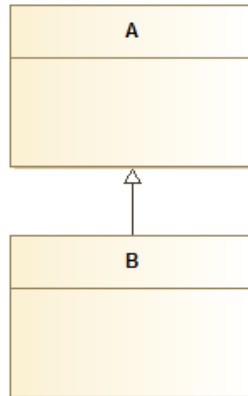
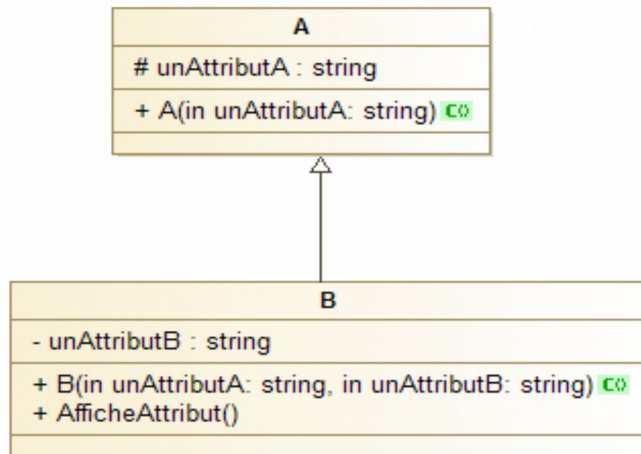


Liaisons UML – JAVA

Relation d'héritage



Exemple :



```
// Relation d'héritage : B hérite de A
// Héritage et constructeur :
// La sous-classe confie au constructeur de la superclasse
// le soin d'initialiser les attributs qu'elle hérite de celle-ci
// à l'aide super()
```

```
class A {

    protected String unAttributA ;

    public A(String unAttributA) {
        this.unAttributA = unAttributA ;
    }

}
```

```
class B extends A {

    private String unAttributB ;

    public B(String unAttributA,String unAttributB){
        super(unAttributA) ;
        this.unAttributB = unAttributB ;
    }

    public void AfficheAttribut(){
        System.out.println("mes attributs sont " + this.unAttributA + " et " +
this.unAttributB ) ; // on peut enlever le this
    }

}

public class Heritage {

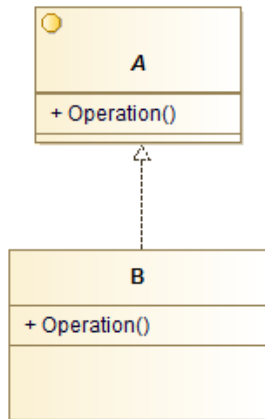
    public static void main(String[] args) {
        B objetB = new B("varA","varB") ;
        objetB.AfficheAttribut() ;
    }

}
```

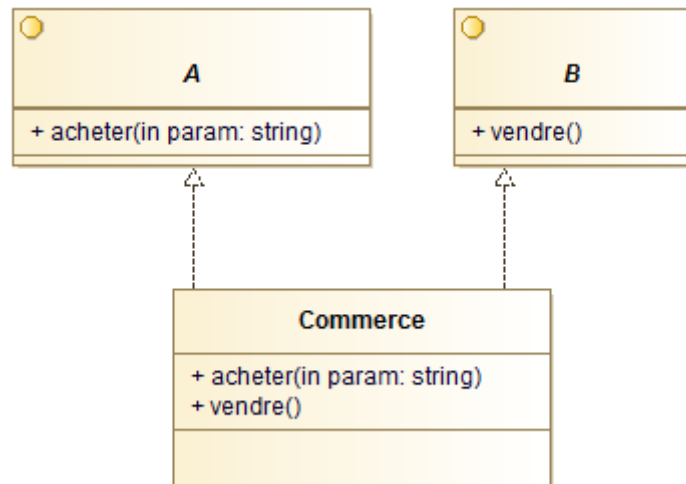
Implémentation d'interfaces

La classe B implémente l'interface A.

Une classe peut implémenter autant d'interfaces qu'elle le souhaite.



