POO EN JAVA

Pr. Abdelmajid HAJAMI

BTS-DSI

2014

PLAN

- Concepts de la POO
- Présentation de JAVA
- Généralités
- Les types primitifs de JAVA
- Les instructions de contrôle de JAVA
- Les classes et les objets
- Les tableaux
- L'héritage
- Les chaînes de caractères et les types énumérés

CONCEPTS DE LA POO

CONCEPTS DE L'ORIENTÉ OBJET

- En approche orientée objet, le logiciel est considéré comme une collection d'objets qui collaborent entre eux, pour réaliser le métier du client.
- Les objets sont identifiés, et possèdent des caractéristiques statiques et dynamiques

CONCEPTS DE L'ORIENTÉ OBJET

Objet

- Un objet du monde réel est une chose concrète ou abstraite.
- Le monde réel n'est composé que de chose ou objets! **Tout est objet!**

- Un objet informatique est une représentation (un modèle) d'un objet du monde réel. Il modélise une chose abstraite ou concrète
- Un modèle est une abstraction, c-à-d un mécanisme qui permet de ne retenir que les caractéristiques essentielles d'un objet

CONCEPTS DE L'ORIENTÉ OBJET

Classe

- Il s'agit d'un modèle abstrait de données et de traitement représentant un objet réel dans la nature et regroupant des caractéristiques (attributs et méthodes) communes à des objets.
- Un objet est une simple instance de la classe

CONCEPTS DE L'ORIENTÉ OBJET

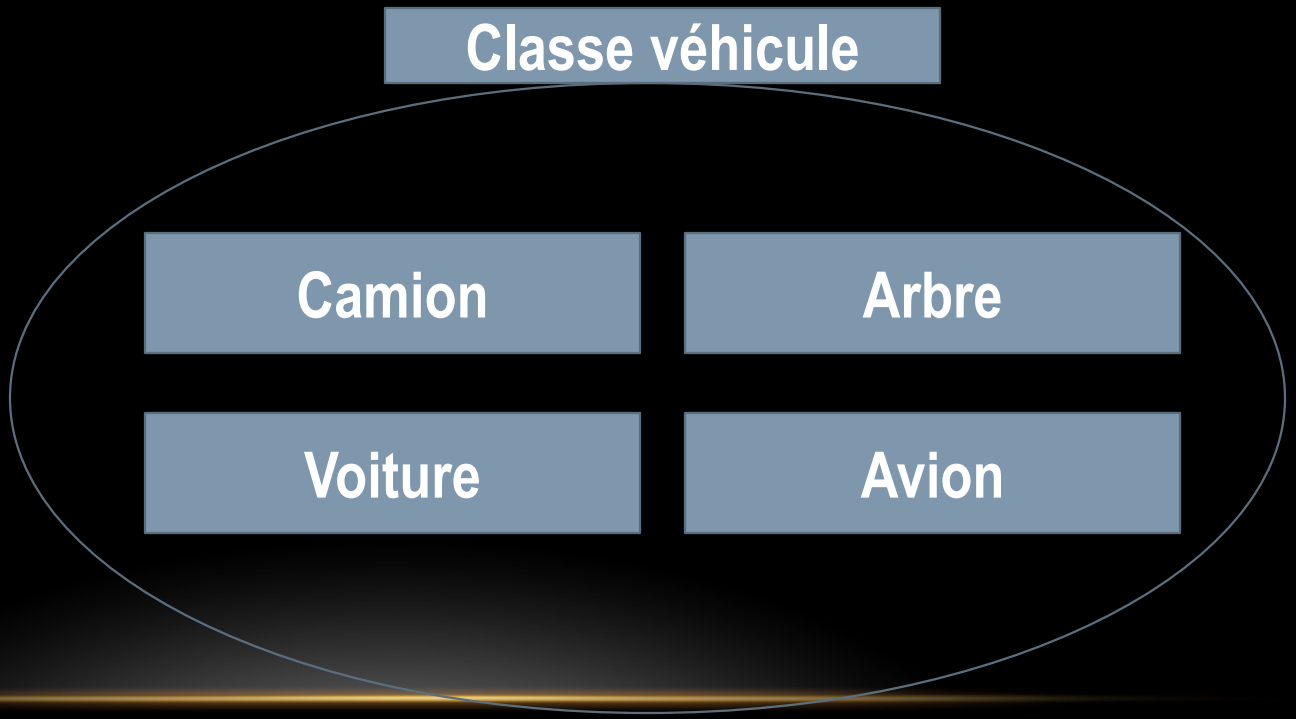
Classe

- En UML, elle est représenté par:

CONCEPTS DE L'ORIENTÉ OBJET

Classe

- Chasser l'intrus :



CONCEPTS DE L'ORIENTÉ OBJET

Identité

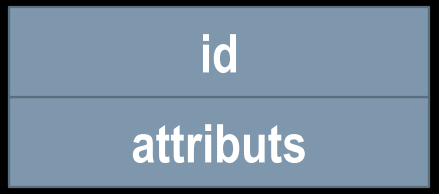
- Tout objet possède une identité , qui permet de l'identifier et de le différencier des autres objets.
- L'id se fait par le choix d'une variable unique qui permet de spécifier l'objet dans un système
- L'id permet de rendre l'objet persistant dans le mapping O/R ou bien avec les annotations de l'API JPA (Java Persistant API)

id

CONCEPTS DE L'ORIENTÉ OBJET

Attributs

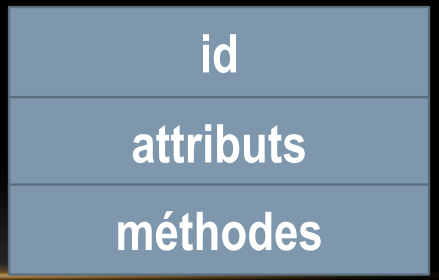
- Ce sont les données qui caractérisent l'objet. Ou des variables qui indiquent l'état de l'objet dans le temps.
- Ils présentent en plus de l'identité, la partie statique d'une classe



CONCEPTS DE L'ORIENTÉ OBJET

Méthodes

- Elles définissent le comportement (behavior) d'un objet, c-à-d l'ensemble des opérations que l'objet est chargé de réaliser dans un logiciel.
- Ces opérations permettent de faire réagir l'objet aux sollicitations extérieurs, ou d'agir sur les autres objets
- Les opérations sont liées aux attributs, car leurs actions peuvent dépondre des valeurs d'attributs ou les modifier.
- L'approche OO associe les données et les traitements au seins de la même classe.



CONCEPTS DE L'ORIENTÉ OBJET

Méthode abstraite

- C'est une opération de la classe avec uniquement une signature et sans corps.
- La signature d'une méthode abstraite décrit le nom, le type de retour, les argument et leurs types

CONCEPTS DE L'ORIENTÉ OBJET

Classe abstraite

- C'est une classe qui possède au moins une méthode abstraite.
- Une classe abstrait *n'est pas instanciable*
- elle représente un modèle de classe comme les interfaces; mais au contraire des interfaces, les classes abstraite peuvent avoir des attributs

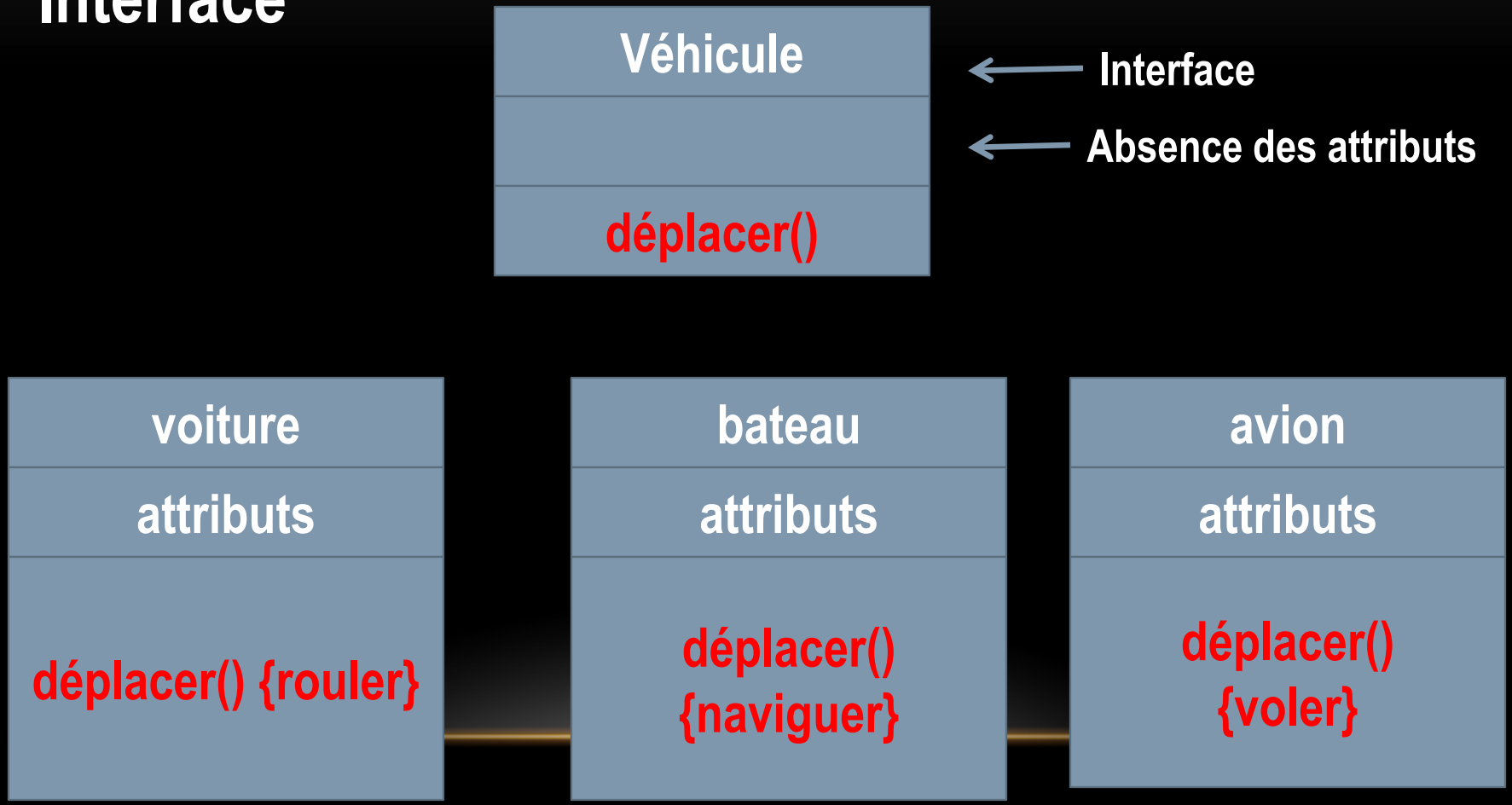
CONCEPTS DE L'ORIENTÉ OBJET

Interface

- C'est un modèle de classe, permettant de rassembler les classes par leur comportements seulement
- Les classes qui implémente une interface, définissent les méthodes abstraite, chacune à sa façon.

CONCEPTS DE L'ORIENTÉ OBJET

Interface



CONCEPTS DE L'ORIENTÉ OBJET

Encapsulation

- Elle consiste à masquer les détails de l'implémentation d'un objet, en définissant des modifications de visibilité à ses attributs et ses méthodes.
- Elle garantit l'intégrité et la sécurité d'accès aux données, car elle interdit l'accès direct aux caractéristiques des objets.

CONCEPTS DE L'ORIENTÉ OBJET

Encapsulation

- Elle est assurée par les modifications de visibilité suivante:

accessibilité	public	private	protected	Package freindly
À l'intérieur de la classe	OUI	OUI	OUI	OUI
Classes du même package	OUI	NON	OUI	OUI
Classes dérivées	OUI	NON	OUI	NON
Classes hors du package	OUI	NON	NON	NON

CONCEPTS DE L'ORIENTÉ OBJET

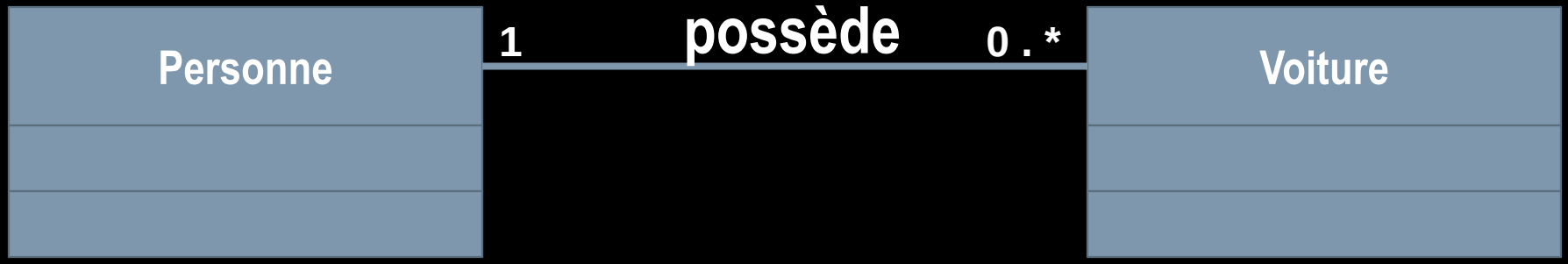
Association

- Une association normale est une ***connexion*** entre classes
- elle possède un ***nom***
- elle est généralement ***navigable*** de manière bidirectionnelle (dans ce cas elle a un nom dans les deux sens si nécessaire)
- elle a une ***multiplicité***
- elle peut être dotée de ***rôles***

CONCEPTS DE L'ORIENTÉ OBJET

Cardinalité (Multiplicité)

- Elle indique le nombre d'instances (objets) d'une classe associées à **une** instance de l'autre classe



CONCEPTS DE L'ORIENTÉ OBJET

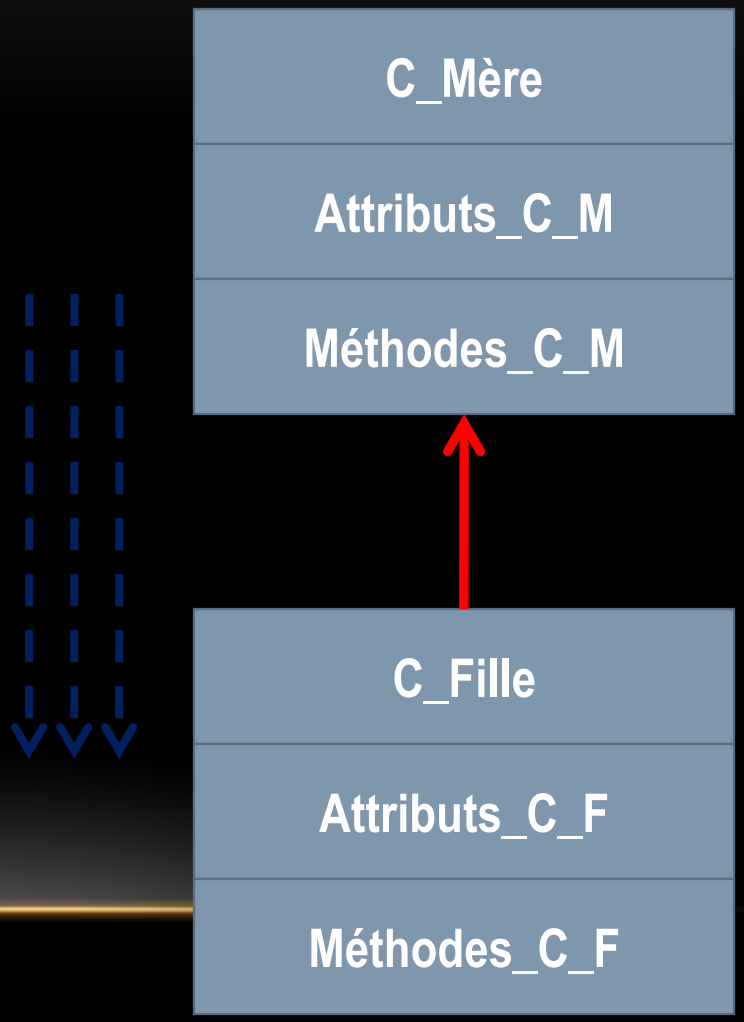
Héritage

- C'est un concept de passage des caractéristiques de la classe **mère** (ses attributs et méthodes) vers la classe **fille** à condition que les contraintes d'encapsulation le permettent (public, protected).
- Une classe peut être dérivable en d'autres classes , à fin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines (spécialisation).
- Plusieurs classes peuvent être « généralisées » en une classe qui les factorise, à fin de regrouper leurs caractéristiques communes (généralisation).
- L'héritage peut être simple ou multiple.
- L'héritage multiple n'est pas supporté par JAVA et il est supporté par C++

