Chapitre 3:
Réutilisations des Classes
Héritage

Classes et Objets

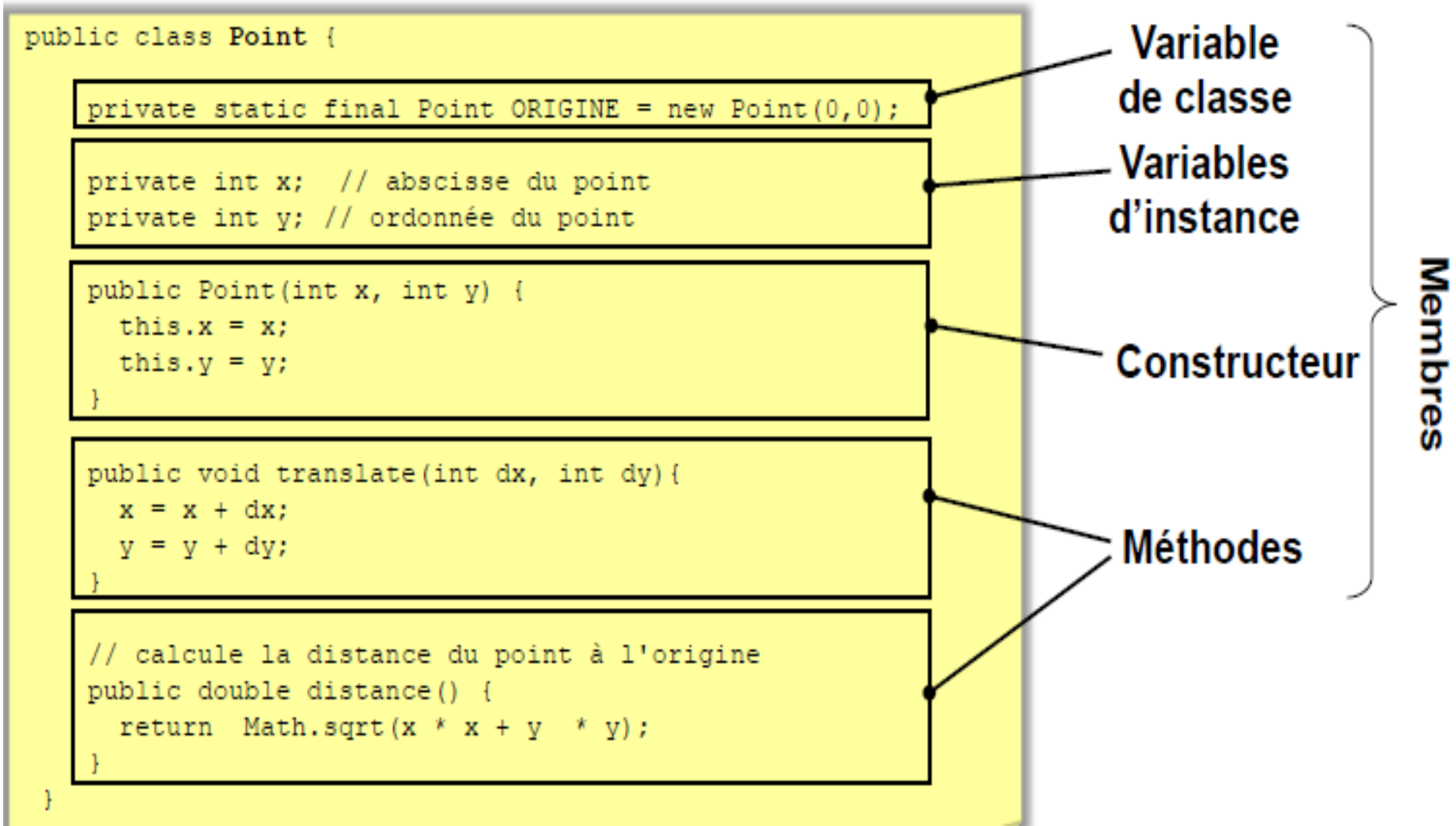
Rappel

- Qu'est ce qu'une classe en POO?
  - Cela correspond à un plan, un moule, une usine…
  - C'est une description abstraite d'un type d'objets
  - Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)
- Qu'est ce que la notion d'instance?
  - Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
  - L'instanciation : création d'un objet à partir d'une classe
  - Objet = instance

Classes et Objets

Rappel

De quoi est composé une classe?



Réutilisation

- Comment utiliser une classe comme brique de base pour concevoir d'autres classes ?

- Dans une conception objet on définit des associations (relations) entre classes pour exprimer la réutilisation.

- UML (Unified Modelling Language) définit toute une typologie des associations possibles entre classes. Nous focaliserons sur deux formes d'association

  - *Un objet peut faire appel à un autre objet :* ***délégation***
  - *Un objet peut être créé à partir du «moule» d'un autre objet :* ***héritage***

Délégation

- Un objet **o1** instance de la classe **C1** utilise les services d'un objet **o2** instance de la classe **C2** (**o1** délègue une partie de son activité à **o2**)

- La classe **C1** utilise les services de la classe **C2**
  - **C1** est la classe cliente
  - **C2** est la classe serveuse

- La classe cliente (**C1**) possède une référence de type de la classe serveuse (**C2**)

Délégation

Notation **UML**

Association
à navigabilité restreinte

La flèche indique que les instances de C1 connaissent les instances de C2 mais pas l'inverse

Traduction en **java**

```
public class C1 {

   private C2 refC2;

   ...

}
```

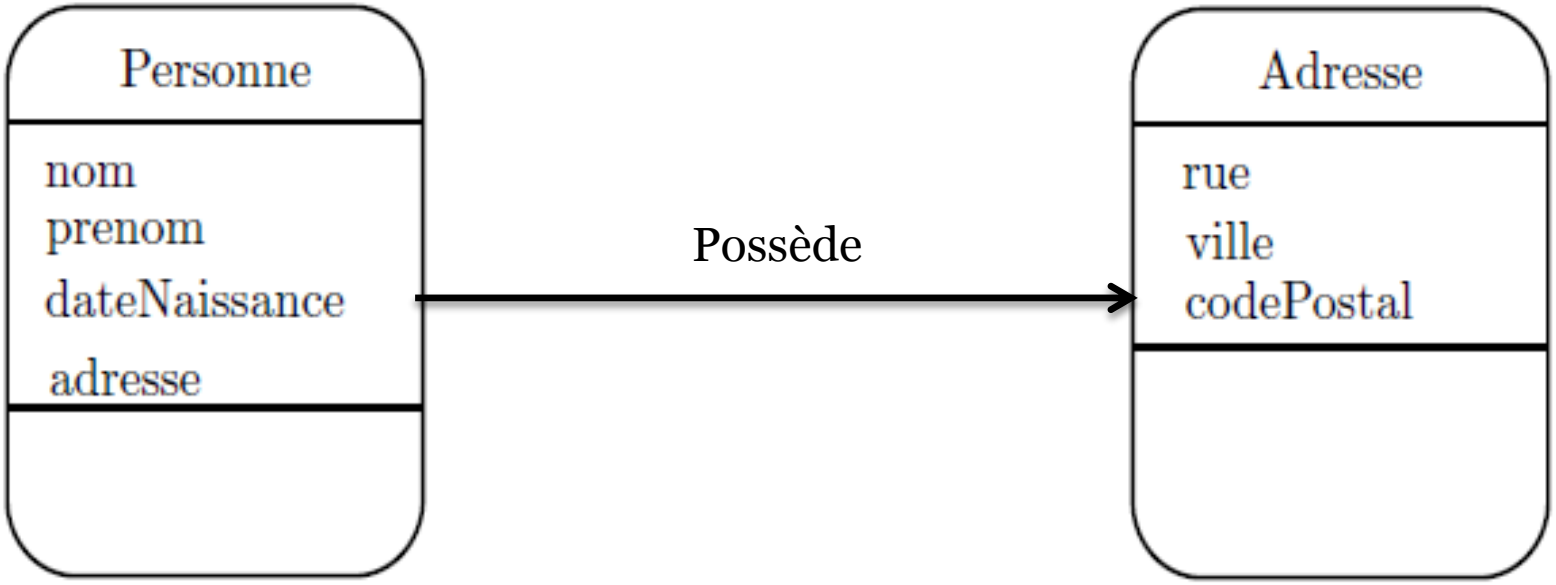
```
public class C2 {

   ...

}
```

Délégation

Exemple 1

- Chaque personne possède une adresse
- Donc, on va ajouter dans la classe Personne un attribut adresse qui est un objet de la classe Adresse

