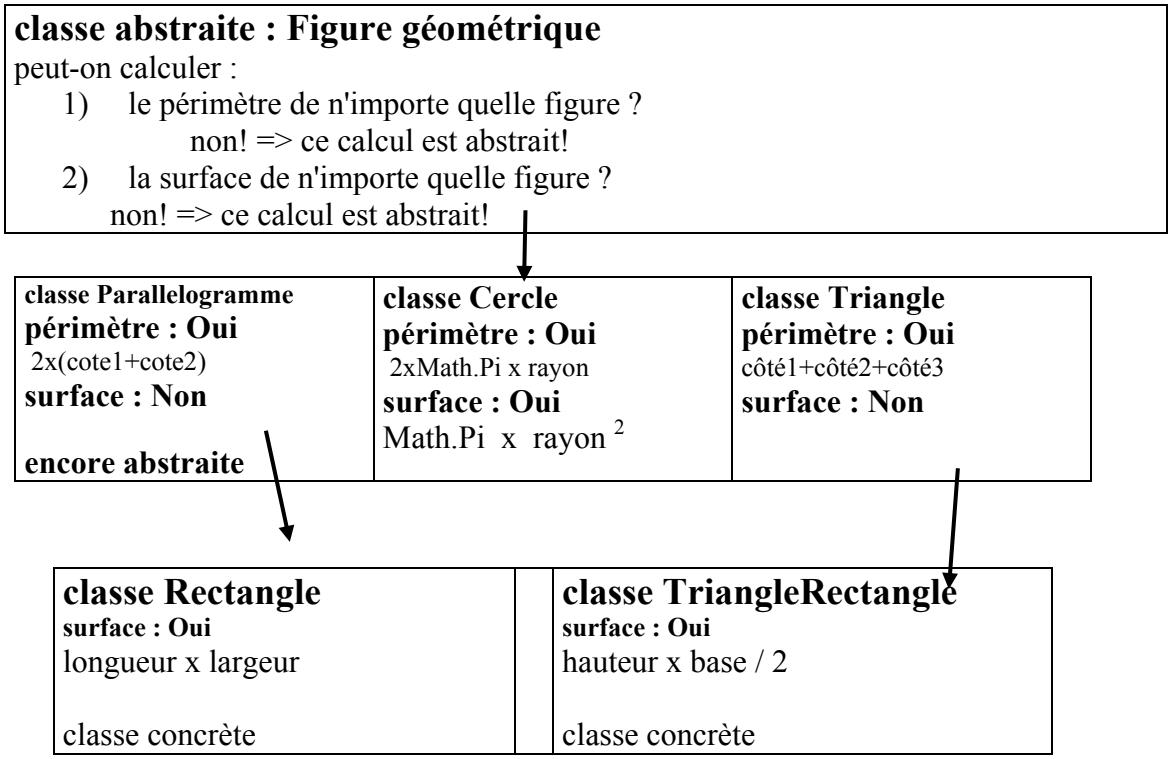
Classe abstraite (abstract class):

Pourquoi ?

Lors de la conception d'une hiérarchie, on a besoin de créer des classes plus générales d'abord et différer l'implémentation à des étapes plus éloignées quand le comportement est très bien défini. Quand on parle de figure géométrique, il est trop vague pour savoir comment calculer un périmètre, une surface : **ces méthodes sont encore abstraites**. Il est impossible de réaliser (implémenter) ces méthodes. Par contre, un rectangle est une figure géométrique dont le calcul du périmètre et de la surface est connu, on peut alors implémenter ces méthodes qui sont très concrètes (pas du tout abstraites) pour un rectangle.



