la **persistance** des attributs (associées à une base de données).

Jacobson distinguent les trois stéréotypes suivants :
« **boundary** » : classes qui servent à modéliser les interactions entre le système et ses acteurs.
« **control** » : classes utilisées pour représenter la coordination, l'enchaînement et le contrôle d'autres objets.
« **entity** » : classes qui servent à modéliser des informations durables et souvent persistantes.

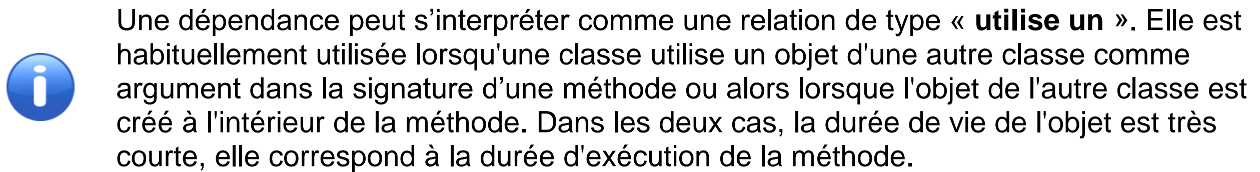




LES RELATIONS ENTRE CLASSES

LA RELATION DE DÉPENDANCE

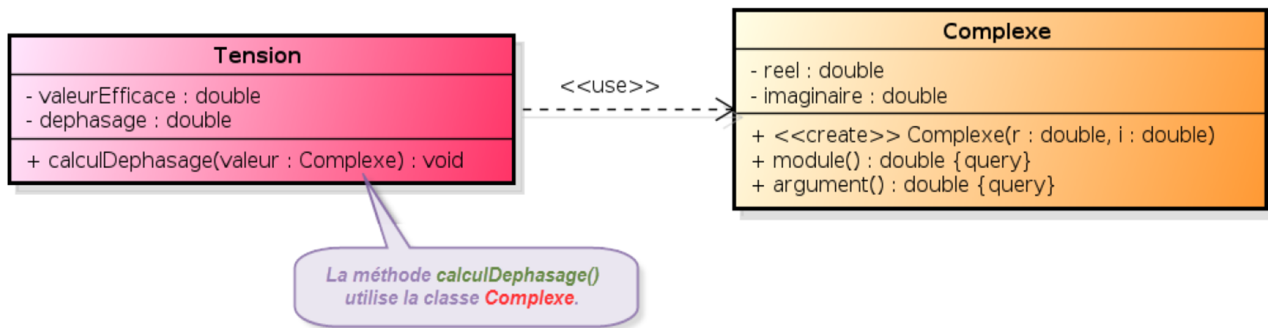
La dépendance est la forme la plus faible de relation entre classes. Une dépendance entre deux classes signifie que l'une des deux utilise l'autre. Typiquement, il s'agit d'une relation transitoire, au sens où la première interagit brièvement avec la seconde sans conserver à terme de relation avec elle (**liaison ponctuelle**).



Une dépendance peut s'interpréter comme une relation de type « **utilise un** ». Elle est habituellement utilisée lorsqu'une classe utilise un objet d'une autre classe comme argument dans la signature d'une méthode ou alors lorsque l'objet de l'autre classe est créé à l'intérieur de la méthode. Dans les deux cas, la durée de vie de l'objet est très courte, elle correspond à la durée d'exécution de la méthode.

La dépendance est représentée par un trait discontinu orienté, reliant les deux classes. La dépendance est souvent stéréotypée (« **use** ») pour mieux expliciter le lien sémantique entre les éléments du modèle.

Exemple :



LES ASSOCIATIONS

Alors que la dépendance autorise simplement une classe à utiliser des objets d'une autre classe, l'association signifie qu'une classe contiendra une référence (ou un pointeur) de l'objet de la classe associée sous la forme d'un attribut.

Cette relation est plus forte. Elle indique qu'une classe est en relation avec une autre pendant un certain laps de temps. La ligne de vie des deux objets concernés ne sont cependant pas associés étroitement (un objet peut être détruit sans que l'autre le soit nécessairement).

L'association est représentée par un simple trait continu, reliant les deux classes. Le fait que deux instances soient ainsi liées permet la navigation d'une instance vers l'autre, et vice versa (chaque classe possède un attribut qui fait référence à l'autre classe).