Relation entre classes : Association

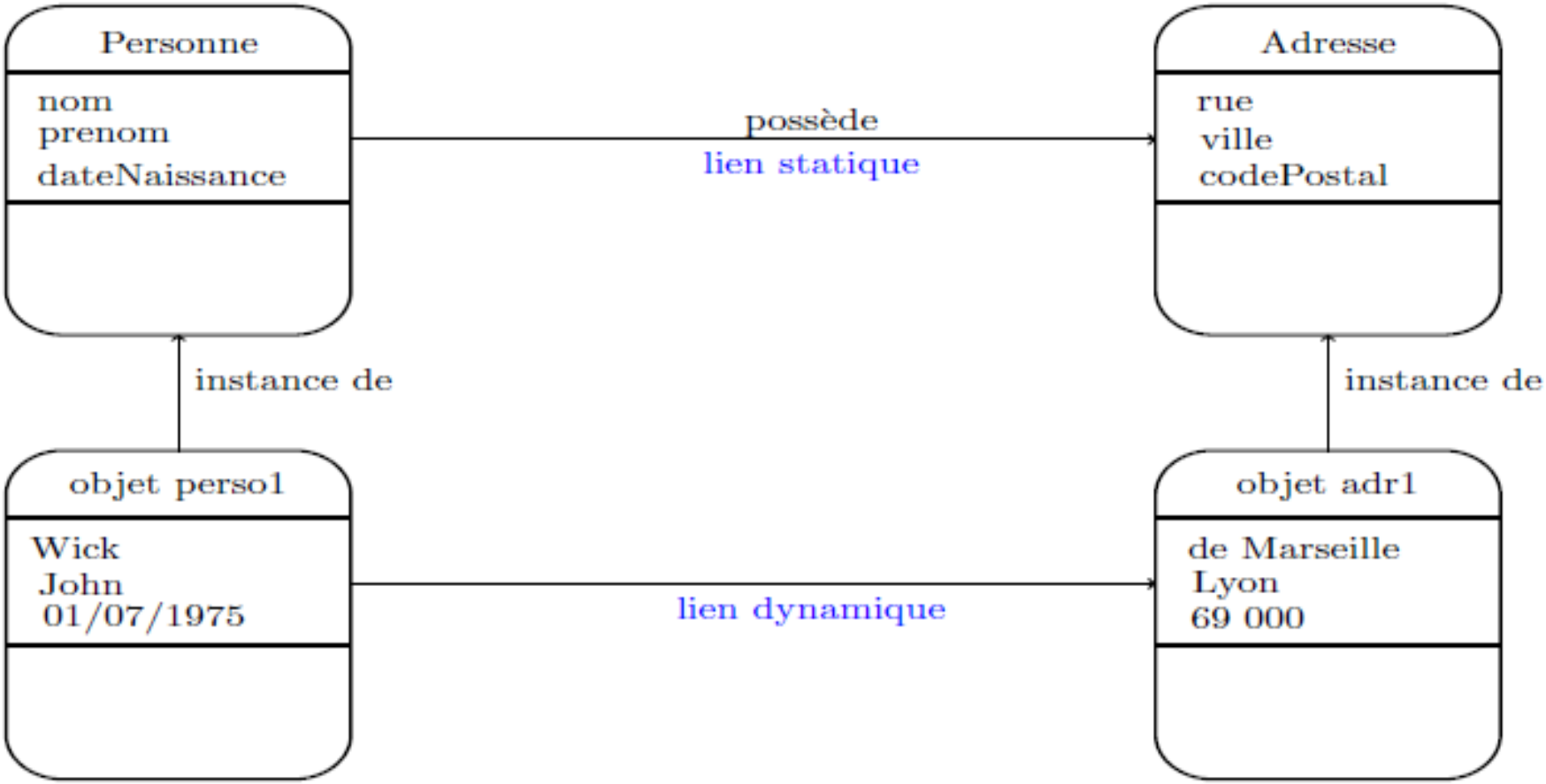


Délégation

Exemple 1

Relation entre objets

- est établie dynamiquement à partir de la relation entre leur deux classes



Délégation

Agrégation / Composition

Les deux exemples précédents traduisent deux nuances (sémantiques) de l'association a-un entre la classe **Cercle** et la classe **Point**

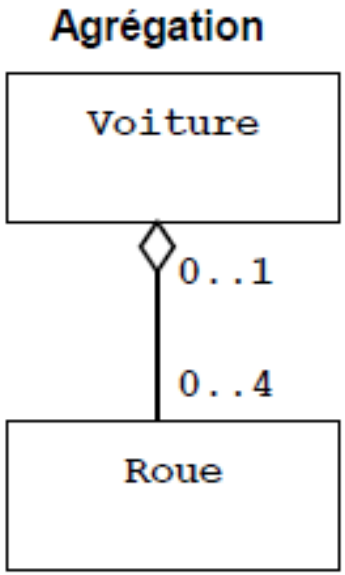
UML distingue ces deux sémantiques en définissant deux type de relations :

Délégation

Agrégation

L'agrégation

- C'est une association non-symétrique
- Elle représente une relation de type ensemble/élément



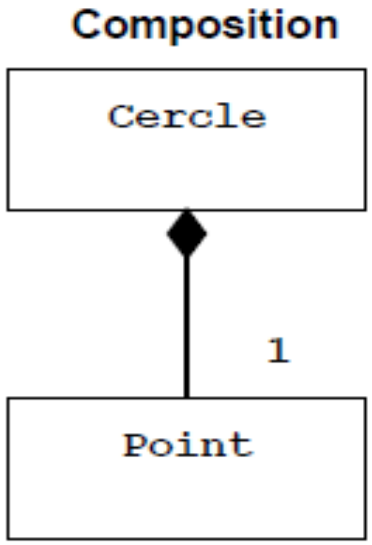
L'élément agrégé (Roue) a une existence autonome
en dehors de l'agrégat (Voiture)

Délégation

Composition

La composition

- C'est une agrégation forte
- L'objet composite n'existe pas sans l'objet composant



A un même moment, une instance de composant (Point)
ne peut être liée qu'à un seul agrégat (Cercle),
et le composant a un cycle de vie dépendant de l'agrégat.

Héritage

Pourquoi?

L'héritage, quand?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une **Classe1** est une sorte de **Classe2**

Héritage

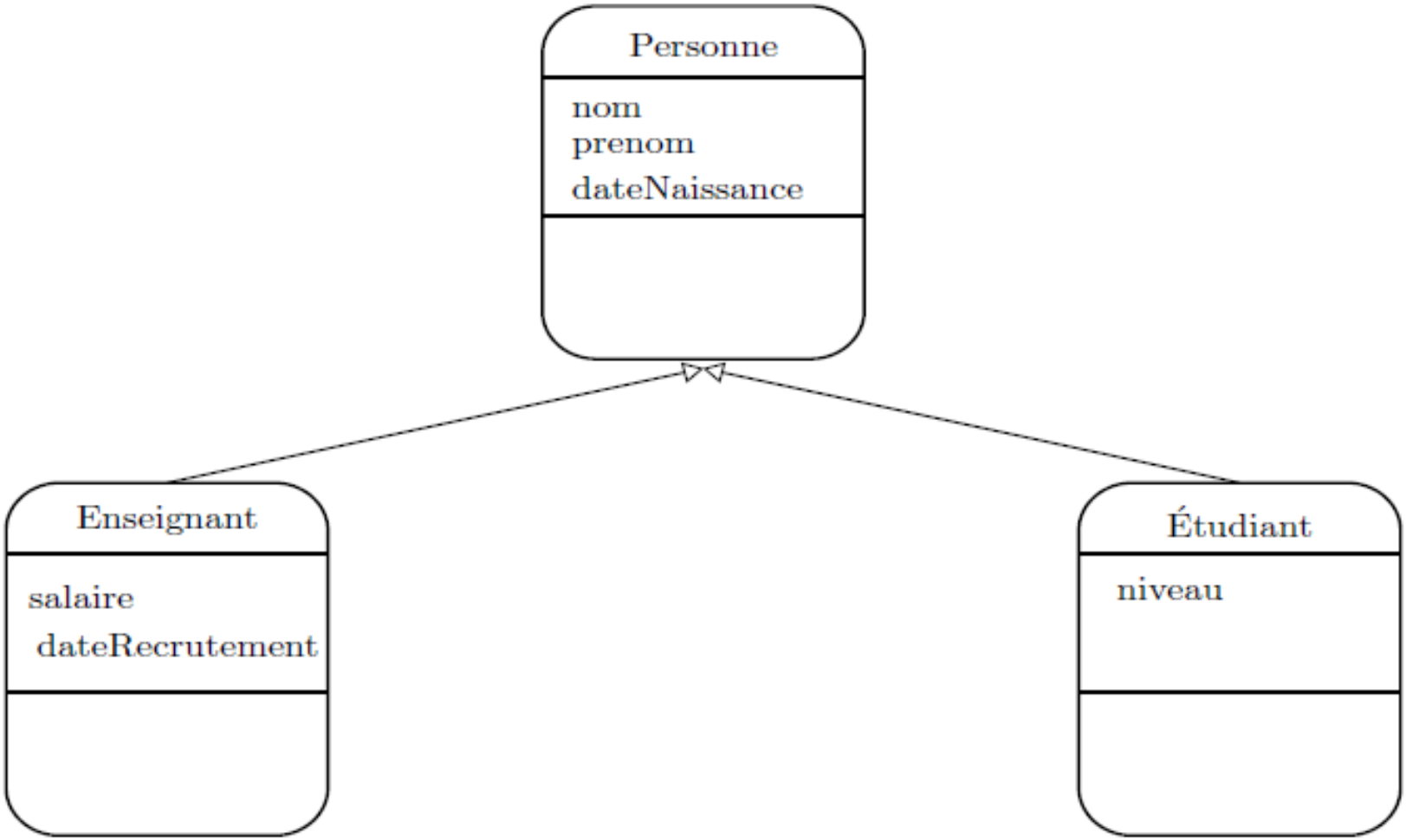
Pourquoi?

Exemple

- Un enseignant a un nom, un prénom, une date de naissance, un salaire et une date de recrutement
- Un étudiant a aussi un nom, un prénom, une date de naissance et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que nom, prénom et date de naissance
- Donc, on peut mettre en commun les attributs nom, prénom, date de naissance dans une classe `Personne`
- Les classes `Étudiant` et `Enseignant` hériteront de la classe `Personne`

Héritage

Pourquoi?



Héritage

Exemple

- Le problème
  - *une application a besoin de services dont une partie seulement est proposée par une classe déjà définie (classe dont on ne possède pas nécessairement le source)*
  - *ne pas réécrire le code*

Un `Point`
- a une position
- peut être déplacé
- peut calculer sa distance à l 'origine
- ...

Application a besoin
- de manipuler des points (comme le permet la classe `Point`)
- mais en plus de les dessiner sur l'écran.

