

Cours  
**Programmation Orientée Objet 2**  
Pour  
**ING 2**

**Chap 00:**

**Rappels sur l'orienté objet**



MEKAHLIA Fatma Zohra LAKRID  
Maître de Conférences Classe B

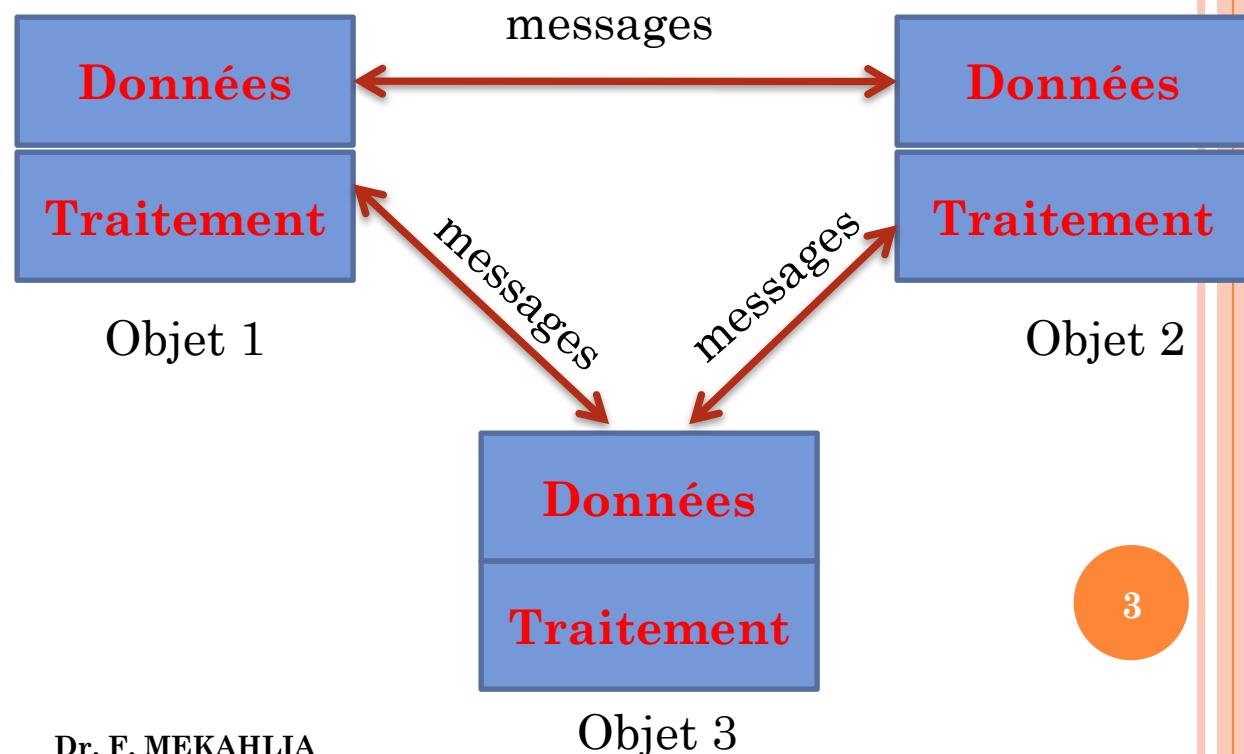
Laboratoire de Modélisation, Vérification et Evaluation des Performances des systèmes  
complexes (MOVEP)  
Bureau 123

# PLAN

- Introduction
- Object et classe
- Encapsulation
- Mot clé final
- Mot clé null
- Exercice
- Héritage
- Exercice
- Polymorphisme
- Exercice
- Entrées clavier
- Classe abstraite
- Exercices
- Classe interface
- Exercices
- Classe finale
- UML

# INTRODUCTION

- Un programme. O. O est constitué d'un ensemble d'objets chacun disposant d'une partie **fonction** (traitement) et d'une partie **données**.
- Les objets interagissent entre eux par **l'envoie des messages**.



# INTRODUCTION

- Un **objet** est une **variable** de type complexe appelé **Classe**.
- Une **classe** est la définition d'un **type**, alors que l'objet est une **déclaration** de variables.
- Après avoir créé une classe, on peut créer autant d'objets que l'on veut de type **classe**.
- Un **objet** est une instance de la classe càd une **valeur particulière de la classe**.

# OBJET

- Un objet est une entité :
  - Ayant une **identité**: valeur unique et invariante qui caractérise l'objet.
  - Ayant des **attributs** : capable de sauvegarder un **état** de l'objet c'est-à-dire l'ensemble des valeurs des attributs de cet objet.
  - Ayant des **méthodes**: répondant à des messages précis en déclenchant des activations internes appropriés qui **changent l'état de l'objet**. Ce **sont les traitements** que l'objet réalise.

# CLASSE

- Un objet est une **instance d'une classe**.
- Une classe consiste à créer **un nouveau type**.  
Une fois créée, la classe devient un type et des objets peuvent être associés à ce type.
- **Une classe** est la représentation de la structure d'une **famille d'objets** partageant des **propriétés** et **méthodes communes**.
- Elle permet la déclaration des attributs (propriétés) ainsi que la définition de l'ensemble de méthodes.

# CLASSE

La définition d'une classe consiste à lui attribuer :

1. Un nom,
2. des **attributs**,
3. des **méthodes**,
4. des **constructeurs**, qui permettent de créer des objets ;

# CLASSE

- Une classe est un ensemble d'objets qui ont en commun :
  - les mêmes méthodes.
  - les mêmes types d'attributs.
- Classe = attributs + méthodes
- Objet = état (attributs) + comportement (méthodes)

66

8

# CLASSE

- Définition d'une classe en java

```
class Rectangle {  
    // définition des Attributs  
    // définition des Méthodes  
}
```

- Conventions de nommage:

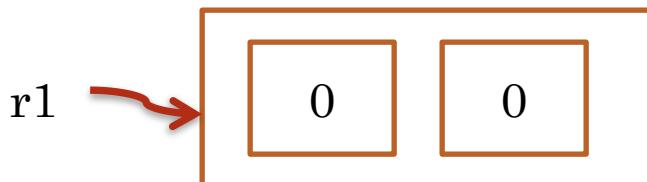
1. Le fichier .java doit avoir le même nom que la classe publique qu'il décrit,
2. le nom de la classe doit débuter par une MajusculeAnsiQueChaquePremiereLettreDeChaqueMot,
3. un fichier .java par classe, même pour celle contenant le main() .

# INSTANCES D'UNE CLASSE / CRÉATION D'OBJETS

- Pour créer une instance (ou objet) d'une classe, on utilise l'opérateur **new** suivi d'un constructeur de la classe. Bien évidemment, l'objet à créer doit être préalablement déclaré avec le type de la classe adéquate.
- **Rectangle r1 ; // déclaration de l'objet r1**



- **r1 = new Rectangle(); // création de l'objet r1**



# ACCÈS AUX MEMBRES D'UN OBJET

- L'accès à un attribut (**variable d'instance**) ou méthode (**méthode d'instance**) d'un objet donné se fait à l'aide de la notation à point.
- *NomDeObjet.NomAttribut*  
*ou*
- *NomDeObjet.NomMethode()*

..

# SURCHARGE DE MÉTHODE

- La surcharge de méthode est un concept qui permet à une classe d'avoir plusieurs méthodes portant le **même nom** et le même **type de retour**, si leurs listes d'arguments sont différentes.

# SURCHARGE DE MÉTHODE

- **Nombre de paramètres:** cas valide de surcharge.
  - ❖ somme(int, int) ;
  - ❖ somme(int, int, int);
- **Type de données des paramètres:** cas valide de surcharge.
  - ❖ somme(int, int);
  - ❖ somme(int, double);
- **Cas non valide de surcharge de méthode:**
  - ❖ **int** somme(int, int);
  - ❖ **double** somme(int, int);

66

# ENCAPSULATION

- L'**encapsulation** consiste à cacher l'état interne d'un objet et d'imposer de passer par des méthodes permettant un accès sécurisé à l'état de l'objet.
- Pour cette raison, la déclaration d'une classe, d'une méthode ou d'un attribut peut être **précédée par un modificateur d'accès** (visibilité).

# MODIFICATEURS DE VISIBILITÉ ET ACCÈS

**public** : toutes les classes peuvent accéder à l'item. La déclaration de variables publiques est contraire au principe d'encapsulation.

- **protected** : seules les classes dérivées et les classes du même package peuvent accéder à l'item (attribut ou méthode) .
- **Private**: un item (attribut ou méthode) private (privé) est accessible uniquement au sein de la classe dans laquelle il est déclaré. Ces éléments ne peuvent être manipulés qu'à l'aide de méthode spécifiques appelés accesseur et mutateur
- **(par défaut)** : sans modificateur d'accès, seules les classes du même package peuvent accéder à l'item

# MODIFICATEURS DE VISIBILITÉ ET ACCÈS

Autres modificateurs :

- **static:** indique, **pour une méthode**, qu'elle peut être appelée sans instancier sa classe i.e. indépendamment de tout objet (méthode de la classe). **Pour un attribut**, qu'il s'agit d'un attribut de classe, et que sa valeur est partagée entre les différentes instances de sa classe (variable de classe).
- **final:** une variable déclarée final est en fait une constante, il n'est plus possible de la modifier. Les méthodes déclarées final ne peuvent pas être remplacée dans une sous classe. Les classes déclarées final ne peuvent pas avoir de sous-classe.

# LE MOT CLÉ FINAL

- Une variable qualifiée de **final** signifie que la variable est **constante**. Une variable déclarée final ne peut plus voir sa valeur modifiée.
- Une **méthode** final ne peut pas être redéfinie dans une sous classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous classée.
- Lorsque le modificateur final est ajouté à une **classe**, il est interdit de créer une classe qui en hérite.

..

# LE MOT CLÉ NULL

- Le mot clé **null** est utilisable partout. Il peut être utilisé à la place d'un objet de n'importe quelle classe ou comme paramètre.
- Il permet de représenter la référence qui ne référence rien.
- Le fait d'initialiser une variable référent un objet à null permet au ramasseur de miettes (garbage collector) de libérer la mémoire allouée.

..

# EXERCICE

- Définir une classe Rectangle ayant les attributs privées suivants : longueur et largeur et qui ce trouve dans le dossier coursPOO.
- Ajouter deux constructeur d'initialisation: Rectangle(double long, double largeur) et le constructeur d'initialisation à null.
- Ajouter les méthodes suivantes :
- isCarre ( ) : vérifie si le rectangle est un carré.

```
package coursPOO;
```

```
public class Rectangle {  
    private double longueur;  
    private double largeur;
```

```
        public Rectangle(double long, double largeur)  
    {  
        longueur = long;  
        this.largeur = largeur;  
    }
```

```
    public Rectangle() { }
```

```
o public double getLongueur() {  
o         return longueur;  
o     }  
  
o  
o public void setLongueur(double longueur) {  
o         this.longueur = longueur;  
o     }  
  
o  
o public double getLargeur() {  
o         return largeur;  
o     }  
  
o  
o public void setLargeur(double largeur) {  
o         this.largeur = largeur;  
o     }
```

- o public boolean isCarre() {
- o           if ( (longueur == largeur) &&  
(longueur <> 0))
- o           return true;
- o       else
- o           return false;
- o }

# HÉRITAGE

- L'héritage est le troisième paradigme de la programmation orientée objet ( le 1er étant la structure de classe, le 2eme l'encapsulation).
- Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir **qu'une seule classe mère** : il n'y a pas d'héritage multiple en java.

..

# HÉRITAGE

- Grâce à l'héritage, les **objets d'une classe fille ont accès** aux **données** et aux **méthodes** de la **classe mère** et peuvent les étendre.
- Les sous classes **peuvent redéfinir** les variables et les méthodes héritées.
- Pour les **variables**, il suffit de **les redéclarer sous le même nom avec un type différent**.
- Les **méthodes** sont redéfinies avec le même nom, le mêmes type de retour et le même nombre d'arguments, sinon il s'agit d'une surdéfinition.

»

# HÉRITAGE(ACCÈS AUX DONNÉES )

1. Une méthode d'une classe dérivée n'accède pas aux membres **private** de sa classe de base.
2. Une méthode d'une classe dérivée accède aux membres **protected** de sa classe de base.

..

# HÉRITAGE (REDÉFINITION)

- On redéfinit une méthode quand une nouvelle méthode de la classe fille **a la même signature** qu'une méthode héritée de la classe mère.
- Consiste à **substituer le corps** d'une méthode par un autre. même nom, même nombre et type d'argument, même type de retour et ne doit pas diminuer les droits d'accès à une méthode.  
**Exemple:** la méthode afficher() est public dans la classe mère ne doit pas devenir private dans la classe fille).

..

# HÉRITAGE (REDÉFINITION)

- o class A
- o { .....
- o     double surface () {return (0) ;}
- o     .....
- o }
- o Class B extends A
- o { ....
- o     double surface () { return (rayon \* 2\* 3.14) ;}
- o ...
- o }

# HÉRITAGE (SURDÉFINITION )

- La surdéfinition cumule plusieurs méthodes de même nom avec des arguments (nombre ou type) différents.
- Si une méthode a le même nom qu'une méthode d'une classe ascendante avec des arguments (nombre ou type) différents, on est dans le cas d'une surdéfinition.
- La nouvelle méthode est utilisable par la classe dérivée et ses descendantes.

..

# HÉRITAGE (SURDÉFINITION )

- **class A**
- {public void f (int n) {...}
- .....
- }
- **Class B extends A**
- { public void f (float x) {...}
- ...
- }

# HÉRITAGE

- **Object** est la classe mère de toutes les classes en java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritage successif toutes les classes héritent d'Object.
- La classe **Object** comporte quelques méthodes qu'on peut soit utiliser telles quelles soit les redéfinir.
  - **toString ()** : fournit une chaîne contenant le nom de la classe à laquelle appartient l'objet ainsi que l'adresse de l'objet en hexadecimal.

## Exemple:

- Rectangle r = new Rectangle (3,5);
- System.out.println(" r = " + r.toString() );
- Affiche: r = Rectangle @fc17kjlf

## SUITE DE L'EXERCICE

- `toString ()` : affiche **C'est un carré** ou **Ce n'est pas un carré**.
- Dans la méthode **main** de la classe **TestRectangle**, créer un tableau de type **Rectangle** et qui contient:
  - `rectangle1` avec `long =5` et `larg=8`
  - `rectangle2` avec `long =3.5` et `larg=9.1`
  - `rectangle3` avec `long =5` et `larg=5`
  - `rectangle4` avec `long =0` et `larg=0`
- En suite, le programme affiche pour chaque rectangle s'il s'agit d'un un carré ou non.

# SUITE DE L'EXERCICE

```
o public String toString() {  
o         String etat = null;  
o         if (this.isCarre())  
o             etat = "C'est un carré";  
o         else  
o             etat = "Ce n'est pas un carré";  
o         return etat;  
o     }
```

..

# SUITE DE L'EXERCICE

```
package coursPOO;
```

```
o  public class TestRectangle {  
o  
o      public static void main(String[] args) {  
o          Rectangle [] rectangles = new Rectangle[4];  
o          rectangles [0] = new Rectangle(5, 8);  
o          rectangles [1] = new Rectangle(3.5, 9.1);  
o          rectangles [2] = new Rectangle(5, 5);  
o          rectangles [3] = new Rectangle();  
o  
o          for(Rectangle e: rectangles)  
o              System.out.println(e.toString());  
o      }  
o  }
```

# HÉRITAGE (APPEL DE CONSTRUCTEUR )

La première instruction d'un constructeur peut être un appel: à un constructeur de la classe mère par **super(...)** **ou** à un autre constructeur de la classe fille par **this(...)**.

Appeler le constructeur de la classe mère garantit que l'on peut initialiser les arguments de la classe mère.

Si l'on n'indique pas **super()**, il y a un appel du constructeur par défaut de la classe mère.

Si la première instruction d'un constructeur n'est ni **super(...)**, ni **this(...)**, le compilateur ajoute un appel implicite au constructeur sans paramètre de la classe mère (erreur s'il n'existe pas)

# EXERCICE

- Soit la classe RectangleColor qui hérite de la classe Rectangle déjà vue.

```
class RectangleColor extends Rectangle {  
    private String color;  
  
    public Rectangle(double long, double largeur, String  
        color) {  
        super (long, largeur); // Obligatoirement  
        this. color= color;  
    }  
    public Rectangle(){  
        this (5,6,bleu); // appel du constructeur de la classe même}  
    }  
}
```

# POLYMORPHISME

## ○ PolyMorphisme ?

- Poly: plusieurs.
- Morphisme: forme.

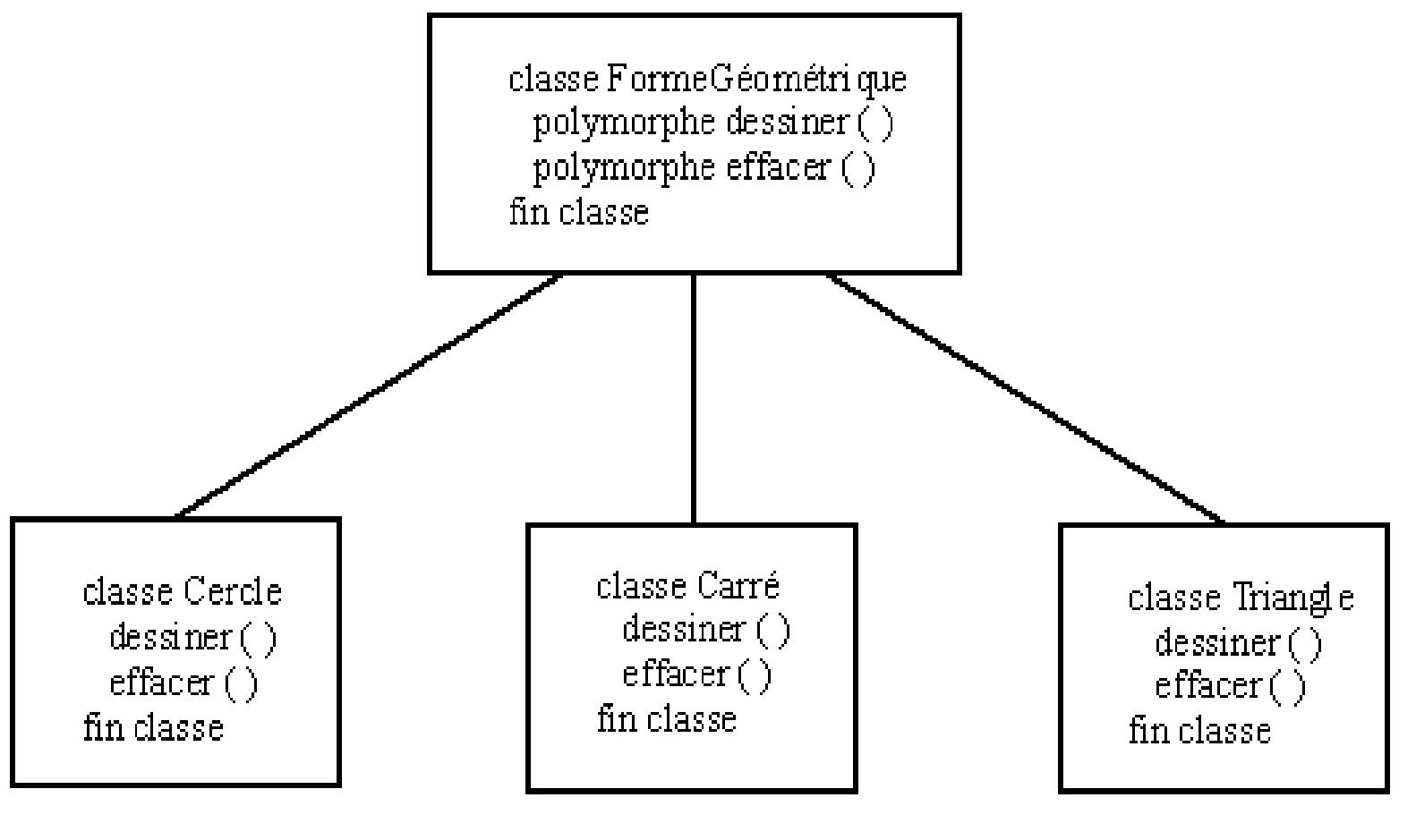
66

# POLYMORPHISME

- La première catégorie de polymorphisme, que vous avez déjà vu c'est **la surcharge du constructeur**
- vous connaissez également: le **polymorphisme par héritage de méthode**, lié à la redéfinition: Il explique comment une méthode peut se comporter suivant l'objet **sur lequel elle s'applique**, ie, quand une même méthode est définie à la fois dans la classe **mère** et dans la classe **fille**, son exécution est réalisée **en fonction de l'objet associé à l'appel.**

# POLYMORPHISME

..



# POLYMORPHISME

- **Exemple:** les méthodes *dessiner()* et *effacer()* de la classe géométrique sont polymorphes. Leur nom est similaire dans les trois classes dérivées (Cercle, Carré et Triangle), mais dessiner un cercle est différent de dessiner un carré ou un triangle.
- Ainsi le **polymorphisme** se résume par : un même nom et plusieurs implémentations.

# POLYMORPHISME

- Nous avons une autre catégorie de polymorphisme, appelé **polymorphisme d'objets**.
- C'est un concept très puissant en P.O.O, qui complète l'**héritage**. Il se base sur le concept d'héritage et la redéfinition des méthodes.
- Le polymorphisme d'objet se base sur cette affirmation : un objet a comme type non seulement sa classe mais aussi n'importe quelle classe dérivée

..

# EXERCICE

- Créer un tableau d'objets, les uns étant de type **Point** et les autres de type **PointCol** et appeler la méthode afficher().
- Chaque objet réagira selon son type.
- Les objets points colorés se sont aussi des points et peuvent donc être traités comme des points,

..

# EXERCICE

Soient les classes Point et PointCol suivantes:

```
class Point {  
    private double x;  
    private double y;  
    Public Point (double x, double y)  
    {  
        this.x =  x  ;  
        this.y =  y  ;  
    }  
    public void afficher()  
    {  
        System.out.println("je suis un point de  
        coordonnées:"+x+" "+y);  
    }  
}
```

```
class PointCol extends Point {  
    private String couleur;  
  
    Public PointCol (double x, double y,  
        String couleur) {  
        super(x,y);  
        this.couleur = couleur;  
    }  
    public void afficher() {  
        super.afficher();  
        System.out.println("couleur:" + couleur);  
    }  
}
```

# EXERCICE

```
class TestPolymorphisme {  
    Public static main (String args[])  
    {  
        Point p;  
        p = new Point(3,5);  
        p.afficher();  
        p = new PointCol (4,9,"Vert");  
        p.afficher();  
    }  
}
```

appelle la méthode afficher de la classe Point

***je suis un point de coordonnées: 3 5***

appelle la méthode afficher de la classe PointCol

***je suis un point de coordonnées: 4 9  
vert***

L'instruction p. afficher() se base non pas sur le type de la variable p mais sur le type effectif de l'objet référencé par p au moment de l'appel car celui-ci peut évoluer dans le temps.

**Règle : Le choix de la méthode appelée se fait selon le type effectif de l'objet référencé au moment de l'exécution**

**Ce choix se fait au moment de l'exécution et non pas au moment de la compilation.**

# EXERCICE

- La méthode **afficher()** est décrite dans la classe Point et dans la classe PointCol.
- Les deux méthodes afficher() sont définies sans aucun paramètre.
- Le choix de la méthode ne peut donc s'effectuer sur la différence de paramètres. Il est effectuée par rapport à l'objet sur lequel la méthode est appliquée

..

# EXERCICE

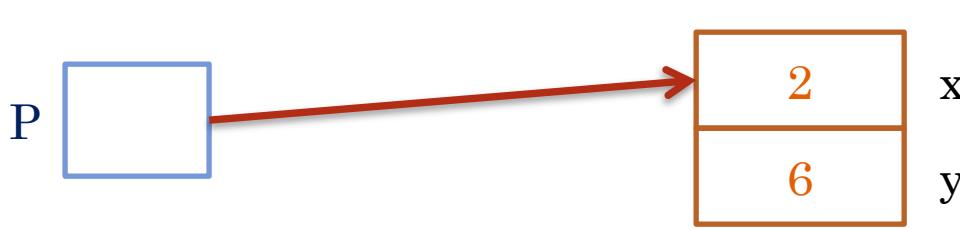
- Exemple:

/\*\*\*\* même référence \*\*\*/

Point p;

p = new Point(2,6);

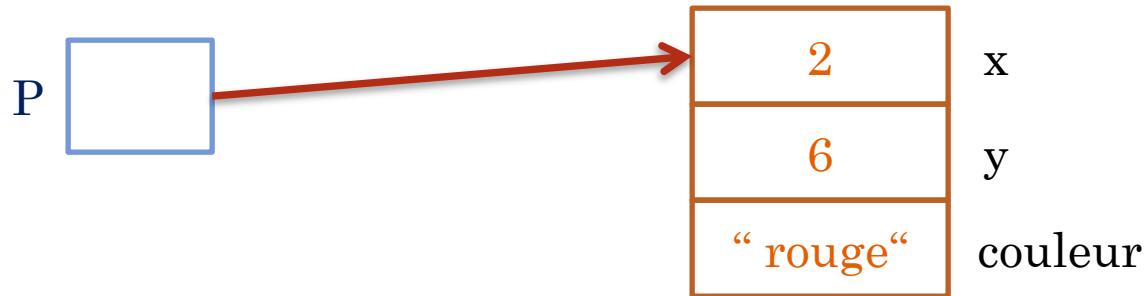
- On aboutit tout naturellement à cette situation



# EXERCICE

*/\*\* change de référence \*\*/*

- Point p;
- p = new PointCol (2,6,"rouge");



**NB:** Malgré que **p** est de type **Point**, Java autorise cette affectation !!

**Règle :** Java permet à une variable objet l'affectation d'une référence à objet d'un type dérivé.

# POLYMORPHISME ET TABLEAUX

- Le polymorphisme peut s'appliquer à des tableaux d'objets.

```
1 package polymorphisme;
2
3 public class Personne {
4
5     public void parler () {
6         System.out.println("cette personne parle");
7     }
8 }
```

..

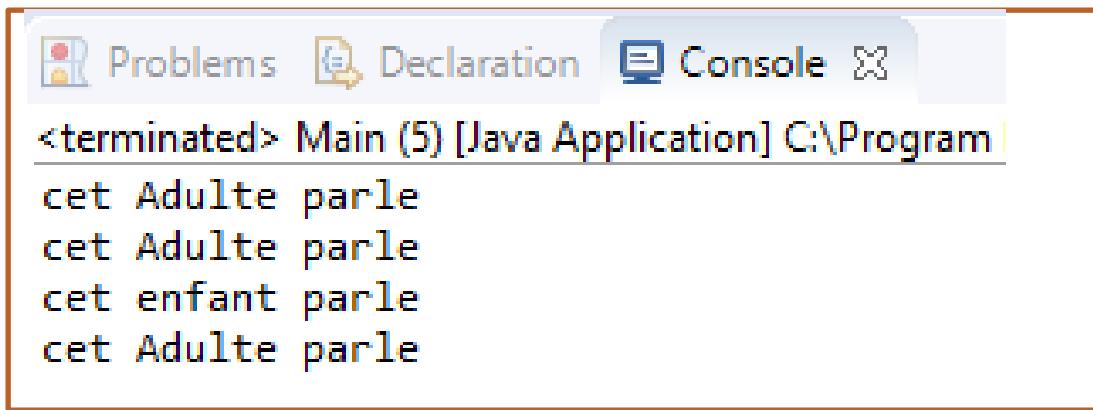
# POLYMERISME ET TABLEAUX

```
1 package polymorphisme;
2
3 public class Adulte extends Personne{
4
5     @Override
6     public void parler() {
7         System.out.println("cet Adulte parle");
8     }
9 }
```

```
1 package polymorphisme;
2
3 public class Enfant extends Personne{
4
5     @Override
6     public void parler() {
7         System.out.println("cet enfant parle");
8     }
9 }
```

```
1 package polymorphisme;
2
3 import java.util.ArrayList;
4
5 public class Main {
6     public static void main(String[] args) {
7         ArrayList <Personne> P = new ArrayList<> ();
8         P.add(new Adulte());
9         P.add(new Adulte());
10        P.add(new Enfant());
11        P.add(new Adulte());
12
13        for(Personne personne:P) {
14            personne.parler();
15        }
16    }
17 }
```

# POLYMORPHISME ET TABLEAUX



The screenshot shows a Java application running in an IDE. The tabs at the top are 'Problems', 'Declaration', 'Console', and 'Console' (highlighted). The console output window displays the following text:

```
<terminated> Main (5) [Java Application] C:\Program  
cet Adulte parle  
cet Adulte parle  
cet enfant parle  
cet Adulte parle
```

# LIRE LES ENTRÉES CLAVIER

- Pour que Java puisse lire ce que vous tapez au clavier, vous allez utiliser un objet de type **Scanner**.
- Lorsque vous faites **System.out.println()**, je vous rappelle que vous appliquez la méthode `println()` sur la sortie standard ; or ici, nous allons utiliser l'entrée standard **System.in**.
- Donc, avant de dire à Java de lire ce que nous allons taper au clavier, nous devrons instancier un objet Scanner.
- **Scanner sc = new Scanner (System.in);**

..

# LIRE LES ENTRÉES CLAVIER

66

```
Scanner sc = new Scanner(System.in);  
System.out.print ← Import 'Scanner' (java.util)  
                  Create class 'Scanner'
```

# LIRE LES ENTRÉES CLAVIER

- l'instruction **nextLine()** renvoie une chaîne de caractères.

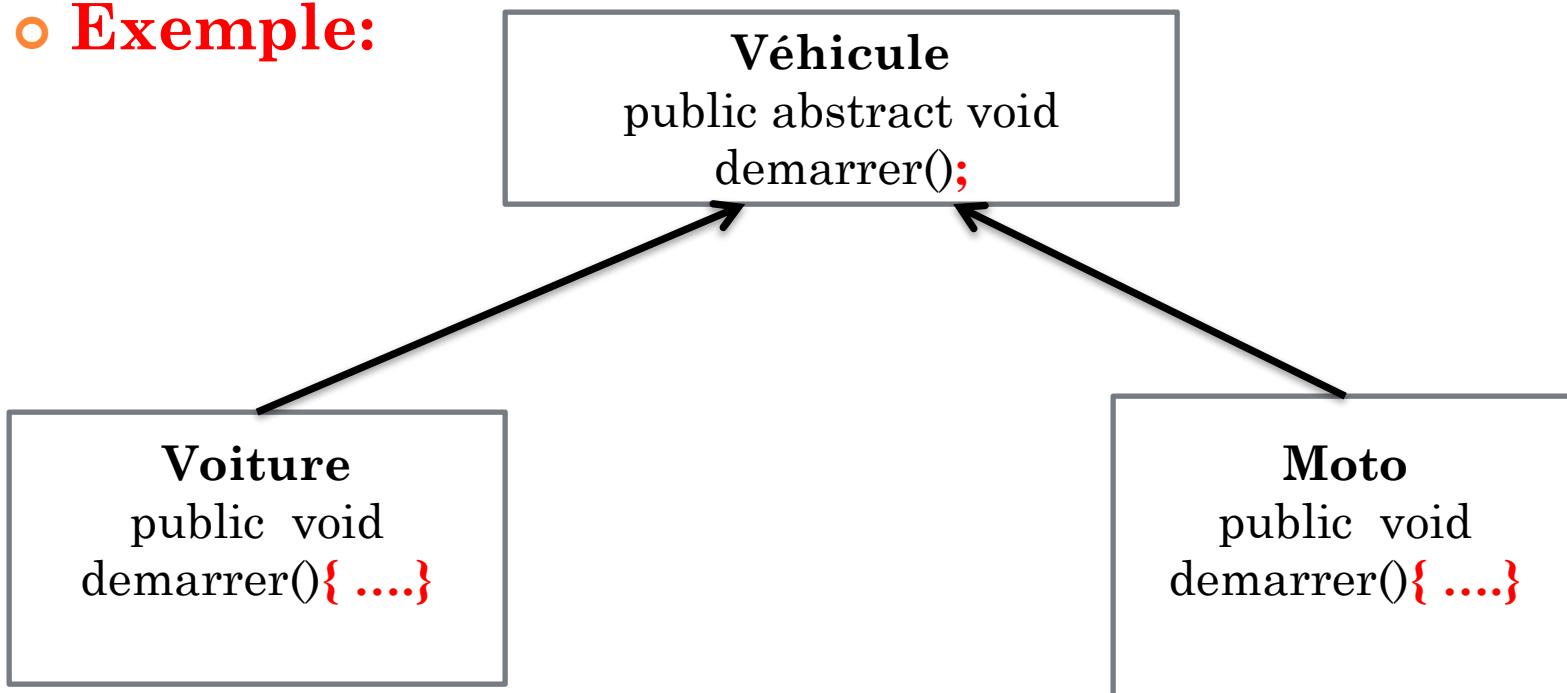
```
1 import java.util.Scanner;
2 public class Sdz {
3
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6         System.out.println("Veuillez saisir un mot :");
7         String str = sc.nextLine();
8         System.out.println("Vous avez saisi : " + str);
9     }
}
```

```
1     Scanner sc = new Scanner(System.in);
2     int i = sc.nextInt();
3     double d = sc.nextDouble();
4     long l = sc.nextLong();
5     byte b = sc.nextByte();
6     //etc
```

# Classe abstraite

# MÉTHODE ABSTRAITE

- Les méthodes abstraites sont principalement déclarées dans la classe de base lorsque deux ou plusieurs sous-classes font également la même méthode de différentes manières à travers des implémentations différentes.
- Exemple:**



# CLASSE ABSTRAITE

- Une classe qui comporte une ou plusieurs méthodes abstraites (des méthodes dont on fournit uniquement la signature ainsi que le type de retour.) est **abstraite**, même si on n'indique pas le mot clé abstract à sa déclaration. Et de ce fait, elle ne sera **pas instanciable** et elle ne peut servir que de classe de base pour être dérivée.
- Exemple:

```
abstract class Vehicule {  
    // variables et autre méthodes  
    public abstract void demarrer () ;  
}
```

# CLASSE ABSTRAITE

- L'intérêt des classes abstraites est de regrouper plusieurs classes sous un même nom de classe ainsi que de **décrire partiellement des attributs et méthodes communs à plusieurs classes.**
- Une méthode abstraite doit obligatoirement être déclarée publique (puisque elle est destinée à être redéfinie dans une classe dérivée),

..