CONCEPTS DE L'ORIENTÉ OBJET

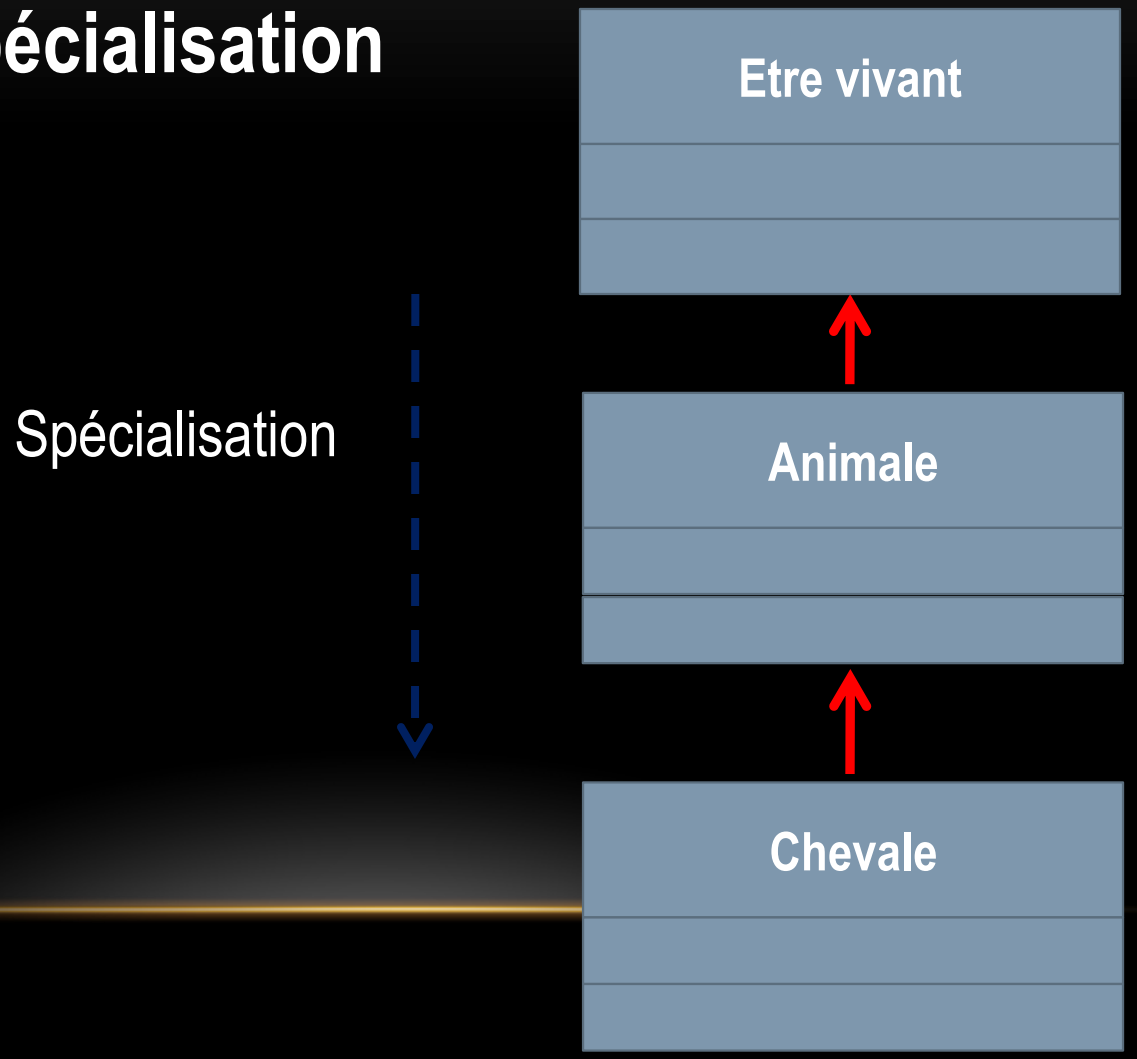
Héritage

Passage des attributs et des méthodes de la classe mère vers la classe fille



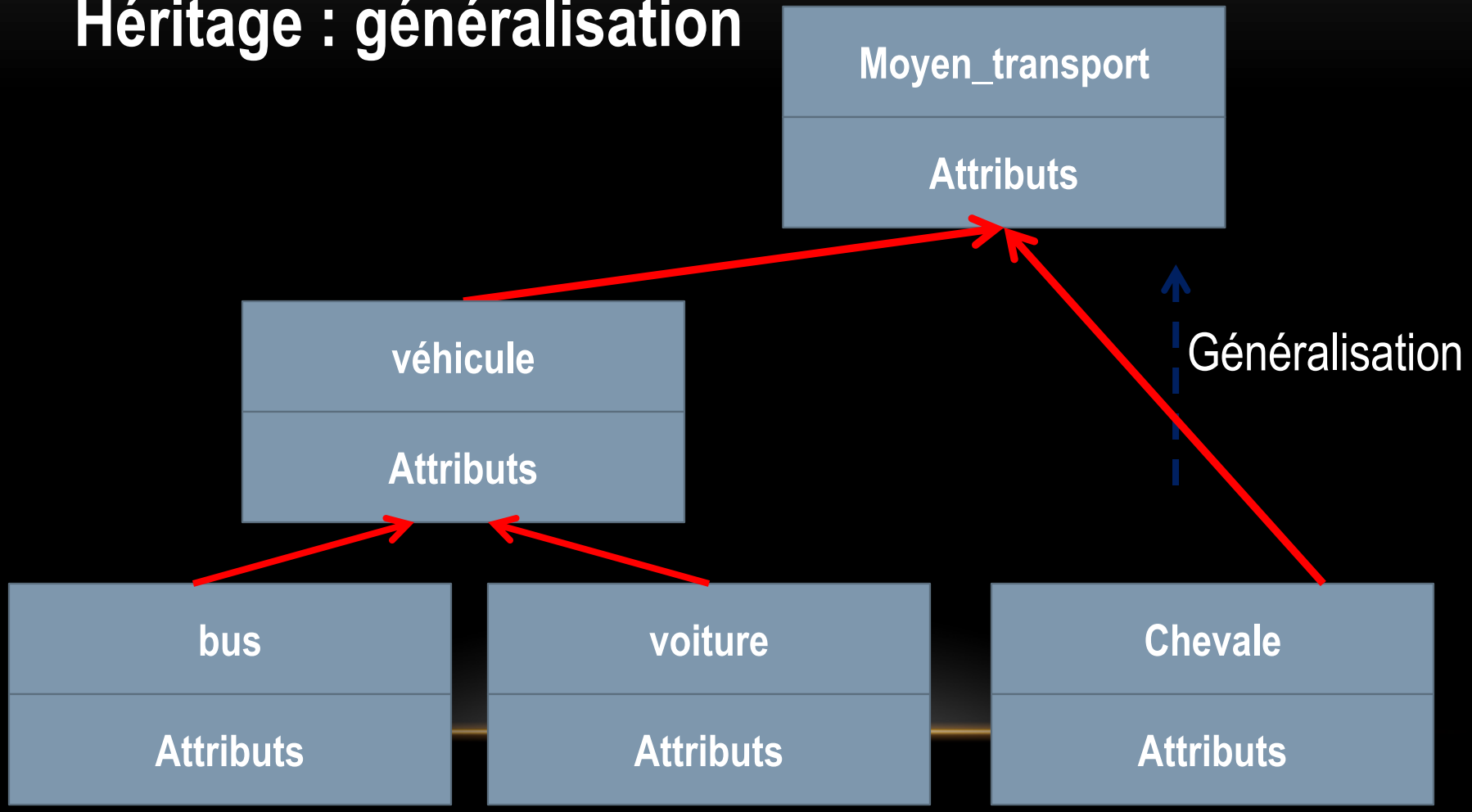
CONCEPTS DE L'ORIENTÉ OBJET

Héritage : spécialisation



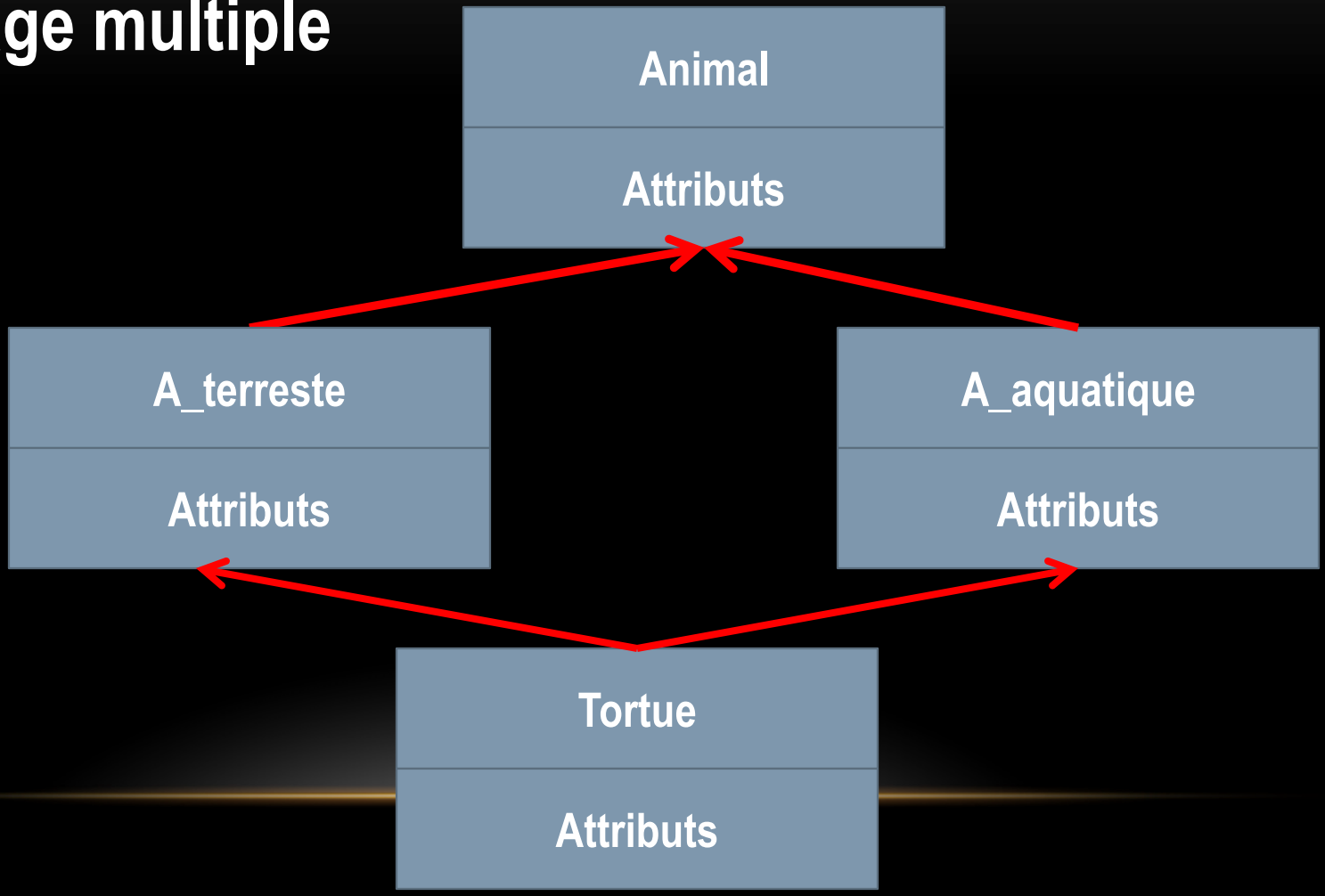
CONCEPTS DE L'ORIENTÉ OBJET

Héritage : généralisation



CONCEPTS DE L'ORIENTÉ OBJET

Héritage multiple



CONCEPTS DE L'ORIENTÉ OBJET

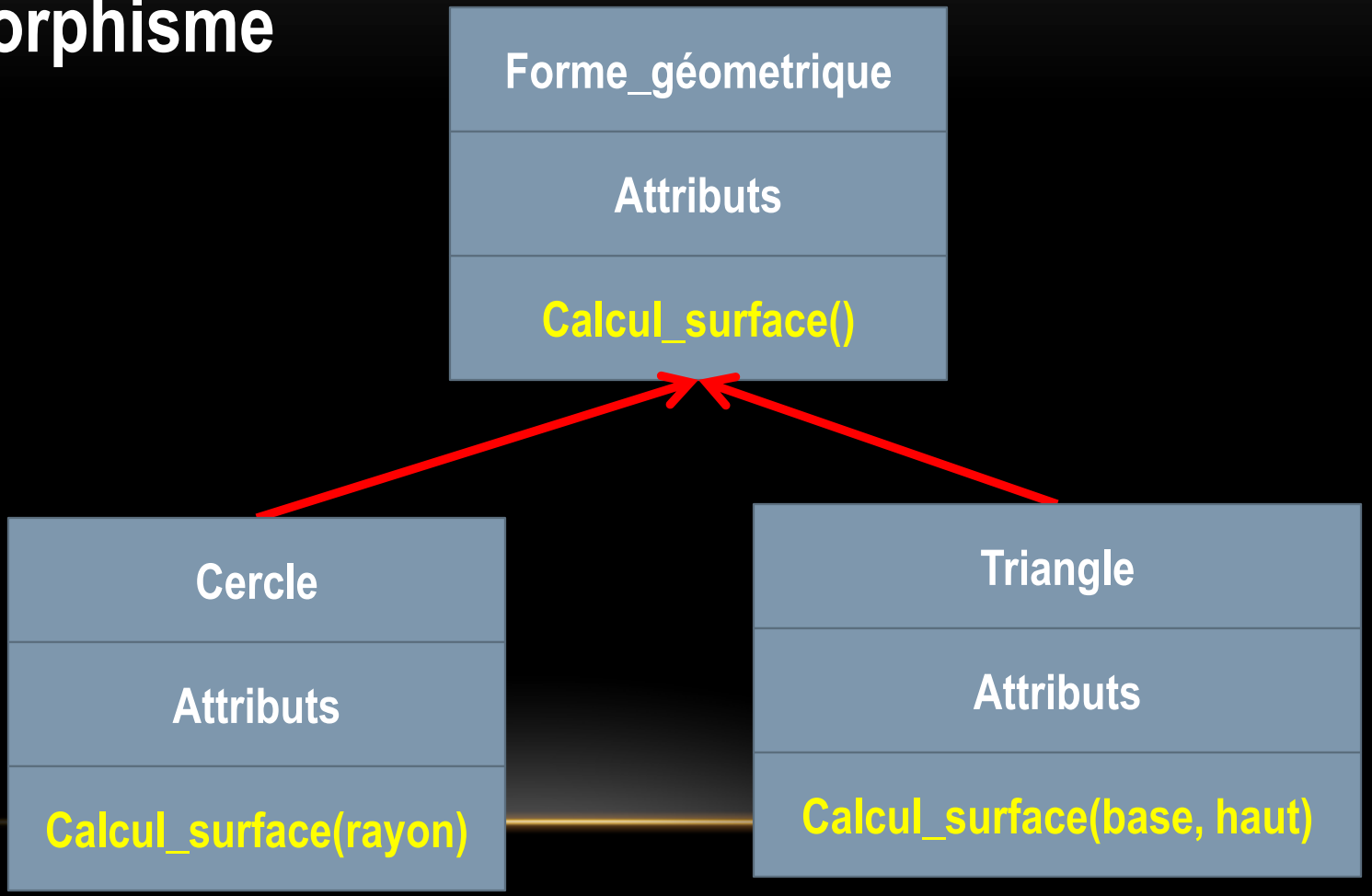
Polymorphisme

Grèc --> Poly : différents; morphisme : forme

- Le concept qui fournit la possibilité d'utiliser la même méthode dans un programme, mais avec des traitements différents, grâce à l'héritage et la redéfinition de la méthode polymorphe dans les classes dérivées

CONCEPTS DE L'ORIENTÉ OBJET

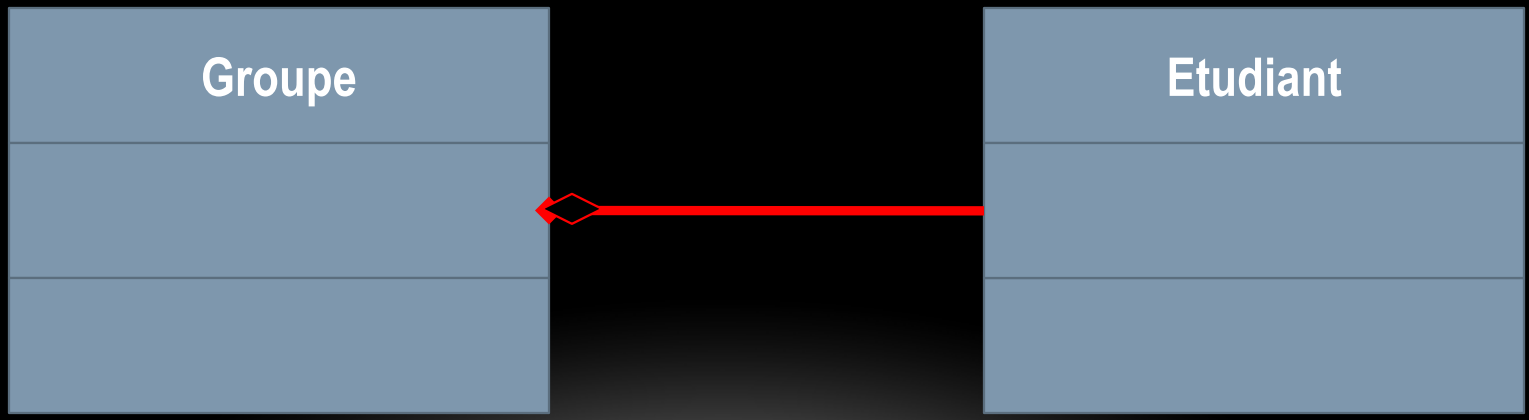
Polymorphisme



CONCEPTS DE L'ORIENTÉ OBJET

Agrégation

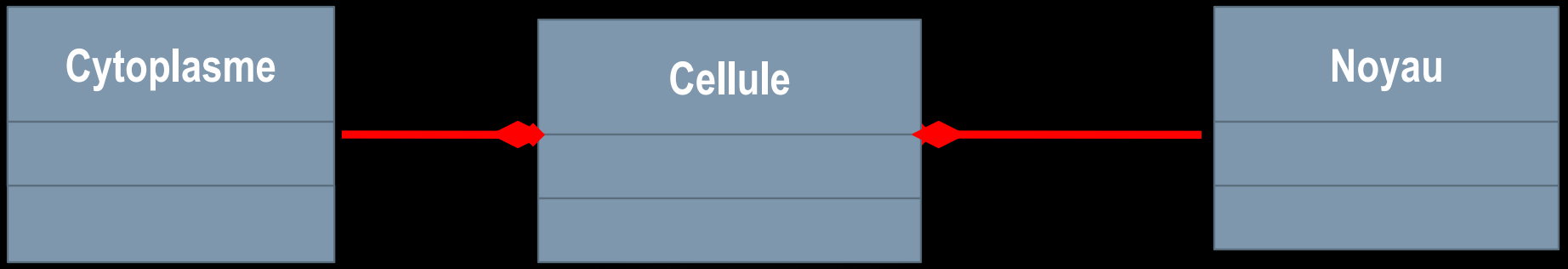
- Il s'agit d'une relation d'inclusion entre deux classes, spécifiant que les objets d'une classe (agrégat) sont des composants (agrégés)de l'autre classe composite.



CONCEPTS DE L'ORIENTÉ OBJET

Composition

- C'est une agrégation forte : la classe composite et les classes composants ont la même durée de vie



PRÉSENTATION DE JAVA

PRÉSENTATION DE JAVA

PHASES DE DEVELOPPEMENT D'UN PROGRAMME JAVA

Phase 1: Edition

- Editeur
  - + vi et emacs sous UNIX
  - + Bloc-notes sous WINDOWS
  - + Environnements de Développement Intégrés (EDI): JBuilder de Borland, NetBeans, Visual Cafe de Symantec, Visual J++ de Microsoft
- Le nom de fichier d'un programme Java se termine toujours par l'extension .java.
- Exemple: Programme.java

PRÉSENTATION DE JAVA

PHASES DE DEVELOPPEMENT D'UN PROGRAMME JAVA

Phase 2: Compilation

- La commande du compilateur Java pour compiler un programme Java et le traduire en byte codes, est javac.

- La compilation génère un fichier possédant le même nom que la classe et contenant les bytes codes avec l'extension .class. Le compilateur génère un fichier compilé pour chaque classe.

Ainsi, si le fichier source contient plusieurs classes, alors plusieurs fichiers ont l'extension .class.

- Exemple: javac Programme.java génère un fichier Programme.class

Mettre l'extension à la suite du nom en respectant la casse du nom de fichier.

Java est sensible à la casse.

PRÉSENTATION DE JAVA

PHASES DE DEVELOPPEMENT D'UN PROGRAMME JAVA

Phase 3: Chargement

- Le chargeur de classe prend le ou les fichiers .class et les transfère en mémoire centrale
- Le fichier .class peut être chargé à partir d'un disque dur de sa propre machine ou à travers un réseau
- 2 types de fichier .class peuvent être chargés: les applications (programmes exécutés sur sa propre machine) et les applets (programmes stockés sur une machine distante et chargés dans le navigateur Web).

- Une application peut être chargée et exécutée par la commande de l'interpréteur Java

java

- Exemple: java Programme

Pas d'extension .class à la suite du nom.

PRÉSENTATION DE JAVA

PHASES DE DEVELOPPEMENT D'UN PROGRAMME JAVA

Phase 4: Vérification

Les byte codes dans une applet sont vérifiés par le vérificateur de byte codes avant leur exécution par l'interpréteur Java intégré au navigateur ou à l'appletviewer. Ce vérificateur vérifie que les byte codes sont conformes aux restrictions de sécurité de Java concernant les fichiers et la machine.

PRÉSENTATION DE JAVA

PHASES DE DEVELOPPEMENT D'UN PROGRAMME JAVA

Phase 5: Exécution

L'ordinateur interprète le programme byte code par byte code.

Les interpréteurs présentent des avantages sur les compilateurs dans le monde Java. En effet, un programme interprété peut commencer immédiatement son exécution dès qu'il a été téléchargé sur la machine cliente, alors qu'un programme source devant subir une compilation supplémentaire entraînerait un délai de compilation avant de pouvoir démarrer son exécution.

Cependant, dans des applets à forte charge de calcul, l'applet doit être compilé pour augmenter la rapidité d'exécution.