`PointGraphique` = `Point`
+ une couleur
+ une opération d'affichage

- Solution en POO : l'héritage (inheritence)
  - *définir une nouvelle classe à partir de la classe déjà existante*

Héritage

Exemple : syntaxe Java

- La classe **PointGraphique** hérite de la classe **Point**

PointGraphique.java

```
import java.awt.Color;
import java.awt.Graphics;
public class PointGraphique extends Point {

    Color coul;

    // constructeur
    public PointGraphique(double x, double y,
                          Color c) {
       this.x = x;
       this.y = y;
       this.coul = c;
    }

    // affiche le point matérialisé par
    // un rectangle de 3 pixels de coté
    public void dessine(Graphics g) {
       g.setColor(coul);
       g.fillRect((int) x - 1,(int)y - 1,3,3);
    }
}
```

PointGraphique hérite de (étend) **Point**
un PointGraphique possède les variables et méthodes définies dans la classe **Point**

PointGraphique définit un nouvel attribut

Attributs hérités de la classe Point

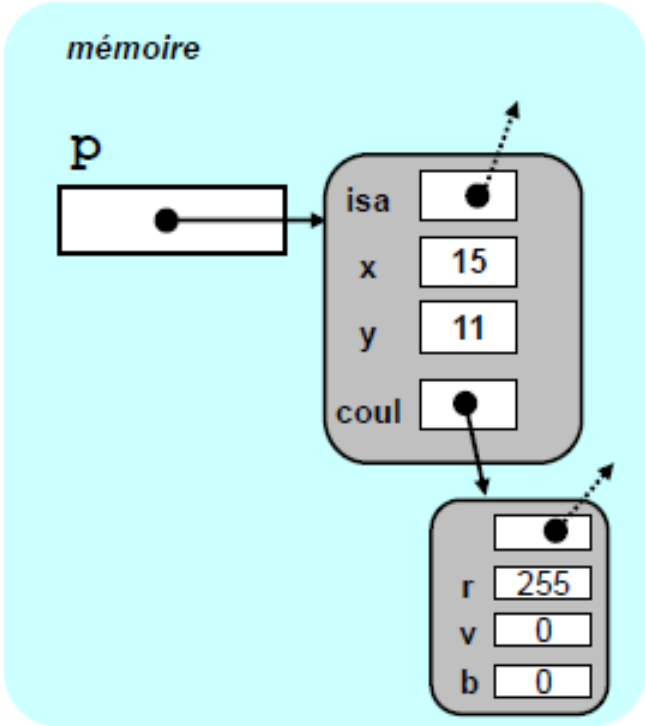
PointGraphique définit une nouvelle méthode

Héritage

Utilisation des instances d'une classe héritée

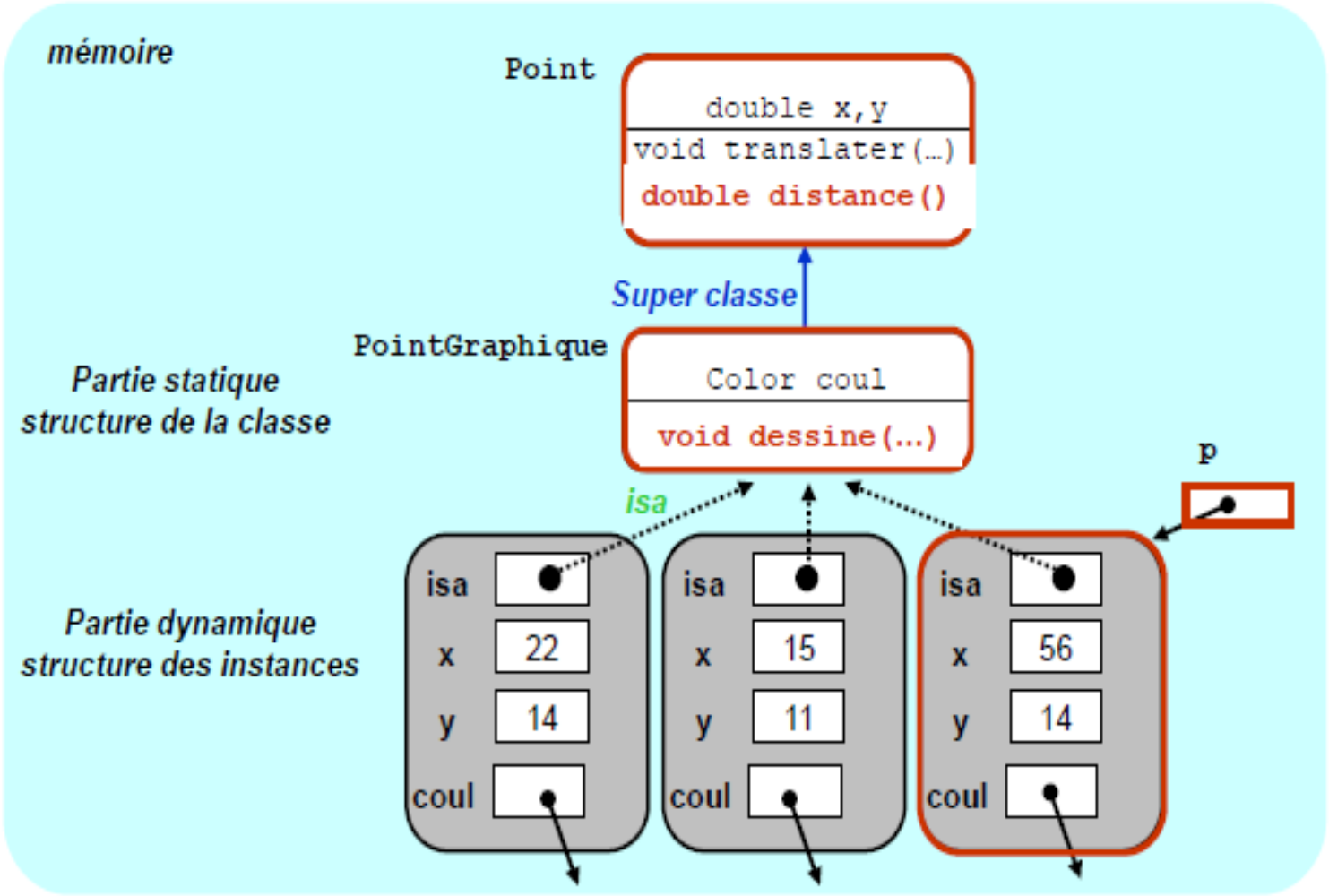
- *Un objet instance de PointGraphique possède les attributs définis dans* **`PointGraphique`** *ainsi que les attributs définis dans* **`Point`** *(un* **`PointGraphique`** *est aussi un* **`Point`** *)*
- *Un objet instance de* **`PointGraphique`** *répond aux messages définis par les méthodes décrites dans la classe* **`PointGraphique`** *et aussi à ceux définis par les méthodes de la classe* **`Point`**

```
PointGraphique p = new PointGraphique();

// utilisation des variables d'instance héritées
p.x = 15;
p.y = 11;

// utilisation d'une variable d'instance spécifique
p.coul = new Color(255,0,0);

// utilisation d'une méthode héritée
double dist = p.distance();

// utilisation d'une méthode spécifique
p.dessine(graphicContext);
```



Héritage

Résolution des messages



Héritage

Terminologie

- Héritage permet de reprendre les caractéristiques d'une classe **M** existante pour les étendre et définir ainsi une nouvelle classe **F** qui hérite de **M**.
- Les objets de F possèdent toutes les caractéristiques de **M** avec en en plus celles définies dans **F**
  - *Point est la classe mère et PointGraphique la classe fille.*
  - *la classe PointGraphique **hérite** de la classe Point*
  - *la classe PointGraphique est une sous-classe de la classe Point*
  - *la classe Point est la super-classe de la classe PointGraphique*

- la relation d'héritage peut être vue comme une relation de "généralisation/spécialisation" entre une classe (la *super-classe*) et plusieurs classes plus spécialisées (ses *sous-classes*).

Héritage

Propriétés

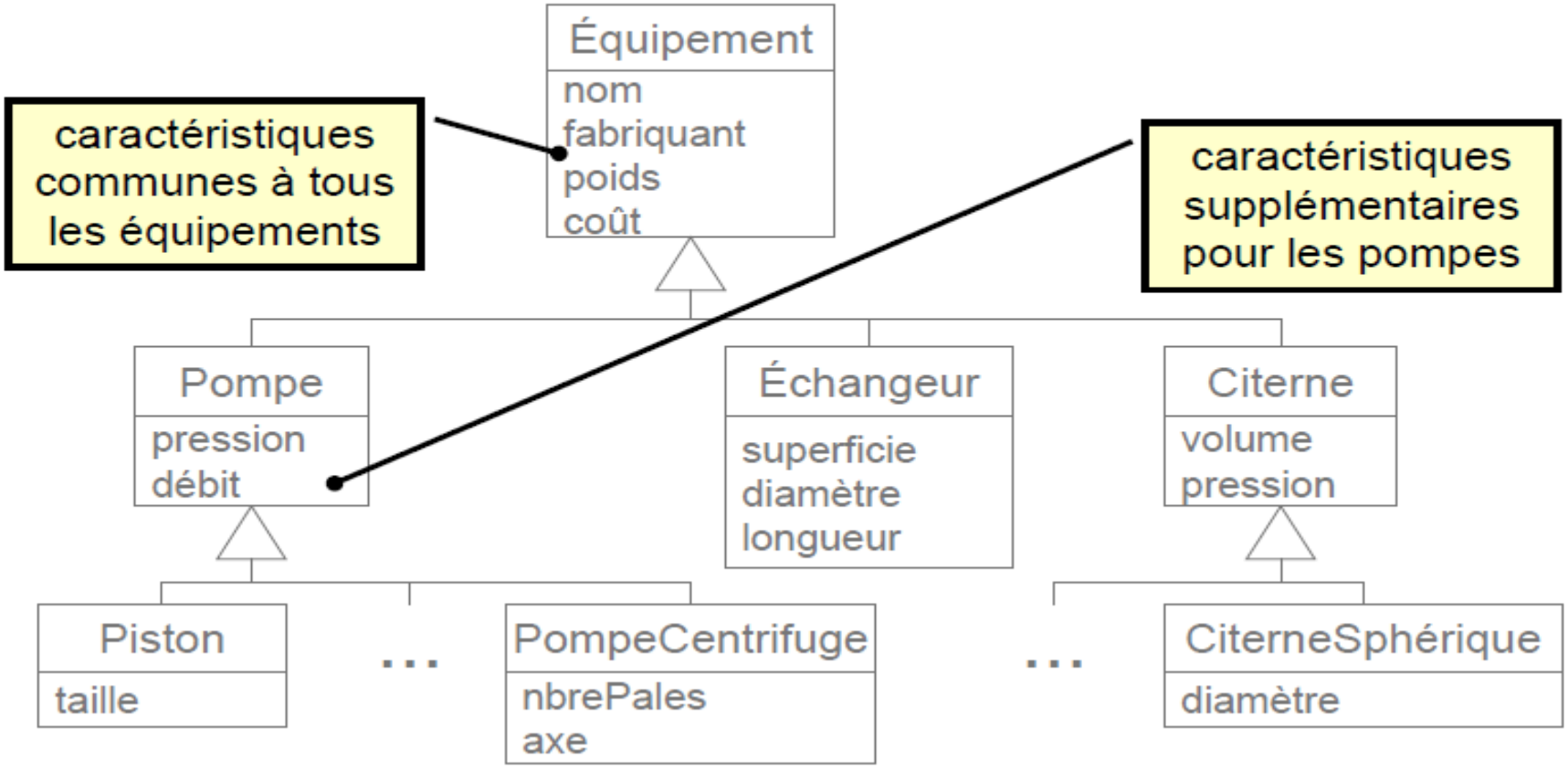
L'héritage : propriétés

- **Transitivité** : si A hérite de B et B hérite de C, alors A hérite de C
- **Non-réflexif** : une classe n'hérite pas d'elle même
- **Non-symétrique** : si A hérite de B, alors B n'hérite pas de A
- **Non-cyclique** : si A hérite de B et B hérite de C, alors C ne peut hériter de A

