

DUT

**Génie Informatique  
Sciences de données**

**PROGRAMMATION ORIENTÉE OBJET  
(JAVA)**

*Pr. Said BENKIRANE  
2020/2021*

# Partie 2: POO(Java)

- Classes et Objets
- Packages, Contrôle d'accès et Encapsulation
- Membres statiques
- Héritage et polymorphisme
- Classes abstraites et interfaces

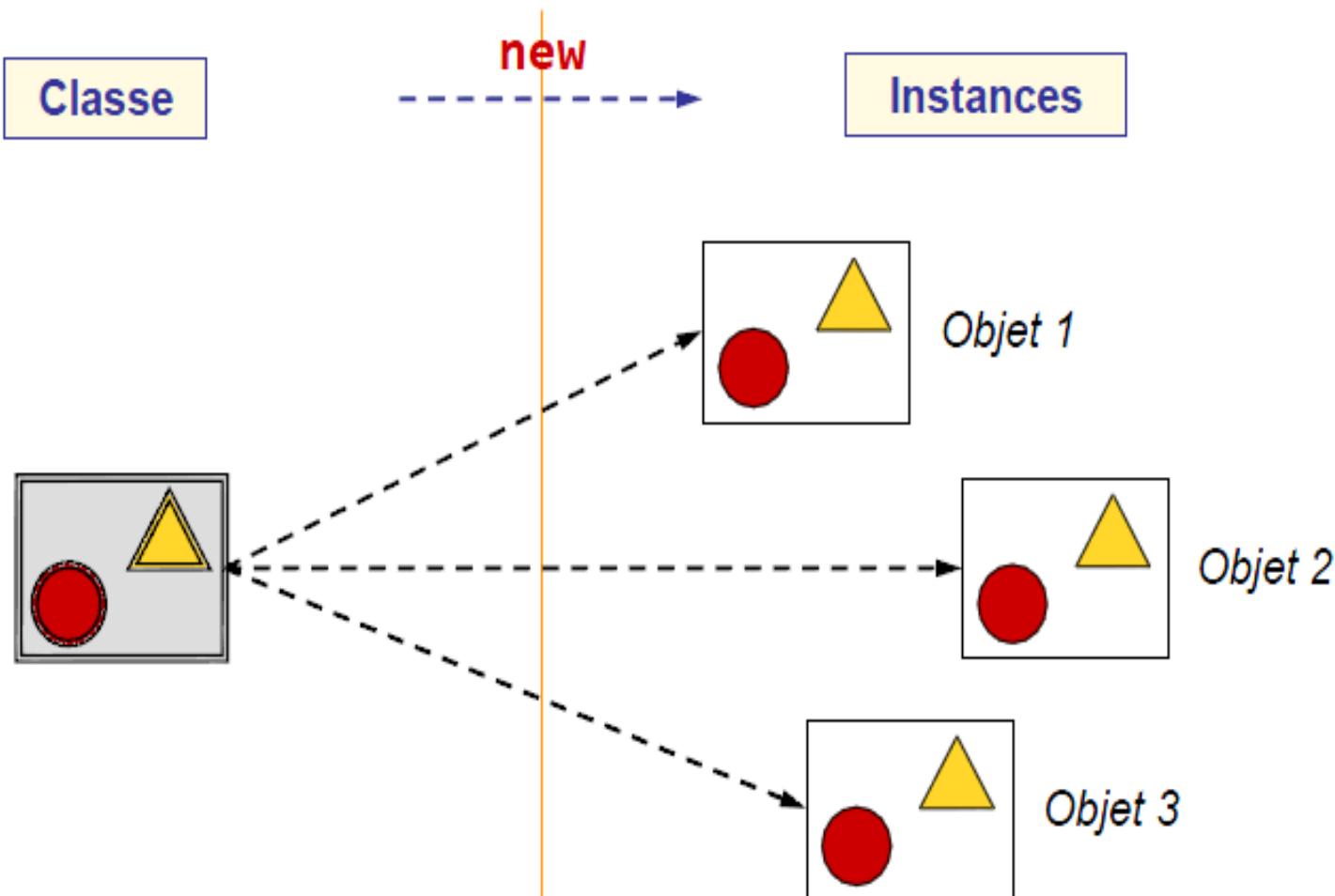


# Classes et Objets

- Une **classe** est une **collection nommée** de
  - de **champs** (ou **attributs**) contenant des valeurs
  - de **constructeurs** servant à créer les objets
  - de **méthodes** définissant des actions (opérations)
- La **classe** est l'élément structurel le plus fondamental de tout programme Java (on ne peut pas écrire de code sans créer au moins une classe).
- Les **champs** et les **méthodes** sont collectivement nommés **membres** de la classe :
  - les champs sont les **membres données** (*data members*)
  - les méthodes sont les **membres fonctions** (*function members*)
- Chaque classe définit un **nouveau type de données** qui permettra de créer (d'instancier) des **objets** de ce type.
- Les **objets** créés sont appelés **instances de la classe**.

## Classes [2]

- Une **classe** peut être considérée comme un moule (ou un plan) permettant de créer des objets qui ont des propriétés similaires mais qui ont tous leur identité propre (indépendance).



# Propriétés des objets

- Un **objet** est principalement caractérisé par :
  - une **identité** (son nom); doit être univoque dans le contexte d'utilisation
  - un **type** (la classe dont il provient, avec les classes ascendantes)
  - un **état** (qui est défini par la valeur actuelle de ses champs); l'état peut évoluer dans le temps (*chaque objet vit sa vie*)
  - un **comportement** (qui est défini par l'ensemble de ses méthodes publiques); ensemble des actions et des opérations possibles
- La **programmation orientée objet** met les objets au cœur de la conception des applications (contrairement à la programmation procédurale qui centre la conception sur les traitements appliqués aux données).
- Une **application orientée objet** est constituée d'un ensemble d'objets qui communiquent en s'échangeant des **messages** (par l'invocation de leurs méthodes).

# Déclaration de classes [1]

- La syntaxe de base pour la **déclaration d'une classe** est la suivante :

```
modificateurs class nom_de_la_classe {  
    déclaration_de_champs  
    déclaration_de_constructeurs  
    déclaration_de_méthodes  
}
```

- Les **modificateurs** (liste de mots-clés) seront vus ultérieurement (**public** pour l'instant).
- Les **noms de classes** sont habituellement écrits avec la première lettre en majuscule et en utilisant la notation *MixedCase / CamelCase* (transition minuscules/majuscules pour séparer les éléments du nom).

Exemples : **Random**, **StringReader**, **FileOutputStream**

## Déclaration de classes [2]

- Exemple de déclaration de classe :

```
public class Point {  
    public double px;      // Coordonnée x du point (champ)  
    public double py;      // Coordonnée y du point (champ)  
  
    public Point() {        // Constructeur 1  
        px = 0.0;  
        py = 0.0;  
    }  
  
    public Point(double x, double y) { // Constructeur 2  
        px = x;  
        py = y;  
    }  
  
    public double distanceOrigin() { // Méthode  
        return Math.sqrt(px*px + py*py);  
    }  
}
```

## Déclaration des champs [1]

- Les **champs** appelés aussi **attributs** (ou membres données) définissent la valeur (l'état) de l'objet.
- La **syntaxe de déclaration** des champs est proche de la déclaration d'une variable locale :

*Modificateurs Type Nom\_du\_champ [ = Expression ] ;*

- Les **modificateurs** sont constitués de mots-clés qui déterminent diverses propriétés, notamment la visibilité (**public**, **protected**, **private**). Ils seront étudiés dans un prochain chapitre.
- Le **type** peut être soit un type primitif, un tableau ou le nom d'une classe (rappelons que chaque classe définit un type référence).

## Déclaration des champs [2]

- Une **expression d'initialisation** peut être mentionnée lors de la déclaration d'un champ (= ...).
- En l'absence d'expressions d'initialisation, les champs sont **automatiquement initialisés** à leurs valeurs par défaut (**false** et **0** pour les types primitifs, **null** pour les objets et les tableaux)

```
public class Circle {  
  
    // Définition et initialisation des champs  
    public Point center = new Point(0, 0);  
    public double radius = 0.0;  
  
    // Suite de la déclaration de la classe Cercle  
    ...  
    ...  
}
```

## Constructeurs [1]

- Un **constructeur** est une sorte de méthode qui sera invoquée lors de la création d'un objet de cette classe (opérateur **new**).
- Un **constructeur** doit porter le **nom de la classe** et ne doit **pas** comporter **de type de retour** dans sa déclaration (même **pas void**).
- Le but du constructeur est d'**initialiser l'objet** (notamment la valeur de ses champs).
- Un constructeur retourne implicitement une **référence** à une instance de la classe (objet).
- Si aucun constructeur n'est défini dans une classe, Java fournit un **constructeur par défaut**, sans argument, et qui se charge uniquement de créer les champs et de leur attribuer une valeur initiale (valeur par défaut ou valeur de l'expression d'initialisation).

## Constructeurs [2]

- Comme les méthodes, les constructeurs peuvent être **surchargés** (en suivant les même règles).
- Lorsqu'une classe possède des constructeurs multiples (surchargés), il est possible, dans le corps d'un constructeur, d'invoquer un autre constructeur, en utilisant la syntaxe spéciale :  
**this(expr1, expr2, ...)**
- Dans la déclaration de la classe **Point**, le premier constructeur aurait pu être écrit ainsi :

```
public Point() {          // Constructeur 1
    this(0.0, 0.0);      // Invocation du Constructeur 2
}
```

**Restriction importante** : *L'appel à this(...) doit apparaître comme première instruction dans un constructeur.*

# Méthodes

---

- Les **méthodes** ont déjà été étudiées dans un chapitre précédent.
- Elles ne sont pas forcément déclarées avec le modificateur **static** (c'est même plutôt l'exception en programmation objet).  
Un prochain chapitre sera consacré à approfondir les notions de champs/méthodes statiques et non-statiques.
- Les méthodes ont accès aux champs de la classe dans laquelle elles sont déclarées. Ces champs sont accessibles par leurs noms simples ou en utilisant le préfixe **this** (explication de ce mot réservé dans les pages suivantes).
- Les méthodes peuvent invoquer d'autres méthodes définies dans la même classe en utilisant leur nom simple ou en utilisant le préfixe **this** (comme pour les champs).
- Les constructeurs ne sont pas des méthodes à strictement parler même si, par beaucoup d'aspects, ils leur ressemblent.

# Déclaration et création d'objets

- La classe **Point** permet de déclarer des objets de ce type :

```
Point p1; // Déclaration d'un objet de type Point
```

- La déclaration ne crée pas un objet mais uniquement une référence vers un objet du type mentionné (comme pour les tableaux).

p1

- Pour créer un objet on utilise l'opérateur **new** suivi du nom de la classe et d'une liste facultative d'arguments entre parenthèses.

```
new nom_de_la_classe(expr1, expr2, ...)
```

```
new Point(1.2, 3.4)
```

1.2	px
3.4	py

- L'opérateur **new** fait appel à l'un des **constructeurs** de la classe en fonction du profil des paramètres transmis (constructeur par défaut ou l'un des constructeurs définis explicitement dans la classe).

# Création d'objets

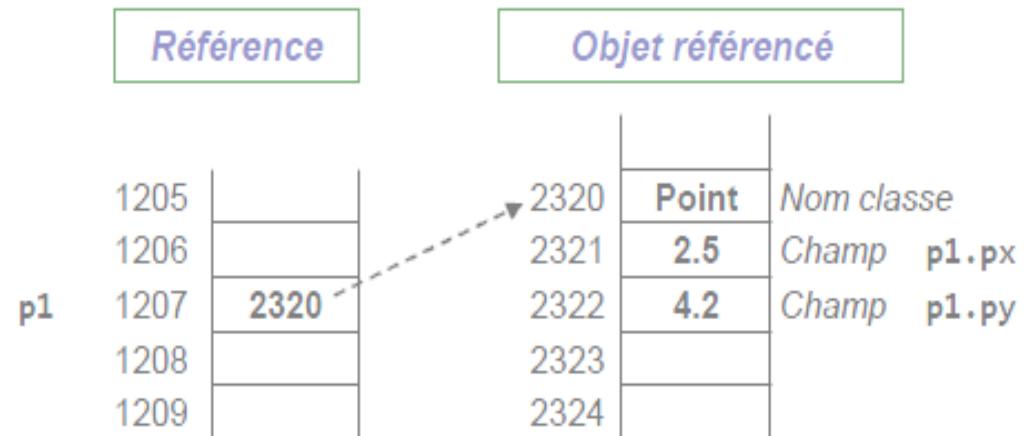
- La classe **Point** déclarée précédemment permet de déclarer et de créer des objets de ce type :

```
Point p1;                                // Déclaration d'un objet de type Point
p1 = new Point();                          // Création d'un objet et assignation
                                           // de la référence à p1
Point p2 = new Point();                    // Déclaration de p2, création d'un
                                           // objet Point et assignation de la
                                           // référence à p2
Point p3 = new Point(2.0, 4.5);           // Idem, mais utilisation du deuxième
                                           // constructeur pour créer l'objet
```

- Les **objets**, comme les tableaux, sont des **types référence**.
- **p1**, **p2** et **p3** ne contiennent donc pas les objets mais sont des références vers les zones mémoires contenant les objets.