GÉNÉRALITÉS

GÉNÉRALITÉS

PROGRAMME ECRITURE CONSOLE

- **Problème: Ecriture d'un texte dans une fenêtre console**
- Fichier: Ecriture.java

```
public class Ecriture
{
    public static void main(String[] args)
    {
     System.out.println("Un programme Java");
    }
}
```

GÉNÉRALITÉS

PROGRAMME ECRITURE CONSOLE

- **Problème: Ecriture d'un texte dans une fenêtre console**
- Fichier: Ecriture.java

```
public class Ecriture
{
        public static void main(String[] args)
        {
         System.out.println("Un programme Java");
        }

}
```

- Exécution:

```
Un programme Java
```

GÉNÉRALITÉS

PROGRAMME ECRITURE FENETRE

- Problème: Ecriture d'un texte dans une fenêtre graphique
- Fichier: EcritureFenêtre.java

```
import javax.swing.*;
public class EcritureFenêtre
{
        public static void main(String[] args)
        {
          JOptionPane.showMessageDialog(null,"Fenêtre Java");
        }
}
```

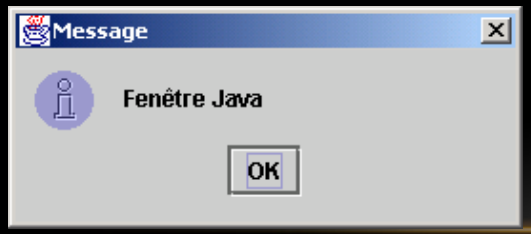
GÉNÉRALITÉS

PROGRAMME ECRITURE FENETRE

- Problème: Ecriture d'un texte dans une fenêtre graphique
- Fichier: EcritureFenêtre.java

```
import javax.swing.*;
public class EcritureFenêtre
{
        public static void main(String[] args)
        {
          JOptionPane.showMessageDialog(null,"Fenêtre
Java");
        }
}
```

Exécution :



GÉNÉRALITÉS

PROGRAMME ECRITURE FENETRE

- Problème: Ecriture d'un texte dans une Applet
- Fichier: EcritureApplet.java

```
import java.awt.*;
import javax.swing.*;

public class EcritureApplet extends JApplet
{
  public void paint(Graphics g)
  {
    g.drawString("APPLET JAVA", 100, 100);
  }
}
```

GÉNÉRALITÉS

PROGRAMME ECRITURE FENETRE

- Problème: Ecriture d'un texte dans une Applet
- Fichier: EcritureApplet.java
- Exécution :


```
import java.awt.*;
import javax.swing.*;

public class EcritureApplet extends JApplet
{
  public void paint(Graphics g)
  {
    g.drawString("APPLET JAVA", 100, 100);
  }
}
```

GÉNÉRALITÉS

PROGRAMME LECTURE

- Problème: Lecture d'un texte d'une fenêtre console
- Fichier: : Lecture.java

Lecture à partir du clavier 1/4

```
import java.io.*;

// Méthodes de lecture au clavier
public class Lecture
{
  // Lecture d'une chaîne
  public static String lireString()
  {
    String ligne_lue = null;
    {
      InputStreamReader lecteur = new InputStreamReader(System.in);
      BufferedReader entree = new BufferedReader(lecteur);
      ligne_lue = entree.readLine();

    try
    }
    catch (IOException err)
    {
      System.exit(0);
    }
    return ligne_lue;
  }
```

Lecture à partir du clavier 2/4

```
// Lecture d'un réel double
 public static double lireDouble()
 {
   double x = 0;
     String ligne_lue = lireString();
     x = Double.parseDouble(ligne_lue);
   try
   {
   }
   catch (NumberFormatException err)
   {
     System.out.println("Erreur de donnée");
     System.exit(0) ;
   }
   return x;
 }
```

Lecture à partir du clavier 3/4

```
// Lecture d'un entier
public static int lireInt()
{
  int n = 0;

  try
  {
    String ligne_lue = lireString();
    n = Integer.parseInt(ligne_lue);
  }
  catch (NumberFormatException err)
  {
    System.out.println ("Erreur de donnée");
    System.exit(0);
  }
  return n;
}
```

Lecture à partir du clavier 4/4

```
// Programme test de la classe Lecture
public static void main (String[] args)
{
  System.out.print("Donner un double:  ");
  double x;
  x = Lecture.lireDouble();
  System.out.println("Résultat " + x);
  System.out.print("Donner un entier: « )
  int n;
  n = Lecture.lireInt();
  System.out.println("Résultat " + n);
 }
}
```

Exécution
Donner un double: 10.01
Résultat 10.01
Donner un entier: 10
Résultat 10

GÉNÉRALITÉS

REGLES GENERALES D'ECRITURE

Identificateurs

Dans un langage de programmation, un identificateur est une suite de caractères servant à désigner les différentes entités manipulées par un programme : variables, fonctions, classes, objets...

GÉNÉRALITÉS

REGLES GENERALES D'ECRITURE

Identificateurs

Par convention

- **Les identificateurs de classes commencent toujours par une majuscule.**
- **Les identificateurs de variables et de méthodes commencent toujours par une minuscule.**
- **Les identificateurs formés par la concaténation de plusieurs mots, comporte une majuscule à chaque début de mot sauf pour le 1er mot qui dépend du type d'identificateur.**
- **Exemple: public class ClasseNouvelle**

GÉNÉRALITÉS

REGLES GENERALES D'ECRITURE

Les mots clés

Certains mots-clés sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identficateurs.

En voici la liste, par ordre alphabétique :

abstract assert boolean break byte
case catch char class const
continue default do double else
extends final finally float for
goto if implements import instanceof
int interface long native new
null package private protected public
return short static super switch
synchronized this throw throws transient
try void volatile while

GÉNÉRALITÉS

REGLES GENERALES D'ECRITURE

Les séparateurs

- Dans notre langue écrite, les mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne. Il en va quasiment de même en Java.

Ainsi, on ne pourra pas remplacer :

```
int  x,y;
```

par

```
intx,y;
```

En revanche, vous pourrez écrire indifféremment :

```
int n,compte,p,total,valeur ;
```

ou, plus lisiblement :

```
int n, compte, p, total, valeur ;
```

voire :

```
int n,
compte,
p,
total,
valeur ;
```

GÉNÉRALITÉS

REGLES GENERALES D'ECRITURE

Les séparateurs : le format libre

```
// Calcul de racines carrees
// La classe Racines utilise la classe Clavier
public class Racines1  { public
static void main (String[] args)  { final int NFOIS = 5 ;  int i ; double
  x ;    double racx ;    System.out.println ( "Bonjour" ) ;  System.out.println ("Je vais vous calculer " +
  NFOIS  + " racines carrees") ;    for (i=0 ; i < NFOIS ; i++)  { System.out.print ("Donnez un nombre : ") ;
x  = Clavier.lireDouble  ()  ;  if (x < 0.0)  System.out.println (x + " ne possede pas de racine carree")
; else  { racx = Math.sqrt(x) ;  System.out.println (x + " a pour racine carree : "  + racx) ; } }
System.out.println("Travail termine - Au revoir") ;   }}
```

Les séparateurs : le format libre (écriture correcte)

```
// Calcul de racines carrees
// La classe Racines utilise la classe Clavier
public class Racines
{ public static void main (String[] args)
  { final int NFOIS = 5 ;
    int i ;
    double x ;
    double racx ;
    System.out.println ("Bonjour") ;
    System.out.println ("Je vais vous calculer " + NFOIS + " racines carrees") ;
    for (i=0 ; i<NFOIS ; i++)
      { System.out.print ("Donnez un nombre : ") ;
        x  = Clavier.lireDouble ()  ;
 if (x < 0.0)
          System.out.println (x + " ne possede pas de racine carree") ;
         else
          { racx = Math.sqrt(x) ;
            System.out.println (x + " a pour racine carree : "  + racx) ;
          }
      }
    System.out.println ("Travail termine - Au revoir") ;
  }
}
```

GÉNÉRALITÉS

REGLES GENERALES D'ECRITURE

Commentaires

3 formes

Commentaire usuel pouvant s'étendre sur plusieurs lignes: /* … */

Commentaire de fin de ligne s'arrêtant en fin de la ligne: //

Commentaire de documentation pouvant être extraits automatiquement par des programmes utilitaires de création de documentation tels que Javadoc qui génère automatiquement une documentation en format HTML: /** … */

GÉNÉRALITÉS

REGLES GENERALES D'ECRITURE

Commentaires

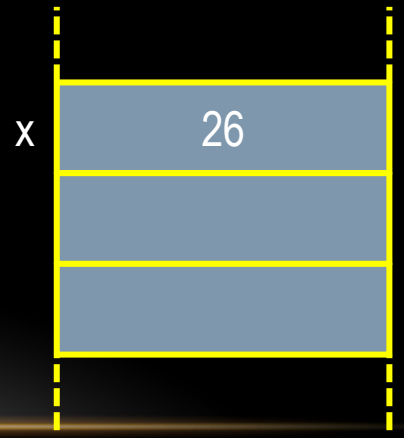
```
/* programme de calcul de racines carrees */
/* commentaire s'etendant
   sur plusieurs lignes
   de programme
*/
/* ================================================== *
 *              UN TITRE MIS EN VALEUR                *
 * ================================================== */
int i ;         /* compteur de boucle */
float x;        /* nombre dont on cherche la racine */
float racx ;    /* pour la racine carre de x */
```

LES TYPES PRIMITIFS DE JAVA

LES TYPES PRIMITIFS DE JAVA

Notion de type

- Dans tout programme, il est nécessaire de stocker, au moins provisoirement, des valeurs, de provenances variées (données d'entrée, résultats intermédiaires, ...)
- Le type d'une valeur permet de différencier la nature de l'information stockée dans une variable.



LES TYPES PRIMITIFS DE JAVA

TYPE BOOLEEN

Déclaration d'une variable booléenne

```
boolean test ;
```

Une variable booléenne prend 2 valeurs: false et true.

Affectation d'une variable booléenne

```
test = (n < m) ;
```

LES TYPES PRIMITIFS DE JAVA

TYPE ENTIER

- Le type entier permet de représenter de façon exacte une partie des nombres entiers relatifs.

Type	Taille (octet)	Valeur minimale	Valeur maximale
byte	1	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	2	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	4	-2 147 483 648 (Integer.MIN_VALUE)	2 147 483 647 (Integer.MAX_VALUE)
long	8	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)

Par défaut: une constante entière est de type int.

LES TYPES PRIMITIFS DE JAVA

TYPE REEL

- Le type réel permet de représenter de façon approchée une partie des nombres réels.

Type7	Taille (octet)	Précision (chiffres significatifs)	Valeur minimale	Valeur maximale
float	4	7	1.402E-45 Float.MIN_VALUE	3.402E38 Float.MAX_VALUE
double	8	15	4.94E-324 Double.MIN_VALUE	1.79E308 Double.MAX_VALUE

Par défaut: une constante réelle est de type double.

LES TYPES PRIMITIFS DE JAVA

TYPE REEL

- Problème: Ecriture d'une variable réelle en utilisant un formatage
- Fichier: EcritureReel.java

```
import java.awt.*;
public class EcritureReel
{
  public static void main(String[] args)
  {
    double  x = 10.123456789;
    System.out.println("x= " + x);
    // au moins 1 chiffre à gauche du point décimal
    // 2 chiffres exactement à droite du point décimal
    DecimalFormat  deuxDecimal = new DecimalFormat("0.00");
    System.out.println("x= " + deuxDecimal.format(x));
  }
}
```

Exécution
x= 10.123456789
x= 10,12

LES TYPES PRIMITIFS DE JAVA

TYPE CARACTERE

Le caractère en Java est représenté en mémoire sur 2 octets en utilisant le code Unicode.

Déclaration d'une variable caractère

```
char c;
```

Une constante caractère est notée entre apostrophe.

Exemple: 'a'

LES TYPES PRIMITIFS DE JAVA

Constantes et expressions constantes

Le mot-clé final

```
final int n = 20 ;
n = n + 5 ;                 // erreur : n a été déclarée final
n = Clavier.lireInt() ;   // idem
```

EXERCICES

1. Ecrire un programme qui permet de permuter deux nombres
2. Ecrire un programme qui lit le prix HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant. Faire en sorte que des libellés apparaissent clairement.

LES INSTRUCTIONS DE CONTRÔLE DE JAVA

LES INSTRUCTIONS DE CONTRÔLE DE JAVA

INSTRUCTION if

if (expression_booléenne) instruction_1

[else instruction_2]

LES INSTRUCTIONS DE CONTRÔLE DE JAVA

INSTRUCTION switch

```
switch (expression)
{
  case constante_1: [ suite_instructions_1 ]
  .....................
  case constante_n: [ suite_instructions_n ]
  [ default: suite_instructions ]
}
```

LES INSTRUCTIONS DE CONTRÔLE DE JAVA

INSTRUCTION while

while (expression_booléenne) instruction

INSTRUCTION do while

do instruction while (expression_booléenne);

LES INSTRUCTIONS DE CONTRÔLE DE JAVA

INSTRUCTION for

for ([initialisation]; [expression_booléenne] ; [incrémentations]) instruction

```
public class InstructionFor
{
  public static void main (String args[])
  {
    for (int i=1, j=1; (i <= 5); i++, j+=i)
     {
       System.out.println ("i= " + i + "   j= " + j);
     }
  }
}
```

EXERCICES

1- Ecrire le programme qui affiche le mois correspondant à un nombre compris entre 1 et 12 .

2- Traduire l'algorithme suivant en un programme JAVA

```
Algorithme DeuxChiffres ;
        constante (STOP : entier) ← 99999 ;
        variables uneVal, : entier ;
début
        ecrire ("Entrer un nombre,  ", STOP, " pour finir. ") ;
        lire (uneVal) ;
        tant que uneVal ≠ STOP faire
                si uneVal ≥ 10 et uneVal < 100
                alors ecrire ( uneVal, " est un nombre à deux chiffres. " ) ;
                finsi
        ecrire ("Entrer un autre nombre, " , STOP, " pour finir. ") ;
        lire (uneVal) ;
        fintantque
        ecrire (" fin de l'algorithme ") ;
fin
```

EXERCICES

3- Ecrire le programme de résolution d'une équation du second ordre dans l'ensemble C:

$$aX^2 + bX + c = 0$$

a, b et c sont lus à partir du clavier.

4- Écrire un programme qui détermine le factoriel d'un entier N en utilisant les boucles du type : Pour, Tant que et Répéter.

5- Écrire un algorithme qui calcule la somme des nombres entrés par l'utilisateur tant que cette somme est inférieure à mille.

EXERCICES

6- Ecrire le programme qui calcule la moyenne de N notes.

7- Ecrire un programme calculant la somme des n premiers termes de la "série harmonique", c'est-à-dire la somme :

1 + 1/2 + 1/3 + 1/4 + ... + 1/n

La valeur de n sera lue en donnée

LES CLASSES ET LES OBJETS

LES CLASSES ET LES OBJETS

- La notion de classe généralise la notion de type. La classe comporte des champs (données) et des méthodes.
- La notion d'objet généralise la notion de variable.
- Un type classe donné permet de créer (d'instancier) un ou plusieurs objets du type, chaque objet comportant son propre jeu de données.

LES CLASSES ET LES OBJETS

Définition d'une classe

- Nous nous proposons de définir une classe nommée Point ,destinée à manipuler les points d'un plan.

```
public class Point
{

    // instructions de définition des champs et des méthodes de la classe

}
```

LES CLASSES ET LES OBJETS

Définition des champs : attributs

- Nous supposons qu'un objet de type Point sera représenté par deux coordonnées entières :

```
public class Point
{

        private int x ;    // abscisse
        private int y ;    // ordonnée

}
```

LES CLASSES ET LES OBJETS

Définition des méthodes

```
public class Point
{
        public void initialise (int abs, int ord)
                {       x = abs ;
                        y = ord ;
                }
        public void deplace (int dx, int dy)
                {       x += dx ;
                        y += dy ;
                }
        public void affiche ()
        { System.out.println ("Je suis un point de coordonnées " + x + " " + y) ;
        }

    private int x ;    // abscisse
    private int y ;    // ordonnée
}
```

LES CLASSES ET LES OBJETS

Utilisation de la classe

```
public class TestPoint
{
        public static void main (String args[])
        {       Point a ; // a est un handle (une référence sur un objet)
                 a = new Point() ;
                a.initialise(3, 5) ;   a.affiche() ;
                a.deplace(2, 0) ;      a.affiche() ;
                 Point b = new Point() ;
                b.initialise (6, 8) ;  b.affiche() ;
        }
}
```

Exécution ➔

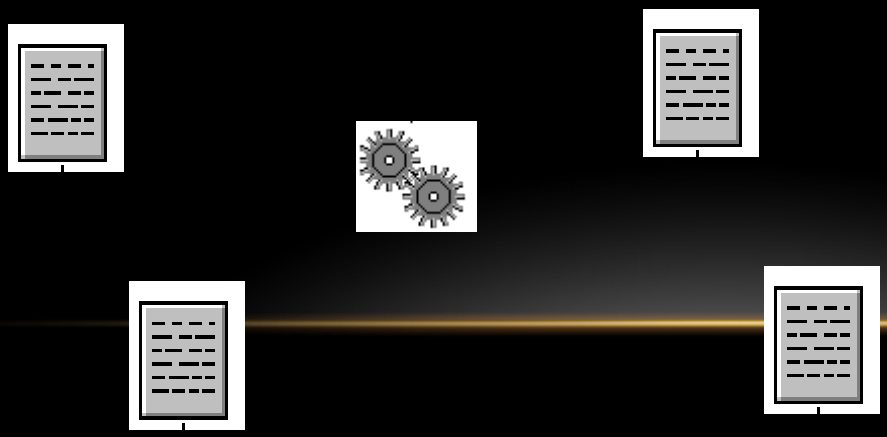
```
Je suis un point de coordonnées 3 5
Je suis un point de coordonnées 5 5
Je suis un point de coordonnées 6 8
```

LES CLASSES ET LES OBJETS

Mise en œuvre d'un programme comportant plusieurs classes

Un fichier source par classe

- Chaque classe est mise dans un fichier à part dans le même dossier :
  - Point.java et TestPoint.java
  - Compiler les fichier Point.java ➔ javac Point.java ➔ point.class
  - Compiler le fichier Testpoint.java ➔ javac TestPoint.java ➔ TestPoint.class
  - Lancer le fichier TestPoint.class ➔ java TestPoint



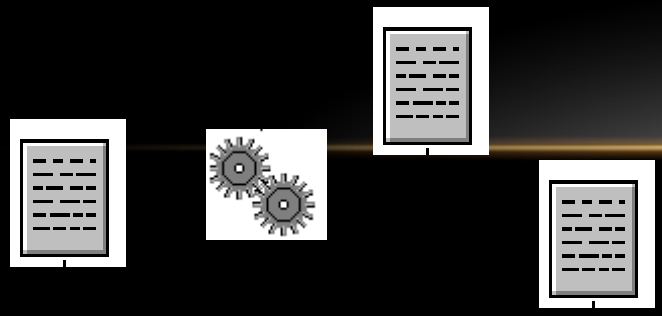
Plusieurs classes dans un seul fichier source

Accessibilité paquetage

Accessibilité JVM

```
class Point
{  public void initialise (int abs, int ord)
                        { x = abs ; y = ord ;
                         }
  public void deplace (int dx, int dy)
                         { x += dx ; y += dy ;
                        }
            public void affiche ()
       { System.out.println ("coordonnées " + x + " " + y) ;
            }
    private int x ;    // abscisse
    private int y ;    // ordonnée
}
```

```
public class TestPoint
{
            public static void main (String args[])
            {           Point a ;
                         a = new Point() ;
                        a.initialise(3, 5) ;   a.affiche() ;
                        a.deplace(2, 0) ;      a.affiche() ;
                         Point b = new Point() ;
                        b.initialise (6, 8) ;  b.affiche() ;
            }
}
```



LA NOTION DE CONSTRUCTEUR

Généralités

```
public class Point
{
        public void initialise (int abs, int ord)
            {       x = abs ;
                    y = ord ;
             }
        ……….
        ……..
}
```

Il est nécessaire de recourir à la méthode initialise pour attribuer des valeurs aux champs d'un objet de type Point.