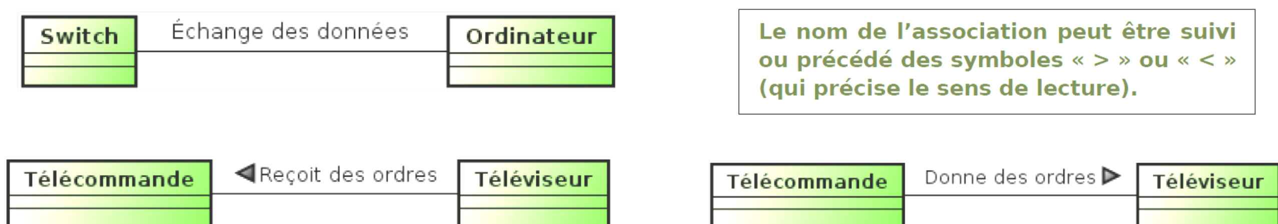
Exemple :

Il <u>est possible</u> de détailler l’association en indiquant :

- le nom de l’association :
L’association peut être orné d’un texte, avec un éventuel sens de lecture, qui permet de nous informer de l’intérêt de cette relation. Nous rajoutons une phrase courte permettant de préciser le contexte de cette association.

Ce texte complémentaire n’est absolument pas exploité dans le code. Le nom d’une association doit respecter les conventions de nommage des classeurs : commencer par une lettre majuscule.

Exemples :



- le rôle :
Chaque extrémité d’une association peut être nommée. Ce nom est appelé **rôle** et indique la manière dont l’objet est vu de l’autre coté de l’association.
Lorsqu’un objet A est lié à un autre objet B par une association, cela se traduit souvent par un attribut supplémentaire dans A qui portera le nom du rôle B (et inversement).

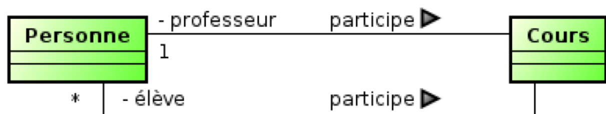
Exemples :

- la cardinalité (ou multiplicité) :
Les associations sont typiquement destinées à représenter des relations durables entre des classes; elles sont souvent utilisées pour représenter les attributs de la classe. Et comme nous l’avons vu pour un attribut, il est possible d’utiliser la multiplicité pour indiquer le nombre d’instances (d’une classe donnée) qui sont impliquées dans la relation.

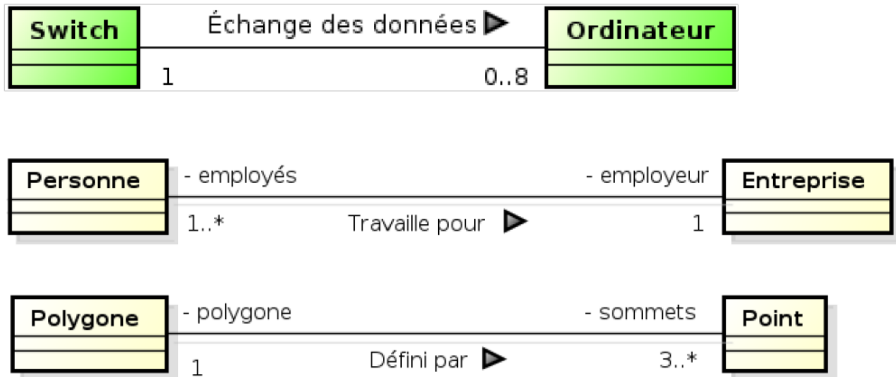
Exemples :

La **cardinalité** indique le nombre d’instances de classe étant en relation avec la classe située à l’autre extrémité de l’association. En l’absence de spécification, la cardinalité vaut 1.

Dans un cours, il y a plusieurs élèves et un professeur.

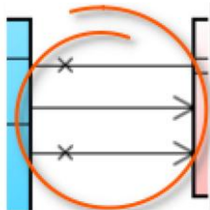


Si un switch dispose de huit ports.



- la navigabilité :
Les associations possèdent une navigation bidirectionnelle par défaut, c'est-à-dire qu'il est possible de déterminer les liens de l'association depuis une instance de chaque classe d'origine. Cela suppose que chaque classe possède un attribut qui fait référence à l'autre classe en association. Une navigation bidirectionnelle est du coup plus complexe à réaliser; il convient de l'éviter dans la mesure du possible.