## Représentation en mémoire

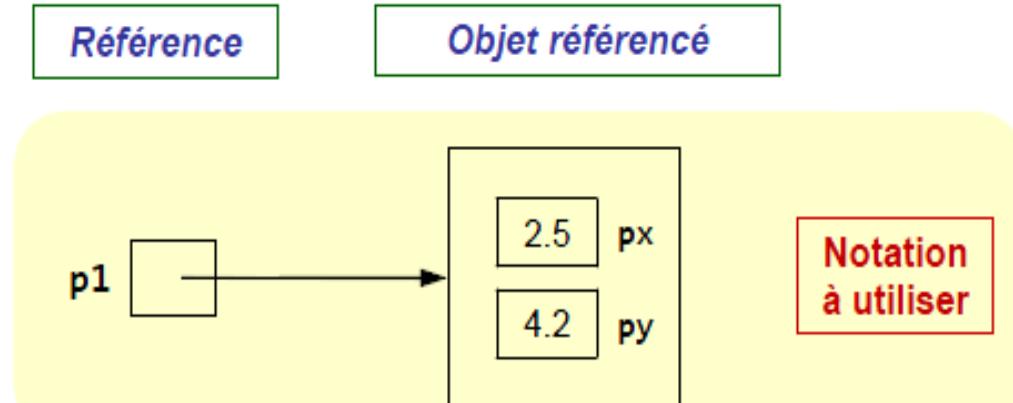
```
p1 = new Point(2.5, 4.2);
```



Création de l'espace mémoire pour contenir l'objet et initialisation des champs (constructeur)

## Notation abstraite

```
p1 = new Point(2.5, 4.2);
```



# Utilisation des objets

- L'accès aux **membres** des objets s'effectue en utilisant l'opérateur **point (.)** selon la syntaxe suivante :

*Nom\_de\_l'objet.Nom\_du\_membre*

- Le membre pouvant être soit un **champ** soit une **méthode** :

```
Point p1 = new Point(1.0, 2.0); // Déclare et crée un objet p1

double vx, vy, d;

vx      = p1.px;                      // Lit un champ de l'objet p1
vy      = p1.py;
p1.px = p1.px + 1.5;                  // Modifie la valeur du champ

d = p1.distanceOrigin();              // Exécute une méthode de p1
```

# Notion d'objet [1]

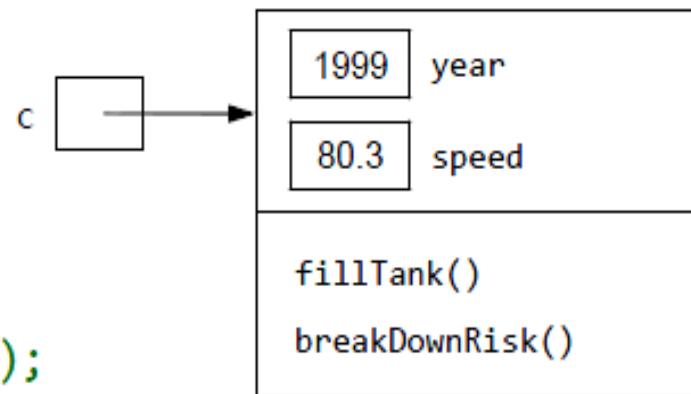
- **Objet** : Entité typée et nommée qui possède des attributs et des méthodes
- **Attribut** : Case mémoire typée et nommée
- **Méthode** : Procédure ou fonction nommée qui a accès aux attributs de l'objet
- Pour accéder aux composants (membres) d'un objet il faut :
  - détenir une variable qui référence cet objet
  - connaître le nom de l'attribut ou de la méthode
  - utiliser l'opérateur ":"
  - utilisation comme une variable, resp. une procédure / fonction

## Notion d'objet [2]

- Exemple :

- Type d'objet `Car`
- Attributs `int year`  
`float speed`
- Méthodes `void fillTank (int liters)`  
`double breakDownRisk(int km)`

```
Car c = new Car();
c.year = 1999;
System.out.println(c.speed);
c.fillTank(30);
System.out.println(c.breakDownRisk(50));
```



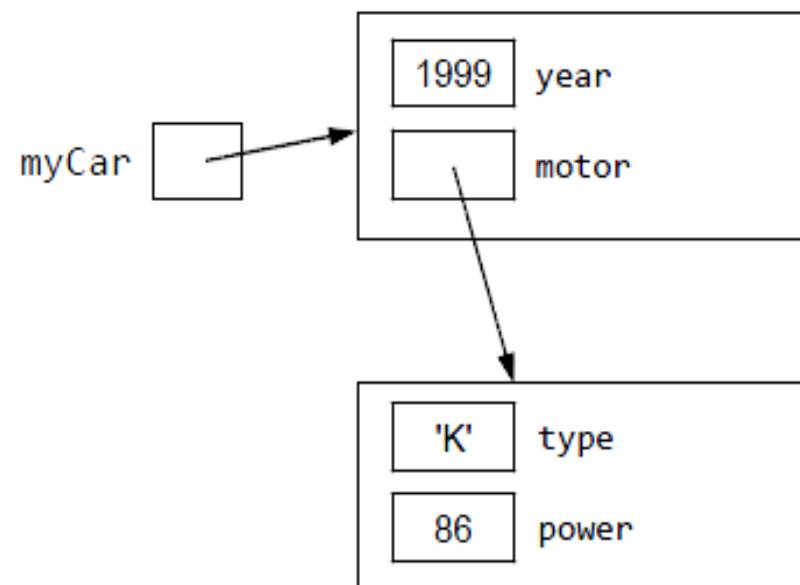
Généralement, les méthodes ne sont pas mentionnées dans la représentation de l'état de la mémoire.

# Notion d'objet [3]

- Les types référence sont de vrais types Java qui peuvent être utilisés :
  - Pour la déclaration de variables locales
  - Pour la déclaration d'attributs d'un objet
  - Comme paramètres de méthodes
  - Comme résultat d'une fonction

## Type d'objet Car

Attributs      int      year  
                    Engine      motor



## Type d'objet Engine

Attributs      char      type  
                    int      power

# Notion d'alias

	Type primitif	Objet / Tableau
Affectation <code>a = b</code>	Copie de la valeur	Copie de la référence L'objet n'est pas modifié
Comparaison <code>a == b</code>	Test sur la valeur	Test sur la référence, pas sur le contenu de l'objet !
Passage en paramètre <code>p(a)</code>	Passage de la valeur (copie locale)	Passage de la référence (copie) L'objet risque d'être modifié !

- **Type primitif** : accès à la zone mémoire par une seule variable
- **Objet / Tableau** : accès à la zone mémoire possible par plusieurs variables

```
Car myCar      = new Car();
Car otherCar   = new Car();

myCar.year     = 1999;
otherCar       = myCar;
otherCar.year  = 2014;

System.out.println(myCar.year);      // 2014 !!!
```

## Référence à l'objet courant : this

- Le mot clé **this** est utilisé pour référencer l'objet à l'intérieur duquel le code est exécuté.
- On peut considérer que l'objet **this** est implicitement passé en référence lors de l'invocation de chaque méthode d'instance :

```
Point p1 = new Point(1.0, 2.0);
double d = p1.distanceOrigin();
```

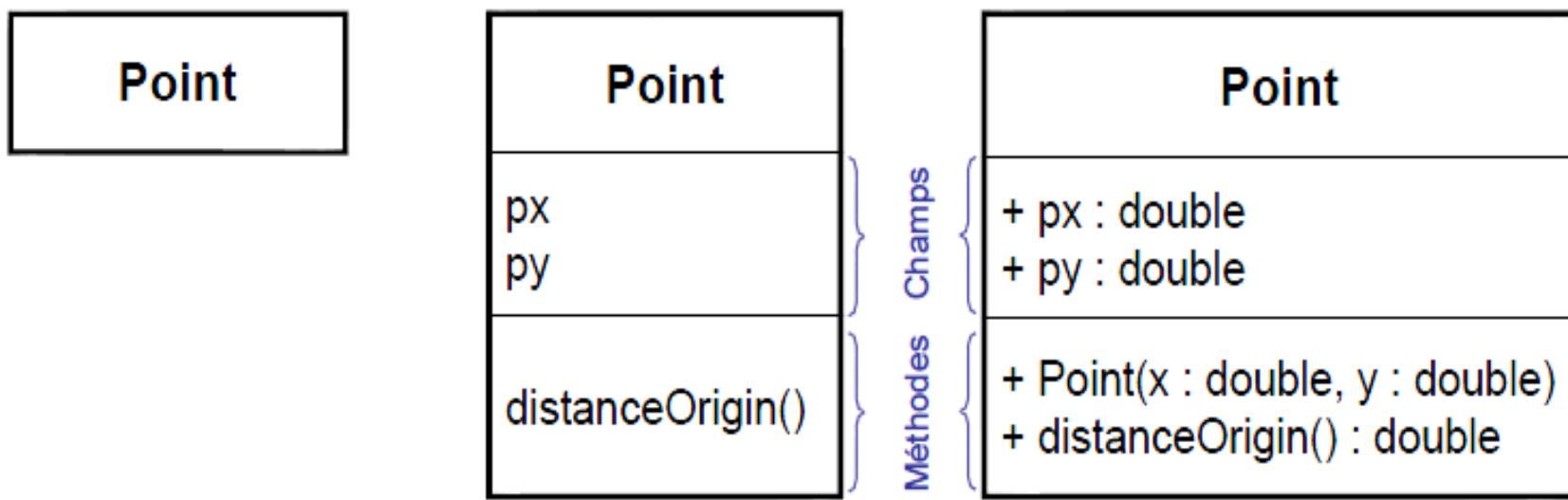
Dans l'exemple ci-dessus, l'objet **p1** est implicitement passé en paramètre (invisible) à la méthode **distanceOrigin()**.

A l'intérieur du corps de la méthode **distanceOrigin()** on peut, si nécessaire, utiliser le mot-clé **this** pour référencer l'objet sur lequel la méthode a été invoquée (**p1** dans cet exemple).

**Attention** : Il ne faut pas confondre **this** utilisé pour référencer l'objet courant avec la forme **this()** qui est utilisée pour invoquer un constructeur depuis un autre constructeur.

# Classe : Notation UML

- Dans le langage de modélisation **UML** (*Unified Modeling Language*), les classes sont représentées graphiquement sous la forme de rectangles comprenant le nom de la classe et éventuellement le nom des champs, des constructeurs et des méthodes.
- Différents niveaux de détails sont possibles :



Indicateurs de visibilité :

+ : public   - : private   # : protected

# En bref, écrire une classe c'est...

- Choisir un **nom** pour la classe
  - Intégrer cette classe dans un paquetage
- Écrire la **spécification** (déclarer la partie publique)
  - Signature des méthodes avec les exceptions prévues
  - Explication de l'effet attendu (rôle des paramètres, valeur de retour, effet sur les données, pré- et post-conditions)
- Écrire une **implémentation** :
  - Définition des attributs (choix des structures de données)
  - Conception du corps des méthodes (choix des algorithmes)
  - Codage proprement dit (identificateur, flux d'instructions, lisibilité, méthodes privées)
- **Tester** le bon fonctionnement
- Éventuellement
  - Documenter la classe avec `/** ... */` (Javadoc)
  - Affiner, optimiser, ...



# **Packages, Contrôle d'accès et Encapsulation**

# Unités de compilation

---

- Un **programme Java** est généralement constitué d'un **ensemble de classes**.
- Le code source de **chaque classe publique** doit être enregistré dans un **fichier séparé** portant le nom de la classe avec l'extension **".java"**.

Un tel fichier constitue une **unité de compilation**.

- Le résultat de la compilation (*Bytecode*) est enregistré dans un fichier portant le nom de la classe avec l'extension **".class"**.
- Exemple :
  - La classe **Point** déclarée précédemment doit être enregistrée dans un fichier nommé "**Point.java**"
  - Après la compilation, un nouveau fichier "**Point.class**" sera créé, il contiendra le *Bytecode* correspondant à la classe **Point**

# Paquetages

- Un **paquetage (Package)** est une **collection nommée de classes** et/ou d'interfaces. Un paquetage peut comprendre des sous-paquetages.
- Les paquetages permettent de grouper des classes apparentées et de définir un **espace de désignation (Namespace)** pour les classes qu'il contient.
- Les paquetages servent également à gérer les droits d'accès (visibilité) des classes les unes par rapport aux autres.
- Les paquetages sont **organisés de manière hiérarchique** (structure arborescente avec racines multiples).
- Dans le nom des paquetages, le point ('.') est utilisé comme séparateur entre les différents niveaux hiérarchiques.
- Exemple :

`java.lang.String` : La classe `String` se trouve dans le sous-paquetage `lang` du paquetage `java`

## Déclaration de paquetage [1]

- Le mot clé **package** est utilisé pour indiquer le paquetage auquel appartiennent la ou les classes de l'unité de compilation.

```
package Nom_du_paquetage ;
```

- Cette pseudo-instruction, si elle est utilisée, doit être **la première** (hormis les commentaires) de l'unité de compilation (fichier source).
- Exemple :

```
package ch.eiafr.tic;  
  
public class Point {  
    ...  
}
```

- La classe **Point** fait partie du paquetage `ch.eiafr.tic`
- Le **nom complet** de la classe est "`ch.eiafr.tic.Point`"

## Déclaration de paquetage [2]

- Si aucune instruction **package** n'apparaît dans un fichier source, (ce qui est déconseillé) les classes définies dans cette unité de compilation seront attribuées à un **paquetage par défaut** (qui est anonyme).
- Les **noms des paquetages** sont habituellement écrits en utilisant **uniquement des minuscules** (convention de codage).
- Pour faciliter l'échange de code Java (librairies publiques), il est utile d'avoir un espace de désignation globalement univoque (pour éviter les conflits dans les noms de classes).
- C'est pour cela qu'il est recommandé de créer l'arborescence des paquetages en utilisant comme racine, les noms de domaines internet (en inversant les éléments de l'URL, tout en respectant la syntaxe des identificateurs Java !).

Exemple : *Nom de domaine* : eia-fr.ch  
*Racine des paquetages* : ch.eiafr

## Importation de paquetage [1]

- Par défaut, une classe d'un paquetage **p** peut faire référence à toute autre classe de **p** par son nom simple (le nom de la classe uniquement, par exemple **Point**).
- Pour référencer une classe d'un paquetage différent de celui dans lequel on se trouve, il faut utiliser son nom complet (paquetage + classe, par exemple **ch.eiafr.tic.Line**).
- Afin d'alléger l'écriture, on peut "*importer*" les classes d'un paquetage externe en utilisant l'instruction **import**.
- Il existe deux formes :
  - *Importation d'une classe individuelle* : **import ch.eiafr.tic.Line;**
  - *Importation de toutes les classes contenues dans un paquetage* : **import ch.eiafr.tic.\*;**
- L'instruction **import** rend accessible, par leurs noms simples, les classes externes mentionnées.

## Importation de paquetage [2]

- L'instruction **import** doit être placée au début de l'unité de compilation, juste après l'instruction **package** (s'il y en a une).
- L'instruction **import** peut être répétée à volonté pour importer plusieurs classes et/ou plusieurs paquetages.
- L'importation d'un paquetage entier (**....\***) **ne rend pas visible le contenu des éventuels sous-paquetages**. Si nécessaire, les sous-paquetages doivent être importés explicitement.
- En cas de conflit (importation de paquetages contenant des classes portant le même nom), le compilateur imposera l'utilisation du nom complet pour accéder aux classes homonymes.
- Le paquetage **java.lang** étant considéré comme fondamental, il est **implicitement importé** et l'on peut donc utiliser ses classes par leurs noms simples (par exemple : **String, Math, System, ...**) sans avoir à les importer.

## Importation statique

---

- Depuis la version 1.5 du langage, il est possible d'**importer** sélectivement ou globalement les **membres statiques** d'une classe en ajoutant le mot-clé **static** à l'instruction **import** (les membres statiques seront vus en détail au chapitre suivant).
- Cette importation statique permet, dans certains cas, d'alléger l'écriture en évitant de devoir préfixer les champs et les méthodes statiques avec le nom du package et/ou de la classe.
- Par exemple :

```
import static java.lang.Math.max;
```

importe la méthode statique `max()` qui pourra donc être invoquée sans préfixe : `r = max(v1, v2);`

- Pour importer tous les membres statiques de la classe **Math** :

```
import static java.lang.Math.*;
```

```
a = PI * pow(r, 2);
```

# Paquetages et unités de compilation [1]

- La **hiérarchie des répertoires** du disque dans lesquels sont enregistrés les fichiers des classes (`.class`) **doit correspondre à la hiérarchie des paquetages**.