La notion de constructeur permet d'automatiser le mécanisme d'initialisation d'un objet.

- Un constructeur n'est rien d'autre qu'une méthode, sans valeur de retour, portant le même nom que la classe.
- Un constructeur peut disposer d'un nombre quelconque d'arguments (éventuellement aucun).

LA NOTION DE CONSTRUCTEUR

Exemple de classe comportant un constructeur

```
public class Point
{
        public Point ( int  abs,  int  ord)
            {          x = abs ;
                       y = ord ;
             }
        ……….
        ……..
}
```

Comment utiliser cette classe ?

\\ Une instruction telle que :

Point a = new Point() ; \\ est refusée par le compilateur.

Ici, notre constructeur a besoin de deux arguments. Ceux-ci doivent obligatoirement être fournis lors de la création, par exemple :

Point a = new Point(1, 3) ;

LA NOTION DE CONSTRUCTEUR

Exemple de classe comportant un constructeur

```
public class Point
{
        public Point ( int  abs,  int  ord)
                {          x = abs ;
                           y = ord ;
                }
        ..................
        ..................
}
```

```
public class TstPnt3
{ public static void main (String args[ ])
  { Point a ;
    a = new Point(3, 5) ;
    a.affiche() ;
    a.deplace(2, 0) ;
    a.affiche() ;
    Point b = new Point(6, 8) ;
    b.affiche() ;
  }
}
```

LA NOTION DE CONSTRUCTEUR

Quelques règles concernant les constructeurs

1. Un constructeur ne fournit aucune valeur. Dans son en-tête, aucun type ne doit figurer devant son nom. Même la présence (logique) de void est une erreur

```
class Test
{ .....
  public void Test ()   // erreur de compilation : void interdit ici
  { .....
  }
}
```

2. Une classe peut ne disposer d'aucun constructeur

```
Point a =  new Point() ;   // OK si Point n'a pas de constructeur
```

LA NOTION DE CONSTRUCTEUR

Quelques règles concernant les constructeurs

4. Un constructeur ne peut pas être appelé directement depuis une autre méthode.
   Par exemple, si Point dispose d'un constructeur à deux arguments de type int:

```
Point a = new Point (3, 5) ;
   .....
a.Point(8, 3) ;  // interdit
```

LA NOTION DE CONSTRUCTEUR

Construction et initialisation d'un objet

la création d'un objet entraîne toujours, par ordre chronologique, les opérations suivantes :

- une initialisation par défaut de tous les champs de l'objet,
- une initialisation explicite lors de la déclaration du champ,
- l'exécution des instructions du corps du constructeur.

LA NOTION DE CONSTRUCTEUR

Construction et initialisation d'un objet

Initialisation par défaut des champs d'un objet

Type du champ	Valeur par défaut
boolean	false
char	caractère de code nul
entier (byte, short, int, long)	0
flottant (float, double)	0.0 ou 0.
objet	null

LA NOTION DE CONSTRUCTEUR

Construction et initialisation d'un objet

Initialisation explicite des champs d'un objet

```
class A
{  public A (...) { ..... }   //  constructeur de A
      .....
   private int n = 10 ;
   private int p ;
}
```

L'instruction suivante :

A a = new A (...) ;

entraîne successivement :

- l'initialisation (implicite) des champs n et p de a à 0,
- l'initialisation (explicite) du champ n à la valeur figurant dans sa déclaration, soit 10,
- l'exécution des instructions du constructeur.

LA NOTION DE CONSTRUCTEUR

Construction et initialisation d'un objet

```
public class Init
{ public static void main (String args[])
  { A a = new A() ;
    a.affiche() ;
  }
}
```

```
class A
{ public A()
  { np = n * p ;
    n = 5 ;
  }
  public void affiche()
  { System.out.println ("n = " + n + ", p = " + p + ", np = " + np) ;
  }
  private int n = 20, p = 10 ;
  private int np ;
}
```

Appel du constructeur

Donnez les valeurs affichées :

n = 5 , p = 10 , np = 200

TYPOLOGIE DES MÉTHODES D'UNE CLASSE

Parmi les différentes méthodes que comporte une classe, on a souvent tendance à distinguer :

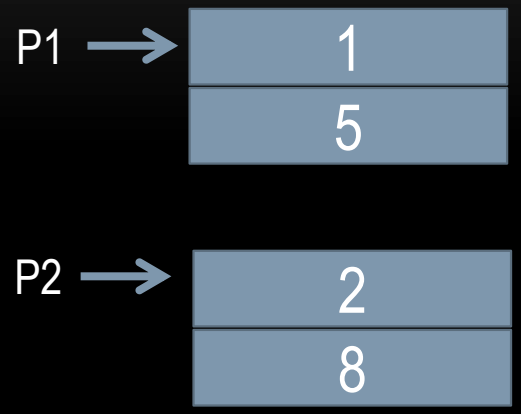
- les constructeurs ;
- les méthodes d'accès (en anglais accessor) qui fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses champs (généralement privés), sans les modifier ;
- les méthodes d'altération (en anglais mutator) qui modifient l'état d'un objet, donc les valeurs de certains de ses champs.

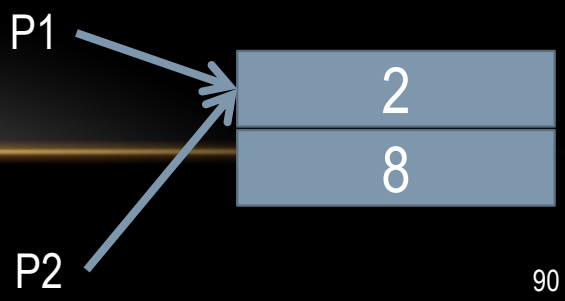
```
public int getX { return x ; }          // getX ou encore getAbscisse
public int getY { return y ; }          // getY ou encore getOrdonnee
public void setX (int abs) { x = abs ; }   // setX ou encore setAbscisse
public void setY (int ord) { x = ord ; }   // setY ou encore setOrdonnee
public void setPosition (int abs, int ord) { x = abs ; y = ord ; }
```

AFFECTATION ET COMPARAISON D'OBJETS

Exemple 1

```
// Soient 2 variables P1 et P2 de type point
Point  P1, P2;
//Soit 2 objets P1 et P2 de type point
P1 = new point(1, 5);
P2 = new point(2, 8);
//L'instruction
P1 = P2 ;
/* affecte à P1 la référence de P2.
Ainsi, P1 et P2 désigne le même objet point(2, 8) et non pas 2 objets de même valeur. */
```





RAMASSE-MIETTES

Garbage Collector

```
Point A;
A = new point(8,12);    // objet avec référence

(new Point(3,5)).affiche() ;    // Objet sans référence
```

candidat au ramasse-miettes

En Java, il n'existe aucun opérateur permettant de détruire un objet dont on n'aurait plus besoin.

En fait, la démarche employée par Java est un mécanisme de gestion automatique de la mémoire connu sous le nom de ramassse-miettes (en anglais Garbage Collector).

On peut activer le garbage collector par l'instruction :

```
System.gc();
```

RÈGLES D'ÉCRITURE DES MÉTHODES

Méthodes ne fournissant aucun résultat

Méthode avec le mot clé void dans son en-tête.
Méthode appelée: objet.méthode(liste arguments).

Méthodes fonction fournissant un résultat

```
public class Point
{           int getX { return x ; }
            int getY { return y ; }
              double distance ()
                        {  double d ;
                           d = Math.sqrt (x*x* + y*y) ;
                           return d ;
                        }
            private int x, y ;
            ..............
}
```

RÈGLES D'ÉCRITURE DES MÉTHODES

Utilisation d'une fonction fournissant un résultat

```
Point a = new Point(...) ;

double u, r ;

   .....

u = 2. * a.distance() ;

r = Math.sqrt( a.getX() * a.getX() + a.getY() * a.getY() );
```

RÈGLES D'ÉCRITURE DES MÉTHODES

Les arguments d'une méthode

les arguments figurant dans l'en-tête de la définition d'une méthode se nomment arguments muets (ou encore arguments ou paramètres formels)

```
void f (final int n, double x)
{  .....
   n = 12 ;    // erreur de compilation
   x = 2.5 ;   // OK
   .....
}
```

Les arguments fournis lors de l'appel de la méthode portent quant à eux le nom d' arguments effectifs (ou encore paramètres effectifs).

```
Point p = new Point(...) ;
int n1, n2 ; byte b ; long q ;
p.deplace (n1, n2) ;     // OK : appel normal
p.deplace (b+3, n1) ;    // OK : b+3 est déjà de type int
p.deplace (b, n1) ;      // OK : b de type byte sera converti en int
p.deplace (n1, q) ;      // erreur : q de type long ne peut être converti en int
p.deplace (n1, (int)q) ; // OK
```

RÈGLES D'ÉCRITURE DES MÉTHODES

Propriétés des variables locales

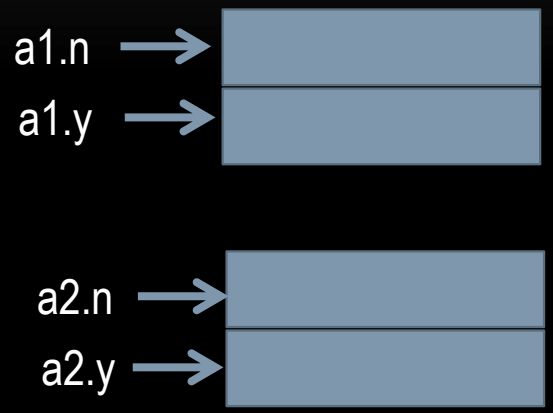
```
void F(int n)
{ float x ;      // variable locale à F
  float n ;      // interdit en Java
  .....
}
void G ()
{ double x ;    // variable locale à G, indépendante de x locale à F
    .....
}
```

RÈGLES D'ÉCRITURE DES MÉTHODES

CHAMPS STATIQUES (Champs de classe)

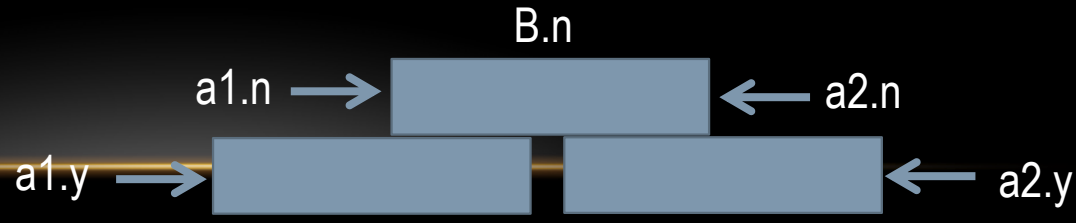
```
class A
{ int n ;
  float y ;
}
```

A a1 = new A(), a2 = new A() ;



Les champs statiques n'existent qu'en un seul exemplaire, indépendamment de tout objet de la classe.

```
class B
{ static int n ;
  float y ;
}
B a1 = new B(), a2 = new B() ;
```



RÈGLES D'ÉCRITURE DES MÉTHODES

CHAMPS STATIQUES

Exemple

```
class Obj {
public Obj() { System.out.print ("++ creation objet Obj ; ") ;
    nb ++ ;
    System.out.println ("il y en a maintenant " + nb) ;
  }
  private static long nb=0 ;
}
public class TstObj
{ public static void main (String args[])
  { Obj a ;
    System.out.println ("Main 1") ;
    a = new Obj() ;
    System.out.println ("Main 2") ;
    Obj b ;
    System.out.println ("Main 3") ;
    b = new Obj() ;
    Obj c = new Obj() ;
    System.out.println ("Main 4") ;
  }
}
```

Exécution

```
Main 1
++ creation objet Obj ; il y en a maintenant 1
Main 2
Main 3
++ creation objet Obj ; il y en a maintenant 2
++ creation objet Obj ; il y en a maintenant 3
Main 4
```

RÈGLES D'ÉCRITURE DES MÉTHODES

METHODES STATIQUES (Méthodes de classe)

une méthode de classe ne pourra en aucun cas agir sur des champs usuels (non statiques)

```
class A
{   .....
  private float x ;        // champ usuel
  private static int n ;   // champ de classe
    .....
  public static void f()   // méthode de classe
  { .....     // ici, on ne peut pas accéder à x, champ usuel,
    .....     //  mais on peut accéder au champ de classe n
  }
}
   .....
A a ;
A.f() ;    // appelle la méthode de classe f de la classe A
a.f() ;    // reste autorisé, mais déconseillé
```

METHODES STATIQUES (Méthodes de classe)

Exemple

```
class Obj
{ public Obj()
  { System.out.print ("++ creation objet Obj ; ") ;
    nb ++ ;
    System.out.println ("il y en a maintenant " + nb) ;
  }
  public static long nbObj ()
  { return nb ;  }
  private static long nb=0 ;
}
public class TstObj2
{ public static void main (String args[])
  { Obj a ;
    System.out.println ("Main 1 : nb objets = " + Obj.nbObj() ) ;
    a = new Obj() ;
    System.out.println ("Main 2 : nb objets = " + Obj.nbObj() ) ;
    Obj b ;
    System.out.println ("Main 3 : nb objets = " + Obj.nbObj() ) ;
    b = new Obj() ;
    Obj c = new Obj() ;
    System.out.println ("Main 4 : nb objets = " + Obj.nbObj() ) ;
  }
}
```

```
Main 1 : nb objets = 0
++ creation objet Obj ; il y en a maintenant 1
Main 2 : nb objets = 1
Main 3 : nb objets = 1
++ creation objet Obj ; il y en a maintenant 2
++ creation objet Obj ; il y en a maintenant 3
Main 4 : nb objets = 3
```

RÈGLES D'ÉCRITURE DES MÉTHODES

initialisation des champs statiques

```
class A
{    .....
   public static void f() ;
     .....
   private static int n = 10 ;
   private static int p ;
}
```

```
A a ;  // aucun objet de type A n'est encore créé, les champs statiques
       // de A sont intialisés : p (implicitement) à 0, n (explicitement) à 10
A.f() ;    // initialisation des statiques de A, si pas déjà fait
```

Bloc d'initialisation statique

```
class A
{ private static int t[] ;
  .....
  static { .....
           int nEl = Clavier.lireInt() ;
           t = new int[nEl] ;
           for (int i=0 ; i<nEl ; i++) t[i] = i ;
         }
  .....
}
```

SURDEFINITION DE MÉTHODES

```
class Point
{ public Point (int abs, int ord)  // constructeur
  { x = abs ;  y = ord ;
  }
public void deplace (int dx, int dy)  // deplace  (int, int)
  { x += dx ; y += dy ;
  }
  public void deplace (int dx)          // deplace  (int)
  { x += dx ;
  }
  public void deplace (short dx)        // deplace  (short)
  { x += dx ;
  }
  private int x, y ;
}
```

Rq :

Le type de la valeur de retour d'une méthode n'intervient pas dans le choix d'une méthode surdéfinie

```
public class Surdef1
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    a.deplace (1, 3) ; // appelle deplace (int, int)
    a.deplace (2) ;    // appelle deplace (int)
    short p = 3 ;
    a.deplace (p) ;    // appelle deplace (short)
    byte b = 2 ;
    a.deplace (b) ;    // appelle deplace (short) apres
//conversion de b en short
  }
}
```

SURDEFINITION DE MÉTHODES

CAS DU CONSTRUCTEUR

```
class Point
{
  public Point ()          // constructeur 1 (sans argument)
  { x = 0 ;  y = 0 ;
  }
public Point (int abs)          // constructeur 2 (un argument)
  { x = y = abs ;
  }
  public Point (int abs, int ord ) // constructeur 3 (deux arguments)
  { x = abs ; y = ord ;
  }
  public void affiche ()
  { System.out.println ("Coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
public class Surdef2
{ public static void main (String args[])
  { Point a = new Point () ;    // appelle constructeur 1
    a.affiche() ;
    Point b = new Point (5) ;   // appelle constructeur 2
    b.affiche() ;
    Point c = new Point (3, 9) ;  // appelle constructeur 3
    c.affiche() ;
  }
}
```

```
Coordonnees : 0 0
Coordonnees : 5 5
Coordonnees : 3 9
```

ECHANGE D'INFORMATIONS AVEC LES MÉTHODES

```
class Util
{ public static void Echange (int a, int b)  // ne pas oublier static
  { System.out.println ("début Echange : " + a + " " + b) ;
    int c ;
    c = a ; a = b ; b = c ;
    System.out.println ("fin Echange   : " + a + " " + b) ;
  }
}
public class Echange
{ public static void main (String args[])
  { int n = 10, p = 20 ;
    System.out.println ("avant appel   : " + n + " " + p) ;
    Util.Echange (n, p) ;
    System.out.println ("après appel   : " + n + " " + p) ;
  }
}
```

```
avant appel   : 10 20
début Echange : 10 20
fin Echange   : 20 10
après appel   : 10 20
```

la transmission des types primitifs, se fait par valeur.

ECHANGE D'INFORMATIONS AVEC LES MÉTHODES

Cas des objets transmis en argument

Problème :

Ecrire un programme qui vérifie si deux points coïncident.

```
class Point
{ public Point(int abs, int ord)
  { x = abs ; y = ord ;  }
  public boolean coincide (Point pt)
  { return ((pt.x == x) && (pt.y == y)) ;
  }
  private int x, y ;
}
public class Coincide
{ public static void main (String args[])
  { Point a = new Point (1, 3) ;
    Point b = new Point (2, 5) ;
    Point c = new Point (1,3) ;
    System.out.println ("a et b : " + a.coincide(b) + " " + b.coincide(a)) ;
    System.out.println ("a et c : " + a.coincide(c) + " " + c.coincide(a)) ;
  }
}
```

```
a et b : false false
a et c : true true
```

On dit que l'unité d'encapsulation est la classe non l'objet

ECHANGE D'INFORMATIONS AVEC LES MÉTHODES

Cas des objets transmis en argument

Problème :

Ecrire un programme qui vérifie si deux points coïncident.

```
class Point
{ public Point(int abs, int ord)
  { x = abs ; y = ord ;  }
  pu
  {
  }
  pr
}
public class Coincide
{ public static void main (String args[])
  { Point a = new Point (1, 3) ;
    Point b = new Point (2, 5) ;
    Point c = new Point (1,3) ;
    System.out.println ("a et b : " + a.coincide(b) + " " + b.coincide(a)) ;
    System.out.println ("a et c : " + a.coincide(c) + " " + c.coincide(a)) ;
  }
}
```

Bien entendu, lorsqu'une méthode d'une classe T reçoit en argument un objet de classe T', différente de T, elle n'a pas accès aux champs ou méthodes privées de cet objet.

On dit que l'unité d'encapsulation est la classe non l'objet

ECHANGE D'INFORMATIONS AVEC LES MÉTHODES

Cas des objets transmis en argument

Conséquences de la transmission de la référence d'un objet

```
class Point
{
  public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }
  public void permute (Point a)  // methode d'Echange les coordonnées du point courant  avec
a
  { Point c = new Point(0,0) ;
    c.x = a.x ; c.y = a.y ;  // copie de a dans c
    a.x = x ;   a.y = y ;    // copie du point courant dans a
    x = c.x ;   y = c.y ;    // copie de c dans le point courant
  }
  public void affiche ()
  { System.out.println ("Coordonnées : " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
public class Permute
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    Point b = new Point (5, 6) ;
    a.affiche() ; b.affiche() ;
    a.permute (b) ;
    a.affiche() ; b.affiche() ;
  }
}
```

```
Coordonnées : 1 2
Coordonnées : 5 6
Coordonnées : 5 6
Coordonnées : 1 2
```

ECHANGE D'INFORMATIONS AVEC LES MÉTHODES

Cas de la valeur de retour

```
class Point
{
  public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }
  public Point symetrique()
  { Point res ;
    res = new Point (y, x) ;
    return res ;
  }
  public void affiche ()
  { System.out.println ("Coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
public class Sym
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    a.affiche() ;
    Point b = a.symetrique() ;
    b.affiche() ;
  }
}
```

Coordonnées : 1 2
Coordonnées : 2 1

Dans ce cas, la fonction fournit une copie de la référence à l'objet concerné.

