Bases orientées objet du langage Java

Chargé du cours :
Dr Pélagie HOUNGUE

Contact :
pelagie.houngue@imsp-uac.org

Plan

- Classes et objets Java
- Héritage, polymorphisme et Classes abstraites
- Surcharge et redéfinition
- Paquetages et interfaces

Classes et objets Java

Classes et Objets

- La programmation orientée objet met dans une même structure, les données et les opérations qui leur sont associées.
- Le concept de classe correspond simplement à la généralisation de la notion de type que l'on rencontre dans les langages classiques.
- En effet, une classe n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes.

Classes et Objets (2)

Chien
taille poids race nom
aboyer()

Une classe chien

58 cm
25 kg
berger allemand
Fritz

55 cm
31 kg
labrador
Noireau

Plusieurs objets Chien

Classes et Objets (3)

- Les objets apparaissent alors comme des variables d'un tel type classe (en P.O.O., on dit aussi qu'un objet est une *instance de sa classe)*.
  - *La structure* est commune,
  - Les valeurs des champs sont propres à chaque objet,
  - En revanche, les méthodes sont effectivement communes à l'ensemble des objets d'une même classe.

Classes et Objets (4)

- Toute entité identifiable, concrète ou abstraite, peut être considérée comme un objet
- Un objet réagit à certains messages qu'on lui envoie de l'extérieur
  - la façon dont il réagit détermine le comportement de l'objet

Classes et Objets (5)

- Un objet possède
  - une adresse en mémoire (identifie l'objet)
  - un comportement (ou interface)
    - Le comportement est donné par des fonctions ou procédures, appelées méthodes
  - un état interne
    - L'état interne est donné par des valeurs des attributs

Déclaration et instanciation des classes

- Une classe java est déclarée par le mot clé class, placé devant l'identificateur de la classe (son nom)
- Voici la définition d'une classe vide :
  - class MaClasse { }
- MaClasse monObjet;
  - Elle crée une référence appelée monObjet, mais n'instancie pas l'objet. Ce sera le rôle de l'opérateur new.
- Instanciation
  - monObjet = new MaClasse();
    - le nom de la classe est utilisé comme s'il s'agissait d'une méthode.
      - Il s'agit du constructeur par défaut

Données membres et méthodes membres ou d'instance

- Une donnée membre d'une classe peut avoir n'importe quel type
  - primitif, composite et types d'objets.
- Pour accéder à une donnée membre, il faut d'abord créer une occurrence ou instance de la classe puis accéder aux données avec l'opérateur "."

Données membres et méthodes membres ou d'instance (2)

- **Exemple de classe décrivant un chien**

```
public class ClasseChien {
        String nom, couleurYeux;
        int age;
        boolean avecQueue;

        public void aboie() {

                //…
        }
}
```

- ClasseChien chien = new ClasseChien();
- chien.age = 4;
- chien.aboie();

Données membres et méthodes membres ou d'instance (3)

- **Exemple de classe décrivant un cercle**

```
class Cercle {
double r; // champs : rayon du cercle
double calculeSurface() // méthode de calcul d'une
                                        //surface
{
      return(3.1416*r*r);
}

}// Fin de class Cercle
```

Données membres et méthodes membres ou d'instance (4)

- **Exemple de classe décrivant un chat**

```
class Chat {
String nom; // nom du chat
int age; // en annees
double tauxRonronnement; // entre 0.0 et 1.0

void viellir()
{
        age += 1;
}

int retournerAge()
{
        return(age);
}

}
```

Point d'entrée d'un programme

- Un programme Java est constitué d'une ou de plusieurs classes.
  - Parmi toutes ces classes, il doit exister au moins une classe qui contient la méthode statique et publique main() qui est le point d'entrée de l'exécution du programme

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

Référence à un objet

- En Java, on ne peut accéder aux objets qu'à travers une référence vers ceux-ci.
- Déclaration d'une variable p avec pour type un nom de classe :
  - Point p; //référence à un objet de la classe Point
    - Lorsque l'on déclare une variable qui est du type d'une classe, cette variable a, par défaut, la valeur null.
      - Ici p a la valeur null

Opérateur new

- **new**
  - création d'une instance d'objet d'une classe ;
  - retourne une référence à cette instance d'objet.
    - Point p = new Point();
    - Point p2 = p;
      - tout changement à p2 affecte l'objet référencé par p et vice-versa
- Objet qui n'est plus utilisé → le ramasse-miettes (garbage collector) récupère automatiquement la mémoire associée.

Protection des variables et méthodes

- **L'encapsulation**
  - Mécanisme par lequel les données d'un objet sont protégées.
  - Elle passe par le contrôle des données et des comportements de l'objet.
  - Le langage Java propose en général quatre niveaux de protection pour les attributs et les méthodes
    - la protection publique, la protection privée, la protection protégée et la protection par défaut (paquetage).

Protection des variables et méthodes

- **Protection public**
  - Les attributs d'une classe déclarés avec le qualificatif (ou spécificateur) public sont accessibles par tous les objets de l'application.
  - Les données peuvent être modifiées par une méthode de la classe, par une méthode d'une autre classe ou depuis la fonction main().
- **Protection private**
  - Les attributs et méthodes d'une classe déclarée private ne sont accessibles que par les méthodes de la même classe.
  - Les attributs private ne peuvent être modifiés ou initialisés que par les méthodes de la même classe.
  - Les attributs et les méthodes private ne peuvent être appelés par la fonction main().

Protection des variables et méthodes (2)

- **Protection protected**
  - Les attributs et méthodes déclarés protected ne sont accessibles que par les méthodes de la même classe ou par les fonctions membres de la classe dérivée.
- **Protection par défaut ou paquetage**
  - Les attributs et méthodes avec le niveau de protection par défaut sont accessibles aux données et aux méthodes du même paquetage.
- Généralement, les attributs sont privés tandis que les méthodes sont publiques.