Paquetages	Répertoires / Fichiers
<pre>package ch.eiafr.tic; public class Point {...}</pre>	<pre>C:   +-ClassesJava ← Racine des paquetages   +--ch       +--eiafr           +--tic               +--test               +--tools</pre>
<pre>package ch.eiafr.tic.tools; public class Complex {...} public class Graph {...}</pre>	<pre>Point.class TestPoint.class Complex.class Graph.class</pre>
<pre>package ch.eiafr.tic.test; public class TestPoint {...}</pre>	

## Paquetages et unités de compilation [2]

- La hiérarchie des fichiers contenant les classes constitue un **sous-arbre** qui peut être placé n'importe où dans l'arborescence générale du disque.
- Le paramètre (ou la variable d'environnement) **classpath** indique à la machine virtuelle Java, la liste des répertoires (racines) dans lesquels se trouvent les classes.

```
java -classpath .;C:\ClassesJava;G:\Info\Lib ...
```

- A partir de ces **racines** (points d'entrée), la machine virtuelle complète le chemin d'accès en y ajoutant les sous-répertoires définis par l'arborescence des paquetages.

**Remarque :** L'ensemble des fichiers `.class` peut également être enregistré dans un seul fichier `.jar` qui contient toutes les classes et leur arborescence sous forme éventuellement compressée (format ZIP).

# Structure des unités de compilation

- En résumé, les unités de compilation (fichiers `.java`) doivent respecter la structure suivante :

- (zéro ou) une instruction **package**
- zéro, une ou plusieurs instructions **import**
- une ou plusieurs définitions de **classes (class)**, d'interfaces ou de classes-internes (mais une seule classe ou interface publique au premier niveau)

```
package ...;  
import ...;  
import ...;  
import ...;  
public class ... {  
    ...  
}
```

- Ces éléments (**package**, **import**, **class**) peuvent naturellement être entourés de commentaires, mais ils doivent impérativement apparaître dans l'ordre mentionné ci-dessus.

# Librairie de classes Java

---

- La plate-forme Java comprend une vaste collection de classes prédéfinies qui peuvent être utilisées indépendamment de l'environnement d'exécution des applications.
- Cette **librairie de classes** est organisée en paquetages (et sous-paquetages) qui rassemblent, par domaine, des classes apparentées.
- L'arborescence des paquetages de la plate-forme Java possède deux racines principales **java** et **javax**.
- Les pages qui suivent donnent un aperçu succinct (quelques paquetages importants) de la plate-forme Java.

**Conseil** : *Inutile de réinventer la roue. Avant de coder une fonction particulière, il faut s'assurer qu'elle n'est pas disponible dans la bibliothèque des classes standard.*

# Librairie de classes Java [1]

<b>java.lang</b>	Le noyau des classes du langage, comprenant notamment String, Math, System, Thread et Exception. <i>Ce package est implicitement importé.</i>
<b>java.io</b>	Classes et interfaces d'entrée/sorties. Bien que certaines des classes de ce paquetage soient conçues pour travailler directement avec les fichiers, la plupart d'entre-elles permet de travailler avec des flux d'octets ou des flux de caractères.
<b>java.math</b>	Petit paquetage qui contient des classes destinées à l'arithmétique entière de précision arbitraire et à l'arithmétique décimale.
<b>java.net</b>	Classes et interfaces destinées à l'interconnexion avec d'autres systèmes (réseau).
<b>java.security</b>	Classes et interfaces de contrôle d'accès et d'authentification. Plusieurs sous-paquetages.
<b>java.util</b>	Diverses classes utilitaires y compris un cadre de collections puissant, à utiliser avec les collections d'objets

# Librairie de classes Java [2]

<b>java.applet</b>	Définit la classe Applet qui est la super-classe de toutes les applets.
<b>javax.accessibility</b>	Ensemble de classes et d'interfaces permettant la réalisation d'applications adaptées à des personnes handicapées (par exemple mal-voyants)
<b>java.awt</b>	Définit le cadre de l'interface utilisateur graphique de base AWT ( <i>Abstract Windowing Toolkit</i> ). Contient également les classes permettant de gérer des graphiques 2D.
<b>java.awt.event</b>	Définit les classes et les interfaces utilisées pour la gestion des événements en AWT et Swing.
<b>java.awt.image</b>	Ensemble de classes et d'interfaces servant à manipuler les images.
<b>java.awt.print</b>	Ensemble de classes et d'interfaces destinées à l'impression de documents.

# Librairie de classes Java [3]

<b>javax.swing</b>	Grand paquetage (comportant de nombreux sous-paquetages) destiné à étendre AWT pour la création d'interfaces utilisateur graphiques (GUI) sophistiquées. Introduit de nouveaux composants graphiques avec des propriétés beaucoup plus étendues qu'AWT.
<b>javax.swing.event</b>	Complète et améliore java.awt.event avec des classes spécialisées pour les composants Swing.

- Une **description** de l'ensemble des classes (et interfaces) contenues dans ces paquetages accompagne le JDK (*Java Development Kit*). La documentation est à télécharger séparément.
- Elle peut être consultée en ligne sur le site d'Oracle :  
<http://docs.oracle.com/javase/8/docs/api/>
- Les environnements de développement (*Eclipse*, *NetBeans*, ...) permettent généralement également de consulter la description des API de la plate-forme Java (*voir documentation correspondante*).

# Contrôle d'accès aux membres

---

- Tous les champs et toutes les méthodes d'une classe peuvent être utilisés dans le corps de la classe elle-même en utilisant leurs noms simples.
- Cependant, le langage Java permet de spécifier des restrictions d'accès aux membres d'une classe (champs et méthodes) en dehors de cette classe (appelée **classe de définition** ou **classe de déclaration**).
- Le **contrôle d'accès** aux membres s'effectue à l'aide de différents mots-clés qui peuvent précéder la déclaration des champs et des méthodes (ces mots-clés font partie d'un groupe de mots réservés appelés **modificateurs**).
- Le **mode d'accès par défaut** (si l'on n'indique aucun mot-clé de contrôle d'accès dans la déclaration) est appelé "*package*" ou parfois "*friendly*" (mais ces termes ne peuvent pas être utilisés comme modificateurs dans le code).

# Modificateurs d'accès [1]

- Les modificateurs qui contrôlent l'accès aux membres (champs ou méthodes) sont les suivants :

Modificateur	Description des droits d'accès sur le membre
<b>public</b>	Accessible à toutes les classes de tous les paquetages
<b>protected</b>	Accessible aux classes dérivées (sous-classes) ainsi qu'aux classes du même paquetage <i>Sera décrit plus en détail dans le cadre de l'héritage</i>
<i>Aucun (⇒ package)</i>	Accessible à toutes les classes du même paquetage
<b>private</b>	Accessible seulement aux autres membres de la même classe (classe de définition)

- Le modificateur **public** peut également être utilisé dans la déclaration des classes. Il indique que la classe est accessible partout où le paquetage est accessible (par **import** ...).  
Sans lui, la classe n'est accessible qu'au sein de son paquetage.

# Modificateurs d'accès [2]

- Les **règles d'accès aux membres** peuvent également être présentées sous la forme suivante :

Accessible à :	public	protected	package	private
Classe de définition	Oui	Oui	Oui	Oui
Classe du même paquetage	Oui	Oui	Oui	Non
Sous-classe dans un paquetage différent	Oui	Oui	Non	Non
Classe (qui n'est pas une sous-classe) dans un paquetage différent	Oui	Non	Non	Non

↑  
Par défaut

# Modificateur d'accès private

- Si, dans une classe **A**, on déclare un objet **k** de type **A** (ou si une méthode de la classe **A** reçoit en paramètre un objet **k** de type **A**) alors les membres privés de l'objet **k** sont accessibles à l'intérieur de la classe **A**.
- Autrement dit, dans une classe **A**, on peut accéder aux membres privés de tous les objets de type **A** (pas seulement ceux de **this**).

```
public class A {  
    ...  
    private int x;  
    private A c;  
    ...  
    public void m1(A k) {  
        k.x = this.x + 2;           // k.x est accessible  
        System.out.println(k.x);  
    }  
    ...  
    public void m2() {  
        x += c.x;                // c.x est accessible  
    }  
    ...  
}
```

# Encapsulation [1]

- L'une des techniques essentielles de la programmation orientée objet est l'**encapsulation**.
- L'encapsulation consiste à **masquer les données** au sein des classes et à **manipuler leur contenu qu'au moyen de méthodes publiques**.
- La mise en place du mécanisme d'encapsulation s'effectue en utilisant judicieusement les **contrôles d'accès** aux membres des classes.
- Avantages de l'encapsulation :
  - **Masquage** des détails de l'implémentation (qui peut être changée sans conséquences pour les utilisateurs externes)
  - **Protection** de la classe contre des utilisations erronées (contrôles de cohérence dans les méthodes)
  - **Simplification** de l'API (les champs et les méthodes internes sont cachés)

## Encapsulation [2]

- En règle générale, il est recommandé de **masquer les champs** en les déclarant **private** et - **seulement si nécessaire** - de définir des **méthodes publiques** pour accéder à ces champs (ces méthodes appelées *accesseurs/mutateurs* ou *getters/setters* sont généralement nommées **get<sub>xxx</sub>( )**, **is<sub>xxx</sub>( )** et **set<sub>xxx</sub>( )**).
- Version encapsulée de la classe **Point** :

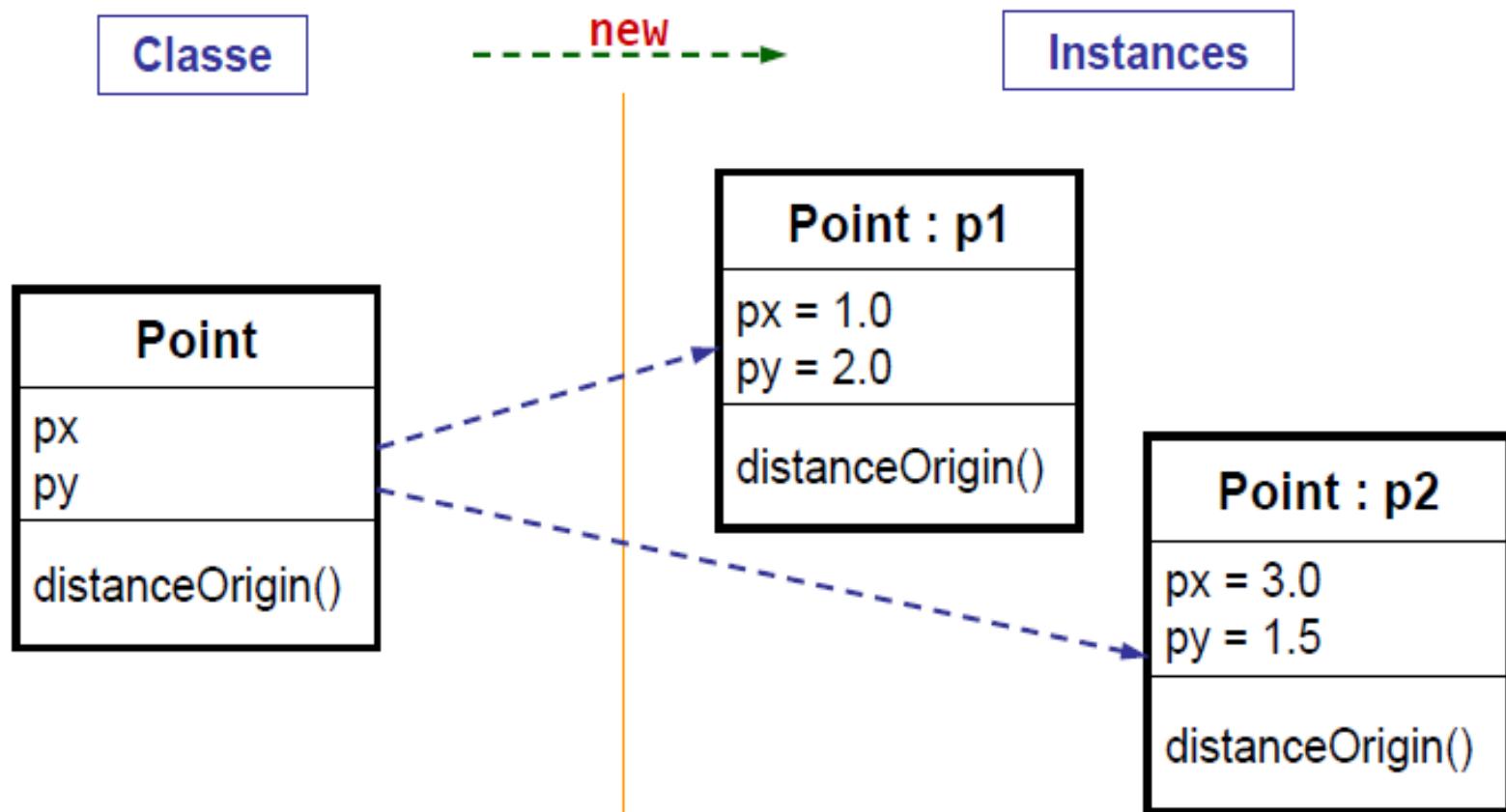
```
public class Point {  
    private double px;          // Coordonnée x du point (champ)  
    private double py;          // Coordonnée y du point (champ)  
    public Point() {...}        // Constructeur 1  
    public Point(double x, double y) {...} // Constructeur 2  
    public double getPx() {return px;} // Get px  
    public double getPy() {return py;} // Get py  
    public void setPx(double x) {px = x;} // Set px (nécessaire ?)  
    public void setPy(double y) {py = y;} // Set py (nécessaire ?)  
    public double distanceOrigin() {...}  
}
```



# Membres statiques

# Membres d'instance

- Par défaut, lors de la création d'un objet (instanciation d'une classe avec l'opérateur **new**), les **membres** (champs et méthodes) sont **associés à chacune des instances** de la classe.  
On les appelle **membres d'instance**.