Exemple :



```
// Implémentation d'interfaces :
// Lien fort qui nécessite l'inclusion du code de l'interface
// dans le code de la classe
// Obligatoirement un fichier pour chaque interface
```

```
// Fichier A.java
```

```
public interface A {
    abstract void acheter(String s) ;
}
```

```
// Fichier B.java
```

```
public interface B {
    public void vendre() ;
}
```

```
// Fichier Commerce.java
```

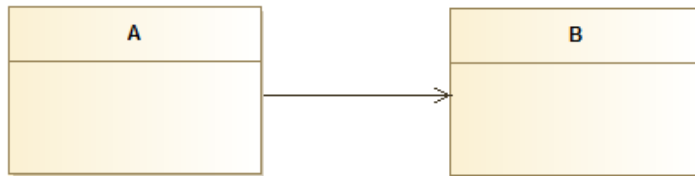
```
public class Commerce implements A, B{

    public void acheter(String param) {
        System.out.println("Acheter " + param + " stylos") ;
    }

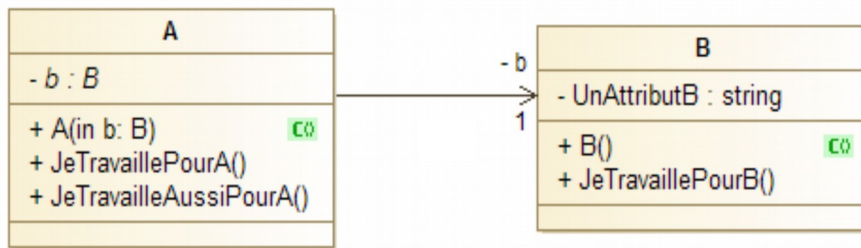
    public void vendre() {
        System.out.println("Vendre du papier") ;
    }

    public static void main(String[] args) {
        Commerce objetCommerce = new Commerce() ;
        objetCommerce.acheter("10") ;
        objetCommerce.vendre() ;
    }
}
```

Relation unidirectionnelle permanente



Exemple :



```
// Relation unidirectionnelle permanente : A ---> B
// Utilisation permanente :
// l'inclusion de la classe utilisée est obligatoire
// Objet de B dans le constructeur de A
```

```
class A {

    private B b ;

    public A(B b) {
        this.b = b ;
    }

    public void JeTravaillePourA(){
        this.b.JeTravaillePourB() ;
    }

    public void JeTravailleAussiPourA(){
        System.out.println("Je travaille aussi pour A") ;
    }

}

class B {

    // private String unAttribut ;

    public B(){}

    public void JeTravaillePourB(){
        System.out.println("Je travaille pour toutes les classes") ;
    }

}

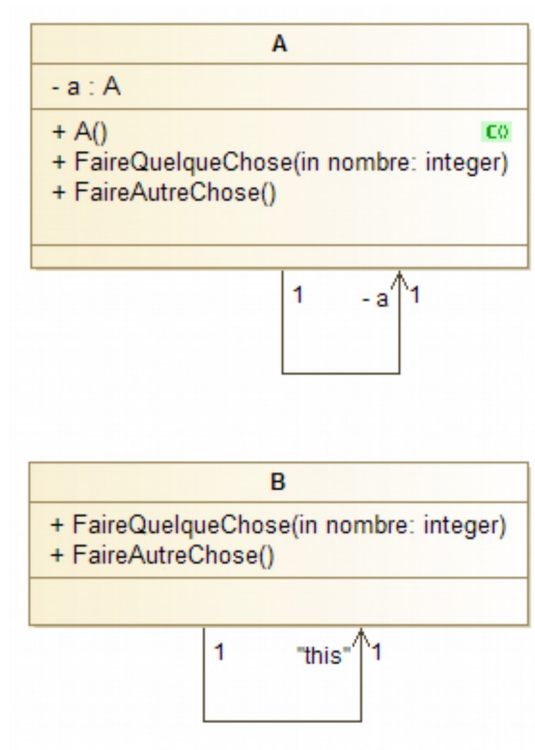
public class Main {

    public static void main(String[] args) {
        B objetB = new B() ;
        A objetA = new A(objetB) ;
        objetA.JeTravaillePourA() ;
    }

}
```