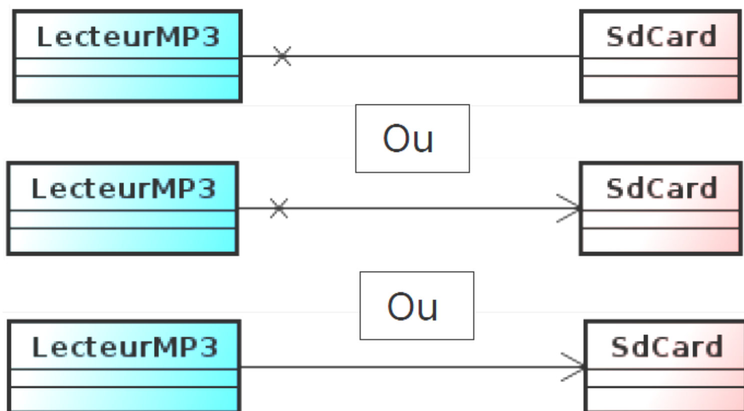
Il est beaucoup plus fréquent d'avoir besoin d'une navigabilité unidirectionnelle. Dans ce cas, une seule classe possède un attribut qui fait référence à l'autre classe, ce qui se traduit par le fait que la première classe peut solliciter une deuxième et que l'inverse est impossible (la deuxième classe ne connaît pas la première).



Une relation unidirectionnelle peut se représenter de 3 façons différentes :
- une croix du coté de l'objet qui ne peut pas être sollicité ;
- une flèche du coté de l'objet qui peut être sollicité ;
- les 2 représentations précédentes à la fois.

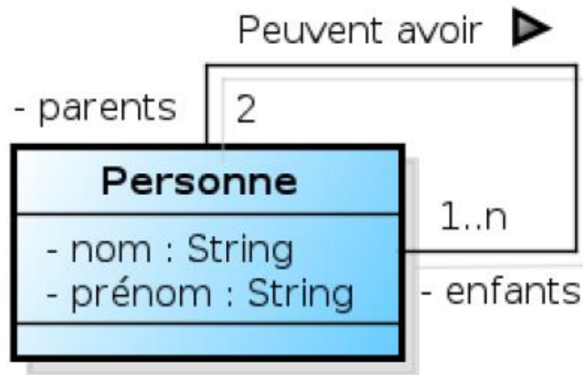
Exemple :

Un lecteur MP3 possède un emplacement pour accueillir et lire une SdCard. C'est toujours le lecteur MP3 qui accède à la SdCard (et jamais l'inverse).



- association réflexive (ou récursive) :
Une association qui lie une classe avec elle-même est une association réflexive.

Exemple :

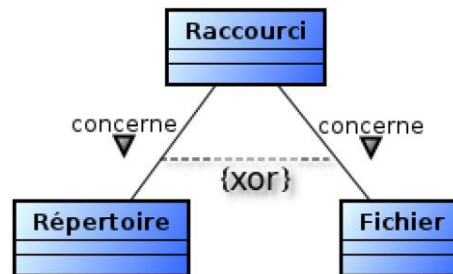


- contraintes et associations :

Exemple 1 :

Contrainte entre deux associations.

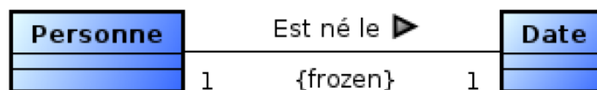
En informatique, un raccourci peut concerner soit un répertoire, soit un fichier (mais pas les 2 à la fois). Nous pouvons exprimer cela sous la forme d'une contrainte **{xor}**.



Exemple 2 :

Contrainte sur une association.

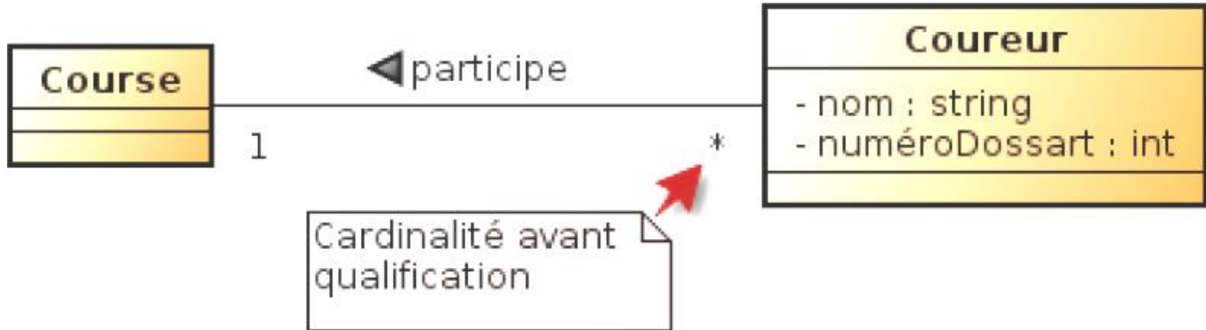
Un objet de la classe **Personne** est associé à un objet de la classe **Date**. La contrainte **{frozen}** indique que cette association ne peut plus être modifiée une fois instanciée.



