

Initiation à la programmation orientée objet

Problématique du développement logiciel

La construction d'un logiciel est complexe quand elle met en oeuvre de nombreuses ressources

1. humaines
2. matériels
3. technologiques

Problématique du développement logiciel

D'où la nécessité de suivre :

1. Un processus bien défini : le cycle de vie d'un logiciel

- Prévoir et planifier les travaux
- Coordonner les activités de conception, de fabrication et de validation
- Réagir à l'évolution des objectifs

2. Une méthode rigoureuse basée sur des modèles

- Représentations sémantiques simplifiées d'un système visant à l'analyser et à le comprendre pour mieux le concevoir

Les étapes du cycle de vie

1. L'expression des besoins (client, fournisseur)
 - Les fonctionnalités du système étudié
 - Comment utiliser ce système ?
2. Les spécifications du système (utilisateur, expert, fournisseur)
 - Lever les ambiguïtés, éliminer les redondances du cahier des charges
3. L'analyse (utilisateur, expert, fournisseur)
 - Phase indépendante de toute considération technique et informatique visant à définir le système (s'accorder sur le «quoi»)

Les étapes du cycle de vie

4. La conception (expert en informatique)

- Prise en compte de l'environnement technique pour déterminer la manière de résoudre le problème posé (s'accorder sur le «comment»)

5. L'implémentation (expert en informatique)

- Traduction de la conception dans un langage de programmation, en une base de données, etc.

6. Les tests (expert en informatique)

- Vérifier que l'implémentation est correcte

Les étapes du cycle de vie

7. La validation (utilisateur, expert, fournisseur)

- Vérification que le système correspond aux besoins

8. La maintenance et l'évolution pendant la phase d'exploitation

L'approche orientée objet

La qualité d'un logiciel

On peut mesurer la qualité d'un logiciel par :

1. **L'exactitude** : aptitude d'un programme à fournir le résultat voulu et à répondre ainsi aux spécifications
2. **La robustesse** : aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation
3. **L'extensibilité** : facilité avec laquelle un programme pourra être adapté pour répondre à l'évolution des spécifications
4. **La réutilisabilité** : possibilité d'utiliser certaines parties du logiciel pour résoudre un autre problème
5. **La portabilité** : facilité avec laquelle on peut exploité un même logiciel dans différentes implémentations
6. **L'efficacité** : temps d'exécution, taille mémoire

La qualité d'un logiciel

La programmation structurée a permis d'améliorer :

1. La robustesse
2. L'exactitude

Etendre et réutiliser : casser les modules intéressants

Les limites de l'approche fonctionnelle

- L'approche fonctionnelle : décomposition hiérarchique en fonctions
- Le découpage impose la hiérarchie : le couplage fonction/ hiérarchie est statique
- Conséquence : l'évolution nécessite des modifications lourdes
- Un seul état partagé (mémoire) et toute fonction peut agir sur n'importe quelle partie : code difficilement réutilisable

L'approche objet

On considère ce que le système doit faire, mais aussi ce qu'il est

1. Ce que le système est : décomposition en objets (représentation du monde réel)
2. Ce que le système fait : les objets communiquent entre eux par appels de fonctions (méthodes) ; envoi dynamique des messages

Avantage : l'évolution ne remet en cause que l'aspect dynamique sans remettre en cause les objets

L'approche objet

Différence fondamentale entre la programmation procédurale et la programmation objet :

1. En programmation procédurale :
décomposition de tâches en sous-tâches
2. En programmation objet : on identifie les acteurs (les entités comportementales) du problème puis on détermine la façon dont ils doivent interagir pour que le problème soit résolu.

Un objet

Objet = Etat + Comportement [+ Identité]

1. Etat : valeurs des attributs (données) d'un objet
2. Comportement : opérations possibles sur un objet déclenchées par des stimulations externes (appels de méthodes ou messages envoyées par d'autres objets)
3. Identité : chaque objet à une existence propre (il occupe une place en mémoire qui lui est propre). Les objets sont différenciés par leurs noms

L'approche objet

On embarque au sein d'une entité (un objet) son état (mémoire)

Une voiture

Bleu
867 kgs

Etat

Une personne

Lucie
20 ans
Célibataire

Etat

Exécution d'un programme objets

- **En séquentiel** : une succession d'envois de messages qui modifient de proche en proche les états des objets.
- **Programmes distribués** : les objets s'exécutent et s'envoient des messages en parallèle.

Pour résumer

- **Un objet** : est une entité autonome regroupant un état et les fonctions permettant de manipuler cet état
- **Attribut** : est une des variables de l'état d'un objet

Les classes

Un système complexe a un grand nombre d'objets. Pour réduire cette complexité, on regroupe des objets en classes.

- Une classe est une description d'un ensemble d'objets ayant une structure de données commune (attributs) et disposant des mêmes méthodes.

- Classe = Type

Classe et Instance

- Une classe définit un ensemble d'objets ayant des propriétés communes. Les objets d'une même classe ont en commun des attributs et des méthodes.
- Une classe peut être vue comme un modèle d'objet
- Une instance d'une classe est l'un des objets représentant la classe

Représentation graphique d'une classe

Nom de la classe

Attribut 1

Attribut 2

...

Méthode 1

Méthode 2

...

Identifier les objets

A partir d'un cahier des charges

1. liste des mots clefs (tout mot clef peut être une classe)
2. réduire la liste en supprimant :
 - les synonymes
 - les classes non pertinentes
3. Les classes peuvent évoluer : ajout de nouvelles classes lors de l'implémentation pour créer une liste de classes par exemple.

La difficulté d'identifier les classes :

4. quand l'application contient des objets physique
5. quand l'application est abstraite

Ajouter des attributs et des méthodes

Les mots clefs qui n'ont pas été utilisés peuvent être des attributs

1. Parfois difficile de choisir entre classe et attribut.

- Un attribut est caractérisé par sa valeur
- Une classe est caractérisée par ses attributs et ses méthodes

2. L'attribut d'une classe peut être un objet (agrégation)

Les services que doivent rendre une classe sont ses opérations

Objets et Le Langage Java

Caractéristiques essentielles de java

Les concepteurs de Java (James Gosling et Bill Joy, Sun Microsystems) ont conçu Java comme un langage :

- Indépendant de la machine et du système d'exploitation sous-jacent
- Complet
 - Programmation réseau avancée
 - Programmation d'interface utilisateur graphique
 - ...
- Simple
 - Objet (syntaxe proche du C/C++)
 - La gestion de la mémoire n'est pas à la charge d'utilisateur

La machine virtuelle de Java

Fichiers sources (.java)



Compilation (javac)

Code exécutable (byte-code)

.class

librairies externes (byte-code)

.class



Java Virtual Machine



OS (Linux, Mac OS, Windows, ...)