Héritage

Héritage à travers tous les niveaux

- pas de limitation dans le nombre de niveaux dans la hiérarchie d'héritage
- méthodes et variables sont héritées au travers de tous les niveaux



Héritage

Héritage à travers tous les niveaux

```
public class A {
  public void hello() {
    System.out.println(«Hello»);
  }
}
```

```
public class B extends A {
  public void bye() {
    System.out.println(«Bye Bye»);
  }
}
```

```
public class C extends B {
  public void oups() {
    System.out.println(«oups!»);
  }
}
```

```
C c = new C();
c.hello();
c.bye();
c.oups();
```

Pour résoudre un message, la hiérarchie des classes est parcourue de manière ascendante jusqu'à trouver la méthode correspondante.

Héritage

Redéfinition des méthodes

- Une sous-classe peut **ajouter** des variables et/ou des méthodes à celles qu'elle hérite de sa super-classe.
- une sous-classe peut **redéfinir(override**) les méthodes dont elle hérite et fournir ainsi des implémentations spécialisées pour celles-ci.
- **Redéfinition d'une méthode** (**method overriding**)
  - *lorsque la classe définit une méthode dont le nom, le type de retour et le type des arguments sont identiques à ceux d'une méthode dont elle hérite*
- Lorsqu'une méthode redéfinie par une classe est invoquée pour un objet de cette classe, c'est la nouvelle définition et non pas celle de la super-classe qui est invoquée.

Héritage

Redéfinition des méthodes

```
public class A {

  public void affiche() {
    System.out.println("Je suis un A");
  }

  public void hello() {
    System.out.println("Hello");
  }
}
```

```
public class B extends A {

  public void affiche() {
    System.out.println("Je suis un B");
  }
}
```

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

a.affiche();  → Je suis un A
a.hello();    → Hello

b.hello();    → Hello
b.affiche();  → Je suis un B
```

la méthode `affiche()` est redéfinie

c'est la méthode la plus spécifique qui est exécutée

Héritage

Redéfinition des méthodes

- Ne pas confondre **redéfinition** (***overriding***) avec **surcharge** (***overloading***)

```
public class A {

  public void methodX(int i) {
        ...
  }
}
```

Surcharge

```
public class B extends A {

  public void methodX(Color i) {
        ...
  }
}
```

B possède deux méthodes **methodX** (methodX(int) et methodX(Color))

Redéfinition

```
public class C extends A {

  public void methodX(int i) {
        ...
  }
}
```

C possède une seule méthode **methodX** (methodX(int))

Héritage

Redéfinition des méthodes

- **Annotations[1] (java 1.5)** : méta-données sur un programme. (Données qui ne font pas partie du programme lui-même)
  - *informations pour le compilateur (détection d'erreurs)*
  - *traitements à la compilation ou au déploiement (génération de code, de fichiers XML, ...)*
  - *traitement à l'exécution*

- Lors d'une redéfinition utiliser l'annotation `@Override`

- Evite de faire une surcharge alors que l'on veut faire une redéfinition

```
class A {

    protected double x;

    public void add(double x) {
        System.out.println("A.add double " + x);
        this.x += x;
    }
}
```

```
class C extends A {

    Add @Override Annotation

    public void add(double x) {
        System.out.println("C.add int " + x);
        this.x += x;
    }

}
```

```
class B extends A {

    method does not override or implement a method from a supertype

    @Override
    public void add(int x) {
        System.out.println("B.add int " + x);
        this.x += x;
    }
```

Héritage

Redéfinition avec réutilisation

- **Redéfinition** des méthodes (method **overriding**) :
  - *possibilité de réutiliser le code de la méthode héritée (**super**)*

this permet de faire référence à l'objet en cours
super permet de désigner la superclasse

```
public class Etudiant {
   String nom;
   String prénom;
   int age;
   ...
   public void affiche() {
     System.out.println("Nom : " + nom + " Prénom : " + prénom);
     System.out.println("Age : " + age);
     ...
   }
   ...
 }
```

```
public class EtudiantSportif extends Etudiant {
   String sportPratiqué;
   ...
   public void affiche(){
     super.affiche();

     System.out.println("Sport pratiqué : "+sportPratiqué);
     ...
   }
 }
```

l'appel super peut être effectué n'importe où dans le corps de la méthode

Héritage

Particularités de l'héritage en Java

- **Héritage simple**
  - une classe ne peut hériter que d'une seule autre classe
    - dans certains autres langages (ex C++) possibilité d'héritage multiple

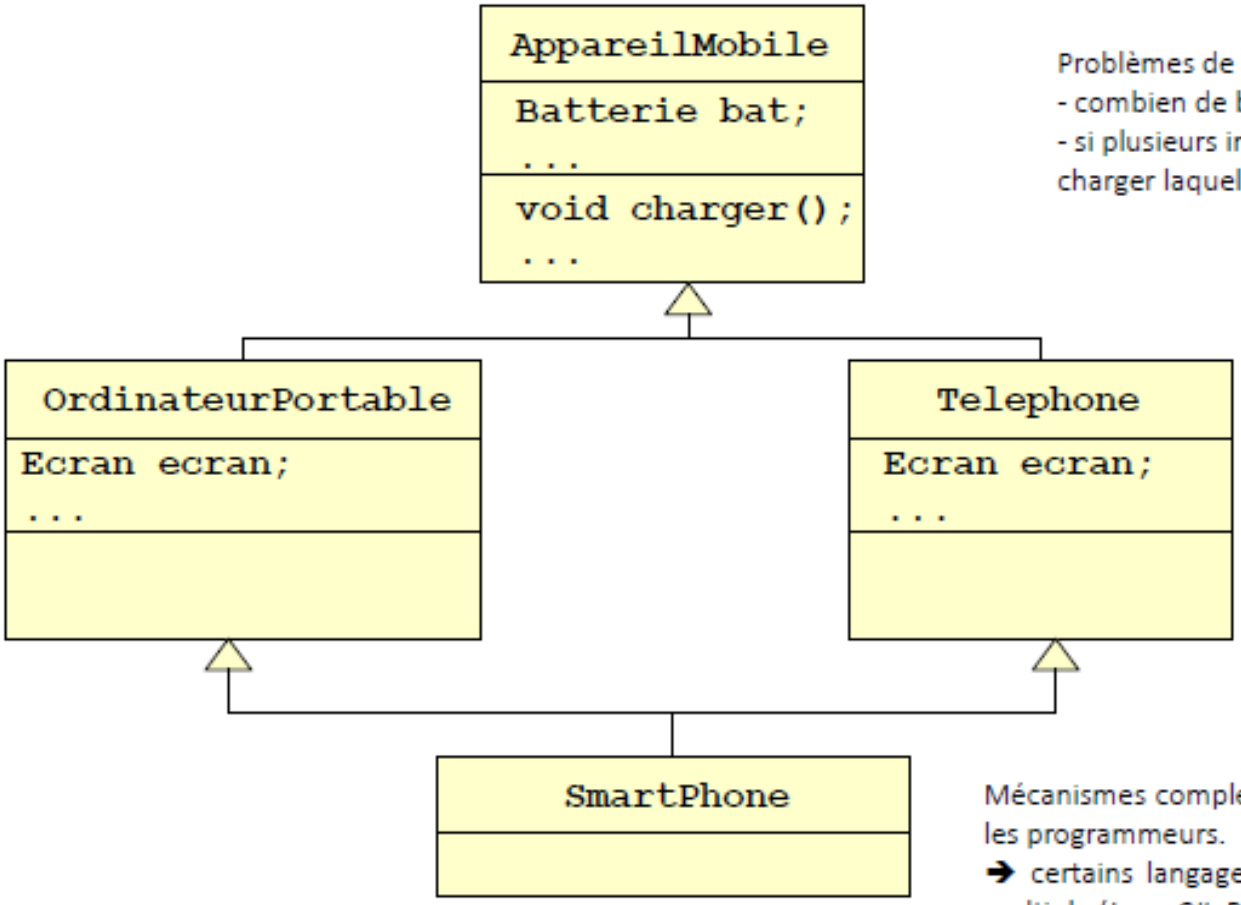
- La hiérarchie d'héritage **est un arbre** dont la racine est la classe **`Object`** (java.lang)
  - toute classe autre que **`Object`** possède une super-classe
  - toute classe hérite directement ou indirectement de la classe **`Object`**
  - par défaut une classe qui ne définit pas de clause **`extends`** hérite de la classe **`Object`**