ÉCHANGE D'INFORMATIONS AVEC LES MÉTHODES

Cas de la valeur de retour

```
class Point
{
public Point(int abs, int ord)
{ x = abs ; y = ord ;
}
public Point symetrique()
{ Point res ;
res = new Point (y, x) ;
return res ;
}
public void affiche ()
{ System.out.println ("Coordonnees : " + x + " " + y) ;
}
private int x, y ;
}
```

```
public class Sym
{ public static void main (String args[ ])
{ Point a = new Point (1, 2) ;
a.affiche() ;
Point b = a.symetrique() ;
b.affiche() ;
}
}
```

```
Coordonnees : 1 2
Coordonnees : 2 1
```

On reçoit toujours la copie de la valeur fournie par une méthode. Dans ce cas, elle fournit une copie de la référence à l'objet concerné (res).

ÉCHANGE D'INFORMATIONS AVEC LES MÉTHODES

Autoréférence : le mot-clé this

```
public boolean coincide (Point pt)
{ return ((pt.x == this.x) && (pt.y == this.y)) ;
}
```

```
public Point(int x, int y) // notez les noms des arguments muets ici
{ this.x = x ;       // ici x désigne le premier argument de Point
                     // le champ x de l'objet courant est masqué ; mais
                     // on peut le nommer this.x
this.y = x ;
}
```

EXERCICE 1

- Modéliser un cercle par les coordonnées de son centre et par son rayon.
- Prévoir un constructeur sans paramètres qui initialise tous les champs à 0.
- Prévoir un constructeur avec trois paramètres pour initialiser tous les attributs.
- Prévoir une méthode pour afficher les valeurs des attributs.
- Prévoir une méthode pour calculer le périmètre d'un cercle.
- Prévoir une méthode pour calculer la surface d'un cercle.
- Donner des exemples d'utilisation de cercle

Proposer une modélisation d'une autre forme géométrique!

Triangle, rectangle, losange, sph

```
class Cercle {
        public Cercle() {
                                x=0;
                                y=0;
                                r=0;
                        }
        public Cercle(int a, int b, int c) {
                                x=a;
                                y=b;
                                r=c;
                        }
public void affiche(){
System.out.println(" Mon centre se trouve à "+x+ " , "+y+" et mon rayon est de : "+r);}

public double perimetre(){ return (2*Pi*r);      }
      public double superficie(){   return (Pi*r*r);     }
      private int x;
      private int y;
      private int r;
      final double Pi = 3.14;
}
```

```
public class TestCercle {
    public static void main(String args []){
                    Cercle C1;
                    C1 = new Cercle(2,6,7);
                    C1.affiche();
                    System.out.println("perimetre = "+ C1.perimetre());
                     System.out.println("superficie = "+ C1.superficie());

    }
}
```

EXERCICE 2

- Modéliser un nombre complexe (Z=a+ ib)
- Prévoir un constructeur sans paramètres qui initialise la partie imaginaire et la partie réelle à 0.
- Prévoir un constructeur avec un seul paramètre qui initialise la partie imaginaire et la partie réelle à la même valeur.
- Prévoir un constructeur avec deux paramètres qui initialise la partie imaginaire et la partie réelle.
- Prévoir les accesseurs et les mutateurs.
- Prévoir une méthode qui renvoie le module d'un nombre complexe $|Z|=\sqrt{a^2 + b^2}$
- Dans le programme principal, créer des nombres complexes, et calculer
  - La somme de deux complexes,
  - Le produit de deux complexes

```
class Complexe {
        public Complexe(){
            a =0;
            b = 0;
        }
        public Complexe(double m){
            a =m;
            b = m;
        }
        public Complexe(double m, double n){
            a = m;
            b = n;
        }
        public void setA(double x)   {  a = x;          }
        public void setB(double x)   {  b = x;          }
        public double getA()                    {       return a;       }
        public double getB()                    {       return b;       }
        public double module()      { return (Math.sqrt(a*a + b*b));   }
        private double a;
        private double b;
}
```

```
public class TestComplexe {
    public static void main(String args [] ){
Complexe A = new Complexe();
                    A.setA(2.6);
                    A.setB(9.21);
                    System.out.println("Mon module est : "+A.module());
                    Complexe B= new Complexe(8);
                    Complexe Z = new Complexe();
                    Z.setA(A.getA()+B.getA());
                    Z.setB(A.getB()+B.getB());
                    System.out.println("Le module de Z est : "+Z.module());
                    // je vous laisse le soin de faire la multiplication


    }
}
```

EXERCICE 3

- Modéliser un élève qui est caractérisé par son nom, son âge sa note max sa note min et sa moyenne.
- Prévoir un constructeur avec paramètres qui initialise les attributs nom et âge et qui met la moyenne à 0.
- Prévoir les accesseurs et les mutateurs
- Prévoir une méthode ayant comme paramètre le nombre de matières, la fonction demande la note de chaque matière et initialise la note max la note min et la moyenne.
- Donner des exemples d'utilisation.

LA RÉCURSIVITÉ DES MÉTHODES

La récursivité directe

```
F1() {
    F1();
}
```

La récursivité croisée

```
F2() {
    F3();
}
```

```
F3() {
    F2();
}
```

Ecrire un programme qui permet de calculer la factorielle d'un nombre en utilisant une fonction récursive.

LA RÉCURSIVITÉ DES MÉTHODES

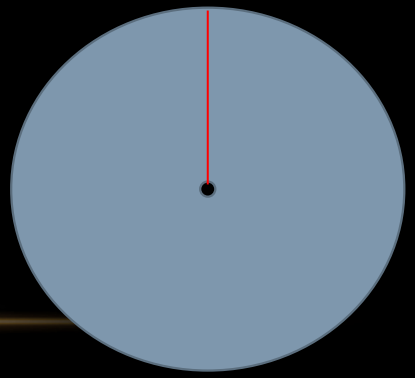
Ecrire un programme qui permet de calculer la factorielle d'un nombre en utilisant une fonction récursive.

```
class Util
{ public static long fac (long n)
{ if (n>1) return (fac(n-1) * n) ;
else return 1 ;
}
}
public class TstFac
{ public static void main (String [ ] args)
{ int n ;
System.out.print ("donnez un entier positif : ") ;
n = Clavier.lireInt() ;
System.out.println ("Voici sa factorielle : " + Util.fac(n) ) ;
}
}
```

LES OBJETS MEMBRES

Soit la classe point :

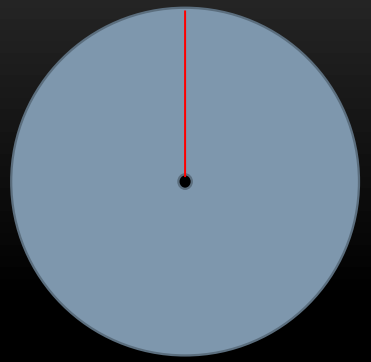
```
class Point
{ public Point(int x, int y) // constructeur
        { this.x = x ;
        this.y = y ;
        }
public void affiche()
{ System.out.println ("Je suis un point de coordonnées " + x + " " + y) ;
}
private int x, y ;
}
```



On souhaite modéliser un cercle par son centre et son rayon !!

LES OBJETS MEMBRES

```
class Point
{ public Point(int x, int y)
                { this.x = x ;
                this.y = y ;
                }
public void affiche()
{ System.out.println ("Je suis un point de coordonnées " + x + " " + y) ;
}
private int x, y ;
}
```

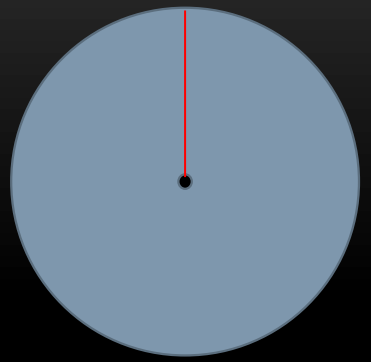


On souhaite modéliser un cercle par son centre et son rayon!!

```
class Cercle {
public Cercle (int x, int y, float r) { ..... } // constructeur
public void affiche() { ..... }
public void deplace (int dx, int dy) { ..... } //          //
private Point c ; // centre du cercle
private float r ; // rayon du cercle
}
```

LES OBJETS MEMBRES

```
class Cercle {
public Cercle (int x, int y, float r) { ..... } // constructeur
public void affiche() { ..... }
public void deplace (int dx, int dy) { ..... }
private Point c ; // centre du cercle
private float r ; // rayon du cercle
}
```

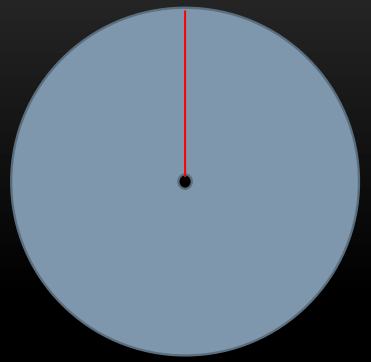


```
public Cercle (int x, int y, float r)
{           c = new Point (x, y) ;
            this.r = r ;
}

public void affiche()
{ System.out.println ("Je suis un cercle de rayon " + r ) ;
            System.out.print (" et de centre ") ;
            c.affiche() ;
}
```

LES OBJETS MEMBRES

```
class Cercle {
public Cercle (int x, int y, float r) { ..... } // constructeur
public void affiche() { ..... }
public void deplace (int dx, int dy) { ..... }
private Point c ; // centre du cercle
private float r ; // rayon du cercle
}
```



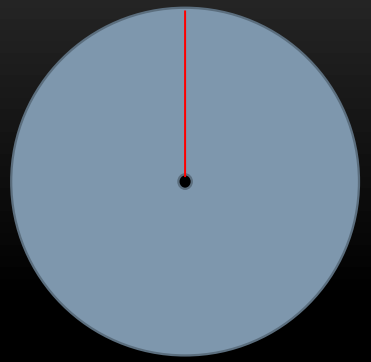
```
public Cercle (int x, int y, float r)
{                c = new Point (x, y) ;
                 this.r = r ;
}
```

```
public void affiche()
{ System.out.println ("Je suis un cercle de rayon " + r ) ;
                 System.out.print (" et de centre ") ;
                 c.affiche() ;
}
```

```
void deplace (int dx, int dy)
{ c.x += dx ; // x n'est pas un champ public de la classe Point ;
                    // on ne peut donc pas accéder à c.x
c.y += dy ; // idem
}
```

LES OBJETS MEMBRES

```
class Cercle {
public Cercle (int x, int y, float r) { ..... } // constructeur
public void affiche() { ..... }
public void deplace (int dx, int dy) { ..... }
private Point c ; // centre du cercle
private float r ; // rayon du cercle
}
```



```
public Cercle (int x, int y, float r)
{               c = new Point (x, y) ;
                this.r = r ;
}
```

```
public void affiche()
{ System.out.println ("Je suis un cercle de rayon " + r ) ;
                System.out.print (" et de centre ") ;
                c.affiche() ;
}
```

```
void deplace (int dx, int dy)
{ c.x += dx ; // x n'est pas un champ public de la classe Point ;
                        // on ne peut donc pas accéder à c.x
c.y += dy ; // idem
}
```

Pour pouvoir réaliser deplace(), il faudrait que la classe Point dispose :

- soit d'une méthode de déplacement d'un point,
- soit de méthodes d'accès et de méthodes d'altération. (les accesseurs)

LES OBJETS MEMBRES – SOLUTION-

```
class Point
{ public Point(int x, int y)
{ this.x = x ; this.y = y ;
}
public void affiche()
{ System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
}
public int getX() { return x ; }
public int getY() { return y ; }
public void setX (int x) { this.x = x ; }
public void setY (int y) { this.y = y ; }
private int x, y ;
}
```

LES OBJETS MEMBRES – SOLUTION- SUITE

```
class Cercle
{ public Cercle (int x, int y, float r)
{ c = new Point (x, y) ;
this.r = r ;
}
public void affiche()
{ System.out.println ("Je suis un cercle de rayon " + r) ;
System.out.println(" et de centre de coordonnées "
+ c.getX() + " " + c.getY()) ;
}
public void deplace (int dx, int dy)
{ c.setX (c.getX() + dx) ; c.setY (c.getY() + dy) ;
}
private Point c ; // centre du cercle
private float r ; // rayon du cercle
}
```

LES OBJETS MEMBRES – SOLUTION- SUITE

```
public class TstCerc
{ public static void main (String args[])
{ Point p = new Point (3, 5) ; p.affiche() ;
Cercle c = new Cercle (1, 2, 5.5) ; c.affiche();
}
}
```

Je suis un point de coordonnées 3 5
Je suis un cercle de rayon 5.5
et de centre de coordonnées 1 2

La situation d'objet membre correspond à ce qu'on nomme généralement la relation (appartenance).

LES CLASSES INTERNES

Imbrication de définitions de classe

Une classe est dite interne lorsque sa définition est située à l'intérieur de la définition d'une autre classe.

- La notion de classe interne correspond à cette situation

```
class E // définition d'une classe usuelle (dite alors externe)
{ ..... // méthodes et données de la classe E
class I // définition d'une classe interne à la classe E
{ ..... // méthodes et données de la classe I
}
..... // autres méthodes et données de la classe E
}
```

LES CLASSES INTERNES

Imbrication de définitions de classe

La définition de I est utilisable au sein de la définition de E,

```
class E
{ public void fe() // méthode de E
        { I i = new I() ; // création d'un objet de type I ; sa
                          //référence est  ici locale à la méthode fe
        }
        class I
        { ..... }
        .....
        private I i1, i2 ; // les champs i1 et i2 de E sont des
                                  //références à des objets de type I
}
```

LES CLASSES INTERNES

Imbrication de définitions de classe

Lien entre objet interne et objet externe

1. Un objet d'une classe interne est toujours associé, au moment de son instanciation, à un objet d'un classe externe dont on dit qu'il lui a donné naissance.

2. Un objet d'une classe interne a toujours accès aux champs et méthodes (même privés) de l'objet externe lui ayant donné naissance (attention : ici, il s'agit bien d'un accès restreint à l'objet, et non à tous les objets de cette classe).

3. Un objet de classe externe a toujours accès aux champs et méthodes (même privés) d'un objet d'une classe interne auquel il a donné naissance.

Exemple

```
class E
{ public void E()
{ i1 = new I() ; }
public void fe()
{ i2 = new I() ; }
public void g ()
{ ..... // ici, on peut accéder non seulement à i1 et i2,  mais aussi à i1.ni ou i2.ni
}
class I
{ .....
private int ni ;
}
private I i1, i2 ; // les champs i1 et i2 de E sont des références  à des objets de type I
}
.....
E e1 = new E() ; // ici, le constructeur de e1 crée un objet de type I associé à e1 et place sa
                 //référence dans e1.i1 (ici privé)
E e2 = new E() ; // ici, le constructeur de e2 crée un objet de type I associé à e1 et place sa
                 //référence dans e2.i1 (ici privé)
e1.fe() ; // la méthode fe crée un objet de type I associé à e1 et place sa référence dans e1.i2
```

LES PAQUETAGES

- La notion de paquetage correspond à un regroupement logique sous un identificateur commun d'un ensemble de classes. Elle est proche de la notion de bibliothèque que l'on rencontre dans d'autres langages (C ou C++).
- Elle facilite le développement et la cohabitation de logiciels conséquents en permettant de répartir les classes correspondantes dans différents paquetages.
- Le risque de créer deux classes de même nom se trouve alors limité aux seules classes d'un même paquetage.

LES PAQUETAGES

Attribution d'une classe à un paquetage

- Un paquetage est caractérisé par un nom qui est soit un simple identificateur, soit une suite d'identificateurs séparés par des points, comme dans :

MesClasses

Utilitaires.Mathematiques

Utilitaires.Tris

LES PAQUETAGES

Attribution d'une classe à un paquetage

- L'attribution d'un nom de paquetage se fait au niveau du fichier source ; toutes les classes d'un même fichier source appartiendront donc toujours à un même paquetage.
- Pour ce faire, on place, en début de fichier, une instruction de la forme :

```
package xxxxxx ;
```

- La notion de paquetage est une notion " logique "

Utilitaires.Tris ≠ Utilitaires.Mathematiques ≠ Utilitaires

LES PAQUETAGES

Utilisation d'une classe d'un paquetage

- Lorsque, dans un programme, vous faites référence à une classe, le compilateur la recherche dans le paquetage par défaut.

Pour ce faire, vous pouvez :

1. • citer le nom du paquetage avec le nom de la classe,
2. • utiliser une instruction import en y citant une classe particulière d'un paquetage,
3. • utiliser une instruction import en y citant tout un paquetage.

LES PAQUETAGES

Utilisation d'une classe d'un paquetage

1. En citant le nom de la classe

Si vous avez attribué à la classe Point le nom de paquetage MesClasses par exemple, vous pourrez l'utiliser simplement en la nommant MesClasses.Point

Par exemple :

```
MesClasses.Point p = new MesClasses.Point (2, 5) ;
```

- .....
- p.affiche() ; // ici, le nom de paquetage n'est pas requis

LES PAQUETAGES

Utilisation d'une classe d'un paquetage

2. En important une classe

- L'instruction import vous permet de citer le nom (complet) d'une ou plusieurs classes, par exemple :
- import MesClasses.Point, MesClasses.Cercle ;

LES PAQUETAGES

Utilisation d'une classe d'un paquetage

3. En important un paquetage

```
import MesClasses.* ;
```

De cette façon on peut utiliser toutes les classes du paquetage.

LES PAQUETAGES STANDARD

Les nombreuses classes standard avec lesquelles Java est fourni sont structurées en paquetages.

par exemple

java.awt, java.awt.event, javax.swing...

Par ailleurs, il existe un paquetage particulier nommé java.lang qui est automatiquement importé par le compilateur. C'est ce qui permet d'utiliser des classes standard telles que

Math, System, Float ou Integer, sans avoir à introduire d'instruction import.

.

PAQUETAGES ET DROITS D'ACCÈS

Droits d'accès aux classes

- Chaque classe dispose de ce qu'on nomme un droit d'accès (on dit aussi un modificateur d'accès). Il permet de décider quelles sont les autres classes qui peuvent l'utiliser.
- Il est simplement défini par la présence ou l'absence du mot-clé public
  - avec le mot-clé public, la classe est accessible à toutes les autres classes (moyennant éventuellement le recours à une instruction import) ;
  - sans le mot-clé public, la classe n'est accessible qu'aux classes du même paquetage.

PAQUETAGES ET DROITS D'ACCÈS

Droits d'accès aux membres d'une classe

- Pour un membre on peut utiliser (champ ou méthode) l'un des attributs public ou private.
- Avec public, le membre est accessible depuis l'extérieur de la classe ; avec private, il n'est accessible qu'aux méthodes de la classe.
- En fait, il existe une troisième possibilité, à savoir l'absence de mot-clé (private ou public). Dans ce cas, l'accès au membre est limité aux classes du même paquetage (on parle d'accès de paquetage).

LES TABLEAUX

LES TABLEAUX

Les tableaux sont considérés comme des objets

Ils fournissent des collections ordonnées d'éléments

Les éléments d'un tableau peuvent être

- Des variables d'un type primitif (int, boolean, double, char, …)
- Des références sur des objets

Création d'un tableau :

1 Déclaration = déterminer le type du tableau

2 Dimensionnement = déterminer la taille du tableau

3 Initialisation = initialiser chaque case du tableau

LES TABLEAUX

1) Déclaration

La déclaration précise simplement le type des éléments du tableau

```
int[] monTableau;
```

/*Peut s'écrire également*/ **int monTableau[];**

On déclare une variable pour un tableau de type int. Cette variable est une télécommande pour un objet tableau.