Un premier programme Java

HelloWorld.java

```
public class HelloWorld {  
    public static void main (String args[]) {  
        System.out.println ("Hello World") ;  
    }  
}
```

Compilation : `javac HelloWorld.java` produisant le fichier `HelloWorld.class`

Exécution (interprétation) : `java HelloWorld`

Les types de base en JAVA

Java est un langage fortement typé

1. **Boolean** : true ou false
2. **char** : caractère sur 16 bits Unicode
3. **byte** : entier signé sur 8 bits
4. **short** : entier signé sur 16 bits
5. **int** : entier signé sur 32 bits
6. **long** : entier signé sur 64 bits
7. **float** : flottant sur 32 bits
8. **double** : flottant sur 64 bits

Les instructions de base en Java

Les instructions en Java sont fortement inspirés du langage C

1. Les instructions conditionnelles : `if ... else`, `switch, ...`
2. L'instruction d'affectation : `=`
3. Les instruction de répétition : `for, while, repeat`

Les instructions arithmétiques et logiques sont aussi (quasi) identiques avec le langage C

Initialisation des variables en Java

Toute variable locale à une méthode doit être initialisée avant d'être utilisée

```
void illustration(){  
    int i, j ;  
  
    i++ ; //Erreur de compilation  
  
    j = 0 ;  
  
    if (j == 1) i = 0;  
  
    i++ ; //Erreur : i n'est toujours pas initialisé  
}
```


Les tableaux en Java

- Déclaration d'un tableau à une dimension :
`type nom[] ;`
- Dimensionnement d'un tableau (l'opérateur `new`) :
`nom = new type [taille] ;`
- Construction des éléments d'un tableau

```
for (int i = 0 ; i < taille ; i++)  
    nom[i] = new type() ; //usage du constructeur
```
- Longueur d'un tableau : `nom.length`

Les chaînes de caractères en Java

- Déclaration d'une chaîne de caractères :
 - `String chaine ;`
- Affectation d'une chaîne :
 - `chaine = "Java" ;`
- Concaténation :
 - `chaine = chaine + " et le reste" ;`
- Longueur d'une chaîne : `chaine.length() ;`

Les classes en Java

```
class Point {
```

```
    int abscisse ;
```

```
    int cordonnee ;
```

```
    String label ;
```

Attributs

```
    void Initialisation (int x, int y, String l) {
```

```
        abscisse = x ; cordonnee = y ; label = l ;
```

```
    }
```

```
    void translation (int xd, int yd) {
```

```
        abscisse +=xd ; coordonnee +=yd ;
```

```
    }
```

Méthodes

```
}
```


Création d'un objet

1. Déclaration d'un objet (instance d'une classe) : `Point a ;`
2. Allocation de l'espace mémoire de l'objet a : `a = new Point() ;`

Manipulation d'un objet

- Accès un à attribut : nom de l'objet • nom de l'attribut
 - Exemple : `a.abscisse` ;
- Appel d'une méthode : nom de l'objet • nom de la méthode ;
 - Exemple : `a.translation(1, 5)` ;

Java vs. C

```
Type struct{
    int solde;
    int numero;
    char *proprietaire;
}Compte;

void creation (Compte *c, int num, char *prop){
    c->solde = 0;
    c->numero = num;
    c->proprietaire = prop;
}

void depot (Compte* c, int montant){
    c->solde = c->solde + montant;
}

void retrait (Compte*c, int montant){
    c->solde = c->solde - montant;
}
```

```
class Compte{
    int solde;
    int numero;
    String proprietaire;

    void creation (int num, String prop){
        solde = 0;
        numero = num;
        proprietaire = prop;
    }

    void depot (int montant){
        solde = solde + montant;
    }

    void retrait (int montant){
        solde = solde - montant;
    }
}
```


Java vs. C

```
Compte* unCompte;  
unCompte =  
(Compte*)malloc(sizeof(Compte));  
  
creation (unCompte, 1234, "Jean Dupont")  
depot (unCompte, 1000);  
retrait (unCompte, 450);  
printf(`Il reste %f a %s",  
        unCompte->solde,  
        unCompte->proprietaire);
```

```
Compte unCompte ;  
unCompte = new Compte();  
  
unCompte.creation (1234, "Jean Dupont");  
unCompte.depot (1000);  
unCompte.retrait (450);  
System.out.println("Il reste " +  
                    unCompte.slode + " a " +  
                    unCompte.proprietaire);
```


Modification d'un objet

- On peut ne pas préfixer un nom d'attribut ou de méthode si on l'écrit à l'intérieur d'une méthode.
- L'attribut ou la méthode désignés sont alors ceux de l'objet auquel appartient la méthode où ils sont écrits.
- Exemple : on ajoute la méthode suivante à la classe Compte :

```
void virement (int montant, Compte destinataire){  
    retrait (montant);  
    destinataire.depot (montant);  
}
```


Surcharge des méthodes

- Plusieurs méthodes peuvent avoir le même nom. Elles doivent se différencier par la liste de leurs arguments

- Exemple :

```
void methode (int i, int j) {...};
```

```
void methode (int i, char j) {...};
```

- Le choix de la méthode à appeler se fait au moment de la compilation : celle qui correspond au mieux à l'appel qui est choisie

Surcharge des méthodes

- Le type de retour d'une méthode n'est pas suffisant pour différencier deux méthodes pourtant le même nom (une erreur est alors générée par le compilateur).
- Emploi de la surcharge : un utilisateur dispose de plusieurs méthodes réalisant le même traitement pour des arguments différents.

Constructeur en Java

- Un constructeur est méthode particulière portant le même nom que la classe la contenant
- Le constructeur est appelé automatiquement lors de la création de l'objet (par la méthode new)
- La syntaxe d'un constructeur est la même que celle d'une méthode ordinaire. Elle n'indique cependant aucun type de paramètre de retour

Constructeur en Java

```
public class Tableau{  
    int[] T ;  
    public Tableau (int nbElements){  
        T = new int [nbElements] ;  
    }  
    ...  
}
```


Constructeur en Java

- Un constructeur est utile pour initialiser les attributs de l'objet qu'il crée.
- Dans la classe Point :

```
Point (int x, int y, String l) {
```

```
    abscisse = x ; cordonnee = y ; label = l ;
```

```
}
```

- Appel : `Point c = new Point (5, 10, "c") ;`

Constructeur en Java

- Un constructeur ne peut pas être déclaré **abstract, synchronized ou final**
- Il peut y avoir plusieurs constructeurs qui se différencient par la liste de leurs paramètres
- Un constructeur ne peut être appelé explicitement
- A toute classe est associé un constructeur par défaut : sans paramètres et n'effectue aucune opération

Constructeur en Java

Un constructeur peut appeler un autre constructeur de la même classe placé avant lui en utilisant le mot clé **this** servant à **l'auto-référencement** (qui doit être la première instruction)

```
public class Voiture {  
  
    int nbPortes ; String marque ;  
  
    Voiture (String marque, int nbPortes){  
  
        //marque = marque ; nbPortes = nbPortes ; Problématique !!  
  
        this.marque = marque ; this.nbPortes = nbPortes ;  
  
    }  
  
    Voiture (String marque){  
  
        this (marque, 5) ;  
  
    }  
}
```


Initialisation par défaut

- Les attributs d'un objet sont toujours initialisés par défaut. Initialisation à :
 - 0 de tout attribut nombre
 - null à toute référence à un objet
- Initialisation explicite de la valeur d'un attribut lors de sa déclaration
 - `int abscisse = 0 ;`

Destruction d'objets

- La libération des ressources mémoires acquises avec l'opérateur new est automatique (grâce au ramasse-miettes) : la mémoire d'un objet qui n'est plus référencé peut être récupérée à tout moment

- Exemple :

```
Voiture voiture1 = new Voiture() ;
```

```
Voiture voiture2 = new Voiture() ;
```

```
voiture1 = null ; // la première voiture n'est plus référencée
```


Destruction d'objets

- Avant de détruire un objet, on fait appel à la méthode `void finalize()`, si celle-ci est définie
- Les ressources non Java doivent être libérées explicitement (les fichiers par exemple)

Exemple :

```
public class Fichier {  
    FileInputStream f ;  
    Fichier (String nom) { ... }  
    public void close() {  
        if (f != null) {f.close() ; f = null ; }  
    }  
    protected void finalize () throws Throwable {  
        super.finalize() ;  
        close () ;  
    }  
}
```