```
public class Point extends Object {

    int x;  // abscisse du point
    int y; // ordonnée du point

    ...
 }
```

Héritage

Problèmes de l'héritage multiple

- Héritage multiple
  - une classe peut hériter de plus d'une seule classe



Problèmes de l'héritage en diamant
- combien de batteries possède un smartphone ?
- si plusieurs implémentation de la méthode charger laquelle est invoquée ?

Les langages de programmation peuvent résoudre ce problème de façons différentes :
- Une solution consiste à rajouter un mécanisme dans le langage pour choisir entre la fusion des entités répétées ou le renommage de celles-ci afin de séparer les entités. (Eiffel, Ocaml).
- Héritage virtuel dans C++
- ...

Mécanismes complexes pas toujours très bien compris et utilisés par les programmeurs.

➔ certains langages on fait le choix de ne pas proposer l'héritage multiple (Java, C#, Ruby, ObjectiveC)

Héritage

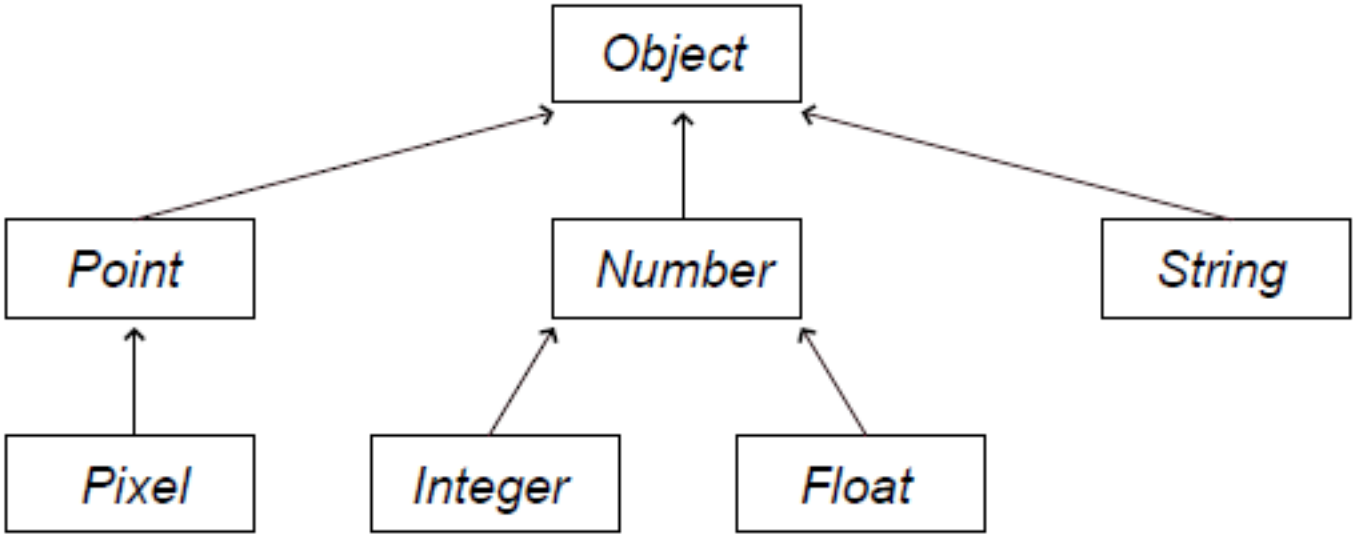
La classe Object

Il existe une classe, nommée `Object`, qui n'a pas de super-classe

Certaines classes ont une super-classe explicite

Les classes sans super-classe explicite ont `Object` pour super-classe

L'ensemble des classes est organisé en une arborescence de racine `Object`



Héritage

La classe Object

- Principales méthodes de la classe **Object**
  - `public final Class getClass()`

    Renvoie la référence de l'objet Java représentant la classe de l'objet
  - `public boolean equals(Object obj)`

    Teste l'égalité de l'objet avec l'objet passé en paramètre

    `return (this == obj);` *(on en reparlera lors du cours sur le polymorphisme)*
  - `protected Object clone()`

    Crée un copie de l'objet
  - `public int hashCode()`

    Renvoie une clé de hashcode pour adressage dispersé

    (on en reparlera lors du cours sur les collections)
  - `public String toString()`

    Renvoie une chaîne représentant la valeur de l'objet

    ```
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
    ```

Héritage

La classe Object

Opérateur de concaténation

<Expression de type String> **+** *<reference>*

<=>

<Expression de type String> **+** *<reference>***.toString()**

du fait que la méthode **toString** est définie dans la classe **Object** , on est sûr que quel que soit le type (la classe) de l'objet il saura répondre au message **toString()**

```
public String toString(){
  return getClass().getName() + "@" +
      Integer.toHexString(hashCode());
}
```

```
public class Object {
   ...
}
```

```
public class Point {
  private double x;
  private double y;

  ...
}
```

```
public String toString(){
   return "Point:[" + x +
                "," + y + "]");
}
```

La classe **Point** ne redéfinit pas **toString**

Point@2a340e

Point p = new Point(15,11);
System.out.println(p);

La classe **Point** redéfinit **toString**

Point:[15.0,11.0]

Héritage

Operateur instanceof

unObjet `instanceof` *uneClasse*

signifie : *unObjet* est-il instance de [une sous-classe de] *uneClasse* ?

c'est-à-dire : *unObjet* est-il *une sorte de uneClasse* ?

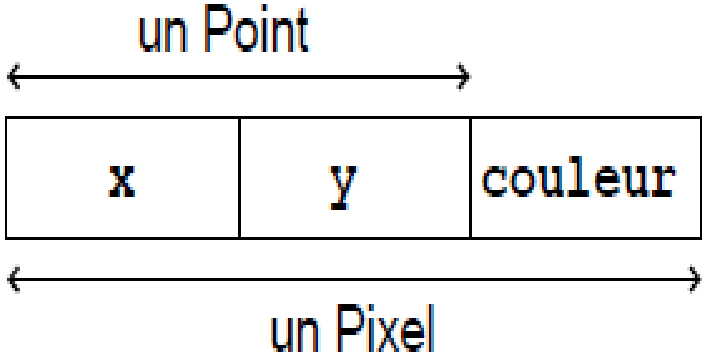
application : redéfinition de la méthode `equals`

```
class Point {
   private int x, y;
   ...
   public boolean equals(Object p) {
      return p instanceof Point &&
         ((Point)p).x == x && ((Point)p).y == y;
   }
}
```

La première condition justifie les deux changements de type

Héritage

Réutilisation des constructeurs



la construction d'une instance de la sous-classe *commence* par la construction de sa partie héritée

en clair : qu'on le veuille ou non, pour initialiser un `Pixel` il faut *commencer* par l'initialiser en tant que `Point`

si on ne fait rien, javac insère *au début* de chaque constructeur de la sous-classe un appel du constructeur sans argument de la super-classe

Héritage

Réutilisation des constructeurs

Constructeur problématique :

erroné

```
class Pixel extends Point {
    Color couleur;
    Pixel(int a, int b, Color c) {
        ici se cache un appel implicite de Point()
        x = a;
        y = b;
        couleur = c;
    }
    ...
}
```

même si `Point()` existe et `x` et `y` sont accessibles, il est maladroit de les initialiser pour rien, puisque tout de suite après ont leur affecte d'autres valeurs

Héritage

Réutilisation des constructeurs

La solution :

```
class Pixel extends Point {
    Color couleur;
    Pixel(int a, int b, Color c) {
        super(a, b);
        couleur = c;
    }
    ...
}
```

cela se lit : « pour initialiser un Pixel avec a, b et c, commencez par l'initialiser *en tant que* Point avec a et b, puis donnez à couleur la valeur c »

l'expression super(...); doit être la *première instruction* d'un constructeur

Héritage

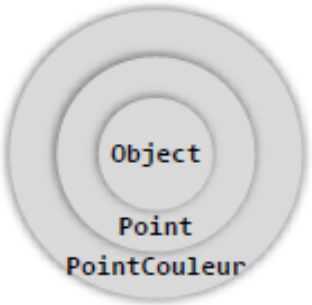
Réutilisation des constructeurs

```
public class Object {

    public Object()
    {
        ...
    }
    ...
}
```

```
public class Point extends Object {
    double x,y;

    public Point(double x, double y)
    {
        super(); // appel implicite
        this.x = x; this.y = y;
    }
    ...
}
```

```
public class PointCouleur extends Point {

    Color c;

    public PointCouleur(double x, double y, Color c)
    {
        super(x,y);
        this.c = c;
    }
    ...
}
```

Chaînage des constructeurs




```
new PointCouleur(…);
```

Héritage

Réutilisation des constructeurs

Constructeur par défaut

- Lorsqu'une classe ne définit pas explicitement de constructeur, elle possède un constructeur par défaut :
  - *sans paramètres*
  - *de corps vide*
  - *inexistant si un autre constructeur existe*

```
public class Object {

    public Object()
    {
        ...
    }
    ...
}
```

```
public class A extends Object {

    // attributs
    String nom;

    // méthodes
    String getNom() {
      return nom;
    }
    ...
}
```

```
public A() {
  super();
}
```

Constructeur par défaut implicite

Garantit chaînage des constructeurs

Héritage

Réutilisation des constructeurs

Constructeur par défaut

```
public class ClasseA {
    double x;
    // constructeur
    public ClasseA(double x) {
            this.x = x;
    }
 }
```

Constructeur explicite
masque constructeur par défaut

Pas de constructeur sans paramètres

```
public class ClasseB extends ClasseA {
    double y = 0;

    // pas de constructeur
}
```

```
public ClasseB(){
   super();
}
```

Constructeur par défaut implicite

```
C:>javac ClasseB.java
ClasseB.java:3: No constructor matching ClasseA() found in class ClasseA.
   public ClasseB() {
          ^
1 error
```

Héritage

Redéfinition des attributs

Lorsqu'une sous classe définit une variable d'instance dont le nom est identique à l'une des variables dont elle hérite, **la nouvelle définition masque la définition héritée**

```
class Article {
    int code = 111;
    ...
}
class Alimentation extends Article {
    int code = 222;
    ...
}
class RayonFrais extends Alimentation {
    int code = 333;
    ...
    void test() {
        System.out.println( this.code );              // ceci écrit 333
        System.out.println( super.code );             // ceci écrit 222
        System.out.println( ((Article) this).code );  // ceci écrit 111
    }
}
```

en général ce n'est pas une très bonne idée de masquer les variables

Héritage

Redéfinition des méthodes

```
class Point {
    private int x, y;
    ...
    public String toString() {
        return "(" + x + "," + y + ")";
    }
    ...
}
```

Un point sous forme de chaîne : "(10,20)"

Erreur :

x et y sont privés.

