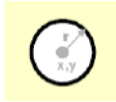
Part 5

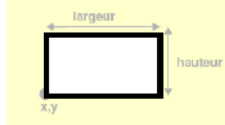
CLASSES ABSTRAITES, INTERFACES ET CLASSES IMBRIQUÉES

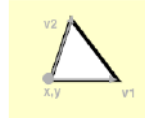
Classes Abstraites

Introduction

- **Exemple introductif :** les formes géométriques.
  - Besoin : application de manipulation de formes géométriques (triangles, rectangles, cercles, …)
  - Chaque forme est définie par sa position dans le plan, peut être déplacée, peut calculer son périmètre et sa surface.
  - Certaines formes peuvent avoir en plus des longueurs pour les côtés.

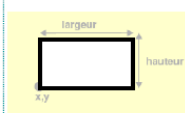




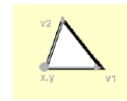


Introduction

Attributs:
double x,y; //centre du cercle
double r; // rayon
Méthodes:
deplacer(double dx, double dy)
double surface()
double périmètre()

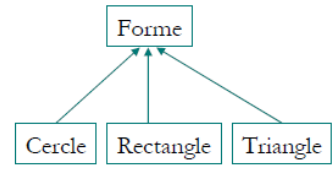


Attributs:
double x,y; //coin inf. gauche
double largeur, hauteur;
Méthodes:
deplacer(double dx, double dy)
double surface()
double périmètre();



Attributs:
double x,y; //un des sommets
double x1,y1; // vecteur 1
double x2,y2; // vecteur 2
Méthodes:
deplacer(double dx, double dy)
double surface()
double périmètre();

Introduction



- Factorisation du code

```
class Forme {
    protected double x,y;
    ……
    public void deplacer(double dx, double dy) {
      x += dx; y += dy;
    }
}
```

```
class Cercle extends Forme {
   protected double r;
   ……
   public double surface(){ return Math.PI * r * r; }
   public double perimetre(){ return 2 * Math.PI * r; }
}
```

Introduction

- Un besoin: profiter du polymorphisme

```
public class ListeDeFormes {
   private Forme[] tabForme = new Forme[10];
   private int nbFormes = 0;
   public void ajouter(Forme f) {
      tabForme[nbFormes++] = f ;
   }
   public void toutDeplacer(double dx,double dy) {
      for (int i=0; i < NbFormes; i++)
         tabFrome[i].deplace(dx,dy);
   }
. . . . . .
```

Polymorphisme: Prise en compte de nouveaux types de forme sans modification du code.

Introduction

- **Problème**

```
 . . .
public double perimetreTotal() {
  double pt = 0.0;
  for (int i=0; i < NbFormes; i++)
     pt += tabForme[i].perimetre(); // ERREUR
  return pt;
}
```

La méthode perimetre() n'est pas définie dans la classe Forme (type déclaré) : appel non valide!!

Introduction

- **Solution ?**
  - Définir une méthode perimetre() dans Forme ?

```
public double perimetre(){ return 0.0; // ?? }
```

  - Les classes abstraites sont une solution propre et élégante.
    - Servent à définir des concepts incomplets qui devront être implémentés dans les sous-classes.

```
public abstract class Forme { // Classe abstraite
   protected double x,y;
   public void deplacer(double dx, double dy) {
      x += dx; y += dy;
   }
   // Méthodes abstraites
   public abstract double perimetre();
   public abstract double surface() ;
}
```

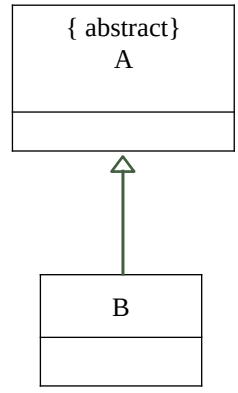
Classes abstraites

- Une classe abstraite est une classe qui ne permet pas d'instancier des objets.
- Elle ne peut servir que de classe de base pour une dérivation.