Le mot clé this

- This
  - Fait référence à l'instance de l'objet courant.
  - Il désigne en Java l'objet lui-même et non l'adresse de l'objet comme en C++.
- Il est permis à une variable locale à une méthode de porter le même nom qu'une variable d'instance
- Exemple d'utilisation de this

```
class Coordonnees{

private int x ;
private int y ;

void init(int x, int y) {
this.x = x;
this.y = y;
}
}
```

Constructeur

- C'est une méthode qui est utilisée pour initialiser correctement les attributs d'un objet au moment de la création de l'objet en mémoire.
- Le constructeur initialise un objet immédiatement après sa création.
- Le ***constructeur par défaut*** est celui que le langage Java définit pour chaque classe construite par le programmeur.
- Le constructeur par défaut initialise les attributs comme suit :
  - s'il s'agit des entiers à 0, à 0.0 s'il s'agit des réels, à '\0' s'il s'agit des caractères, et à null pour les String ou autres types de données composés.

Constructeur (2)

- Même nom que celui de la classe
- Pas de type de retour (pas même void)
- Exemple de classe décrivant un cercle

```
class Cercle {
private double r; // champs : rayon du
cercle
// Constructeur : initialisation des champs
public Cercle(double nouvRayon) {
r = nouvRayon;
}
public double calculeSurface() {
return(3.1416*r*r); // méthode de calcul
}
}// Fin de class Cercle
```

Surcharge de constructeur

- Il est possible de surcharger le constructeur d'une classe.
  - Cela signifie que Java permet la définition de plusieurs constructeurs dans une même classe. L'interpréteur observe les paramètres pour déterminer le constructeur à utiliser lors de l'appel.

Surcharge de constructeur (2)

- Exemple décrivant un point (.)

```
class Point{
private int abscisse ;
private int ordonnee ;
private int cote ;
public Point() {
abscisse = 0 ;
ordonnee = 0 ;
cote = 0 ;
}
public Point(int abs, int ord) {
abscisse = abs ;
ordonnee = ord ;
}
public Point(int abs, int ord, int cot) {
this(abs, ord) ; //permet de faire appel au
constructeur à deux paramètres
cote = cot ;
}
}
```

Déclaration *static* des données membres et des méthodes

- Il est possible en Java de définir des données membres et des méthodes de classe existantes en un seul exemplaire quelque soit le nombre d'objets instances d'une même classe.
- Ainsi, ces données membres et ces méthodes peuvent être appelées indépendamment de toute allocation d'objet.
- Règles
  - Une méthode statique appartenant à une classe donnée ne peut faire appel qu'à un attribut de type static (hormis la fonction main)
  - En revanche, une méthode non statique peut faire appel aussi bien à des attributs déclarés static qu'à ceux déclarés non static.

Déclaration *static* des données membres et des méthodes (2)

```
class MaClasse{
static int n;
int q;
static void affiche()
{
    System.out.println(n);
   // q++; impossible
}

void test()
{
    System.out.println(n);
    q++;
    System.out.println("q = "+q);
}
}//fin classe MaClasse
```

```
public class MembreStatic {
public static void main(String[] args) {
MaClasse m1 = new MaClasse();
MaClasse m2 = new MaClasse();
m1.affiche();
m2.affiche();
MaClasse.n = 3;
m1.affiche();
m2.affiche();
MaClasse.n = 10;
MaClasse.affiche();
m2.affiche();
m1.affiche();
m1.q = 4;
m1.test();
m2.test();
}// fin main
}// fin classe MembreStatic
```

Accesseurs et modificateurs

- Deux types de méthodes servent à donner accès aux variables depuis l'extérieur de la classe :
  - les **accesseurs** pour lire les valeurs des variables ;
  - les **modificateurs** en écriture pour modifier leur valeur.

```
public class Employe {
private double salaire;
//Constructeur
public Employe(double unSalaire) {
    salaire = unSalaire;
}
//Modificateur
public void setSalaire(double unSalaire) {
if (salaire <= 100000.0)
    salaire + = unSalaire;
}
//Accesseur
public double getSalaire() {
   return salaire;
}
}//fin classe Employe
```

Exercice 1

- Créer une classe nommée AnimalFamilier.
- Définir pour la classe, les attributs suivants : âge, taille, poids et couleur.
- Définir le constructeur de la classe AnimalFamilier.
- Définir dans la classe AnimalFamilier, les méthodes manger, dormir et dire pour implémenter les comportements d'un animal familier.
- Définir dans la classe AnimalFamilier, une méthode nommée afficher qui affiche les caractéristiques d'un animal familier.
- Définir la fonction principale dans laquelle vous créerez un objet sur lequel vous ferez appel aux différentes méthodes implémentées.

Exercice 2

- À chaque étape de cet exercice, compilez votre fichier pour contrôler qu'il est syntaxiquement correct.
- 1. Créez dans un fichier nommé Personne.java, une classe PersonneDetails.
- 2. Ajoutez dans cette classe les attributs (avec une visibilité par défaut) nom et prenom de type chaîne de caractères.
- 3. Ajoutez un attribut age de type entier.
- 4. Ajoutez un constructeur qui initialise nom et prenom.
- 5. Ajoutez une fonction principale (main) dans une classe nommée Personne qui crée un objet de type PersonneDetails.
  - Dans ce main, faites afficher le nom et le prénom de l'objet créé. Exécutez votre programme.