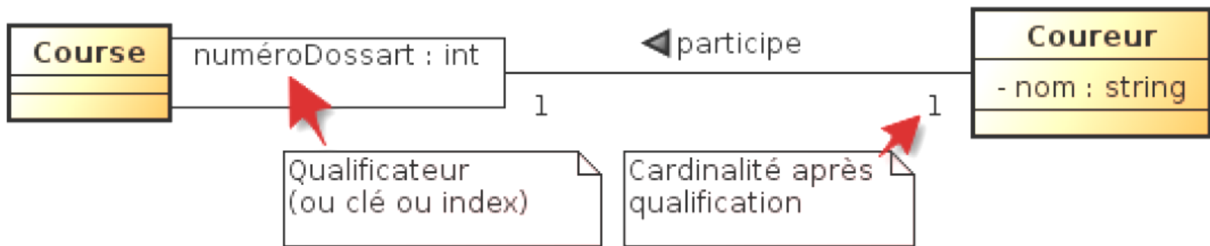
- association qualifiée (Qualification) :
Une association qualifiée permet de restreindre la cardinalité d'une association en ajoutant un qualificateur (aussi appelé **clé** ou **index**). Ce qualificateur est constitué de un ou plusieurs attributs qui permettent de cibler un ou plusieurs objets en particulier.
Le qualificateur est placé dans un rectangle à l'extrémité de l'association (extrémité opposée à la classe dont nous limitons la cardinalité).

Exemple 1 :

Un objet de la classe **Course** est relié à un nombre indéterminé d'objet de la classe **Coureur**.



Par contre, la classe **Course** associée au qualificateur **numéroDossart** n'est reliée qu'a un seul objet de la classe **Coureur**.

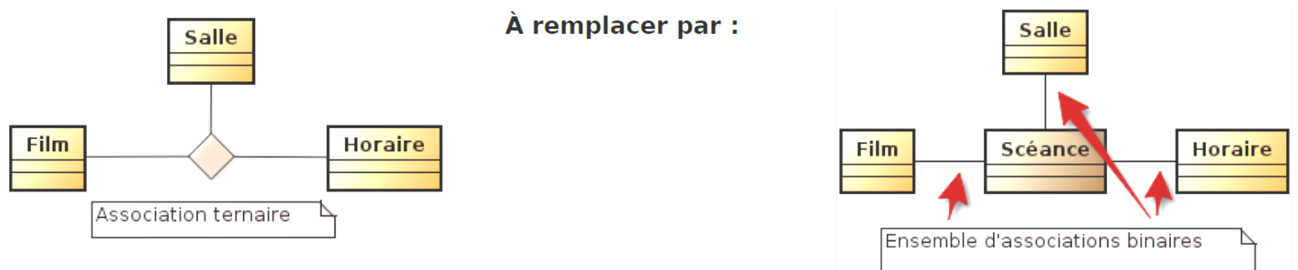


Exemple 2 :

- association n-aire :
Une association qui lie plus de 2 classes entre elles, est une association n-aire. L'association n-aire se représente par un losange d'où part un trait allant à chaque classe. L'association n-aire est imprécise, difficile à interpréter et souvent source d'erreur, **elle est donc très peu utilisée.** La plupart du temps nous nous en servons que pour esquisser la modélisation au début du projet, puis elle est vite remplacée par un ensemble d'associations binaires afin de lever toute ambiguïté.

Exemple 1 :

Une séance de cinéma peut correspondre à l'association ternaire de 3 classes.