# CLASSE ABSTRAITE

- Une classe dérivée d'une classe abstract n'est pas obligée de redéfinir toutes les méthodes abstraites de sa classe de base et peut même n'en redéfinir aucune et restera abstraite elle-même,
- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et / ou contenir des méthodes abstraites.

# EXERCICE

- Créer une classe *Vehicule* qui peut retourner son nombre de roues à l'aide d'une méthode. Chaque véhicule possède une marque et un nombre max de vitesse.
- Créer deux classes concrètes (*Voiture* et *Moto*) qui héritent de *Vehicule* et qui doivent, maintenant, fournir une implémentation de la méthode *getNbRoues* pour pouvoir compiler.

»

# EXERCICE

```
public abstract class Vehicule {  
    private String marque;  
    private int vitesse;  
  
    public Vehicule() ;  
    public Vehicule(String marque, int vitesse) {  
        this.marque = marque;  
        this.vitesse = vitesse;}  
  
    public abstract int getNbRoues();  
}
```

# EXERCICE

```
public class Voiture extends Vehicule {  
  
    @Override  
    public int getNbRoues() {  
        return 4; }  
  
}
```

# EXERCICE

```
public class Moto extends Vehicule {
```

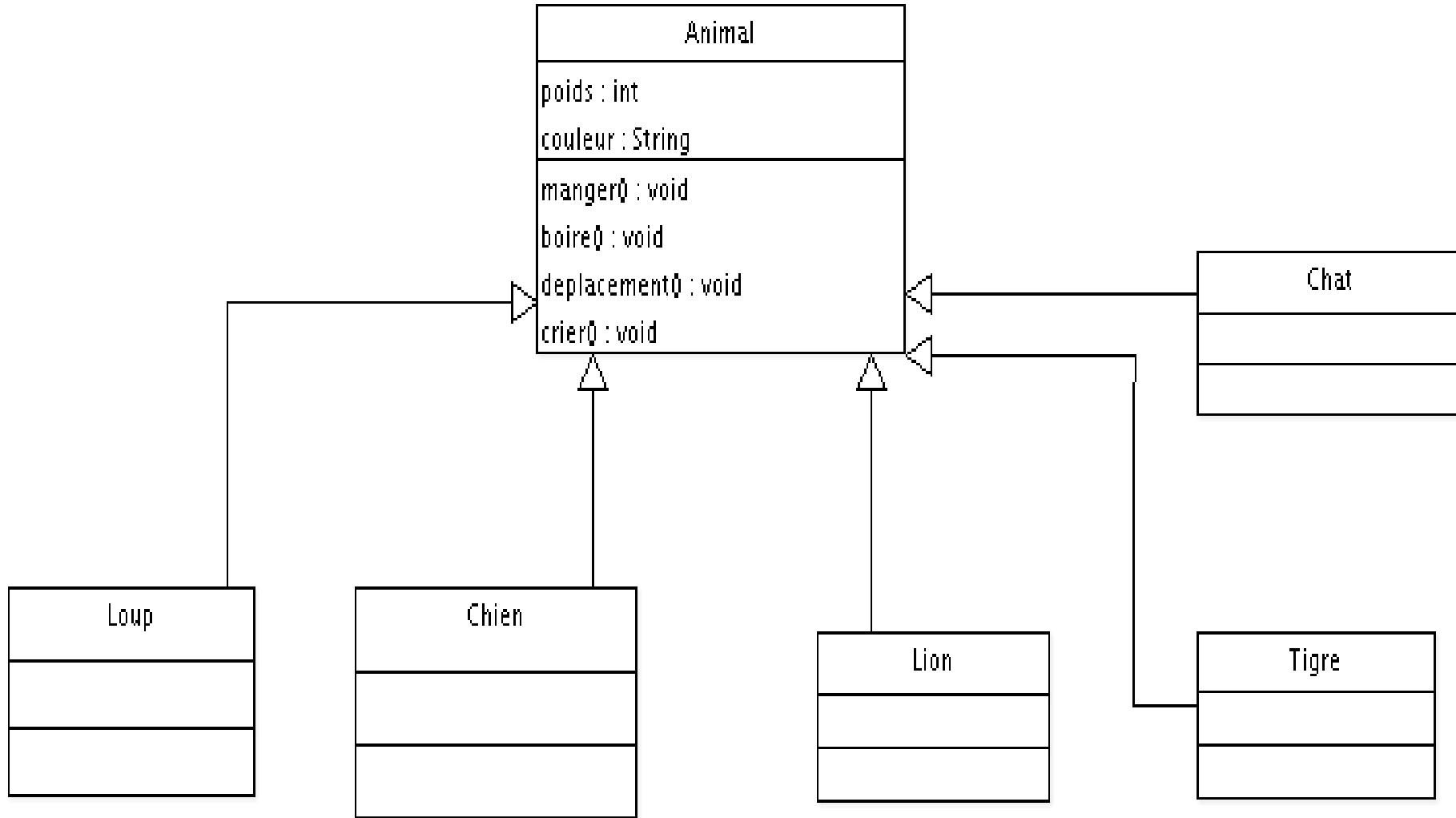
```
    @Override  
    public int getNbRoues() {  
        return 2; }  
    }
```

66

# EXEMPLE D'APPLICATION

- Notre programme intéressé à la gestion de différents types d'animaux tel que: **des loups, des chiens, des chats, des lions, des tigres.**
- Que pouvons-nous définir de commun à tous ces animaux ? : **une couleur, un poids, qu'ils crient, qu'ils se déplacent, qu'ils mangent, qu'ils boivent.**
- Nous pouvons donc faire une classe mère, appelons-la **Animal !!**
- Voici donc à quoi pourraient ressembler nos classes:

# EXEMPLE D'APPLICATION



Nous avons bien notre classe mère **Animal** et nos animaux qui en héritent.

# EXEMPLE D'APPLICATION

- Il est possible de créer un objet **Animal** ? Quel est son poids, sa couleur, que mange-t-il ?

```
1  public class Test{  
2      public static void main(String[] args){  
3          Animal ani = new Animal();  
4          ani.manger(); //Que doit-il faire ? ?  
5      }  
6  }
```

- On ne sais pas comment mange un objet **Animal**... Donc il faut empêcher la classe Animale d'être instanciable !!
- Pour répondre à ce besoin: la classe Animal doit être une classe **Abstraite**.

```
1  public class Test{  
2      public static void main(String[] args){  
3          Animal ani = new Animal(); ... //Erreur de compilation ! !
```

# EXEMPLE D'APPLICATION

```
1 abstract class Animal{  
2     abstract void manger(); //une méthode abstraite  
3 }
```

- Pour pouvoir utiliser les méthodes abstraites de animal, nos classes enfants seront **OBLIGÉES** de redéfinir ces méthodes !

```
1 public class Test{  
2  
3     public static void main(String args[]){  
4  
5         Animal loup = new Loup();  
6         Animal chien = new Chien();  
7         loup.manger();  
8         chien.crier();  
9  
10    }  
11 }
```

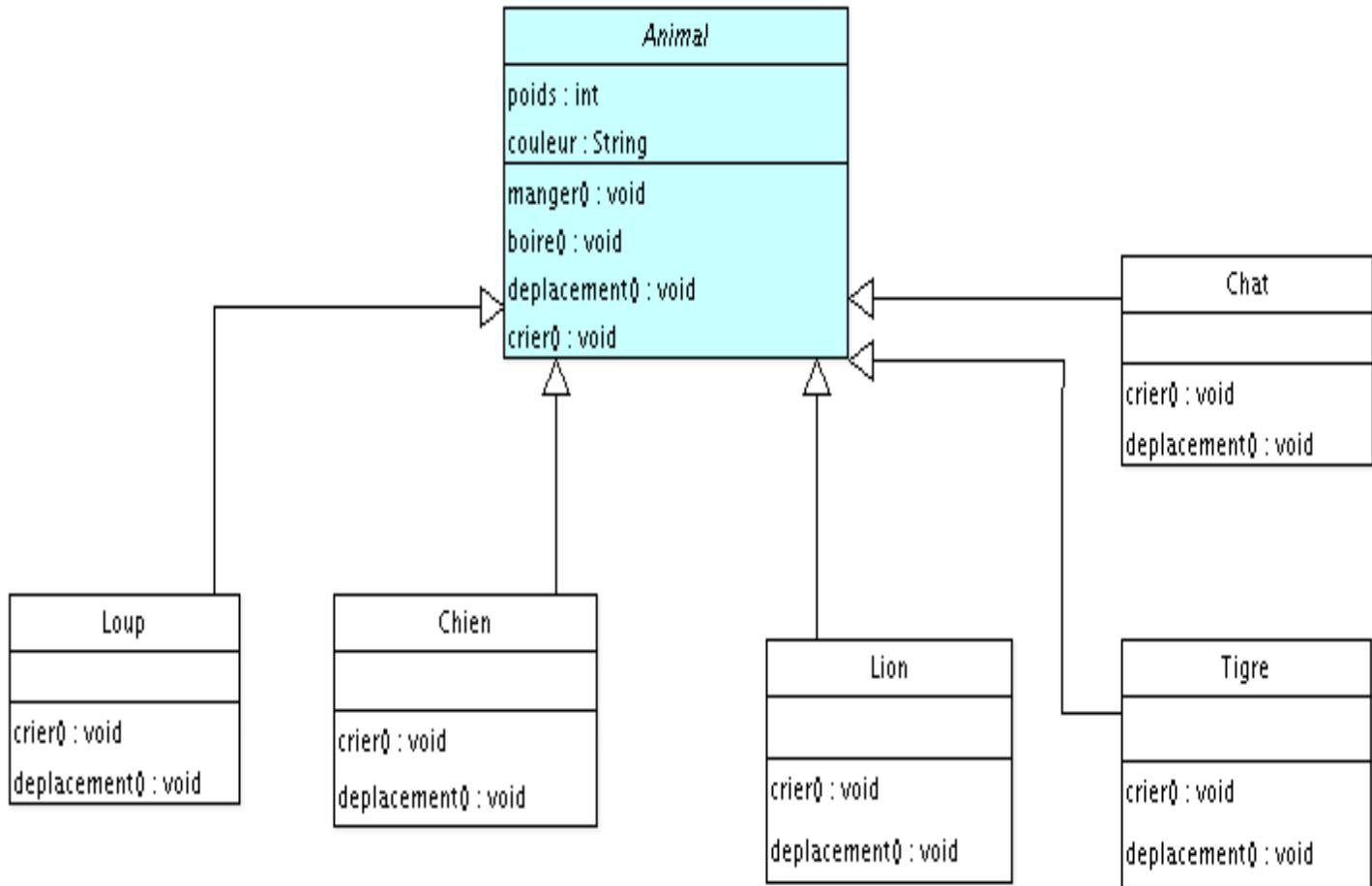
Nous avons instancié un objet **Loup** que nous avons mis dans un objet de type **Animal** qui n'a pas été instancié !!!

# EXEMPLE D'APPLICATION

- Dans ce programme, nos objets auront tous une couleur et un poids différents. Nos classes auront donc le droit de modifier ceux-ci.
- tous nos animaux mangeront de la viande. Donc, la méthode **manger()** sera définie dans la classe Animal.
- Idem pour la méthode **boire()**. Ils boiront tous de l'eau .
- Par contre, **ils ne crient pas et ne se déplaceront pas de la même manière**. Nous ferons donc des méthodes polymorphes et déclarerons les méthodes **crier()** et **deplacement()** abstraites dans la classe Animal.

# EXEMPLE D'APLICATON

Voici ce que donneraient nos classes :



# EXEMPLE D'APLICATON

- Maintenant, nous voulons créer deux sous classes qui permettent de préciser la manière de déplacement de chaque animal !! les **félins** se déplacent d'une certaine façon, et les **canins** d'une autre.
- Implémenter les classes de diagramme de classe suivant. Créer une classe de Test qui affiche le résultat suivant:

Je bois de l'eau !

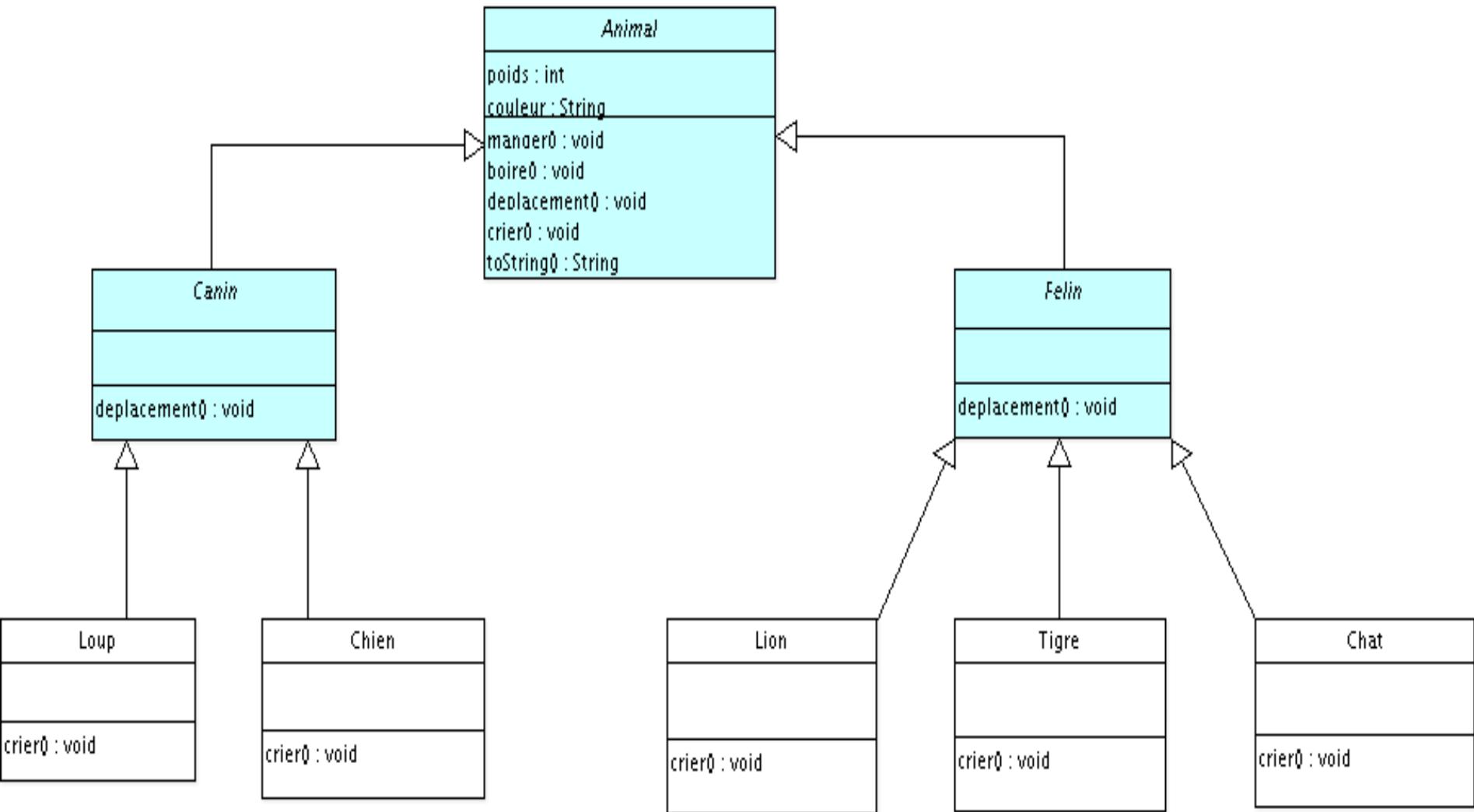
Je mange de la viande

Je me déplace en meute !

J'hurle à la lune

Je suis un objet de la class Loup, je suis Gris bleuté, je pèse 20

# EXEMPLE D'APPLICATION



# Classe Interface

66

# CONCEPT D'INTERFACE

- Si on considère une classe abstraite **n'implantant aucune méthode et aucun champ** (sauf les constantes), on aboutit à la notion **d'interface**.
- Une interface c'est une collections de méthodes abstraites.
- Une interface représente une classe abstraite à 100%.
- Une classe X peut **implémenter** plusieurs interfaces.
- Les interfaces peuvent se dériver.

..

# DÉFINITION D'UNE INTERFACE

- La définition d'une interface est identique à la définition des classes, en remplaçant le mot clé **class** par **interface**.
- Les méthodes d'une interface sont toutes **abstraites** et **publiques**, il n'est donc pas **obligatoire** de le mentionner comme **abstraites**. **Exemple:**

**public interface NomInterface {....}**

```
public interface I
{
    final int MAX = 100;
    void f(int n); // public et abstract sont facultatifs
    void g();
}
```

# CONCEPT D'INTERFACE

- Maintenant, nous voulons utiliser l'architecture précédente de classe abstraite dans une autre application où les chiens vont devoir apprendre à faire de nouvelles choses comme : faire des câlins.
- Donc, on vas ajouter cette méthode dans la classe **Animal !!**
- **Mais NON**, car on auras des lions qui vont faire des câlins !!
- Dans ce cas, on n'a qu'à mettre cette méthode dans la classe **Chien !**
- **Mais avec cette solution**, vous ne pourrez pas appeler vos objets Chien par le biais d'un super type. Pour pouvoir accéder à cette méthode, vous devrez obligatoirement passer par une référence à un objet **Chien. ADIEU LE POLYMORPHISME !**

# CONCEPT D'INTERFACE

- Pour utiliser au mieux le **polymorphisme**, nous devions définir les méthodes au plus haut niveau de la hiérarchie. Et comme l'héritage multiple est interdit en Java !!
- Il faudrait pouvoir développer un nouveau super type qui est **les interfaces**. Donc, nous allons créer l'interface **Cal** pour ensuite l'implémenter dans notre objet **Chien**, comme suit:

```
public interface Cal{  
    public void faireCalin(); }
```

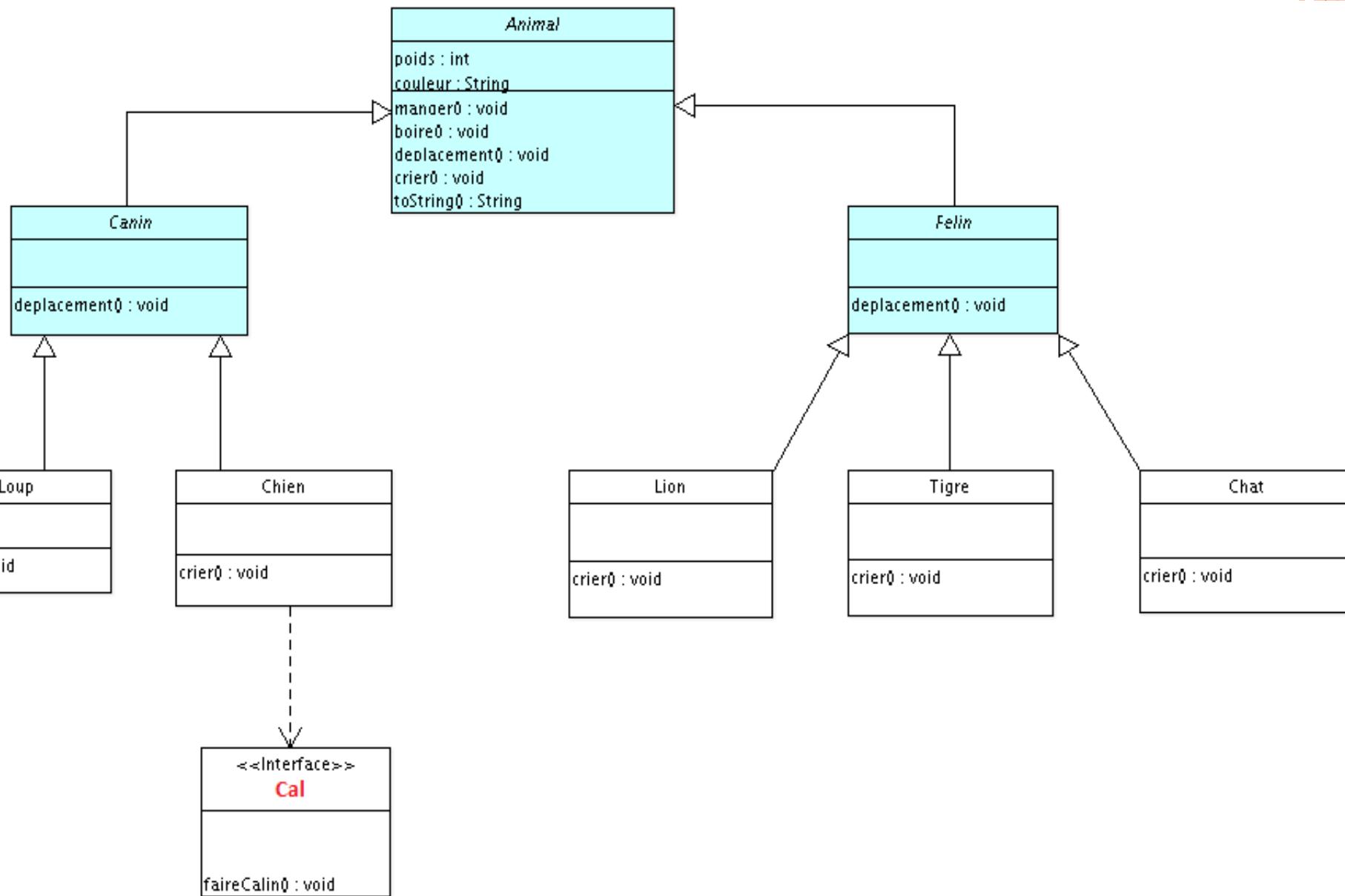
- il ne nous reste plus qu'à implémenter l'interface dans notre classe **Chien**. Ce qui nous donne :

# CONCEPT D'INTERFACE

```
public class Chien extends Canin implements Cal {  
    public Chien(){ }  
    public Chien(String couleur, int poids){  
        this.couleur = couleur;  
        this.poids = poids; }  
    void crier() {  
        System.out.println("J'aboie sans raison ! "); }  
    public void faireCalin() {  
        System.out.println("Je te fais un GROS CÂLIN"); }  
}
```

L'ordre des déclarations est PRIMORDIAL. Vous DEVEZ mettre l'**expression d'héritage AVANT l'expression d'implémentation**, SINON votre code ne compilera pas !

# CONCEPT D'INTERFACE



# IMPLEMENTATION D'UNE INTERFACE

- Une classe peut **implémenter** une ou plusieurs interfaces

## Exemple

```
class A implements I
{ /** A doit redéfinir toutes les méthodes de I
    sinon erreur à la compilation **/ }
```

## Exemple

```
public interface I1
{ void f(); }
public interface I2
{ void g(); }
class A implements I1, I2
{// A doit définir les méthodes f et g
}
```

# IMPLEMENTATION D'UNE INTERFACE

- Supposons que la méthode « g » est présente de 2 façons différentes (`void g()` et `int g()`) dans I1 et I2.

Exemple:

```
class A implements I1, I2 {  
    /*** Erreur car void g() et int g() ne peuvent pas  
    coexister au sein de la même classe d'après les  
    règles de la surcharge des méthodes***/
```

# INTERFACE ET CLASSE DÉRIVÉE

- Le mot clé **implements** est une garantie de la part d'une classe d'implémenter toutes les méthodes proposées dans une interface. Une classe dérivée peut implémenter une ou plusieurs interfaces.

## Exemple 1:

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

class A {...}

class B extends A implements I
// les méthodes f et g doivent
// être soit déjà définies dans A
// soit définies dans B
}
```

## Exemple 2:

```
interface I1{...}
interface I2{...}
class A implements I1{...}
class B extends A implements I2
{...}
```

# DÉRIVATION D'UNE INTERFACE

- On peut définir une interface comme étant **dérivée d'une autre interface** (mais pas d'une classe) en utilisant le mot clé **extends**. La dérivation d'interfaces revient simplement à **concaténer les déclarations et n'est pas aussi riche que celles des classes**.

```
interface I1
{
    static final MAXI = 100;
    void f (int n);
}
interface I2 extends I1
{
    static final MINI = 10;
    void g();
}
```



```
interface I2
{
    static final MINI = 10;
    static final MAXI = 100;
    void f (int n);
    void g();
}
```

# DÉRIVATION D'UNE INTERFACE

- Une interface peut dériver de plusieurs interfaces. L'héritage multiple est autorisé pour les interfaces.

```
interface I1
{ void f(); }
interface I2
{ void f1(); }
interface I3 extends I1, I2
{ void f2(); }
```

..

# AVANTAGE DE CLASSE INTERFACE

- Soit les classes définies comme suit:
- Public **interface** Animal {....}
- Public **class** Chat **implements** Animal {...}
- On ne peut créer un objet de la classe **Animal**,
- On peut créer des objets de la classe **Chat** ainsi:
  - **Chat** cat = new Chat();
  - ou
  - **Animal** cat=new Chat();
- La conversion est implicite en Java.
- En utilisant la liaisons dynamiques de Java.

..

```
public interface Vehicule {  
    int nbr_roues=4; // automatiquement final  
    public void klaxonner(); // automatiquement abstract  
}
```

```
Class Camion implements Vehicule{
```

```
    @Override
```

```
        public String klaxonner() { return « BomBom » ;}  
    }
```

```
Class Voiture implements Vehicule{
```

```
    @Override
```

```
        public String klaxonner() { return « TitTit» ;}  
    }
```

```
}
```

Dans main, nous pouvons créer les objets enfants ainsi:

**Voiture** v = new **Voiture**();

Ou

**Vehicule** v = new **Voiture**();

# EXEMPLE DE L'INTERFACE COMPARABLE DE JAVA

- L'interface Comparable de Java consiste en une seule méthode (et pas de constantes) int compareTo(T obj) qui compare l'objet à un objet de type T .
- A.compareTo(B) retourne un entier négatif si l'objet A est plus petit que B, zéro si les deux objets sont égaux et un entier positif si l'objet A est plus grand que l'objet B.

Exemple:

- "Bonjour ".compareTo ("Bonsoir ") renvoie un entier négatif.

..

# EXEMPLE DE L'INTERFACE COMPARABLE DE JAVA

- Une classe implémente l'interface Comparable si ses objets peuvent être ordonnés selon un ordre particulier.
- Par exemple la classe String implémente Comparable parce que les chaînes de caractères peuvent être ordonnées selon l'ordre alphabétiques.
- Les classes numériques comme Integer et Double implémentent Comparable parce que les nombres peuvent être ordonnés selon l'ordre numérique.

..

# Exercices

# EXERCICE

- Créez une classe CompteEnBanque munie d'un seul attribut solde, et définissez l'égalité de deux objets CompteEnBanque, de telle manière qu'elle soit vérifiée dès que les deux soldes sont égaux.

# SOLUTION

```
class CompteEnBanque implements Comparable{  
  
    private float solde;  
  
    public float getSolde(){ return solde; }  
    public void setSolde(float s){ solde=s; }  
  
    public int compareTo(Object cmp) {  
  
        float solde2 = ((CompteEnBanque)cmp).getSolde();  
  
        if(this.solde>solde2)  
            return 1;  
        else if(this.solde <solde2)  
            return -1;  
        else  
            return 0;  
    }  
}
```

# EXERCICE: INTERFACE

```
interface Affichable{  
    void afficher() ;  
}
```

```
class Entier implements  
Affichable{  
  
private int val ;  
public Entier (int n) {val = n ;}  
  
public void afficher() {  
    System.out.println (« Je suis  
un entier de valeur » + val) ;  
}
```

```
class Flottant implements  
Affichable{  
  
private float val ;  
public Flottant (float n) {val = n ;}  
  
public void afficher() {  
    System.out.println (« Je suis un  
flottant de valeur » + val) ;  
}
```

# EXERCICE: INTERFACE

```
public class TestInterface {  
    public static void main (String [] args) {  
        Affichable [] tab = new Affichable [2];  
        tab[0] = new Entier (25);  
        tab[1] = new Flottant (1.25);  
        tab[0].afficher();  
        tab[1].afficher(); } }
```

## Résultat de l'exécution

- Je suis un entier de valeur 25
- Je suis un flottant de valeur 1.25

# Classe Finale

# CLASSE FINALE

- Ne peut être hérité par aucune autre classe ni de redéfinir ses méthodes.
- L'intérêt est de protéger les classes et leurs implémentations.
- Définie avec le mot clé **final**.

..

## Syntaxe:

```
public final class NomClasse{...}
```

# CLASSE FINALE

- La classe **String** de Java est une classe final.
- Il est impossible de créer des sous classes de String, par mesure de sécurité Java.
- La classe **Math** de Java également.

66

# UML

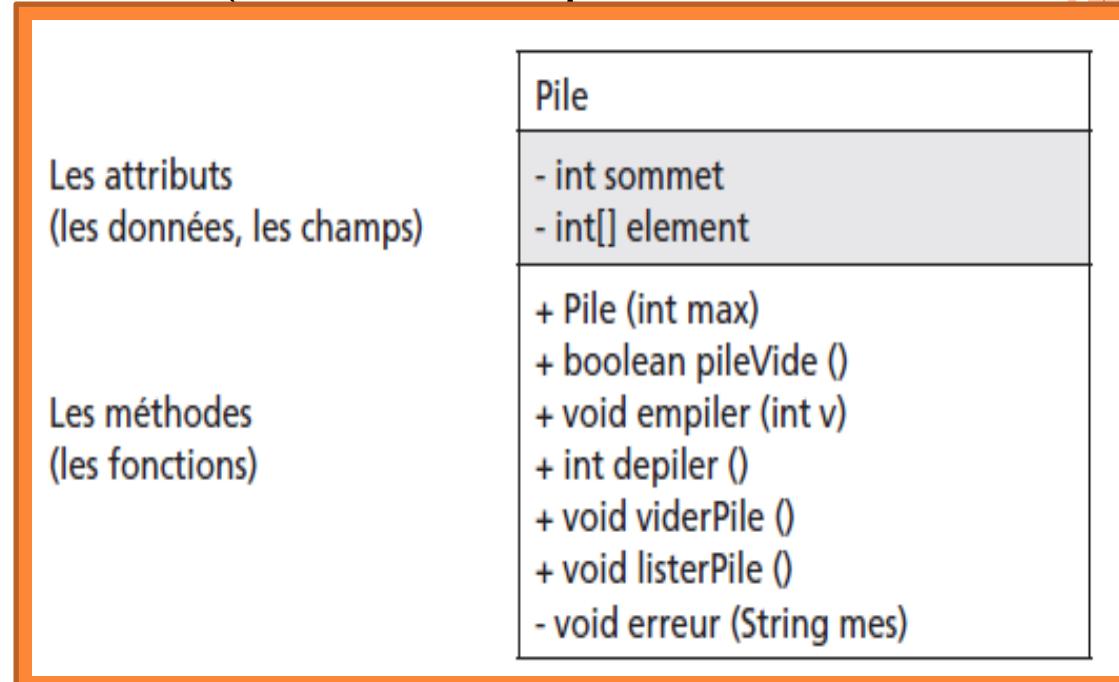
## REPRÉSENTATION D'UNE CLASSE EN UML

- UML, c'est l'acronyme anglais pour « Unified Modeling Language ». On le traduit par « Langage de modélisation unifié ». La notation UML est un langage visuel constitué d'un ensemble des diagrammes qui donnent chacun une vision différente du projet à traiter. UML nous fournit donc des diagrammes pour représenter le logiciel à développer : son fonctionnement, sa mise en route, les actions susceptibles d'être effectuées par le logiciel, etc.
- Dans ce module, nous utiliserons le diagramme de classe.

# REPRÉSENTATION D'UNE CLASSE EN UML

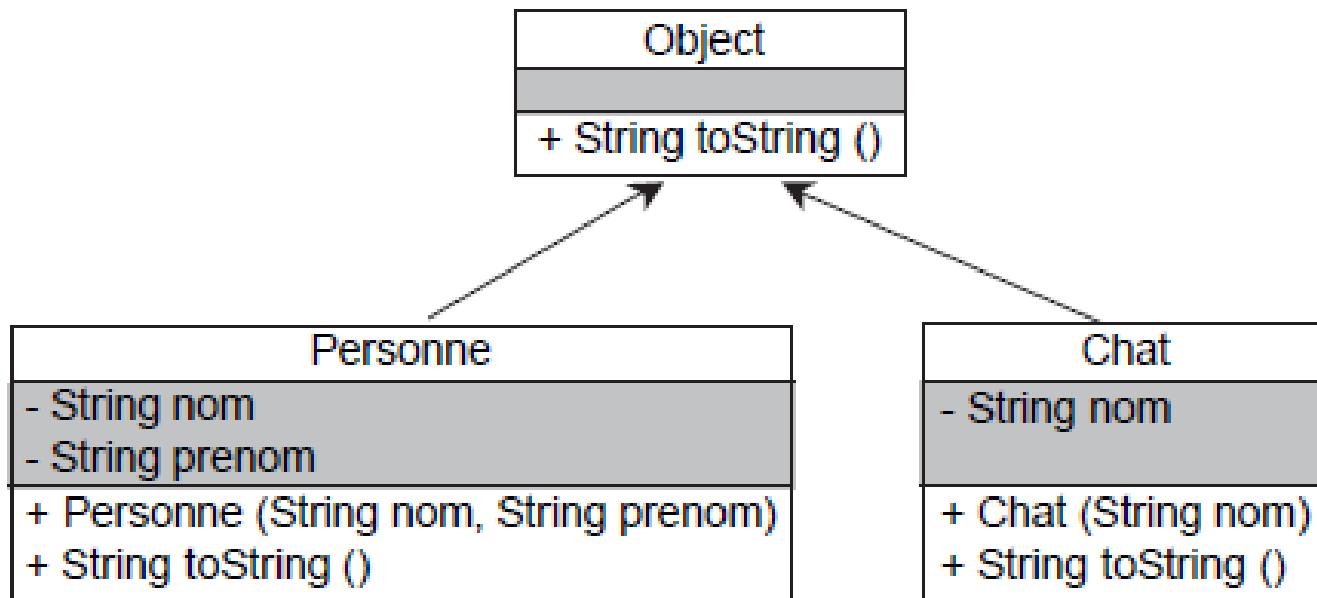
Rectangle composé de quatre parties:

- **Partie 1:** Nom de la classe (commence par une majuscule, en gras)
- **Partie 2:** Attributs
- **Partie 3:** Méthodes



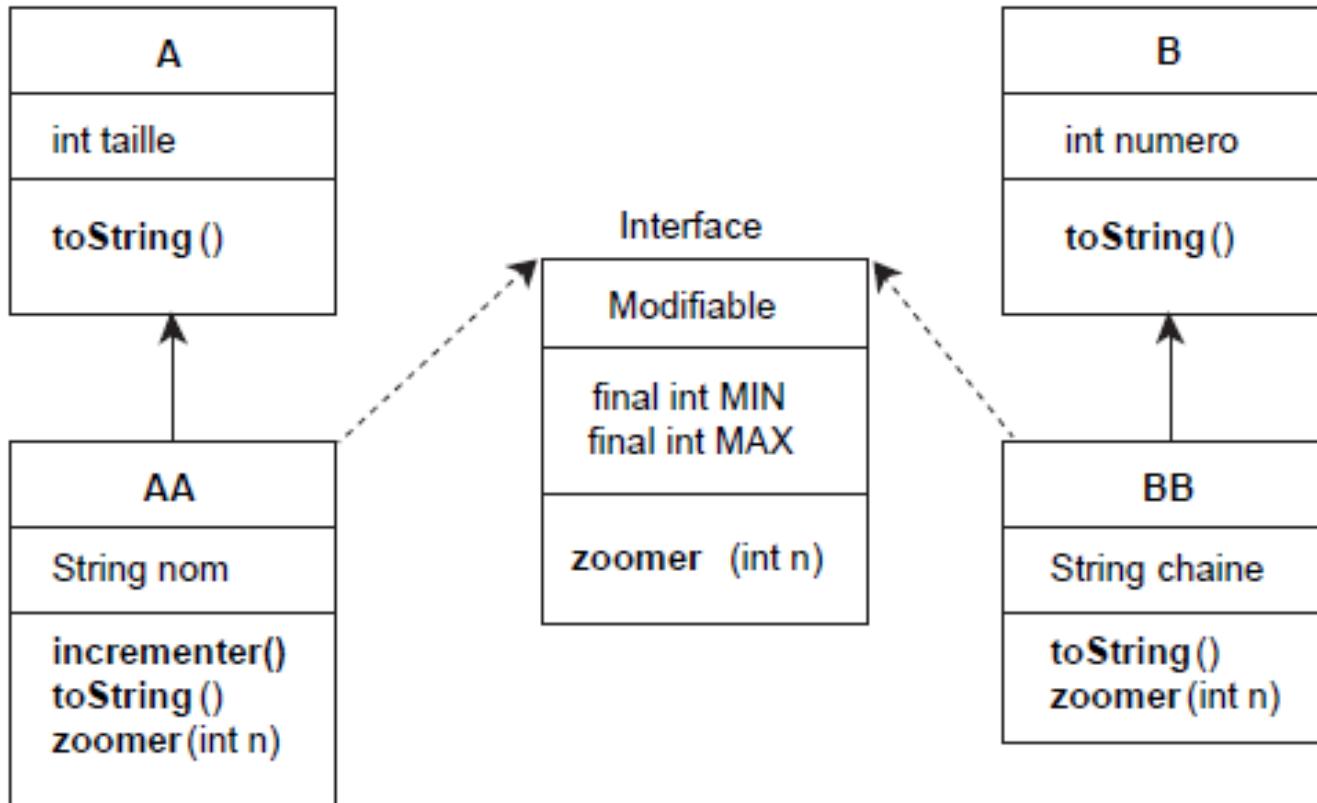
- indique un attribut / méthode **private**
- + indique un attribut / méthode **public**
- # indique un attribut / méthode **protected**
- Rien indique un attribut / méthode **par défaut**

# EXEMPLE DE MODÉLISATION: HÉRITAGE



Les classes Personne et Chat héritent par défaut de la classe Object.

# EXEMPLE DE MODÉLISATION: HÉRITAGE & INTERFACE

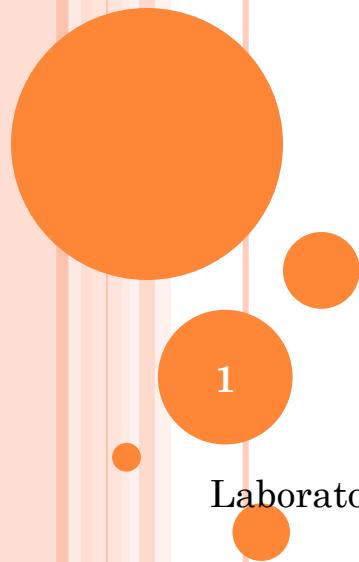


Héritage et interface.

Merci !!

Cours  
**Programmation Orientée Objet 2**  
Pour  
**ING 2**

**Chap 02:**  
**Les exceptions**



MEKAHLIA Fatma Zohra LAKRID  
Maître de Conférences Classe B

Laboratoire de Modélisation, Vérification et Evaluation des Performances des systèmes  
complexes (MOVEP)  
Bureau 123

# PLAN

- Introduction.
- Définition de la notion d'exception.
- Les différents types d'exceptions.
- Hiérarchie des classes d'exception.
- Gestion des exceptions (bloc try .. catch).
- Plusieurs exceptions dans un bloc try..catch..finally.
- Déclenchement manuel d'une exception prédéfinie.
- Définir une classe d'exception.
- Utiliser une classe d'exception définie par le programmeur.
- Capturer une exception définie par le programmeur.

# INTRODUCTION

- Il vous est sûrement déjà arrivé d'avoir un gros message affiché en rouge dans la console d'eclipse : ceci a été généré par une exception **qui n'a pas été capturée**.
- La gestion des exceptions s'appelle aussi **la capture d'exception** !
- Donc, un programme Java doit traiter des **situations inattendues** (exceptionnelles) en dehors de sa tâche principale.



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The console window displays a red error message:

```
<terminated> TestSansException [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (14 déc. 2021 à 13:11:16)
Exception in thread "main" java.lang.ArithmetricException: / by zero
at TestSansException.main(TestSansException.java:7)
```

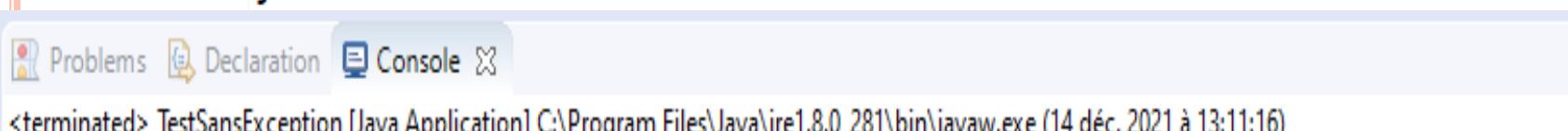
# DÉFINITION DE LA NOTION D'EXCEPTION

- Une exception est un événement qui se produit lors de l'exécution d'un programme et qui perturbe le flux normal des instructions du programme.
- Le principe de la gestion consiste à repérer un morceau de code qui pourrait générer une exception (une division par zéro, par exemple):
  - 1) De détecter et traiter ces erreurs en utilisant les trois mots clés (try, catch et finally ).
  - 2) Ou bien, de les lever ou les propager en utilisant (throw et throws).
- Et dans le but d'afficher un message personnalisé et de continuer le traitement de programme principal.

# DIFFÉRENTS TYPES D'EXCEPTIONS: RUNTIMEEXCEPTION

exception de la division par zéro:  
**ArithmeticException**

```
2 public class TestSansException {  
3  
4 public static void main(String[] args) {  
5 int i = 10;  
6 int j = 0;  
7 System.out.println("résultat = " + (i / j));  
8 System.out.println("et moi !!je ne serai jamais exécutée !");  
9 }  
0 }
```



Exception in thread "main" java.lang.ArithmeticException: / by zero  
at TestSansException.main(TestSansException.java:7)

## RunTimeException:

### Exception de la division par zéro: **ArithmeticException**

- lorsque l'exception a été levée, **le programme s'est arrêté !**
- Lorsqu'un événement que la JVM ne sait pas gérer, une exception est levée (**ex : division par zéro**) !
- le type de l'exception qui a été déclenchée (levée) est **ArithmeticException**.
- On dit, qu'une division par zéro est une ArithmeticException.
- Donc, nous allons pouvoir **la capturer**, et réaliser **un traitement** en conséquence.

# DIFFÉRENTS TYPES D'EXCEPTIONS: RUNTIMEEXCEPTION

exception de débordement d'indices dans un tableau: **ArrayIndexOutOfBoundsException**

```
public class TestException1 {  
  
    public static void main (String[] args) {  
        int tabEnt[] = { 1, 2, 3, 4 };  
  
        // Exception de type java.lang.ArrayIndexOutOfBoundsException  
        // non contrôlée par le programme; erreur d'exécution  
        tabEnt [4] = 0;                                // erreur et arrêt  
  
        System.out.println ("Fin du main");      // le message n'apparaît pas  
    } // main  
  
} // class TestException1
```

# DIFFÉRENTS TYPES D'EXCEPTIONS: RUNTIMEEXCEPTION

exception de débordement d'indices dans un tableau: **ArrayIndexOutOfBoundsException**

- L'exécution du programme ci-dessus provoque un message d'erreur indiquant une exception du type **ArrayIndexOutOfBoundsException**, et l'arrêt du programme. Le tableau de 4 entiers tabEnt a des indices de 0 à 3. L'accès à la valeur d'indice 5 provoque une exception non captée.

# DIFFÉRENTS TYPES D'EXCEPTIONS: RUNTIMEEXCEPTION

Accès au contenu d'un objet null :  
**NullPointerException**

## Exemple:

- String s1 = null;
- System.out.println (s1.charAt (3)); // exception de type **NullPointerException**.
  
- s1 est une référence null sur un (futur) objet de type String. On ne peut pas accéder au troisième caractère de l'objet String référencé par null.

# DIFFÉRENTS TYPES D'EXCEPTIONS: RUNTIMEEXCEPTION

Transfert de type non cohérent de deux objets de classes différentes:

## ClassCastException

### Exemple:

- Object p = new Point();
- Color coul = (Color) p; // exception

**ClassCastException** : p n'est pas de la classe Color

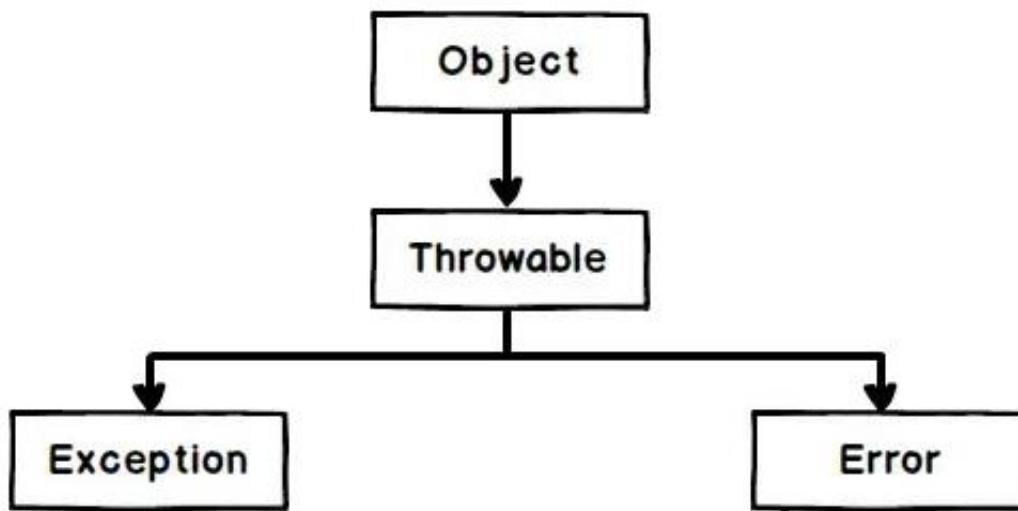
# DIFFÉRENTS TYPES D'EXCEPTIONS: RUNTIMEEXCEPTION

Lecture de nombres erronés:  
**NumberFormatException**

- Exemple: la lecture d'un nombre entier depuis un champ de texte.
- int n1 = Integer.parseInt ("231"); // exception car la chaîne "231" n'est pas un entier
- System.out.println ("n1 : " + n1);

# HIÉRARCHIE DES CLASSE D'EXCEPTION

- En Java, une erreur (exception) ne provoque pas l'arrêt brutal du programme mais **la création d'un objet Throwable**. Cette classe descend directement de la classe Object : c'est la classe de base pour le traitement des exceptions en Java.



# HIÉRARCHIE DES CLASSE D'EXCEPTION

Cette classe possède deux constructeurs :

- **Throwable();**
- **Throwable(String);**

Les principales méthodes de la classe Throwable sont :

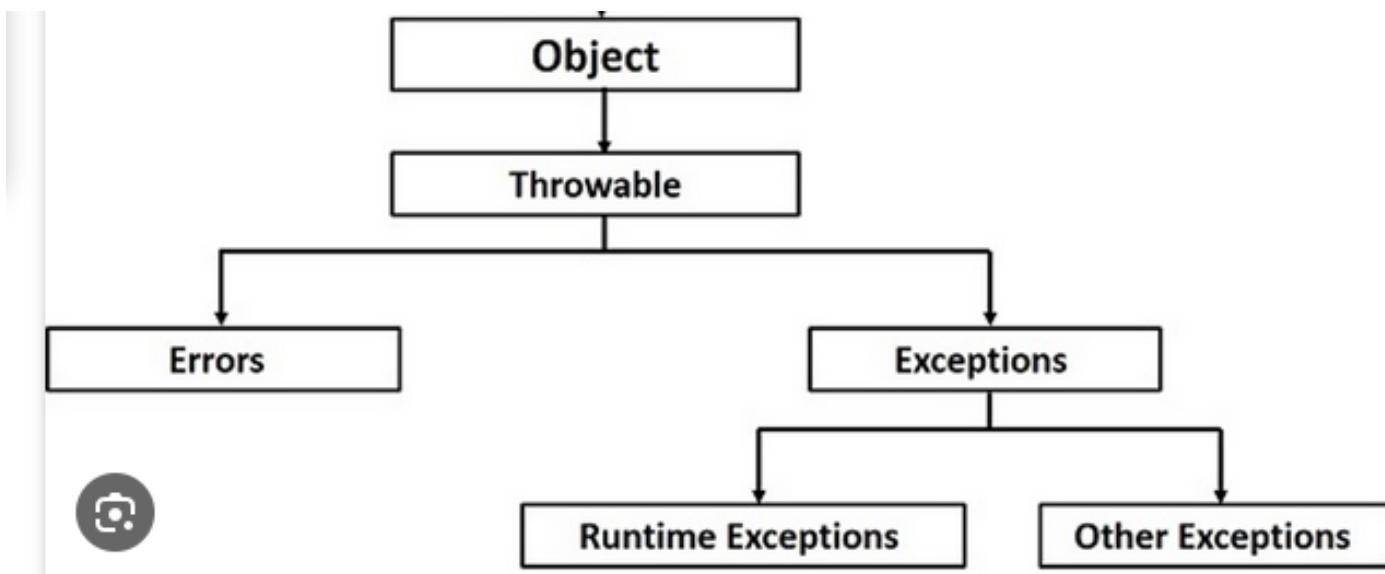
- **String getMessage( ); // lecture du message**
- **void printStackTrace( ); /\* affiche l'exception et l'état de la pile d'exécution au moment de son appel\*/**
- **void printStackTrace(PrintStream s); /\* Idem mais envoie le résultat dans un flux \*/**

# HIÉRARCHIE DES CLASSE D'EXCEPTION

- **Exception** et **Error** sont des sous-classes de la classe **Throwable**.
- La sous-classe **Error** est formée des exceptions qui ne sont pas considérées comme récupérables. Elle est utilisée par Java Virtual Machine (**JVM**) et traite les exceptions qui ne sont pas *liée au développeur* c.-à-d. ce sont des choses liées soit au **matériel** soit aux **d'autres trucs** par exemple **réseaux**.
- Exemple: dans un programme Java qui veut stocker des données sur le disque, mais le disque est plein ! **OutOfMemoryError**

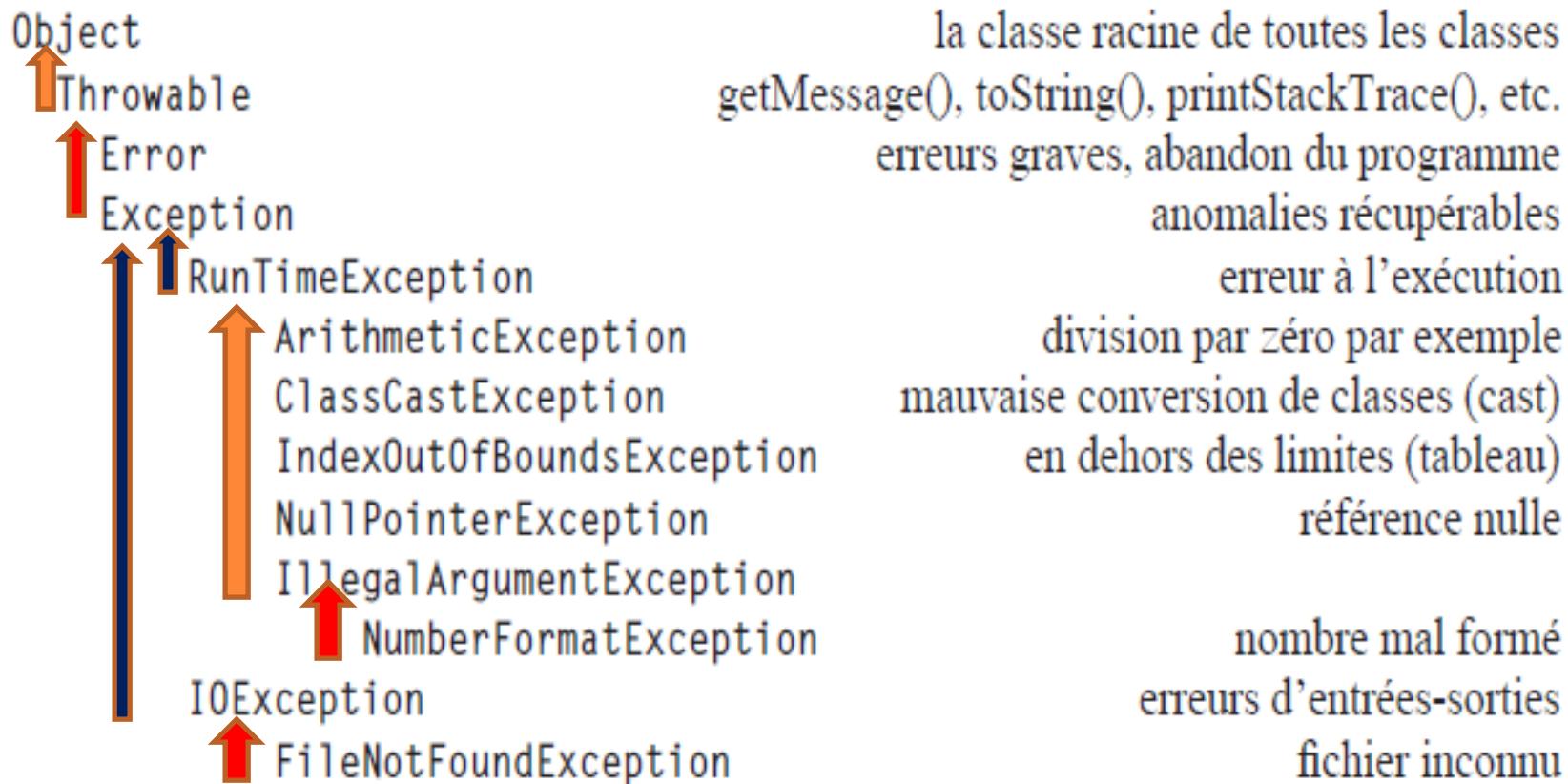
# HIÉRARCHIE DES CLASSE D'EXCEPTION

- La sous-classe **Exception** représente des erreurs moins graves qui peuvent être gérées au moment de l'exécution. C'est la classe mère des classes **RunTimeException**, **IOException**, **SQLException**, etc.



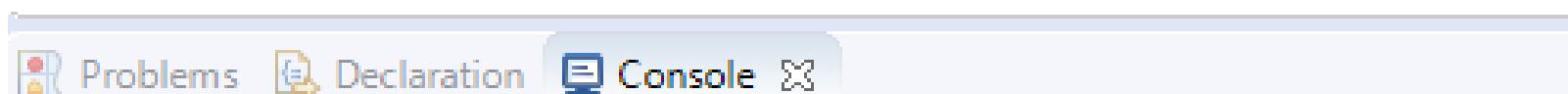
# HIÉRARCHIE DES CLASSE D'EXCEPTION

Les exceptions constituent un arbre hiérarchique utilisant la notion d'héritage.



# GESTION DES EXCEPTIONS (BLOC TRY .. CATCH)

```
2 public class TestAvecException {  
3     public static void main(String[] args) {  
4         int i = 10;  
5         int j = 0;  
6  
7         try {  
8             System.out.println("résultat = " + (i / j));  
9         } catch(ArithmetricException e) {  
10             System.out.println("Erreur de division par zéro !");  
11  
12             System.out.println("En fin !!je pourrai exécuter maintenant!");  
13         }  
14     }
```



# GESTION DES EXCEPTIONS (BLOC TRY .. CATCH)

- Le bloc **catch** contient un objet de type **ArithmeticException** en paramètre. Nous l'avons appelé **e**.
- Comme l'exception de type ArithmeticException est capturée, le déroulement normal du programme est interrompu et l'instruction du bloc catch de même type s'exécute ! Par la suite, le programme principal reprend son déroulement normal.
- De ce fait, notre message d'erreur personnalisé s'affiche à l'écran.
- Maintenant, le paramètre de notre bloc catch (**e**), il **sert à quoi, au juste ?**

# GESTION DES EXCEPTIONS (BLOC TRY .. CATCH)

- Il sert à savoir quel type d'exception doit être capturé.
- Voici le même test, en remplaçant l'instruction du catch par celle-ci :

```
try {  
    System.out.println("résultat = " + (i / j));  
} catch(ArithmetricException e) {  
    System.out.println("Division par zéro !" + e.getMessage());  
}  
//System.out.println("Erreur de division par zéro !");  
System.out.println("En fin !!je pourrai exécuter maintenant!");  
}
```

Division par zéro !/ by zero  
En fin !!je pourrai exécuter maintenant!

```
1 public class TestException {  
2     public static void main(java.lang.String[] args) {  
3         // Insert code to start the application here.  
4         int i = 3;  
5         int j = 0;  
6         try {  
7             System.out.println("résultat = " + (i / j));  
8         } catch (ArithmaticException e) {  
9             System.out.println("getmessage");  
10            System.out.println(e.getMessage());  
11            System.out.println(" ");  
12            System.out.println("toString");  
13            System.out.println(e.toString());  
14            System.out.println(" ");  
15            System.out.println("printStackTrace");  
16            e.printStackTrace();  
17        }  
18    }
```

Résultat :

```
1 C:>java TestException  
2 getmessage  
3 / by zero  
4  
5 toString  
6 java.lang.ArithmaticException: / by zero  
7  
8 printStackTrace  
9 java.lang.ArithmaticException: / by zero  
10          at tests.TestException.main(TestException.java:24)
```

# PLUSIEURS EXCEPTIONS DANS UN BLOC TRY..CATCH..FINALLY

```
try {  
    opération_risquée1;  
    opération_risquée2;  
}  
    catch (ExceptionDeClasseFilleException e) { traitements }  
    catch (ExceptionParticulière e) { traitements }  
    catch (ExceptionDeClasseMèreException e) { traitements }  
finally { traitement_pour_terminer_proprement; }
```

# PLUSIEURS EXCEPTIONS DANS UN BLOC

## TRY..CATCH..FINALLY

- S'il y a **plusieurs types d'exceptions** à intercepter, il faut définir autant de blocs catch que de types d'événements.
- Il faut faire attention à **l'ordre des clauses catch**, un type d'exception d'une **sous-classe doit venir avant** un type d'une exception d'une **super-classe**.
- C'est dans le but de traiter **en premier** les exceptions les **plus précises (sous-classes)** avant les exceptions plus générales (super-classe), car le traitement des exceptions sera transmit au bloc de niveau supérieur (super-classe)

..

# PLUSIEURS EXCEPTIONS DANS UN BLOC TRY..CATCH..FINALLY

- Si vous ne respectez pas l'ordre attendu, l'erreur de compilation suivante sera produite. **Exemple:**

```
public class TestException {  
    public static void main(java.lang.String[] args) {  
        int i = 3;  int j = 0;  
        try { System.out.println("résultat = " + (i / j));  
        } catch (Exception e) {}  
        catch (ArithmeticException e) {}  
    }  
}
```

```
TestException.java:11: catch not reached.  
        catch (ArithmeticException e) {  
            ^  
1 error
```

# PLUSIEURS EXCEPTIONS DANS UN BLOC TRY..CATCH..FINALLY

- le message précise que l'exception **ArithmeticException** est déjà attrapée.
- Effectivement, comme la classe `java.lang.ArithmeticException` dérive de la classe `java.lang.Exception`, elle est déjà gérée par le premier bloc catch associé au type **Exception**.

..

# PLUSIEURS EXCEPTIONS DANS UN BLOC TRY..CATCH..FINALLY

- Le bloc **finally** est en général utilisé pour effectuer des nettoyages (fermer des fichiers, libérer des ressources...).
- Dans tout les cas de l'exécution ou non du catch, le bloc **finally** est exécutée mais il est optionnel.

..

# LES EXCEPTIONS PRÉDÉFINIES

- Toute méthode pouvant lancer une exception contrôlée doit contenir soit un bloc try/catch/finally (si elle traite l'exception localement).
- Soit utilise le mot clé **throws** ( si l'exception est traitée par une autre méthode).

# LES EXCEPTIONS PRÉDÉFINIES

- le mot clé **throws** signifie qu'**une méthode est susceptible de lever une exception** mais elle ne peut pas la traiter localement.

```
public void ma_methode (int x) throws IOException {  
...  
}
```

- Il est également possible de désigner l'interception de plusieurs exceptions :

```
public void ma_methode (int x) throws IOException, EOFException{  
...  
}
```

# DÉFINIR UNE CLASSE D'EXCEPTION

- Jusqu'à présent on a parlé des exceptions prédéfinies qui se déclenchent toutes seules. Java offre au programmeur la possibilité de définir ses propres exceptions. Ces exceptions doivent hériter de la classe Exception. Le programmeur doit lui-même lever ses exceptions.
- Pour se faire Java met à sa disposition le mot-clé **throw** (**à ne pas confondre avec throws**).

»

# DÉFINIR UNE CLASSE D'EXCEPTION

## throws:

- Ce mot clé permet de dire à une instruction Java (condition, déclaration de variable...) ou à une classe entière qu'une exception potentielle sera gérée par une autre classe.
- Ce mot clé **est suivi** du nom de la classe qui va gérer l'exception. Ceci a pour but de définir le type d'exception qui risque d'être générée par l'instruction, ou la classe qui précède le mot clé **throws**.

..

# DÉFINIR UNE CLASSE D'EXCEPTION

**throw:**

- Celui-ci permet d'instancier un objet dans la classe suivant l'instruction **throws**. Cette instruction est suivie du mot clé **new** ainsi que d'un objet cité avec **throws**. En fait, il lance une exception, tout simplement.

# UTILISER UNE CLASSE D'EXCEPTION DÉFINIE PAR LE PROGRAMMEUR

```
1 package ExecpPersonalisé;  
2  
3 public class Ville{  
4     |  
5     String nomVille;  
6     int nbreHabitant;  
7     public Ville(String nomVille, int nbreHabitant){  
8         this.nomVille = nomVille;  
9         this.nbreHabitant = nbreHabitant;  
0     }  
1 }
```

**Q:** Si l'utilisateur crée une instance ville avec un nombre d'habitants négatif ?

**R:** pour remédier, le programmeur doit créer un objet de type NombreHabitantException (qui hérite d'Exception).

Exemple: NombreHabitant**Exception** extends **Exception**

# UTILISER UNE CLASSE D'EXCEPTION DÉFINIE PAR LE PROGRAMMEUR

```
1 package ExecpPersonalisé;  
2  
3 public class NombreHabitantException extends Exception{  
4  
5     public NombreHabitantException(){  
6         |  
7         System.out.print("Vous essayez d'instancier une classe Ville");  
8         System.out.println("avec un nombre d'habitants négatif !");  
9     }  
0  
1 }
```

# CAPTURER UNE EXCEPTION DÉFINIE PAR LE PROGRAMMEUR

```
1 package ExecpPersonalisé;  
2  
3 public class Ville{  
4  
5     String nomVille;  
6     int nbreHabitant;  
7     public Ville(String nVil, int nHab) throws NombreHabitantException  
8     {  
9         if(nHab| < 0)  
0             throw new NombreHabitantException();  
1         else  
2         {  
3             nomVille = nVil;  
4             nbreHabitant = nHab;  
5         }  
6     }  
7 }
```

# CAPTURER UNE EXCEPTION DÉFINIE PAR LE PROGRAMMEUR

- `throws NombreHabitantException` nous indique que si une erreur est capturée, celle-ci sera traitée en tant qu'objet de la classe `NombreHabitantException` ! Ce qui, au final, nous renseigne sur le type de l'erreur en question.

`throw new NombreHabitantException();` instancie la classe `NombreHabitantException` si la condition `if(nHab < 0)` est remplie.

»

34

Merci ...



## SUITE ...

- Maintenant, pour crée un objet ville, le constructeur est devenu une **méthode à risque**, vous devrez gérer les exceptions possibles sur cette instruction. Avec un bloc **try{} catch{}**.

```
3 public class Main {  
4     public static void main(String[] args){  
5  
6         Ville v = new Ville("Blida", -15000);  
7     }  
8 }
```

# SUITE ...

The screenshot shows a Java code editor and a terminal window. The code in the editor is:

```
3 public class Main {  
4     public static void main(String[] args){  
5         try {  
6             Ville v = new Ville("Blida", -15000);  
7         } catch (NombreHabitantException e) {System.out.println("e = "+ e);}  
8     }  
9 }
```

The terminal window below shows the execution results:

Problems Declaration Console

<terminated> Main (6) [Java Application] C:\Program Files\Java\jre1.8.0\_281\bin\javaw.exe (15 déc. 2021 à 11:50:43)

Vous essayez d'instancier une classe Ville avec un nombre d'habitants négatif !  
e = ExecpPersonalisé.NombreHabitantException

- Maintenant ,l'erreur a disparu et le code compile et s'exécute correctement.

# LA GESTION DE PLUSIEURS EXCEPTIONS

- nous voulons lever une deuxième exception, si le nom de la ville fait moins de 3 caractères.

```
3 public class NomVilleException extends Exception {  
4     public NomVilleException(String message){  
5         super(message);  
6     }  
7 }
```

Avec cette redéfinition (`super`), nous pourrons afficher notre message d'erreur en utilisant la méthode `getMessage()`.

# LA GESTION DE PLUSIEURS EXCEPTIONS

```
public class Ville{  
String nomVille;  
int nbreHabitant;  
public Ville(String nVil, int nHab) throws NombreHabitantException,NomVilleException  
{  
    if(nHab < 0) throw new NombreHabitantException();  
    if(nVil.length() < 3)  
        throw new NomVilleException("le nom de la ville est inférieur à 3 caractères ! nom : "+nVil);  
    else  
    {  
        nomVille = nVil;  
        nbreHabitant = nHab;  
    } } }
```

# LA GESTION DE PLUSIEURS EXCEPTIONS

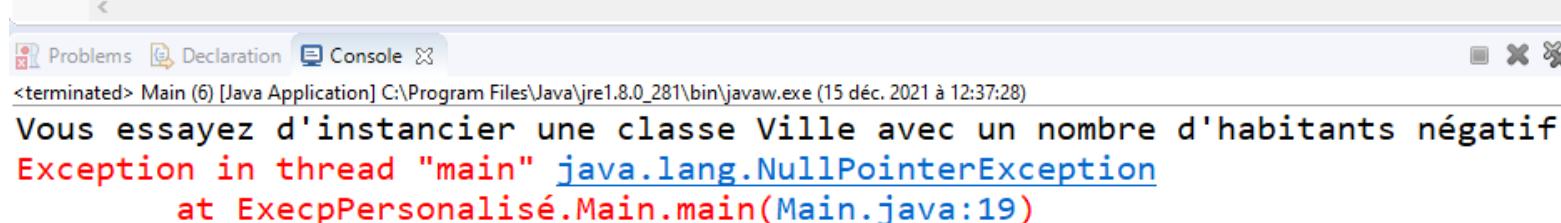
```
public static void main(String[] args) {  
    | try {  
    Ville v = new Ville("Bl", -15000);  
    }  
    //Gestion de l'exception sur le nombre d'habitants  
    catch (NombreHabitantException e) {  
        }  
    //Gestion de l'exception sur le nom de la ville  
    catch(NomVilleException e2){  
        System.out.println(e2.getMessage()); }  
    System.out.println(v.toString());  
}  
}
```

Vous pouvez constater qu'Eclipse affiche une erreur !!

R: car la déclaration de l'objet **Ville** est faite dans le premier sous-bloc **try{}}** et une variable déclarée dans un bloc d'instructions n'existe que dans ce bloc d'instructions ! **Pour pallier ce problème**, il nous suffit de déclarer notre objet en dehors du bloc **try{}** et de l'instancier à l'intérieur !

# LA GESTION DE PLUSIEURS EXCEPTIONS.