Pour le moment, la variable monTableau ne référence aucun objet, elle est donc null.

Attention : une déclaration de tableau ne doit pas préciser de dimensions

```
int monTableau[5]; // Erreur
```

LES TABLEAUX

2) Dimensionnement

Le nombre d'éléments du tableau sera déterminé quand l'objet tableau sera effectivement créé en utilisant le mot clé **new**

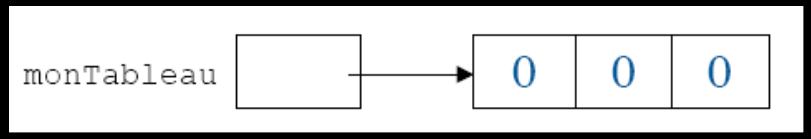
La taille déterminée à la création du tableau est fixe, elle ne pourra plus être modifiée par la suite

La création d'un tableau par **new :**

- Alloue la mémoire en fonction du type de tableau et de la taille
- Initialise le contenu du tableau à 0 pour les types simples

```
int[] monTableau;        // Déclaration
monTableau = new int[3];          // Dimensionnement
```

On crée un nouveau tableau d'entiers de 3 éléments et on l'affecte à la variable monTableau.



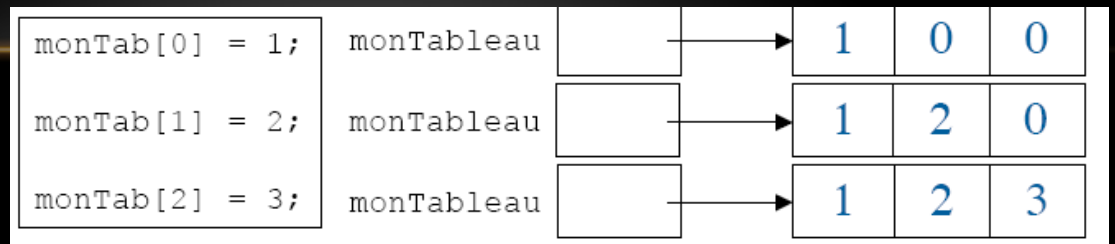
LES TABLEAUX

3) Initialisation

L'accès à un élément d'un tableau s'effectue suivant cette forme :

monTableau[varInt]; // 0 <= varInt <monTableau.length

- monTableau.length nous donne la taille du tableau, dans notre cas 3.
- Le premier élément du tableau commence toujours à l'indice 0, et le dernier élément du tableau correspond à l'indice tailleDuTableau – 1
- Java vérifie automatiquement l'indice lors de l'accès (pendant l'exécution du programme et non lors de la compilation)



EXERCICES

Exercice 1

Ecrire un programme qui permet de gérer un tableau; votre programme doit prévoir entre autre des fonctions qui permettent de :

- Lire le nombre d'éléments d 'un tableau
- Saisir les éléments du tableau
- Trier le tableau par l'ordre indiqué par l'utilisateur
- Retourner la plus grande valeur et la plus petite valeur du tableau
- Retourner l'indice d'une valeur indiquée par l'utilisateur
- Retourner le nombre d'occurrences d'une valeur indiquée par l'utilisateur

EXERCICES

Exercice 2

Ecrire un programme qui permet de modéliser une pile d'entier par une classe. Les éléments de la pile seront conservés dans un tableau. La classe comportera les fonctions membres suivantes:

- Un constructeur sans paramètre allouant un emplacement pour 20 entiers
- Un constructeur avec un seul paramètre allouant n emplacements
- Une fonction pour empiler un entier dans la pile [void empile(int);]
- Une fonction pour dépiler un entier de la pile [int depile( );] . Cette fonction retourne la valeur de l'élément situé en haut de la pile en le supprimant de la pile.
- Une fonction [boolean pleine( );] qui retourne true si la pile est pleine et false sinon.
- Une fonction [boolean vide( );] qui retourne true si la pile est vide et false sinon.

Donner un exemple d'instanciation et de manipulation de pile.

EXERCICES

Exercice 2

```
public class Pile {
                    public pile() { ….. .}
                    public pile (int n) {…….}
                    public void empile (int  e) {…….}
                    public int depile () {……..}
                    public boolean pleine() {……..}
                    public boolean vide() {……..}

                    private int nbr_elem;
                    …….
}
```

EXERCICES

Exercice 3

Reprendre l'exercice 2 avec une pile de points:

- Un constructeur sans paramètre allouant un emplacement pour 20 points
- Un constructeur avec un seul paramètre allouant n emplacements
- Une fonction pour empiler un point dans la pile [void empile(point);]
- Une fonction pour dépiler un point de la pile [point depile( );] . Cette fonction retourne la valeur de l'élément situé en haut de la pile en le supprimant de la pile.
- Une fonction [boolean pleine( );] qui retourne true si la pile est pleine et false sinon.
- Une fonction [boolean vide( );] qui retourne true si la pile est vide et false sinon.

Donner un exemple d'instanciation et de manipulation de pile de points.

EXERCICES

Exercice 3

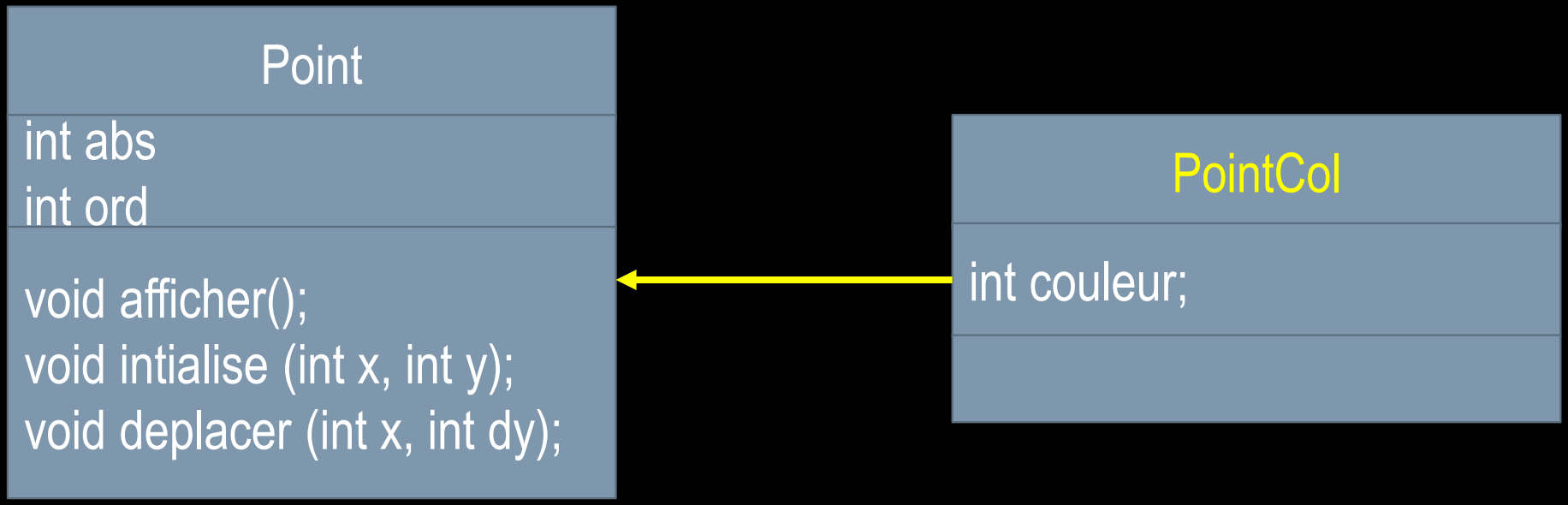
```
public class Pile {
                    public pile() { ….. .}
                    public pile (int n) {…….}
                    public void empile (point  p) {…….}
                    public point depile () {……}
                    public boolean pleine() {……}
                    public boolean vide() {……}

                    private int nbr_elem;
                    …….
}
```

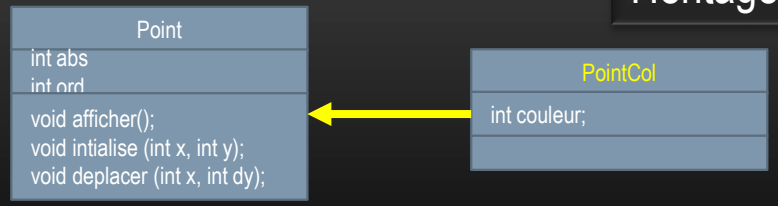
HÉRITAGE

NOTION D'HÉRITAGE

Imaginons le schéma suivant



NOTION D'HÉRITAGE

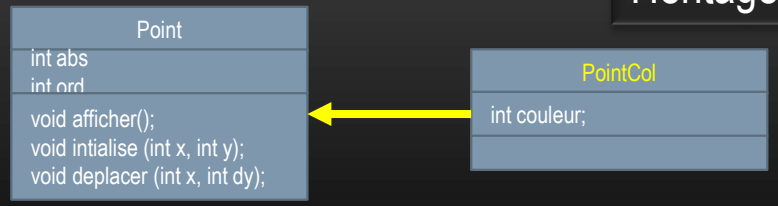


En Java

```
// classe de base
class Point
{ public void initialise (int abs, int ord)
{ x = abs ; y = ord ; }
public void deplace (int dx, int dy)
{ x += dx ; y += dy ; }
public void affiche ()
{ System.out.println ("Je suis en " + x + " " + y) ; }
private int x, y ;
}
```

```
// classe derivée de Point
class Pointcol extends Point
{ public void colore (byte couleur)
{ this.couleur = couleur ; }
private byte couleur ;
}
```

NOTION D'HÉRITAGE



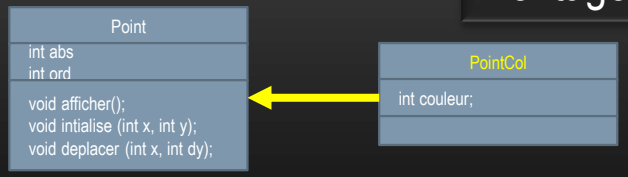
Utilisation des deux classes

```
// classe utilisant Pointcol
public class TstPcol1
{ public static void main (String args[])
{ Pointcol pc = new Pointcol() ;
pc.affiche() ;
pc.initialise (3, 5) ;
pc.colore ((byte)3) ;
pc.affiche() ;
pc.deplace (2, -1) ;
pc.affiche() ;
Point p = new Point() ; p.initialise (6, 9) ;
p.affiche() ;
}
}
```

un objet d'une classe dérivée accède aux membres publics de sa classe de base,

```
Je suis en 0 0
Je suis en 3 5
Je suis en 5 4
Je suis en 6 9
```

NOTION D'HÉRITAGE



- Accès d'une classe dérivée aux membres de sa classe de base

Une classe dérivée n'accède pas aux membres privés

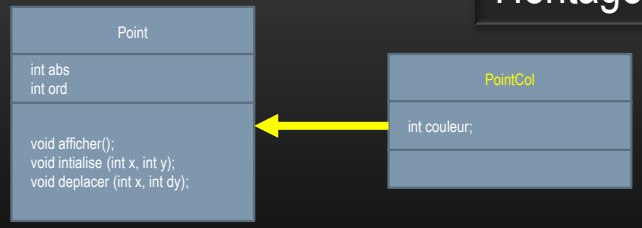
```
void affichec() // méthode affichant les coordonnees et la couleur
{ System.out.println ("Je suis en " + x + " " + y) ; // NON : x et y sont privés
System.out.println (" et ma couleur est : " + couleur) ;
}
```

Elle accède aux membres publics

```
public void affichec ()
{ affiche() ; // this.affiche();
System.out.println (" et ma couleur est : " + couleur) ;
}

public void initialisec (int x, int y, byte couleur)
{ initialise (x, y) ;
this.couleur = couleur ;
}
```

NOTION D'HÉRITAGE

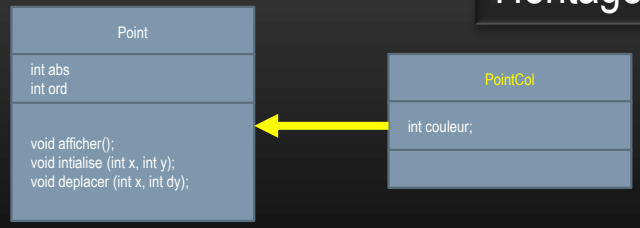


Construction et initialisation des objets dérivés

En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur et ce dernier est désigné par le mot-clé super.

NOTION D'HÉRITAGE

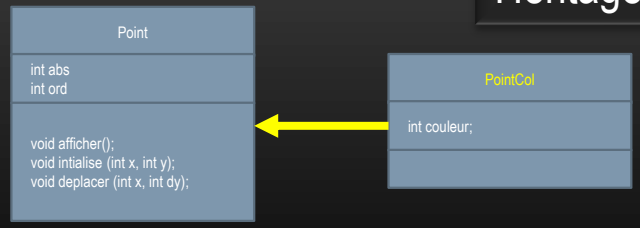


Construction et initialisation des objets dérivés

```
class Point
{ public Point (int x, int y)
{ this.x = x ; this.y = y ;
}
public void deplace (int dx, int dy)
{ x += dx ; y += dy ;
}
public void affiche ()
{ System.out.println ("Je suis en " +
}
private int x, y ;
}
```

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ; // obligatoirement comme première instruction
this.couleur = couleur ;
}
public void affichec ()
{ affiche() ;
System.out.println (" et ma couleur est : " + couleur) ;
}
private byte couleur ;
}
```

NOTION D'HÉRITAGE



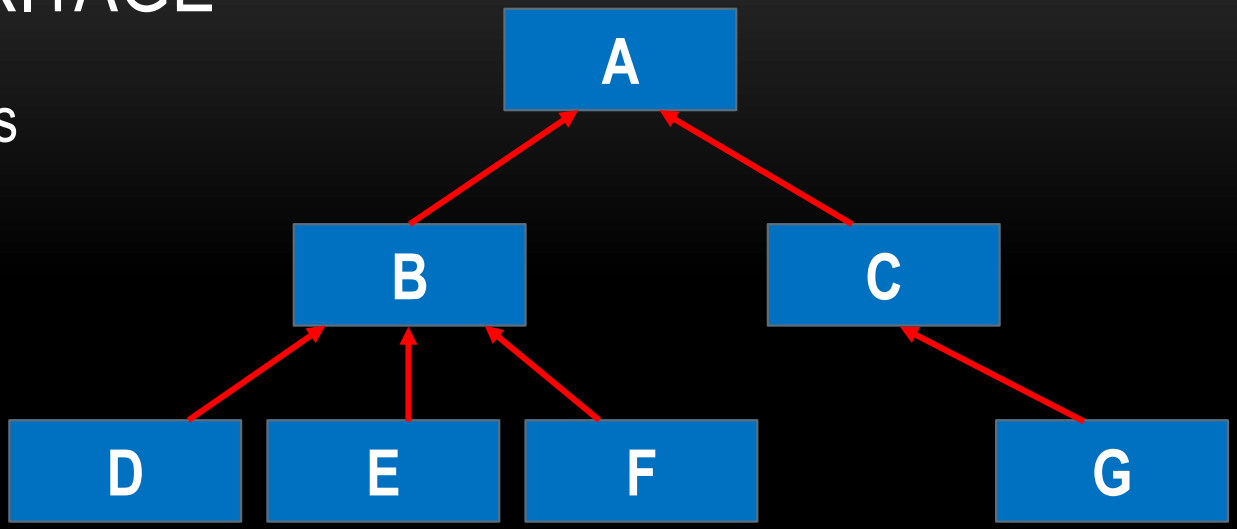
Construction et initialisation des objets dérivés

```
public class TstPcol3
{ public static void main (String args[])
{ Pointcol pc1 = new Pointcol(3, 5, (byte)3) ;
pc1.affiche() ; // attention, ici affiche
pc1.affichec() ; // et ici affichec
Pointcol pc2 = new Pointcol(5, 8, (byte)2) ;
pc2.affichec() ;
pc2.deplace (1, -3) ;
pc2.affichec() ;
}
}
```

```
Je suis en 3 5
Je suis en 3 5
et ma couleur est : 3
Je suis en 5 8
et ma couleur est : 2
Je suis en 6 5
et ma couleur est : 2
```

NOTION D'HÉRITAGE

Dérivations successives



- d'une même classe peuvent être dérivées plusieurs classes différentes,
- les notions de classe de base et de classe dérivée sont relatives puisqu'une classe dérivée peut, à son tour, servir de classe de base pour une autre.

NOTION D'HÉRITAGE

Redéfinition et surdéfinition de membres

La notion de redéfinition de méthode

```
class Point
{ public Point (int x, int y)
{ this.x = x ; this.y = y ;
}
public void deplace (int dx, int dy)
{ x += dx ; y += dy ; }
public void affiche ()
{ System.out.println ("Je suis en " + x + " " + y) ;
}
private int x, y ;
}
```

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ; // obligatoirement comme première instruction
this.couleur = couleur ;
}
public void affiche () // ici redéfinition de la méthode affiche()
{ super.affiche() ; // appel de affiche() de la classe de base
System.out.println (" et ma couleur est : " + couleur) ;
}
private byte couleur ;
}
```

NOTION D'HÉRITAGE

Redéfinition et surdéfinition de membres

La notion de redéfinition de méthode

```
public class TstPcol4
{ public static void main (String args[])
{ Pointcol pc = new Pointcol(3, 5, (byte)3) ;
pc.affiche() ; // ici, il s'agit de affiche de Pointcol
pc.deplace (1, -3) ;
pc.affiche() ;
}
}
```

```
Je suis en 3 5
et ma couleur est : 3
Je suis en 4 2
et ma couleur est : 3
```

NOTION D'HÉRITAGE

Redéfinition et surdéfinition de membres

Surdéfinition et héritage

```
class A
{ .....
public void f (int n) { ..... }
public void f (float x) { ..... }
}
class B extends A
{ .....
public void f (int n) { ..... } // redéfinition de f(int) de A
public void f (double y) { ..... } // surdéfinition de f (de A et de B)
}
A a ; B b ;
int n ; float x ; double y ;
.....
a.f(n) ; // appel de f(int) de A (mise en jeu de surdéfinition dans A)
a.f(x) ; // appel de f(float) de A (mise en jeu de surdéfinition dans A)
a.f(y) ; // erreur de compilation
b.f(n) ; // appel de f(int) de B (mise en jeu de redéfinition)
b.f(x) ; // appel de f(float) de A (mise en jeu de surdéfinition dans A et B)
b.f(y) ; // appel de f(double) de B (mise en jeu de surdéfinition dans A et B)
```

// Surdéfinition et redéfinition peuvent cohabiter.