Les membres statiques

Chaque objet a ses propres données propres

```
public class Compteur{  
    int i ;  
    public Compteur (int j){i = j ;}  
    public static void main(){  
        Compteur objet1 = new Compteur (1) ;  
        Compteur objet2 = new Compteur (2) ;  
    }  
}
```

<u>objet1</u>
i = 1

<u>objet2</u>
i = 2

Les membres statiques

- **Static** : partager un membre entre toutes les instances d'une classe

```
public class Compteur{
```

```
    static int cpt ;
```

```
    int i;
```

```
    public Compteur (int x){i=x; cpt++ ;}
```

```
    public static void main(){
```

```
        Compteur objet1 = new Compteur (1) ;
```

```
        Compteur objet2 = new Compteur (2) ;
```

```
    }
```

```
}
```

objet2

objet1

objet1

i = 1

cpt = 2

objet2

i = 2

Bloc d'initialisation statique


Les champs statiques peuvent être initialisés dans un bloc static

```
public class Tableau{  
    //Création d'un tableau  
    static int[] tableau = new int [10] ;  
    //Initialisation du tableau  
    static{  
        for (int i = 0 ; i < tableau.length ; i++)  
            tableau[i] = i*i ;  
    }  
    //....  
}
```


Méthodes statiques

- Commune à toutes les instances d'une classe
- Ne peuvent accéder qu'aux données membres statiques

Méthode appelée
par le nom de la
classe



```
public class Compteur{
```

```
    static private int cpt ;
```

```
    public Compteur(){cpt++ ;}
```

```
    static int combien(){return cpt ;}
```

```
    protected void finalize throws Throwable{
```

```
        super.finalize() ; cpt-- ;
```

```
    }
```

```
    public static void main(){
```

```
        Compteur objet1 = new Compteur () ;
```

```
        int cpt = Compteur.combien() ;
```

```
    }
```

```
}
```


Les références

- Java ne dispose pas de pointeurs, mais utilise uniquement des références

A reference = new A() ; // création d'un objet référencé

reference = null ; // l'objet n'est plus référencé donc plus accessible

- Porté des références :

{ //sous-programme

A reference = new A() ; // une référence sur un objet créé

A referencesupplementaire = reference ;

}

La visibilité des 2 références s'arrêtent ici, l'objet n'est plus accessible

Intérêts des références

- Les références sont utilisées pour passer ou retourner des paramètres à une méthode

```
void f (A reference){ // nouvelle référence
```

```
    reference = ... //agit directement sur l'objet référencé
```

```
}
```

- Il est possible de retourner une référence sur un objet créé dans une méthode

```
A g() {
```

```
    A reference = new A() ; //Création d'une référence
```

```
    return reference ; //transmission de la référence
```

```
}
```

```
A reference = g() ; //l'objet a toujours une référence
```


Propriétés des références

- Une référence peut référencer successivement plusieurs objets

```
void f (A reference){ //nouvelle référence sur un objet créé à l'extérieur de cette méthode  
    reference = new A() ; //décrémente le nombre de références sur l'objet extérieur à cette méthode  
}
```

- Le passage d'arguments à une méthode se fait toujours par valeur (c'est la valeur d'une référence qui est transmise)
- Comment changer d'objet référencé dans une méthode ?

```
void g (B b) {  
    b.setReference(new A());  
}  
  
Class B{  
    A objet ;  
    public void setReference (A objetprime){objet = objetprime ;}  
}
```


Usage des références

- Copie et comparaison d'objets

Class A { ... } A a = new A(), b = new A();

- $a == b$: se sont les adresses qui sont comparées
- $a = b$: c'est l'adresse de b qui est copiée dans a

Comparer des objets

Les attributs des objets sont comparés deux à deux.

```
class Point {
```

```
    int x, y ;
```

```
    boolean egale (Point p) {
```

```
        return (p.x == x) && (p.y == y) ;
```

```
    }
```

```
}
```


Copier des objets

- Une instance de la même classe est créée
- Les attributs des objets sont copiés deux à deux

```
class Point {  
  
    int x, y ;  
  
    Point copie () {  
  
        Point q = new Point() ;  
  
        q.x = x ; q.y = y ;  
  
        return q ;  
  
    }  
  
}
```


Composition de classes

- **Principe** : Les attributs d'un objet peuvent eux-mêmes être des objets
- **En réalité** : comme pour une variable dont le type est une classe, un attribut ne mémorise que la référence de son objet
- **Remarque** : la composition peut-être récursive : un attribut d'une instance de la classe A peut être une instance de A

```
class A {  
  
    B b ;  
  
    A a;  
  
}
```


Composition : comparaison et copie

- Comparaison : il faut comparer récursivement les attributs
- Copie : il faut copier récursivement les attributs

```
class A {  
    B b ;  
    A a;  
    A copie(){  
        A copieA = new A();  
        copieA.b = b.copie();    //Appel à l'opérateur de copie de B  
        //Appel récursif à l'opérateur de copie de la classe A  
        if (a == null) copieA.a = null;  
        else copieA.a = a.copie();  
        return copieA;  
    }  
}
```


Encapsulation

- **Principe** : masquer le plus possible les détails d'implémentation et le fonctionnement interne des objets
- **Effet** : découpler les classes constituant un programme afin que la modification de la structure interne de l'une n'oblige pas à modifier l'autre

Niveaux d'accès

- En java, on peut restreindre l'accès à des classes, des méthodes et des attributs
- Cette restriction les rend non référençable en dehors de leurs niveaux d'accès

Niveaux d'accès

Concernant les membres (méthodes et attributs) des classes, il existe 4 niveaux d'accès :

1. **public (public)** : accessible de n'importe où.
2. **protégé (protected)** : cf. héritage
3. **privé paquetage** : pas de mot-clé, niveau d'accès par défaut, accessible uniquement depuis les autres classes du paquetage où il est déclaré
4. **privé (private)** : accessible uniquement depuis l'intérieur de la classe où il est déclaré

Niveaux d'accès

- Concernant les classes, il y a deux niveaux d'accès public ou privé
paquetage
- Dans un même fichier, **une seule classe au maximum** peut être déclarée public
- Si une classe contient la méthode de classe main alors c'est cette classe qui doit être public

Restrictions des accès

Principe de base : rendre tout le moins accessible possible

- Par défaut, tous les attributs et les méthodes sont privées paquetage
- L'allègement de toute restriction doit être motivée

Restrictions des accès

Si l'accès d'une classe est privé paquetage alors on sait que si on la modifie ou on la supprime, aucune classe extérieure au paquetage n'en sera affectée. Il n'y aura qu'à modifier éventuellement certaines classes du paquetage

Restrictions des accès

Si l'accès d'un membre d'une classe est privé (privé paquetage) alors on pourra la modifier ou le supprimer sans qu'une autre classe (aucune classe en dehors du paquetage) ne soit affectée.

Restrictions des accès

- **En pratique** : dans toutes les classes tous les attributs seront privés.
- **Exemple** : dans la classe Compte, on rend privé l'accès aux attributs et à la méthode retrait à un seul paramètre, afin qu'un objet extérieur ne puisse pas modifier une instance de Compte n'importe comment.

Restrictions des accès

```
class Compte{  
    private int solde, numero;  
    private String proprietaire;  
    public Compte(int numero, String prop){...}  
    public void depot(int montant){...}  
    private void retrait(int montant, String prop){...}  
    public void virement(int montant, Compte dest){...}  
}
```


Méthodes d'accès

Remarque : on peut avoir besoin de connaître le solde d'un compte.