Exercice 2 (suite)

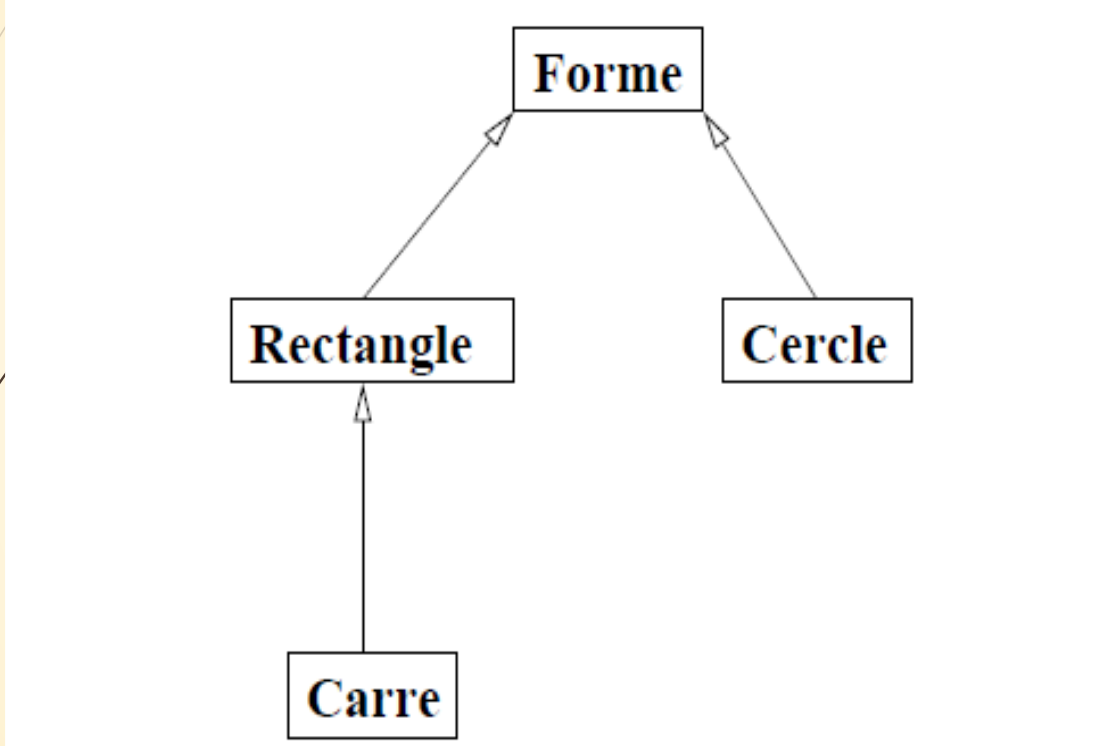
- Dans la classe PersonneDetails,
  - 6.Ajoutez un constructeur qui initialise age en plus de nom et prenom et créez dans le main un objet de type PersonneDetails qui fait appel à ce 2ème constructeur.
  - 7.Ajoutez une méthode nommée anniversaire qui augmente age de 1 et faites lui appel sur un objet de type PersonneDetails.
  - 8.Rendez tous les attributs privés et ajoutez maintenant une méthode nommée setNom qui modifie la valeur de nom en fonction d'un paramètre. Faites de même pour prénom et âge.
  - 9.Ajoutez une méthode nommée getNom qui retourne la valeur de nom. Faites de même pour prenom et age.
  - 10. Ajoutez une méthode nommée afficher qui affiche pour une personne ses trois attributs.
- Dans la classe Personne
  - Faites appel aux différentes méthodes définies ci-dessus, sur un objet PersonneDetails.

Exercice 3

- Créer une classe compte avec quatre méthodes à savoir :
  - Depot : permet de déposer un montant donné sur le compte de l'objet courant
  - Retrait : permet de retirer un montant donné du compte de l'objet courant
  - Virement : permet de virer un montant donné d'un compte pris en paramètre (référence) vers le compte de l'objet courant
  - Afficher : affiche le solde du compte de l'objet courant
- La classe dispose d'un attribut solde de type entier
- Créer la fonction principale dans laquelle vous ferez les opérations suivantes :
  - Créez deux comptes que vous affecterez à deux variables
  - dépôt de 500.000 fcfa sur le premier compte.
  - dépôt de 100.000 fcfa sur le second compte.
  - retrait de 10.000 fcfa du second compte.
  - virement de 200.000 fcfa du premier compte vers le second.
  - Effectuer l'affichage des soldes des deux comptes.

Héritage, Polymorphisme et Classes abstraites

Héritage



Exemple de relations d'héritage

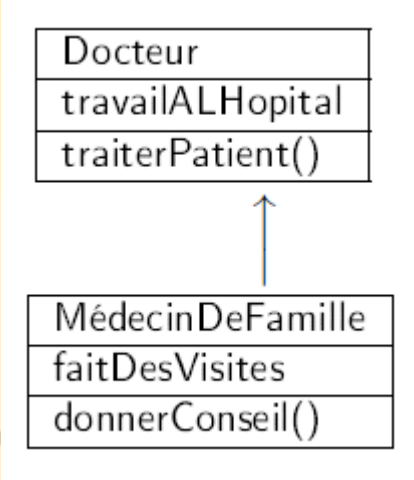
Héritage (2)

- Concept d'héritage
  - permet la réutilisation des fonctionnalités d'une classe tout en apportant certaines variations spécifiques de l'objet dérivé.
  - Avec l'héritage, les attributs définis pour un ensemble de données sont réutilisables pour des variantes de cet ensemble.
  - Confère à une classe dérivée certaines propriétés et attributs suivant le niveau de visibilité.
  - En pratique, la relation « est un » permet de déterminer si une classe hérite d'une autre classe.
- Les descendants par héritage sont nommés ***« sous classes ».***
- Le parent direct est une **super classe**.
- Une sous classe est une version spécialisée d'une classe qui hérite de toutes les variables d'instance et méthodes.