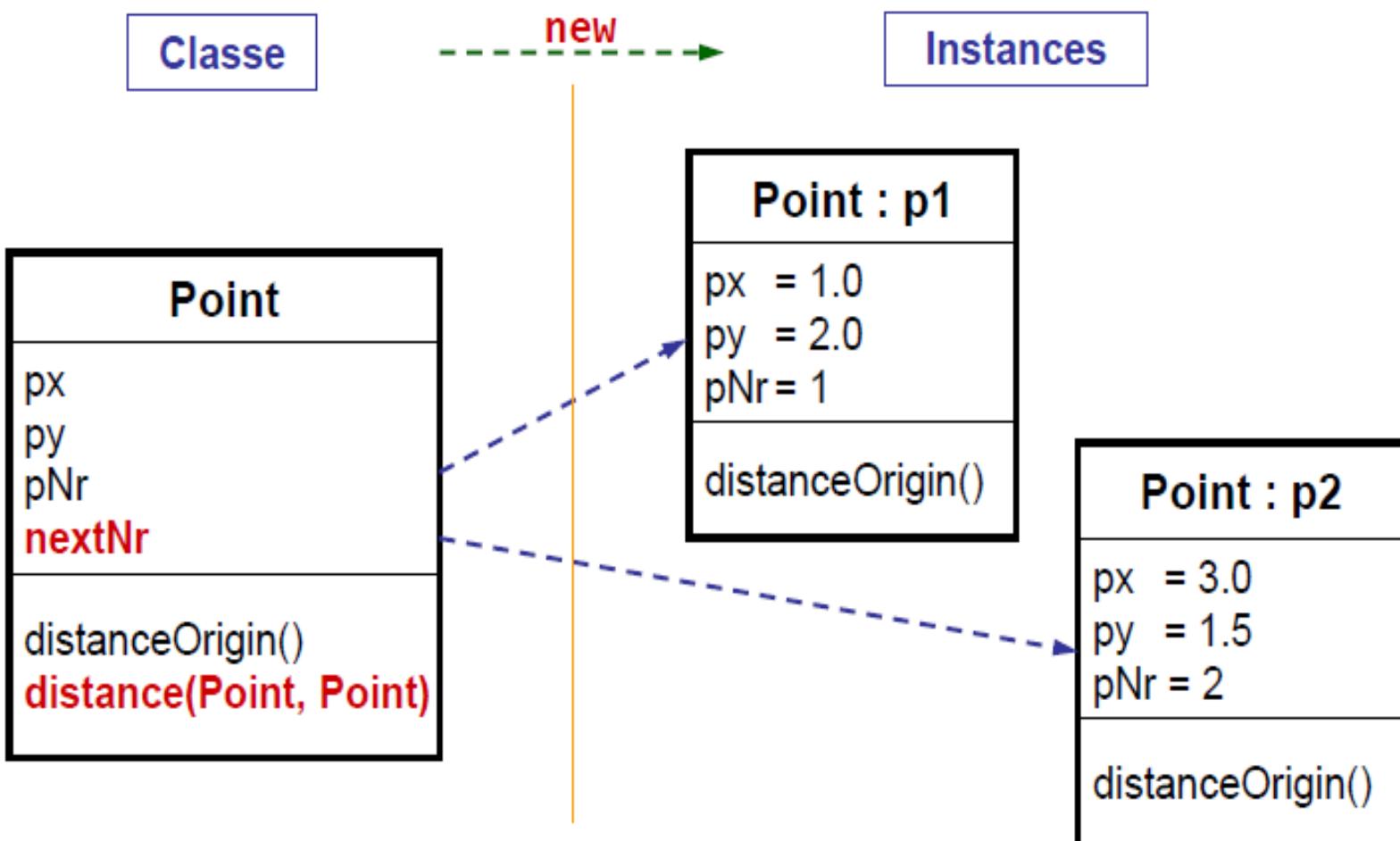
## Membres de classe [1]

- On peut également créer des **membres associés à la classe** qui sont appelés **membres de classe** ou **membres statiques**.
- Pour rendre un membre statique, on utilise le mot-clé (modificateur) **static** devant la déclaration du champ ou de la méthode :

```
public class Point {  
    public      double px;                      // Champ d'instance  
    public      double py;                      // Champ d'instance  
    public      int   pNr;                      // Champ d'instance  
    public static int   nextNr = 1;              // Champ statique  
    public Point(double x, double y) {          // Constructeur  
        ...  
        pNr = nextNr++;  
    }  
    public double distanceOrigin() {...}        // Méthode d'instance  
                                              // Méthode statique  
    public static double distance(Point p1, Point p2) {...}  
}
```

## Membres de classe [2]

- Les **membres de classe** ne sont pas associés aux instances (objets) mais seulement à la classe. Ils sont accessibles par toutes les instances de la classe (ils sont partagés par toutes les instances).



## Membres de classe [3]

- On peut donc déclarer :
  - des **champs statiques** (ou **champs de classe**)
  - des **méthodes statiques** (ou **méthodes de classe**)
- Les **champs statiques** sont **enregistrés au niveau de la classe** et ils peuvent être accédés et manipulés par toutes les instances (objets) de cette classe.
- Indépendamment du nombre d'objets créés, il n'existe **qu'un seul exemplaire** des données associées aux champs statiques (contrairement aux champs d'instance dont il existe un exemplaire pour chaque objet créé).
- On peut considérer les **champs statiques** comme étant des **variables globales partagées par toutes les instances**. Il faut donc les utiliser avec précaution et parcimonie afin d'éviter des couplages inutiles (ou même dangereux) entre objets.

## Membres de classe [4]

- Les **méthodes statiques** sont liées à une classe et non pas à une instance (objet) de la classe.
- Dans une **méthode statique**, on ne peut pas faire référence à une méthode d'instance sans créer d'objet, car les méthodes statiques ne s'exécutent pas dans le contexte d'un objet (autrement dit, pour les méthodes statiques, il n'existe pas de référence **this**).
- Pour accéder à un membre statique d'une classe en dehors de cette classe, il faut **préfixer le nom du membre** (champ ou méthode) **avec le nom de la classe** (ou utiliser `import static...`).

```
Point p1 = new Point(1.0, 2.0);
Point p2 = new Point(2.0, 3.0);
int n = Point.nextNr;           // Champ statique
double d1 = Point.distance(p1, p2); // Méthode statique
double d2 = p1.distanceOrigine(); // Méthode d'instance
```

# Importation statique

- Il est possible d'**importer** sélectivement ou globalement les **membres statiques** d'une classe en ajoutant le mot-clé **static** à l'instruction **import**.
- Cette importation statique permet, dans certains cas, d'alléger l'écriture en évitant de devoir préfixer les champs et les méthodes statiques avec le nom du package et/ou de la classe.
- Par exemple :

```
import static java.lang.Math.max;
```

importe la méthode statique **max()** qui pourra donc être invoquée sans préfixe : **r = max(v1, v2);**

- Pour importer tous les membres statiques de la classe **Math** :

```
import static java.lang.Math.*;
```

```
a = PI * pow(r, 2);
```

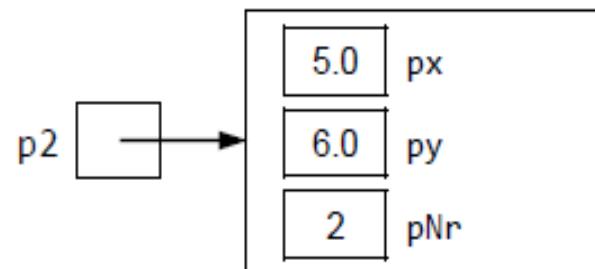
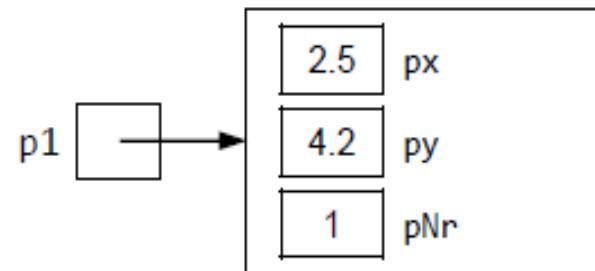
# Représentation en mémoire

- Les **champs statiques** existent et sont accessibles même si l'on n'a pas créé d'objets.
- On les représentera donc dans une zone mémoire liée à la classe et commune à toutes les instances de cette classe.

```
p1 = new Point(2.5, 4.2);
p2 = new Point(5.0, 6.0);
n  = Point.nextNr;
n  = p1.nextNr;
```

Bien que déconseillée, cette notation est acceptée par le langage.  
Le compilateur la remplacera par  
n = Point.nextNr;

Le champ statique nextNr est associé à la classe **Point** et est commun à toutes ses instances



# Modificateur final

- Le modificateur **final** peut être utilisé lors de la déclaration de variables locales, de paramètres de méthodes et de champs.
- Il indique que la valeur de la variable ou du champ ne peut plus être modifiée après l'affectation initiale.  
Cela revient donc à **déclarer des constantes**.
- Il est très fréquent de déclarer les constantes générales comme **champs statiques** au niveau de la classe et de les déclarer **public**.  
Exemple : `public static final double PI = 3.141592653589793;`
- Les champs déclarés constants (**static final**) sont habituellement écrits en majuscules et utilisent le caractère sous-ligné '\_' comme séparateur.  
Exemple : `private static final double FREQUENCY_MAX = 3.5E9;`
- Pour les classes et les méthodes, une autre utilisation du mot-clé **final** sera vue dans le cadre de l'héritage.

## Initialisation de classes

- Le langage Java permet d'écrire du code d'initialisation de classe qui sera exécuté lors de l'élaboration de la classe (en même temps que l'initialisation des champs statiques).
- Ce code appelé **initialiseur statique** est simplement constitué d'un **bloc d'instructions** (entre accolades) **précédé du mot-clé static**.
- Une classe peut posséder un nombre quelconque d'initialiseurs statiques qui peuvent être placés partout où une déclaration de champ ou une déclaration de méthode est permise (ils seront exécutés dans ordre d'écriture du code dans le fichier source).
- Les initialiseurs statiques peuvent être considérés comme des méthodes statiques anonymes.
- Les restrictions définies pour les méthodes statiques s'appliquent également aux instructions des initialiseurs statiques.

## Initialiseur statique

- Exemple d'initialisation statique :

```
-----  
// Calcule à l'avance et mémorise certaines valeurs  
// trigonométriques dans un tableau  
-----  
public class Sinusoid {  
  
    private static int      nbPoints = 500;  
    public  static double[] valSine  = new double[nbPoints];  
  
    // Initialisation du tableau valSine  
    static {  
        double  x = 0.0;  
        double dx = (Math.PI/2)/(nbPoints-1);  
  
        for(int i=0; i<nbPoints; i++, x+=dx) {  
            valSine[i] = Math.sin(x);  
        }  
    }  
    ...  
}
```

## Initialiseur d'instance [1]

- Les classes peuvent également posséder des **initialiseurs d'instance**.
- Un initialiseur d'instance **initialise un objet** (contrairement à l'initialiseur statique qui initialise une classe)
- Un initialiseur d'instance est constitué d'un **bloc d'instructions entre accolades** (il peut y en avoir plusieurs dans une classe, ils seront exécutés dans l'ordre d'écriture du code dans le fichier source).
- Les initialiseurs d'instance sont exécutés après le constructeur de la classe parente et avant le constructeur de la classe dans laquelle ils sont déclarés.

Conseil : Il est généralement préférable d'effectuer les initialisations de l'objet dans le **constructeur** de la classe.

## Initialiseur d'instance [2]

- Exemple d'initialiseur d'instance :

```
public class ... {  
    ...  
    private static int    dim    = 1000;  
    private      int[] intArr = new int[dim];  
  
    // intArr initialization  
    {  
        for(int i=0; i<dim; i++) {  
            intArr[i] = i;  
        }  
    }  
    ...  
}
```

## Finaliseur d'objet : finalize

- Un **finaliseur** représente l'opposé d'un constructeur et exécute du code de finalisation de l'objet.
- Le ramasse-miettes (*garbage-collector*) se charge de récupérer automatiquement l'espace mémoire utilisé par les objets (dès que ces derniers ne sont plus référencés). Le programmeur n'a pas à se soucier de cet aspect.
- Cependant, si l'objet contient d'autres types de ressources (fichiers ouverts, connexions réseau, transactions dans une base de données, etc.) il peut être nécessaire d'écrire du code de finalisation qui devra s'exécuter avant que l'objet disparaisse.
- Un finaliseur est une méthode d'instance (héritée de `Object`)
  - dont le nom est `finalize()`
  - qui ne possède aucun paramètre
  - qui ne retourne aucune valeur (`void`)
- La méthode `finalize()` (si elle a été définie) sera invoquée automatiquement par la machine virtuelle avant que le ramasse-miettes ne détruisse l'objet.

# Wrappers [1]

- En Java, pour des raisons d'efficacité, les types primitifs ne sont pas des objets.
- Dans certaines circonstances, il peut être utile de pouvoir traiter les types primitifs comme des objets (par exemple pour pouvoir les enregistrer dans des structures de données abstraites de type *liste*, *pile*, *arbre*, ... ne manipulant généralement que des objets).
- Ainsi, il existe pour chaque type primitif, une classe **Wrapper** (classe d'emballage, classe enveloppe) qui permet de convertir (d'emballer) une variable (ou une valeur littérale) de type primitif en un objet correspondant (une instance d'une des classes d'emballage).
- Les classes *Wrapper* sont déclarées dans le paquetage `java.lang` (et sont donc accessibles sans importation explicite).
- Les classes *Wrapper* disposent d'un certain nombre de constantes et de méthodes (statiques et non-statiques) permettant d'effectuer diverses conversions.

# Wrappers [2]

- Liste des classes d'emballage (*Wrappers*) :

Type primitif	Classe Wrapper
byte	Byte
short	Short
int	Integer <i>(Et non pas Int !)</i>
long	Long
float	Float
double	Double
char	Character <i>(Et non pas Char !)</i>
boolean	Boolean

- A deux exceptions près, le nom de la classe *Wrapper* correspond à celui du type primitif avec une majuscule initiale.
- Les classes *Wrapper* créent des **objets immuables** !

## Wrappers [3]

- Exemples d'utilisation des classes *Wrapper* :

```
String str    = "1.23";
float  primF = 3.456F;
Float  objF;

primF = Float.MAX_VALUE;           // Plus grande valeur positive de type float
primF = Float.MIN_VALUE;           // Plus petite valeur positive de type float

objF  = new Float(primF);          // Conversion float -> Float
primF = objF.floatValue();          // Conversion Float -> float
objF  = new Float("2.34");          // Conversion String -> Float
objF  = Float.valueOf(str);          // Conversion String -> Float
str   = objF.toString();            // Conversion Float -> String
str   = Float.toString(primF);          // Conversion float -> String
primF = Float.parseFloat(str);          // Conversion String -> float
```

Ces exemples, donnés pour les types **float** et **Float**, s'appliquent par analogie aux autres types primitifs (avec quelques légères différences pour les types **boolean** et **char**).