Comme on doit interdire de modifier l'attribut solde de l'extérieur de la classe, on va simplement écrire la méthode :

```
public int solde(){return solde;}
```


Méthodes d'accès

De façon plus générale : si on a besoin de connaître la valeur d'un attribut ou de le modifier, on effectuera cette opération via une méthode (non privée) dites d'accès :

1. **accesseur** : méthode fournissant la valeur d'un attribut

2. **modifieur** : méthode transmettant une nouvelle valeur à un attribut

Exemple

Liste chaînée dont on veut connaître la longueur (`int getLongueur()`)

1. Soit la méthode parcourt la liste chaînée à chaque fois
2. Soit la méthode lit l'attribut longueur. Il est modifié par les autres méthodes (ajout ou suppression de chaînons)

Encapsulation «interne»

On peut même limiter l'accès aux attributs dans la classe où ils sont déclarés : on y accède que grâce à leurs méthodes d'accès

Intérêt : si les attributs changent, seuls les méthodes d'accès doivent être modifiées

A ne plus faire

```
class A{  
    X x; Y y;  
    void f(...){ x = ...}  
    int g(...){ int a = y; ...}  
}
```

On change les attributs alors on change toutes les classes qui les désignent.

A faire au moins

```
class A{  
    private X x; private Y y;  
    X getX(){return x;}  
    void setX(X x){this.x=x;}  
    Y getY(){return y;}  
    void setY(Y y){this.y=y;}  
    void f(...){ x = ...}  
    int g(...){ int a = y; ...}  
}
```

On change les attributs alors on change toutes les méthodes de la classe

Le mieux à faire

```
class A{  
    private X x; private Y y;  
    X getX(){return x;}  
    void setX(X x){this.x=x;}  
    Y getY(){return y;}  
    void setY(Y y){this.y=y;}  
    void f(...){ setX(...); ...}  
    int g(...){ int a = getY(); ...}  
}
```

On change les attributs alors on ne change que les méthodes d'accès de la classe

Paquetage

- Un paquetage (package) est le regroupement sous un nom d'un ensemble de classes
- Les paquetages de Java sont hiérarchisés sous la forme d'une arborescence
- Un paquetage se désigne par un ensemble d'identifiants séparés par des points
- Exemple : `java.awt.geom` est un paquetage de Java contenant des classes définissant des objets géométriques

Paquetage

- A l'arborescence logique des paquetages correspond l'arborescence physique des répertoires mémorisants les fichiers *.class des paquetages
- Exemple : les classes du paquetage `java.awt.geom` sont stockées dans le répertoire `java/awt/geom` à partir du répertoire racine des paquetages

Paquetage

- Pour inclure les classes d'un fichier source dans un paquetage, il faut placer la commande `package MonPaquetage;` en tête du fichier
- Les fichiers *.class créés sont stockés dans le répertoire MonPaquetage

Paquetage

Pour utiliser les classes d'un paquetage, 3 solutions :

- Expliciter le paquetage de la classe en préfixant le nom de la classe par son nom de paquetage :
`MonPaquetage.A a = new A();`
- Utiliser en tête du fichier l'instruction `import` suivi du nom du paquetage et la classe : `import MonPaquetage.A;` On n'a alors plus besoin de préfixer le nom de la classe dans le reste du fichier : `A a = new A();`
- Importer toutes classes d'un paquetage : `import Monpaquetage.*;` Plus besoin de préfixer aucun nom de classe du paquetage

Paquetage

- Un package (paquetage) est un ensemble de classes
- Les classes d'un même package sont compilées dans un répertoire portant le nom du package
- Le mot clef package doit apparaître en premier dans un fichier source

Fichier Distributeur.java

```
package banque ;  
public class Distributeur {  
    ....  
}
```

- `javac -d . Distributeur.java` //Compilation
 - Dans le répertoire courant : `Distributeur.java`
 - Dans le répertoire banque : `Distributeur.class`

Paquetage

- Par défaut, un champ est accessible depuis le code situé dans des fichiers différents du même package

```
package banque ;
```

```
public class Distributeur{  
    int solde = 1000 ;  
}
```

```
package banque ;
```

```
public class Banque {  
    public static void main(String[] argv){  
        Distributeur d = new Distributeur() ;  
        d.solde+=100 ;  
    }  
}
```

- Les champs sans modificateur de visibilité ne sont pas accessibles depuis un autre package

- Utiliser un package : **import**

Encapsulation

- Rappel -

- Pour accéder à l'état d'un objet (à ses données) il faut utiliser des opérateurs (méthodes)
- L'état d'un objet est caché en son sein : pour changer l'état d'un objet, il faut lui envoyer un message (appeler une fonction)

Encapsulation et niveaux d'accès

Niveaux d'accès pour les méthodes et les attributs

1. **public** : accessible de toute endroit
2. **protected** : accessible depuis les classes filles (héritage)
3. **privé package** (par défaut) : accessible depuis les autres classes du même package
4. **private** : accessible que depuis la classe où il est déclaré

Encapsulation et niveaux d'accès

- Pratiquement, les attributs sont déclarés `private`
- Manipulation des attributs privés :
 1. **accesseur** : méthode fournissant la valeur d'un attribut
 2. **modifieur** : méthode modifiant la valeur actuelle d'un attribut

Les relations entre les classe :

association et Lien

- Une association exprime une relation sémantique entre classes



- L'association est un concept de même niveau d'abstraction que les classes.
- Une association n'est pas contenue dans une classe
- C'est une connexion qui existe dans le domaine de l'application
- Un lien est une instance d'association



Les relations entre les classe :

Agrégation

Une agrégation est une relation du type composé / composant (relation du type «est partie de» ou «est composé de»)

- L'agrégation exprime un couplage fort entre classes
- La durée de vie des composants est liée à celle de l'agrégat

Agrégation

Composition de classes

Exemple : La classe Droite est composée de deux attributs qui sont instances de la classe Point

```
Class Droite {
```

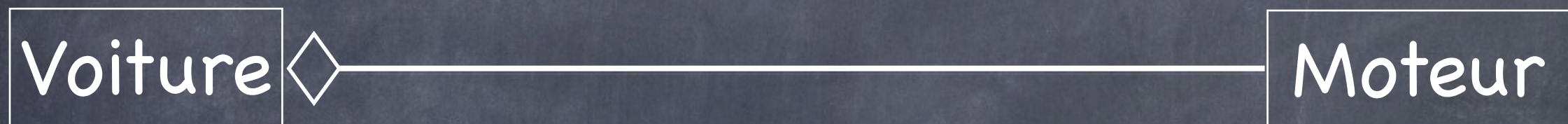
```
    Point a ;
```

```
    Point b ;
```

```
    ....
```

```
}
```