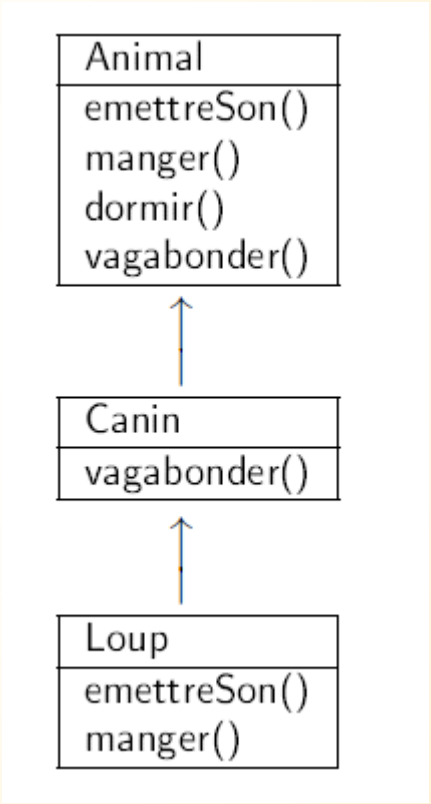
Quand utiliser l'héritage?

- Utiliser l'héritage lorsqu'une classe est un type plus spécifique qu'une super classe.
- Utiliser l'héritage quand un comportement doit être partagé par plusieurs classes du même type général.
- Ne pas utiliser l'héritage uniquement pour pouvoir réutiliser le code d'une autre classe sans qu'il y ait de relation entre les deux classes.

Types d'héritage en Java



Héritage simple



Héritage multi-niveaux

Implémentation de l'héritage

- Pour qu'une classe B hérite d'une classe A (on dit aussi que B dérive de la classe A, ou que A est une super classe pour B, ou que B est une sous-classe de A), on utilise le mot clé extends.
- La syntaxe est la suivante :

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

- En Java
  - une classe ne peut avoir qu'une seule super classe.
  - Ce qui la différencie fondamentalement du C++ qui autorise l'héritage multiple.

Exemple

```
class Point{

protected int x ;
protected int y ;

public Point(int x, int y) {
this.x = x;
this.y = y;
}
}
```

```
class Point3D extends Point {
private int z;
public Point3D(int x, int y, int
z) {
this.x = x;
this.y = y;
this.z = z;
}
public Point3D() {
this(-1, -1, -1);
}
}
```

Quelques règles

- Une classe dérivée ne peut accéder aux attributs et méthodes **privés** d'une super classe ;
- Une variable d'une super classe peut faire référence à un objet de la classe dérivée ***mais la réciproque est fausse***.
- En Java on peut créer l'héritage simple sur plusieurs niveaux.

Les constructeurs dans le cas de l'héritage

- Une classe dérivée peut faire appel à un constructeur de la super classe par l'utilisation du mot **clé super**
  - **Super** (listedesparametres)
  - Cette dernière doit être la première instruction dans le constructeur.
- ***Pour construire un objet dérivé, il est indispensable de construire d'abord l'objet associé à la classe de base.***
  - S'il n'y a pas d'appel explicite au constructeur de la classe de base, le compilateur recherche lui-même le constructeur par défaut (celui sans paramètre) de la classe base.

Exemple

- Une boîte est caractérisée par sa longueur, sa largeur et sa hauteur. On considère ici des boîtes coloriées qui dérivent de boîtes.
- Ecrivons un programme Java qui traduit cet énoncé en définissant les constructeurs appropriés et permet de calculer le volume d'une boîte.

Exemple

```
// Ce programme s'appelle
//BoiteCouleur.java
class Boite{
protected int longueur ;
protected int largeur ;
protected int hauteur ;
public Boite(int lon, int lar, int
ht){
longueur = lon ;
largeur = lar ;
hauteur = ht ;
System.out.println("Constructeur
de  Boite\n");
}
public void message(){
System.out.println("Message de
Boite\n");
}
}
```

```
class BoiteColoriee extends Boite{
private String couleur ;
public BoiteColoriee(int lon, int lar, int ht,
String cl){
super(lon, lar, ht) ;//doit être la 1ère
                     //instruction
couleur = cl ;
System.out.println("Constructeur de
BoiteColoriee\n");
}
public int volume(){
return (longueur*largeur*hauteur) ;
}
public  void message1(){
System.out.println("Message de
BoiteColoriee\n");
}
}
```

Exemple

```
public class BoiteCouleur{
public static void main(String[] args) {
Boite B1 = new Boite(20, 10, 15);
B1.message();
BoiteColoriee B2 = new BoiteColoriee(30,
20, 15, "Jaune" );
B2.message1();
System.out.println("Le volume de B2 est :" +
B2.volume());
B1=B2;//contraire interdit ou casting
B1.message();//B1.message1() interdit
B2=(BoiteColoriee)B1;
B2.message1();
B2.message();
}
}
```

Polymorphisme

- Le polymorphisme est la faculté attribuée à un objet d'être une instance de plusieurs classes.
- Cette propriété est très utile pour la création d'ensemble regroupant des objets de classes différentes.
- En d'autres termes
  - Le fait qu'une variable *Oi* de la classe *Ci* puisse référencer non seulement un objet de la classe *Ci* mais en fait tout objet dérivé de la classe *Ci* est appelé **polymorphisme**
    - la faculté pour une variable de référencer différents types d'objets.