```
8     Ville v = null;
9     try {
10         v = new Ville("Bl", -15000);
11     }
12     //Gestion de l'exception sur le nombre d'habitants
13     catch (NombreHabitantException e) { }
14     //Gestion de l'exception sur le nom de la ville
15     catch(NomVilleException e2){
16         System.out.println(e2.getMessage()); }
17     System.out.println(v.toString());
18 }
```



The screenshot shows a Java development environment with the following details:

- Code Area:** Displays the Java code provided above, with line numbers 8 through 18.
- Console Output:** Shows the terminal window with the following text:
  - <terminated> Main [6] [Java Application] C:\Program Files\Java\jre1.8.0\_281\bin\javaw.exe (15 déc. 2021 à 12:37:28)
  - Vous essayez d'instancier une classe Ville avec un nombre d'habitants négatif
  - Exception in thread "main" java.lang.NullPointerException
  - at ExecpPersonalisé.Main.main(Main.java:19)

lorsque nous arrivons sur l'instruction  
"System.out.println(v.toString());", notre objet est **null** !  
Donc, Une **NullPointerException** est levée !

# LA GESTION DE PLUSIEURS EXCEPTIONS

```
8     Ville v = null;
9     try {
10         v = new Ville("Bl", -15000);
11     }
12     //Gestion de l'exception sur le nombre d'habitants
13     catch (NombreHabitantException e) { }
14     //Gestion de l'exception sur le nom de la ville
15     catch(NomVilleException e2){
16         System.out.println(e2.getMessage()); }
17     System.out.println(v.toString());
18 }
```

Problems Declaration Console

<terminated> Main (6) [Java Application] C:\Program Files\Java\jre1.8.0\_281\bin\javaw.exe (15 déc. 2021 à 12:37:28)

Vous essayez d'instancier une classe Ville avec un nombre d'habitants négatif  
Exception in thread "main" java.lang.NullPointerException  
at ExecpPersonalisé.Main.main(Main.java:19)

Pour pallier ce problème, Il suffit d'instancier un objet Ville par défaut dans le bloc **catch{}**. Grâce à cela, si notre instantiation avec valeur échoue, on fait une instantiation par défaut qui n'est pas une méthode à risque !,

# LA GESTION DE PLUSIEURS EXCEPTIONS

```
8     Ville v = null;
9     try {
10         v = new Ville("Bl", -15000);
11     }
12     //Gestion de l'exception sur le nombre d'habitants
13     catch (NombreHabitantException e) { v = new Ville(); }
14     //Gestion de l'exception sur le nom de la ville
15     catch(NomVilleException e2){
16         System.out.println(e2.getMessage());
17         v = new Ville();}
18     System.out.println(v.toString());
19 }
```

The screenshot shows an IDE interface with several tabs: Problems, Declaration, and Console. The Console tab is active and displays the following output:

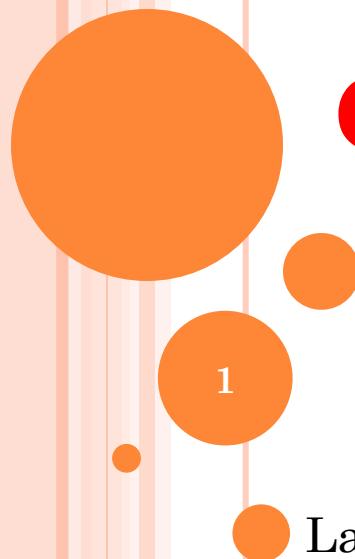
```
Problems Declaration Console
<terminated> Main (6) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (15 déc. 2021 à 15:50:12)
Vous essayez d'instancier une classe Ville avec un nombre d'habitants négatif
ExcepPersonalisé.Ville@15db9742
```

Si vous mettez un nom de ville de moins de 3 caractères, et un nombre d'habitants négatif, c'est l'exception du nombre d'habitants qui sera levée en premier ! Et pour cause... il s'agit de notre première condition dans notre constructeur

Merci !!

Cours  
**Programmation Orientée Objet 2**  
Pour  
**ING 2**  
**Chap 03:**

**Gestion des collections et  
Généricités**



MEKAHLIA Fatma Zohra LAKRID  
Maître de Conférences Classe B

Laboratoire de Modélisation, Vérification et Evaluation des  
Performances des systèmes complexes (MOVEP)  
Bureau 123

# Collections

# INTRODUCTION

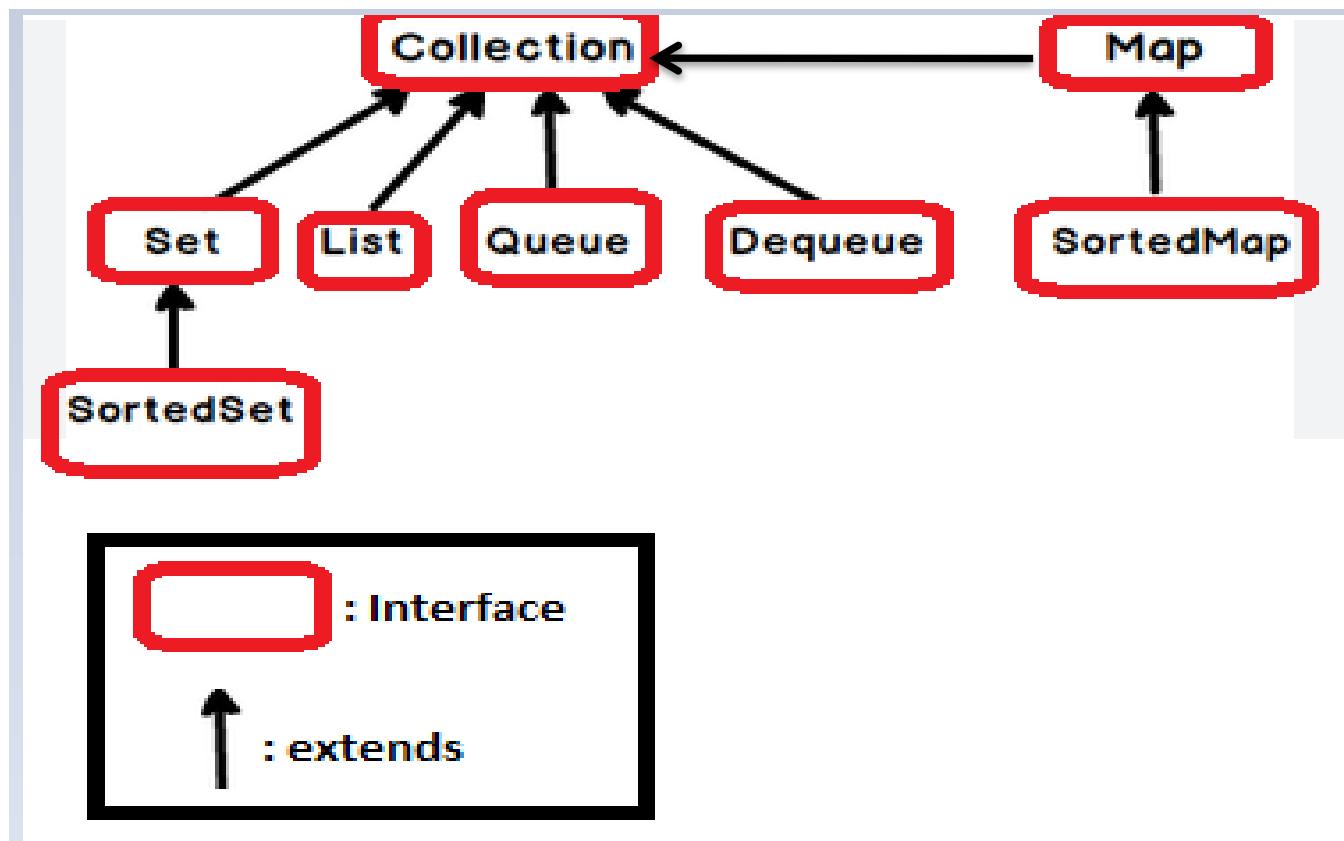
- On appelle *collections* un ensemble **de classes** et **d'interfaces** fournies par Java et disponibles pour la plupart dans le package **java.util**.
- Parmi ces collections, on trouve les listes (*lists*), les ensembles (*sets*) et les tableaux associatifs (*maps*). Elles forment ce que l'on appelle le **Java Collections Framework**.
- En effet, une collection est un objet servant de conteneur à d'autres objets. Exemples :
  - les tableaux,
  - les listes et leurs variantes (piles, queues),
  - les ensembles,
  - les tables associatives,

# INTRODUCTION

- Chaque genre de collection a ses caractéristiques propres, ses forces et ses faiblesses.
- en commun :
  - **mêmes questions** : est-ce qu'elles contiennent des éléments ? combien ?
  - **mêmes opérations** : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure.
  - **Mais**, avec des implémentations différentes.
- Le choix de la collection à utiliser dans un cas particulier dépend de ce qu'on veut en faire.

# HIÉRARCHIE D'INTERFACE

- Toutes ces structures sont organisés sous forme d'une hiérarchie d'interface.



# HIÉRARCHIE D'INTERFACE

- **Collection:** une interface qui contient des méthodes de base pour parcourir, ajouter, enlever des éléments.
- **Set :** cette interface représente un ensemble, et donc, ce type de collection n'admet aucun doublon.
- **List :** cette interface représente une séquence d'éléments : l'ordre d'ajout ou de retrait des éléments est important (doublons possibles).
- **Queue :** **file d'attente**, il y a l'élément en tête et les éléments qui suivent. L'ordre d'ajout ou de retrait des éléments est important (doublons possibles).
- **Double-endedqueue :** cette interface ressemble aux files d'attente, mais les éléments importants sont les éléments en tête et en queue.
- **SortedSet :** est la version ordonnée d'un ensemble.

# HIÉRARCHIE D'INTERFACE

- **Map** : cette interface représente une relation binaire (surjective) : chaque élément est associé à une clé et chaque clé est unique (mais on peut avoir des doublons pour les éléments).
- **SortedMap** : est la version ordonnée d'une relation binaire où les clés sont ordonnées.

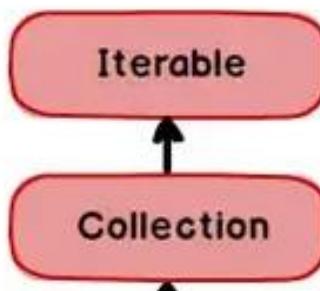
- .....

Ces interfaces sont **génériques**, i.e. on peut leur donner un paramètre pour indiquer qu'on a une collection d'Integer, de String, etc..

- .....

# HIÉRARCHIE D'INTERFACE

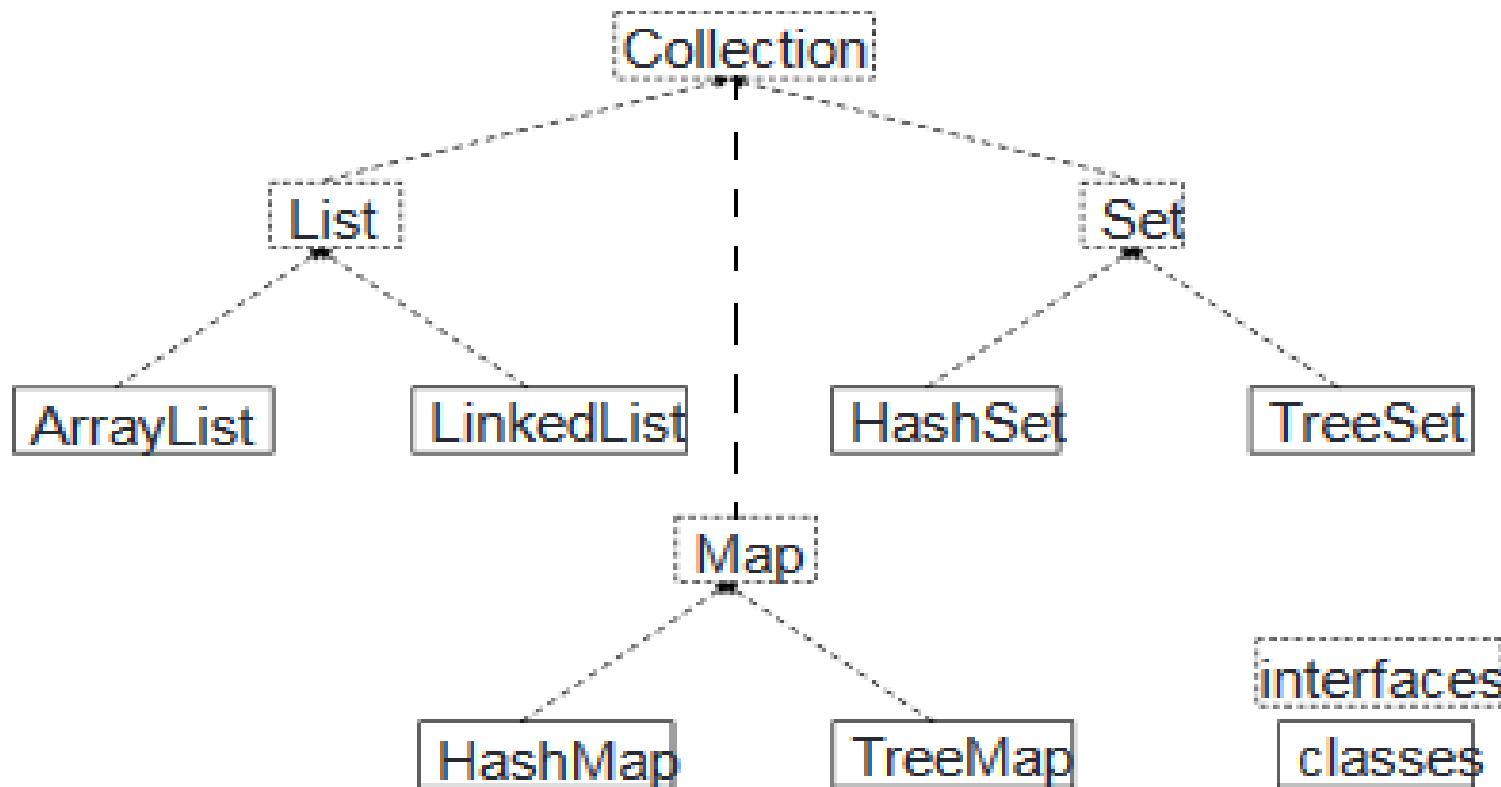
- L'interface **Iterable** (`java.lang.Iterable`) est l'une des interfaces racine de l'arborescence des collections. **Une classe** qui implémente l'interface **Iterable** peut être itérée avec la boucle `for-each`.



```
List list = new ArrayList();
list.add("Java"); list.add("Pascal");
list.add("PHP"); list.add("C++");
for(Object obj : list){
    System.out.println(obj.toString());
}
```

# IMPLÉMENTATION DES INTERFACES COLLECTION

- Pour chacune des interfaces, il existe plusieurs implémentations



# L'INTERFACE COLLECTION

- L'interface **Collection** définit la notion de collection d'objets d'une façon assez générale.

- **Les opérations sont :**

- Obtenir le nombre d'éléments de la collection,
- Rechercher un objet donné
- Ajouter un objet
- supprimer un objet,
- ... etc.

# L'INTERFACE MAP

- Les collections de type Map, tableau associatif ou dictionnaire en Java, sont définies à partir de la racine Interface Map <K, V>.
- La raison est qu'une telle collection est un ensemble de paires d'objets, chaque paire associant un objet de l'ensemble de départ K à un objet de l'ensemble d'arrivée V ; on parle de paires (clé, valeur)

..

# L'INTERFACE LIST

- Interface pour des objets qui autorisent des doublons et un accès direct à un élément.
- **Plusieurs implémentations possibles tq:**
  - **ArrayList** : Liste implantée dans un tableau.
  - **LinkedList**: Liste doublement chaînée.
- **Quelques méthodes de ArrayList:**

# L'INTERFACE LIST

- **add(Object element)** permet d'ajouter un élément ;
- **add(int index, Object element)** permet d'ajouter un élément à l'indice index ;
- **get(int index)** retourne l'élément à l'indice demandé.
- **remove(int index)** efface l'elt à l'indice demandé.
- **isEmpty()** renvoie « vrai » si l'objet est vide ;
- **size()** retourne la taille de l'ArrayList;
- **contains(Object element)** retourne « vrai » si l'élément passé en paramètre est dans l'ArrayList.

# L'OBJET ARRAYLIST

- Une ArrayList stocke ses éléments dans un tableau dynamique, donc il n'a pas de taille limite, et en plus, ils acceptent n'importe quel type de données ! **null** y compris !
- Vous devez par contre importer la classe **ArrayList**.

```
o package RepertoireDeTelephone;

o import java.util.ArrayList;
o import java.util.Iterator;
o import java.util.Map;

o public class MyArrayList {

o     public static void main(String[] args) {
o         // TODO Auto-generated method stub
o         ArrayList al = new ArrayList();
o         al.add("Bonjour");
o         al.add(5);
o         al.add('.');

o         // afficher ArrayList par une boucle for
o         System.out.println("Boucle for");
o         for (int i =0;i<al.size(); i++)
o             System.out.println(al.get(i));

o         // afficher ArrayList par une boucle for avancée
o         System.out.println("Boucle for avancée");
o         for (Object n : al)
o             System.out.println(n);

o         //afficher ArrayList par une boucle while+iterator
o         System.out.println("Boucle while & iterator");
o         Iterator it = al.iterator();
o         while (it.hasNext()){
o             System.out.println(it.next());
o         }
o     }
}
```

Problems @ Javadoc Declaration Console  
<terminated> MyArrayList [Java Application] C:\Program Files\Java\jre1.8.0\_251\bin\javaw.exe (3)

Boucle for  
Bonjour

5

.

Boucle for avancée  
Bonjour

5

.

Boucle while & iterator  
Bonjour

5

.

# L'OBJET LINKEDLIST

- Une LinkedList utilise une liste **doublement chaînée**.
- Une liste chaînée est une liste dont chaque élément est relié au suivant par une référence à ce dernier, sa taille n'est pas fixe : on peut ajouter et enlever des éléments selon nos besoins.
- Les LinkedList acceptent tout type d'objet.
- Chaque élément contient une référence sur l'élément suivant sauf pour le dernier : son suivant est en fait **null**.
- Vous devez importer la classe LinkedList .

# L'OBJET ARRAYLIST VS LINKEDLIST

- Puisque la recherche dans ArrayList est basée sur l'index de l'élément alors elle est très rapide. La méthode `get(index)` a une complexité de  $O(1)$ , mais la suppression est coûteuse parce que vous devez décaler tous les éléments. Dans le cas de LinkedList, elle ne possède pas un accès direct aux éléments, vous devez parcourir toute la liste pour récupérer un élément, sa complexité est égale à  $O(n)$ .

# L'OBJET ARRAYLIST VS LINKEDLIST

- Les insertions sont faciles dans LinkedList par rapport à ArrayList parce qu'il n'y a aucun risque lors de la redimensionnement et l'ajout de l'élément à LinkedList et sa complexité est égale à O(1), tandis que ArrayList décale tous les éléments avec une complexité de O(n) dans le pire des cas.
- La suppression est comme l'insertion, meilleure dans LikedList que dans ArrayList.
- LinkedList a plus de mémoire que ArrayList parce que dans ArrayList chaque index est relié avec l'objet actuel, mais dans le cas de LinkedList, chaque nœud est relié avec l'élément et l'adresse du nœud suivant et précédent.

## L'OBJET **ARRAYLIST** VS **LINKEDLIST**

- On utilise le plus souvent **ArrayList** si l'ajout et l'accès sont direct (indicé).
- Mais, **LinkedList** est utile s'il y a beaucoup d'opérations d'insertions / suppressions afin d'éviter les décalages.

..

# L'INTERFACE SET

- Eléments non dupliqués
- Plusieurs implémentations possibles tq:
  - **HashSet** : table de hashage (très utilisée).
  - **TreeSet** : arbre binaire de recherche.
- Quelques méthodes de HashSet:

..

# L'INTERFACE SET

- **add(Object element)** ajoute un élément.
- **contains(Object element)** retourne « vrai » si l'objet contient element.
- **isEmpty()** retourne « vrai » si l'objet est vide.
- **iterator()** renvoie un objet de type Iterator.
- **remove(Object element)** retire l'objet element de la collection ;
- **toArray()** retourne un tableau d'Object.

« «

# L'OBJET HASHSET

- Un Set est une collection qui n'accepte pas les doublons. Elle n'accepte qu'une seule fois la valeur **null**, car deux fois cette valeur est considérée comme un doublon.
- On peut dire que cet objet n'a que des éléments différents.
- Certains Set sont plus restrictifs que d'autres, n'acceptent pas null ou un certain type d'objet.
- On peut parcourir ce type de collection avec un objet **Iterator** où, cet objet peut retourner un tableau d'**Object**.

# Généricité

# GÉNÉRICITÉ

- **Exemple:**

```
public class CompteBancaire {  
    private String nomProp;  
    private double solde;  
    private char devise;  
}
```

## // Constructeur

```
public CompteBancaire(String nomProp, double solde, char devise){  
    this.nomProp= nomProp; this.solde=solde; this.devise=devise;  
}
```

# GÉNÉRICITÉ

## // Méthodes

```
public String getNomProp(){ return nomProp;}  
public double getSolde(){ return Solde;}  
public char getDevise(){ return devise;}
```

```
public void addSolde(double solde){ this.solde += solde; }  
public void removeSolde(double solde){ this.solde -= solde; }  
Public void showCompte(){  
System.out.println (" votre solde est "+solde+ " " + devise);  
}
```

..

25

# GÉNÉRICITÉ

```
Public class Main{  
    public static void main (String[] args){  
        CompteBancaire monCompte = new CompteBancaire("Douaa ", 100, '€');  
        CompteBancaire autreCompte = new CompteBancaire("Douaa", 100,  
            "DA");  
    }  
}
```

## ○ Attention

- Dans le deuxième objet j'aurai une erreur car le troisième paramètre demande un caractère et non pas une chaîne .
- Si je change le type de devise de char en String !!
- ça va pas fonctionné dans le premier objet car le troisième paramètre demande char au lieu String!!
- On se retrouve dans **une boucle infinie d'erreur !!**

# GÉNÉRICITÉ D'OBJET

**Solution:** on doit créer un **objet générique**.

- On lieu de se limiter à un seul type (char) pour l'attribut devise on le rendre générique.

```
public class CompteBancaire <T> { /* les types génériques sont  
représentés avec un T par convention */
```

```
    private String nomProp;  
    private double solde;  
    private T devise; }
```

```
public CompteBancaire(String nomProp, double solde, T devise){  
    this.nomProp= nomProp; this.solde=solde; this.devise=devise;  
}
```

```
public T getDevise(){ return devise;}
```

# GÉNÉRICITÉ D'OBJET

Maintenant on doit passer le type de T quand on créer des objets dans le main !!

- Par substitution de T on créer un système dynamique.

```
Public class Main{  
public static void main (String[] args){
```

```
CompteBancaire <Character> cpmt1=new CompteBancaire <>  
("Douaa", 100, '€');
```

```
CompteBancaire <String> cmpt2=new CompteBancaire <> ("Douaa",  
100, 'DA');
```

```
}
```

## Généricité et héritage

# GÉNÉRICITÉ ET HÉRITAGE

Soit la classe paramétrée (générique) Paire suivante:

```
public class Paire <T> { // classe générique  
    T premier ;  
    T second ;  
    public Paire (T a, T b){ premier=a; second = b; }  
    public T getPremier (){ return premier ; }  
    public T getSecond (){ return second ; }  
}
```

# GÉNÉRICITÉ ET HÉRITAGE

Soit la classe exécutable TableauG:

```
class TableauG { ...  
public static Paire<T> getDeux(T[ ] tab,int i, int j){  
    return new Paire<T> (tab[i],tab[j]);  
}  
...  
} // fin classe TableauG
```

# GÉNÉRICITÉ ET HÉRITAGE

Et soit la classe Personne suivante:

```
class Personne {  
    private String nom, prénom;  
    public Personne(String nom, String prénom) {  
        this.nom=nom; this.prénom=prénom; }  
    public String getNom() { return nom; }  
    public String getPrénom() { return prénom; }  
  
    // méthode .....  
}
```

# GÉNÉRICITÉ ET HÉRITAGE

Et soit la classe fille **Elève** de Personne:

```
class Elève extends Personne {  
    private double[] notes = new double[10];  
    private int nombreNote = 0;  
    public Elève(String nom, String prénom) {  
        super(nom, prénom); }  
    ...  
}
```

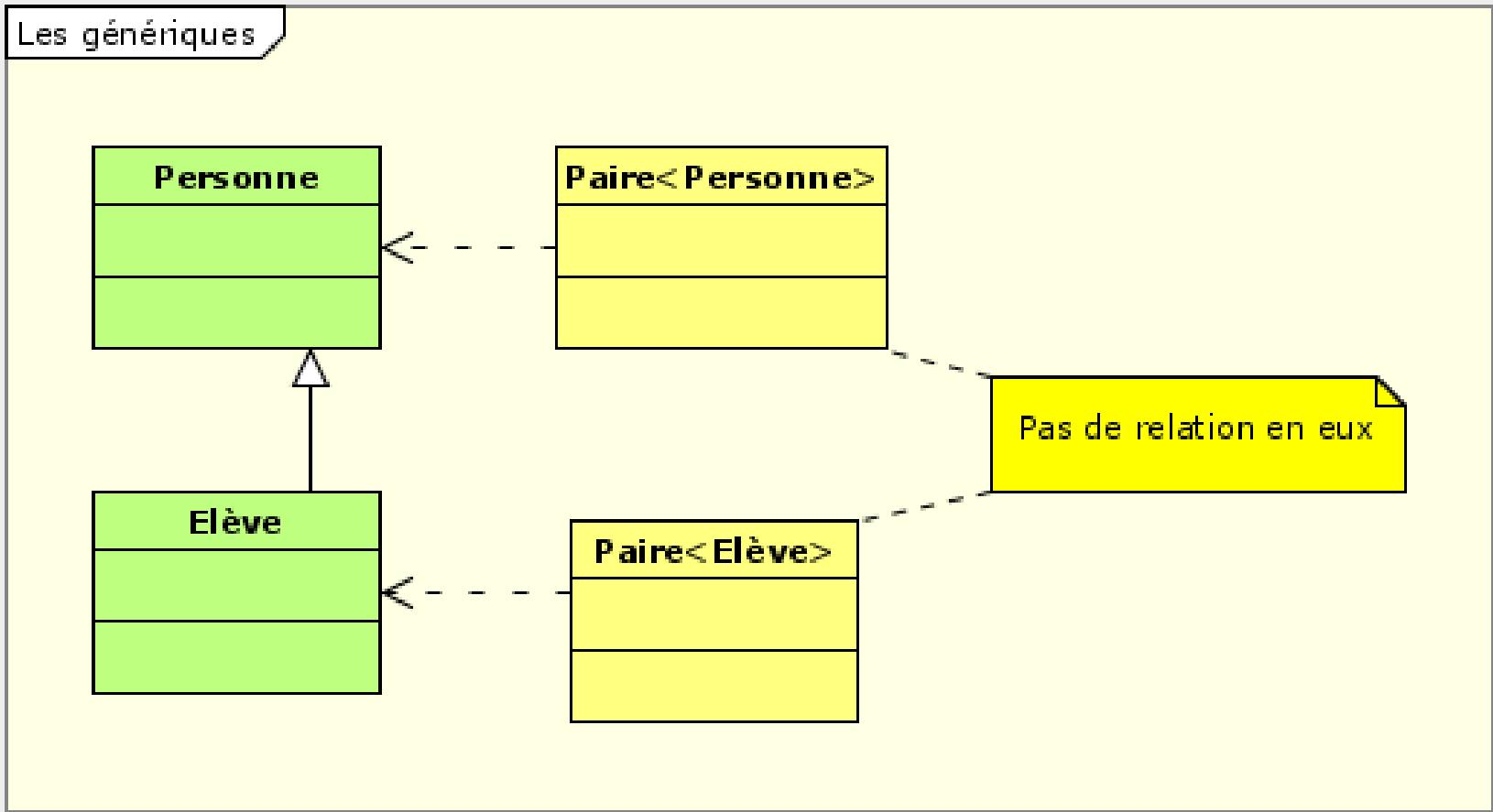
..

## GÉNÉRICITÉ ET HÉRITAGE

- La question qui se pose:

"Est-ce que **Paire<Elève>** est une **sous-classe de Paire<Personne>**" ?.

# GÉNÉRICITÉ ET HÉRITAGE



## GÉNÉRICITÉ ET HÉRITAGE

- La réponse est « **Non** »

Par exemple, **le code suivant ne sera pas compilé :**

Elève[] élèves = ... ;

Paire <Personne>personne =

TableauG.getdeux(élèves,1,2);

## GÉNÉRICITÉ ET HÉRITAGE

En général, il n'existe pas de relation entre `Paire<S>` et `Paire<T>`,

quels que soient les éléments auxquels `S` et `T` sont reliés  
(même s'il y a de l'héritage entre `S` et `T`)

# Généricité et interface

# GÉNÉRICITÉ ET INTERFACE

- Exemple d'une interface de pile d'entiers qui est dynamique mais elle est limité pour représenter que des entiers.
- **NB:** une pile sert à stocker des valeurs de même type.

```
public interface IntStack {
```

```
// Ajoute l'élément au sommet de la pile.
```

```
public void addFirst(int value);
```

```
// déplier et empiler l'élément au sommet de la pile.
```

```
// Lève IllegalStateException si la pile est vide.
```

```
public int removeFirst();
```

```
// Retourne vrai ssi la pile est vide.
```

```
public boolean isEmpty();
```

```
}
```

..

# GÉNÉRICITÉ ET INTERFACE

- les classes qui implémentent l'interface IntStack ne peuvent représenter que des piles d'entiers.
- **Comment faire pour représenter des piles d'un autre type ????**
- **Solution:** écrire autant d'interfaces et de classes qu'il existe de types (spécialisation).

..

# GÉNÉRICITÉ ET INTERFACE

- **interface IntStack {**  
void addFirst(int value); ... }
- **interface BooleanStack {**  
void addFirst(boolean value); ... }
- **interface StringStack {**  
void addFirst(String value); ... }

**duplication de code !**

..

# GÉNÉRICITÉ ET INTERFACE

**Solution 2:** faire des piles de Object

- **interface Stack {**

- public void addFirst(Object value);

- public Object removeFirst();

- public boolean isEmpty();

- }

- possible mais nécessitera des grande quantité de transtypages (*casts*) explicites !!

# GÉNÉRICITÉ ET INTERFACE

- En raison des problèmes posés par la spécialisation et la solution basée sur le type Object, la notion de **généricité** (genericity), aussi appelée **polymorphisme paramétrique** (parametric polymorphism) a été introduite dans la version 5 de Java.
- Au moyen de la généricité, il est possible de définir des piles génériques, c'est-à-dire capables de contenir des éléments de différents type.

»

# GÉNÉRICITÉ ET INTERFACE

- Une interface peut se définir ainsi :
- **interface Stack<E> {**  
    public void addFirst(**E** value);  
    public **E** removeFirst();  
    public boolean isEmpty();  
}
- E est un **paramètre de type** de cette interface.  
Il s'agit d'une variable (de type !) représentant le type des éléments de la pile.

66

# GÉNÉRICITÉ ET INTERFACE

Cas d'utilisation:

- **public class LinkedStack <E> implements Stack <E> { ... }**
- **Stack <Character> s = new LinkedStack <Character>();**

# TYPES DE BASE

- Java possède 8 types dits *de base qui ne sont pas* des objets (boolean, byte, short, int, long, char, float, double). Malheureusement, les types de base ne peuvent pas être utilisés comme paramètres de type d'un type générique. Dès lors, le code suivant est erroné :
- Stack<int> s = ...; // interdit !
- s.add(2);
- s.add(3);
- Que faire si l'on désire créer une pile d'entiers ?

# EMBALLAGE

- **Solution** : stocker chaque valeur de type int dans un objet de type **java.lang.Integer**, et créer une pile de valeurs de ce type. L'exemple devient :
- Stack<Integer> s = ...;  
    s.add(**new Integer(2)**);  
    s.add(**new Integer(3)**);
- On dit alors que les entiers ont été **emballés** (*wrapped ou boxed*) *dans des objets de type Integer*.
- Le paquetage **java.lang** contient une class d'emballage par type de base (Boolean pour boolean, Character pour char, Double pour double, etc.)

..

## L'OBJET HASHMAP: EXERCICE

- on utilise **HashMap** pour simuler un répertoire dans lequel le numéro de téléphone est la clé et le nom du propriétaire est la valeur. Les clés ne sont jamais dupliquées.
- 2eme version en TP: maintenant on vous demande d'insérer deux clés identiques. Expliquez le résultat.

..