NOTION D'HÉRITAGE

Redéfinition et surdéfinition de membres

Contraintes portant sur la redéfinition

Lorsqu'on surdéfinit une méthode, on n'est pas obligé de respecter le type de la valeur de retour.

```
class A
{ .....
public int f(int n) { .....}
}
class B extends A
{ .....
public float f(float x) { ..... }
}
```

En revanche, en cas de redéfinition, Java impose non seulement l'identité des signatures, mais aussi celle du type de la valeur de retour

```
class A
{ public int f (int n) { ..... }
}
class B extends A
{ public float f (int n) { ..... } // erreur
}
```

NOTION D'HÉRITAGE

Redéfinition et surdéfinition de membres

Les droits d'accès

La redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode.

```
class A
{ public void f (int n) { ..... }
}
class B extends A
{ private void f (int n) { ..... } // tentative de
redéfinition de f de A
}
```

La redéfinition d'une méthode peut augmenter les droits d'accès à cette méthode.

```
class A
{ private void f (int n) { ..... }
}
class B extends A
{ public void f (int n) { ..... } // redéfinition de f avec extension des droits d'accès

}
```

NOTION D'HÉRITAGE

Duplication de champs

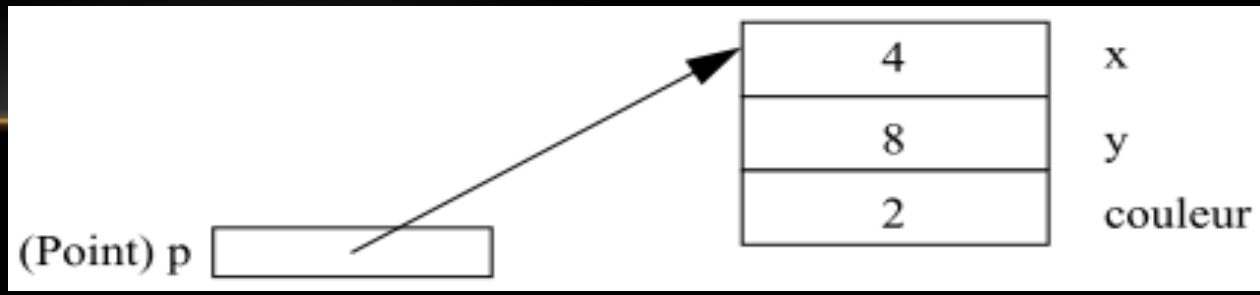
```
class A
{ public int n ;
.....
}
class B extends A
{ public float n ;
public void f()
{ n = 5.25f ; // n désigne le champ n (float) de B
super.n = 3 ; // tandis que super.n désigne le champ n (int) de la super-classe de B
}
}
A a ; B b ;
a.n = 5 ; // a.n désigne ici le champ n(int) de la classe A
b.n = 3.5f ; // b.n désigne ici le champ n(float) de la classe B
```

LE POLYMORPHISME

```
class Point
{ public Point (int x, int y) { ..... }
public void affiche () { ..... }
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
public void affiche () { ..... }
}
```

```
Point p ;
p = new Point (3, 5) ;
```

```
p = new Pointcol (4, 8, (byte)2) ; // p de type Point contient la référence
                                   // à un objet de type Pointcol
```



LE POLYMORPHISME

Exemple complet 1/2

```
class Point
{ public Point (int x, int y)
{ this.x = x ; this.y = y ;
}
public void affiche ()
{ System.out.println ("Je suis en " + x + " " + y) ;
}
private int x, y ;
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
{ super (x, y) ; // obligatoirement comme première instruction
this.couleur = couleur ;
}
public void affiche ()
{ super.affiche() ;
System.out.println (" et ma couleur est : " + couleur) ;
}
private byte couleur ;
}
```

LE POLYMORPHISME

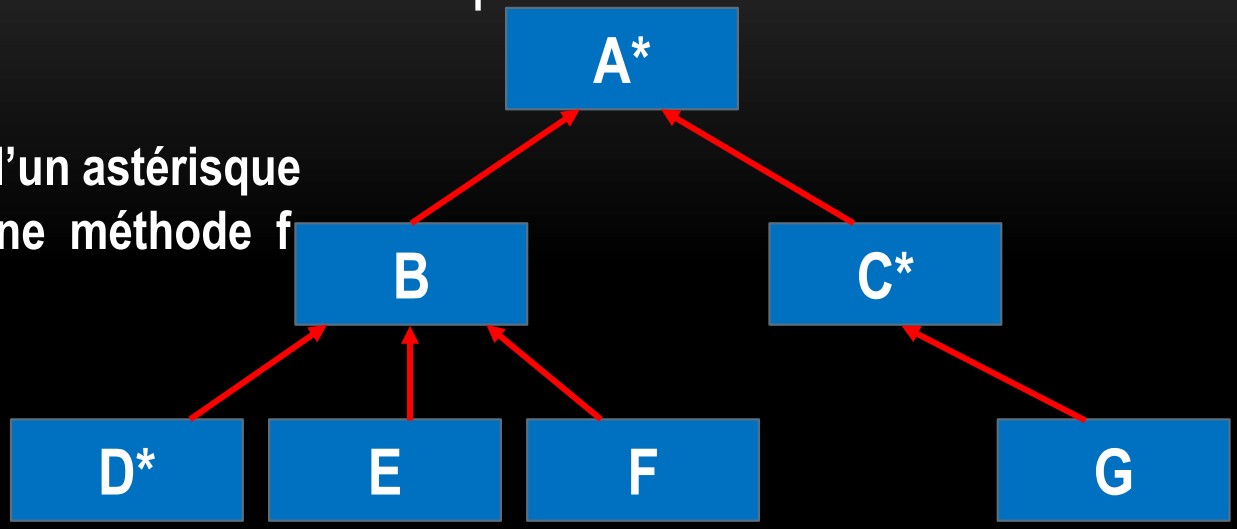
Exemple complet 2/2

```
public class TabHeter
{ public static void main (String args[])
{ Point [] tabPts = new Point [4] ;
tabPts [0] = new Point (0, 2) ;
tabPts [1] = new Pointcol (1, 5, (byte)3) ;
tabPts [2] = new Pointcol (2, 8, (byte)9) ;
tabPts [3] = new Point (1, 2) ;
for (int i=0 ; i< tabPts.length ; i++) tabPts[i].affiche() ;
}
}
```

```
Je suis en 0 2
Je suis en 1 5
et ma couleur est : 3
Je suis en 2 8
et ma couleur est : 9
Je suis en 1 2
```

LE POLYMORPHISME

Généralisation à plusieurs classes



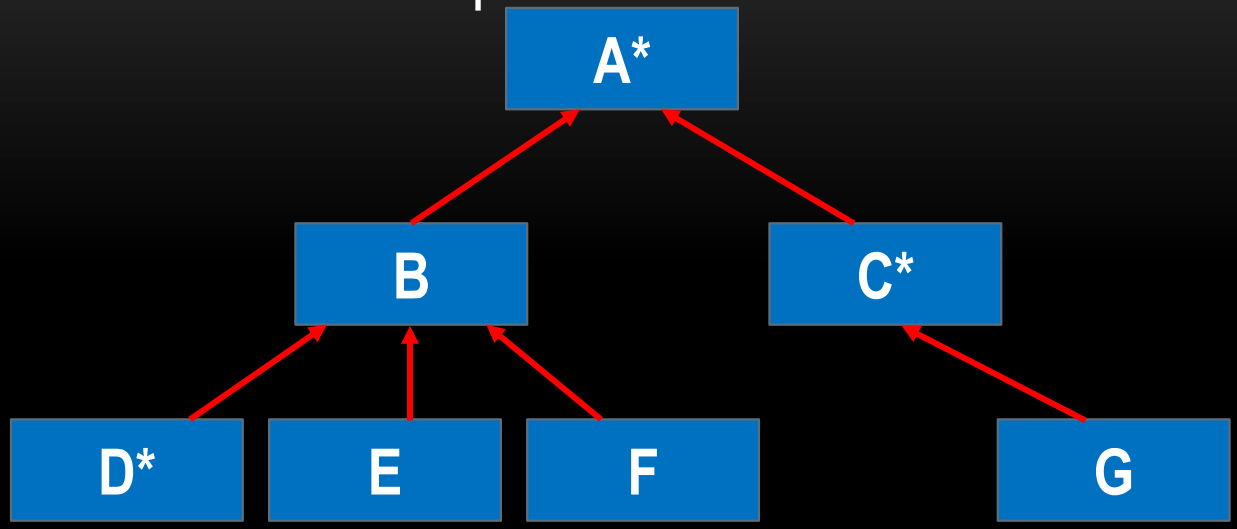
Seules les classes marquées d'un astérisque définissent ou redéfinissent une méthode f

```
//Avec ces déclarations :
A a ; B b ; C c ; D d ; E e ; F f ;
//les affectations suivantes sont légales :
a = b ; a = c ; a = d ; a = e ; a =f ;
b = d ; b = e ;
//En revanche, celles-ci ne  sont pas légales :
b = a ; // erreur : A ne descend pas de B
d = c ; // erreur : C ne descend pas de D
c = d ; // erreur : D ne descend pas de C
```

LE POLYMORPHISME

Généralisation à plusieurs classes

Appel de f



a référence un objet de type A : méthode f de A
a référence un objet de type B : méthode f de A
a référence un objet de type C : méthode f de C
a référence un objet de type D : méthode f de D
a référence un objet de type E : méthode f de A
a référence un objet de type G : méthode f de C.

LA SUPER-CLASSE OBJECT

En JAVA, il existe une classe nommée Object dont dérive implicitement toute classe simple.
Ainsi, lorsque l'on définisse une classe Point de cette manière :

```
class Point
{ .....
}
```

tout se passe en fait comme si l'on avait écrit (On peut d'ailleurs le faire) :

```
class Point extends Object
{ .....
}
```

LA SUPER-CLASSE OBJECT

Utilisation d'une référence de type Object

Une variable de type Object peut être utilisée pour référencer un objet de type quelconque :

```
Point p = new Point (...) ;
Pointcol pc = new Pointcol (...) ;
Fleur f = new Fleur (...) ;
Object o ;
.....
o = p ; // OK
o = pc ; // OK
o = f ; // OK
```

LA SUPER-CLASSE OBJECT

Utilisation d'une référence de type Object

```
Point p = new Point (...) ;
Object o ;
.....
o = p ;
o.deplace() ; // erreur de compilation (Object ne contient pas deplace())
((Point)o).deplace() ; // OK en compilation (attention aux parenthèses)
Point p1 = (Point) o ; // OK : idem ci-dessus, avec création d'une référence
p1.deplace() ; // intermédiaire dans p1
```

LA SUPER-CLASSE OBJECT

Utilisation de méthodes de la classe Object

La classe Object dispose de quelques méthodes qu'on peut soit utiliser telles quelles, soit redéfinir. Les plus importantes sont toString et equals.

LA SUPER-CLASSE OBJECT

Utilisation de méthodes de la classe Object

La méthode toString

La méthode toString de la classe Object fournit une chaîne contenant :
- le nom de la classe concernée,
- l'adresse de l'objet en hexadécimal (précédée de @).

```
class Point
{ public Point(int abs, int ord)
{ x = abs ; y = ord ;
}
private int x, y ;
}
```

```
public class ToString1
{ public static void main (String args[])
    { Point a = new Point (1, 2) ;
Point b = new Point (5, 6) ;
System.out.println ("a = " + a.toString() ) ;
System.out.println ("b = " + b.toString() ) ;
    }
}
```

```
a = Point@fc17aedf
b = Point@fc1baedf
```

LA SUPER-CLASSE OBJECT

Utilisation de méthodes de la classe Object

La méthode equals

La méthode equals définie dans la classe Object se contente de comparer les adresses des deux objets concernés. Ainsi, avec :

```
Object o1 = new Point (1, 2) ;
Object o2 = new Point (1, 2) ;
```

L'expression o1.equals(o2) a pour valeur false.

On peut bien sûr redéfinir cette méthode à sa guise dans n'importe quelle classe. Toutefois, il faudra tenir compte des limitations du polymorphisme.

LA SUPER-CLASSE OBJECT

Utilisation de méthodes de la classe Object

La méthode equals

```
class Point
{ .....
boolean equals (Point p) { return ((p.x==x) && (p.y==y)) ; } // redéfinition
}
Point a = new Point (1, 2) ;
Point b = new Point (1, 2) ;

// l'expression a.equals(b) aura bien sûr la valeur true. En revanche, avec :
Object o1 = new Point (1, 2) ;
Object o2 = new Point (1, 2) ;
/* l'expression o1.equals(o2) aura la valeur false car on aura utilisé la
méthode equals de Object et non celle de Point. */
```

LES MEMBRES PROTÉGÉS

Les différents droits d'accès aux membres d'une classe :
public (mot-clé public), privé (mot-clé private), de paquetage (aucune mention).

Il existe un quatrième droit d'accès dit protégé (mot-clé protected).

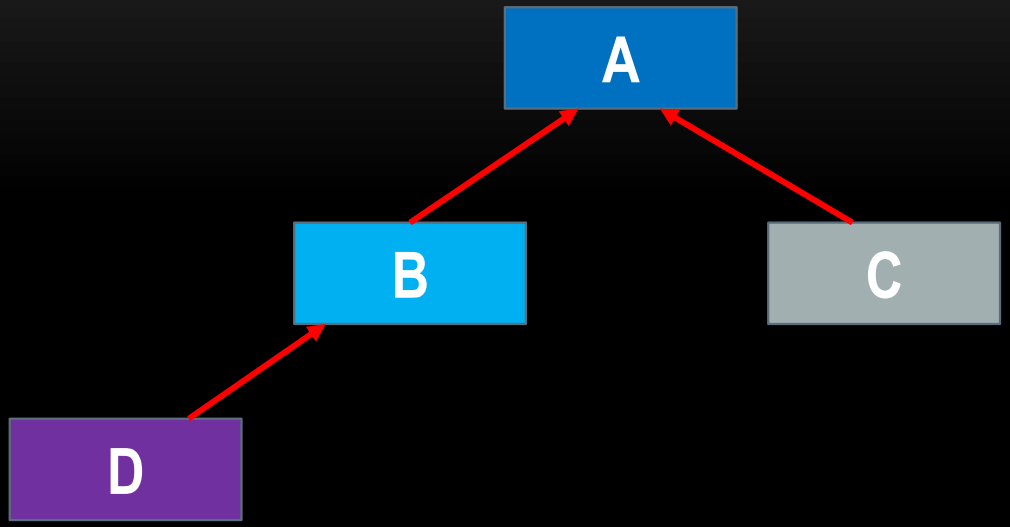
Mais, Java le fait intervenir à deux niveaux totalement différents :

- le paquetage de la classe d'une part,
- ses classes dérivées d'autre part.

En effet, un membre déclaré protected est accessible à des classes du même paquetage, ainsi qu'à ses classes dérivées (qu'elles appartiennent ou non au même paquetage).

LES MEMBRES PROTÉGÉS

```
class A
{ .....
   protected int n ;
}
```



B accède à n de A
D accède à n de B ou de A
C accède à n de A mais pas à n de B ou D

CAS PARTICULIER DES TABLEAUX

Jusqu'ici, nous avons considéré les tableaux comme des objets. Cependant, il n'est pas possible de définir exactement leur classe.
En fait les tableaux ne jouissent que d'une partie des propriétés des objets.

1. Un tableau peut être considéré comme appartenant à une classe dérivée de Object

```
Object o ;
o = new int [5] ; // correct
.....
o = new float [3] ; // OK
```

CAS PARTICULIER DES TABLEAUX

2. Le polymorphisme peut s'appliquer à des tableaux d'objets. Plus précisément, si B dérive de A , un tableau de B est compatible avec un tableau de A

```
class B extends A {…….}
A ta[] ;
B tb[] ;
.....
ta = tb ; // OK car B dérive de A
tb = ta ; // erreur
```

Malheureusement, cette propriété ne peut pas s'appliquer aux types primitifs :

```
int ti[] ; float tf[] ;
.....
ti = tf ; // erreur (on s'y attend car float n'est pas compatible avec int)
tf = ti ; // erreur bien que int soit compatible avec float
```

CAS PARTICULIER DES TABLEAUX

3. Il n'est pas possible de dériver une classe d'une hypothétique classe tableau :

```
class Bizarre extends int [] // erreur
```

CLASSES ET MÉTHODES FINALES

Le mot-clé final peut s'appliquer à des variables locales ou à des champs d'une classe et Il interdit la modification de leur valeur.
Ce mot-clé peut aussi s'appliquer à une méthode ou à une classe, mais avec une signification totalement différente.

Une méthode déclarée final ne peut pas être redéfinie dans une classe dérivée.

Le comportement d'une méthode finale est donc complètement défini et il ne peut plus être remis en cause, sauf si la méthode appelle elle-même une méthode qui n'est pas déclarée final.

CLASSES ET MÉTHODES FINALES

Une classe déclarée final ne peut plus être dérivée.

On pourrait croire qu'une classe finale est équivalente à une classe non finale dont toutes les méthodes seraient finales.

En fait, ce n'est pas vrai car :

- ne pouvant plus être dérivée, une classe finale ne pourra pas se voir ajouter de nouvelles fonctionnalités,
- une classe non finale dont toutes les méthodes sont finales pourra toujours être dérivée, donc se voir ajouter de nouvelles fonctionnalités.

LES CLASSES ABSTRAITES

1 - Présentation

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

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

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

LES CLASSES ABSTRAITES

1 - Présentation

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

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

```
A a = new B(...) ; // OK
```

LES CLASSES ABSTRAITES

2 - Quelques règles

R1 Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot-clé abstract devant sa déclaration (ce qui reste quand même vivement conseillé). Ceci est correct :

```
class A
{ public abstract void f() ; // OK
.....
}
```

Malgré tout, A est considérée comme abstraite et une expression telle que new A(...) sera rejetée.

LES CLASSES ABSTRAITES

2 - Quelques règles

R2

Une méthode abstraite doit obligatoirement être déclarée public, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.

R3

Dans l'en-tête d'une méthode déclarée abstraite, les noms d'arguments muets doivent figurer (bien qu'ils ne servent à rien) :

```
abstract class A
{ public abstract void g(int) ; // erreur : nom d'argument (fictif) obligatoire
}
```

LES CLASSES ABSTRAITES

2 - Quelques règles

R4

Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base (elle peut même n'en redéfinir aucune). Dans ce cas, elle reste simplement abstraite (il est quant même nécessaire de mentionner abstract dans sa déclaration) :

```
abstract class A
{ public abstract void f1() ;
public abstract void f2 (char c) ;
.....
}
        abstract class B extends  A        // abstract  obligatoire  ici
        { public void f1(){…….};           // définition de f1
        …….                                // pas de définition de f2
        }
```

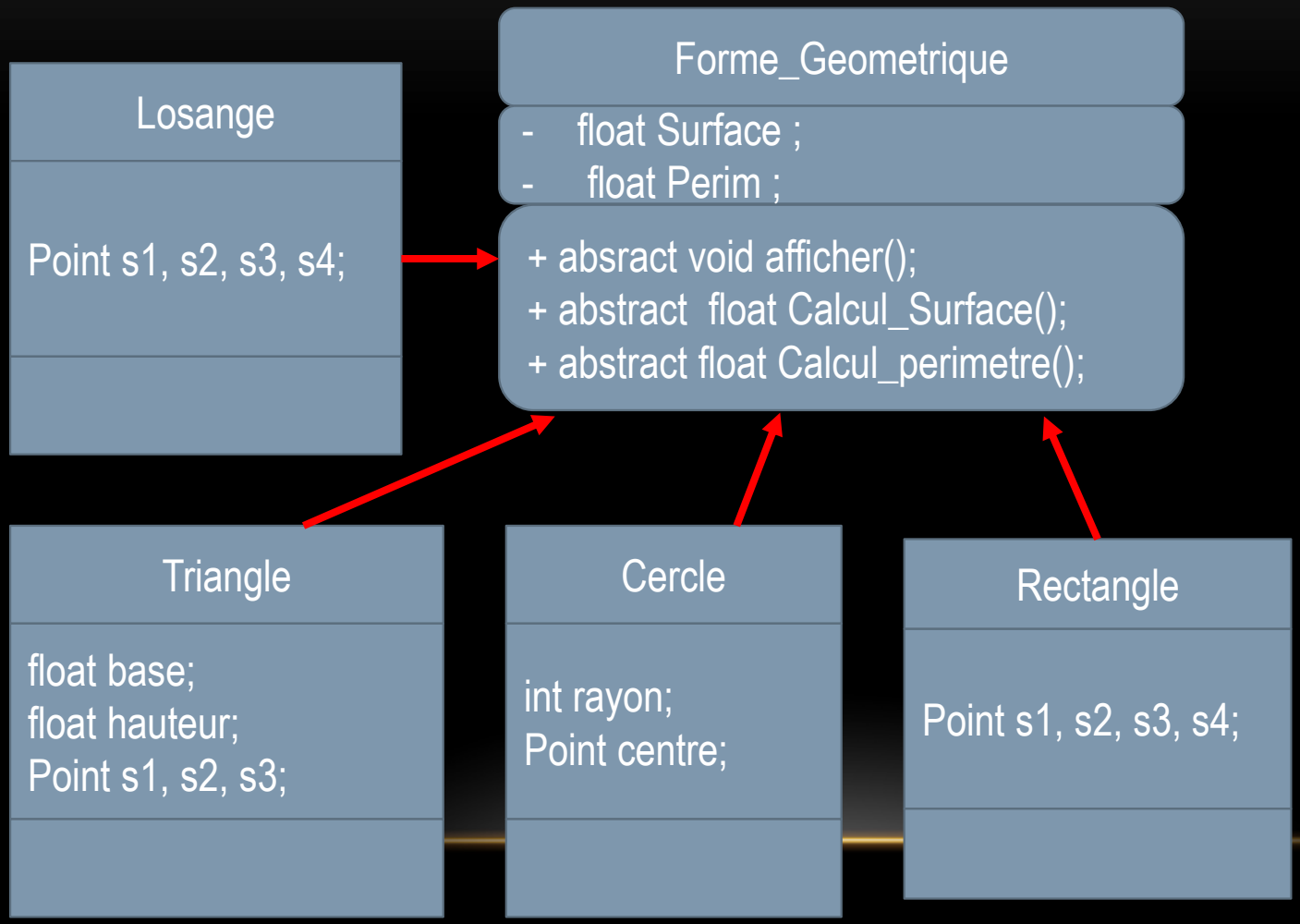
LES CLASSES ABSTRAITES

2 - Quelques règles

R5

Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites. Notez que, toutes les classes dérivant de Object, nous avons utilisé implicitement cette règle dans tous les exemples précédents.