Exemple

```
class Point
{
protected int x, y ;
public Point (int x, int y)
{
 this.x = x ;
this.y = y ;
}
}
```

```
class Pointcol extends Point
{
private String couleur ;
public Pointcol (int x, int y, String
couleur)
{
super (x, y) ; // obligatoirement
        //comme première instruction
this.couleur = couleur ;
}
}
```

Exemple suite

```
public class TestPoly
{
public static void main (String args[])
{
Point [] tabPts = new Point [4] ;//création d'un tableau d'objets
tabPts [0] = new Point (0, 2) ;
tabPts [1] = new Pointcol (1, 5, "rouge") ;
tabPts [2] = new Pointcol (2, 8, "bleu") ;
tabPts [3] = new Point (1, 2) ;
for (int i=0 ; i < tabPts.length ; i++) {
      if (tabPts[i] instanceof Point)
         System.out.println("element " + i + " est un point");
      if (tabPts[i] instanceof Pointcol)
         System.out.println("element " + i + " est un pointcol");
}
}//fin main
}
```

La classe Object

- Classe très spéciale en Java qui s'appelle Object.
- Toutes les classes sont des sous classes de Object.
  - Object est une super classe pour toutes les autres classes.
  - une variable de type Object, peut faire référence à n'importe quel objet de n'importe quelle classe.

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

La Classe Object (2)

- La classe Object définit des méthodes qui par conséquent sont disponibles dans toutes les classes par héritage

Méthodes	Objectif
Object clone()	Créé un objet identique à l'objet cloné
Boolean equals(Object obj)	Détermine si un objet est égal à un autre
void notify()	Reprend l'exécution d'un *thread* en attente sur l'objet qui effectue l'appel
String toString()	Restitue une chaîne qui décrit l'objet
class getClass()	Obtient la classe d'un objet en phase d'exécution

Classe abstraite

- Il est parfois utile de définir une super classe qui déclare la structure d'une classe sans concrètement proposer une implémentation complète de chaque méthode.
  - L'utilité d'une classe abstraite est de permettre le regroupement (la factorisation) des attributs et méthodes communs à un groupe de classes.
    - En d'autres termes,
      - on définit parfois une super classe qui propose juste une forme générique de ses méthodes que partagent toutes ses classes dérivées
      - Ainsi, chaque classe dérivée, s'occupera elle-même des détails de l'implémentation

Classe abstraite (2)

- Pour déclarer une classe abstraite, il suffit alors d'utiliser le qualificatif abstract devant la définition de la classe.
- Exemple
  - abstract class A{
  - }
- Par conséquent, il n'est pas possible de créer une instance d'une classe abstraite.

Exemple 1

- Prenons l'exemple de la classe Animal. Pour empêcher son instanciation, il suffit de déclarer la classe Animal abstract.
- Dès lors, le compilateur interdira de créer une instance de ce type. Il reste cependant possible d'utiliser des classes abstraites comme super classe ou avec le polymorphisme.

Exemple 1

```
abstract class Animal {
...
}//fin Animal

abstract class Canin extends Animal {//une classe
//abstraite peut dériver d'une autre classe abstraite
...
}//fin Canin
class  Loup extends Canin{
...
}//fin Loup

public class TestCanin {
public void creerCanin () {
Canin c; // possible
c = new Loup (); // possible si une classe Loup dérive
                          //de Canin
c = new Canin (); // impossible :
                    // la classe Canin ne peut être instanciée
}
}
```

- Une classe abstraite ne sert à rien à moins d'être étendue !

Exemple 2

```
abstract class A
{
public void f() { ..... } // f est definie dans A
public abstract void g(int n) ;
// g n'est pas definie dans A ; on n'en a fourni que l'entête
}//fin class A
Bien entendu, on pourra déclarer une variable de type A :
A a ; // OK : a n'est qu'une reference sur un objet de type A
a = new A() ; // erreur : pas d'instanciation d'objets d'une
                    //classe abstraite

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

- on pourra alors instancier un objet de type *B par new B() et même affecter sa référence à une variable de type A :*
  - A a = new B() ; // OK

Méthodes abstraites

- Il est aussi possible de définir des méthodes abstraites. Comme pour les classes abstraites, une méthode abstraite doit obligatoirement être redéfinie dans les sous-classes plus spécifiques.
- Toute classe qui contient au moins une méthode abstraite devient une classe abstraite.
- Deux points importants :
  - Une méthode abstraite n'a pas de corps !
  - Une méthode abstraite est toujours contenue dans une classe abstraite.
- Une classe abstraite peut contenir à la fois des méthodes abstraites et concrètes.