- Elle se déclare à l'aide du mot-clé **abstract:**

```
// Declaration d'une classe abstraite
public abstract class NomDeLaClasse {
   ...
}
```


Classe abstraite: Notation UML



- La classe B **hérite** (est une **sous-classe**) de la classe A

Contenu d'une classe abstraite

- Une classe abstraite peut contenir des champs et des méthodes comme toute autre classe
- Mais elle peut également contenir des méthodes abstraites
- Une méthode est dite abstraite (ou différée) si :
  - seule sa déclaration est fournie (signature et type de retour)
  - elle ne dispose pas d'implémentation
  - elle est destinée à être redéfinie dans une classe dérivée

```
public abstract class A {
   public void f() { ..... } // f est définie dans A
   public abstract void g(int n);
   /* g n'est pas définie dans A ; on n'en a fourni
   que l'en-tête */
}
```


Dérivation et instanciation d'une classe abstraite

```
public abstract class A{
   public void f() { ..... } // f est définie dans A
   public abstract void g(int n); // méthode abstraite
}

public class B extends A{
   public void g(int n) { ..... } // ici, on définit g
   .....
}
```

```
A a ;                 // OK : a n'est qu'une référence sur un
                         objet de type A ou dérivé
a = new A(...) ;      // ERREUR: pas d'instanciation d'objets
                         d'une classe abstraite
A a = new B(...) ; // OK
B b = new B(...) ; // OK
```

Règles pour la définition des classes abstraites

- Dès lors qu'une classe contient au moins une méthode abstraite, elle est abstraite.
- Une classe abstraite ne peut être instanciée, car elle contient au moins une méthode qui n'a pas d'implémentation.
- Une méthode abstraite doit être publique, car elle est destinée à être redéfinie dans une classe dérivée.
- Une classe dérivée d'une classe abstraite doit implémenter toutes les méthodes abstraites de sa classe mère, ou sinon elle reste abstraite.
- Il suffit donc que la classe dérivée ne redéfinisse aucune ou une des méthodes abstraites de sa classe mère pour rester elle-même abstraite.

Exemple complet (1/4)

```
//classe abstraite Forme
abstract class Forme{
 protected int x, y;
 public Forme(int x, int y){
    this.x = x;
    this.y = y;
 }
 public void deplacer( int dx, int dy){
   x+=dx;
   y+=dy;
 }
 public abstract double perimetre();
 public abstract double surface();
 public abstract void dessiner();
}
```

Exemple complet (2/4)

```
//classe Rectangle dérivée de la classe Forme
class Rectangle extends Forme{
 private int largeur, hauteur;
 public Rectangle(int x, int y, int la, int ha){
    super(x, y);
    largeur = la;
    hauteur = ha;
 }
 public double perimetre(){
   return 2*(largeur+hauteur);
 }
 public double surface(){
   return largeur*hauteur;
 }
 public void dessiner(){
   System.out.print("Rectangle à la position " + x +","+y);
   System.out.println(" et de dimensions " + largeur +","+ hauteur + " est dessiné!");
 }
}
```


Exemple complet (3/4)

```
//classe Cercle dérivée de la classe Forme
class Cercle extends Forme{
 private int rayon;
 public Cercle(int x, int y, int r){
     super(x, y);
     rayon = r;
 }
 public double perimetre(){
    return 2*Math.PI*rayon;
 }
 public double surface(){
    return Math.PI*rayon*rayon;
 }
 public void dessiner(){
    System.out.print("Cercle à la position " + x +","+y);
    System.out.println(" et de rayon " + rayon +" est dessiné!");
 }
}
```

Exemple complet (4/4)

```
//classe contenant le main
public class ListeDeFormes{
    private Forme[] tabForme = new Forme[10];
    private int nbFormes = 0;
    // Ajouter une forme au tableau
    public void ajouter(Forme f) {
    // Calcul du périmètre total
    public double perimetreTotal() {
    // Deplacer toutes les formes
    public void toutDeplacer(int dx,int dy) {
    // Dessiner toutes les formes
    public void toutDessiner() {
    public static void main (String[] args){
        ListeDeFormes dessin = new ListeDeFormes();
        dessin.ajouter(new Rectangle(0,0,2,1));
        dessin.ajouter(new Rectangle(5,1,3,2));
        dessin.ajouter(new Cercle(1,3,5));
        dessin.ajouter(new Cercle(2,2,4));
        dessin.toutDeplacer(10,10);
        dessin.toutDessiner();
        System.out.println("Le périmètre total des formes est : " + dessin.perimetreTotal());
    }
}
```


Exercice 7

- Dans l'exemples ci dessous, qu'est-ce que vous mettriez à la place de xxx ? (Il peut s'agir d'une chaîne quelconque.)