Relation unidirectionnelle permanente : auto-association

Exemple :



// Relation d'auto-association : A ---> A ou B ---> B
// Appel de méthodes de la classe par une autre méthode de la même classe

```
package relation_auto_association;
```

```
class A {
    private A a ;

    public A() { this.a = this ; }

    public void FaireQuelqueChose(int nombre){
        System.out.println(this.getClass().getName());
        System.out.println("Nombre : " + nombre);
        a.FaireAutreChose(nombre); }

    public void FaireAutreChose(int nombre){
        nombre++ ;
        System.out.println("Nombre : " + nombre); }
}
```

// Autre variante plus simple :

// Utilisation de l'instance courante de la classe : this

```
class B {

    public void FaireQuelqueChose(int nombre){
        System.out.println(this.getClass().getName());
        System.out.println("Nombre : " + nombre);
        this.FaireAutreChose(nombre); }

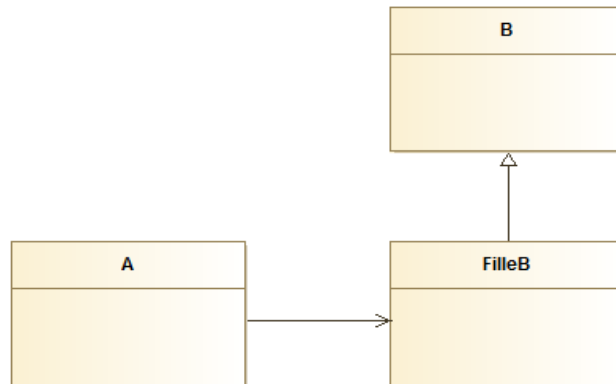
    public void FaireAutreChose(int nombre){
        nombre++ ;
        System.out.println("Nombre : " + nombre); }

}
```

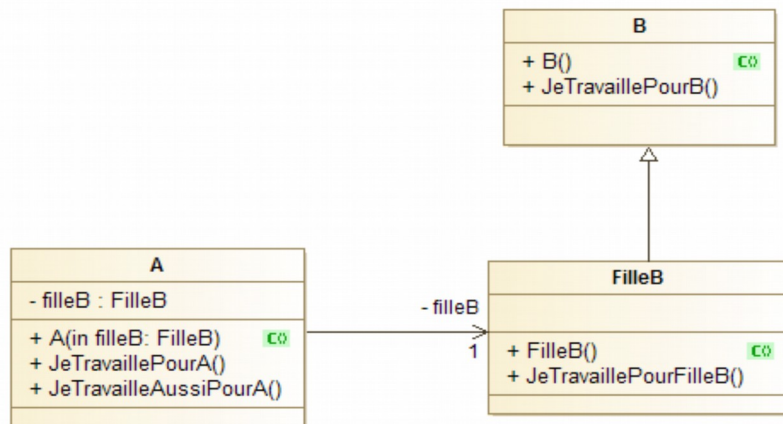
```
public class Main {

    public static void main(String[] args) {
        A objetA = new A();
        objetA.FaireQuelqueChose(1);
        B objetB = new B();
        objetB.FaireQuelqueChose(1);
    }
}
```

Héritage et relation unidirectionnelle



Exemple :



```
// Relation de la classe A vers une sous-classe FilleB qui hérite de B
// Surcharge de la méthode JeTravailleAussiPourA()
// plutôt que d'utiliser instanceof qui est fortement déconseillé
```