# Autoboxing / Unboxing

- Un mécanisme de conversions automatiques entre types primitifs et classes *Wrapper* a été introduit dans le langage.
- Lorsque c'est possible, le compilateur génère automatiquement le code de conversion entre les variables de types primitifs et les objets des classes *Wrapper* :

Type primitif → Classe *Wrapper* [ **Autoboxing** ]

Classe *Wrapper* → Type primitif [ **Unboxing** ]

```
...
int      i = 12, j, r;
Integer objA = new Integer(34), objB;

objB = i;                      // Autoboxing
j    = objA;                    // Unboxing
r    = ++objA + i;
...
```



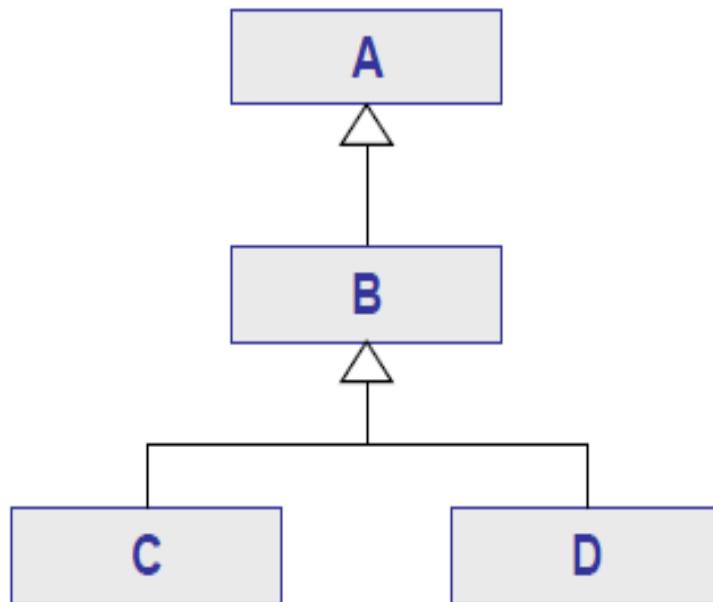
# Héritage et Polymorphisme

## Sous-classe et héritage

- L'**héritage** est une **propriété essentielle** de la programmation orientée objet.
- Ce mécanisme permet d'ajouter des fonctionnalités à une classe (une **spécialisation**) en créant une **sous-classe** qui hérite des propriétés de la classe parente et à laquelle on peut ajouter des propriétés nouvelles.
- La classe qui hérite est appelée **sous-classe** (ou **classe dérivée**) de la **classe parente** (qui est également appelée **super-classe**).
- L'héritage permet à une sous-classe d'**étendre les propriétés** de la classe parente tout en héritant des champs (attributs) et des méthodes (comportement) de cette classe parente.
- En Java, une classe ne peut hériter que d'une seule classe parente. On parle dans ce cas d'**héritage simple** (par opposition à l'héritage multiple qui permet à une classe d'hériter de plusieurs classes parentes).

# Arborescence de classes

- L'héritage induit une relation arborescente entre les classes.



En notation UML, le symbole



signifie

"est une sous-classe de"  
"hérite de"

- La classe **A** est la classe parente (super-classe) de **B**
- La classe **B** est la classe parente de **C** et **D**
- La classe **B** est une sous-classe de **A**
- Les classes **C** et **D** sont des sous-classes de **B**

**B** joue plusieurs rôles

## Exemple : classe Vehicule

- Pour illustrer le concept, prenons l'exemple d'une classe **Vehicule** permettant de décrire (très sommairement) les propriétés et les comportements d'un véhicule du monde réel.
- Avec une modélisation très simpliste, la classe pourrait être représentée de la manière suivante :

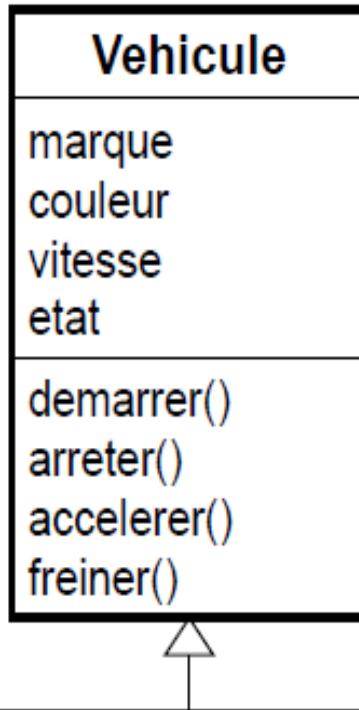
Vehicule	
marque	
couleur	
vitesse	
etat	
demarrer()	
arreter()	
accelerer()	
freiner()	

# Implémentation classe Vehicule

```
public class Vehicule {  
    private String marque;  
    private String couleur;  
    private double vitesse;          // Vitesse actuelle  
    private int etat;                // 0:arrêt, 1:marche, ...  
  
    public Vehicule(String marque, String couleur) {  
        this.marque = marque;  
        this.couleur = couleur;  
        vitesse = 0.0;  
        etat = 0;  
    }  
  
    public void demarrer() { etat = 1; }  
    public void arreter() { if (vitesse == 0) etat = 0; }  
    public void accelerer() {  
        if (etat == 1) vitesse += 5.0;  
    }  
  
    public void freiner() {  
        if (vitesse >= 5.0) vitesse -= 5.0;  
        else vitesse = 0.0;  
    }  
}
```

# Sous-classes de Vehicule

A partir de la classe **Vehicule** on peut, en utilisant l'héritage, créer de **nouvelles classes** (**Voiture** et **Camion**) qui **spécialisent** la classe **Vehicule** en y ajoutant de nouvelles propriétés.



Dans le langage UML, un tel diagramme est appelé "**Diagramme de classes**". Il montre les relations entre les classes d'une application

## Sous-classes Voiture et Camion

---

- La classe **Voiture** est une sous-classe de **Vehicule**. Elle hérite de tous les attributs et méthodes de **Vehicule** (**marque**, **couleur**, ..., **freiner()**) en y ajoutant deux nouveaux attributs (**modele** et **nbPortes**).
- La classe **Camion** est également une sous-classe de **Vehicule** et hérite de tous ses attributs et méthodes. La classe **Camion** étend la classe parente (**Vehicule**) en y ajoutant deux nouveaux attributs (**chargeMax** et **poidsChangement**) ainsi que deux nouvelles méthodes (**charger()** et **decharger()**).
- Dans les sous-classes **Voiture** et **Camion**, on peut utiliser les attributs et les méthodes héritées (par exemple **couleur** ou **freiner()**) comme si ces membres avaient été déclarés dans chacune des sous-classes (sous réserve, naturellement, que les **droits d'accès** le permettent).

# Déclaration de sous-classe

- La déclaration d'une sous-classe s'effectue en utilisant le mot-clé **extends** suivi du nom de la classe parente :

```
public class Voiture extends Vehicule {  
  
    private String modele;                      // Nouveau champ  
    private int    nbPortes;                     // Nouveau champ  
  
    public Voiture(String marque,               // Constructeur  
                   String modele,  
                   String couleur,  
                   int    nbPortes) {  
  
        super(marque, couleur); // Appelle le constructeur de la classe parente  
  
        this.modele  = modele;  
        this.nbPortes = nbPortes;  
    }  
}
```

## Relations entre classes [1]

- L'héritage entre une sous-classe et sa classe parente est caractérisé par une **relation** de type

ou      ***"Est un..."***      ("Is a...")  
ou      ***"Est une sorte de..."***      ("Is kind of...")

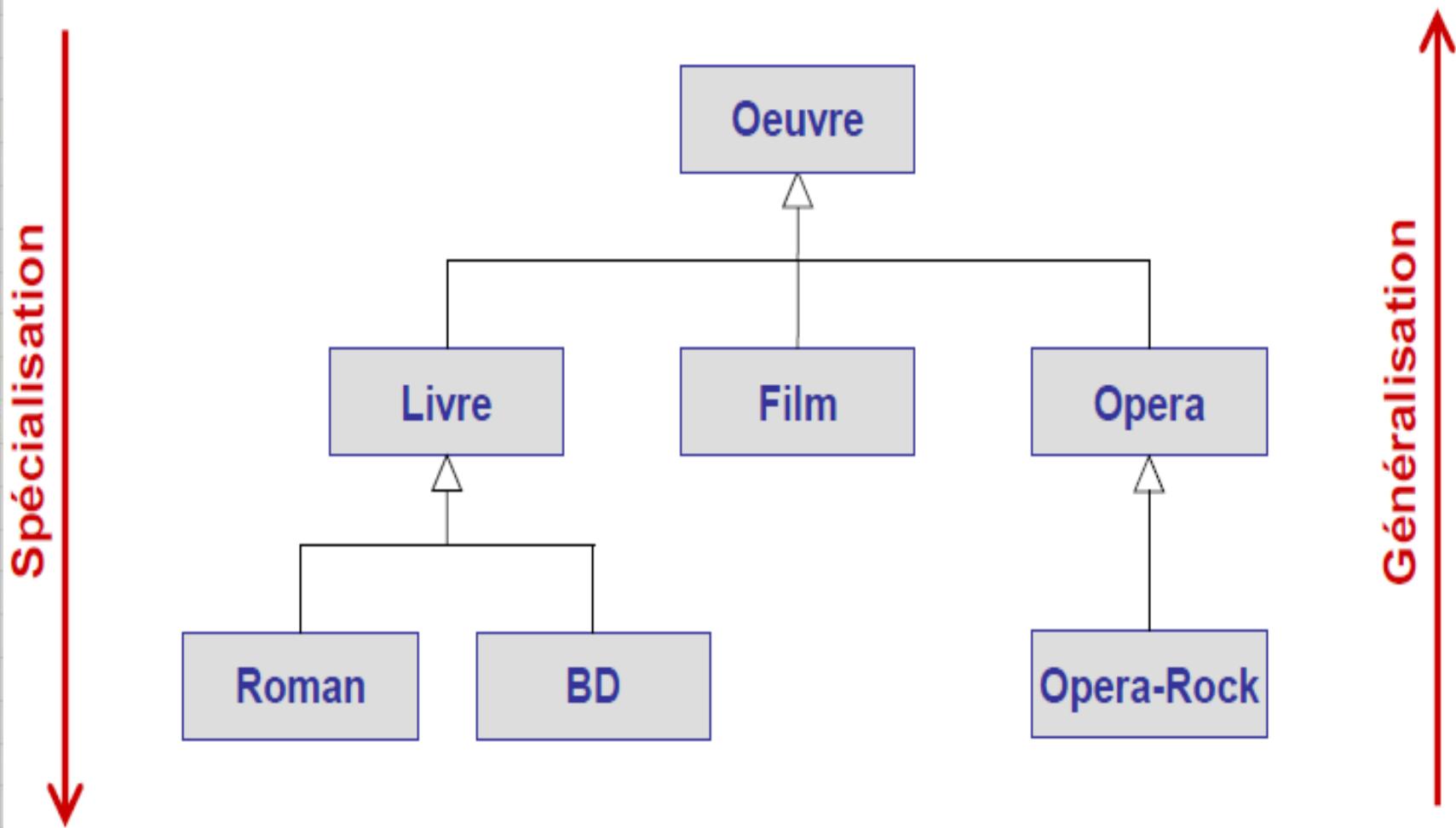
- Une voiture **est un** véhicule
  - Un camion **est un** véhicule

**Attention : l'inverse n'est pas forcément vrai !**

- Une sous-classe crée une **spécialisation** de la classe parente. A l'inverse, on parle de **généralisation** lorsqu'on passe des sous-classes à leur classe parente.
  - Point à retenir lors de la conception d'une application :

"Est un..." ⇒ Héritage

# Généralisation / Spécialisation



## Relations entre classes [2]

- Il existe une autre relation importante qui peut exister entre deux classes. Il s'agit de la relation de **composition** (ou **agrégation**) qui est caractérisée par une **relation** de type

- "*A un...*" ("Has a...")
- ou "*Possède un...*"
- ou "*Est composé de...*"
- ou "*Contient...*"

- Une voiture **possède un** moteur
  - Une voiture **a un** propriétaire

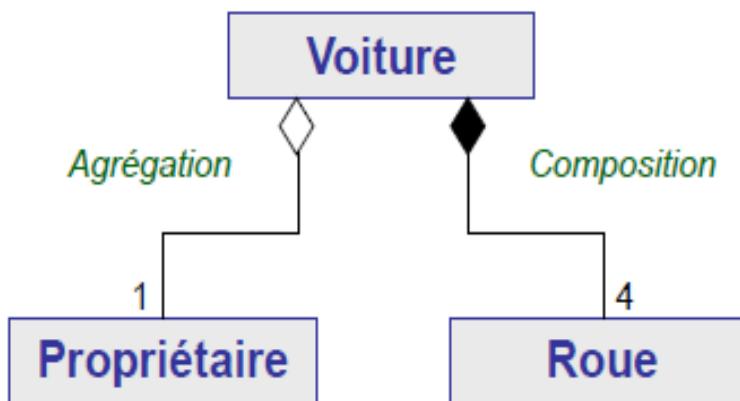
"A un..." ⇒ Composition  
⇒ Agrégation

- En Java, les relations de composition sont réalisées en créant dans la classe "contenant" une **référence vers** un objet de la classe "contenu".

```
public class Voiture extends Vehicule {  
    private Personne propriétaire;  
    private FuelMotor moteur;  
    ...  
}
```

## Relations entre classes [3]

- En **notation UML**, la relation "*A un*", "*Possède un*", ... est représentée de la manière suivante :



- La distinction sémantique entre **composition** et **agrégation** ne se traduit pas (en Java) par des différences d'implémentation.

```
public class Voiture extends Vehicule {  
    ...  
    private Personne leProprietaire;  
    private Roue[] lesRoues;  
    ...  
}
```

# Généralisation

- La relation d'héritage ("*Est un...*") permet de traiter les objets des sous-classes comme s'ils étaient des objets de leur classe parente (par généralisation).

Important !

```
Camion    c1 = new Camion(...);  
Vehicule  v1 = c1;           // Ok, un camion est un véhicule  
Camion    c2 = v1;           // ERREUR, un véhicule n'est pas forcément un camion !  
Camion    c3 = (Camion)v1;   // Ok, v1 référence effectivement un camion
```

- Si nécessaire, le système effectue donc une **conversion élargissante** (automatique) de la sous-classe vers la classe parente (*Upcasting*).
- Une **conversion explicite** (transtypage, *casting*) d'un objet de la classe parente vers un objet de la sous-classe (*Downcasting*) est possible si l'instance à convertir référence effectivement (au moment de l'exécution) un objet de la sous-classe considérée (sinon, il y aura une erreur *ClassCastException* à l'exécution).

# Opérateur instanceof

- L'opérateur **instanceof** permet de tester (à l'exécution) l'appartenance d'un objet à une classe (ou une interface) donnée.
- Dans l'exemple précédent, on aurait pu écrire :

```
if (v1 instanceof Camion) {  
    c3 = (Camion)v1;  
}  
else {  
    ... // v1 ne référence pas un Camion  
}
```

Un tel test permet d'éviter une erreur fatale (à l'exécution) si la variable **v1** ne référence pas un objet de type **Camion**.

Remarque 1 : **(v1 instanceof Vehicule)** est également vrai !

Remarque 2 : L'*upcasting* et le *downcasting* n'engendrent aucune opération ni changement en mémoire (c'est une autre "vue" de l'objet)

# Classe Object

- En Java, chaque classe que l'on crée possède une classe parente.
- Si l'on ne définit pas explicitement une classe parente (avec `extends`), la super-classe par défaut est **Object** (qui est déclarée dans le paquetage `java.lang`).
- La classe **Object** est donc l'ancêtre de toutes les classes Java (c'est la racine unique de l'arbre des classes).
- La classe **Object** est la seule classe Java qui ne possède pas de classe parente.
- Toutes les classes héritent donc des méthodes de la classe **Object** (par exemple `toString()`, `equals()`, `finalize()`, etc).
- La classe **Object** constitue la généralisation ultime :  
*Tous les objets sont des Object !*

## Chaînage des constructeurs

- Un constructeur d'une sous-classe peut faire **appel à un constructeur de la classe parente** en utilisant le mot réservé **super** selon la syntaxe suivante :  
*super(Expr1, Expr2, ...);*
- Si un constructeur d'une sous-classe invoque explicitement un constructeur de la classe parente, l'instruction **super(...)** doit être **la première instruction du constructeur**.
- Si l'on ne fait pas explicitement appel à un constructeur de la classe parente, une **invocation du constructeur par défaut** de la super-classe (constructeur sans paramètre) sera **automatiquement ajoutée** (comme première instruction). Si un tel constructeur n'existe pas dans la classe parente, une erreur sera générée à la compilation.
- Le langage garantit ainsi que tous les éléments des classes parentes aient été élaborés avant la création d'un objet de la classe considérée.