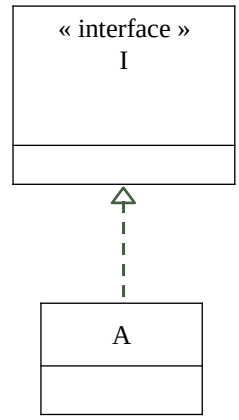
```
xxx A {
  xxx void hello() {
    System.out.println("Hello");
  }
  xxx String bye();
  public abstract void affiche();
}
xxx B xxx A {
  xxx void hello() {
    System.out.println("Hello World");
  }
  xxx String bye(){
   return ("bye");
  }
}
```

Interfaces

Interfaces

- Une interface est une « classe » purement abstraite dont **toutes** les méthodes sont abstraites et publiques.
  - Une façon de décrire **ce que** les classes doivent faire, sans préciser **comment** elles doivent le faire.
- Une interface ne peut contenir que
  - des méthodes abstract et public
  - des définitions de constantes publiques (« public static final »)
- Une interface peut avoir la même accessibilité que les classes :
  - public : utilisable de partout
  - sinon : utilisable seulement dans le même package

Interface: Notation UML



- La classe A **implémente** l'interface I

La définition d'une interface

- La définition d'une interface se présente comme celle d'une classe. On y utilise le mot-clé interface à la place de class:

```
public interface I {
   static final int MAXI = 100 ;
   // en-tete d'une methode f
   public abstract void f(int n);
   // en-tete d'une methode g
   public abstract void g() ;
}
```

Il n'est pas nécessaire de mentionner les mots-clés *public et abstract* ils sont facultatifs.

Implémentation d'une interface

- Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot-clé implements, comme dans :

```
public class A implements I {
  // A doit définir les méthodes f et g prévues dans
  l'interface I
}
```

- 2 seuls cas possibles :
  - soit la classe **A** implémente **toutes** les méthodes de I
  - soit la classe **A** doit être déclarée abstract ; Les méthodes manquantes seront implémentées par les classes filles de A.

Implémentation de plusieurs interfaces

- Une même classe peut implémenter plusieurs interfaces :

```
public interface I1{
  void f() ;
}
```

```
public interface I2{
  int h() ;
}
```

```
public class A implements I1, I2{
  // A doit obligatoirement définir les méthodes
  f et h prévues dans I1 et I2
}
```

Variables de type interface

- Une interface peut **seulement** servir à déclarer une variable, un paramètre, une valeur retour, un type de base de tableau, …

```
public interface I { .....}
.....
I i ; /* i est une référence à un objet d'une classe
implémentant l'interface I */
```

- En revanche, on pourra affecter à i n'importe quelle référence à un objet d'une classe implémentant l'interface I.

```
class A implements I { ..... }
.....

I i = new A(...) ; // OK
```


Interface et classe dérivée

- La clause implements est totalement indépendante de l'héritage.
- Une classe dérivée peut implémenter une interface (ou plusieurs) :

```
public interface I {
  void f(int n) ;
  void g() ;
}
```

```
public class A {
  .....
}
```

```
public class B extends A implements I {
  /* les méthodes f et g doivent soit être déja
  définies dans A, soit définies dans B */
}
```

Dérivation d'une interface

- On peut définir une interface comme une généralisation d'une autre.
- On utilise là encore le mot-clé extends, ce qui conduit à parler d'héritage ou de dérivation.

```
public interface I1{
   static final int MAXI = 100 ;
   void f(int n) ;
}
```

```
public interface I2 extends I1{
   static final int MINI = 20 ;
   void g() ;
}
```

La définition de I2 est équivalente à

```
public interface I2 {
   static final int MAXI = 100 ;
   static final int MINI = 20 ;
   void f(int n) ;
   void g() ;
}
```


Exemple complet(1/3)