En résumé : une classe abstraite est celle qui n'est que **partiellement implantée** et dont l'intérêt est la **conception d'une hiérarchie de l'héritage simple**.

```
/** classe abstraite FigureGeometrique
 *  1. Cette classe contient au moins une méthode abstraite
 *     la classe est donc abstraite
 *  2. On n'a pas de formule pour calculer le périmètre
 *     ni de surface de n'importe quelle figure.
 *     À cette étape-ci, les deux méthodes sont encore abstraites
 *  3. On n'a pas le droit d'avoir des objets construits (avec new)
 *     d'une classe abstraite
 *  4. On a le droit d'avoir des méthodes non abstraites (concrètes)
 *     (la réalisation de la méthode est faite au complet,
 *      exemple toString() de FigureGeometrique)
 */
```

```
public abstract class FigureGeometrique
{
public abstract double perimetre();
public abstract double surface();

public String toString() {
return "Perimetre : " + perimetre() + "\n" +
"Surface   : " + surface() + "\n";
}}

/**
 *   Pour un parallélogramme (côtés opposés parallèles) :
 *    - le calcul du périmètre est concret : 2 x (côté1 + côté2)
 *    - le calcul de la surface est encore abstraite
 *      la classe Parallélogramme est encore abstraite
 *
 *   On n'a pas le droit d'avoir d'objets construits avec new
 *   pour la classe abstraite Parallélogramme
 *   Cependant, on a le droit d'avoir des constructeurs!
 *   (pour appeler avec this(...) ou super(....)
 */

public abstract  class Parallelogramme extends FigureGeometrique
{
private double cote1, cote2 ;

public Parallelogramme(double cote1, double cote2) {
      this.cote1 = cote1;
      this.cote2 = cote2;
   }

public double perimetre() { return 2 * (cote1+cote2); }

          // public abstract double surface();

          public double getCote1() { return cote1; }
          public double getCote2() { return cote2; }
}

/**
 *   Les 2 méthodes :
 *      perimetre() : hériter de Parallelogramme
 *      surface() : implémentée ici (réaliser de A à Z)
 *   La classe Rectangle2 n'est plus abstraite. Elle est
 *   concrète.
 */
```

```
public class Rectangle2 extends Parallelogramme
{
        // appel le constructeur d'une super-classe (qui est abstraite!)

public Rectangle2(double lo, double la) {
                 super(lo, la);

                // ainsi côté1 <-> longueur,côté2 <-> largeur
        }

        public double surface() {
                return getCote1() * getCote2();
        }

        public String toString() {
                return "Rectangle : <longueur = " + getCote1() +
                        ", largeur = " + getCote2() + ">\n" +
                        super.toString();
        // appel d'une méthode de la super-classe
           /* notez que Java ne trouve pas toString() dans
         Parallelogramme. Il va monter jusqu'à FigureGeometrique
         et applique le toString() de FigureGeometrique
              C'est le polymorphisme : appliquer la bonne méthode
           */
        }

}

// Révision de l'héritage, du polymorphisme

public class Carre2
        extends Rectangle2
{
        public Carre2(double cote) {
                 super(cote,cote);
        }

        public double diagonale() {
                return getCote1() * Math.sqrt(2.0);
        }

        public String toString() {
                 return super.toString() + "Diagonale : " +
                diagonale() + "\n";
        }
}
/**
 *   Circle.java
 *
 *   Les 2 méthodes :
```

```
 *      perimetre() : implémentée ici (réaliser de A à Z)
 *      surface() : implémentée ici (réaliser de A à Z)
 *   La classe Circle n'est pas abstraite. Elle est
 *   concrète.
 */

public class Circle  extends FigureGeometrique
{
  private double rayon;

  public Circle( double rayon ) {
            this.rayon = rayon;
  }

  public double perimetre() { return 2 * Math.PI * rayon; }
  public double surface() { return Math.PI * rayon * rayon;}

  public String toString() {
           return "Cercle de rayon " + rayon + "\n" +
                              super.toString();
  }}

/**  classe TestAbstraite
 *   Pour cet exemple, on se  limite au minimum pour faire
 *   ressortir le concept de classe abstraite.
 *
 *   Exercice :
 *     ajouter une classe Triangle. Pourquoi est-elle abstraite ?
 *     ajouter une sous-classe TriangleRectangle.
 *     Pourquoi est-elle concrète ?
 *     Tester le bon fonctionnement
 */


public class TestAbstraite
{
        static double surfaceTotale(FigureGeometrique[] tableau) {
                double totSurface = 0.0;
                for (int i = 0 ; i < tableau.length ; i++)
                        totSurface += tableau[i].surface();
                return totSurface;
        }

        public static void main (String[] args)
        {
          /*
                        Message d'erreur du genre suivant:
                        L'operateur new est interdit pour une instance de la
                        classe abstraite FigureGeometrique

                        FigureGeometrique fg = new FigureGeometrique();
```

```
        */

            Rectangle2 rect1 = new Rectangle2(10.0,6.5);
            System.out.println("Infos de rect1 :\n" + rect1);

            Circle c1 = new Circle(10.0);
            System.out.println("Infos du cercle c1 :\n" + c1);


            FigureGeometrique[] tabFG =
          { new Rectangle2(10.0, 4.0),
                    new Circle(5.0),
                    new Rectangle2(8.6, 4.4),
                    new Rectangle2(14.2, 10.8),
                 new Carre2(6.0)};

            System.out.println("Surface totale pour acheter " +
          "de la peinture:  " +
                                        surfaceTotale(tabFG));
      }
}
/* Exécution:

Infos de rect1 :
Rectangle : <longueur = 10.0, largeur = 6.5>
Perimetre : 33.0
Surface   : 65.0

Infos du cercle c1 :
Cercle de rayon 10.0
Perimetre : 62.83185307179586
Surface   : 314.1592653589793

Surface totale pour acheter de la peinture:  345.73981633974483
*/
```