```
class Pixel extends Point {
    private Color couleur;
    ...
    public String toString() {
        return "(" + x + "," + y + ")-" + couleur;
    }
    ...
}
```

Un pixel sous forme de chaîne : "(10,20)-red"

Héritage

Redéfinition des méthodes

```
class Point {
    protected int x, y;
    ...
    public String toString() {
        return "(" + x + "," + y + ")";
    }
    ...
}
```

```
class Pixel extends Point {
    private Color couleur;
    ...
    public String toString() {
        return "(" + x + "," + y + ")-" + couleur;
    }
    ...
}
```

Un point sous forme de chaîne : "(10,20)"

Cela passe, mais c'est mal conçu : la classe Pixel s'appuie sur des détails internes de la classe Point

Un pixel sous forme de chaîne : "(10,20)-red"

Héritage

Redéfinition des méthodes

```
class Point {
    private int x, y;
    ...
    public String toString() {
        return "(" + x + "," + y + ")";
    }
    ...
}
```

Un point sous forme de chaîne : "(10,20)"

La bonne solution

```
class Pixel extends Point {
    private Color couleur;
    ...
    public String toString() {
        return super.toString() + "-" + couleur;
    }
    ...
}
```

Un pixel sous forme de chaîne : "(10,20)-red"

Encapsulation

Visibilité des variables et méthodes

- principe **d'encapsulation** : les données propres à un objet ne sont accessibles qu'au travers des méthodes de cet objet
  - *sécurité des données : elles ne sont accessibles qu'au travers de méthodes en lesquelles on peut avoir confiance*
  - *masquer l'implémentation : l'implémentation d'une classe peut être modifiée sans remettre en cause le code utilisant celle-ci*
- en JAVA possibilité de contrôler l'accessibilité (visibilité) des membres (variables et méthodes) d'une classe
  - **`public`** *accessible à toute autre classe*
  - **`private`** *n'est accessible qu'à l'intérieur de la classe où il est défini*
  - **`protected`** *est accessible dans la classe où il est défini, dans toutes ses sous-classes et dans toutes les classes du même package*
  - **-** *(visibilité par défaut* **`package`***) n'est accessible que dans les classes du même package que celui de la classe où il est défini*

Encapsulation

Visibilité des variables et méthodes

	`private`	- (package)	`protected`	`public`
La classe elle même	oui	oui	oui	oui
Classes du même package	non	oui	oui	oui
Sous-classes d'un autre package	non	non	oui	oui
Classes (non sous-classes) d'un autre package	non	non	non	oui

Encapsulation

Visibilité des variables et méthodes

PointGraphique.java

```
import java.awt.Color;
import java.awt.Graphics;
public class PointGraphique extends Point {

    Color coul;

    // constructeur
    public void PointGraphique(double x, double y,
                    Color c) {
       super(x,y);
       this.coul = c;
    }

    // affiche le point matérialisé par
    // un rectangle de 3 pixels de coté
    public void dessine(Graphics g) {
       g.setColor(coul);
       g.fillRect((int) x - 1,(int) y - 1,3,3);
    }
}
```

getX() gety()

```
public class Point {
  private double x;
  private double y

  ...

  public double getX() {
    return x;
  }

  ...
}
```

Attributs hérités de la classe Point

Les attributs sont privés dans la super-classe **on ne peut les utiliser directement** dans le code de la sous-classe

Encapsulation

Visibilité des variables et méthodes

PointGraphique.java

```
import java.awt.Color;
import java.awt.Graphics;
public class PointGraphique extends Point {

    Color coul;

    // constructeur
    public void PointGraphique(double x, double y,
                    Color c) {
       super(x,y);
       this.coul = c;
    }

    // affiche le point matérialisé par
    // un rectangle de 3 pixels de coté
    public void dessine(Graphics g) {
       g.setColor(coul);
       g.fillRect((int) x - 1,(int)y - 1,3,3);
    }
 }
```

```
public class Point {
  protected double x;
  protected double y

  ...

  public double getX() {
    return x;
  }

  ...
}
```

Attributs hérités de la classe Point

Les attributs sont protégés dans la super-classe **on peut les utiliser directement** dans le code de la sous-classe

Encapsulation

Visibilité des classes

- Deux niveaux de visibilité pour les classes :
  - ***public*** *: la classe peut être utilisée par n'importe quelle autre classe*
  - *- (**package**) : la classe ne peut être utilisée que par les classes appartenant au même package*

Package A

```
package A;
public class ClasseA {

    ClasseB b;


}
```

```
package A;
class ClasseB
     extends ClasseA {



}
```

Package B

```
package B;
import A.ClasseA;
public class ClasseC {

  ClasseA a;

  ClasseB b;

}
```

Méthodes et classes finales

Mot-clé final

- Méthodes finales
  - `public final void méthodeX(….) {`
    `…. }`
  - *« verrouiller » la méthode pour interdire toute éventuelle redéfinition dans les sous-classes*
- Une classe peut être définie comme finale
  - `public final class UneClasse {`
    `…`
    `}`
  - *interdit tout héritage pour cette classe qui ne pourra être sous-classée*
  - *toutes les méthodes à l'intérieur de la classe seront implicitement finales (elles ne peuvent être redéfinies)*
  - *exemple : la classe **`String`** est finale*

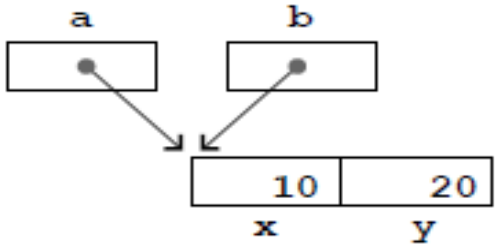
Recopier les attributs d'un objet

clone( )

l'affectation « *a* = *b* » d'un objet n'en fait pas une copie

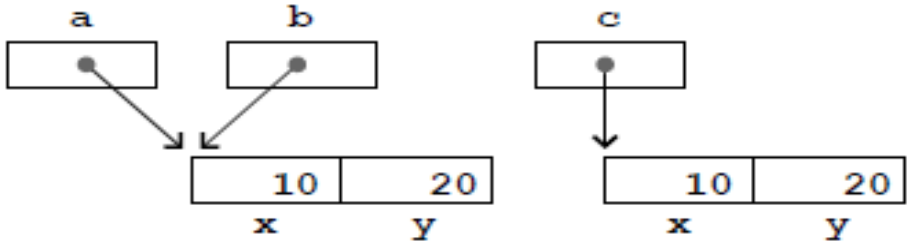
```
Point a = new Point(10, 20);
Point b = a;
```

a et *b* ne sont pas les noms de deux objets, mais deux noms pour *le même objet* :



pour avoir une vraie duplication :

```
Point c = (Point) a.clone();
```

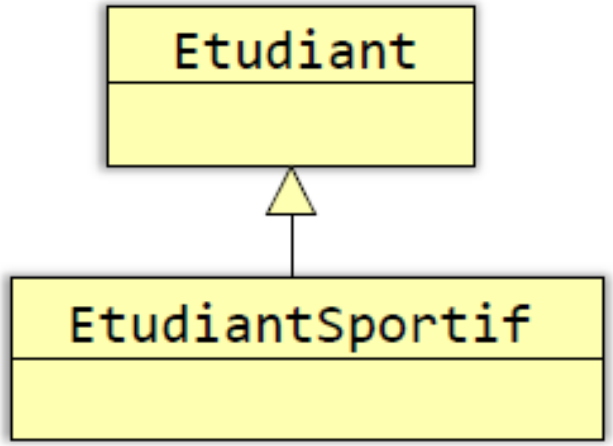


Polymorphisme

Surclassement

- La réutilisation du code est un aspect important de l'héritage, mais ce n'est peut être pas le plus important
- Le deuxième point fondamental est la relation qui relie une classe à sa superclasse:

Une classe B qui hérite de la classe A peut être vue comme un sous-type (sous ensemble) du type défini par la classe A.



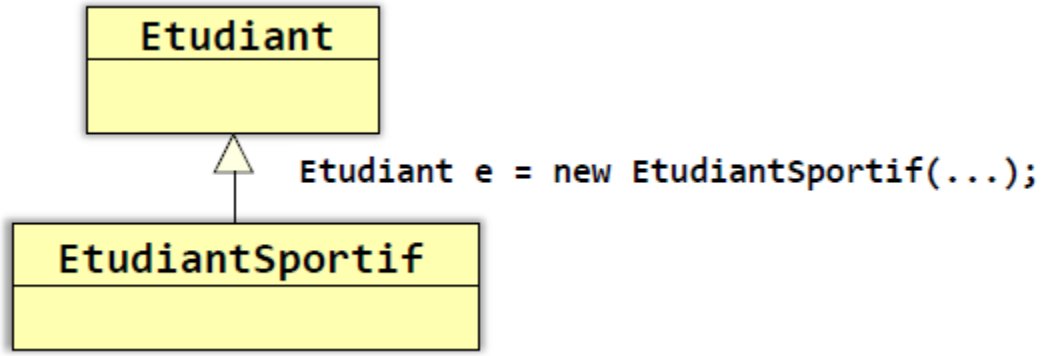
Un **EtudiantSportif** est un **Etudiant**

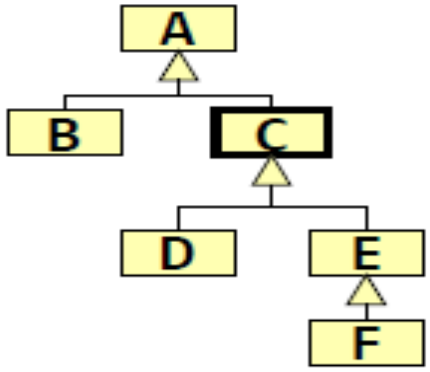
L'ensemble des étudiants sportifs est inclus dans l'ensemble des étudiants

Polymorphisme

Surclassement

- tout objet instance de la classe B peut être aussi vu comme une instance de la classe A.
  - Cette relation est directement supportée par le langage JAVA :
  - *à une référence déclarée de type A il est possible d'affecter une valeur qui est une référence vers un objet de type B(surclassementou upcasting)*