- On veut définir:
  - Une interface nommée Affichable, dotée d'une seule méthode affiche.
  - Deux classes Entier et Flottant implémentent cette interface (aucun lien d'héritage n'apparaît ici)
    - Chaque classe est dotée d'un attribut **valeur**
    - La méthode **affiche()** affiche un message de l'ordre: "Je suis un flottant/entier de valeur xy")
  - La méthode main utilise un tableau hétérogène d'objets de type Affichable qu'elle remplit en instanciant des objets de type Entier et Flottant puis les affiche.

```java
// Interface Affichable
interface Affichable{
    void affiche() ;
}
// classe Flottant implémentant l'Interface Affichable
class Entier implements Affichable{
    private int valeur ;
    public Entier (int n){
        valeur = n ;
    }
    public void affiche(){
        System.out.println ("Je suis un entier de valeur " + valeur) ;
    }
}
// classe Flottant implémentant l'Interface Affichable
class Flottant implements Affichable{
    private float valeur ;
    public Flottant (float x){
        valeur = x ;
    }
    public void affiche(){
        System.out.println ("Je suis un flottant de valeur " + valeur) ;
    }
}
```


Exemple complet(3/3)

```java
// classe contenant le main
public class TestInterface{
    public static void main (String[] args){
        int i ;
        Affichable [] tab ;
        tab = new Affichable [3] ;
        tab [0] = new Entier (25) ;
        tab [1] = new Flottant (1.25f) ;
        tab [2] = new Entier (42) ;

        for (i=0 ; i<3 ; i++)
            tab[i].affiche() ;
    }
}
```

Intérêt des interfaces

- Donner une alternative à l'héritage multiple: une classe peut implémenter plusieurs interfaces.
- Faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage (l'interface joue le rôle de la classe mère, avec upcasting et downcasting).
- Favoriser la réutilisation: si le type d'un paramètre d'une méthode est une interface, cette méthode peut s'appliquer à toutes les classes qui implémentent l'interface, et pas seulement à toutes les sous-classes d'une certaine classe.

Classe abstraite, sous-classe et interface : faire le bon choix

- Créer une classe qui n'étend rien quand elle ne réussit pas le test du **EST-UN** pour tout autre type.
- Créer une sous-classe uniquement si vous voulez obtenir une version plus spécifique d'une classe, redéfinir ou ajouter des comportements.
- Utiliser une classe abstraite quand vous voulez définir un patron pour un groupe de sous-classes et qu'une partie du code est utilisée par toutes les sous-classes.
- Utiliser une interface pour définir un rôle que d'autres classes puissent jouer, indépendamment de leur place dans la structure d'héritage.

Exercice: Les exemples suivants sont-ils corrects ?

Exemple1

```
interface Vehicule {
   public void accelerer ();
   public void freiner ();
   public boolean hasEssence ();
   public int getNombrePassagers ();
}
class Voiture {
   private int essence;
   private int passagers;
   public boolean hasEssence(){ return (essence > 0); }
   public int getNombrePassagers() { return passagers; }
}
class VoitureQuiBouge extends Voiture implements Vehicule {
   public void accelerer () {
      System.out.println("Vrooom");
   }
   public void freiner () {
      System.out.println("Skreeee");
   }
}
```

A l'exemple 1, on ajoute

Exemple2

```
interface Habitation {
   public boolean isMeublee();
   public int getNombrePieces ();
}
class CampingCar extends VoitureQuiBouge implements Habitation{
   public boolean isMeublee() {
      return true;
   }
   public int getNombrePieces () {
      return 1;
   }
   public static void main (String [] args){
      Vehicule v = new CampingCar();
      v.accelerer();
   }
}
```

Exercice 8

1. Etablissez une interface 'Car' représentant une voiture du monde réel, prévoyez des méthodes pour changer et accéder à la couleur, au nombre de roues, à la vitesse courante du boîtier de vitesse ("gear"), à la vitesse courante de déplacement ("speed").

2. Ecrivez ensuite la classe 'BasicCar' implémentant cette interface : vous devez définir les attributs, les initialiser avec des valeurs par défaut et écrire le code des méthodes de l'interface.

Exercice 8 (suite)

3. Ecrivez ensuite la classe 'BasicTruck' (camion), sous-classe de la classe 'BasicCar', rajoutant une caractéristique par rapport à une voiture : on est capable de spécifier son type de chargement sous la forme d'une chaîne de caractères et d'accéder à ce type (ex. vide, cailloux, beton, ...).