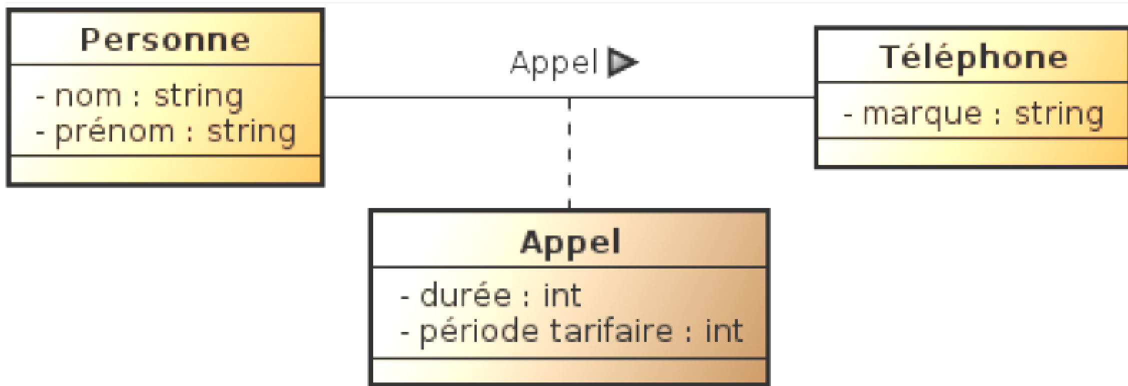
Exemple 2 :

Un cours peut correspondre à l'association ternaire de 3 classes.

- classe association :
Une association peut apporter de nouvelles informations (**attributs** et **méthodes**) qui n'appartiennent à aucune des deux classes qu'elle relie et qui sont spécifiques à l'association. Ces nouvelles informations peuvent être représentées par une nouvelle classe attachée à l'association via un trait en pointillés.

Exemple :

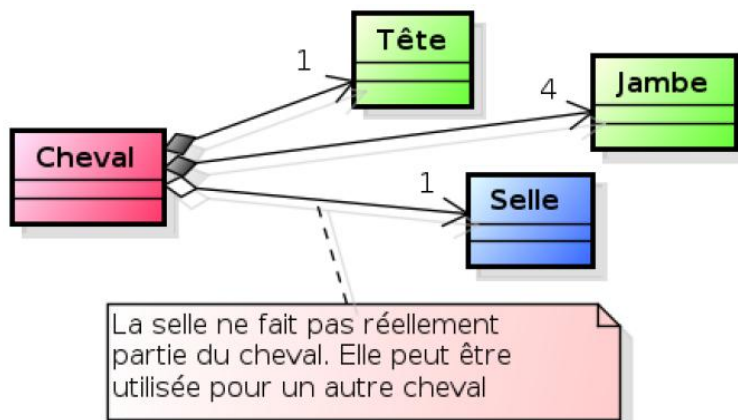
Lorsqu'une personne utilise un téléphone, il faut pouvoir mesurer la durée de l'appel et savoir à quel moment il a lieu afin de le tarifer. Nous ajoutons donc deux attributs **durée** et **période tarifaire** qui n'appartiennent ni à la classe **Personne** ni à la classe **Téléphone.** Ces deux attributs sont mis dans une nouvelle classe (la classe **Appel**) qui est attachée à l'association.



Comme pour l'association ternaire, nous pouvons convertir la classe association en un ensemble d'associations binaires.

- la composition indique qu'un objet **A** (appelé conteneur) est constitué d'un autre objet **B**. Cet objet **A** n'appartient qu'a l'objet **B** et ne peut pas être partagé avec un autre objet.
C'est une relation très forte : si l'objet **A** disparaît, alors l'objet **B** disparaît aussi.
Un cheval possède une tête et 4 jambes.
Elle se représente par un losange plein du coté de l'objet conteneur.

La manière habituelle d'implémenter la composition en C++ se fait soit directement au travers d'un attribut normal en utilisant la liste d'initialisation pour la création de l'objet composite, soit à l'aide d'une variable dynamique, auquel cas, nous ne devons pas oublier de rajouter un destructeur pour libérer cette variable dynamique.



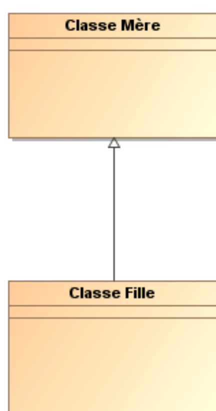
- l'agrégation indique qu'un objet **A** possède un autre objet **B**, mais contrairement à la composition, l'objet **B** peut exister indépendamment de l'objet **A**. La suppression de l'objet **A** n'entraîne pas la suppression de l'objet **B**. L'objet **A** est plutôt à la fois possesseur et utilisateur de l'objet **B**.
Un cheval possède une selle sur son dos.
Elle se représente par un losange vide du coté de l'objet conteneur.

La manière habituelle d'implémenter l'agrégation en C++ se fait au travers d'un pointeur ou d'une référence qui pointe ou fait référence à l'objet agrégé déjà créé en dehors de l'objet conteneur (chaque objet a sa propre durée de vie).