```

○ package RepertoireDeTelephone;

○ import java.util.HashMap;
○ import java.util.Iterator;
○ import java.util.Map;

○ public class Main {

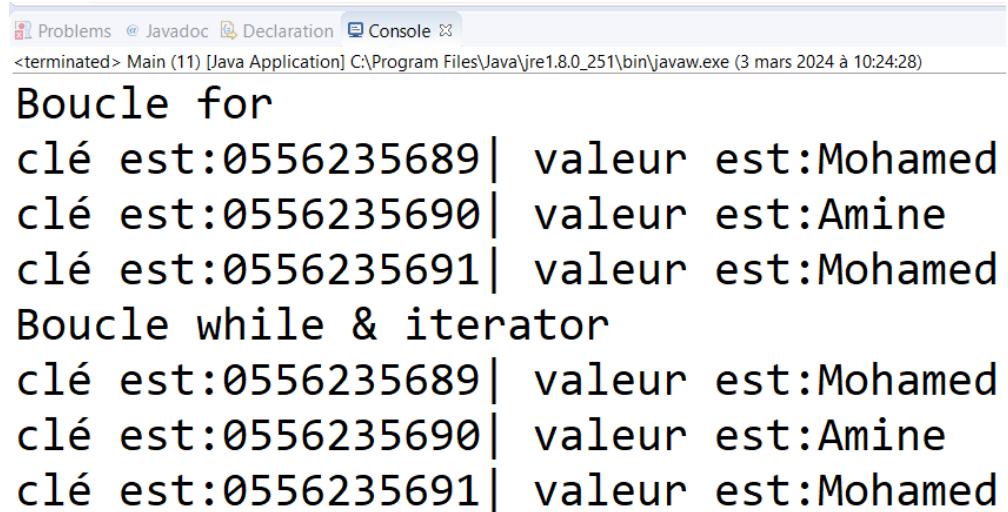
○ public static void main(String[] args) {
○ // TODO Auto-generated method stub
○ //Créer et remplir HashMap

○ Map <String, String> repPhonne = new HashMap<>();
○ repPhonne.put("0556235689", "Mohamed");
○ repPhonne.put("0556235690", "Amine");
○ repPhonne.put("0556235691", "Mohamed");

○
○ // afficher HashMap par une boucle for
○ System.out.println("Boucle for");
○ for (Map.Entry mp: repPhonne.entrySet())
○ System.out.println("clé est:" +mp.getKey()+" | valeur est:"+mp.getValue());

○
○ //afficher HashMap par une boucle while+iterator
○ System.out.println("Boucle while & iterator");
○ Iterator it = repPhonne.entrySet().iterator();
○ while (it.hasNext()){
○ Map.Entry mp =(Map.Entry)it.next();
○ System.out.println("clé est:" +mp.getKey()+" | valeur est:"+mp.getValue());
○ }}}

```



Boucle for

clé est:0556235689 | valeur est:Mohamed  
 clé est:0556235690 | valeur est:Amine  
 clé est:0556235691 | valeur est:Mohamed

Boucle while & iterator

clé est:0556235689 | valeur est:Mohamed  
 clé est:0556235690 | valeur est:Amine  
 clé est:0556235691 | valeur est:Mohamed

# L'OBJET HASHSET

```
10 public static void main(String[] args) {  
11     HashSet<String> hset = new HashSet<String>();  
12     hset.add("h1");  
13     hset.add("h2");  
14     hset.add("h3");  
15  
16     System.out.println("Boucle for avancée");  
17     for(String s : hset)  
18         System.out.println(s);  
19  
20     System.out.println("Boucle While+Iterator");  
21     Iterator it = hset.iterator();  
22     while(it.hasNext())  
23         System.out.println(it.next());  
24  
25     System.out.println("Boucle While+Ennumération");  
26     // récupérer l'objet Ennumeration  
27     Enumeration enumeration = Collections.enumeration(hset);  
28     // lire à travers les éléments de HashSet  
29     while(enumeration.hasMoreElements())  
30         System.out.println(enumeration.nextElement());}}
```

# L'OBJET HASHSET

Boucle for avancée

h1

h2

h3

Boucle While+Iterator

h1

h2

h3

Boucle While+Ennumération

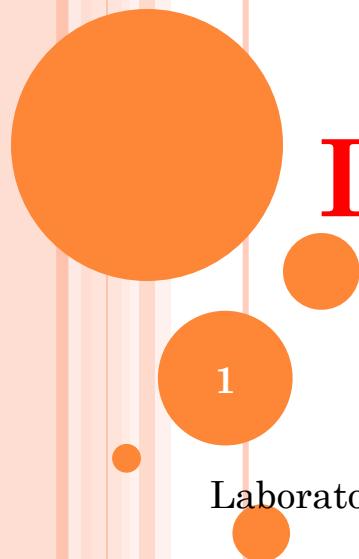
h1

h2

h3

66

Cours  
**Programmation Orientée Objet 2**  
Pour  
**ING 2**



**Chap 04:**  
**Interfaces Graphiques**

MEKAHLIA Fatma Zohra LAKRID  
Maître de Conférences Classe B

Laboratoire de Modélisation, Vérification et Evaluation des Performances des systèmes  
complexes (MOVEP)  
Bureau 123

# PLAN

- Généralités sur les interfaces graphiques.
- Composants des interfaces graphiques.
- Les packages AWT et Swing.
- Classes de base.
- Création et affichage d'une fenêtre.
- Placer des composants dans une fenêtre .
- Création de Jar exécutable.
- Gestion des événements.
- Le modèle MVC.

..

# GÉNÉRALITÉS SUR LES INTERFACES GRAPHIQUES

- Généralement les programmes informatiques nécessitent:
  - l'affichage de questions posées à l'utilisateur,
  - l'entrée de données par l'utilisateur au moment de l'exécution,
  - l'affichage d'une partie des résultats obtenus par le traitement informatique.
- Cet échange d'informations peut s'effectuer avec une interface utilisateur (**User Interface**) :
  - en mode texte,
  - ou console,
  - ou encore en **mode graphique**.

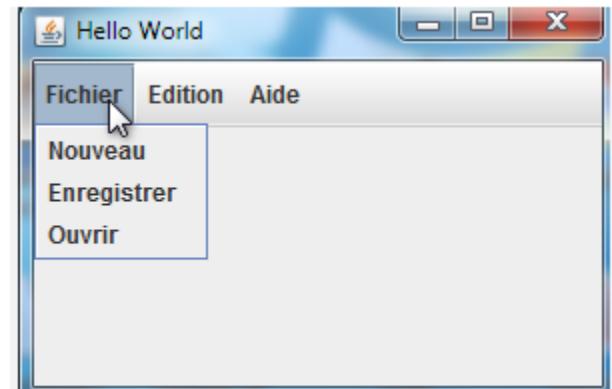
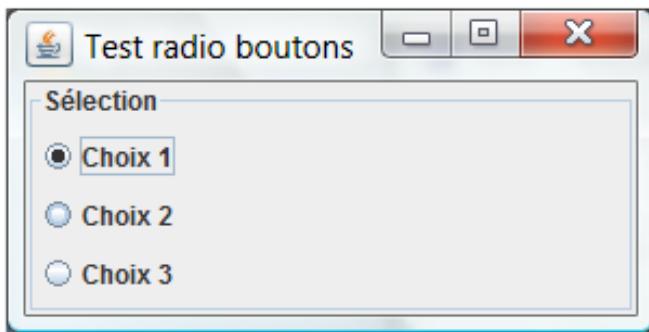
# GÉNÉRALITÉS SUR LES INTERFACES GRAPHIQUES

- Une interface graphique est souvent appelé **GUI** (Graphical User Interface), est un ensemble de **composants graphiques permettant à un utilisateur de communiquer avec un logiciel.**

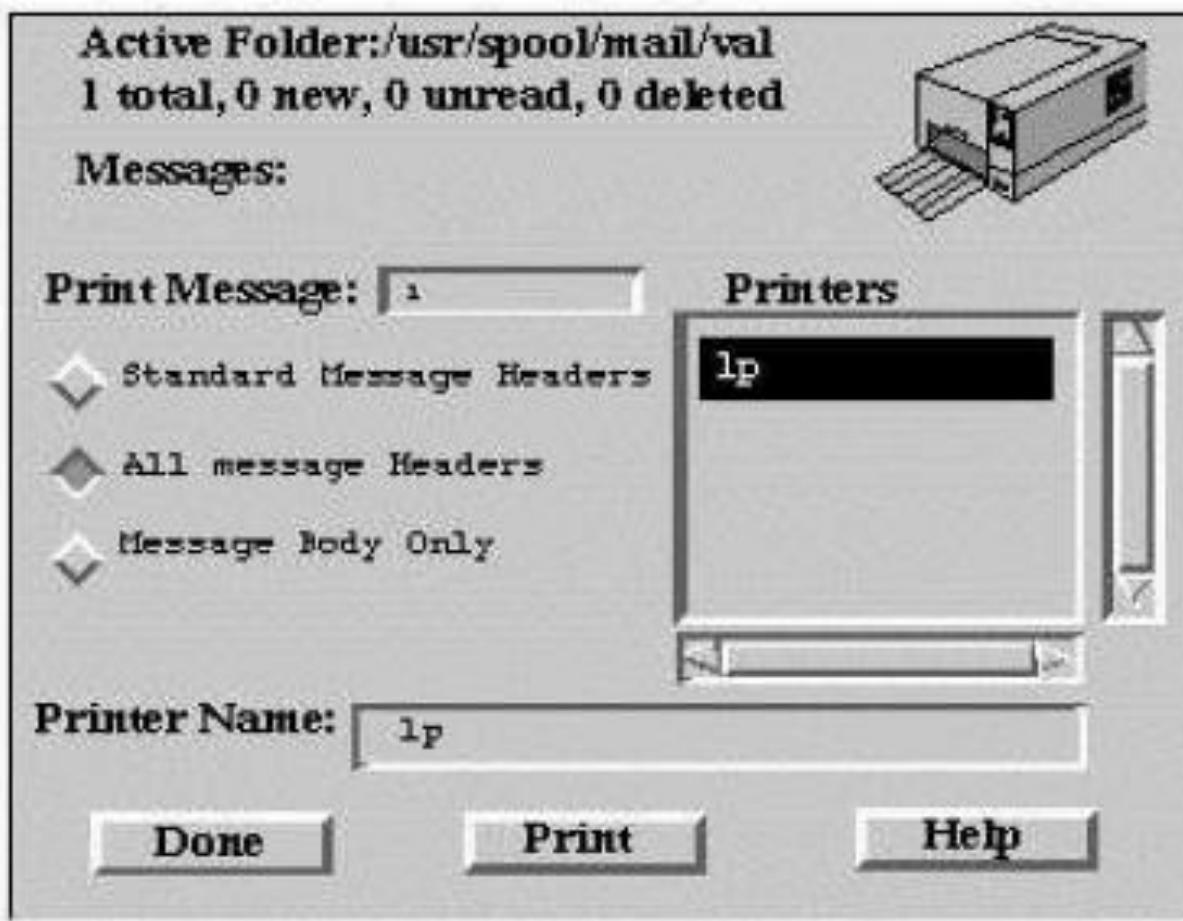
The screenshot shows a window titled "Gestion des salles ENSAJ". Inside the window, there are several input fields and buttons:

- A radio button group for "Salle de cours" (selected) and "Salle de réunion".
- A text input field for "Nom".
- A text input field for "Créneau".
- A text input field for "Date".
- Text labels: "Numéro de la salle", "Numéro de l'étage", "Capacité", and "Type".
- A stack of three empty text input fields next to "Numéro de la salle".
- A stack of two empty text input fields next to "Numéro de l'étage".
- A single empty text input field next to "Capacité".
- A radio button group for "Amphi" and "Salle".
- A large "Valider" button at the bottom right.

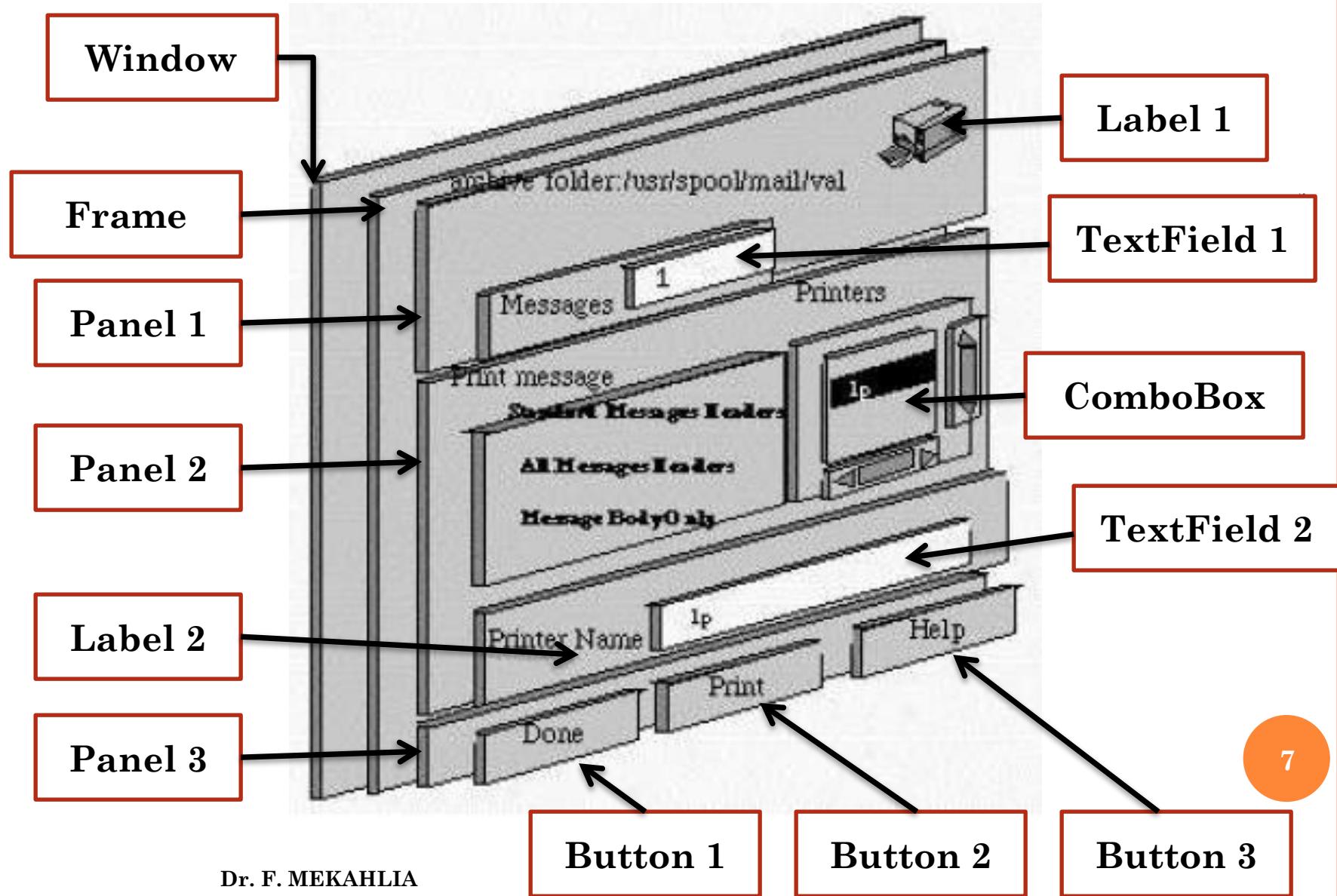
# GÉNÉRALITÉS SUR LES INTERFACES GRAPHIQUES



# COMPOSANTS DES INTERFACES GRAPHIQUES



# COMPOSANTS DES INTERFACES GRAPHIQUES



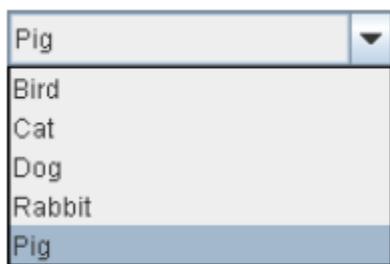
# COMPOSANTS DES INTERFACES GRAPHIQUES



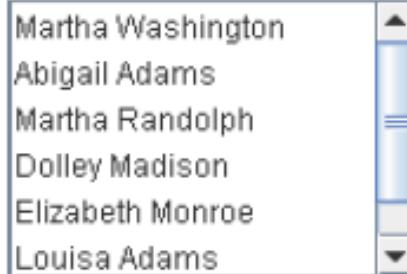
ProgressBar



Slider



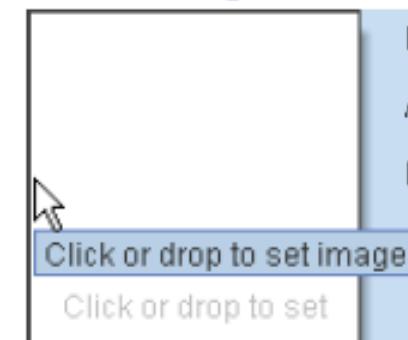
ComboBox



List

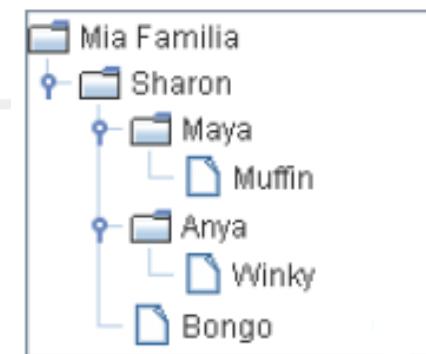


Dialog



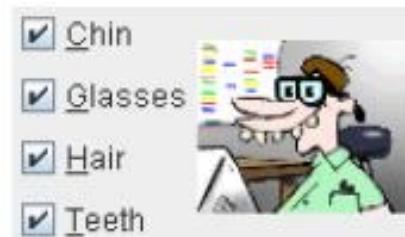
ToolTip

Tree



TextArea

This is an editable JTextArea. A text area is a "plain" text component, which means that although it can display text in any font, all of the text is in the same font.



8

More component in:

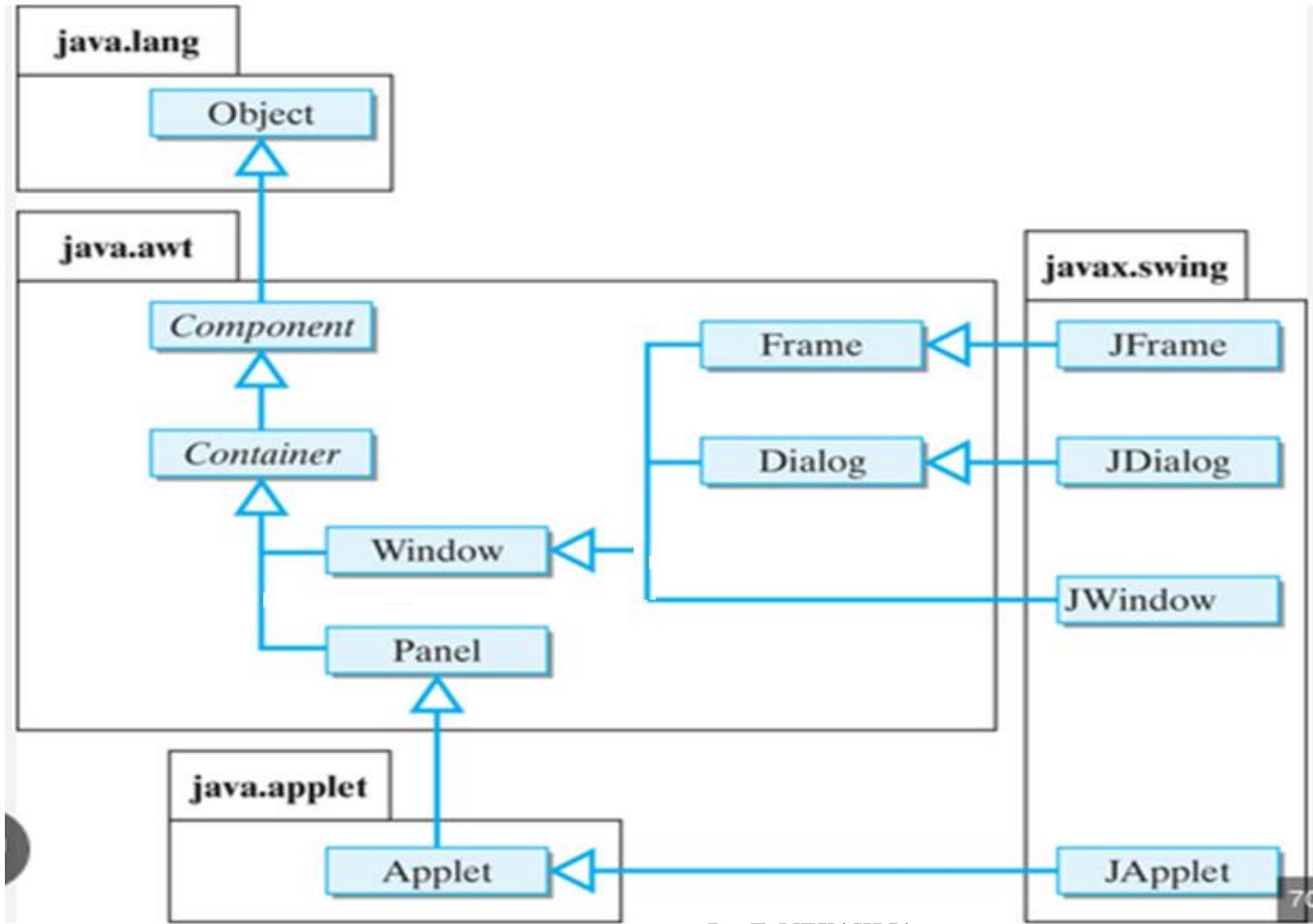
<https://docs.oracle.com/javase/8/docs/api/java.awt/Component.html>

CheckBox

# LES PACKAGES AWT ET SWING

- Généralement, les éléments graphiques sont définis dans deux grandes familles de composant graphique:
  - **AWT (abstract window toolkit, JDK 1.1).**
  - **Swing (JDK 1.2).**
- **Swing** est construit au-dessus de **AWT**:
  - même gestion des événements.
  - les classes de Swing héritent des classes de AWT.

# LES PACKAGES AWT ET SWING



# LES PACKAGES AWT ET SWING

- Swing et AWT font partie de JFC (Java Foundation Classes) qui offre des facilités pour construire des interfaces graphiques.
- Vous pouvez identifier les composants de swing à partir de leur nom qui débute par la lettre **J** (JDialog, JFrame etc.). Ces composants se trouvent dans le paquetage: **javax.swing**
  - **x** vient du mot "eXtension".
- Par contre les composants awt se trouvent dans le package **java.awt**

# Les packages AWT

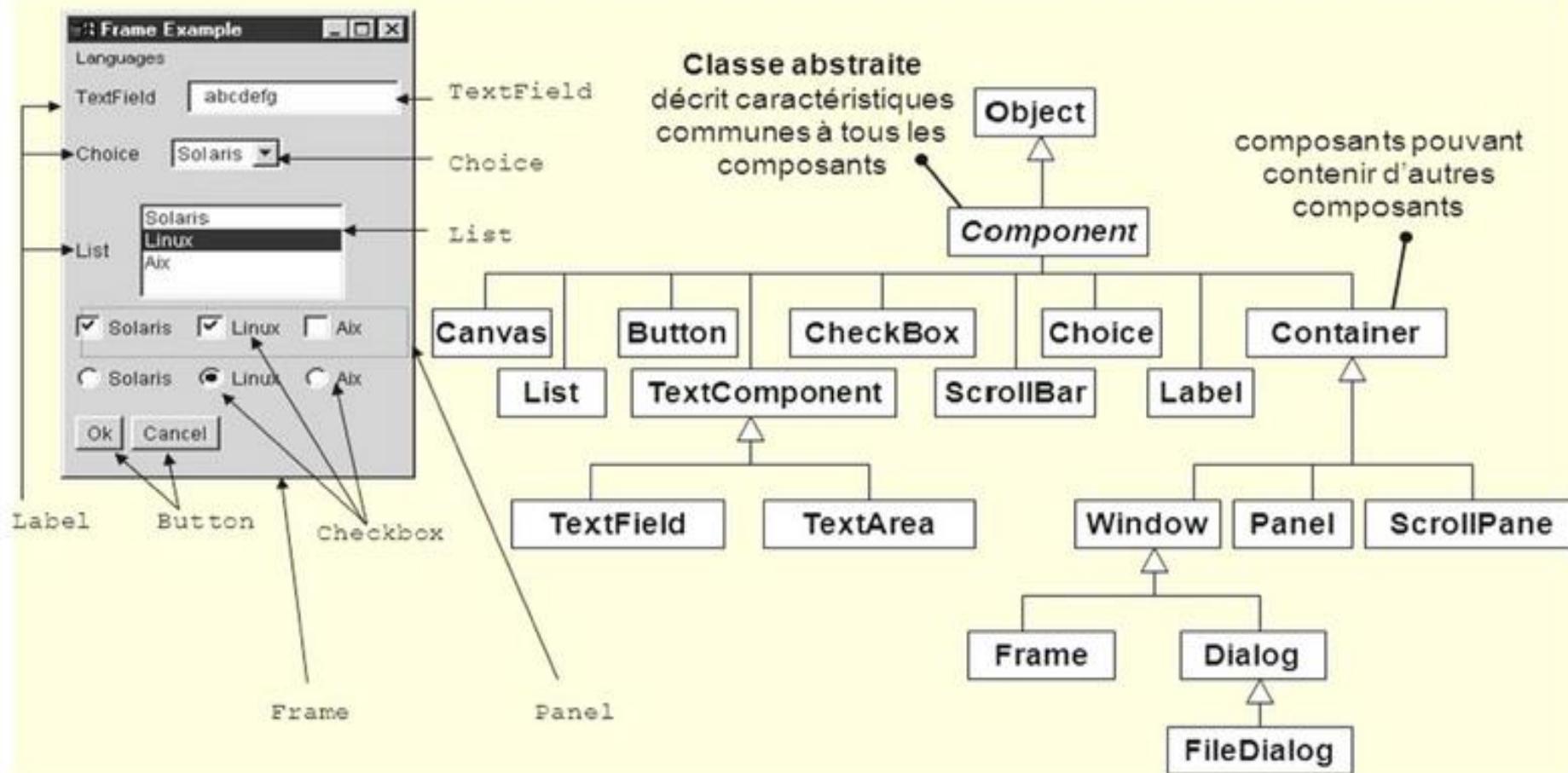
# LES PACKAGES AWT

- Les classes du toolkit AWT permettent d'écrire des interfaces graphiques indépendantes du système d'exploitation sur lesquelles elles vont fonctionner.
- Cette librairie utilise le système graphique de la plateforme d'exécution (Windows, MacOS ou autre) pour afficher les objets graphiques.
- Le toolkit contient des classes décrivant les composants graphiques, les polices, les couleurs et les images.

# LES PACKAGES AWT

- L'apparence des fenêtres et boutons diffère d'un système d'exploitation à l'autre, car chaque composant AWT est dessiné et contrôlé par un composant **tiers natif** spécifique au système d'exploitation.

# LES PACKAGES AWT



# Les packages SWING

# LES PACKAGES SWING

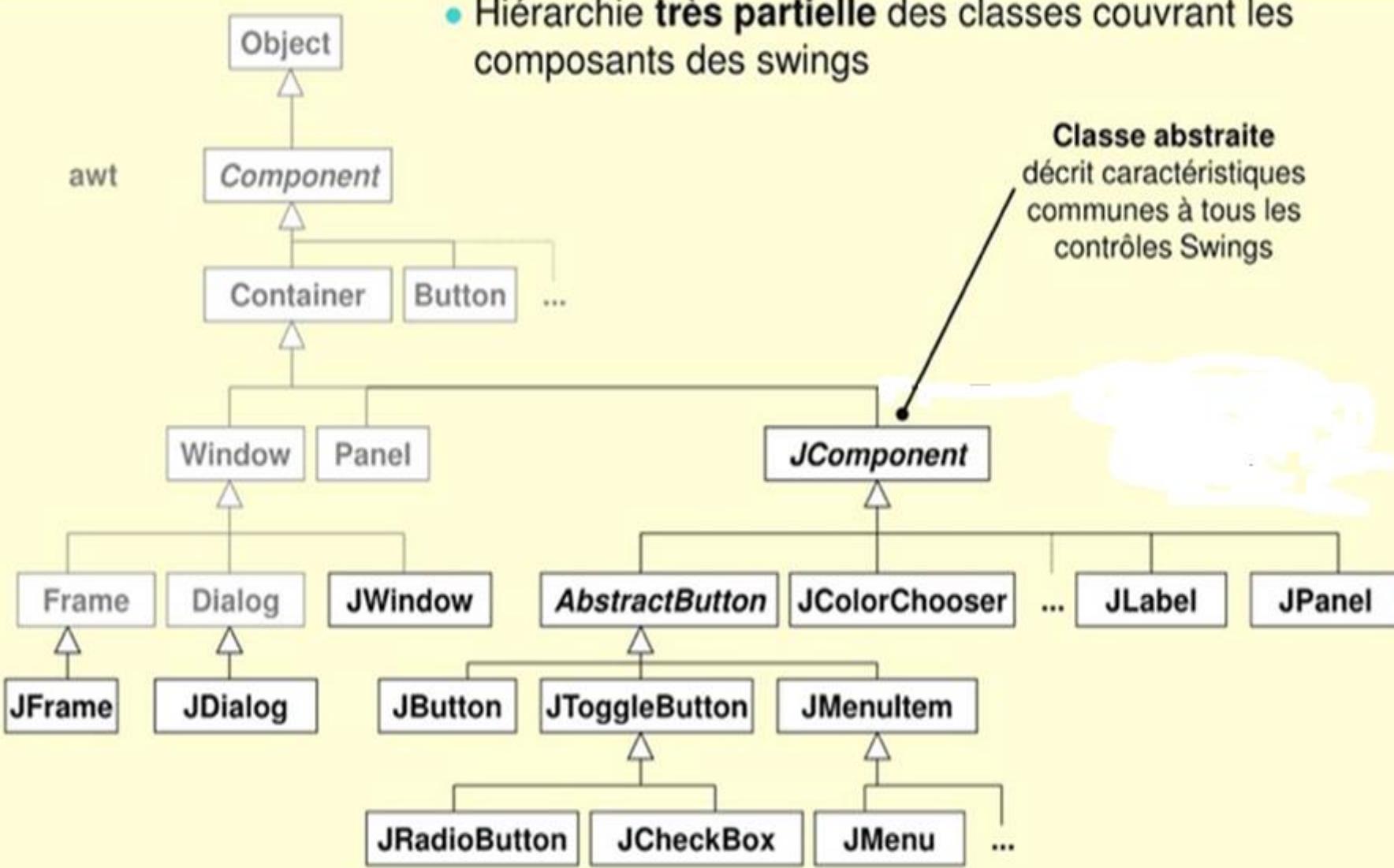
- Il existe un autre package d'interfaces graphiques et qui hérite de AWT appelée **Swing** qui est entièrement autonome, qui ne dépend pas du système d'exploitation.
- SWING ne requièrent pas d'allocation de ressources natives de la part du système d'exploitation, mais utilise les ressources de leurs ancêtres.

# LES PACKAGES SWING

- Swing est une API dont le but est similaire à celle de l'AWT mais dont le mode de fonctionnement et d'utilisation est totalement différent.
  
- Swing a été intégrée au JDK depuis sa version 1.2. Cette bibliothèque existe séparément pour le JDK 1.1.
  
- La plupart des programmes utilisant Swing nécessite aussi l'importation de deux importants paquetages de AWT :\*
  1. **import java.awt.\*;**
  2. **import java.awt.event.\*;**

# LES PACKAGES SWING

- Hiérarchie très partielle des classes couvrant les composants des swings



# LES PACKAGES SWING

- Les composants Swing forment une nouvelle hiérarchie parallèle à celle de l'AWT.
- Presque tous ces composants sont écrits en pure Java : ils ne possèdent **aucune partie native sauf** ceux qui assurent l'interface avec le système d'exploitation : JApplet, JDialog, JFrame, et JWindow.

# AWT vs SWING

# AWT vs SWING

- AWT utilise des composants natifs du système alors que Swing utilise des composants dessinés par Java.
- AWT est Lourd. Alors que les composants SWING sont généralement légers car ils n'ont pas besoin d'objets OS natifs pour implémenter leurs fonctionnalités.

..

# AWT vs SWING

- Les composants AWT sont moins nombreux que les composants Swing. Alors que , les composants Java Swing sont bien plus nombreux.
- Les composants AWT ne suivent pas l'architecture MVC (Model View Controller). Tandis que, les composants Swing suivent le modèle MVC (Model View Controller).
- AWT signifier Abstract Windows Toolkit. Alors que, les composants Swing en Java sont également appelés JFC(Java Foundation Classes).

..

# AWT vs SWING

- Les composants AWT ont besoin du package **java.awt**. Alors que, les composants Swing ont besoin du package **javax.swing**.
- Les composants AWT nécessitent et occupent un espace mémoire plus important. Tandi que, Les composants Swing n'occupent pas autant d'espace mémoire que les composants AWT.

..

## POUR INFORMATION ...

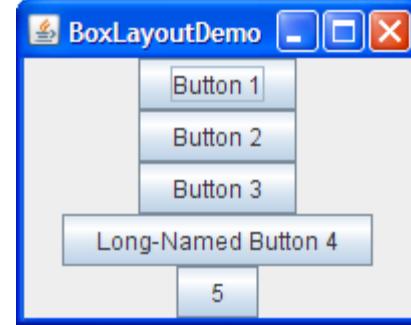
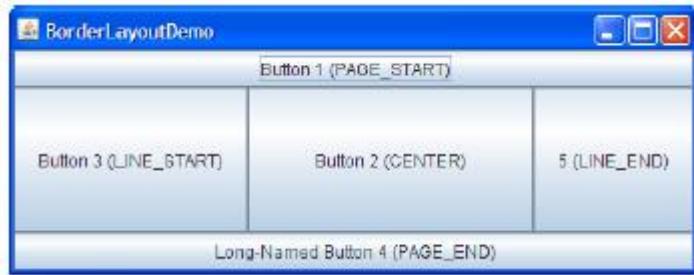
- il existe deux autres librairies de mise en oeuvre d'IHM:
  1. Avec l'apparition de Java 8 **JavaFX** est désormais l'API d'interface graphique principale du Java. Dans sa conception, elle est plus moderne que Swing et permet de produire des interfaces graphiques pouvant facilement être utilisées sur différents types d'écrans (écran standard de PC, smartphone & tablettes et applications Web).

# Architecture d'une application Swing

66

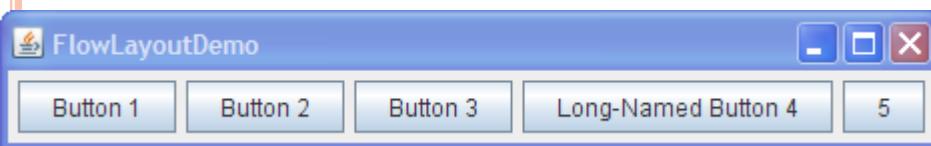
# ARCHITECTURE D'UNE APPLICATION SWING

- Une application est un ensemble de fenêtres qui contient des items bien placées.
- Un conteneur (container) en top-level c'est le composant racine par exemple **JFrame**.
- JFrame contient d'autres composants qui peuvent être :
  - Composants atomiques (simples), par ex: un **bouton**.
  - Des composants intermédiaires qui permettent de diviser la fenêtre comme le **JPanel** (des panneaux).
- Le placement des composants dans un conteneur correspond à une stratégie de placement, par exemple soit délégué à un *LayoutManager* (qui est interface).



BorderLayout : présentation avec bordures.

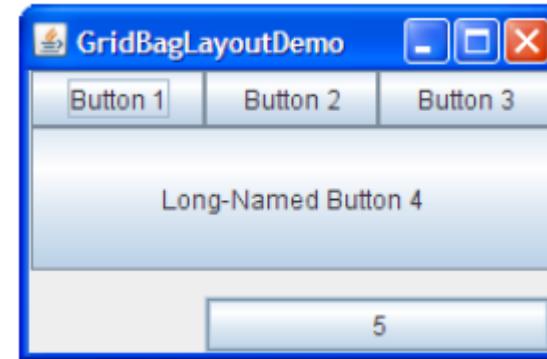
BoxLayout : en ligne ou  
en colonne



FlowLayout : présentation en file



GridLayout : en grille

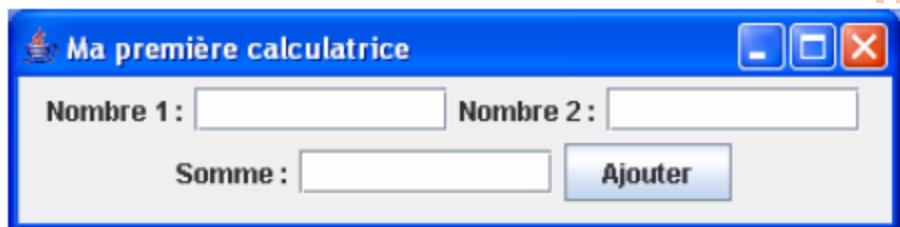


GridBagLayout : en grille composite qui  
est plus sophistiqué.

# Cas d'études

# CAS D'ÉTUDES

- Nous voulons créer une calculatrice simple capable d'ajouter deux nombres et d'afficher le résultat.



**JFrame:** Ma première calculatrice

**JPanel:** contenuFenêtre

**JLabel:**  
Nombre 1

**JLabel:**  
Nombre 2

**JLabel:**  
Somme

**JButton:**  
lancer

**JTextField**  
Id :  
entrée1

**JTextField**  
Id :  
entrée2

**JTextField**  
Id :  
résultat