```
abstract public class Canin extends Animal {
public abstract void manger () ; // pas de corps
}
```

Méthodes abstraites (2)

- Elles ne servent qu'à mettre en œuvre le polymorphisme.
  - En réalité, **l'intérêt des méthodes abstraites vient du fait que l'on peut les appeler de la même façon pour tous les objets dérivés.**
- Une méthode abstraite **peut être implémentée dans une sous-classe abstraite**.
- La **première classe concrète dans l'hiérarchie d'héritage** doit implémenter toutes les méthodes abstraites qui n'ont pas encore été implémentées.

Utilisation de final dans l'héritage

- **Première utilisation (méthode « final »)**
  - On utilise parfois **final comme qualificatif d'une méthode** dans la classe base pour **interdire sa redéfinition** dans la classe dérivée.
    - En général, Java fait appel aux méthodes en phase d'exécution.
      - le dynamic binding ou late binding.
      - Toutefois, puisque les méthodes avec le qualificatif final ne peuvent être redéfinies, l'appel à ces méthodes se fait en phase de compilation.
        - On parle de static binding ou early binding.

Utilisation de final dans l'héritage (2)

- **Deuxième utilisation (classe « final »)**
  - On peut également **interdire la dérivation d'une classe** en préfixant la définition de la classe **avec le qualificatif final**.
  - **Une classe final peut évidemment être la fille d'une autre classe** (non final!).
    - Dans ce cas, **aucune autre classe ne peut dériver de cette dernière**.
      - Une classe déclarée final comporte de façon implicite, des méthodes de qualificatif final.
  - Evidemment, **une classe ne peut à la fois être abstract et final**.

Utilisation de final dans l'héritage (3)

- **Syntaxe**
  - final class Exemple{
  - }
- **class Derivee extends Exemple{ }**
  - **génère une erreur du compilateur**.

Exercice 4

- a) Créer une classe objetassure comprenant :
  - les données membres : montant (float), type (chaîne de caractères)
  - un constructeur initialisant les données membres
  - une fonction affiche qui affiche les informations sur un objet objetassure
- b) Créer une classe ordinateur dérivant de la classe objetassure et comprenant :
  - les données membres : ram (int), disque(int)
  - un constructeur initialisant les données membres
  - une fonction affiche1 qui affiche les informations spécifiques sur l'ordinateur
- c) Créer une classe bagage dérivant de la classe objetassure et comprenant :
  - les données membres : type (chaîne de caractères), poids (float)
  - un constructeur initialisant les données membres
  - une fonction affiche2 affichant les informations spécifiques sur le bagage
- d) Proposer une fonction principale qui crée différents objets et teste les différentes méthodes implémentées dans les classes.

Exercice 5

- Un parc auto se compose des voitures et des camions qui ont des caractéristiques communes regroupées dans la classe Véhicule.
- Chaque véhicule est caractérisé par son matricule, l'année de son modèle et son prix.
  - Lors de la création d'un véhicule, son matricule est incrémenté selon le nombre de véhicules créés.
  - Définir le constructeur de la classe Vehicule,
  - Tous les attributs de la classe véhicule sont supposés privés. Ce qui oblige la création des accesseurs (get…) et des modificateurs (set….).
- La classe Vehicule possède également deux méthodes abstraites démarrer() et accélérer() qui seront définies dans les classes dérivées et qui afficheront des messages personnalisés.
- La méthode Affichage() de la classe Vehicule retourne une chaîne de caractères qui contient les valeurs du matricule, de l'année du modèle et du prix.
- Les classes Voiture et Camion étendent la classe Véhicule en définissant concrètement les méthodes accélérer() et démarrer() en affichant des messages personnalisés et en implémentant leur propre constructeur.

Exercice 5(2)

- **Travail à faire:**
  - Créer la classe abstraite Véhicule.
  - Créer les classes Camion et Voiture.
  - Créer une classe TestVehicule qui permet de tester la classe Voiture et la classe Camion.

Surcharge et redéfinition

Redéfinition

- La redéfinition est la possibilité d'utiliser exactement la même signature pour définir une méthode dans une super classe et dans une sous classe
- Des attributs peuvent être aussi redéfinis
- Lorsqu'on redéfinit une méthode d'une super classe, on doit respecter les règles suivantes :
  - Une méthode redéfinie doit avoir une signature identique à la méthode de sa super classe
  - La visibilité peut changer
    - la méthode ne peut pas être moins accessible.

Exemple 1

```
class M {
public boolean faire () {
            System.out.println ("Je travaille");
            return true ;
}
}
class V extends M {
public boolean faire() {
            System.out.println (“Je mange”);
            return true ;
}
}
class Z extends M {
public boolean faire(){
            System.out.println ("Je dors");
            return true ;
}
}
```

Exemple 1 suite

```
public class Testclasse {
public static void main (String[] args) {
        M[] me = new M[3];
        me[0] = new V ();
        me[1] = new Z ();
        me[2] = new M ();
        for (int i = 0; i < me.length ; i++)
        {
                me[i].faire();
        }
}
}
```

Exemple 2

```
class Parent {
protected int x = 1;
public int uneMethode(){
return x;
}
}
class Enfant extends Parent {
private int x; // ce x fait partie de cette classe
public int uneMethode() { // ceci redéfinit la méthode de la classe parent
x = super.x + 1; // accès au x de la classe parent avec super
return super.uneMethode() + x;
}
}
```

Exemple 2 suite

```
public class Test {
public static void main(String[] args) {
Enfant P1 = new Enfant();
System.out.println(P1.uneMethode());
Parent P2 = new Parent();
System.out.println(P2.uneMethode());
}
}
```

Exemple 3

```
class Point
{
protected int x, y ;
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) ;
}
}
```

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

Exemple 3 suite

```
public class Poly
{
public static void main (String args[])
{
Point p = new Point (3, 5) ;
p.affiche() ; // appelle affiche de Point
Pointcol pc = new Pointcol (4, 8, " Jaune ") ;
p = pc ; // p de type Point,fait référence à un objet de type Pointcol
p.affiche() ; // on appelle affiche de Pointcol. La méthode affiche est
                //  accessible dans la sous classe si elle est redéfinie
p = new Point (5, 7) ; // p fait référence à nouveau à un objet de type Point
p.affiche() ; // on appelle affiche de Point
}
}
```

Exemple 3 suite (fonction main alternative)

```
public class Poly2
{
public static void main (String args[])
{
Point [] tabPts = new Point [4] ;
tabPts [0] = new Point (0, 2) ;
tabPts [1] = new Pointcol (1, 5, “Jaune ”) ;
tabPts [2] = new Pointcol (2, 8, “ Noir ”) ;
tabPts [3] = new Point (1, 2) ;
for (int i=0 ; i< tabPts.length ; i++)
        tabPts[i].affiche() ;
}
}
```

La surcharge de méthode

- Dans Java, il est possible de créer dans une même classe, plusieurs méthodes qui portent le même nom mais avec différents paramètres et/ou valeurs de retour.
- Lorsqu'on surcharge une méthode, on se doit de respecter les règles suivantes :
  - Les types de retour peuvent être différents ;
  - Il n'est pas possible de ne changer que le type de retour. Il faut impérativement modifier les paramètres ;
  - Le niveau d'accès ou de visibilité peut varier

La surcharge de méthode (2)

- **Exemple**