# Masquage des champs

- Si une sous-classe définit un champ avec le même nom qu'un champ de sa classe parente (une pratique **à éviter**), alors le champ de la super-classe est **masqué** dans le corps de la sous-classe.
- Dans ce cas, on peut, dans le corps de la sous-classe accéder au champ de la classe parente en utilisant le mot-clé **super** suivi d'un point et du nom du champ.
- Si les classes **A** et **B** définissent toutes deux un champ **x** et que **B** est une sous-classe de **A** alors, dans les méthodes de **B**, on peut utiliser :

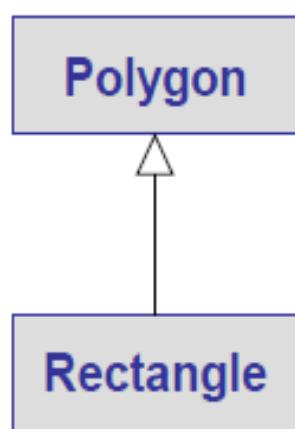
<code>x</code>	Champ <b>x</b> de la classe <b>B</b>
<code>this.x</code>	Champ <b>x</b> de la classe <b>B</b>
<code>super.x</code>	Champ <b>x</b> de la classe <b>A</b>
<code>((A)this).x</code>	Champ <b>x</b> de la classe <b>A</b> (par transtypage)

*La notation `super.super.x` n'est pas autorisée, seul le casting `((A)c).x` permet d'accéder à un champ masqué d'une classe ancêtre.*

- Les champs statiques peuvent également être masqués mais ils restent accessibles en les préfixant avec le **nom de la classe**.

# Redéfinition des méthodes [1]

- Lorsqu'une classe définit une **méthode d'instance** en utilisant la même signature (même nom, type de retour [év. une sous-classe], paramètres, etc.) qu'une méthode de sa classe parente, cette méthode **redéfinit (overrides)** la méthode de sa super-classe.
- Par exemple :



```
public class Polygon {  
    . . .  
    public double area() {  
        . . . // Calcul de la surface d'un polygone quelconque  
    }  
    . . .  
}
```

```
public class Rectangle extends Polygon {  
    . . .  
    public double area() {  
        . . . // Calcul de la surface du rectangle (longueur·largeur)  
    }  
    . . .  
}
```

## Redéfinition des méthodes [2]

- Les classes `Polygone` et `Rectangle` définissent toutes deux une méthode `area()` avec une signature identique.
- La méthode invoquée par un appel `obj.area()` dépendra du type de l'objet référencé par la variable `obj` lors de l'exécution de cette instruction.
- C'est **toujours la méthode associée au type effectif de l'objet référencé** qui est exécutée, même si l'objet est enregistré dans une variable déclarée avec le type d'une classe parente :

```
Rectangle r = new Rectangle(2.6, 5.3);

Polygon p = r;           // Conversion élargissante (Upcasting)

double s = p.area();     // Invoque r.area() et non pas area() de la
                        // classe Polygon car la variable p référence
                        // un objet de type Rectangle
```

## Redéfinition des méthodes [3]

- Dans la sous-classe `Rectangle`, il est possible d'invoquer la méthode `area()` de la classe parente `Polygon` en utilisant la notation suivante :

super.area()

Remarque : La notation `super.super.f()` n'est pas autorisée et il est impossible d'accéder à une méthode masquée d'une classe ancêtre (autre que la classe parente).

- A l'extérieur de la classe de déclaration, il n'est, par contre, pas possible pour un objet de type `Rectangle`, d'invoquer la méthode `area` de la classe `Polygon`.
- Attention à bien distinguer (et ne pas confondre) les notions de **surcharge** (*overloading*) et de **redéfinition** (*overriding*) de méthodes.

## Redéfinition des méthodes [4]

- L'annotation `@Override` peut être utilisée pour indiquer explicitement l'intention de redéfinir une méthode.
- Bien qu'étant facultative, elle apporte deux avantages :
  - Si on se trompe en écrivant le nom de la méthode lors de la redéfinition, le compilateur peut alors informer le programmeur de son erreur.
  - La redéfinition étant explicite, le code est plus clair.

```
@Override  
public double area() { ... }
```

## Redéfinition des méthodes [5]

- Les **méthodes statiques** (méthodes de classe) peuvent être masquées, dans des sous-classes, par des méthodes statiques possédant le même nom (\*\* une pratique **à éviter** \*\*).
- Ces méthodes masquées restent cependant accessibles en les préfixant avec le nom de la classe dans laquelle elles sont définies (comme il est recommandé de le faire avec toutes les méthodes statiques).
- On ne peut donc pas spécialiser une méthode statique dans une sous-classe (comme les méthodes statiques ne sont jamais associées à un objet, le polymorphisme ne s'applique pas).
- En outre, une **méthode statique ne peut pas masquer une méthode d'instance** d'une super-classe (erreur à la compilation).

## Redéfinition des méthodes [6]

- Si une méthode est redéfinie dans une sous-classe, le **type de retour doit être identique** à celui de la méthode correspondante de la classe parente.
- En redéfinissant une méthode, il est possible d'étendre sa **zone de visibilité** (`private` → `package` → `protected` → `public`) mais non de la restreindre.
- Une méthode redéfinie ne peut pas générer plus d'**exceptions** contrôlées que celles qui sont déclarées dans la méthode de la classe parente (mais elle peut en déclarer moins).
- Éviter de :
  - surcharger une méthode qui est redéfinie
  - redéfinir une partie des méthodes surchargées

car cela provoque des risques de confusion et crée des pièges lors des modifications ultérieures des classes.

# Modificateur final

- Dans la **déclaration d'une classe**, le modificateur **final** indique que la classe ne peut pas être sous-classée (on ne peut pas créer de classes dérivées).
- Dans la **déclaration d'un champ**, le modificateur **final** indique que la valeur du champ ne peut pas être modifiée après l'affectation initiale (dans la déclaration ou dans le constructeur). Permet de définir des valeurs constantes.
- Dans la **déclaration d'une méthode**, le modificateur **final** indique que la méthode ne peut pas être redéfinie dans une sous-classe.
- Dans la **déclaration des paramètres d'une méthode**, le modificateur **final** indique que la valeur de ces paramètres ne peut pas être modifiée (il s'agit de paramètres *d'entrée* de la méthode).

**Remarque :** L'utilisation du modificateur **final** permet en outre au compilateur d'effectuer certaines optimisations : mise en ligne de méthodes (*inlining*), suppression de la recherche dynamique (*late binding*), etc.

# Modificateur **protected**

---

- Le modificateur **protected** appliqué aux membres (champs ou méthodes) d'une classe indique que ces champs ne sont accessibles que dans la classe de définition, dans les classes du même paquetage et dans les sous-classes de cette classe (indépendamment du paquetage).
- Il s'agit d'un accès plus restrictif que **public** mais - contrairement à ce que son nom pourrait laisser croire - **moins restrictif que l'accès par défaut (package)**.
- Le modificateur **protected** devrait être utilisé avec les champs et les méthodes qui ne sont pas requis par les utilisateurs de la classe mais qui pourraient s'avérer utiles à la création de sous-classes dans d'autres paquetages.

# Modificateur **private**

- Le modificateur **private**, appliqué aux membres (champs ou méthodes) d'une classe, indique que ces champs ne sont accessibles que dans la classe de définition.
- Même s'il ne sont pas accessibles dans les sous-classes, les champs privés sont **malgré tout hérités** dans les sous-classes (une zone mémoire leur est allouée).
- Il s'agit de l'accès **le plus restrictif**.
- Le modificateur **private** devrait être utilisé avec les champs et les méthodes qui ne sont utilisés qu'au sein de la classe de définition et qui devraient être cachés partout ailleurs.

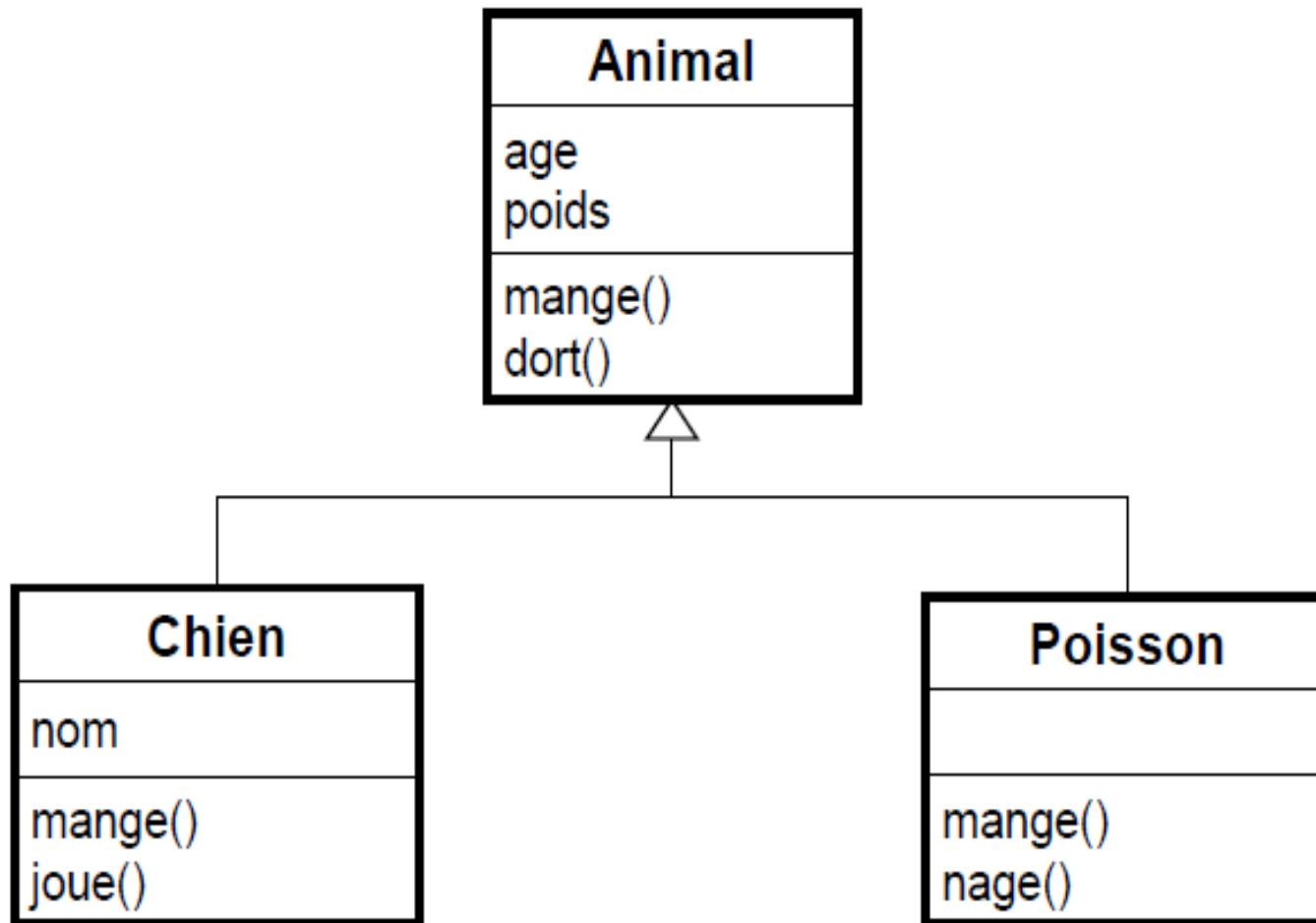
**Conseil** : D'une manière générale il vaut mieux **commencer par un accès restrictif** aux membres d'une classe (⇒ **private**).

Si nécessaire, il est toujours possible de relâcher les restrictions dans des versions ultérieures de la classe.

Un relâchement des restrictions préserve la **compatibilité ascendante** (ce qui n'est pas le cas si l'on restreint les droits d'accès).

# Polymorphisme [1]

- Si l'on considère le diagramme de classes suivant :



## Polymorphisme [2]

- On peut déduire du diagramme de classes les considérations suivantes :
  - Chien et Poisson sont des sous-classes d'Animal
  - Animal est la classe parente de Chien et de Poisson
  - Chien hérite des membres d'Animal (age, poids, dort())
  - Chien ajoute le champ nom
  - Chien ajoute la méthode joue()
  - Chien redéfinit la méthode mange()
  - Poisson hérite des membres d'Animal (age, poids, dort())
  - Poisson ajoute la méthode nage()
  - Poisson redéfinit la méthode mange()

# Polymorphisme [3]

- Si l'on implémente les classes `Animal`, `Chien` et `Poisson` et que l'on écrit le code suivant :

```
Chien milou = new Chien(...);  
milou.mange();
```

- La méthode `mange()` qui sera invoquée est celle qui est définie dans la sous-classe `Chien` (car `milou` est du type `Chien` et la méthode `mange()` est redéfinie pour cette sous-classe).
- Si la sous-classe `Chien` ne redéfinissait pas la méthode `mange()`, ce serait alors la méthode `mange()` d'`Animal` qui serait invoquée (la classe `Chien` en hériterait).

# Polymorphisme [4]

- Comme un **Chien** est un **Animal** (relation d'héritage) on peut écrire le code suivant :

```
Animal toutou = new Chien(...);  
toutou.mange();
```

- Quelle méthode **mange()** sera invoquée dans ce cas ?
- C'est toujours la méthode **mange()** définie dans la sous-classe **Chien** qui sera invoquée.
- Même si **toutou** est une référence de type **Animal**, c'est **le type de l'objet référencé qui détermine la méthode qui sera appelée** (comportement décrit sous "*Redéfinition des méthodes*").

# Polymorphisme [5]

---

- En Java, dans la plupart des situations où il y a des relations d'héritage, la détermination de la méthode à invoquer n'est pas effectuée lors de la compilation.
- C'est seulement à l'exécution que la machine virtuelle déterminera la méthode à invoquer selon le type effectif de l'objet référencé à ce moment là.
- Ce mécanisme s'appelle "**Recherche dynamique de méthode**" (**Late Binding** ou **Dynamic Binding**).
- Ce mécanisme de recherche dynamique (durant l'exécution de l'application) sert de base à la mise en oeuvre de la propriété appelée **polymorphisme**.

# Polymorphisme [6]

- On pourrait définir le **polymorphisme** comme la propriété permettant à un programme de **réagir de manière différenciée à l'envoi d'un même message** (invocation de méthode) en fonction des objets qui reçoivent ce message.
- Il s'agit donc d'une aptitude **d'adaptation dynamique du comportement** selon les objets en présence.
- Avec l'**encapsulation** et l'**héritage**, le **polymorphisme** est une des propriétés essentielles de la programmation orientée objet.