La composition et l'agrégation sont des cas particuliers d'association.

LA RELATION D'HÉRITAGE

Le mécanisme d'héritage permet de mettre en relation des classes ayant des caractéristiques communes (**attributs** et **comportements**) en respectant une certaine filiation.



L'héritage indique qu'une classes B est une spécialisation d'une classe A. La classe B (appelé classe fille, classe dérivée ou sous classe) hérite des **attributs** et des **méthodes** de la classe A (appelée classe mère, classe de base ou super classe).

Il se représente par un triangle vide afin d'indiquer le sens de la généralisation (inverse de la spécialisation).

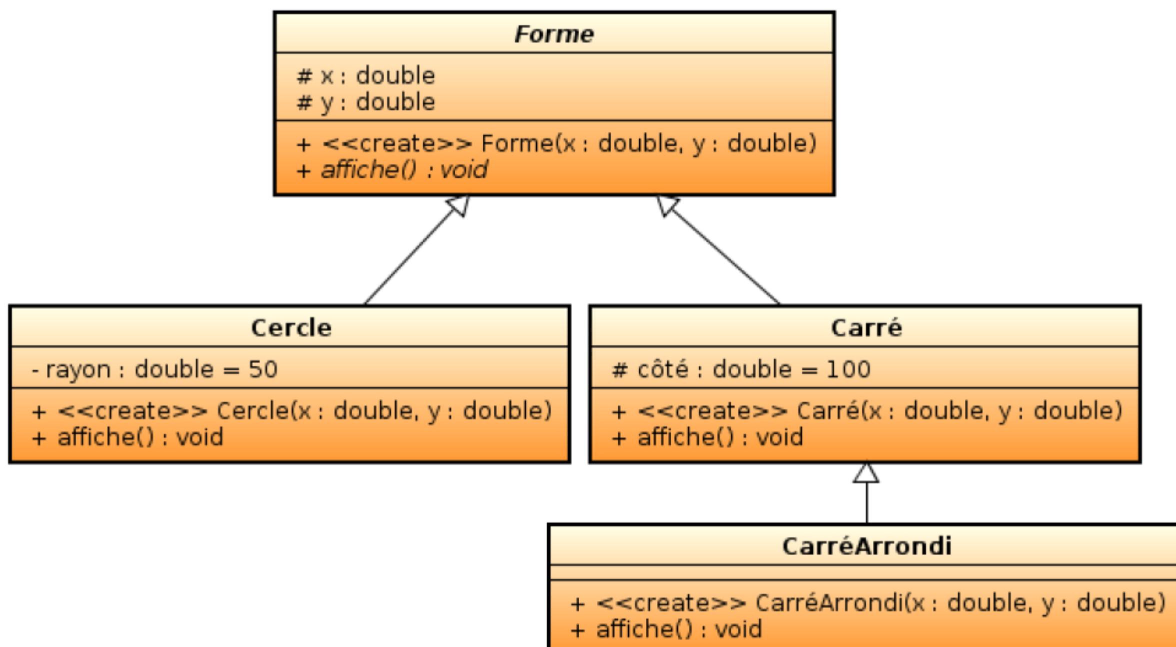
CLASSES CONCRÈTES ET ABSTRAITES

Une **classe concrète** possède des instances. Elle constitue un modèle complet d'objet (tous les attributs et méthodes sont complètements décrits).

À l'opposé, une **classe abstraite** ne peut pas posséder d'instance directe car elle ne fournit pas une description complète. Elle a pour vocation de posséder des sous-classes concrètes et sert à factoriser des attributs et des méthodes à ses sous-classes.

Une classe abstraite possède généralement des méthodes communes aux sous-classes qui sont uniquement déclarées (sans codage interne).

Une méthode introduite dans une classe avec sa seule signature et sans code est appelée une **méthode abstraite**.



En UML, une classe ou une méthode abstraite sont représentées avec une mise en italique du nom de la classe ou de la méthode.
Toute classe possédant au moins une méthode abstraite est une classe abstraite. En effet, la seule présence d'une méthode incomplète (le code est absent) implique que la classe ne soit pas une description complète.

Ci-dessus, la classe **Forme** est abstraite puisqu'elle est constituée de la méthode abstraite **affiche()**. Cette méthode est spécifiée abstraite puisqu'il est impossible de décrire un tracé particulier, ne connaissant pas la forme exacte. Ce n'est qu'avec une classe concrète comme **Cercle** que nous sommes capables de réaliser le tracé correspondant.