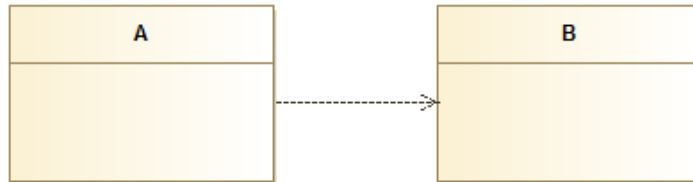
```
class A {
    private FilleB filleB ;
    public A(FilleB filleB) {
        this.filleB = filleB ; }
    public void JeTravaillePourA(){
        this.filleB.JeTravaillePourB() ;
        this.filleB.JeTravaillePourFilleB() ; }
    public void JeTravailleAussiPourA(FilleB b){
        b.JeTravaillePourB() ;
        b.JeTravaillePourFilleB() ; }
    public void JeTravailleAussiPourA(B b){
        b.JeTravaillePourB() ; }
}

class B {
    public B() {}
    public void JeTravaillePourB(){
        System.out.println("Je suis un service rendu par la classe B") ; }
}

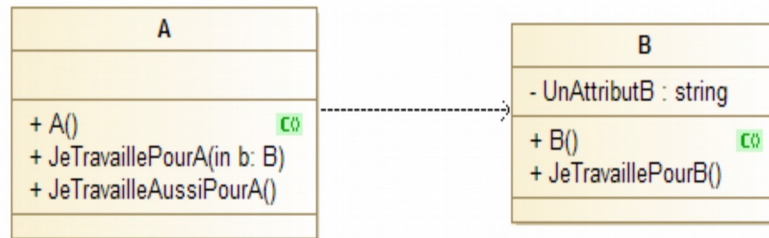
class FilleB extends B {
    public FilleB(){}
    public void JeTravaillePourFilleB(){
        System.out.println("Je suis un service rendu par la classe
FilleB") ; }
}

public class Main {
    public static void main(String[] args) {
        B objetB = new B() ;
        FilleB filleB = new FilleB() ;
        A objetA = new A(filleB) ;
        objetA.JeTravaillePourA() ;
        objetA.JeTravailleAussiPourA(objetB) ;
        objetA.JeTravailleAussiPourA(filleB) ; }
}
```

Relation unidirectionnelle ponctuelle (relation de dépendance)



Exemple :



```
// Relation unidirectionnelle ponctuelle : A ---> B
// Utilisation ponctuelle :
// la classe A peut ne jamais utiliser la classe B
// Objet de B dans une méthode de A
// ou bien on inclut la classe B dans une méthode de A
```

```
class A {

    public A() {}

    public void JeTravaillePourA(B b){
        b.JeTravaillePourB() ; }        // envoi vers B }

    public void JeTravailleAussiPourA(){
        B autre_b = new B() ;
        autre_b.JeTravaillePourB() ; } // envoi vers B }

}

class B {

    private String unAttribut ;

    public B() {}

    public void JeTravaillePourB(){
        System.out.println("Je travaille pour toutes les classes") ;
    }

}

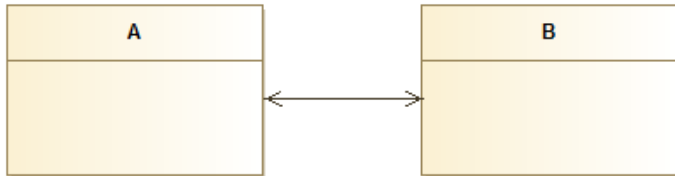
public class Main {

    public static void main(String[] args) {
        B objetB = new B() ;
        A objetA = new A() ;
        objetA.JeTravaillePourA(objetB) ;
        objetA.JeTravailleAussiPourA() ;
    }

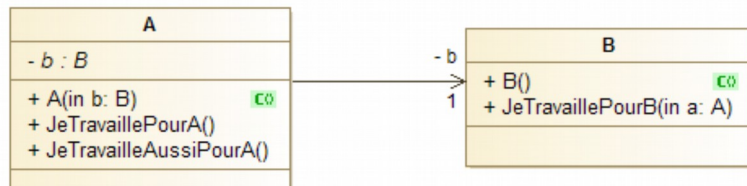
}
```

Relation bidirectionnelle (unidirectionnelle vers B et ponctuelle vers A)

Il est conseillé d'indiquer la navigabilité de l'association dans les deux sens.