```
○ import javax.swing.*;  
○ import java.awt.FlowLayout;  
○ public class CalculatriceSimple {  
○ public static void main(String[] args) {  
// Crée la fenetre  
○ JFrame fenetre= new JFrame("Ma première calculatrice");  
// Créer un panneau  
○ JPanel contenuFenêtre = new JPanel();  
// Affecter un gestionnaire de disposition à ce panneau  
○ FlowLayout disposition = new FlowLayout();  
○ contenuFenêtre.setLayout(disposition);  
// Créer les contrôles en mémoire  
○ JLabel label1 = new JLabel("Nombre 1 :");  
○ JTextField entrée1 = new JTextField(10);  
○ JLabel label2 = new JLabel("Nombre 2 :");  
○ JTextField entrée2 = new JTextField(10);  
○ JLabel label3 = new JLabel("Somme :");  
○ JTextField résultat = new JTextField(10);  
○ JButton lancer = new JButton("Ajouter");
```

..

// Ajoute les contrôles au panneau

- contenuFenêtre.add(label1);
- contenuFenêtre.add(entrée1);
- contenuFenêtre.add(label2);
- contenuFenêtre.add(entrée2);
- contenuFenêtre.add(label3);
- contenuFenêtre.add(résultat);
- contenuFenêtre.add(lancer);

// ajouter le panneau dans la fenetre

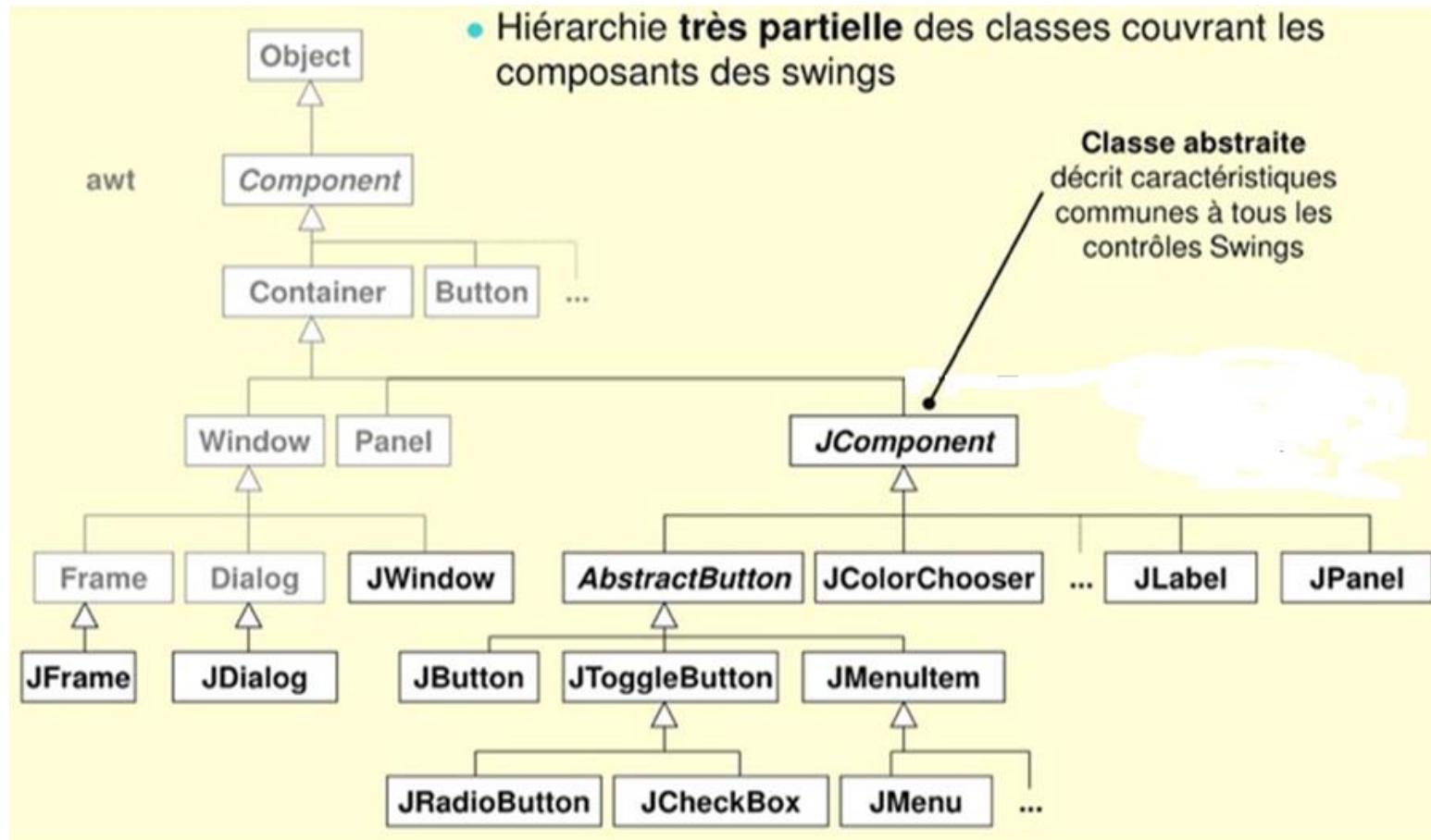
- fenetre.setContentPane(contenuFenêtre);

// Positionner les dimensions et rend la fenêtre visible

- fenetre.setSize(400,100);
- fenetre.setVisible(true);} }

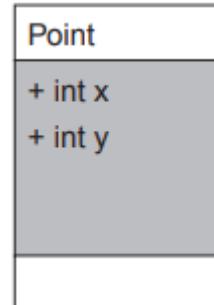
# Classes de base

# CLASSES DE BASE



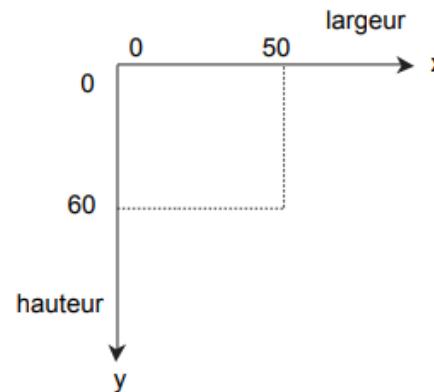
# CLASSES DE BASE

- Pour pouvoir comprendre les fonctionnalités des classes de base, nous allons commencer par la présentation des classes Point, Dimension, Rectangle, Color et Graphics qui sont souvent utilisées dans la création des interfaces graphiques.



# LA CLASSE POINT

- **Point:**
- La classe Point ([java.awt.Point](#)) définit un point repéré par ses attributs entiers x et y sur un plan ou à partir d'un autre Point.



- Elle implémente l'interface:

[java.io.Serializable](#)

- Elle hérite les packages:

[java.lang.Object](#)

[java.awt.geom.Point2D](#)

· Le système des coordonnées graphiques : l'axe des y est vers le bas.

Dimension
+ int width
+ int height

# LA CLASSE DIMENSION

## Dimension:

- La classe Dimension (**java.awt.Dimension**) définit deux attributs entiers représentant les dimensions d'un objet en largeur et en hauteur (width et height).
- Un objet Dimension peut aussi être construit à partir d'un autre objet Dimension, ou à partir de deux entiers.
- **Elle implémente l'interface:**

java.io.Serializable

- **Elle hérite les packages:**

java.lang.Object

java.awt.geom.Point2D

Rectangle
+ int x
+ int y
+ int width
+ int height

# LA CLASSE RECTANGLE

## Rectangle:

- La classe Rectangle définit un rectangle sur le plan caractérisé par le point **p** repérant son coin supérieur gauche (CSG),
- et par sa dimension **d** (largeur et hauteur).
- Rectangle a 4 attributs public : x et y, coordonnées du point, et width et height, valeurs de d.
- **Elle implémente les interfaces:**

java.awt.Shape

java.io.Serializable

- **Elle hérite les packages :**

java.lang.Object

java.awt.geom.RectangularShape

java.awt.geom.Rectangle2D

Color
+ static final Color black
+ static final Color blue
+ etc.

# LA CLASSE COLOR

## Color:

- La classe Color permet d'accéder à des constantes static indiquant 13 couleurs prédéfinies :
- Color.black Color.blue Color.cyan Color.darkGray  
Color.gray Color.green Color.lightGray Color.magenta  
Color.orange Color.pink Color.red Color.white  
Color.yellow
- Une couleur est créée à partir de ses trois composantes (RVB : rouge, vert, bleu), chaque composante ayant une valeur sur 8 bits (de 0 à 255).
- On peut créer une nouvelle couleur en indiquant ses 3 composantes. Exemple : new Color (100, 100, 100) ; crée un objet référençant une nouvelle couleur.

# LA CLASSE COLOR

## Color:

- Elle implémente les interfaces:

java.io.Serializable

Java.awt.Paint

- Elle hérite les packages:

java.lang.Object

Color
+ static final Color black
+ static final Color blue
+ etc.

..

# LA CLASSE GRAPHICS

## Graphics:

- La classe Graphics est une classe abstraite encapsulant (contenant) des informations permettant de dessiner des formes géométriques ou du texte pour un composant.
- Chaque composant a un contexte graphique.
- Les principales méthodes sont :
  - `. drawImage (Image, int, int, Color, ImageObserver)` : trace une image.
  - `drawLine (int, int, int, int)` : trace une ligne droite entre deux points.
  - `drawRect (int, int, int, int)` : trace un rectangle.
  - `drawOval (int, int, int, int)` : trace un ovale (ellipse) dans le rectangle défini.
  - `drawArc (int, int, int, int, int, int)` : trace un arc (une partie d'ellipse)

# LA CLASSE GRAPHICS

- `drawString (String, int, int)` : trace une chaîne de caractères.
- `fillRec (int, int, int, int)` : remplit un rectangle.
- `getColor ()` : fournit la couleur courante.
- `getFont ()` : fournit la fonte courante.
- `setColor (Color)` : change la couleur courante.
- `setFont (Font)` : change la fonte courante.

**NB:** La classe `Font` définit les caractéristiques d'une police de caractères

# LA CLASSE GRAPHICS

## Exemple d'utilisation:

(g est un objet de type Graphics)

```
g.setColor (Color.red); // couleur courante : rouge
```

```
g.drawString ("Bonjour", 5, 50); // "Bonjour" en x = 5  
et y = 50
```

```
g.drawRect (10, 10, 50, 50); // Rectangle de CSG  
(10,10); w = 50; h = 50
```

# LA CLASSE COMPONENT (COMPOSANT GRAPHIQUE)

- La classe **abstraite Component** de AWT dérive de la classe Object et contient une bonne centaine de méthodes.
- Elle contient des méthodes permettant d'accéder ou de modifier les caractéristiques communes à tous les composants : dimension du composant, coordonnées du coin supérieur gauche du composant, couleur du texte ou du fond est-il visible ou non ? etc

..

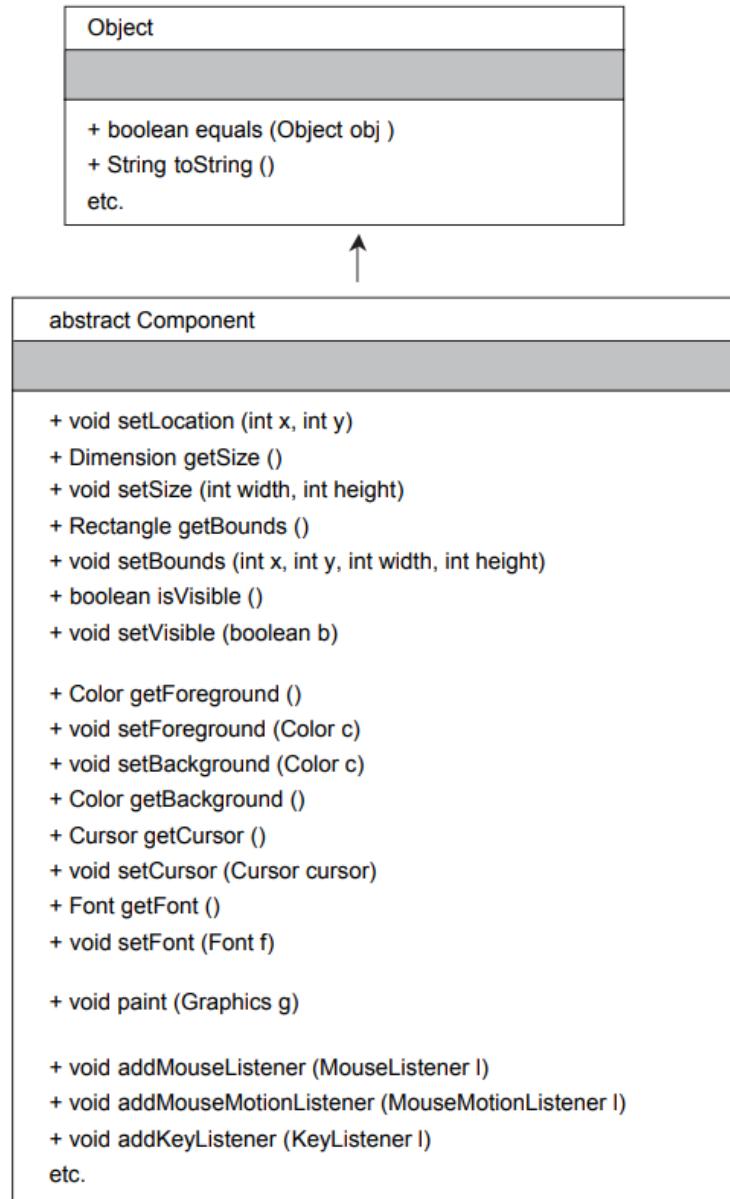
# LA CLASSE COMPONENT (COMPOSANT GRAPHIQUE)

- Les composants sont disposés sur la fenêtre initiale. Certains composants peuvent recevoir d'autres composants : on les appelle des conteneurs (container en anglais). L'ensemble des composants d'une application constitue un arbre des composants, la racine de l'arbre étant la fenêtre initiale de l'application.
- La disposition des composants dans la fenêtre peut être imposée par le programme qui indique, pour chaque composant, le coin supérieur gauche du composant (CSG), sa largeur et sa hauteur ou se faire suivant un des gestionnaire de mise en page (**LayoutManager**).

..

# LA CLASSE COMPONENT (COMPOSANT GRAPHIQUE)

Il s'agit d'une classe abstraite qui hérite de Object et définit les caractéristiques communes aux composants graphiques.



# LA CLASSE COMPONENT (COMPOSANT GRAPHIQUE)

- Les principales méthodes de la classe Component sont:
  - public **Rectangle getBounds()** : fournit un objet Rectangle indiquant la position du composant (CSG : coin supérieur gauche) par rapport au conteneur et ses dimensions (largeur et hauteur) lors de l'appel de cette méthode,
  - public **Dimension getSize ()** : fournit la taille du composant (largeur et hauteur).
  - public **void setBounds (int x, int y, int width, int height)** : modifie le point CSG par rapport au conteneur et la taille du composant.
  - public **void setSize (int width, int height )** : modifie la taille du composant.

# LA CLASSE COMPONENT (COMPOSANT GRAPHIQUE)

- public **void setVisible (boolean b)** : le composant est visible si b est vrai, invisible sinon. Par défaut, les composants sont visibles, sauf les fenêtres qui sont a priori invisibles (Frame par exemple).
- public **Color getBackground ()** : fournit la couleur du fond du composant. Si aucune couleur n'est définie pour le composant, c'est celle du conteneur qui est utilisée.
- public **Color getForeground ()** : fournit la couleur du premier plan (texte ou dessin) du composant. Si aucune couleur n'est définie pour le composant, c'est celle du conteneur qui est utilisée.

..

# LA CLASSE COMPONENT (COMPOSANT GRAPHIQUE)

- public **void setBackground (Color c)** : modifie la couleur du fond du composant.
- public **void setForeground (Color c)** : modifie la couleur du premier plan du composant.
- public **Graphics getGraphics ()** : fournit un objet de type Graphics pour ce composant.
- public **void paint (Graphics g)** : dessine le composant en utilisant le contexte graphique g.
- public **void update (Graphics g)** : redessine le fond et appelle la méthode paint (g).
- public **void repaint ()** : appelle update (g), g étant le contexte graphique du composant

..

## LA CLASSE LABEL

- La classe **Label** définit une ligne de texte que le programme peut changer en cours d'exécution. Pour l'utilisateur, ce Label est une chaîne constante qu'il ne peut pas modifier sur son écran.
- Ce composant dispose de toutes les méthodes de Component (héritage). On peut lui définir un emplacement, une taille, une couleur de fond ou de texte, une fonte particulière.
- A cela, s'ajoute quelques spécificités des Label concernant la justification (gauche, centré, droite) dans l'espace attribué.

# LA CLASSE LABEL

- Attributs et méthodes de Label :
  - Label.CENTER, Label.LEFT, Label.RIGHT : constantes static pour justifier le texte.
  - Label (), Label (String eti), Label (String eti, int justification) : constructeurs de Label,
  - String getText () : fournit le texte du Label.
  - void setText (String text) : modifie le texte du Label,

..

## LA CLASSE BUTTON

- La classe **Button** définit un bouton affichant un texte et déclenchant une action lorsqu'il est activé à l'aide de la souris.
- Exemple : un bouton Quitter.
- Méthodes :
  - Button (String label) : constructeur d'un bouton ayant l'étiquette label.
  - String getLabel () : fournit l'étiquette du bouton.
  - void setLabel (String label) : modifie l'étiquette du bouton.

## LA CLASSE BUTTON

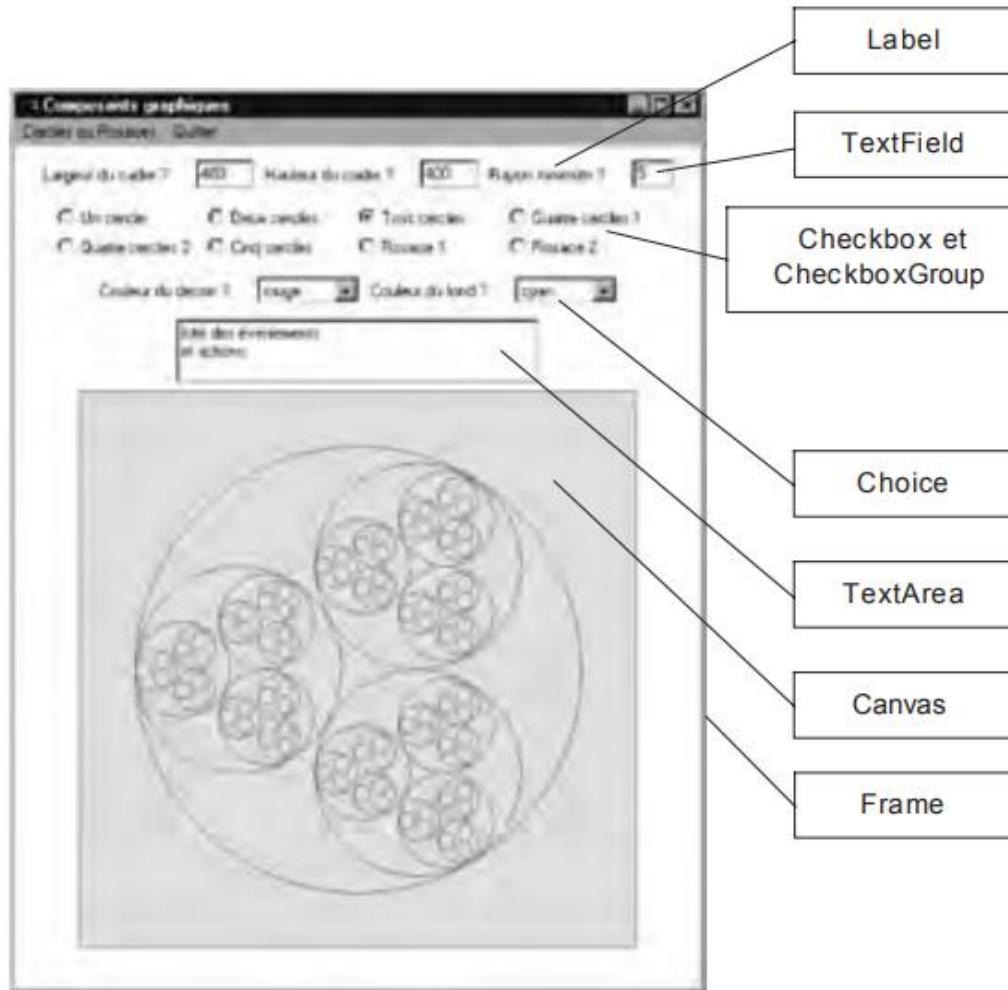
- void addActionListener (ActionListener l) : enregistre un objet de type ActionListener qui indique ce qu'il faut faire quand on appuie ou relâche le bouton. Cet enregistrement est retiré à l'aide de la méthode removeActionListener(). Le bouton devient alors inopérant.
- void setActionCommand (String command) : définit la chaîne de caractères qui est délivrée lorsque le bouton est activé.
- public String getActionCommand () : fournit la chaîne définie par setActionCommand() ou l'étiquette du bouton par défaut.

## LA CLASSE CANEVA

- La classe **Canvas** définit une zone rectangulaire de dessin.
- Méthodes sont :
  - Canvas () : constructeur de la zone de dessin.
  - void paint (Graphics g) : repeint le composant avec la couleur du fond.
  - Les applications doivent dériver la classe Canvas et redéfinir la méthode paint() en fonction de l'application.

66

# LA CLASSE CANEVA



# LA CLASSE CHECKBOX

- La classe Checkbox définit une boîte à cocher. Celle-ci a une étiquette et un état booléen coché ou non.
- Les méthodes sont:
  - public **Checkbox (String label, boolean etat)** : crée une boîte à cocher d'étiquette label avec l'état coché ou non suivant le booléen etat,
  - public **Checkbox (String label, boolean etat, CheckboxGroup group)** : constructeur qui insère la boîte dans un groupe de boîtes.
  - String **getLabel ()** : fournit l'étiquette de la boîte à cocher.
  - boolean **getState ()** : fournit l'état de la boîte à cocher.

# LA CLASSE CHECKBOX

- **void addItemListener (ItemListener l) :** l'objet l de type ItemListener indique ce qu'il faut faire si la boîte à cocher est sélectionnée.
- **.removeItemListener (ItemListener l):** annule l'opération précédente ; l'écouteur l devient inopérant.

»

# LA CLASSE CHOICE, LIST ET SCROLLBAR

- La **classe Choice** définit un menu contenant une liste d'options dont une seule peut être activée (choisie).
- La **classe List** définit une liste déroulante d'éléments. Le nombre d'éléments affichés peut être modifié. On peut également sélectionner un ou plusieurs éléments de la List.
- La **classe Scrollbar** définit une barre de défilement vertical ou horizontal. Cette barre représente les valeurs entières comprises entre les deux entiers minimum et maximum. Le curseur représente la valeur courante de la barre. La valeur courante peut être modifiée (avec un incrément paramétrable) suite à des clics de la souris sur cette barre.

..

## LA CLASSE TEXTFIELD ET TEXTAREA

- La **classe TextField** est une zone de saisie d'une seule ligne de texte. Les caractères entrés peuvent être remplacés par un caractère choisi dit écho (pour les mots de passe par exemple),
- La **classe TextArea** est une zone de saisie ou d'affichage de texte sur plusieurs lignes. Cette zone peut avoir ou non des barres de défilement lorsque la fenêtre est trop petite pour afficher tout le texte horizontalement ou verticalement.

# LA CLASSE CONTAINER

*abstract Container*

- + Component add (Component comp)
- + Component getComponent (int n)
- + void remove (int n)
- + void remove (Component comp)
- + int getComponentCount ()
- + void setLayout (LayoutManager mgr)
- + paint (Graphics g)
- + etc.

- Un objet de type **Container** est un composant graphique qui peut contenir d'autres composants graphiques tels que ceux définis précédemment : Label, Button, Checkbox, Choice, Scrollbar, etc. ou un autre container.
- Les composants ajoutés à un conteneur sont enregistrés dans une liste. On peut retirer un composant de la liste. À un conteneur, on peut ajouter

## LA CLASSE PANEL

- La **classe Panel** définit un conteneur fournissant de l'espace dans lequel on peut ajouter d'autres composants y compris d'autres Panel. La mise en page est par défaut de type FlowLayout. La classe définit deux constructeurs, le deuxième précisant le type de gestionnaire de mise en page choisi. Les autres caractéristiques sont définies à l'aide des méthodes de la super-classe Component (espace occupé, couleur de fond, etc.)

..

# LA CLASSE WINDOW ET FRAME

- La **classe Window** définit une fenêtre sans bordure et sans barre de menu. Par défaut, la mise en page est de type BorderLayout et la fenêtre est invisible. La fenêtre est sensible aux clics de la souris qui la font passer par défaut au premier plan. Un objet Window a obligatoirement pour père un objet de type Frame, Window ou Dialog qui doit être fourni au constructeur,
- La **classe Frame** définit une fenêtre principale d'application ayant un titre et une bordure. Par défaut, la mise en page est de type BorderLayout et la fenêtre est invisible.

# LA CLASSE DIALOG

- La **classe Dialog** définit une fenêtre de dialogue (saisie ou affichage de messages). On ne peut rien faire d'autre tant que la fenêtre n'est pas fermée.
- **Remarque** : un composant de type Window (et donc Frame et Dialog) ne peut être ajouté à un conteneur. C'est un composant de niveau 1 (top-level window).

..

# LA CLASSE JCOMPONENT

- **JComponent** est la classe de base de tous les composants Swing (JButton, JPanel, etc.) à l'exception des conteneurs de niveau supérieur (JFrame et JDialog). hérite la classe Container qui hérite elle-même Component.

# LA CLASSE JFRAME

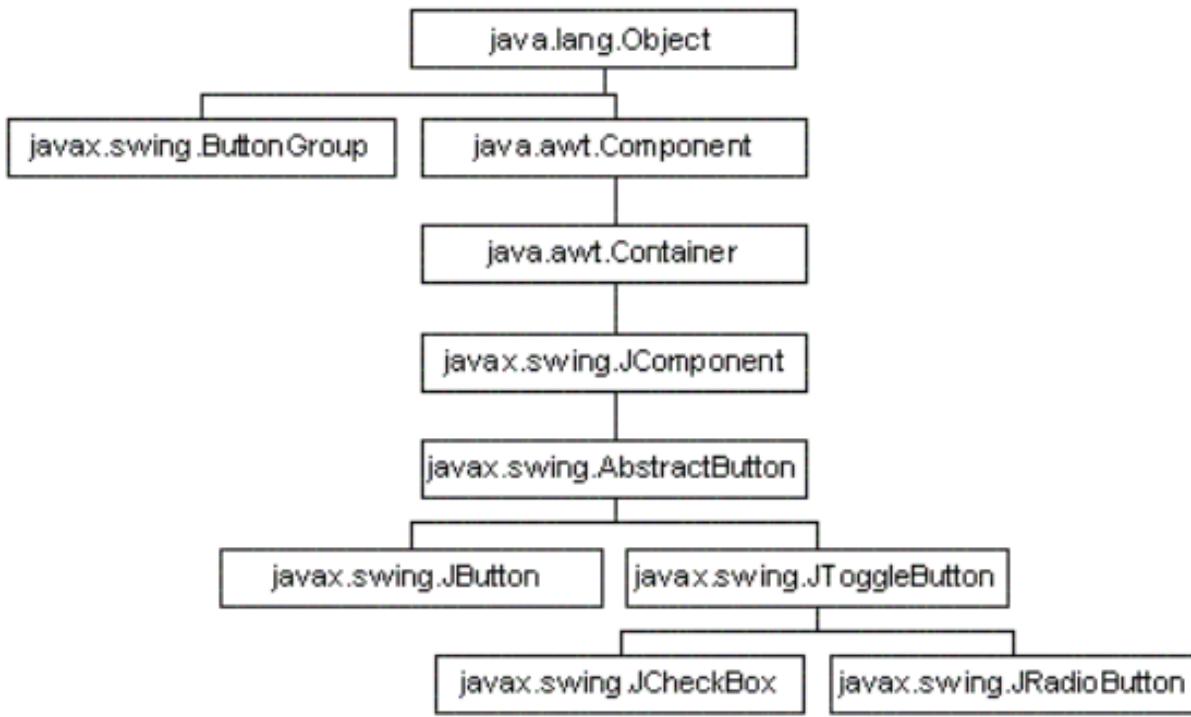
- Contre Frame, la Jframe utilise un Panneau de contenu pour insérer des composants (ils ne sont plus insérés dans le JFrame mais dans l'objet contentPane qui lui est associé). Elle représente une fenêtre principale qui possède un titre, une taille modifiable et éventuellement un menu.
- Par défaut, une JFrame est affichée dans le coin supérieur gauche de l'écran. Pour la centrer dans l'écran, il faut procéder comme pour une Frame : déterminer la position de la Frame en fonction de sa dimension et de celle de l'écran et utiliser la méthode setLocation() pour affecter cette position.
- La méthode setIconImage() permet de modifier l'icône de la Jframe,

# LA CLASSE JLABEL ET JPANEL

- JLabel propose les mêmes fonctionnalités que les intitulés AWT mais ils peuvent en plus contenir des icônes .
- La classe JPanel est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation (layout manager). Le gestionnaire par défaut d'un JPanel est un objet de la classe FlowLayout.

# LA CLASSE JButton

- Il existe plusieurs boutons définis par Swing.



# LA CLASSE JButton

1. **JButton** est un composant qui représente un bouton : il peut contenir un texte et/ou une icône.
2. **JToggleButton** définit un bouton à deux états : c'est la classe mère des composants JCheckBox et JRadioButton.
  - La méthode **setSelected()** héritée de AbstractButton permet de mettre à jour l'état du bouton. La méthode **isSelected()** permet de connaître cet état.
3. **ButtonGroup** permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné

..

## LA CLASSE JButton

4. **JRadioButton** représente un groupe de boutons . A un instant donné, un seul des boutons radio associés à un même groupe peut être sélectionné. La classe JRadioButton hérite de la classe AbstractButton.

- pour chacun des états du bouton, en utilisant les méthodes setIcon(), setSelectedIcon() et setPressedIcon().

# LA CLASSE **JTEXTCOMPONENT**

- La classe abstraite **JTextComponent** est la classe mère de tous les composants permettant la saisie de texte.
- **JTextField** permet la saisie d'une seule ligne de texte simple.
- **JPasswordField** permet la saisie d'un texte dont tous les caractères saisis seront affichés sous la forme d'un caractère particulier ('\*' par défaut). Cette classe hérite de la classe JTextField.
  - La méthode setEchoChar(char) permet de préciser le caractère qui sera montré lors de la saisie.
  - la méthode getPassword() est utilisée pour obtenir la valeur du texte saisi.

# LA CLASSE JTEXTCOMPONENT

- **JEditorPane** permet la saisie de texte multilignes. Ce type de texte peut contenir des informations de mise en pages et de formatage. En standard, Swing propose le support des formats RTF et HTML.
- **JTextArea** permet la saisie de texte simple en mode multiligne.

# Cas d'étude: Création et affichage d'une fenêtre

# CRÉATION ET AFFICHAGE D'UNE FENÊTRE