4. Ecrivez une classe contenant une méthode main pour tester.

Classes Imbriquées

Classes imbriquées

- Java vous permet de définir une classe dans une autre classe. Une telle classe est appelée une classe imbriquée (nested class).

```
public class OuterClass {
    ...
   public class ClasseImbriquee {...}
}
```

- En tant que membre de OuterClass, une classe imbriquée peut être déclarée private, public, protected ou de visibilité du package.
- Il est possible d'imbriquer plusieurs classes en Java. Mais la difficulté réside dans leur utilisation, c-à-d l'accés au membre de la classe imbriquée dans les autres classes.

Classes imbriquées

- Les classes imbriquées sont divisées en **quatre catégories** :
  - **Classes internes (inner classes)**: membres d'une classe englobante et ont accès aux autres membres de cette dernière, même s'ils sont déclarées privées.
  - **Classes locales**: classes définies dans un bloc de code. Elles peuvent être statiques ou non.
  - **Classes anonymes**: classes locales sans nom qui sont définies et instanciées à la volée.
  - **Classes internes statiques**: membres statiques d'une autre classe englobante et n'ont accès qu'aux membres statiques de cette dernière.

Pourquoi utiliser des classes imbriquées ?

- C'est une façon de **regrouper logiquement des classes qui ne sont utilisées qu'à un seul endroit** : si une classe n'est utile qu'à une seule autre classe, alors il est logique de l'embarquer dans cette classe et de garder les deux ensemble.
- **Cela augmente l'encapsulation**: Considérez deux classes de niveau supérieur, A et B, où B a besoin d'accéder aux membres de A qui seraient autrement déclarés private. En cachant la classe B dans la classe A, les membres de A peuvent être déclarés privés et B peut y accéder. De plus, B lui-même peut être caché du monde extérieur.
- Cela peut conduire à **un code plus lisible et maintenable**: l'imbrication de petites classes dans des classes de niveau supérieur rapproche le code de l'endroit où il est utilisé.

Classes internes

- Comme pour les méthodes et les variables d'instance, une classe interne **est associée à une instance de sa classe englobante** et a un accès direct aux méthodes et aux champs de cet objet.
- Peut **accéder aux autres membres de la classe englobante**, même ceux déclarées privées.
- Comme une classe interne est associée à une instance, elle **ne peut pas elle-même définir de membres statiques**.
- Pour instancier une classe interne, vous devez d'abord instancier la classe externe. Ensuite, créez l'objet interne dans l'objet externe.

```
OuterClass oc = new OuterClass();
OuterClass.InnerClass ic1 = oc.new InnerClass();
OuterClass ic2 = new OuterClass().new InnerClass();
```


Classes internes - Exemple

```
public class OuterClass {

    private String outerField = "Outer field";
    private static String staticOuterField = "Static outer field";

    public class InnerClass {
        void accessMembers() {
            System.out.println(outerField);
            System.out.println(staticOuterField);
        }
    }

    public static void main(String[] args) {
        System.out.println("Inner class:");
        System.out.println("------------");
        OuterClass oc = new OuterClass();
        OuterClass.InnerClass ic = oc.new InnerClass();
        ic.accessMembers();
    }
}
```

This

- Le mot clé **this** fait toujours référence **à l'instance en cours**.
- L'utilisation du mot clé this dans une classe interne fait donc référence à l'instance courante de cette classe interne.
- Si la **classe englobante et la classe interne possèdent toutes les deux un membre de même nom**. Dans ce cas, il faut utiliser la version qualifiée du mot clé this pour accéder au membre de la classe englobante.
- Lors de l'instanciation d'une classe interne, si aucune instance de la classe englobante n'est utilisée, c'est l'instance courante qui est utilisée (mot clé this).