```
public class ArithmLib
{
public static double min(int i, double d){
     ..........
}
public static double min(double d, int i){
     ..........
}
}
```

Paquetages et interfaces

Paquetage

- Un grand nombre de classes, fournies par Java SE, implémentent des données et traitements génériques utilisables par un grand nombre d'applications. Ces classes forment l'API (*Application Programmer Interface)* du langage Java.
- Toutes ces classes sont organisées en *packages* (ou bibliothèques) dédiés à un thème précis.

Paquetage (2)

Package	Description
java.awt	Classes graphiques et de gestion d'interfaces
java.io	Gestion des entrées/sorties
java.lang	Classes de base (importé par défaut)
java.util	Classes utilitaires
javax.swing	Autres classes graphiques

Importation des paquets

- Toutes les classes standard de Java sont mémorisées dans un paquet appelé **java**
- Les fonctions élémentaires du langage Java sont mémorisées dans un sous paquet de java qui est java.lang
- Ce dernier paquet est importé automatiquement par défaut dans tous les compilateurs Java.
- Sinon pour l'importer soi même, il faut faire :
  - import java.lang.* ;

Importation des paquets (2)

- **import** pour permettre la visibilité de certaines classes ou des paquets entiers.
- Syntaxe
  - import NomPaquet1.NomPaquet2.NomClasse ; ou
  - import NomPaquet1.NomPaquet2.* ;
- Exemple
  - import java.util.Date ;
  - import java.io.* ;

```
import java.util.Date ;
public class DateMain {
public static void main(String[] args) {
Date today = new Date();
System.out.println("Nous sommes le " + today.toString());
//affiche le jour et l'heure
}
}
```

Création de paquet

- Il est possible de créer vos propres packages en précisant, avant la déclaration d'une classe, le package auquel elle appartient. Pour assigner la classe précédente à un package, nommé test.premierpaquet, il faut modifier le fichier de cette classe comme suit :

```
package test.premierpaquet;
import java.util.Date ;
public class DateMain {
...
}
```

Interface

- Une interface est un type, au même titre qu'une classe, mais abstrait et qui donc ne peut être instancié (par appel à new plus constructeur).
- Une interface décrit un ensemble de signatures de méthodes, sans implémentation, qui doivent être implémentées dans toutes les classes qui *implémentent* l'interface.
- L'utilité du concept d'interface réside dans le regroupement de plusieurs classes, tel que chacune implémente un ensemble commun de méthodes, sous un même type.

Interface (2)

- Une interface possède les caractéristiques suivantes :
  - elle contient des signatures de méthodes ;
  - elle ne peut pas contenir de variables ;
    - elle peut contenir des constantes
  - une interface peut hériter d'une autre interface (avec le mot-clé extends) ;
  - une classe (abstraite ou non) peut implémenter plusieurs interfaces.
    - La liste des interfaces implémentées doit alors figurer après le mot-clé implements placé dans la déclaration de classe, en séparant chaque interface par une virgule.

Interface (3)

- Les interfaces constituent une solution proposée en Java pour l'héritage multiple
- Déclaration d'une interface
  - qualificatif interface NomInterface {
  - }
- Implémentation de l'interface par une classe
  - mot clé implements
  - La syntaxe est la suivante :
    - qualificatif class NomClasse implements NomInterface {
    - }
- Une classe peut implémenter plusieurs interfaces.
  - Dans ce cas, on utilise la virgule comme séparateur.
  - La syntaxe est suivante :
    - qualificatif class NomClasse implements NomInterface 1, NomInterface 2{ }

Exemple

```
interface Forme {
public double surface() ;
public void affiche() ;
}
```

```
class Rectangle implements Forme {
…
}
```

```
class Cercle implements Forme {
…
}
```

Classes internes

- Une classe peut contenir la définition d'une autre classe. Considérons l'exemple suivant :

```
public class Magasin {
private class article{
// on définit la structure
private String code;
private String nom;
private double prix;
private int stockActuel;
private int stockMinimum;
// constructeur
public article(String code, String nom, double prix, int stockActuel, int
stockMinimum){
// initialisation des attributs
this.code=code;
this.nom=nom;
this.prix=prix;
this.stockActuel=stockActuel;
this.stockMinimum=stockMinimum;
}//constructeur
```

Classes internes (2)

```
//méthode toString
public String toString(){
    return "Description :
Article("+code+","+nom+","+prix+","+stockActuel+","+stockMinimum+")";
}//toString
}//fin classe article
// Attribut de la classe Magasin
private article art;
// constructeur
public Magasin(String code, String nom, double prix, int stockActuel,
int stockMinimum){
// définition attribut
    art=new article(code, nom, prix, stockActuel,stockMinimum);
}//Magasin
// accesseur
public article getArticle(){
    return art;
}//getArticle
```

Classes internes (3)