Agrégation



Une voiture est composé d'un moteur

Généralisation et spécialisation

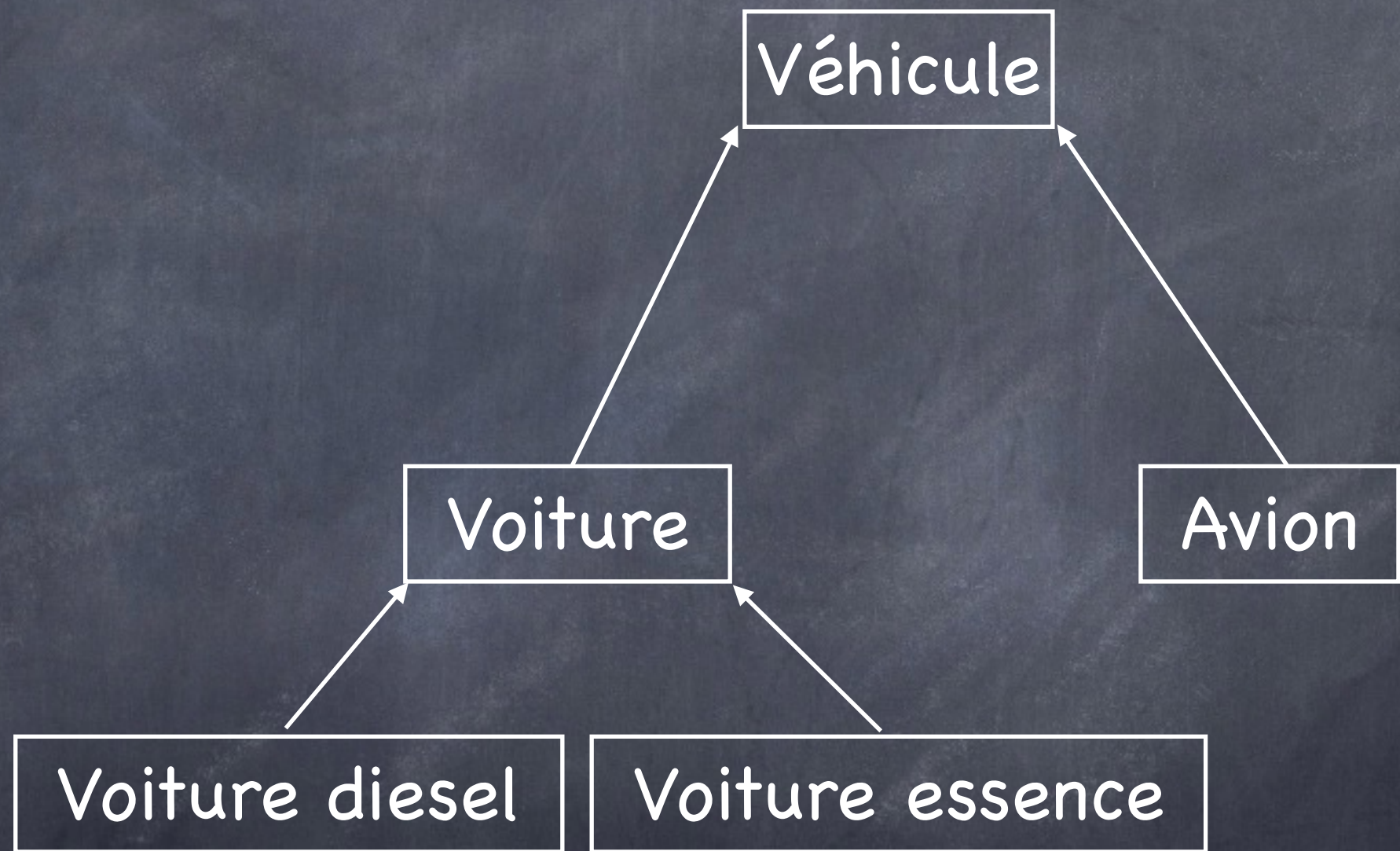
- La généralisation consiste à favoriser les éléments communs à plusieurs classes (**sous-classes**) dans une classe plus générale appelée une **super-classe**
- Elle modélise des relations du type «est un» ou «une sorte de»

Généralisation et spécialisation

Généraliser



Spécialiser

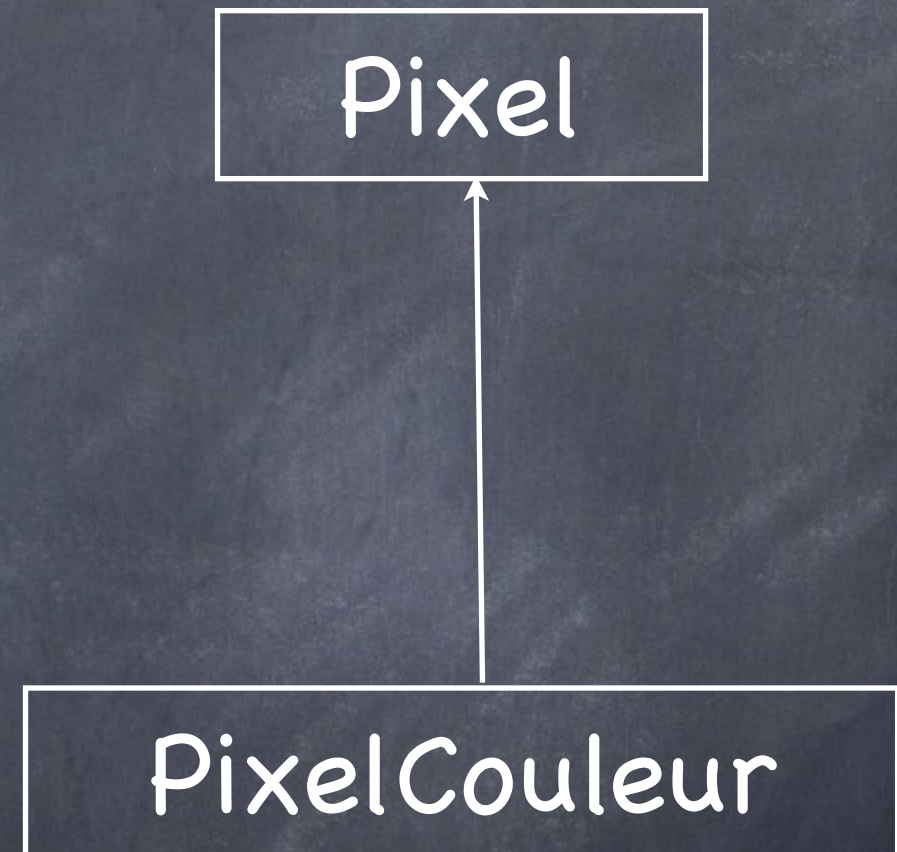


Difficulté à généraliser

- Les arbres de classes poussent à partir des feuilles et non pas de leurs racines, car les feuilles appartiennent au monde réel alors que les niveaux supérieurs sont des abstractions construites pour ordonner
- L'identification des super-classes demande un esprit d'abstraction, tandis que la réalisation des sous-classes demande une connaissance du domaine d'application