Règles à respecter d'une classe abstraite :

- **pas d'instanciation (trop abstraite pour la**
  - **construction d'un objet avec new)**
- une classe abstraite a le droit des constructeurs
  - (qui seront appelés par autres constructeurs avec
  - this(…) ou super(…))
- une classe abstraite doit contenir **au moins une**
  - déclaration de **méthode abstraite (cas très fréquent)**
  - ou de **variable abstraite**
- **une méthode abstraite** déclarée dans la classe
- abstraite doit être implémentée dans une de ses
- sous-classes (pas nécessairement dans toutes ses sous-classes)

- la classe dans laquelle une méthode abstraite est déclarée ne comporte pas de code pour cette méthode.
- Il n'est pas nécessaire d'implémenter une méthode abstraite dans chaque sous-classe.
- une classe abstraite ne peut pas être déclarée comme **final** (pas de sous-classes) ou **private** (car on ne peut pas utiliser ses méthodes à l'extérieur de cette classe abstraite).

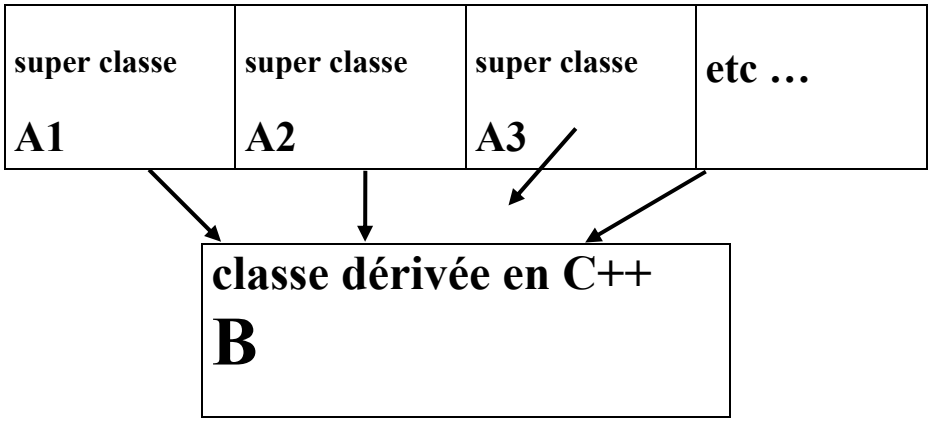
Déclaration :

```
visibilité  abstract class  nomDeClasse {
        ….
        visibilité abstract type nomDeMethode(liste d'arguments) ;
        ….
    }
```