**Remarque :** La surcharge de méthodes peut également être considérée comme une forme de polymorphisme. Le choix de la méthode à invoquer est cependant déterminé à la compilation

## Exemple de polymorphisme [1]

- L'exemple qui suit est destiné à illustrer le principe du polymorphisme. Il se base sur les classes précédemment définies (**Animal**, **Chien** et **Poisson**).
- Si l'on souhaite enregistrer et manipuler une collection d'animaux (une ménagerie) on peut créer et alimenter le tableau suivant :

```
// Déclaration et création du tableau
Animal[] menagerie = new Animal[6];

// Alimentation du tableau
menagerie[0] = new Poisson(...);
menagerie[1] = new Chien(...);
menagerie[2] = new Chien(...);
menagerie[3] = new Animal(...);
menagerie[4] = new Poisson(...);
menagerie[5] = new Chien(...);
```

## Exemple de polymorphisme [2]

- Le polymorphisme nous permet d'écrire une méthode `nourrir` dont la fonction est de donner à manger à chaque animal contenu dans le tableau passé en paramètre en appelant successivement la méthode `mange()` pour chacun d'eux.

```
-----  
// Appelle la méthode mange() pour chaque animal contenu  
// dans le tableau passé en paramètre  
-----  
public static void nourrir(Animal[] tabAnimaux) {  
    if (tabAnimaux == null) return;  
    for (int i=0; i<tabAnimaux.length; i++) {  
        if (tabAnimaux[i] != null) {  
            tabAnimaux[i].mange();  
        }  
    }  
}
```

## Exemple de polymorphisme [3]

- On peut ensuite appeler la méthode `nourrir()` en lui passant en paramètre la ménagerie précédemment créée :

```
nourrir(menagerie);
```

- Sur la base du contenu de `menagerie`, la méthode `nourrir()` appellera successivement :  
`mange() de la classe Poisson`  
`mange() de la classe Chien`  
`mange() de la classe Chien`  
`mange() de la classe Animal`  
`mange() de la classe Poisson`  
`mange() de la classe Chien`
- La méthode `nourrir()` n'a pas besoin de déterminer elle-même quelle méthode doit être appelée pour chaque animal.
- Le **polymorphisme** fera en sorte que le message `mange()` soit interprété (à l'exécution) de manière appropriée selon les objets qui le reçoivent (ainsi chaque animal mangera selon ses goûts !).

# Polymorphisme – Synthèse du mécanisme

- En Java, les variables sont typées. Avec la notion d'héritage, plusieurs «types» peuvent être compatibles avec une déclaration de variable.

```
short s; // Ne peut rien contenir d'autre qu'un short
Object o; // Peut contenir une référence vers un Object,
// ou un objet d'une sous-classe
```

- Le **compilateur** s'assure que les instructions sont compatibles avec la déclaration :

```
Object o;
o = "ah";
int i=o.length(); // Refusé, car length() &gt; Object
int i=((String)o).length(); // Ok si o référence un String
```

- A l'**exécution**, Java « suit toujours la flèche » pour atteindre les méthodes :

```
Animal a;
a = new Chien();
a.mange(); // Ce sera bien la méthode redéfinie par la
// classe Chien qui sera exécutée
```

```
public class Animal {
    void mange() {...}
}
```

```
public class Chien extends Animal {
    void mange() {...} // Redéfinition !
}
```



# Classes abstraites et Interfaces

# Classes et méthodes abstraites

- Une **classe abstraite** est une classe qui contient une ou plusieurs **méthodes abstraites**. Elle peut malgré tout contenir d'autres méthodes *habituelles* (appelée parfois méthodes concrètes).
- Une **méthode abstraite** est une méthode qui **ne contient pas de corps**. Elle possède simplement une signature de définition suivie du caractère point-virgule (pas de bloc d'instructions).
- Une **méthode abstraite** doit obligatoirement être déclarée avec le modificateur **abstract**.
- Exemple :  

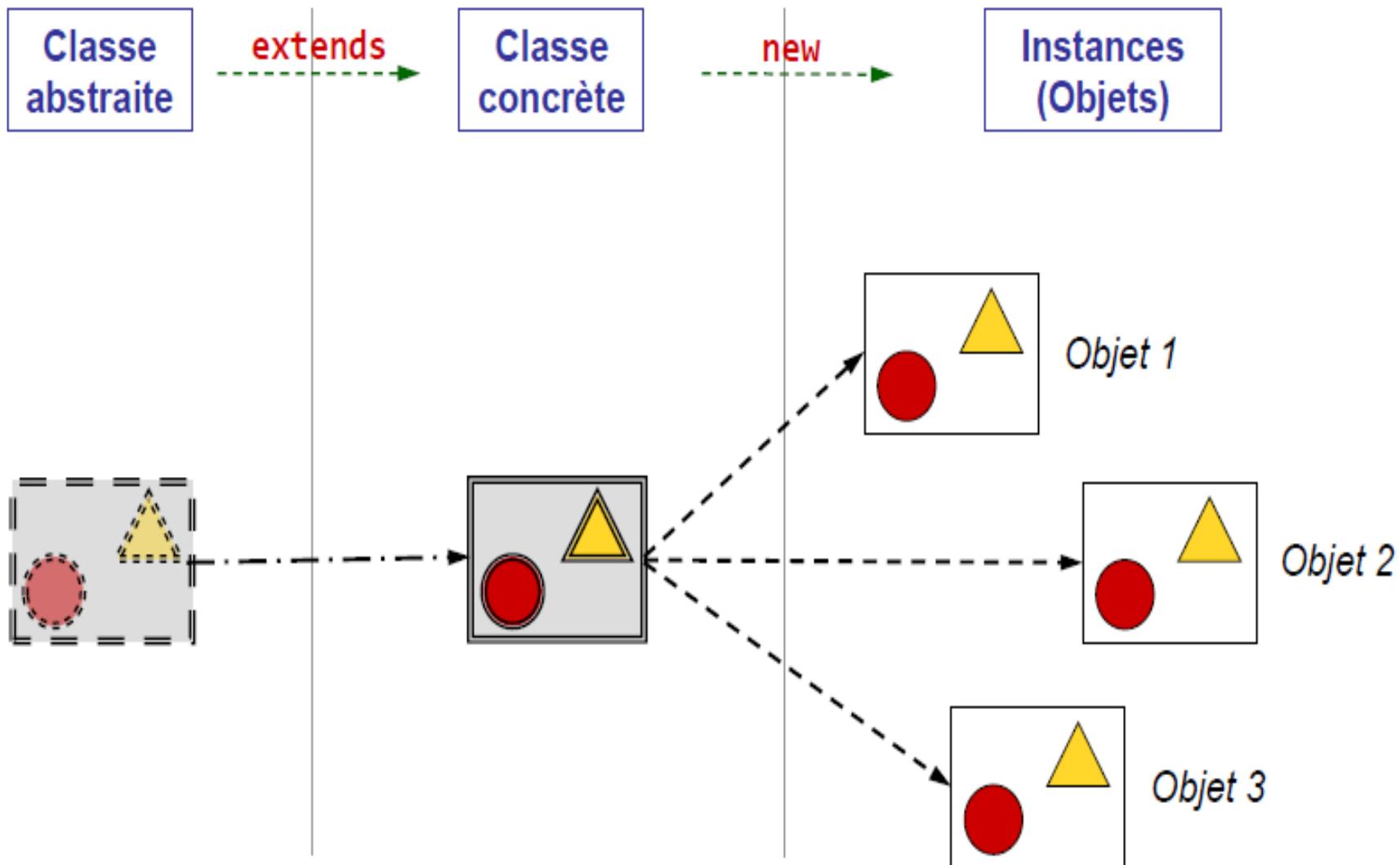
```
public abstract double area();
```
- Une classe possédant une ou plusieurs méthodes abstraites devient obligatoirement une **classe abstraite**. Cette classe doit également être déclarée avec le modificateur **abstract**.

## Classes abstraites : métaphore

---

- Comme les autres classes (et les interfaces), une **classe abstraite** définit un **nouveau type** qui peut être associé à un identificateur (déclaration de variables, champs, tableaux, paramètres formels, ...).
- Comme nous l'avons vu dans un des chapitres précédents, les classes peuvent être considérées comme étant des moules (des modèles, des plans, ...) permettant de fabriquer des objets.
- Si l'on poursuit avec cette métaphore, les **classes abstraites** peuvent être considérées comme étant des **moules incomplets** (des **modèles partiels**, des **plans non terminés**, ...) qui ne peuvent pas être utilisés tels quels pour créer des objets mais qui peuvent être utilisés pour fabriquer d'autres plans plus précis (représentés par des sous-classes) qui seront complétés et qui permettront, eux, de créer des objets (des instances des sous-classes).

# Classes abstraites



# Classes abstraites : règles [1]

- Les **règles** suivantes s'appliquent aux **classes abstraites** :
  - **Une classe abstraite ne peut pas être instanciée** : on ne peut pas créer d'objet en utilisant l'opérateur **new**.
  - Une sous-classe d'une classe abstraite ne peut être **instanciée** que si elle redéfinit chaque méthode abstraite de sa classe parente et qu'elle **fournit une implémentation** (un corps) pour **chacune des méthodes abstraites**.  
Une telle sous-classe est souvent appelée **sous-classe concrète** afin de mettre en évidence le fait qu'elle n'est pas abstraite.
  - Si une sous-classe d'une classe abstraite n'implémente pas toutes les méthodes abstraites dont elle hérite, cette sous-classe est elle-même abstraite (et ne peut donc pas être instanciée).
  - Les **méthodes** déclarées avec l'un des modificateurs suivants : **static**, **private** ou **final** ne peuvent pas être abstraites étant donné qu'elles ne peuvent pas être redéfinies dans une sous-classe.

## Classes abstraites : règles [2]

- Autres **règles** s'appliquant aux **classes abstraites** :
  - Une classe déclarée **final** ne peut pas contenir de méthodes abstraites car elle ne peut pas être sous-classée.
  - Une classe peut être déclarée **abstract** même si elle ne possède pas réellement de méthode abstraite.

Cela signifie que son implémentation est incomplète en certains points (p. ex. le corps de certaines méthodes doit être complété en fonction du contexte) et qu'elle est destinée à jouer le rôle de classe parente pour une ou plusieurs sous-classes qui achèveront l'implémentation.

Même si elle ne possède pas de méthodes abstraites, une telle

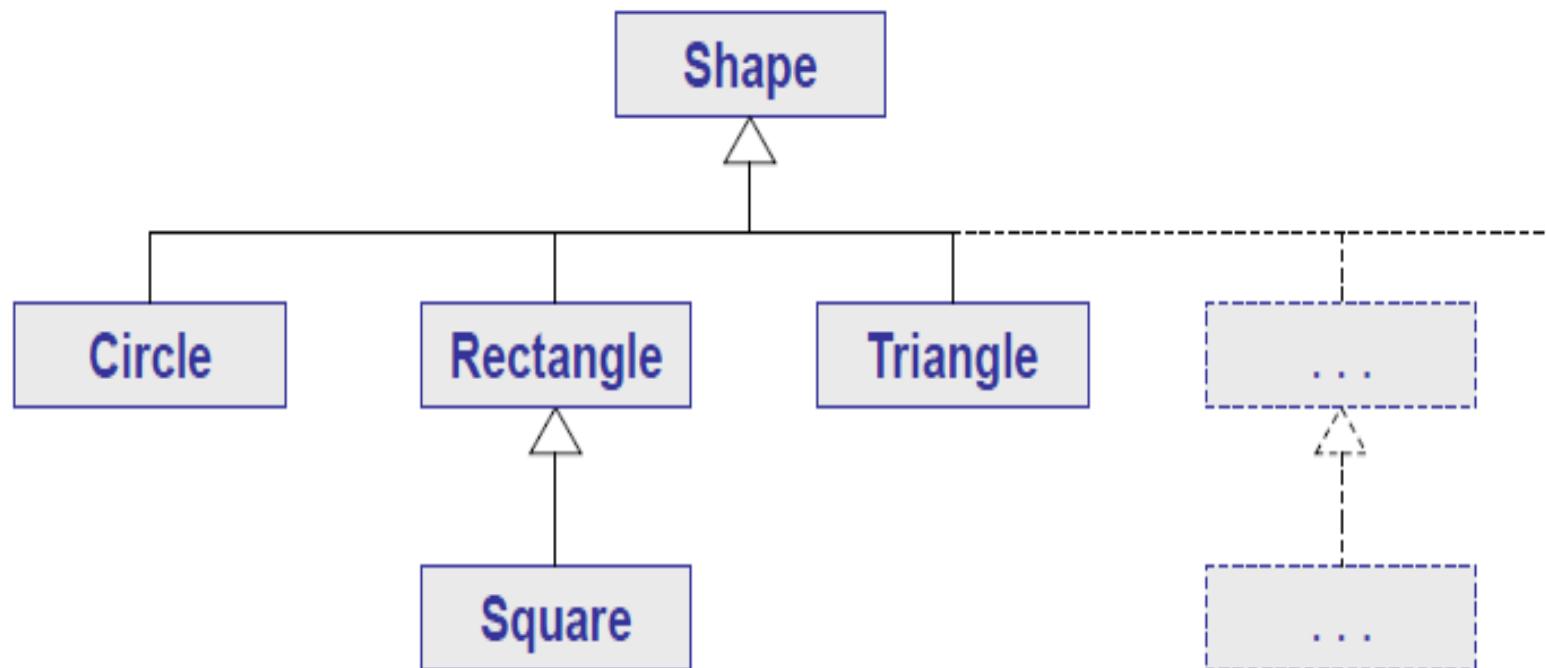
# Utilité des classes abstraites

---

- En pratique, les classes abstraites permettent de **définir des fonctionnalités** (des comportements) **que les sous-classes devront impérativement implémenter** même si la classe abstraite n'est pas en mesure de fournir elle-même une implémentation pour ces méthodes.
- Les utilisateurs des sous-classes d'une classe abstraite sont donc assurés de trouver toutes les méthodes définies dans la classe abstraite dans chacune des sous-classes concrètes.
- Les classes abstraites constituent donc une sorte de **contrat** (spécification contraignante) qui garantit que certaines méthodes seront disponibles dans les sous-classes et qui oblige les programmeurs à les implémenter dans toutes les sous-classes concrètes.

## Exemple de classe abstraite [1]

- Si l'on souhaite, par exemple, créer une hiérarchie de classes qui décrive des formes géométriques à deux dimensions, on pourrait envisager l'arborescence représentée par le diagramme de classes suivant :



## Exemple de classe abstraite [2]

- Sachant que toutes les formes possèdent les propriétés *périmètre* et *surface*, il est judicieux de placer les méthodes définissant ces propriétés (`perimeter()` et `area()`) dans la classe qui est à la racine de l'arborescence (`Shape`).
- La classe `Shape` ne peut cependant pas implémenter ces deux méthodes car on ne peut pas calculer le périmètre et la surface sans connaître le détail de la forme.
- On pourrait naturellement implémenter ces méthodes dans chacune des sous-classes de `Shape` mais il n'y a aucune garantie que toutes les sous-classes les possèdent (ce serait laissé au bon vouloir des programmeurs des sous-classes).
- Il est possible de résoudre ce problème en déclarant dans la classe `Shape` deux méthodes abstraites `perimeter()` et `area()` ce qui rend la classe `Shape` elle-même abstraite et impose aux créateurs de sous-classes de les implémenter.