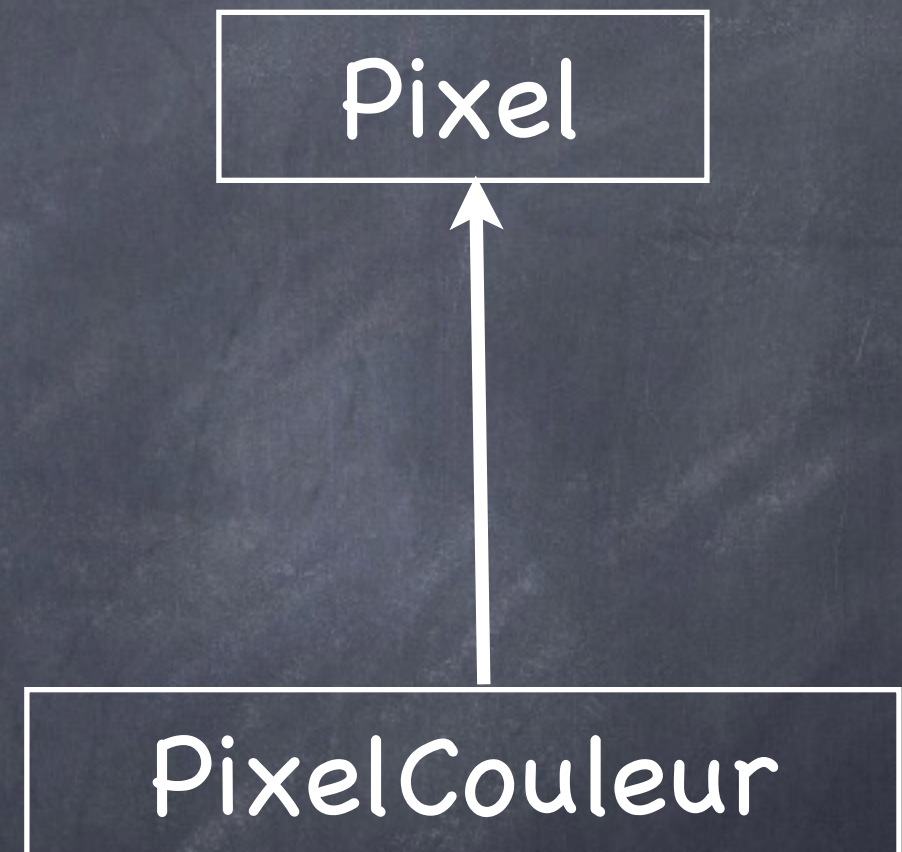
Héritage

- L'héritage est une technique des langages de programmation orientée objet pour réaliser la généralisation
- Le but de l'héritage est de créer facilement de nouvelles classes à partir de classes existantes



Héritage

- Pixel est la **classe de base**
- PixelCouleur est la **classe dérivée**
- Pixel est la **super-classe (classe mère)** de Pixelcouleur
- PixelCouleur est la **sous-classe (classe fille)** de Pixel



Héritage

Pour que l'héritage soit une implémentation de la généralisation, il doit vérifier le principe de substitution :

Il doit être possible de substituer un objet d'une sous-classe dans n'importe quel programme où un objet d'une super-classe est utilisé

L'héritage en Java

- Déclaration d'une classe **Fille** comme étant dérivée d'une classe **Mere** :

class Fille extends Mere

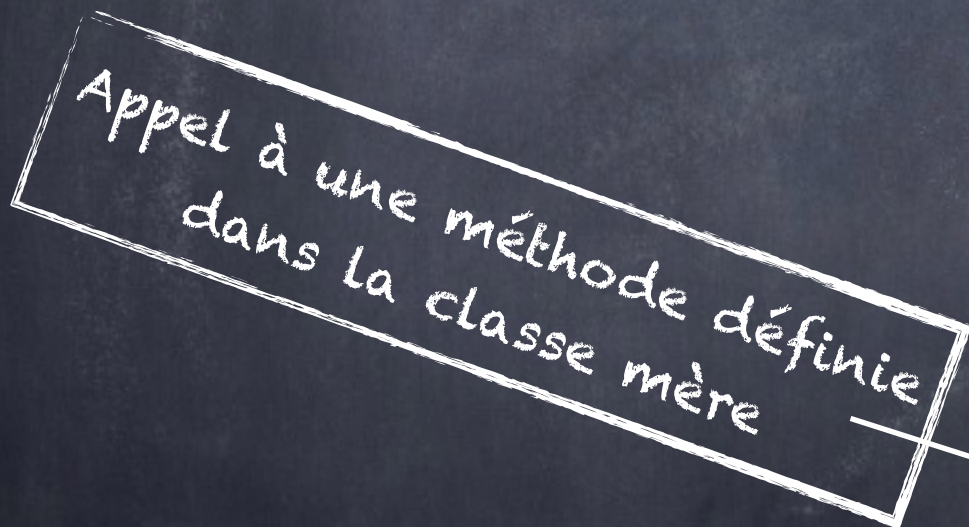
- La classe Fille possède implicitement tous les attributs et toutes les méthodes de la classe Mere
- Dans la classe Fille, d'autres attributs et méthodes supplémentaires peuvent être rajoutés

L'héritage en Java

```
public class Pixel{  
    private int x, y;  
    public Pixel (int a, int b){...}  
    public void affiche(){...}  
}
```

```
public class PixelCouleur extends Pixel{  
    private int couleur;  
    public PixelCouleur (int c){...}  
    public static void main (String[] argv){  
        PixelCouleur p = new PixelCouleur  
        (3);  
        p.affiche() ;  
    }  
}
```

Appel à une méthode définie
dans la classe mère



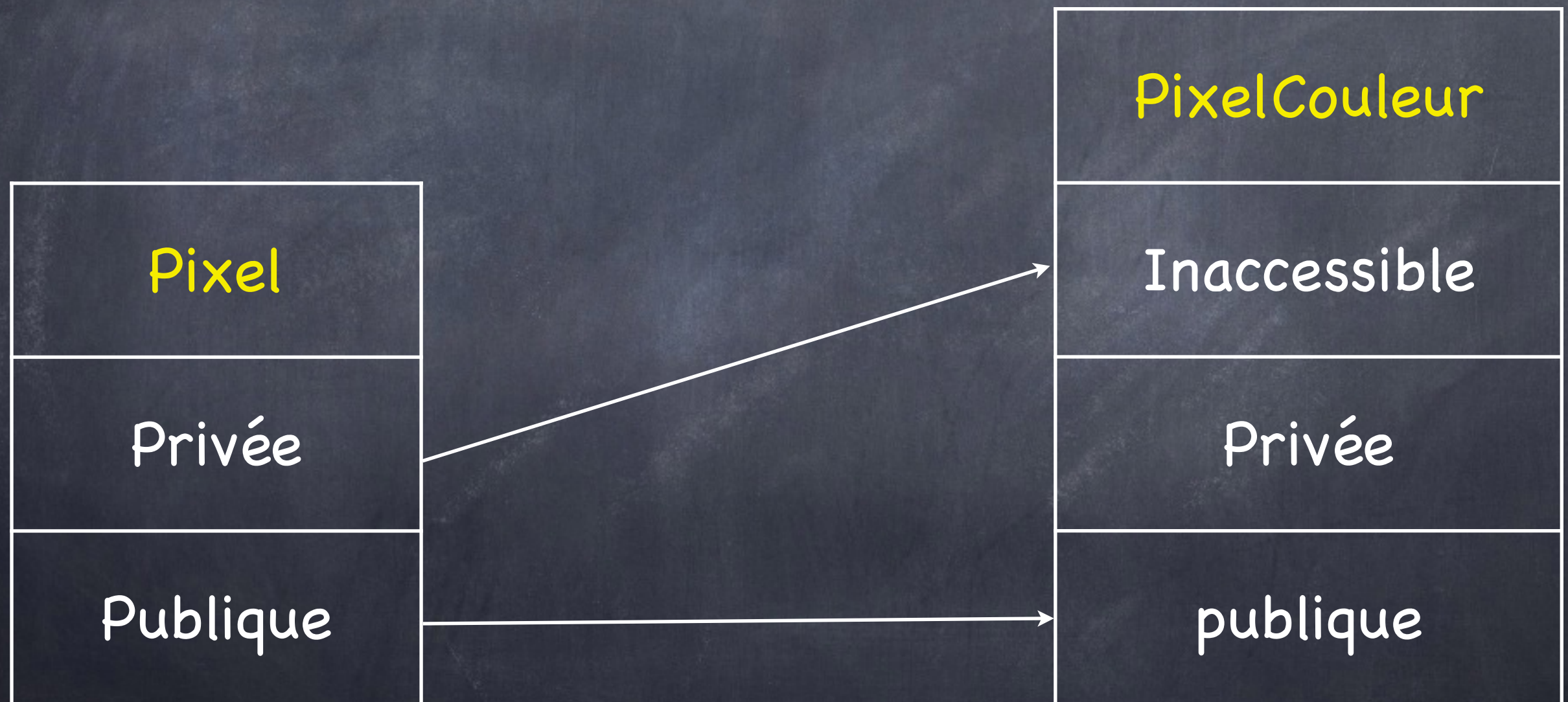
Accès aux données membres de la classe de base

```
public class Pixel{  
    private int x, y;  
    public Pixel (int a, int b){...}  
    public void affiche(){...}  
}
```

```
public class PixelCouleur extends Pixel{  
    private int couleur;  
    public PixelCouleur (int c){...}  
    public void deplacer (int dx, int dy){  
        x = x + dx; // erreur à la compilation  
        y = y + dy; // x et y inaccessibles  
    }  
}
```