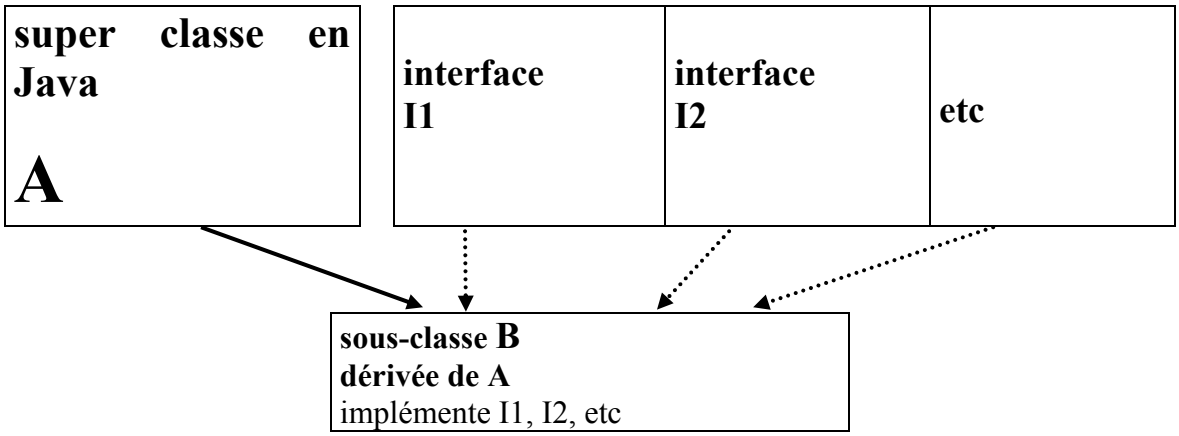
Interface :

Pourquoi ?

Le Java ne supporte pas **l'héritage multiple** comme C++ :

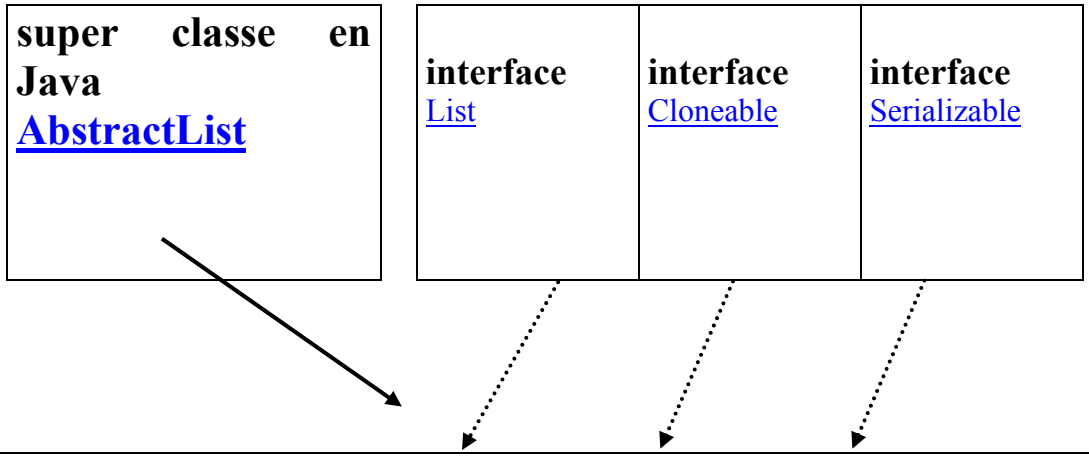


En Java une classe n'a pas le droit d'hériter directement deux super-classes. Par contre, Java permet à une classe d'être **dérivée d'une seule super-classe** mais **implémente une ou plusieurs interfaces**.



De plus, une interface contient souvent ses méthodes abstraites qui présentent certaines **comportements assez générales** (**hasMoreElements()** : avoir encore d'éléments, **nextElement()** : prochain élément pour les structures linéaires, **writeInt()** : écrire un entier, **writeDouble()** : écrire un réel, … pour les fichiers binaires, **run()** : déplacer, pour animation, …). Ces méthodes abstraites seront réalisées dans les classes qui implémentent cette interface.

Cas de Java : Vector



public class Vector
extends AbstractList
implements List, Cloneable, Serializable

(note : Vector n'implémente pas l'interface Enumeration mais une de ses méthodes retourne un objet de la « classe » Enumeration)

. . .
// cette méthode retourne une énumération des éléments de ce
// vecteur
public final Enumeration elements();
. . .