```
Title("Jeu de dame");
```

```
Size(500,500);
```

```
Locate(-1,-1);
```

```
p=Image("damier.png");
```

```
Constrain(p,0,0,500,500);
```

```
Visu(p);
```

```
for(i=0;i<8;i++)
```

```
    for(j=0;j<8;j++)
```

```
        if((i+j)%2==0)
```

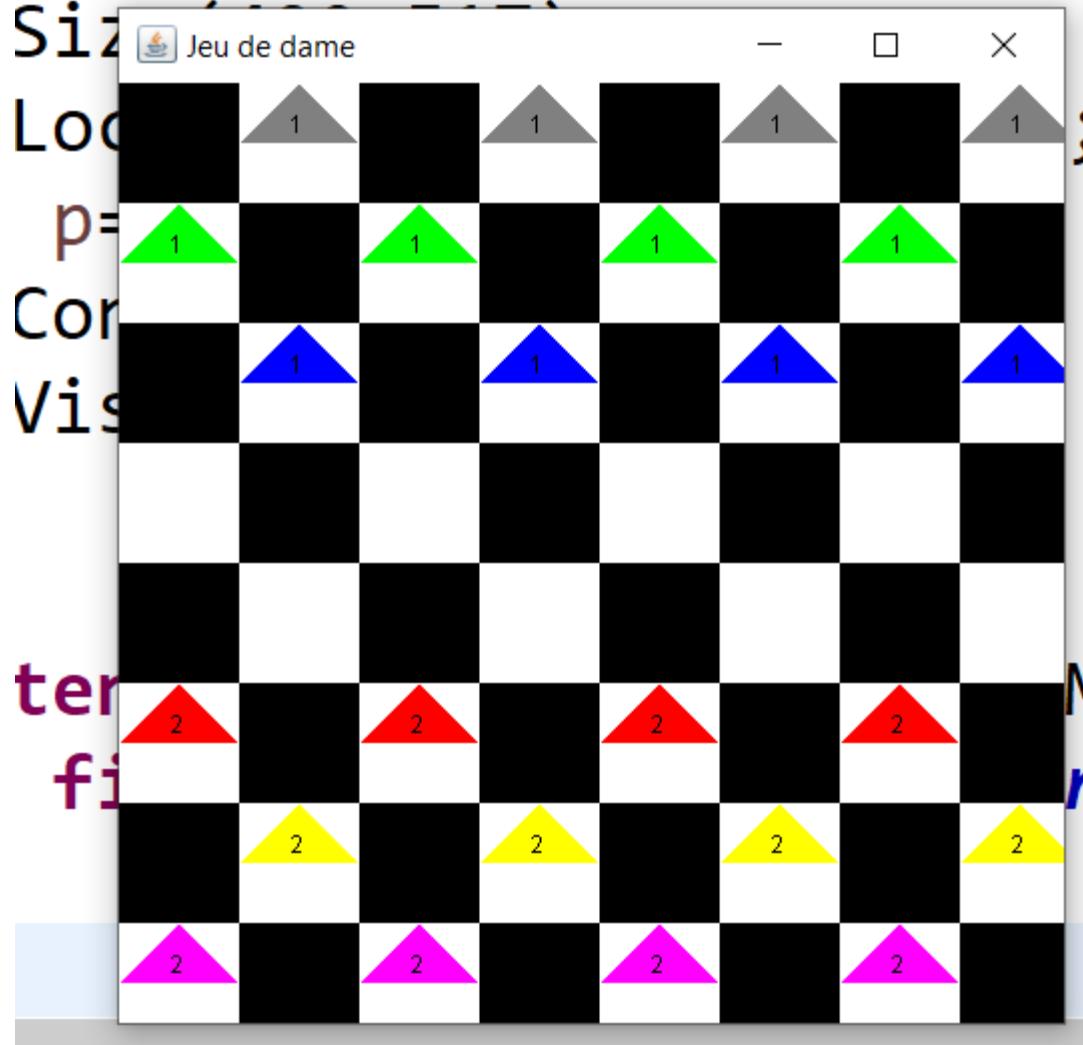
```
            if((i+j)<8)
```

```
                if((i+j)>=8)
```

```
                    if((i+j)<16)
```

```
                        if((i+j)>=16)
```

```
                            if((i+j)<24)
```



# CRÉATION ET AFFICHAGE D'UNE FENÊTRE

```
import javax.swing.*;  
  
public class Table {  
    public static void main(String[] args) {  
        JFrame jf=new JFrame();  
        jf.setTitle("Jeu de dame");  
        jf.setSize(490,517);  
        jf.setLocationRelativeTo(null);  
        Dessin p=new Dessin();  
        jf.getContentPane().add(p);  
        jf.setVisible(true) ;  
    }  
}
```

..

# Placer des composants dans une fenêtre

# PLACER DES COMPOSANTS DANS UNE FENÊTRE

```
class Dessin extends JPanel
{
    int [][]m = M();
    public void paint(Graphics g)
    {
        new Case(g,m);
        new Pièce(g,m);
    }
    public static int[][] M() {
        int [][] m=
        {
            {-1,1,-1,2,-1,3,-1,4},
            {5,-1,6,-1,7,-1,8,-1},
            {-1,9,-1,10,-1,11,-1,12},
            {-3,-1,-3,-1,-3,-1,-3,-1},
            {-1,-3,-1,-3,-1,-3,-1,-3},
            {13,-1,14,-1,15,-1,16,-1},
            {-1,17,-1,18,-1,19,-1,20},
            {21,-1,22,-1,23,-1,24,-1}};
```

..

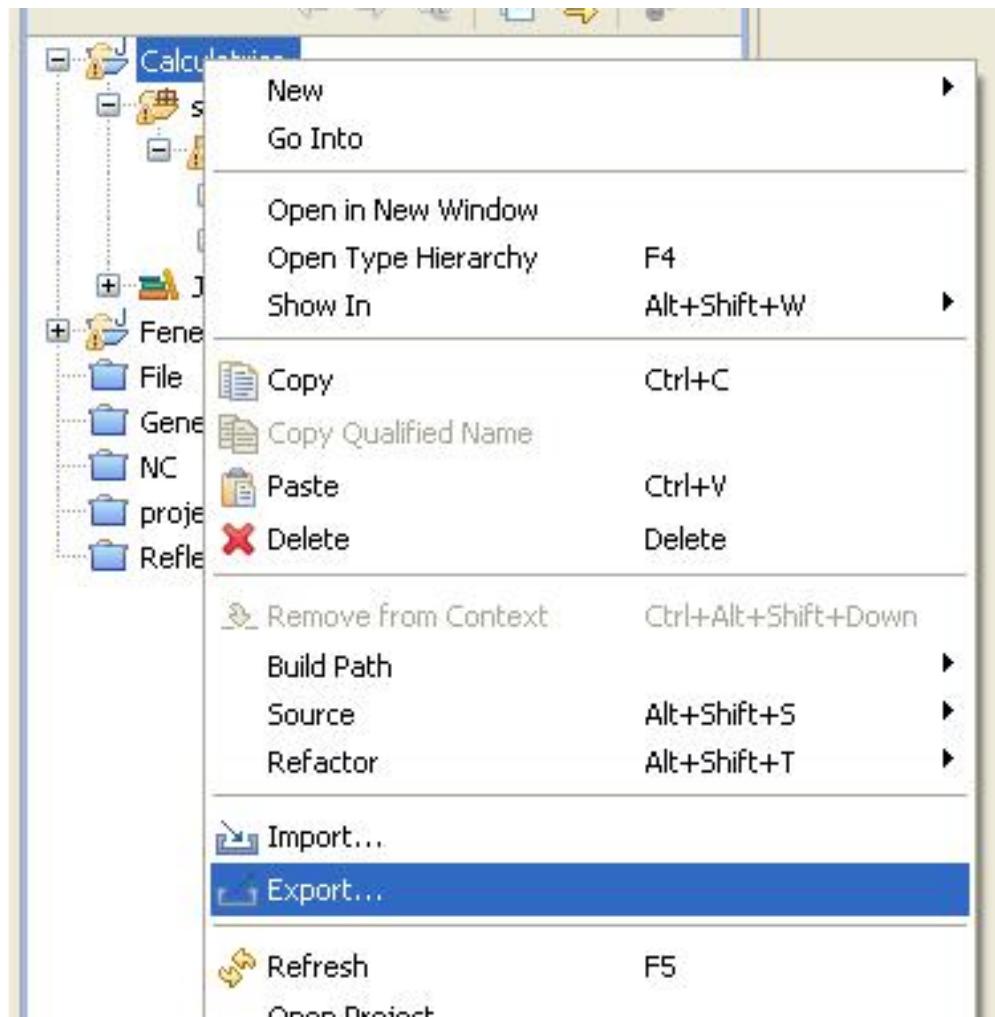
# Création de Jar exécutable

# CRÉATION DE JAR EXÉCUTABLE

- Un fichier **jar** est une archive Java (**J**ava **A**Rchive). Ce type de fichier contient tout ce dont a besoin la JVM pour lancer l'exécution du programme !
- Une fois votre archive créée, il vous suffira juste de double cliquer sur celle-ci pour lancer l'application.
- **Avec la condition d'ajouter les exécutables de JRE (présent dans le répertoire /bin) dans la variable d'environnement PATH !**

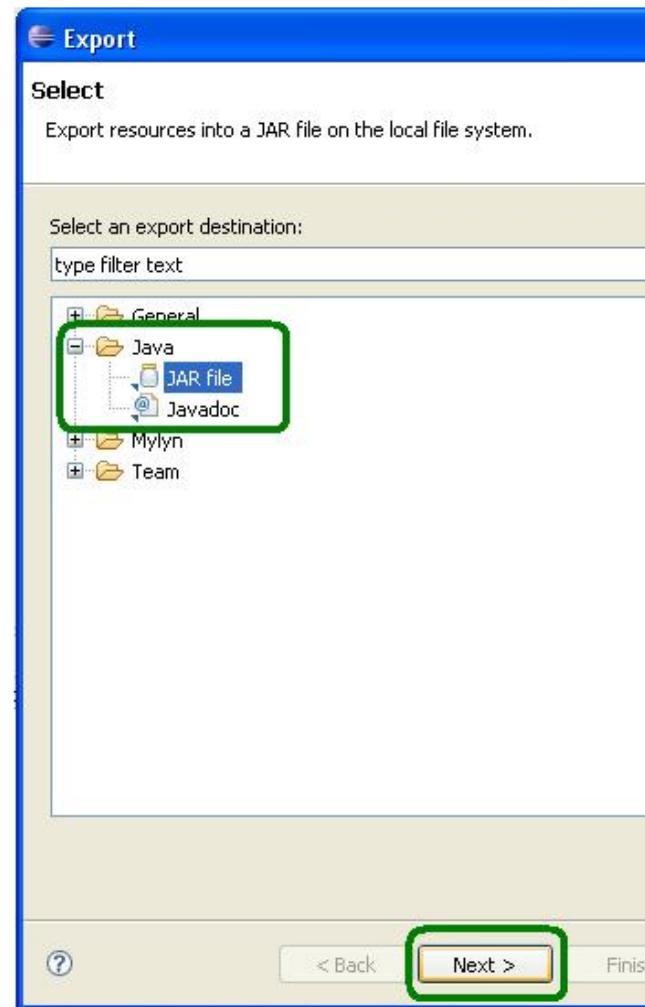
# CRÉATION DE JAR EXÉCUTABLE

1. Sous Eclipse, cliquer droit sur le projet et choisissez l'option Export, comme ceci :



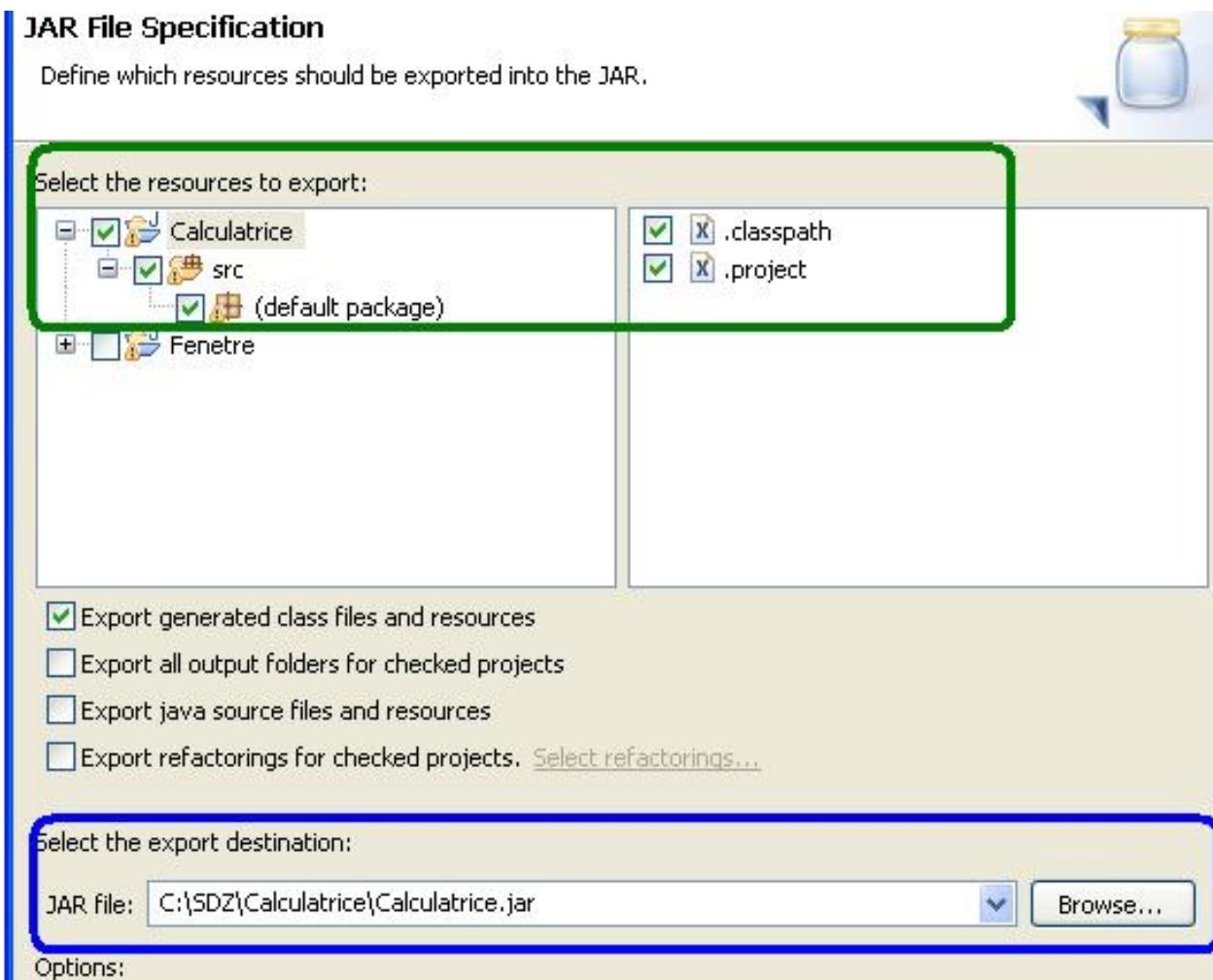
# CRÉATION DE JAR EXÉCUTABLE

2. Ensuite, Eclipse vous demande quel type d'export vous souhaitez faire : sélectionnez **JAR File** et cliquez sur **Next**.



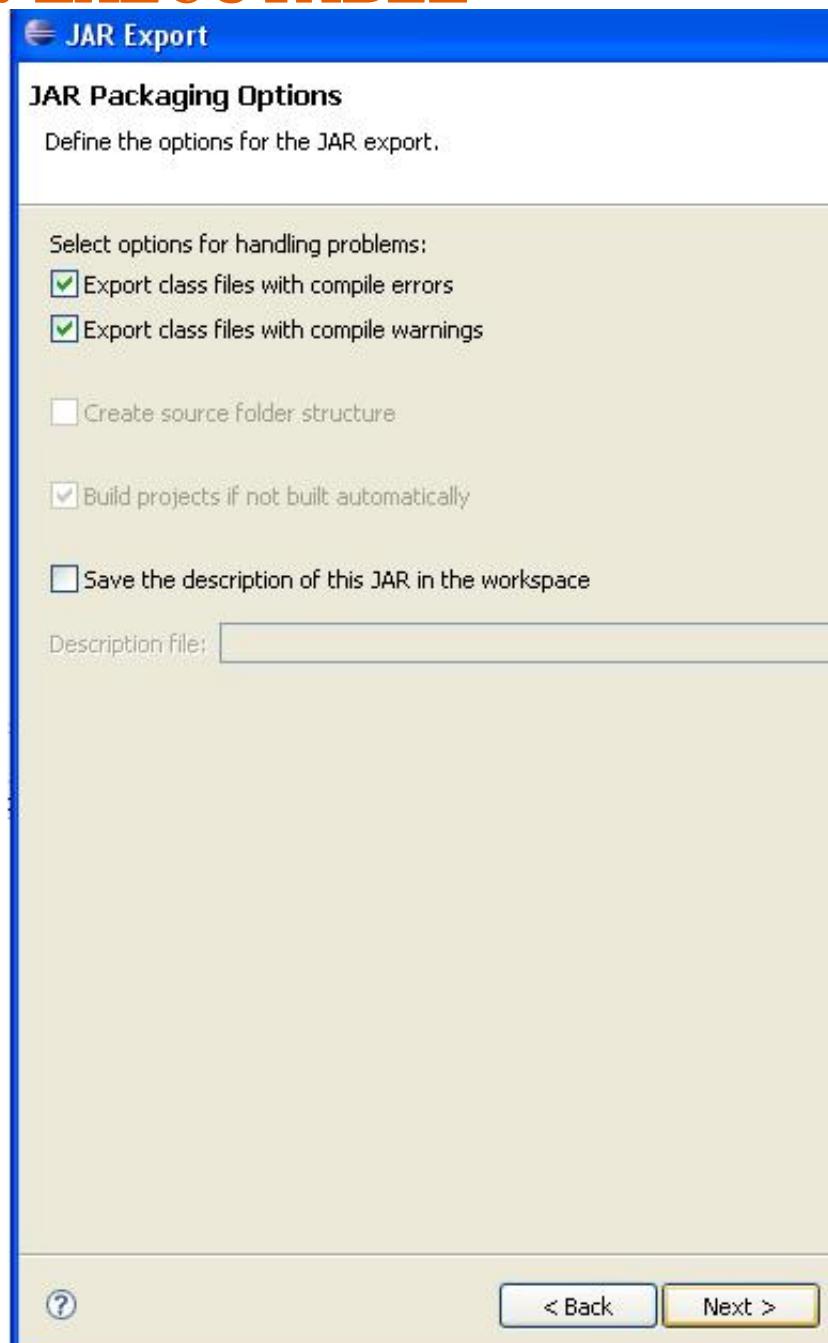
# CRÉATION DE JAR EXÉCUTABLE

- Maintenant, il demande quels fichiers vous voulez mettre dans votre archive.



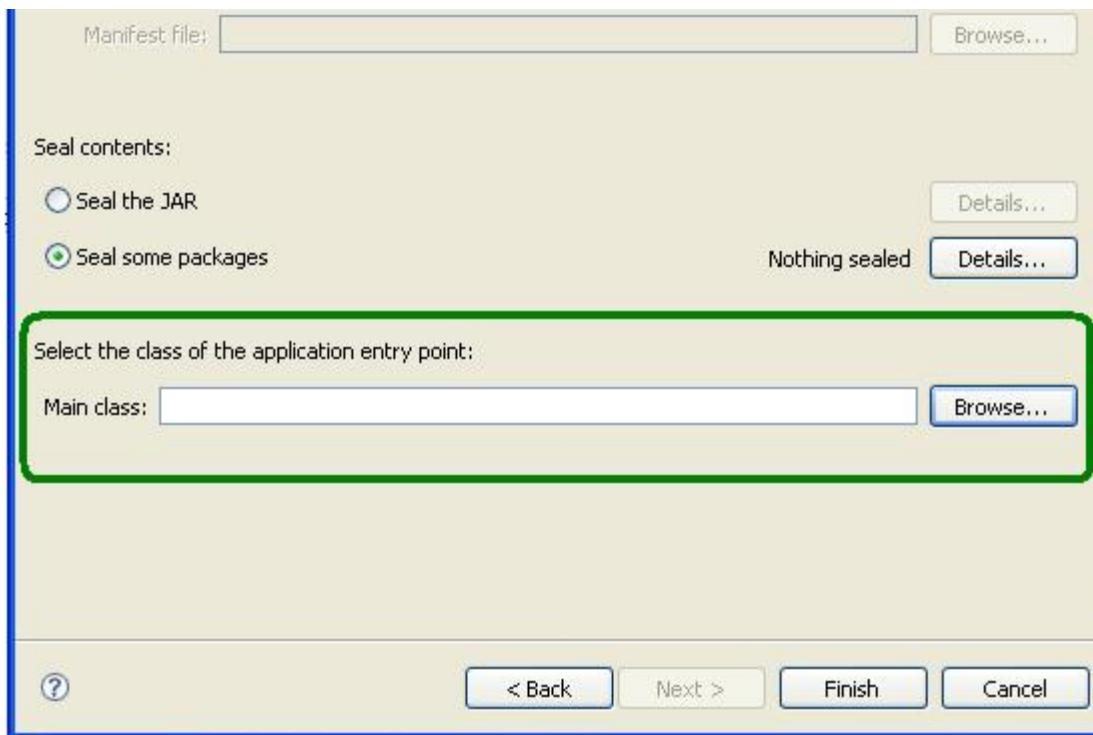
# CRÉATION DE JAR EXÉCUTABLE

Cliquez sur  
Next



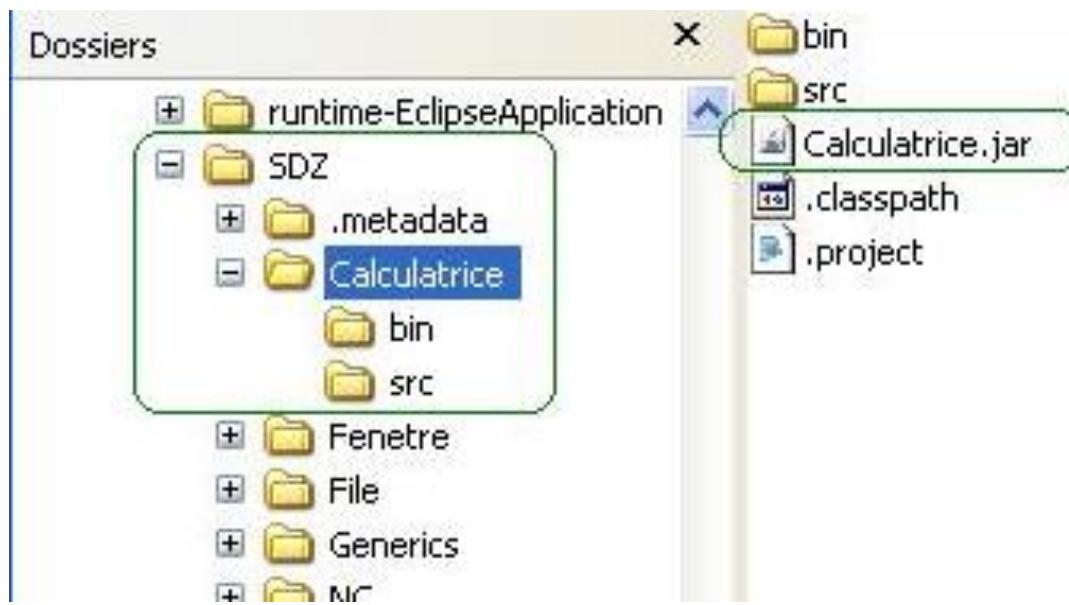
# CRÉATION DE JAR EXÉCUTABLE

- Il faut sélectionner la méthode **main** du programme en cliquant sur **Browse ...** en suite **Finish:**



# CRÉATION DE JAR EXÉCUTABLE

- Une fois cette étape validée, vous pouvez voir qu'un fichier **.jar** a bien été généré dans le dossier spécifié ! Double cliquez sur lui pour lancer la calculatrice .



Cours  
Programmation Orientée Objet 2  
Pour  
ING 2

**Chap 04:  
Interfaces Graphiques**

MEKAHLIA Fatma Zohra LAKRID  
Maître de Conférences Classe B

Laboratoire de Modélisation, Vérification et Evaluation des Performances des systèmes complexes  
(MOVEP)  
Bureau 123

“

## Gestion des événements

## Gestion des événements

- Le principal objectif d'une application graphique est la programmation événementielle càd l'utilisateur peut déclencher des événements et réagir à ce qui se passe dans la fenêtre.
- Il utilise le clavier et la souris pour intervenir sur le déroulement du programme.
- Le système d'exploitation engendre des événements à partir des actions de l'utilisateur.
- Le programme doit lier des traitements à ces événements.

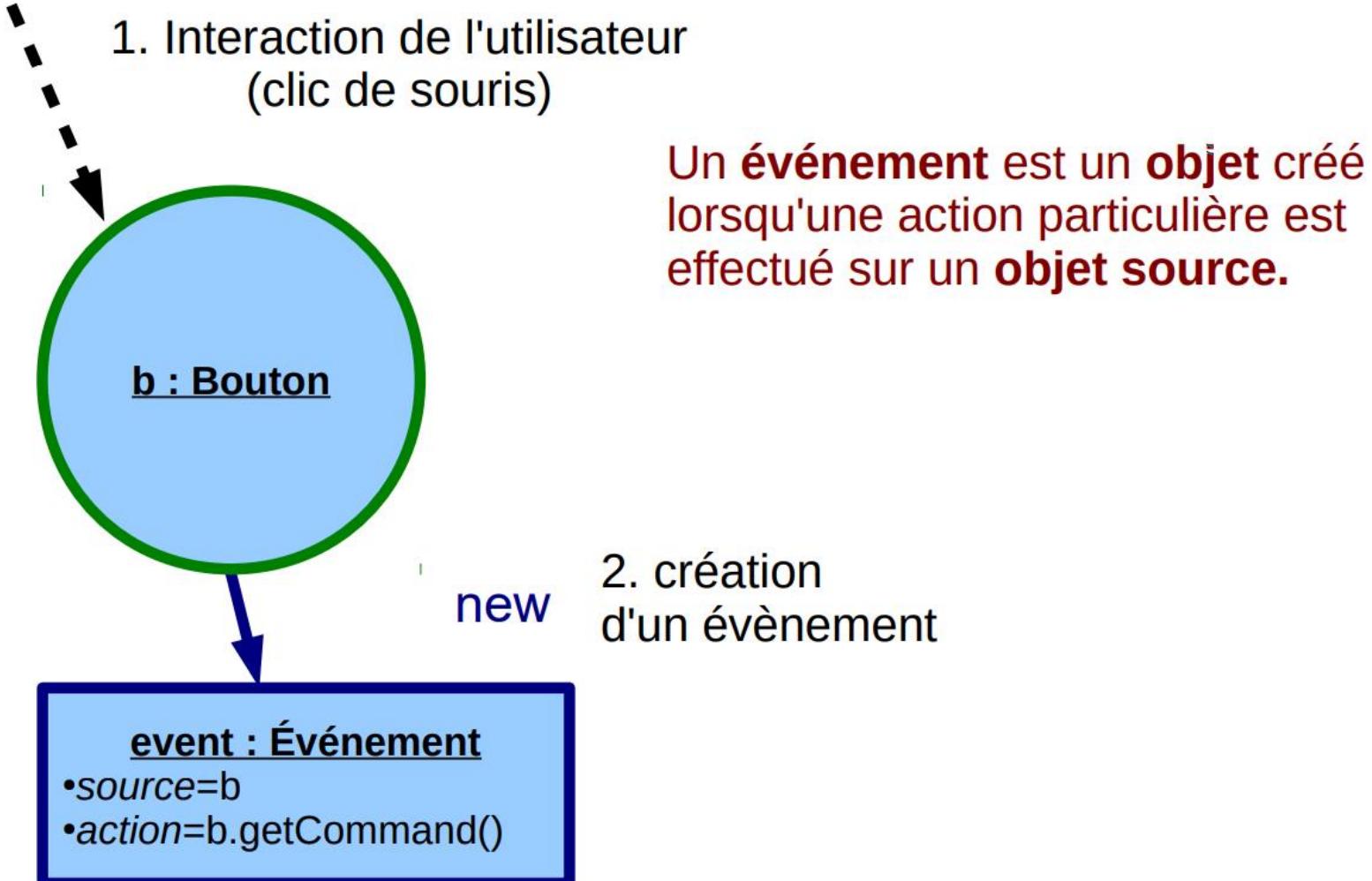
## Gestion des événements

- Des événements provoqués par la souris ou le clavier peuvent concerner un composant graphique et le faire réagir. Si on veut que le composant soit sensible à certaines catégories d'événements, il faut le demander explicitement en mettant un objet à l'écoute de l'événement (un Listener) et en indiquant ce qui doit être fait lorsque l'événement survient.

# Exemples d'événements

- appui sur un bouton de souris ou une touche du clavier.
- relâchement du bouton de souris ou de la touche.
- déplacer le pointeur de souris.
- clic de souris: clic sur un bouton par exemple.
- choisir un élément dans une liste.
- modifier le texte d'une zone de saisie.

# Evénements / Listener



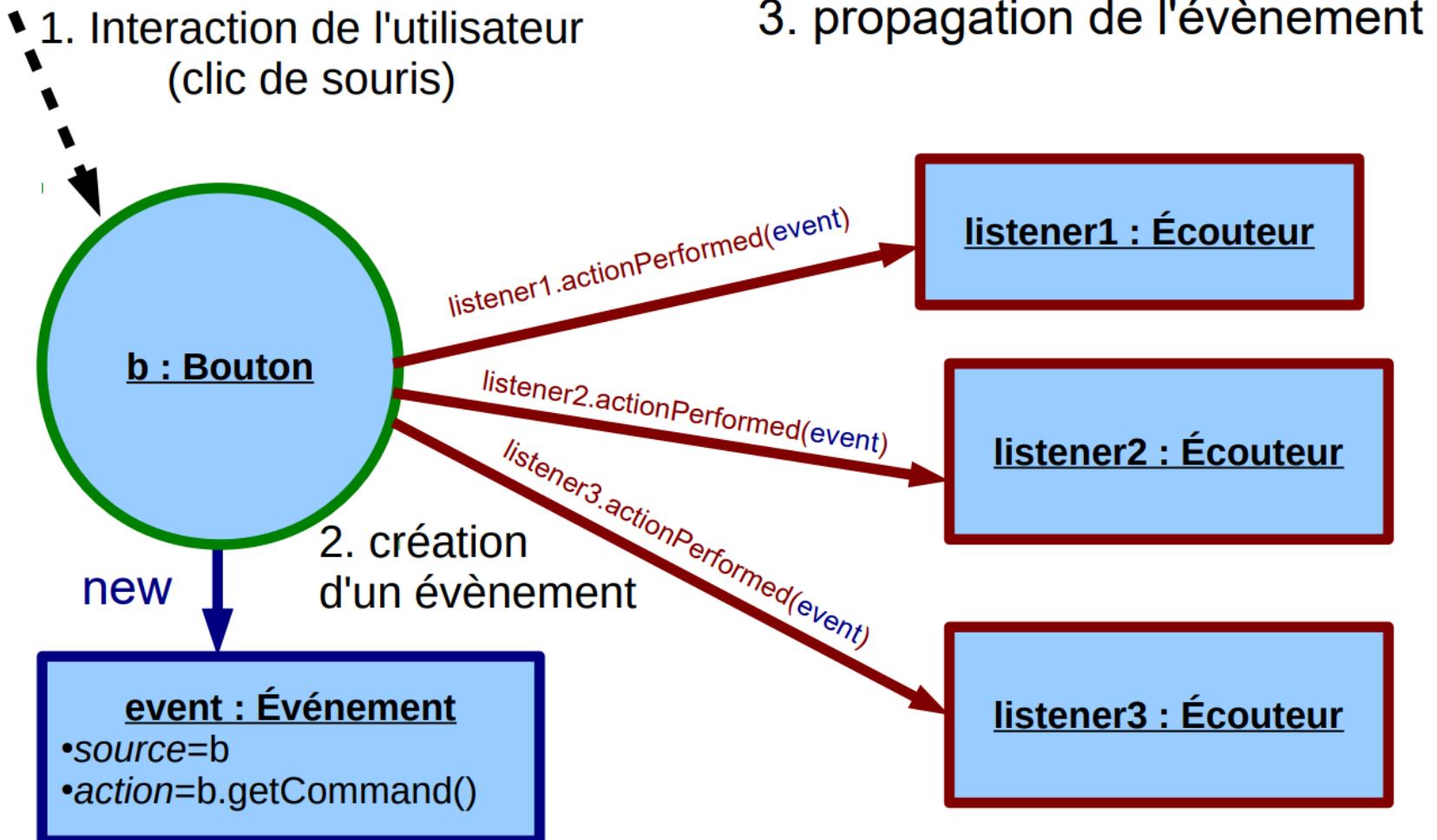
# Evénements / Listener

- Un composant qui crée des événements est appelé **source** (Exemple: bouton Ajouter de notre calculatrice).
- Le composant **source délègue** le traitement de l'événement au composant **auditeur (listener)**, c'est le composant qui traite l'événement.
- Chacun des composants graphiques a ses écouteurs (*listeners*)

# Evénements / Listener

- Un composant peut avoir plusieurs écouteurs (par exemple, 2 écouteurs pour les clics de souris et un autre pour les frappes de touches du clavier).
- Un écouteur peut écouter plusieurs composants.

# Evénements / Listener



Un écouteur est un objet réagissant aux évènements d'un objet source. Pour qu'un objet puisse écouter un objet source, il doit s'enregistrer auprès de celui ci

# Evénements / Listener

- **Question:** Quel message sera envoyé par le composant à ses écouteurs pour les prévenir que l'événement qui les intéresse est arrivé ?
- **Réponse :** à chaque type d'écouteur correspond une interface que doit implémenter la classe de l'écouteur .

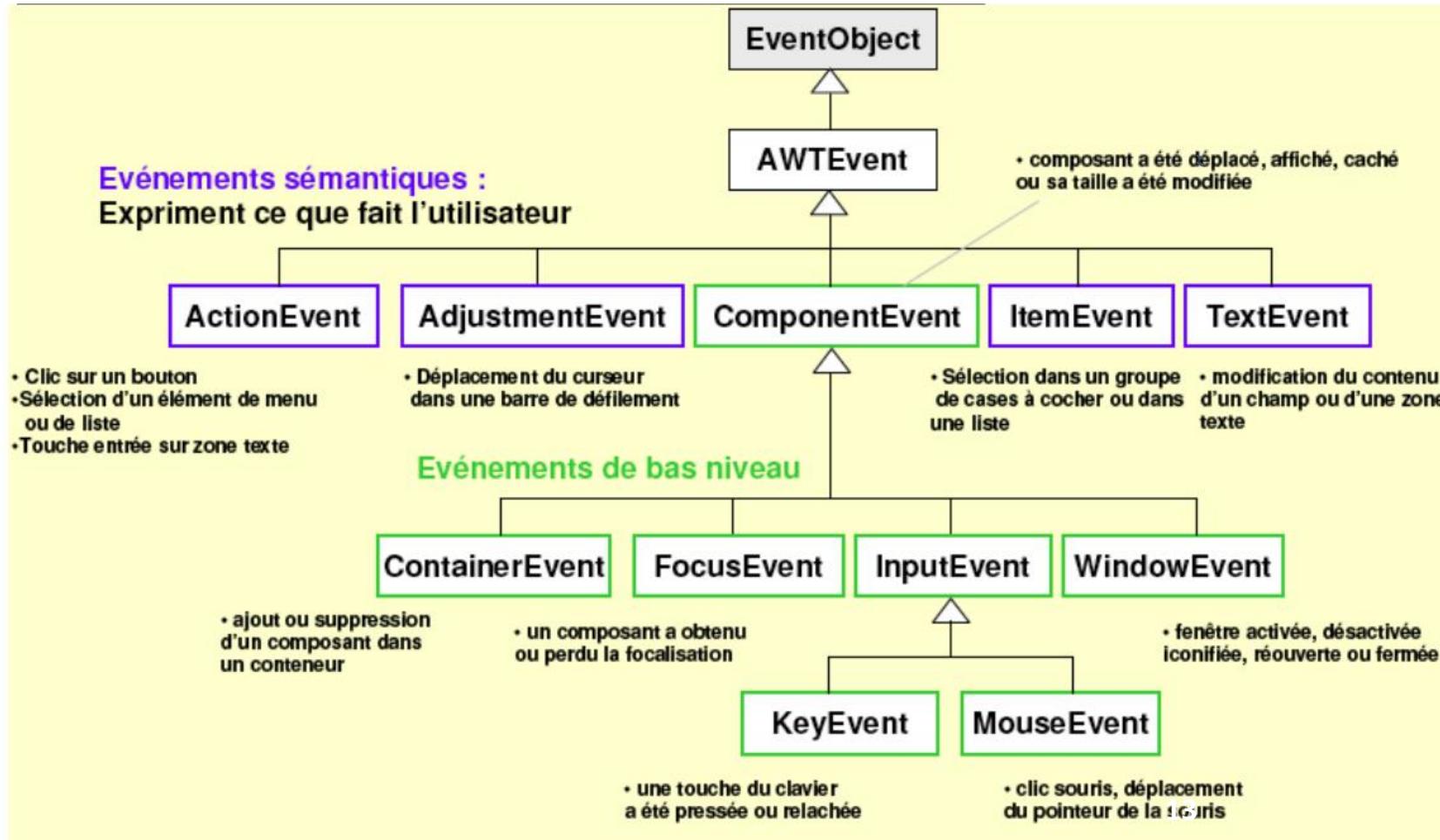
# Evénements / Listener

- Tout **événement** hérite de la classe **EventObject**. Les différents types d'événements sont représentés par des classes différentes
- Tout **listener** correspond à une interface qui hérite de **EventListener**.
- Toute classe désirant recevoir des notifications d'un type d'événement donné devra implémenter l'interface correspondante :
  - ActionEvent ..... ActionListener
  - MouseEvent..... MouseListener
  - KeyEvent..... KeyListener
  - WindowEvent ..... WindowListener
  - ...

# Classes d'événements / Listener

- **Les sources :**
  - Boutons : JButton, JRadioButton, JCheckBox, JToggleButton
  - Menus : JMenuItem, JMenu, JRadioButtonMenuItem, JCheckBoxMenuItem
  - Texte : JTextField
  - ...
- **Les listeners:**
  - Il faut implémenter l'interface qui correspond au type de l'événement !!

# Une partie de la hiérarchie des événements AWT



# Classe : ActionEvent

- Cette classe décrit des événements qui vont le plus souvent déclencher un traitement (une *action*) :
  - clic sur un bouton.
  - return dans une zone de saisie de texte.
  - choix dans un menu.
- Ces événements sont très fréquemment utilisés et ils sont très simples à traiter.

# Interface: ActionListener

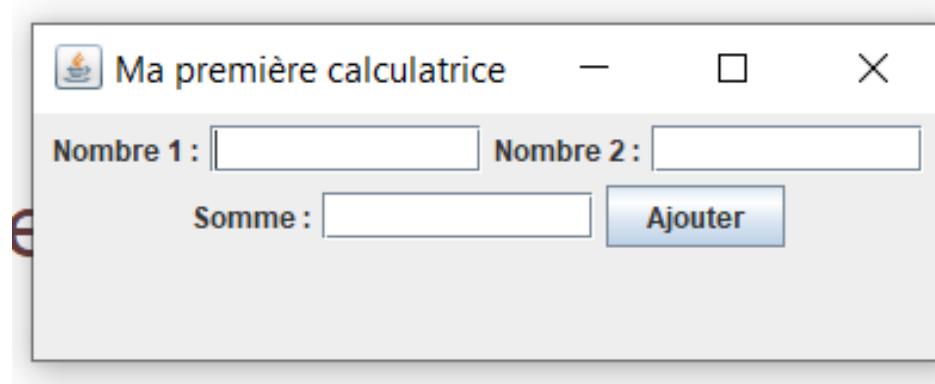
- Un objet *ecouteur intéressé par les événements de type « action »* (classe **ActionEvent**) **doit** appartenir à une classe qui implémente l'interface **java.awt.event.ActionListener**

## Exemples d'utilisation : bouton

- Exemple d'un **bouton** :
  - Un bouton est un élément graphique sur lequel l'utilisateur peut cliquer pour déclencher une action.
  - Le bouton ne fait rien tant que l'utilisateur n'a pas cliqué dessus.
  - Lors d'un clique un événement est créé ... reste à le traiter !

## Ecouter un bouton

- Le bouton « **Ajouter** » de notre calculatrice de TP4 (partie 1) n'été pas encore prêt à réagir afin d'additionner les deux nombres (voir projet CalculatriceVer1).



## Ecouter un bouton

- Si on veut que le bouton **Ajouter** soit sensible au clic et affiche la somme des deux nombres, il faut le demander explicitement en **ajoutant un objet à l'écoute de l'événement** (un Listener) et en indiquant ce qui doit être fait lorsque l'événement survient.

# Comment Activer le bouton Ajouter

- Créer une sous-classe qui implémente l'**interface ActionListener**.
- Implémenter la seule méthode abstraite de l'interface ActionListener (actionPerformed).
- L'implémentation de la méthode actionPerformed (ActionEvent e) sera exécuté suite au clic sur le bouton.
- Faire une liaison entre notre bouton Ajouter et l'**objet** de la sous-classe créée, en utilisant la méthode **addActionListner (Object o)** sur le bouton.

# Sous-classe: EventAjouter implements ActionListener

- `*****Sous-classe*****`
- `class EventAjouter implements ActionListener{`
- `@Override`
- `public void actionPerformed(ActionEvent arg0) {`
- `// TODO Auto-generated method stub`
- `try{`
- `int num1, num2, somme;`
- `num1=Integer.parseInt(entrée1.getText());`
- `num2=Integer.parseInt(entrée2.getText());`
- `somme=num1+num2;`
- `résultat.setText(Integer.toString(somme));`
- `}`
- `catch (Exception e){`
- `JOptionPane.showMessageDialog(null, "entrer un nombre");}`
- `}}`
- `/**Liaison entre le bouton Ajouter et l'objet de la sous-classe***/`
- `lancer.addActionListener(new EventAjouter());`

# Interface: MouseListener

- L'interface MouseListener prend en compte:
  - L'appui ou le relâchement du bouton de la souris dans l'espace du composant,
  - l'entrée ou la sortie du curseur de la souris de cet espace, ou un clic sur le composant.
  - A noter qu'un clic génère aussi les événements appui et relâchement.
  - Cinq méthodes de l'interface MouseListener traitent ces 5 événements :

# Interface: MouseListener

- void mouseClicked (MouseEvent evt) : clic sur l'espace du composant.
- void mouseEntered (MouseEvent evt) : le curseur entre dans l'espace du composant.
- void mouseExited (MouseEvent evt) : le curseur sort de l'espace du composant.
- void mousePressed (MouseEvent evt) : bouton de la souris appuyé sur le composant.
- void mouseReleased (MouseEvent evt) : bouton de la souris relâché sur le composant.

# Interface: MouseListener

- `/** Un programme qui permet d'afficher la position (X,Y) de clic souris dans le panel */`
- `import java.awt.event.MouseEvent;`
- `import java.awt.event.MouseListener;`
- `import javax.swing.JFrame;`
- `import javax.swing.JPanel;`
- `public class EcoutSouris implements MouseListener {`
- `@Override`
- `public void mouseClicked(MouseEvent arg0) {`
- `// TODO Auto-generated method stub`
- `System.out.println("La pos du clic : x="+ arg0.getX()+"y="+arg0.getY());`
- `}`
- `@Override`
- `... etc`

# Interface: MouseListener

- `public static void main (String [] args){`
- `JFrame fenetre= new JFrame("Ma première calculatrice");`
- `fenetre.setSize(400,150);`
- `fenetre.setVisible(true);`
- `fenetre.setLocationRelativeTo(null);`
- `JPanel p = new JPanel ();`
- `fenetre.add(p);`
- `EcoutSouris ecout = new EcoutSouris ();`
- `p.addMouseListener (ecout);`
- `}`

# Interface: MouseListener