Accès aux données membres de la classe de base

Le concepteur de PixelCouleur n'a pas accès aux attributs privés de Pixel



Accès aux données membres de la classe de base

```
public class Pixel{  
    protected int x, y;  
    public Pixel (int a, int b){...}  
    public void affiche(){...}  
}
```

```
    public void deplacer (int dx, int dy){  
        x = x + dx;  
        y = y + dy;  
    }
```



Pixel et PixelCouleur sont dans le même package ou dans des packages différents

Accès aux données membres de la classe de base

```
package image ;  
public class Pixel{  
    protected int x;  
    protected int y;  
    // ...  
}
```

Un membre protégé est accessible dans une classe fille du même package ou d'un package différent

```
package image.couleur ;  
class PixelCouleur extends Pixel{  
    // ...  
    public void deplacer (int dx, dy){  
        x = x + dx ;  
        y = y + dy ;  
    }  
}
```

Un membre protégé est accessible dans toutes les classes du même package

```
package image ;  
class Television{  
  
    public void allumer(){  
        Pixel p = new Pixel(0, 0);  
        p.x = 1 ;  
    }  
}
```


Accès aux données membres de la classe de base

Impossible d'accéder à un membre protégé dans une classe dérivée via une référence sur une classe de base si les deux classes sont dans des packages différents

```
package image.couleur ;  
class PixelCouleur extends Pixel{  
    public static void main (String argv[]){  
        Pixel p = new Pixel(0, 0) ;  
        p.x = 1 ; //ERREUR  
    }  
}
```

```
package image ;  
public class Pixel{  
    protected int x;  
    protected int y;  
}
```


Redéfinition des fonctions membres

Quand une méthode (ou un attribut) est redéfinie dans une classe dérivée la méthode de la classe de base est masquée : elle n'est pas directement accessible par son nom

```
public class Pixel{  
    //...  
    public void affiche(){...}  
    public static void main(String[]){  
        Pixel p = new Pixel() ;  
        p.affiche() ;  
    }  
}
```

Appel de affiche() de Pixel

```
public class PixelCouleur extends Pixel{  
    //...  
    public void affiche(){...}  
    public static void main(String[]){  
        PixelCouleur pc = new PixelCouleur() ;  
        pc.affiche() ;  
    }  
}
```

Appel de affiche() de PixelCouleur

Redéfinition

Redéfinition des fonctions membres

1. Le mot-clé `protected` sert à ne permettre l'accès d'un membre que depuis un même paquetage ou une de ses sous-classes

◦ `private` < privé-paquetage < `protected` < `public`

2. Quand une méthode est redéfinie, on ne peut changer son accessibilité qu'en l'augmentant

3. En préfixant la déclaration d'une méthode avec le mot-clé `final`, on interdit qu'une de ses sous-classes puisse redéfinir cette méthode

4. En préfixant la déclaration d'une classe avec le mot-clé `final` on interdit sa dérivation

Redéfinition des fonctions membres

1. Le mot-clé `protected` sert à ne permettre l'accès d'un membre que depuis un même paquetage ou une de ses sous-classes

◦ `private` < privé-paquetage < `protected` < `public`

2. Quand une méthode est redéfinie, on ne peut changer son accessibilité qu'en l'augmentant

3. En préfixant la déclaration d'une méthode avec le mot-clé `final`, on interdit qu'une de ses sous-classes puisse redéfinir cette méthode

4. En préfixant la déclaration d'une classe avec le mot-clé `final` on interdit sa dérivation

Redéfinition des fonctions membres

Pour désigner un attribut (ou une méthode) masqué de la super-classe depuis une sous-classe, il faut le précéder par le mot clé **super**

Quand un attribut ou une méthode sont désignés, on les cherche :

1. d'abord dans la classe de l'instance sur laquelle on les invoque

2. puis (dans le cas où ils ne seraient pas définis) dans sa superclasse

3. puis dans la superclasse de la superclasse

4. ...

Le cas des constructeurs

- La définition d'un constructeur ayant la même signature que celui de sa super-classe remplace ce dernier
- Appel au constructeur de la super-classe par `super` suivi des paramètres du constructeur
- L'appel au constructeur de la super-classe doit se faire en début de la définition du constructeur
- Si aucun appel explicite n'est fait à un constructeur de super-classe alors appel implicite à `super()`
- Utilisation standard d'un constructeur
 - appel à un constructeur de la super-classe : initialisation des attributs de la super-classe
 - initialisation des attributs déclarés dans la classe

Le polymorphisme

Le polymorphisme consiste à changer le comportement d'une classe grâce à l'héritage tout en continuant à l'utiliser comme une classe de base

```
public class Pixel{  
    //...  
    public Pixel(){ ... }  
    public void affiche(){...}  
    // ...  
}
```

```
public class PixelCouleur extends Pixel{  
    //...  
    private int couleur ;  
    public PixelCouleur(){...}  
    public void affiche(){...}  
    //...  
}
```

```
Pixel pixel = new Pixel() ;  
pixel.affiche() ; // appel de affiche() de Pixel  
PixelCouleur pixelcouleur = new PixelCouleur() ;  
pixel = pixelcouleur ;  
pixel.affiche(); // appel de affiche() de PixelCouleur
```


Le polymorphisme

- Le polymorphisme est la capacité d'une variable de changer dynamiquement son type
- En Java, on peut affecter à une variable a de type A, la valeur d'une variable b de type B dérivant de A (a référence alors le même objet que référence b)
 - Le type de a est devenu B de façon dynamique

Le polymorphisme

Il est possible d'appeler des méthodes définies dans B sur a

- Si la méthode appelée n'est définie que dans la classe mère, elle est appelée
- Si la méthode appelée a été redéfinie dans la classe fille, cette dernière est appelée
- Si la méthode appelée n'est définie que dans la classe fille, l'appel direct provoque une erreur de type à la compilation :
 - il faut transtyper a en B pour appeler la méthode de la classe fille

Le polymorphisme

```
public class Pixel{  
    //...  
    public void affiche(){...}  
    public void changePosition(int, int){...}  
}
```

```
public class PixelCouleur extends Pixel{  
    //...  
    public void affiche(){...}  
    public void changeCouleur(int){...}  
}
```

```
Pixel p = new Pixel();
```

```
PixelCouleur pc = new PixelCouleur();
```

```
p.changePosition(2,5); // La méthode de Pixel est appelée
```

```
p.affiche(); // La méthode de Pixel est appelée
```

```
pc = p ; // interdit : erreur de compilation
```

```
p = pc ; // p référence maintenant un objet de type PixelCouleur
```

```
p.changePosition(3,6); // La méthode de Pixel est appelée
```

```
p.changeCouleur(3) ; // interdit : erreur de type à la compilation
```

```
((PixelCouleur)p).changeCouleur(3) ; // La méthode de PixelCouleur  
// est appelée
```

```
p.affiche () ; // Idem
```


Le polymorphisme

- Regrouper une collection d'objet héritant tous d'une même classe mère
- Appeler la même méthode sur ces objets
- Tester le type de l'objet : `objet instanceof classe`