This - exemple

```
public class OuterClass2 {
    private int var = 3;
    InnerClass2 i = this.new InnerClass2();

    public class InnerClass2 {
        private int var = 2;
        public void accessMembers(int var) {
            System.out.println("var:" + var);
            System.out.println("this.var:" + this.var);
            System.out.println("OuterClass2.this.var:" + OuterClass2.this.var);
        }
    }

    public static void main(String[] args) {
        OuterClass2 oc = new OuterClass2();
        OuterClass2.InnerClass2 ic = oc.new InnerClass2();
        ic.accessMembers(1);
    }
}
```

Héritage d'une classe interne

- Une classe peut hériter d'une classe interne. Dans ce cas, il faut
  1. fournir aux constructeurs de la classe une référence sur la classe principale de la classe mère.
  2. appeler explicitement dans le constructeur le constructeur de cette classe principale avec une notation particulière du mot clé super.

```
public class OuterClass {
  public class InnerClass { ... }
  public class ClasseFille extends OuterClass.InnerClass {
    public ClasseFille(OuterClass oc) {
      oc.super();
    }
  }
}
```

- Une classe interne peut être déclarée avec les modificateurs **final** et **abstract**.

Classes locales

- Ces classes internes locales (**local inner-classes**) sont définies à l'intérieur d'une méthode ou d'un bloc de code.
  - **ne peuvent pas utiliser les modificateurs public, private, protected.**
  - **ne sont utilisables que dans le bloc de code** où elles sont définies.
  - ont toujours accès aux **membres de la classe englobante**.
  - ont aussi accès à certaines **variables locales** du bloc où est elles sont définie:
    - Ces variables doivent être définies dans la méthode avec le mot clé **final**.
    - Ces variables doivent être initialisées avant leur utilisation par la classe interne.
    - Elles sont utilisables n'importe où dans le code de la classe interne.

Classes locales - exemple

```
public class OuterClass3 {
    private int varInstance = 1;
    public void myMethod() {
        final int varLocale = 2;
        class LocalClass {
            public void affiche(final int varParam) {
                System.out.println("varInstance = " + varInstance);
                System.out.println("varLocale = " + varLocale);
                System.out.println("varParam = " + varParam);
            }
        }
        LocalClass lc = new LocalClass();
        lc.affiche(3);
    }

    public static void main(String args[]) {
        OuterClass3 oc = new OuterClass3();
        oc.myMethod();
    }
}
```

Classes anonymes

- Les classes internes anonymes (**anonymous inner-classes**) sont des classes internes qui ne possèdent pas de nom.
  - ne peuvent donc être **instanciées qu'à l'endroit où elles sont définies**.
  - **ne peuvent pas avoir de constructeur** puisqu'elle ne possèdent pas de nom mais elle peuvent avoir des initialisateurs.
  - peuvent **soit hériter d'une classe soit implémenter une interface** mais elle ne peuvent pas explicitement faire les deux.

```
btnAnnuler.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
```

Classes interne statiques

- Les classes internes statiques (**static member inner-classes**) sont des classes internes qui ne possèdent pas de référence vers leur classe englobante.
  - ne **peuvent pas accéder directement aux membres d'instance** de leur classe englobante: ne peuvent les utiliser que via une référence d'objet.
  - **peuvent accéder aux variables statiques** de la classe englobante.
  - leur **utilisation est obligatoire si la classe est utilisée dans une méthode statique**.
- On instancie une classe interne statique de la même manière qu'une classe de niveau supérieur :

```
StaticInneClass sic = new StaticInnerClass();
```


Classes interne statiques - Exemple

```
public class OuterClass4 {
    private String outerField = "Outer field";
    private static String staticOuterField = "Static outer field";

    public static class StaticInnerClass {
        public void accessMembers(OuterClass4 outer) {
            System.out.println(outer.outerField);
            System.out.println(staticOuterField);
        }
    }

    public static void main(String[] args) {
        OuterClass4 oc = new OuterClass4();
        StaticInnerClass sic = new StaticInnerClass();
        sic.accessMembers(oc);
    }

}
```

Exercice 9

- Réécrire la classe **Cercle** en utilisant une classe interne **Point** qui représente un point dans un espace à deux dimensions, avec des coordonnées x et y. La classe Cercle a deux variables d'instance, rayon et centre et possède entre autres:
  - un constructeur qui prend un rayon et les coordonnées x et y et crée un nouvel objet Point pour représenter le centre.
  - des méthodes getter et setter
  - une méthode toString
  - des méthodes pour calculer l'aire et la circonférence du cercle
  - une méthode main pour tester