```
public static void main(String[] args) {
        // création d'une instance Magasin
Magasin t1=new Magasin("V100","velo",50000,10,5);
// affichage Magasin.art
System.out.println(t1.getArticle());
}//fin main
}//fin Magasin
```

Exercice 6

- Créez une classe Liquide contenant une méthode imprimer() qui affiche : "je suis un liquide"
- Créez deux classes dérivées de la classe Liquide, les classes Cafe et Lait, dont les méthodes imprimer() affichent respectivement "je suis un Café" et "je suis du Lait"
- Enfin vous créerez une classe Tasse dans laquelle vous déclarez un attribut qui est objet de la classe Liquide appelé L. Définissez également une méthode AjouterLiquide (Liquide li) et une méthode imprimer() implémentée de telle sorte qu'on puisse connaître le liquide contenu dans une tasse.
- Créer une fonction main dans une autre classe appelée Testtasse qui crée un tableau de Tasses contenant différents liquides. Appeler la méthode imprimer() sur les objets du tableau.

Exercice 7

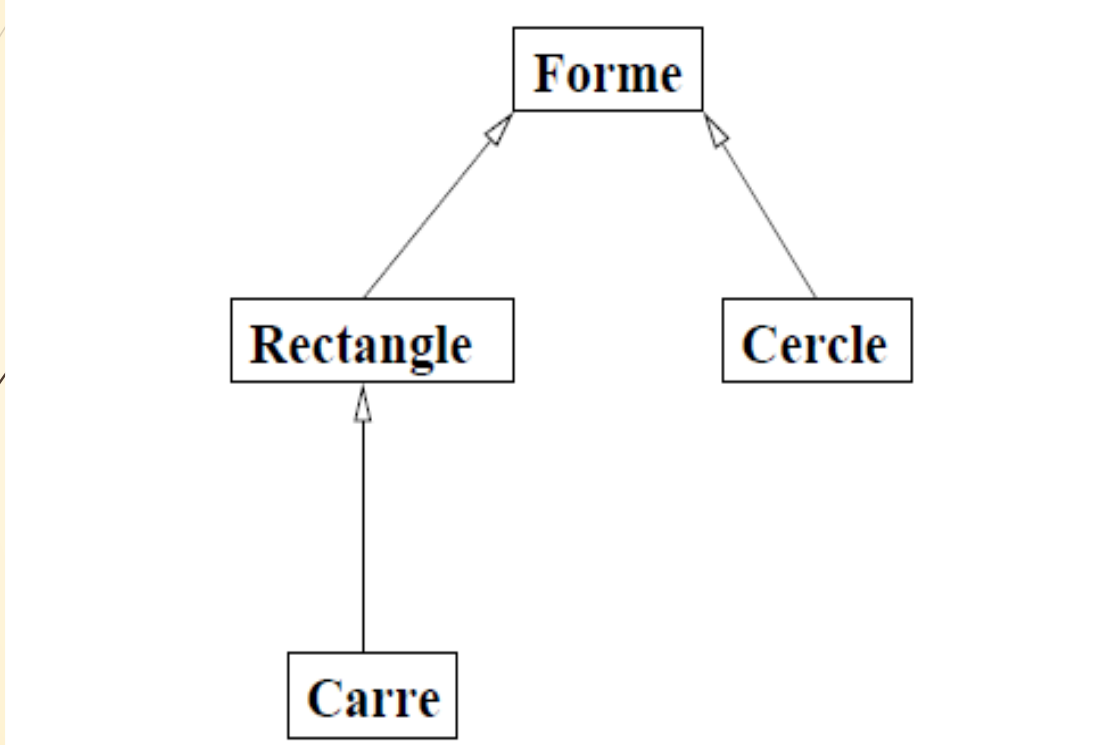
- 1- Créer un fichier nommé TestVehicule1.java
  - Créez dans ce fichier la classe Vehicule qui contient les données
    - private boolean moteur
    - private int vitesseMax
  - La classe Vehicule contient également les méthodes suivantes :
    - Constructeur qui initialise les attributs
    - String toString(), qui renvoie les caractéristiques du véhicule c'est-à-dire donne des informations sur le moteur et la vitesse maximale d'un véhicule,
    - void Vmax() qui affiche la vitesse maximale du véhicule.
  - Créez ensuite dans ce même fichier .java une classe Voiture_Composee dont les membres sont :
    - private Vehicule ve,
    - private int nombreDePortes,
- 2- Recréer dans un autre fichier nommé TestVehicule2.java, la classe Vehicule avec les attributs et méthodes précédemment cités sauf que les attributs ont ici une visibilité protected.
  - Créer dans ce même fichier une autre classe nommée Voiture_Derivee qui dérive de la classe Vehicule avec la donnée membre private int nombreDePortes.

Exercice 7 (suite)

- 3- Pour chacune des classes, dans les deux fichiers .java, définissez leur constructeur et leur méthode String toString().
- 4- Définir la fonction principale dans chacun des fichiers .java et y créer des objets de type Voiture_Composee d'une part et des objets de type Voiture_Derivee d'autre part. Faire afficher les informations relatives à chacun des objets en faisant appel à la méthode toString().
- 5- Comparez les deux implémentations.
- 6- Comment accède-t-on aux différents attributs des classes Voiture_Composee et Voiture_Derivee depuis les fonctions principales?
  - Implémenter les solutions pour rendre les accès possibles.
- 7- Affichez depuis les fonctions principales de chaque type d'implémentation, la vitesse maximale d'un véhicule.

Exercice 8

- Considérer la hiérarchie de classes suivante :



Exercice 8 (suite)

Définir une interface Forme qui est donc abstraite, qui déclare la méthode surface qui retourne la surface d'une forme et la méthode affiche qui imprime un message contenant les caractéristiques d'une forme.

- Définir les classes Rectangle et Cercle qui implémentent chacune l'interface Forme. Elles doivent donc redéfinir les méthodes surface et affiche.
- En plus de redéfinir ces deux méthodes, chacune de ces 2 classes définit des attributs qui lui sont propres (longueur, largeur, rayon) et un constructeur.
- Définir en plus, dans la classe Rectangle, les méthodes getLargeur() et getLongueur() qui retourne la longueur et la largeur d'un objet de la classe.
- Définir la classe Carre qui dérive de la classe Rectangle et qui redéfinit aussi les méthodes surface et affiche. Cette classe définit aussi son constructeur en faisant appel au constructeur de la super classe Rectangle.
- Ecrire enfin, la fonction principale qui crée différents objets des différentes classes et teste leurs différentes méthodes.