Exercice


Forme_Geometrique
- float Surface ;
- float Perim ;
+ absract void afficher();
+ abstract float Calcul_Surface();
+ abstract float Calcul_perimetre();
Losange
Point s1, s2, s3, s4;
Triangle
float base;
float hauteur;
Point s1, s2, s3;
Cercle
int rayon;
Point centre;
Rectangle
Point s1, s2, s3, s4;

LES INTERFACES

Si l'on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion d'interface.
En effet, une interface définit les en-têtes d'un certain nombre de méthodes, ainsi que des constantes.

LES INTERFACES

1 - Mise en œuvre d'une interface

1 – 1 Définition d'une interface

On utilise simplement le mot-clé interface à la place de class :

```
public interface I
{
   void f(int n) ; // en-tête d'une méthode f (public abstract facultatifs)
   void g() ; // en-tête d'une méthode g (public abstract facultatifs)
}
```

Dans la définition d'une interface, on ne peut trouver que des en-têtes de méthodes ou des constantes.

Par essence, les méthodes d'une interface sont abstraites (puisqu'on n'en fournit pas de définition) et publiques (puisqu'elles devront être redéfinies plus tard).

LES INTERFACES

1 - Mise en œuvre d'une interface

1 – 2 Implémentation d'une interface

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

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

Ici, on indique que A doit définir les méthodes prévues dans l'interface I, c'est-à-dire f et g.

LES INTERFACES

1 - Mise en œuvre d'une interface

1 – 2 Implémentation d'une interface

Une même classe peut implémenter plusieurs interfaces :

```
public interface I1
{ void f() ;
}
public interface I2
{ int h() ;
}
class A implements I1, I2
{ // A doit obligatoirement définir les méthodes f et h prévues dans
        //I1 et I2
}
```

LES INTERFACES

2 - Variables de type interface et polymorphisme

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

class A implements I { ..... }  // pas de relalion d'héritage
.....
I i = new A(...) ; // OK
```

LES INTERFACES

Exemple

```
interface Affichable
{ void affiche() ;
}
class Entier implements Affichable
{ public Entier (int n)
{ valeur = n ;
}
public void affiche()
{ System.out.println ("Je suis un entier de valeur " + valeur) ;
}
private int valeur ;
}
```

```
class Flottant implements Affichable
{ public Flottant (float x)
{ valeur = x ;
}
public void affiche()
{ System.out.println ("Je suis un flottant de valeur " + valeur) ;
}
private float valeur ;
}
```

LES INTERFACES

Exemple

```
public class Tabhet4
{ public static void main (String[] args)
{ Affichable [] tab ;
tab = new Affichable [3] ;
tab [0] = new Entier (25) ;
tab [1] = new Flottant (1.25f) ;
tab [2] = new Entier (42) ;
int i ;
for (i=0 ; i<3 ; i++)
tab[i].affiche() ;
}
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

Remarque :
les droits d'accès des interfaces sont régis par les mêmes règles que ceux des classes.

LES INTERFACES

3 - Interface et classe dérivée

La clause implements est une garantie qu'offre une classe d'implémenter les fonctionnalités proposées dans une interface. Elle est totalement indépendante de l'héritage ; autrement dit, une classe dérivée peut implémenter une interface (ou plusieurs) :

```
interface I
{ void f(int n) ;
void g() ;
}
class A { ..... }
class B extends A implements I
{    // les méthodes f et g doivent soit être déjà définies dans A,
    // soit définies dans B
}
```

LES INTERFACES

3 - Interface et classe dérivée

On peut même rencontrer cette situation :

```
interface I1 { ..... }

interface I2 { ..... }

class A implements I1 { ..... }

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

LES INTERFACES

4 - Interfaces et constantes

une interface peut aussi renfermer des constantes symboliques qui seront alors accessibles à toutes les classes implémentant l'interface :

```
interface I
{ void f(int n) ;
void g() ;
static final int MAXI = 100 ;
}
class A implements I
{ // doit définir f et g
  // dans toutes les méthodes de A, on a accès au symbole MAXI :
   // par exemple : if (i < MAXI) .....
}
```

LES INTERFACES

4 - Interfaces et constantes

une interface peut aussi renfermer des constantes symboliques qui seront alors accessibles à toutes les classes implémentant l'interface :

```
interface I
{ void f(int n) ;
void g() ;
static final int MAXI = 100 ;
}
class A implements I
{ // doit définir f et g
  // dans toutes les méthodes de A, on a accès au symbole MAXI :
   // par exemple : if (i < MAXI) .....
}
```

Les constantes sont accessibles en dehors d'une classe implémentant l'interface. Par exemple, la constante MAXI de l'interface I se notera simplement I.MAXI.

LES INTERFACES

5 - Dérivation d'une interface

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

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

La définition de I2 est totalement équivalente à :

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

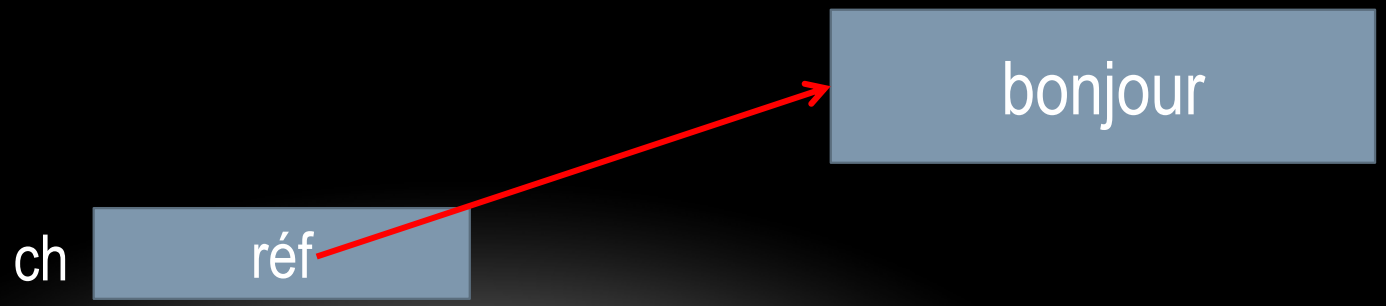
LES CHAÎNES DE CARACTÈRES ET LES TYPES ÉNUMÉRÉS

FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

1 Introduction

Java dispose d'une classe standard nommée String, permettant de manipuler des chaînes de caractères, c'est-à-dire des suites de caractères.

```
String ch ; // ch est une référence sur un objet de type String
ch = "bonjour" ;
```

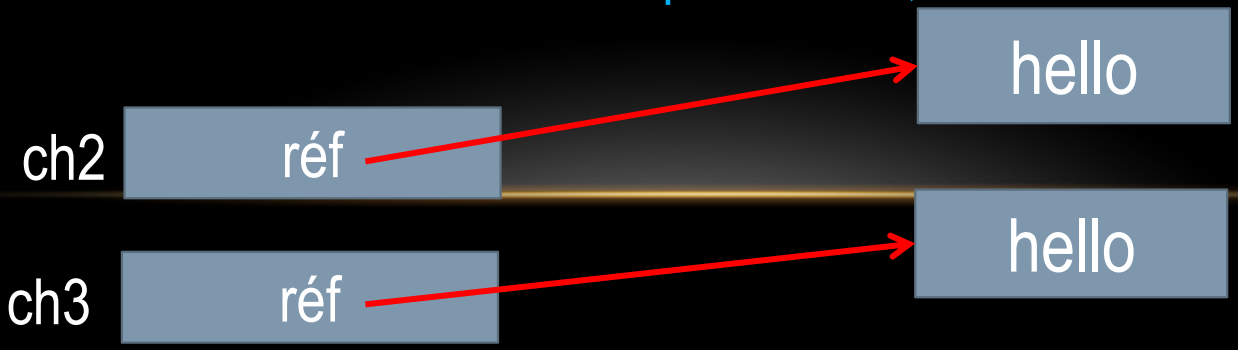


FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

1 Introduction

La classe String dispose de deux constructeurs, l'un sans argument créant une chaîne vide, l'autre avec un argument de type String qui en crée une copie :

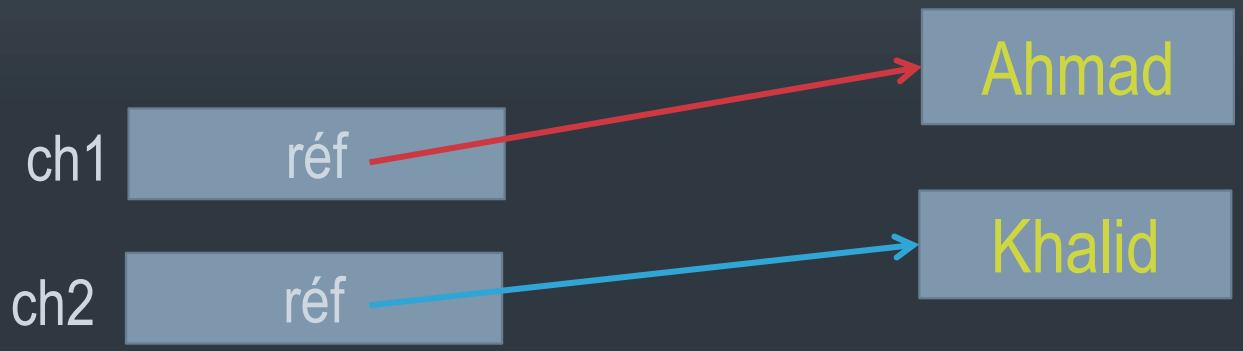
```
String ch1 = new String () ; // ch1 contient la référence à une chaîne vide
String ch2 = new String("hello") ;  // ch2 contient la référence à une chaîne
                                    // contenant la suite "hello"
String ch3 = new String(ch2) ; // ch3 contient la référence à une chaîne
                               // copie de ch2, donc contenant "hello"
```



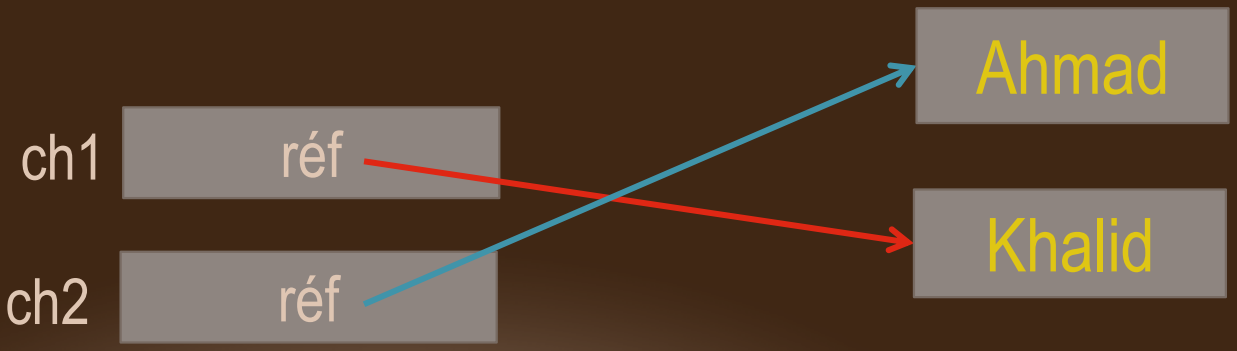
FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

2 Un objet de type String n'est pas modifiable

```
String ch1, ch2, ch ;
ch1 = “Ahmad" ;
ch2 = “Khalid" ;
```




```
ch = ch1 ;
ch1 = ch2 ;
ch2 = ch ;
```



FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

3 Longueur d'une chaîne : length

```
String ch = "bonjour" ;
int n = ch.length() ;                    // n contient 7
ch = "hello" ; n = ch.length () ; // n contient 5
ch = "" ; n = ch.length () ;        // n contient 0
```

Remarque
Contrairement à ce qui se passait pour les tableaux où length désignait un champ, nous avons bien affaire ici à une méthode.
Les parenthèses à la suite de son nom sont donc indispensables.

FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

4 Accès aux caractères d'une chaîne : charAt

La méthode charAt de la classe String permet d'accéder à un caractère de rang donné d'une chaîne (le premier caractère porte le rang 0).
Ainsi, avec :

```
String ch = "bonjour" ;
```

ch.charAt(0) correspond au caractère 'b',
ch.charAt(2) correspond au caractère 'n'.

FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

4 Accès aux caractères d'une chaîne : charAt

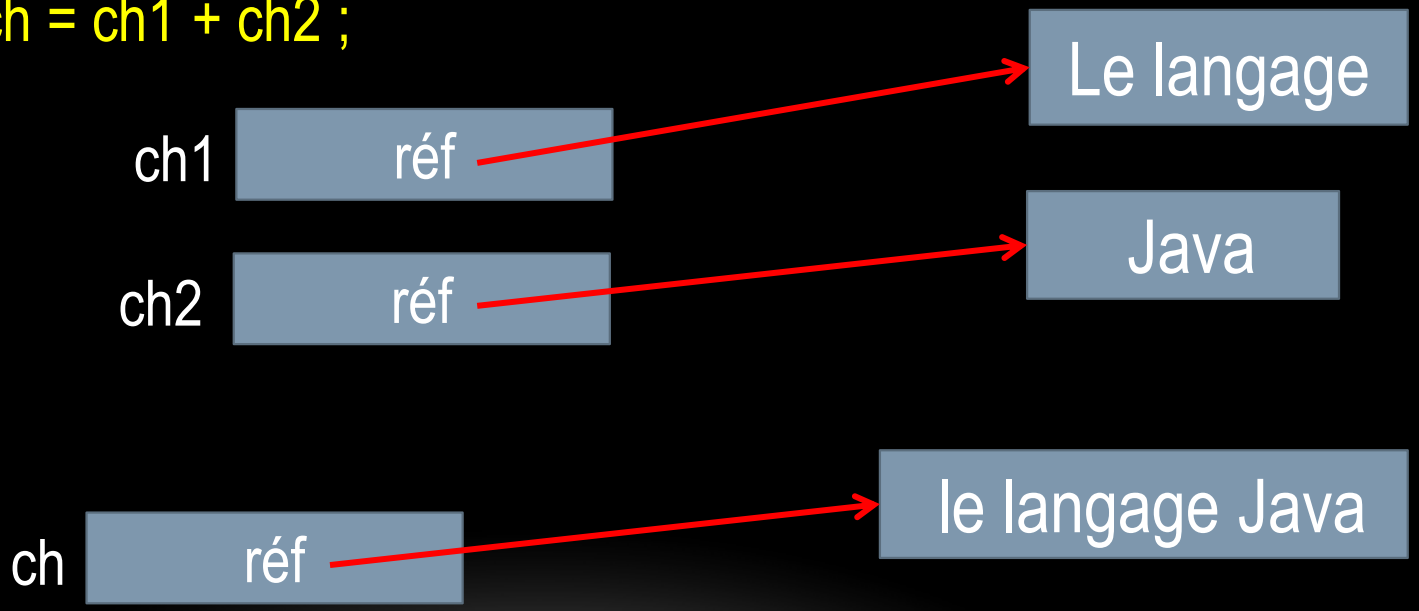
```
public class MotCol
{ public static void main (String args[])
{ String mot ;
        System.out.print ("donnez un mot : ") ;
        mot = Clavier.lireString() ;
        System.out.println ("voici votre mot en colonne :") ;
        for (int i=0 ; i<mot.length() ; i++) // ou (JDK 5.0) : for (char c : mot)
        System.out.println (mot.charAt(i)) ; // System.out.println (c) ;
}
}
```

```
donnez un mot : Ali
voici votre mot en colonne :
A
l
i
```

FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

5 Concaténation de chaînes

```
String ch1 = "Le langage " ;
String ch2 = " Java" ;
String ch = ch1 + ch2 ;
```



FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

5 Concaténation de chaînes

Les instructions suivantes sont aussi autorisées

```
System.out.println (ch1 + ch2) ;
ch = ch1 + "C++" ;
ch = ch1 + " le plus puissant est : " + ch2 ;
```

```
int n = 26 ;
String titre = new String ("résultat : ") ;
String monnaie = "$"
String resul = titre + n + " " + monnaie ;
System.out.println (resul) ; // affichera : résultat : 26 $
```

FONCTIONNALITÉS DE BASE DE LA CLASSE STRING

5 Concaténation de chaînes

Les instructions suivantes sont aussi autorisées

```
String ch = "bonjour" ;
ch += " monsieur" ; // ch désigne la chaîne "bonjour monsieur"
```

6 Écriture des constantes chaînes

```
String ch = "bonjour\nmonsieur" ;
System.out.println (ch) ;
```

```
// On aura en exécution
bonjour
monsieur
```

RECHERCHE DANS UNE CHAÎNE

La méthode indexOf surdéfinie dans la classe String permet de rechercher, à partir du début ne chaîne ou d'une position donnée :
• la première occurrence d'un caractère donné,
• la première occurrence d'une autre chaîne.

Dans tous les cas, elle fournit :
• la position du caractère (ou du début de la chaîne recherchée) si une correspondance a effectivement été trouvée,
• la valeur -1 sinon.

Il existe également une méthode lastIndexOf, surdéffinie pour effectuer les mêmes recherches que indexOf, mais en examinant la chaîne depuis sa fin.

RECHERCHE DANS UNE CHAÎNE

Exemple

```
String mot = "anticonstitutionnellement" ;
int n ;
n = mot.indexOf (’t’) ; // n vaut 2
n = mot.lastIndexOf (’t’) ; // n vaut 24
n = mot.indexOf ("ti") ; // n vaut 2
n = mot.lastIndexOf ("ti") ; // n vaut 12
n = mot.indexOf (’x’) ; // n vaut -1
```

COMPARAISONS DE CHAÎNES

La méthode equals

La classe String dispose d'une méthode quals qui compare le contenu de deux chaînes

```
String ch1 = "hello" ;
String ch2 = "bonjour" ;
.....
ch1.equals(ch2) // cette expression est fausse
ch1.equals("hello") // cette expression est vraie
```

COMPARAISONS DE CHAÎNES

La méthode equalsIgnoreCase

La méthode equalsIgnoreCase effectue la même comparaison, mais sans distinguer les majuscules des minuscules :

```
String ch1 = "HeLlo" ;
String ch2 = "hello" ;
.....
ch1.equalsIgnoreCase(ch2) // cette expression est vraie
ch1.equalsIgnoreCase("hello") // cette expression est vraie
```