Pour afficher tous les éléments d'un vecteur v :

```
for (Enumeration elem  = v.elements() ; elem.hasMoreElements() ;  )
        System.out.println(elem.nextElement());
```

Programmer nous-même les interfaces :

Pour trier un tableau des personnes selon leurs tailles, la méthode de tri suivant fonctionne :

```
public static void trier(Personne [] pers, nbPers) {
    Personne tempo;
    for(int i = 0 ; i < nbPers-1; i++) {
        int indMin = i;
        for(int j = i+1 ; j < nbPers; j++)
            if (pers[j].getTaille() < pers[indMin].getTaille() )
                indMin = j;
        if (indMin != i) {
            tempo = pers[i];
            pers[i]=pers[indMin];
            pers[indMin]=tempo;
        } }}
```

Pour trier un tableau des rectangles selon leurs surfaces, la méthode de tri suivant répond aussi à nos besoins :

```
public static void trier(Rectangle [] rect, nbRect) {
    Rectangle tempo;
    for(int i = 0 ; i < nbRect-1; i++) {
        int indMin = i;
        for(int j = i+1 ; j < nbRect; j++)
            if (rect[j].getSurface() < rect[indMin].getSurface() )
                indMin = j;
        if (indMin != i) {
            tempo = rect[i];
            rect[i] = rect[indMin];
            rect[indMin] = tempo;
        }
    }
}
```

La seule différente entre ces deux méthodes est la
Comparaison selon les tailles ou selon les surfaces.

Peut-on écrire une seule méthode de tri ?

Supposons qu'on déclare l'interface suivante :

```
interface Comparable
{
    public abstract boolean plusPetit(Object obj);
}
```

Supposons aussi que les classes Personne, Rectangle, … implémentent l'interface Camparable (en réalisant au complet la méthode plusPetit dans Personne, dans Rectangle, etc . . .).
Il suffit d'écrire une seule méthode de tri comme la suivante :

// comment trier un tableau des objets COMPARABLES ?

public class Tri

// tri par sélection, basée sur la méthode plusPetit de l'interface Comparable

```
{
public static void trier(Comparable[] tableau, int nbElem) {
        Comparable tempo;
        for(int i = 0 ; i < nbElem-1; i++) {
                int indMin = i;
                for(int j = i+1 ; j < nbElem; j++)
                        if (tableau[j].plusPetit(tableau[indMin]))
                                indMin = j;
                if (indMin != i) {
                        tempo = tableau[i];
                        tableau[i]=tableau[indMin];
                        tableau[indMin]=tempo;
                } } }}
```