Autre notation possible avec les valeurs de multiplicité :



```
// Relation bidirectionnelle : A <--> B
// unidirectionnelle vers B, ponctuelle vers A
class A {

    private B b ;

    public A(B b) {
        this.b = b ;
    }

    public void JeTravaillePourA(){
        System.out.println("Je travaille pour A") ;
        this.b.JeTravaillePourB(this) ; // envoi de message vers B
    }

    public void JeTravailleAussiPourA(){
        System.out.println("Je travaille aussi pour A") ; }
}

class B {

    public B() {}

    public void JeTravaillePourB(A a){ // association ponctuelle vers A
        System.out.println("Je travaille pour B") ;
        a.JeTravailleAussiPourA() ; // message vers A
    }
}

public class Main {

    public static void main(String[] args) {
        B objetB = new B() ;
        A objetA = new A(objetB) ;
        objetA.JeTravaillePourA() ;
        objetB.JeTravaillePourB(objetA) ;
    }
}
```

Relation d'agrégation

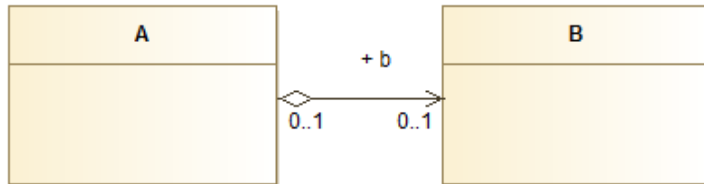
Association non symétrique dans laquelle une extrémité joue un rôle prépondérant par rapport à l'autre extrémité.

De l'objet agrégé B vers l'objet agrégeant A se traduit par : "est une partie de ...".

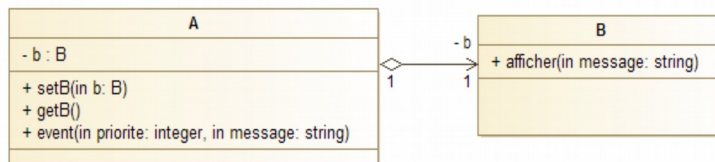
De l'objet agrégeant A vers l'objet agrégé B se traduit par : "utilise les services d'un ...".

Exemple : l'adhérent est membre (une partie d'un) d'un club, le club utilise les services d'un adhérent.

Relation d'agrégation (avec multiplicité unique) :



Exemple :



```
// Relation d'agrégation
// Association dans laquelle un objet est encapsulé dans un autre
// avec possibilité d'entrée-sortie. Présence de getters/setters.
// Cette relation est utilisée pour effectuer une délégation de responsabilités.
// Utilisations : propagation des valeurs d'attributs d'une classe vers une autre classe,
// action sur une classe qui implique une action sur une autre classe, subordination des
// objets, ...
// Différence sémantique entre association et agrégation, pas au niveau du code
```

```
class A {

    private B b ;

    public void setB(B b) {
        this.b = b ; }

    public void getB(){
        System.out.println("ObjetB " + this.b) ; }

    public void event(int priorite,String message){
        if (this.b instanceof B){
            String priorite_S = String.format("%d",priorite);
            String texte = "Priorite " +""+ priorite_S + " Message " + message ;
            this.b.afficher(texte);    // Affichage délégué à la classe B
        }
    }
}
```