```
C c;
c = new D();
c = new E();
c = new F();
c = new A();
c = new B();
```

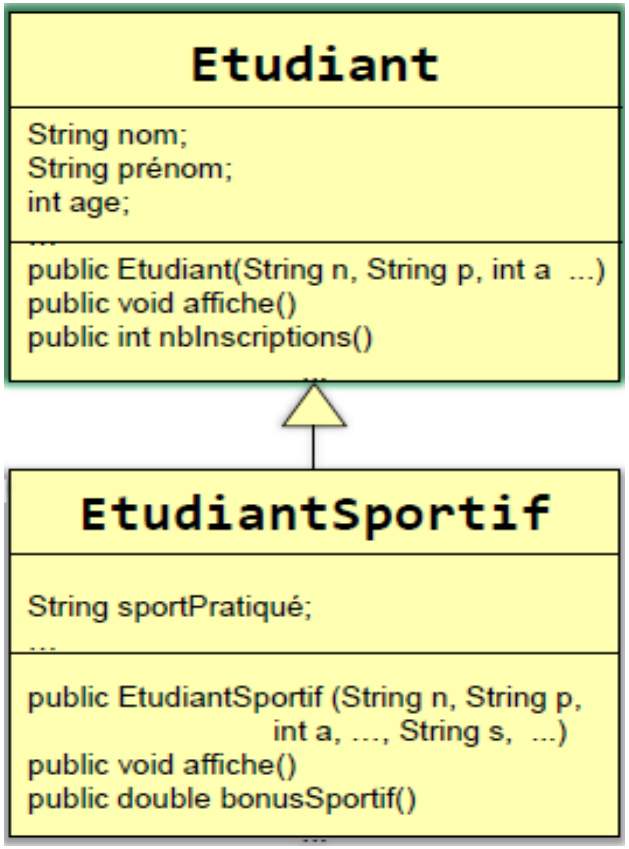
plus généralement à une référence d'un type donné, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous-classe directe ou indirecte du type de la référence

Polymorphisme

Surclassement

- Lorsqu'un objet est "sur-classé" il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
  - *Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence*

```
EtudiantSportif es = new
EtudiantSportif("FRAHAN","ANAS",25,..,
"tennis",..);
Etudiant e;
e = es; // upcasting
e.affiche();
es.affiche();
e.nbInscriptions();
es.nbInscriptions();
es.bonusSportif();
e.bonusSportif(); //erreur
```



Le compilateur refuse ce message:
pas de méthode `bonusSportif` définie dans la classe `Etudiant`

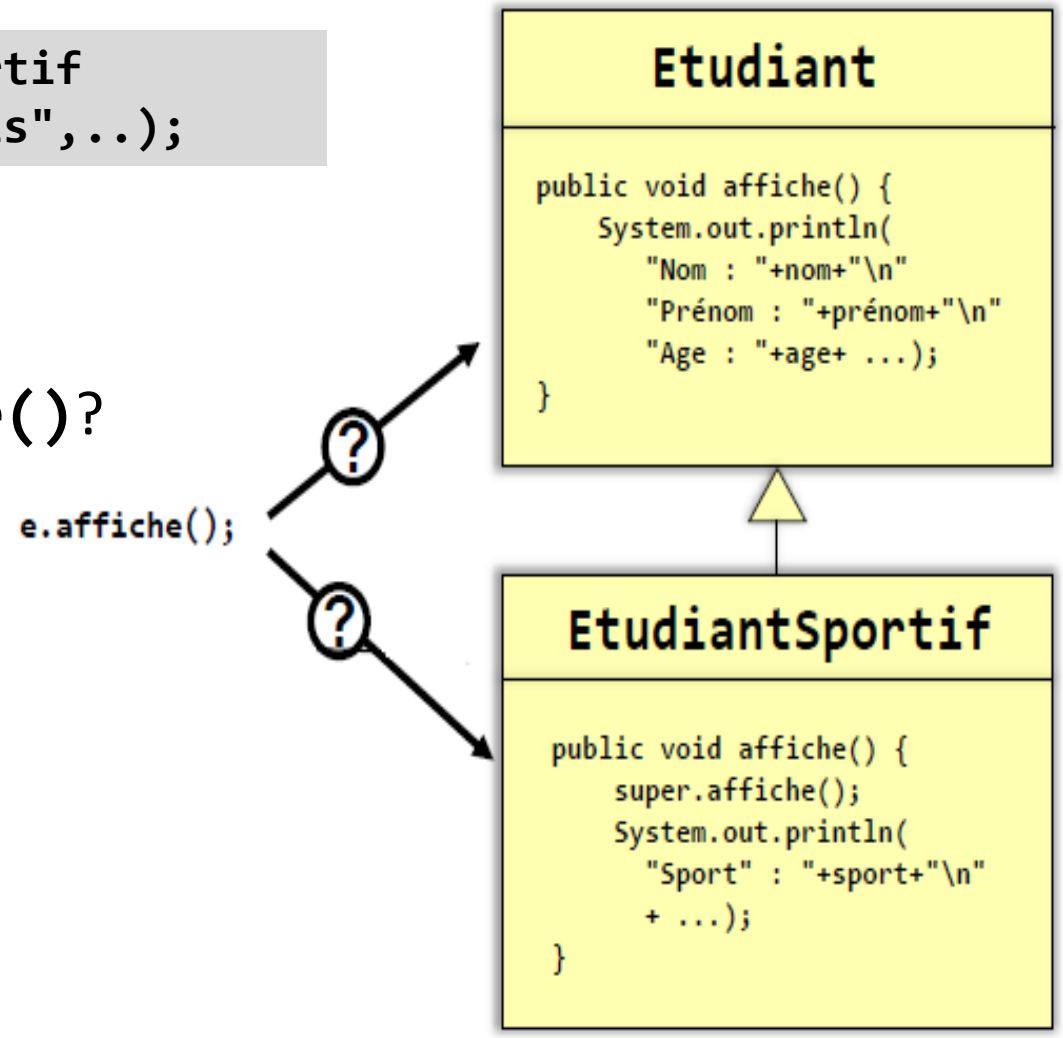
Polymorphisme

Lien dynamique

Résolution des messages

```
Etudiant e = new EtudiantSportif
("FRAHAN","ANAS",25,..,"tennis",..);
```

Que va donner **`e.affiche()`**?



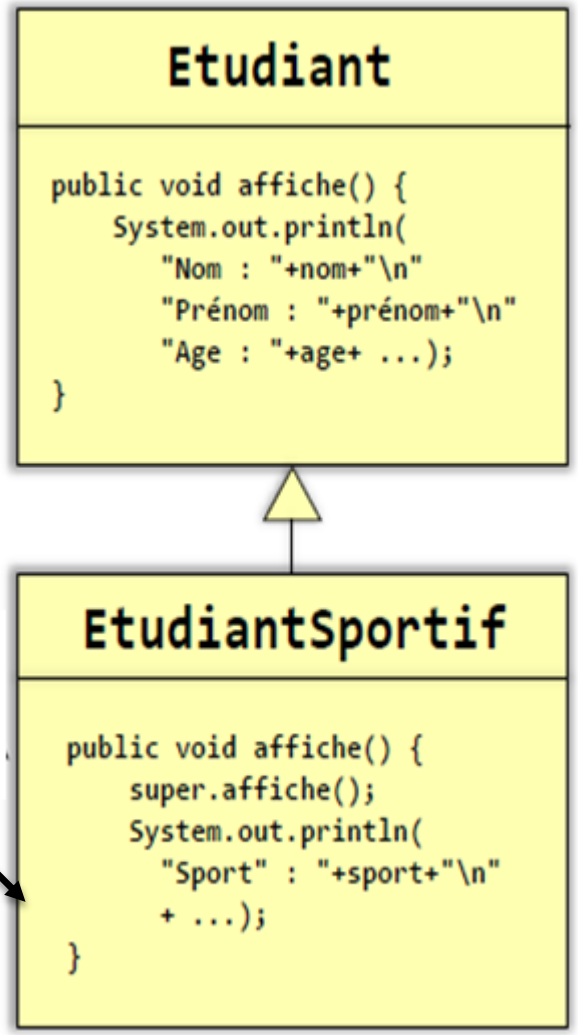
Polymorphisme

Lien dynamique

Résolution des messages