## Exemple de classe abstraite [3]

```
//
// Shape
//
public abstract class Shape {          // Classe abstraite
    public abstract double perimeter(); // Méthode abstraite
    public abstract double area();     // Méthode abstraite
}

//
// Circle
//
public class Circle extends Shape {
    public static final double PI = 3.14159265358979;
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() { return radius; }

    public double perimeter() { return 2*PI*radius; }

    public double area()      { return PI*radius*radius; }
}
```

## Exemple de classe abstraite [4]

```
-----  
// Rectangle  
-----  
public class Rectangle extends Shape {  
    protected double length, width;  
  
    public Rectangle(double length, double width) {  
        this.length = length;  
        this.width = width;  
    }  
  
    public double getLength() { return length; }  
  
    public double getWidth() { return width; }  
  
    public double perimetre() {  
        return 2*(length + width);  
    }  
  
    public double surface() {  
        return length * width;  
    }  
}
```

## Exemple de classe abstraite [5]

- Les autres sous-classes (Square, Triangle, ...) pourraient être implémentées selon le même modèle.
- Même si la classe Shape est abstraite, il est tout de même **possible de déclarer des variables de ce type** qui pourront recevoir des objets créés à partir des sous-classes concrètes :

```
Shape[] drawing = new Shape[3];  
  
drawing[0] = new Rectangle(2.5, 6.8);    // Conversion élargissante  
drawing[1] = new Circle(4.66);          // Conversion élargissante  
drawing[2] = new Square(0.125);         // Conversion élargissante
```

- Le polymorphisme permet ensuite d'invoquer une méthode commune sur chacun des objets :

```
double totalArea = 0.0;  
  
for (int i=0; i<drawing.length; i++) {    // Calcule la somme  
    totalArea += drawing[i].area();          // des surfaces  
}
```

## Interface [1]

- Comme une classe et une classe abstraite, une **interface** permet de définir un **nouveau type** (référence).
- Le comportement des objets de ce type (les opérations que l'on peut effectuer) sera déterminé par le contenu (la définition) de cette interface.
- Une interface constitue essentiellement une **spécification** qui laisse de côté les détails d'implémentation.
- L'interface définit ainsi une sorte de **contrat** que les classes qui l'implémenteront s'engagent à respecter.
- Les interfaces jouent un rôle important dans la phase de conception (API) d'applications, de bibliothèques ou de systèmes basés sur des ensembles d'objets qui communiquent.
- Par certains aspects, la notion d'**interface** est assez proche de la notion de **classe abstraite** mais elle comporte cependant un certain nombre de spécificités qui l'en distinguent et qui en étendent le potentiel.

## Interface [2]

- Du point de vue syntaxique, la définition d'une interface est proche de celle d'une classe abstraite.
- Il suffit de remplacer les mots-clés `abstract class` par le mot-clé **interface** :

```
public interface Printable {  
    public void print();  
}
```

- Une fois l'interface déclarée, il est possible de déclarer des variables de ce (nouveau) type :

```
Printable    document;  
Printable[]  printSpool;
```

- Le **corps d'une interface** peut contenir :
  - Des **constantes** qui sont implicitement publiques
    - ✓ Les modificateurs **static** et **final** ne sont pas nécessaires (implicites).
  - Des **méthodes abstraites** qui sont obligatoirement publiques
    - ✓ Le modificateur **abstract** n'est pas nécessaire (implicite).
    - ✓ Le modificateur **public** n'est pas nécessaire (implicite).
  - Des **méthodes avec une implémentation par défaut**
    - ✓ Le modificateur **default** doit être utilisé.
  - Des **méthodes statiques**.
- Par contre
  - Une interface ne peut **pas** définir **de champs d'instance**.
  - Une interface ne définit **pas de constructeur** (on ne peut pas l'instancier et elle n'intervient pas dans le chaînage des constructeurs).

## Utilisation des interfaces [1]

- En Java, une classe ne peut hériter que d'une et d'une seule classe parente (héritage simple).
  - Une classe peut, par contre, **implémenter une ou plusieurs interfaces** en utilisant la syntaxe suivante :

**implements** *interface1*, *interface2*, ...

- L'implémentation d'une ou de plusieurs interfaces (`implements`) peut être combinée avec l'héritage simple (`extends`). La clause `implements` doit suivre la clause `extends`.
  - Exemples :

```
public class Report implements Printable {...}
```

```
public class Book    implements Printable, Zoomable {...}
```

```
public class Circle extends Shape implements Printable {...}
```

```
public class Square extends Rectangle
```

```
implements Printable, Zoomable {...}
```

## Utilisation des interfaces [2]

- Lorsqu'une classe déclare une interface dans sa clause **implements**, elle indique ainsi qu'elle s'engage à fournir une implémentation (c'est-à-dire un corps) pour chacune des méthodes abstraites de cette interface.
- Si une classe implémente une interface mais ne fournit pas d'implémentation pour toutes les méthodes abstraites de l'interface, elle hérite des méthodes (abstraites) non implémentées de l'interface et doit elle-même être déclarée **abstract**.
- Si une classe implémente plus d'une interface, elle doit implémenter toutes les méthodes abstraites de chacune des interfaces mentionnées dans la clause **implements** (ou alors être déclarée **abstract**).
- Les méthodes de l'interface qui sont déclarées avec une implémentation par défaut peuvent être redéfinies (ou non) dans les classes qui implémentent l'interface.

## Exemple d'interface [1]

- Si l'on souhaite caractériser la **fonctionnalité** de comparaison qui est commune à tous les objets qui ont une relation d'ordre (plus petit, égal, plus grand), on peut définir l'interface **Comparable** de la manière suivante :

```
public interface Comparable {  
    public int compareTo(Object v);  
}
```

- Cette interface définit un **type de données** qui peut ensuite être utilisé dans des traitements qui nécessitent une relation d'ordre, dans une méthode de tri par exemple :

```
public class Tools {  
    ...  
    public static void sort(Comparable[] t) {  
        ...  
        if (t[i].compareTo(t[j]) > 0) ...  
        ...  
    }  
    ...  
}
```

## Exemple d'interface [2]

- Tous les objets dont la classe implémente l'interface Comparable pourront ensuite être triés.

```
public class Person implements Comparable {  
    . . .  
    private int age;  
    . . .  
    public int compareTo(Object p) {  
        if (p instanceof Person) {  
            if (this.age < ((Person)p).age) return -1;  
            if (this.age > ((Person)p).age) return 1;  
            return 0;  
        }  
        else . . .  
    }  
    . . .  
}
```

```
    . . .  
    Person[] population;  
    . . .  
    Tools.sort(population);  
    . . .
```

## Exemple d'interface [3]

- Supposons que nous voulions que les classes dérivées de la classe `Shape` disposent toutes d'une méthode `print()` permettant d'imprimer les formes géométriques.
- Il serait naturellement possible d'ajouter une méthode abstraite `print()` à la classe `Shape` et ainsi chacune des sous-classes concrètes devrait implémenter cette méthode.  
L'utilisateur de ces sous-classes serait ainsi sûr de disposer d'une méthode `print()` lui permettant d'imprimer la forme.
- Avec cette solution, le problème serait résolu pour toutes les sous-classes de `Shape`, mais si d'autres classes (qui n'héritent pas de `Shape`) souhaitaient également disposer des fonctions d'impression, elles devraient à nouveau déclarer des méthodes abstraites d'impression dans leur arborescence.
- L'utilisation d'une interface permet de résoudre ce problème de manière plus élégante en découplant la fonctionnalité d'impression de la hiérarchie des classes.

## Exemple d'interface [4]

- Au lieu d'ajouter une méthode abstraite `print()` dans la classe `Shape` nous créons une **interface** nommée `Printable` qui définit la méthode abstraite `print()` :

```
public interface Printable {  
    public void print();  
}
```

- Et nous déclarons que la classe `Shape` implémente l'interface `Printable` (le corps de la classe `Shape` reste inchangé) :

```
public abstract class Shape implements Printable {  
    public abstract double perimeter(); // Méthode abstraite  
    public abstract double area(); // Méthode abstraite  
}
```

## Exemple d'interface [5]

- Chaque sous-classe concrète de `Shape` devra obligatoirement implémenter le corps de la méthode `print()` (en plus des méthodes abstraites `perimeter()` et `area()`) :

```
-----  
// Circle  
-----  
public class Circle extends Shape {  
    public static final double PI = 3.14159265358979;  
    private double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    public double getRadius() { return radius; }  
    public double perimeter() { return 2*PI*radius; }  
    public double area() { return PI*radius*radius; }  
    public void print() { ... }  
}
```

## Exemple d'interface [6]

- L'implémentation des sous-classes de `Shape` s'effectue donc de la même manière que si la méthode `print()` avait été déclarée comme méthode abstraite dans `Shape`.
- L'avantage de l'interface `Printable` réside dans le fait qu'elle peut être utilisée avec d'autres classes qui n'ont rien à voir avec la classe `Shape` :

```
public class Book extends Document implements Printable {  
    ...  
}
```

- Un autre avantage est que l'interface `Printable` peut être utilisée en combinaison avec d'autres interfaces, ce qui n'est pas possible avec les classes (car une sous-classe ne peut pas avoir plusieurs super-classes).

```
public class Book extends Document  
    implements Printable, Zoomable {  
    ...  
}
```

# Interface fonctionnelle

- Une **interface fonctionnelle** est une interface comprenant exactement une méthode abstraite.
- Une interface fonctionnelle peut malgré tout comporter des constantes, des méthodes avec une implémentation par défaut, des méthodes redéclarant certaines méthodes de la classe `Object` ainsi que des méthodes statiques.
- La notion d'interface fonctionnelle joue un rôle important en lien avec l'utilisation des *expressions Lambda*.
- L'annotation `@FunctionalInterface` peut précéder la déclaration des interfaces fonctionnelles. Ainsi, le compilateur vérifiera que les règles énoncées soient respectées.

```
@FunctionalInterface
public interface SimpleFuncInterface {
    public void doWork();
}
```

# Classe abstraite ou interface ?

---

- Lors de la conception d'une application ou d'une librairie, le choix conceptuel entre une classe abstraite et une interface n'est pas toujours facile.
- Parmi les points à prendre en considération, on peut rappeler que :
  - Une classe peut implémenter plusieurs interfaces mais ne peut hériter que d'une seule classe abstraite.
  - Une interface peut être implémentée par une classe sans qu'il y ait de rapports étroits entre les deux. Une sous-classe est liée par une relation plus forte ("*Est un ...*").
  - Les classes abstraites et les interfaces permettent de définir des implémentation par défaut pour les méthodes.
  - Les interfaces fonctionnelles (une seule méthode abstraite) peuvent être implémentées par des expressions *Lambdas* ou des références de méthodes permettant ainsi d'alléger le code.

# Extension des interfaces

- Les interfaces peuvent avoir des **sous-interfaces** (tout comme les classes peuvent avoir des sous-classes). Comme pour les classes, le mot-clé **extends** est utilisé pour créer une sous-interface.

```
public interface Zoomable extends Observable {...}
```

- Une sous-interface hérite de toutes les méthodes et de toutes les constantes de son interface parente et peut définir de nouvelles méthodes ainsi que de nouvelles constantes.
- Contrairement aux classes, une interface peut posséder plusieurs interfaces parentes (héritage multiple).

```
public interface Transformable  
    extends Scalable, Translatable, Rotatable {...}
```

- Une classe qui implémente une sous-interface doit implémenter les méthodes abstraites définies directement par l'interface ainsi que les méthodes abstraites héritées de toutes les interfaces parentes de la sous-interface.

# Gestion des exceptions

- Les méthodes concrètes ne peuvent pas annoncer plus d'exceptions contrôlées dans leurs clauses **throws**, que celles qui sont annoncées dans la déclaration des méthodes abstraites qu'elles implémentent (déclaration dans classe abstraite ou interface).

```
public interface Writable {  
    public void write(Object o) throws IOException, MemOverflow;  
}
```

```
public class Buffer implements Writable {  
    public void write(Object o) throws MemOverflow,  
                                ArithmeticException {  
        ...  
    }  
}
```

La méthode concrète `write()` n'a pas l'obligation d'annoncer toutes les exceptions définies dans l'interface **Writable** (on ne peut pas imposer une liste d'exceptions).

Elle ne peut pas annoncer d'exceptions contrôlées supplémentaires mais peut ajouter des exceptions non-contrôlées (comme `ArithmeticException` par exemple).

# Conflits de noms [1]

- Des **conflits de noms** (collision) peuvent se produire lorsqu'une classe implémente plusieurs interfaces comportant des noms de méthodes ou de constantes identiques.
- Il faut distinguer plusieurs situations :
  - Plusieurs méthodes portent des signatures identiques :
    - ✓ Pas de problème, la classe doit implémenter cette méthode
  - Mêmes noms de méthodes mais profils de paramètres différents :
    - ✓ Implémentation de deux ou plusieurs méthodes surchargées
  - Mêmes noms de méthodes, profils de paramètres identiques, mais types des valeurs de retour différents :
    - ✓ Pas possible d'implémenter les deux interfaces (cela provoquera une erreur à la compilation : *Interface-Collision*)
  - Seulement les listes d'exceptions sont différentes :
    - ✓ La méthode ne peut pas générer plus d'exceptions contrôlées que celles correspondant à l'intersection de l'ensemble des clauses d'exception

## Conflits de noms [2]

- Noms de constantes identiques dans plusieurs interfaces :
  - ✓ Doivent être accédées en utilisant la notation qualifiée (*Interface.Constante*)
- Plusieurs méthodes portent des signatures identiques mais ont des implémentations par défaut différentes :
  - ✓ Erreur détectée par le compilateur
  - ✓ Obligation de redéfinir la méthode en conflit dans la classe qui implémente les interfaces
  - ✓ Les méthodes par défaut peuvent malgré tout être invoquée avec la syntaxe *Interface.super.Méthode*

# Interface de marquage

- Il est parfois utile de définir une interface entièrement vide appelée **interface de marquage**.
- Une classe peut implémenter cette interface en la nommant dans la clause **implements** sans avoir à implémenter de méthodes.
- Dans ce cas, toutes les instances de la classe deviennent des instances valables de l'interface. Il est donc possible de tester si un objet implémente l'interface de marquage à l'aide de l'opérateur **instanceof** :

```
if (obj instanceof Cloneable) {...}
```

- Une interface de marquage sert à **communiquer** aux instances qui l'implémentent **des informations supplémentaires** concernant l'objet, son comportement, son implémentation, ...
- Les interfaces **java.lang.Cloneable** et **java.io.Serializable** constituent deux exemples d'interfaces de marquage de la plate-forme Java.



**FIN DE LA PARTIE II**