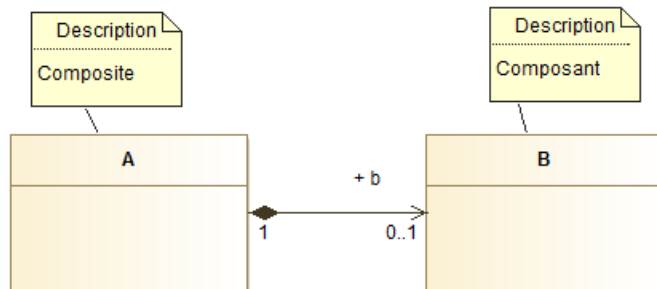
```
class B {

    public void afficher(String texte){
        System.out.println(texte) ; } }

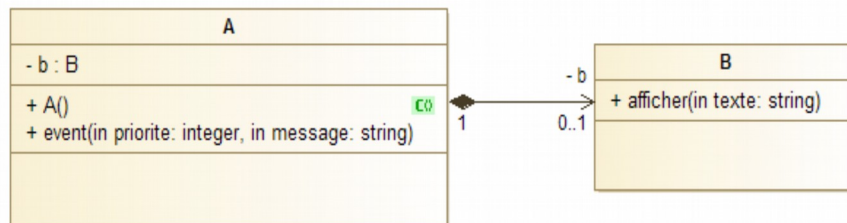
public class Main {
```

```
    public static void main(String[] args) {
        A objetA = new A() ;
        B objetB = new B() ;
        objetA.setB(objetB) ;
        objetA.getB() ;
        objetA.event(1,"toto") ;
    }
}
```


Relation de composition



Exemple :



```

// Relation de composition
// Couplage très fort entre deux classes
// La destruction de l'objet composite entraîne celle de l'objet composant
// "new" au sein du constructeur d'une classe et absence de get/set
  
```

```

class A {

    private B b ;

    public A() {
        this.b = new B() ;
    }

    public void event(int priorite,String message){
        if (this.b instanceof B){
            String priorite_S = String.format("%d",priorite);
            String texte = "Priorite " +""+ priorite_S + " Message " +
message ;

            this.b.afficher(texte);
            // Affichage délégué à la classe B
        }
    }
}

class B {

    public void afficher(String texte){
        System.out.println(texte) ;
    }
}

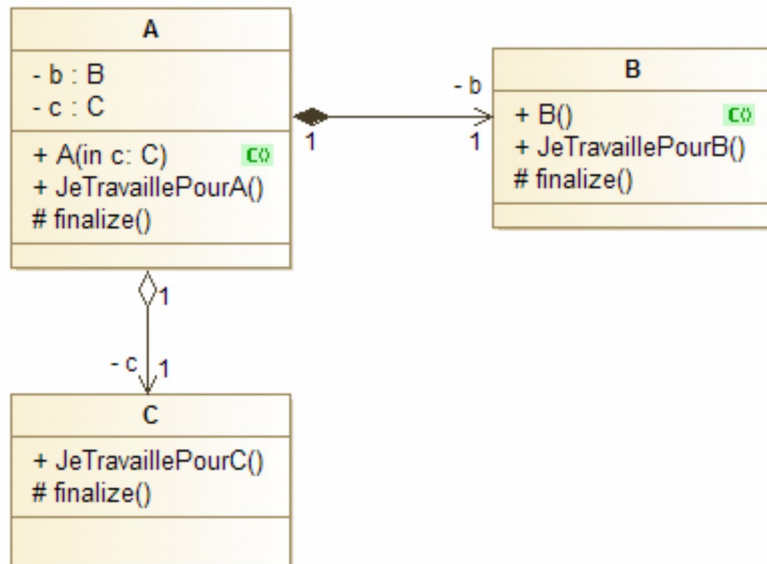
public class Main {

    public static void main(String[] args) {
        A objetA = new A() ;
        objetA.event(1,"toto") ;
    }
}
  
```

Relation de composition et d'agrégation

Exemple :