Pour trier les personnes : Tri.trier(pers, nbPers) ;
Pour trier les rectangles : Tri.trier(rect, nbRect) ;
ect …..

```
/** Doit-on écrire plusieurs fonctions de tri pour :
 *    - trier des personnes selon leur taille ?
 *    - trier des rectangles selon leur surface ?
 *    - trier des pays selon leur capital ?
 *    - trier des . . . ?
 *
 *  Réponse : NON, on peut utiliser une seule méthode
 *  Comment ?
 *     Pour trier, il faut COMPARER. Cependant, l'opérateur
 *        <  (est plus petit que) n'est pas défini sur les objets
 *     On déclare une INTERFACE dont la méthode plusPetit n'est pas
 *     "réalisée" :
 *
 *          interface Comparable {
 *                                        boolean plusPetit(Object c);
 *          }
 *
 *     Chacune de ses sous-classes (exemple Personne,
 *      Rectangle, ...) doit s'ENGAGER à définir (réaliser) cette
 *      méthode
 *
 *     Dans cet exemple, on n'a pas la notion d'héritage multiple */
```

```
public class TestInterface
{
	// afficher le tableau d'objets (personnes, rectangles, ...)
	static void afficher(Object [] obj, String message) {
		System.out.println(message);
		for(int i = 0 ; i < obj.length ; i++)
			System.out.println(obj[i]);
		System.out.println();
	}

	public static void main (String[] args)
	{
	  Personne2 pers[] = { new Personne2(1.75, 65.3, 'M'),
				new Personne2(1.62, 69.1, 'F'),
				new Personne2(1.89, 76.5, 'F'),
				new Personne2(1.45, 50.3, 'M'),
				new Personne2(1.77, 90.1, 'M')
			};
		afficher(pers, "Lite des personnes avant le tri");
		Tri.trier(pers, pers.length);
		afficher(pers, "Lite des personnes apres le tri");

		Rectangle3 rect[] = { new Rectangle3(12, 5),
					new Rectangle3(9, 6),
					new Rectangle3(60, 12),
					new Rectangle3(5, 12)};

afficher(rect, "Lite des rectangles avant le tri" +
			" selon la surface : ");
	  Tri.trier(rect, rect.length);
	  afficher(rect, "Lite des rectangles apres le tri selon" +
			" la surface : ");
	}
}

/* Exécution :
Lite des personnes avant le tri
M mesure 1.75 metre et pese 65.3 kgs
F mesure 1.62 metre et pese 69.1 kgs
F mesure 1.89 metre et pese 76.5 kgs
M mesure 1.45 metre et pese 50.3 kgs
M mesure 1.77 metre et pese 90.1 kgs

Lite des personnes apres le tri
M mesure 1.45 metre et pese 50.3 kgs
F mesure 1.62 metre et pese 69.1 kgs
M mesure 1.75 metre et pese 65.3 kgs
M mesure 1.77 metre et pese 90.1 kgs
F mesure 1.89 metre et pese 76.5 kgs
```

```
Lite des rectangles avant le tri selon la surface :
<longueur : 12, 5, surface : 60>
<longueur : 9, 6, surface : 54>
<longueur : 60, 12, surface : 720>
<longueur : 5, 12, surface : 60>

Lite des rectangles apres selon la surface :
<longueur : 9, 6, surface : 54>
<longueur : 12, 5, surface : 60>
<longueur : 5, 12, surface : 60>
<longueur : 60, 12, surface : 720>
*/

public class Personne2 implements Comparable
{ private double taille, poids;
  private char sexe ;
  public Personne2(double t, double p, char s) {
            taille = t ;
            poids = p;
            sexe = s;
  }

  public double getTaille(){
            return taille;
  }

  public char getSexe(){
            return sexe;
  }

  public String toString() {
            return sexe + " mesure " + taille + " metre et pese " + poids +
                     " kgs";
  }

  // On s'engage à implémenter la méthode "plusPetit" :
  public boolean plusPetit(Object p) {
             return this.taille < ( (Personne2) p).getTaille();
  }
}


public class Rectangle3 implements Comparable
{   // attributs (champs de données, membres de données)
          private int longueur, largeur;

          public Rectangle3() { }
          public Rectangle3(int lo, int largeur) {
                    longueur = lo;
                    this.largeur = largeur;
```

```
    }

    public Rectangle3(int c) {
        this(c, c);
    }

    // méthode pour calculer et retourner le périmètre
    public int perimetre(){
        return 2 * (longueur + largeur);
    }

    // méthode pour calculer et retourner la surface
    public int surface() {
        return longueur * largeur ;
    }

    public int getLongueur() { return longueur ; }
    public int getLargeur() { return largeur ; }


    // On s'engage à IMPLÉMENTER la méthode "plusPetit"   public
  public boolean plusPetit(Object r) {
        return this.surface() < ( (Rectangle3) r).surface();
    }

    public String toString() {
    return "<longueur : " + longueur + ", " + largeur + ", surface : " + surface() + ">";
    }

}
```

interface Comparable
{
public abstract boolean plusPetit(Object obj);
}

Classes concrètes vs abstraites vs interfaces :

	Concrète	Abstraite	Interface
Création une instance avec new	Oui	Non	Non
Supporte des méthodes abstraites	Non	Oui	Oui
Supporte de l'héritage multiple	Non	Non	Oui
Droits des constructeurs	Oui	Oui	Oui
Implémentation d'une méthode	nécessaire	Oui si non abstraite	Non car toutes sont abstraites
Contenu	Illimité	Illimité	public static final et Méthodes abstraites