Les classes abstraites

- Les classes abstraites sont des classes qui ne peuvent pas être instanciées
- En général, elles déclarent que les signatures des méthodes (méthodes sans corps) : **méthodes abstraites**
- Les classes héritantes définiront concrètement les méthodes abstraites

Les classes abstraites

```
abstract class Figure{  
    ...  
    abstract void dessiner() ;  
    ...  
}
```


Les classes abstraites en Java

- La signature d'une méthode abstraite est préfixée par le mot-clé **abstract**
- La signature d'une classe abstraite est préfixée par le mot-clé **abstract**
- **Toute classe contenant une méthode abstraite est aussi abstraite**
- Il n'est pas nécessaire qu'une classe contienne une méthode abstraite pour qu'elle puisse être déclarée abstraite

Les classes abstraites

- Toute classe qui hérite d'une classe abstraite est aussi abstraite et doit être explicitement déclarée abstraite sauf si toutes les méthodes de la classes mère sont redéfinies
- On peut déclarer un objet dont la classe est abstraite
- Il est possible de lui affecter donc une instance d'une classe fille concrète (polymorphisme)

Les classes abstraites

- Les classes abstraites servent à modéliser des types d'objets qui n'ont pas une réalité concrète
- Si dans une application on a besoin de modéliser des objets dont les comportements sont voisins, il peut être utile de créer une super-classe abstraite qui caractérise et factorise ce qui est commun
 - **Méthodes concrètes** : ce qu'ils font tous de la même manière
 - **Méthodes abstraites** : ce qu'ils font chacun à leur manière

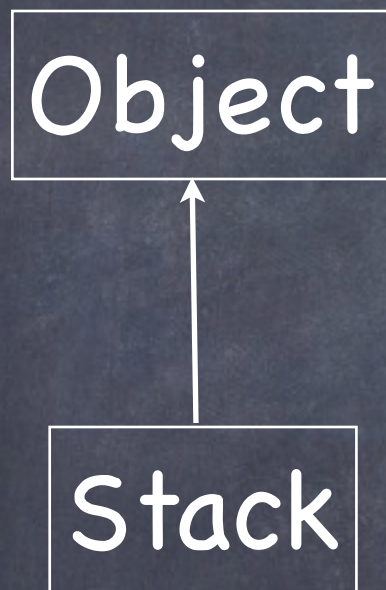
La classe Object

La classe Object est la classe mère de toutes les classes

Object
<code>wait()</code> <code>notify()</code> <code>toString()</code> <code>equals()</code> <code>hashCode()</code> <code>clone()</code> <code>getClass()</code>

Intérêt de la classe Object

Une référence de type Object peut référencer n'importe quel objet



```
package java.util;  
public class Stack ...{  
    public push(Object item){...}  
    public Object pop(){ ...}  
}
```

Exemple d'une pile de String

```
Stack s = new Stack() ;           // créé une pile  
s.push("chaine") ;                // empile "chaine"  
s.push("autre chaine") ;  
String c = (String) s.pop() ;    // extrait "autre chaine" de la pile
```


La méthode toString() de la classe Object

La méthode toString peut être utilisé pour lire l'état d'un objet

```
Vector v = new Vector() ;  
String s = "chaine" ;  
v.addElement(s) ;  
Object o = new Object() ;  
System.out.println("v = " + v + ", o = " + o) ;  
//appel de toString pour v et pour o
```

à l'écran :

v = [chaine], o = java.lang.Object @1e19ac

La méthode toString() de la classe Object

La méthode toString() peut être redéfinie

```
public class Maclasse{
    private int i ;
    private String s ;
    public Maclasse(int i, String s){
        this.i = i ; this.s = s ;
    }
    public String toString(){
        String desc = i + " " + s ;
        return desc ;
    }
    public static void main (String[] argv){
        Maclasse monobjet = new Maclasse(1, "azerty") ;
        System.out.println("Mon objet = " + monobjet) ;
        //appel de toString
    }
}
```


La méthode equals de la classe Object

Deux objets ayant le même état, ne sont pas identiques au sens de la fonction equals de Object

```
public class Maclasse{  
    private String s ;  
  
    public Maclasse (String s) {this.s = s ;}  
  
    public static void main(String[] argv){  
        Maclasse o1 = new Maclasse("chaine") ;  
        Maclasse o2 = new Maclasse("chaine") ;  
        if (o1.equals(o2)) System.out.println("identiques") ;  
        else System.out.println("différents") ;  
    }  
}
```


La méthode equals de la classe object

Redéfinir la méthode equals pour que deux objets de même état soient égaux

```
public class Maclasse{
    private String s ;
    public Maclasse (String s) {this.s = s ;}
    public boolean equals (Object o){
        if (o != null && o instanceof Maclasse)
            return s.equals(((Maclasse)o).s);
        else return false ;
    }
    public static void main(String[] argv){
        Maclasse o1 = new Maclasse("chaine") ;
        Maclasse o2 = new Maclasse("chaine") ;
        if (o1.equals(o2)) System.out.println("identiques") ;
    }
}
```


Compatibilité entre objets dérivés et objets de la classe de base

En Java, les transtypages (cast) sont vérifiés au moment de la compilation et au moment de l'exécution : l'exception `ClassCastException` est levée si le transtypage est illégal

```
PixelCouleur pc = new PixelCouleur() ;  
Pixel p = pc ; //conversion type dérivé vers type de base  
pc = p ; //erreur de compilation  
pc = (PixelCouleur) p; //conversion type de base vers  
                        //type dérivé
```


Interfaces

- Ce sont des classes abstraites particulière :
 - tous les attributs doivent explicitement être static et final
 - toutes leurs méthodes sont implicitement abstraites
- On les déclare comme des classes mais en remplaçant le mot-clé class par **interface**
- Elles sont implicitement abstraites

```
Interface I{  
    final static int constante = 10 ;  
    void f(int x) ;  
}
```


Interface et classes abstraites

Ecrire une interface équivaut presque à écrire une classe abstraite ne déclarant que des méthodes abstraites **sauf** qu'une classe peut hériter de plusieurs interfaces

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

```
abstract class I{  
    abstract void f() ;  
    abstract int g(float x);  
}
```


Interface et héritage

- Contrairement à une classe standard, une interface peut hériter de plusieurs interfaces
- On utilise le mot-clé **extends** mais en indiquant la suite des interfaces en les séparant par des virgules
- Une classe peut aussi hériter de plusieurs interfaces, en utilisant le mot-clé **implements**
- Une classe peut hériter d'une autre classe et de plusieurs interfaces

Interface et Héritage

- Une interface sert de relais entre une classe qui implémente et une classe qui l'utilise
- Une interface indique les méthodes qu'une classe met à la disposition des autres classes mais sans indiquer son implémentation ni sa structure interne
- Lorsqu'une application nécessite de définir de nombreuses classes qui vont interagir et qui seront écrites par des programmeurs différents, chaque programmeur n'a besoin de connaître que les interfaces des autres classes

Les interfaces

```
public interface IA{  
    void fA();  
}
```

```
public class A implements IA{  
    void fA(){ ...}  
}
```


Les interfaces

```
public class B{  
    private IA ia;  
    public B (IA ia){  
        this.ia = ia ; //polymorphisme  
    }  
    public void fB(){  
        ia.fA() ; //polymorphisme  
    }  
}
```

Celui qui implémente la classe B ne connaît que l'interface de A

Les interfaces

Dans le main() :

```
A a = new A() ;  
B b = new B(a) ;  
b.fB() ;
```

Ce n'est que lors de l'usage des classes, qu'il sera nécessaire de connaître de la classe A pour créer un objet de ce type