Ensemble d'objets à répétition (boucle for)
pour que le ramasse-miettes soit appelé automatiquement
et récupère un certain nombre d'objets devenus inutiles.
Pas d'appel explicite au ramasse-miettes :
il décide seul de son intervention quand la mémoire
commence à saturer.
Visualiser la sortie console pour voir les appels de la
méthode finalize().



```
class A {
    //private int unAttributA ;
    private B b ; // Il ne faut pas que ce lien apparaisse dans un "return"
    private C c ;

    public A(C c) {
        b = new B() ;// un lien de composition est créé
        this.c = c ;// un lien d'agrégation est créé}

    public void JeTravaillePourA() {
        b.JeTravaillePourB() ; // un message vers B
        c.JeTravaillePourC() ; // un message vers C
    }

    protected void finalize() { // Appel de cette méthode quand l'objet
        // est effacé de la mémoire
        System.out.println(" ... un objet A se meurt ... ");
    }
}

class B {
    B() {}
    public void JeTravaillePourB() {
        System.out.println("Je suis une instance de B au service de toutes
les classes"); }

    protected void finalize() { // Appel de cette méthode quand l'objet
        // est effacé de la mémoire
        System.out.println(" ... un objet B se meurt ... ");
    }
}

class C {
    public void JeTravaillePourC() {
        System.out.println("Je suis une instance de C au service de toutes
les classes"); }

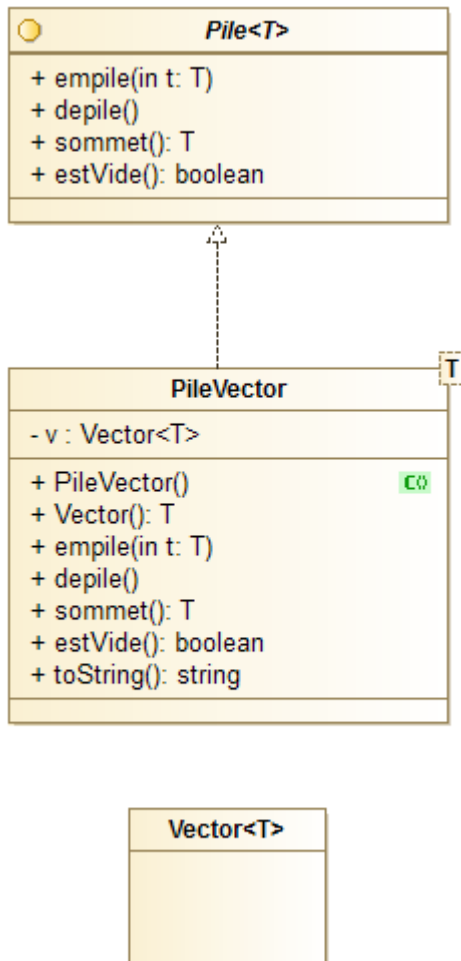
    protected void finalize() { // Appel de cette méthode quand l'objet
        // est effacé de la mémoire
        System.out.println(" ... un objet C se meurt ... ");
    }
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        C[] c = new C[1000000] ;  
        for (int i=0; i<1000000; i++) {  
            c[i] = new C() ;  
            A a = new A(c[i]) ;           // On passe le référent de l'objet  
C à l'objet A  
            a.JeTravaillePourA();  
            a = null ;                   // On cherche à se  
débarasser de l'objet A (instruction pas nécessaire)  
        }  
    }  
}
```

Généricité : Paramètre Template T

Permet d'utiliser une classe avec des types différents

Exemple :



```
public interface Pile<T> { // fichier Pile.java
    void empile(T t);
    void depile();
    T sommet();
    boolean estVide(); }
```

```
// Fichier PileVector.java
Voici une classe implémentant l'interface Pile
// à l'aide d'un vecteur pour stocker les éléments.
```

```
import java.util.Vector;
```

```
public class PileVector<T> implements Pile<T> {
    Vector<T> v=new Vector<T>();
    public PileVector(){}
    public void empile(T t){
        v.add(t); }
    public void depile(){
        v.remove(v.size()-1); }
    public T sommet(){
        return v.get(v.size()-1); }
    public boolean estVide(){
        return v.isEmpty(); }
    public String toString(){
        return "Pile "+v.toString(); }
}
```

```
// Fichier ProgrammePile.java
// La fonction main montre comment on crée une pile
// en passant un paramètre de type réel (Integer).
// Cela s'appelle la générique : T est un paramètre de générique.
```

```
public class ProgrammePile {
    public static void main(String[] a) {
        Pile<Integer> p1 = new PileVector<Integer>();
        p1.empile(7);
        p1.empile(5);
        p1.empile(4);
        System.out.println(p1);
        p1.depile();
        System.out.println(p1);
    } }
```


Les énumérations

Une énumération se déclare comme une classe, en remplaçant le mot-clé "class" par "enum".