```
11 @Override  
12 public void mouseClicked(MouseEvent e) {  
13     // TODO Auto-generated method stub  
14     System.out.println("la pos du clic:x=" + e.getX() + "y=" + e.getY());  
15 }  
16  
17 public static void main(String[] args) {  
18     new ApplicationFrame("Ma première calculatrice");  
19 }
```

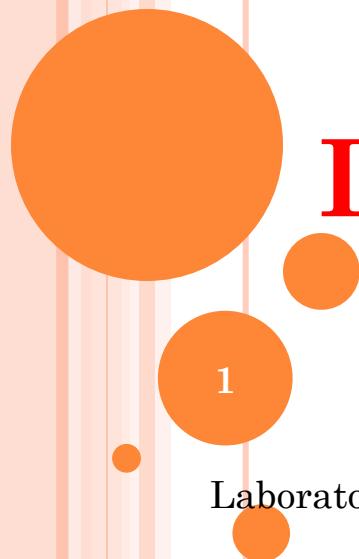
# L'interface MouseMotionListener

- MouseMotionListener prend en compte les déplacements de la souris quand le curseur se trouve dans l'espace du composant ; il gère deux événements : le déplacement du curseur sans appui sur le bouton, et le déplacement avec bouton de la souris constamment appuyé. Deux méthodes de l'interface MouseMotionListener traitent ces 2 événements :
- **void mouseMoved (MouseEvent evt)** : survol du composant, bouton non appuyé.
- **void mouseDragged (MouseEvent evt)** : survol du composant, bouton appuyé.

z

**MVC ... Le design pattern : l'architecture Modèle-Vue-Contrôleur**

Cours  
**Programmation Orientée Objet 2**  
Pour  
**ING 2**



**Chap 04:**  
**Interfaces Graphiques**

MEKAHLIA Fatma Zohra LAKRID  
Maître de Conférences Classe B

Laboratoire de Modélisation, Vérification et Evaluation des Performances des systèmes  
complexes (MOVEP)  
Bureau 123

# PLAN

- Généralités sur les interfaces graphiques.
- Composants des interfaces graphiques.
- Les packages AWT et Swing.
- Classes de base.
- Création et affichage d'une fenêtre.
- Placer des composants dans une fenêtre .
- Création de Jar exécutable.
- Gestion des événements.
- Le modèle MVC.

..

2

# MVC ... Le design pattern : l'architecture Modèle-Vue-Contrôleur

## DESIGN PATTERN

- les design patterns (patrons de conception en français) ?
- Ce sont des solutions générales et réutilisables d'un problème récurrent et considérées comme des "bonnes pratiques".
- Les patrons de conception sont une boîte à outils permettant de résoudre des problèmes classiques de la conception de logiciels. Ils définissent un langage commun pour aider votre équipe à communiquer plus efficacement.

## DESIGN PATTERN

- Les patrons de conception diffèrent par leur complexité, leur niveau de détails et l'échelle à laquelle ils peuvent être mis en œuvre.
- Permettent de résoudre des problèmes courants (par exemple : la conception d'une interface graphique).

# MVC: MODÈLE-VUE-CONTRÔLEUR

- Pour Modèle-Vue-Contrôleur ou en anglais Model-View-Controller.
- Le problème : la conception d'interface graphique et la programmation client/serveur.
- Solution datant de la fin des années 70 avec le développement des premières interfaces graphiques et indépendante des langages de programmation.
- C'est une manière de structurer une application graphique.

# MVC: MODÈLE-VUE-CONTRÔLEUR

En effet, MVC est un modèle de conception pour le développement d'applications logicielles qui sépare le modèle de données, l'interface utilisateur et la logique de contrôle.

**Son principe:** Organiser son architecture et son code en séparant trois rôles :

# MVC: MODÈLE-VUE-CONTRÔLEUR

- le modèle : la logique interne du programme, la gestion des données, les calculs, etc.
- la vue : l'affichage pour l'utilisateur final, tous les aspects graphiques
- le contrôleur : la gestion des évènements graphiques lancés par l'utilisateur et le lien entre la vue et le modèle.

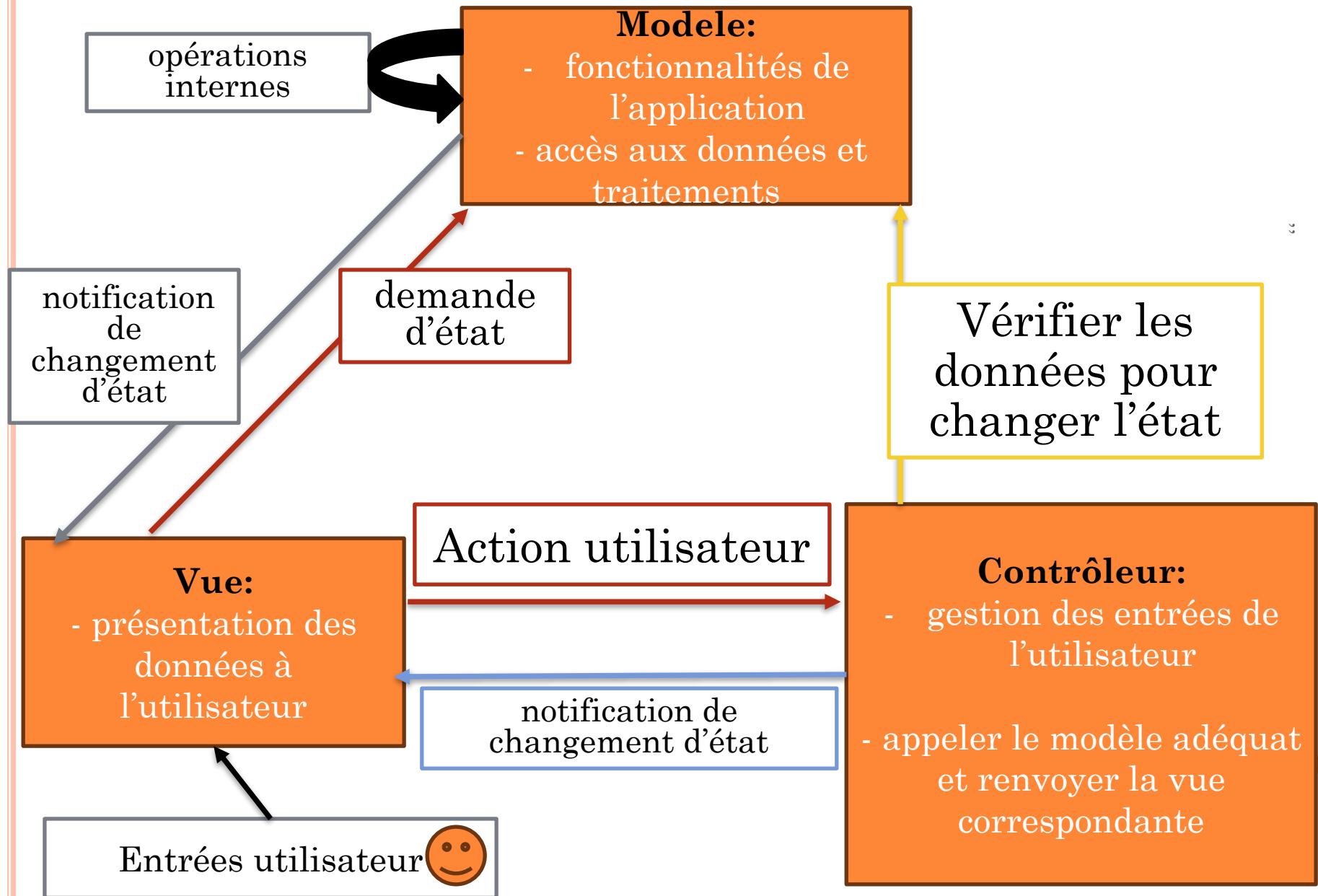
**Pourquoi ?** Organisation, modularité, maintenabilité, séparation des compétences et expertises entre les membres de l'équipe.

# MVC: MODÈLE-VUE-CONTRÔLEUR

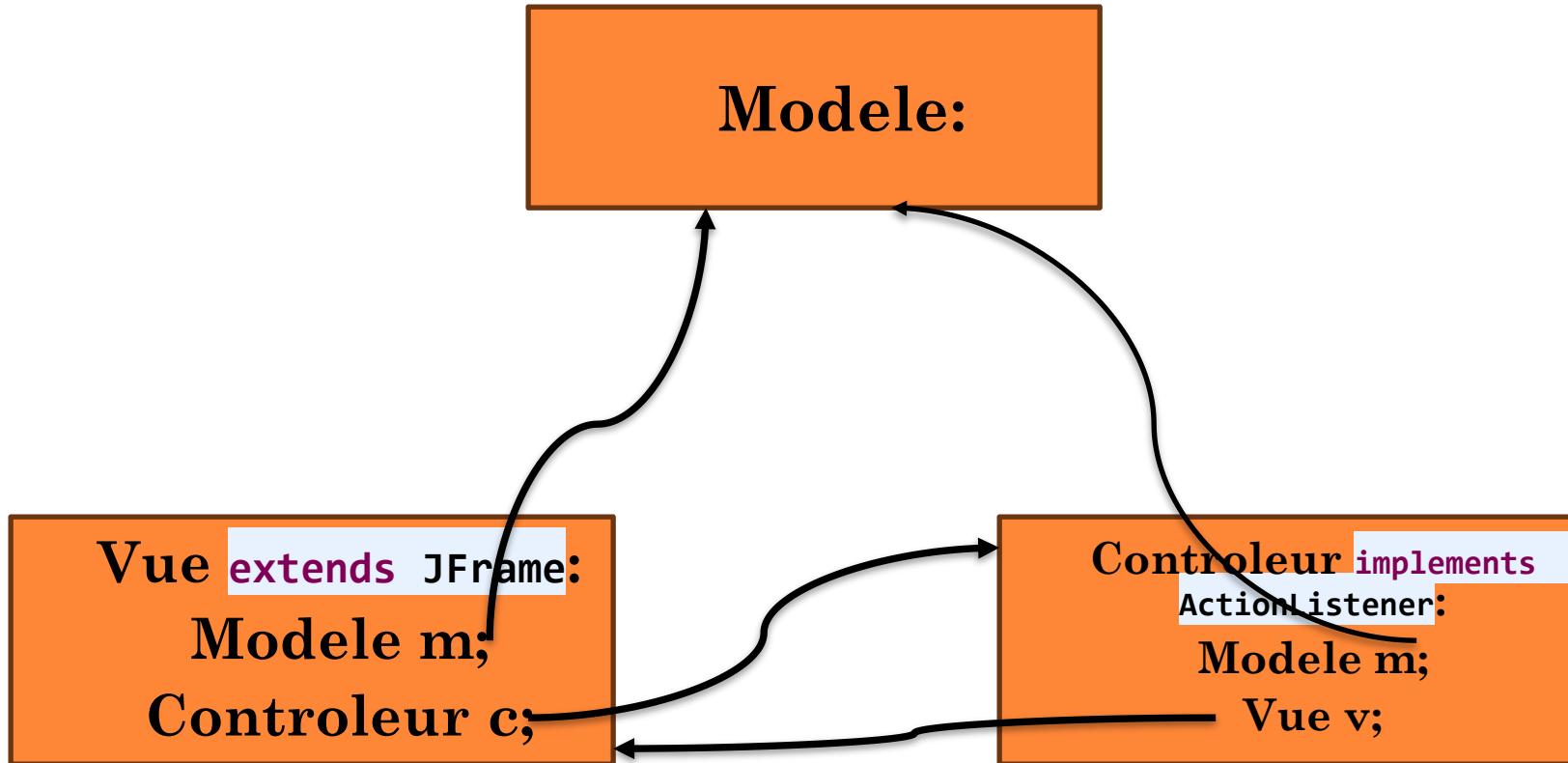
Dans une application structurée en MVC en trouve les étapes suivantes:

- L'utilisateur effectue une action sur l'application (un bouton par exemple),
- Le contrôleur capte l'action et contrôle la cohérence des données ainsi que leurs transformations au modèle. Il peut demander aussi à la vue de changer l'état.
- Le modèle reçoit les données et change l'état ( la valeur d'une variable qui change par exemple).
- Le modèle notifie la vue qu'il faut se mettre à jour. En conséquence, l'affichage de la vue est modifier.

# FLUX D'INFORMATION ENTRE LES COMPOSANTES



# RÉFÉRENCES ENTRE COMPOSANTS



# FLUX D'INFORMATION ENTRE LES COMPOSANTES

- **Vue:** c'est l'IHM qui représente l'état du modèle et ce que l'utilisateur a sous les yeux. Peut être:
  - ✓ Une application graphique Swing, AWT pour Java, Form pour C#.
  - ✓ Une page web.
  - ✓ Une console Windows ou encore un terminal Linux. etc

# L'INTÉRÊT EST DE:

- Séparer les responsabilités,
- Simplifier la maintenance.

..

# FLUX D'INFORMATION ENTRE LES COMPOSANTES

- **Modèle:** c'est là que se trouvent les données:
  - ✓ Il décrit les données manipulées par l'application,
  - ✓ Définit les méthodes d'accès au données,
  - ✓ Fournit les traitements applicables aux données,
  - ✓ Gère les interactions avec la base de données, etc.
- **Contrôleur:** cet objet permet de faire le lien entre la vue et le modèle en repenant sur les actions utilisateurs intervenant sur la vue:
  - ✓ Mettre à jour la vue ou le modèle.
  - ✓ Il n'effectue aucun traitement, ne modifie aucune donnée, il analyse la requête du client pour appeler le modèle adéquat et renvoi la vue correspondante à la demande.

# **EXEMPLE : GESTION D'UN POINT DU PLAN**

- Créer une application qui permet à l'utilisateur d'entrer les  **coordonnées du point.**
- L'application doit : **Afficher les coordonnées du point.**

66

# EXEMPLE : GESTION D'UN POINT DU PLAN

- `public class MauvaiseClassePoint {`
- `private float x, y;`
  
- `public MauvaiseClassePoint(float x, float y) {`
- `this.x = x; this.y = y; }`
  
- `public float getAbscisse () { return this.x;}`
- `public float getOrdonnee() { return this.y;}`
  
- `public void saisirPoint() {/*...*/}`
- `public void afficherPoint(){/*...*/}`
- `public void activerVuePoint(){/*...*/}`
- `public void saisirAbscisse(){/*...*/}`
- `public void saisirOrdonnee(){/*...*/}`
- `}`

# EXEMPLE : GESTION D'UN POINT DU PLAN

```
○ public class MauvaiseClassePoint {  
○     private float x, y;  
  
○     public MauvaiseClassePoint(float x, float y) {  
○         this.x = x; this.y = y; }  
  
○     public float getAbscisse () { return this.x;}  
○     public float getOrdonnee() { return this.y;}  
  
○     public void saisirPoint() {/*...*/}  
○     public void afficherPoint(){/*...*/}  
○     public void activerVuePoint(){/*...*/}  
○     public float saisirAbscisse(){/*...*/}  
○     public float saisirOrdonnee(){/*...*/}  
○ }
```

Gestion  
d'un point =  
**Modèle**

Gestion  
d'interactions  
avec  
l'utilisateur  
=  
**Vue**

# EXEMPLE : GESTION D'UN POINT DU PLAN

- **public void saisirPoint() {**
- **float unX = this.saisirAbscisse();**
- **float unY = this.saisirOrdonnee();**
- **This.modifierPoint(unX, unY);**
- **}**

*saisirPoint()* correspond à une requête d'utilisateur qui est proposée par la vue

Gérer par la vue  
(interactions avec l'utilisateur)

L'interprétation  
relève du contrôleur

Instruction s'adressant au *modèle* pour affecter les coordonnées saisies au Point courant

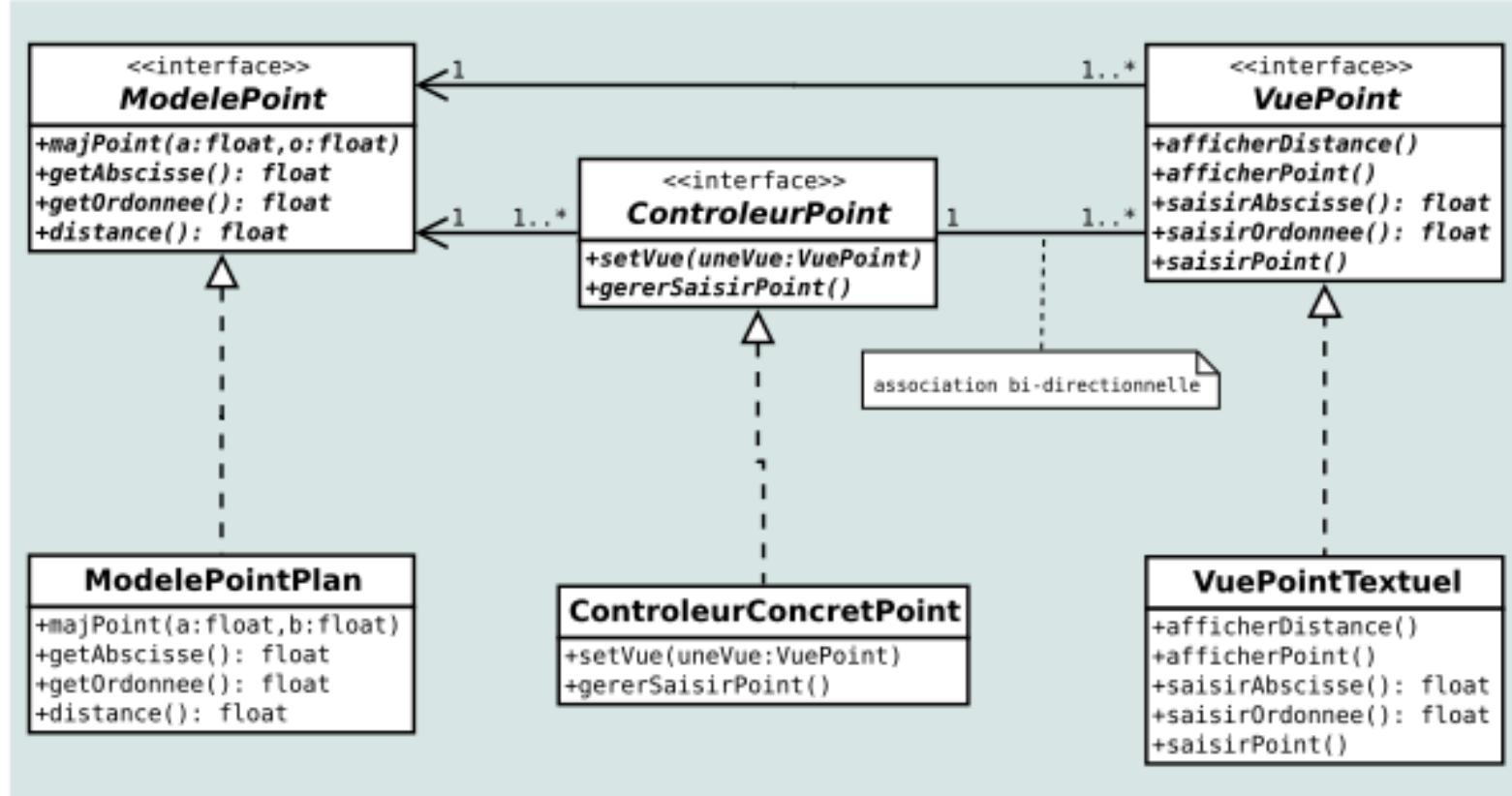
# LA CLASSE APPLICATION QUI ACTIVE LA VUE

```
o public class Application {  
  
o public static void main (String[] args){  
o   MauvaiseClassePoint p = new MauvaiseClassePoint(2,3);  
o   p.activerVuePoint();  
o } } ***** MENU POINT *****  
  
1 - entrer/modifier le point  
2 - sortir du programme
```

Activer la vue = menu affiché : les choix de l'utilisateur correspondent à des requêtes prédéfinies,

Constat: la classe Application n'interagit qu'avec la vue

# UTILISATION ET RÉALISATION DE MVC



il peut y avoir plusieurs *vues* et plusieurs *contrôleurs* associés au même *modèle*  
(chaque *vue* est en générale associée à un *contrôleur*)

# UTILISATION ET RÉALISATION DE MVC

## Classes :

- **ModelePointPlan;**
- **VuePointTextuel;**
- **ControleurConcretPoint;**

66

```
○ public class ModelPointPlan implements ModelePoint{  
○     private float x,y;  
  
○     public ModelPointPlan(){  
○     public ModelPointPlan(float x,float y){  
○         this.x=x;  
○         this.y=y;  
○     }  
○     public float getAbscisse(){  
○         return this.x;  
○     }  
○     public float getOrdonnee(){  
○         return this.y;  
○     }  
  
○     public void modifierPoint(float ab, float or) {  
○         this.x=ab;  
○         this.y=or; }  
○ }
```

..

```

○ import java.util.Scanner;
○ public class VuePointTextuel implements VuePoin
○     private ControleurConcretPoint contrroleur;
○     private ModelPointPlan model;
○
○     public VuePointTextuel(ControleurConcretPoint co, ModelPointPlan mo){
○         this.contrroleur=co; this.model=mo; }
○     public void activerVuePoint(){
○         saisirPoint();
○         afficherPoint();}
○     public void saisirPoint() {
○         System.out.println ("donner Les coordonnées d'un point");
○             this.contrroleur.gererSaisirPoint();}
○     public float saisirAbscisse(){
○         Scanner s = new Scanner(System.in);
○         return s.nextFloat();}
○     public float saisirOrdonnee(){
○         Scanner s = new Scanner(System.in);
○         return s.nextFloat();}
○     public void afficherPoint(){
○         System.out.println("abscisse = " + this.model.getAbscisse());
○         System.out.println("ordonnee = " + this.model.getOrdonnee());
○     }
○ }
```

Association avec le contrôleur et  
le modèle pour afficher les  
données qui se trouve dans le  
model

La vue délègue au contrôleur  
l'interprétation de la requête

La vue interroge le modèle  
pour afficher certaines  
données

```

○ public class ControleurConcretPoint implements ControleurPoint {

○     private ModelPointPlan model;
○     private VuePointTextuel vue; . . .

○

○     public ControleurConcretPoint(ModelPointPlan unModelPoint){
○         this.model = unModelPoint;
○     }

○

○     public void setVue (VuePointTextuel uneVuePoint){
○         this.vue = uneVuePoint;
○     }

○     public void gererSaisirPoint (){
○         float abs = this.vue.saisirAbscisse();
○         float ord = this.vue.saisirOrdonnee();

○         this.model.modifierPoint(abs, ord);
○     }
○ }

```

Si plusieurs vues sont contrôlées alors déclarez un ArrayList< VuePoint>

Pour changer la vue devant être contrôlée

Interprétation de la requête saisirPoint() de la vue

S'adresse à la vue pour que l'utilisateur entre l'abscisse et l'ordonnée

S'adresse au modèle pour enregistrer les nouvelles coordonnées

- **public class Application {**
- **public static void main(String[] args) {**
- **ModelPointPlan m = new ModelPointPlan();**
- **ControleurConcretPoint c = new ControleurConcretPoint(m);**
- **VuePointTextuel v = new VuePointTextuel(c,m);**
- **c.setVue(v);**
- **v.activerVuePoint();**
- **}**
- **}**

On crée le contrôleur et on lui associe le modèle déjà créée

Association de la vue au contrôleur

On crée une et on lui associe le contrôleur et le modèle

Activation de la vue qui attend les requêtes de l'utilisateur

```
<terminated> Application (5) [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (21 avr. 202)
```

donner les coordonnées d'un point

2

3

abscisse = 2.0  
ordonnee = 3.0

# AVANTAGES ET INCONVÉNIENTS DE MVC

## Avantages

- Séparation des responsabilités.
- Le modèle peut être partagé parmi divers vues et contrôleurs.

## Inconvénients

- L'architecture peut augmenter la complexité du système.

The END .... Thank you ☺