Les énumérations héritent de la classe java.lang.Enum.

Chaque élément d'une énumération est un objet à part entière.

Exemple :

12...	Langage
	JAVA C CPlus PHP
	- name : string - editor : string
	Langage(in name: string, in editor: string)  getName() getEditor() toString(out name: string)

```
public enum Langage {  
  
    //Objets directement construits  
  
    JAVA("Langage JAVA", "Eclipse"),  
    C ("Langage C", "Code Block"),  
    CPlus ("Langage C++", "Visual studio"),  
    PHP ("Langage PHP", "PS Pad");  
  
    private String name = "";  
    private String editor = "";  
  
    //Constructeur  
  
    Langage(String name, String editor){  
        this.name = name;  
        this.editor = editor;  
    }  
  
    public void getName(){  
        System.out.println("Nom : " + toString());  
    }  
  
    public void getEditor(){  
        System.out.println("Editeur : " + editor);  
    }  
  
    public String toString(){  
        return name;  
    }  
  
    public static void main(String args[]){  
        Langage l1 = Langage.JAVA;  
        Langage l2 = Langage.PHP;  
        l1.getName();  
        l2.getName();  
        l1.getEditor();  
        l2.getEditor();  
    }  
}
```

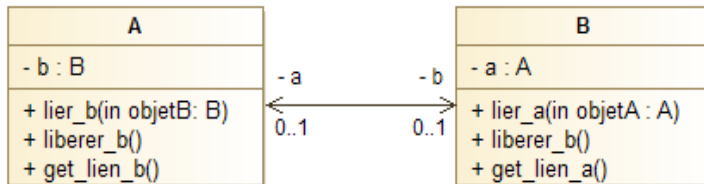
Relation d'association bidirectionnelle

La seule présence des attributs **a** et **b** ne suffit pas à assurer une association bidirectionnelle valide.

Rien n'empêche d'avoir une instance X de A associée, par l'intermédiaire de l'attribut b, à une instance Y de B elle-même associée, par l'intermédiaire de l'attribut a, à une autre instance Z de A.

Or, si X est associé à Y, Y doit être associé à X et non pas à Z. On doit assurer un comportement cohérent.

Exemple :



```
// Relation d'association bidirectionnelle
// Cette implémentation dépendra du contexte, elle peut être factorisée
```

```
public class A {

    private B b ;

    public void lier_b(B objetB){
        if(objetB !=null && b != objetB){
            this.liberer_b();
            b = objetB ;
            b.lier_a(this);
        }
    }
    public void liberer_b(){
        if (b !=null){
            B objetB = b ;
            b = null ;
            objetB.liberer_a();
        }
    }
    public void get_lien_b(){
        System.out.println("ObjetB " + this.b) ; }
}

public class B {

    private A a ;

    public void lier_a(A objetA){
        if(objetA !=null && a != objetA){
            this.liberer_a();
            a = objetA ;
            a.lier_b(this);
        }
    }
    public void liberer_a(){
        if (a !=null){
            A objetA = a ;
            a = null ;
            objetA.liberer_b();
        }
    }
}
```

```
        public void get_lien_a(){
            System.out.println("ObjetA " + this.a) ; }
    }

    public class Main {

        public static void main(String[] args) {
            A objetA = new A() ;
            A objetAA = new A();
            B objetB = new B() ;
            objetA.lier_b(objetB) ;
            objetB.lier_a(objetA) ;
            System.out.println("ObjetA " + objetA) ;
            System.out.println("ObjetB " + objetB) ;
            objetB.get_lien_a();
            objetB.lier_a(objetAA) ;
            objetB.get_lien_a();
        }
    }
```