```
Etudiant e = new EtudiantSportif
("FRAHAN","ANAS",25,..,"tennis",..);
```

Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée",
c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est en fait invoquée et exécutée



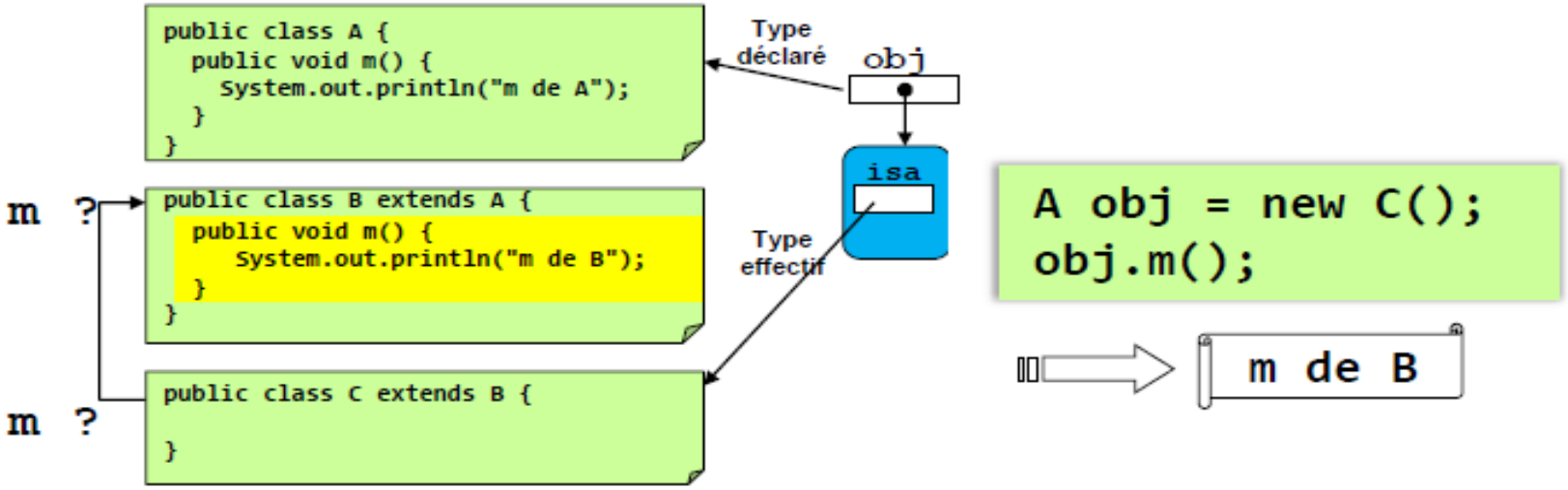
Polymorphisme

Lien dynamique

Mécanisme de résolution des messages

Les messages sont résolus à l'exécution

- *la méthode exécutée est déterminée à l'exécution (run-time) et non pas à la compilation*
- à cet instant le type exact de l'objet qui reçoit le message est connu
  - *la méthode définie pour le type réel de l'objet recevant le message est appelée (et non pas celle définie pour son type déclaré).*



ce mécanisme est désigné sous le terme de **lien-dynamique** (dynamicbinding, late-bindingou run-time binding)

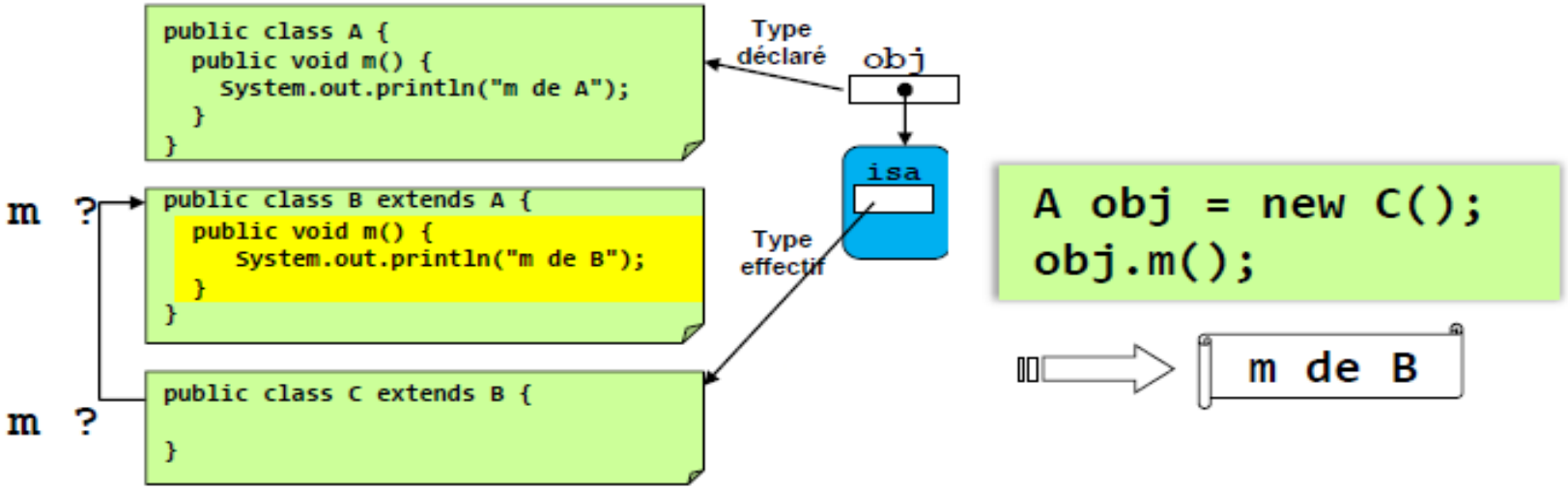
Polymorphisme

Lien dynamique

Mécanisme de résolution des messages

Les messages sont résolus à l'exécution

- *la méthode exécutée est déterminée à l'exécution (run-time) et non pas à la compilation*
- à cet instant le type exact de l'objet qui reçoit le message est connu
  - *la méthode définie pour le type réel de l'objet recevant le message est appelée (et non pas celle définie pour son type déclaré).*

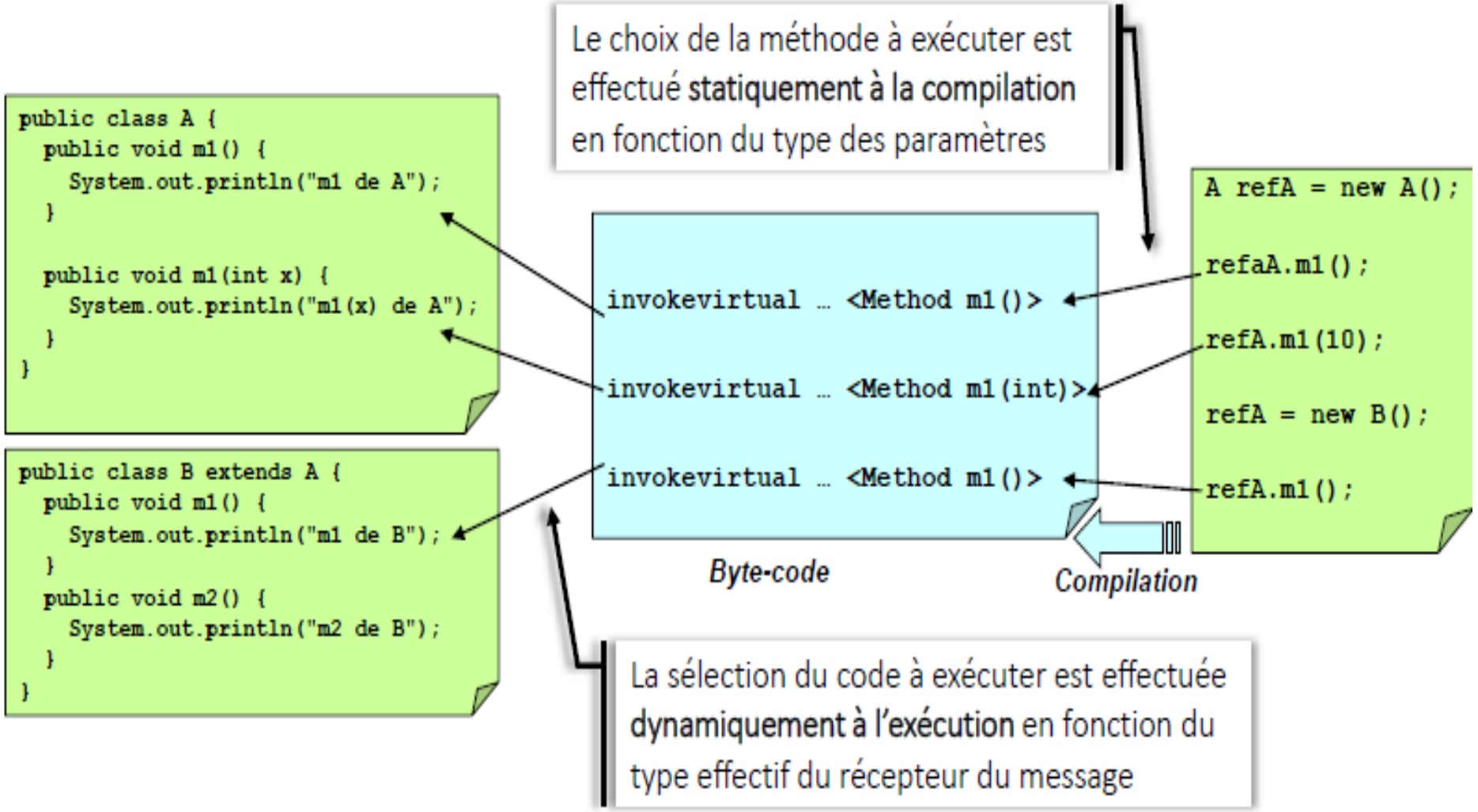


ce mécanisme est désigné sous le terme de **lien-dynamique** (dynamicbinding, late-bindingou run-time binding)

Polymorphisme

Lien dynamique

Choix des méthodes, sélection du code



Polymorphisme

A quoi servent l'upcasting et le lien dynamique ?

A la mise en oeuvre du polymorphisme

- Le terme polymorphisme (du grec, « multiforme ») décrit la caractéristique d'un élément qui peut se présenter sous différentes formes.
- En programmation par objets, on appelle polymorphisme
  - *le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.*
  - *le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.*
- Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage *Bruce Eckel " Thinkingin JAVA"*

Polymorphisme

```
ClasseA objA;

objA = ...

objA.methodeX();
```

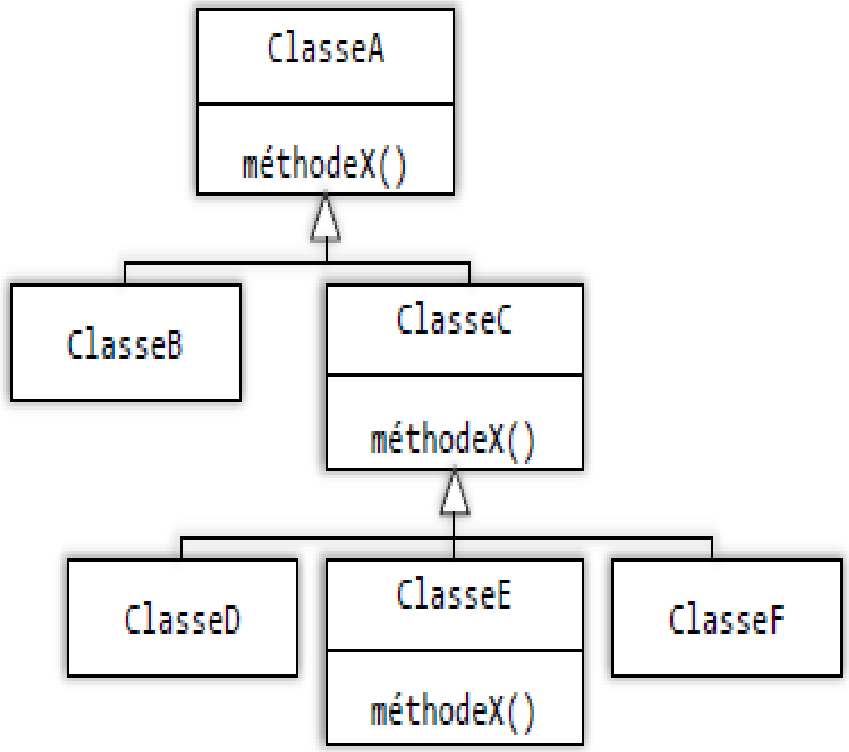
Surclassement

la référence peut désigner des objets de classe différente (n'importe quelle sous classe de ClasseA)

+

Lien dynamique

Le comportement est différent selon la classe effective de l'objet

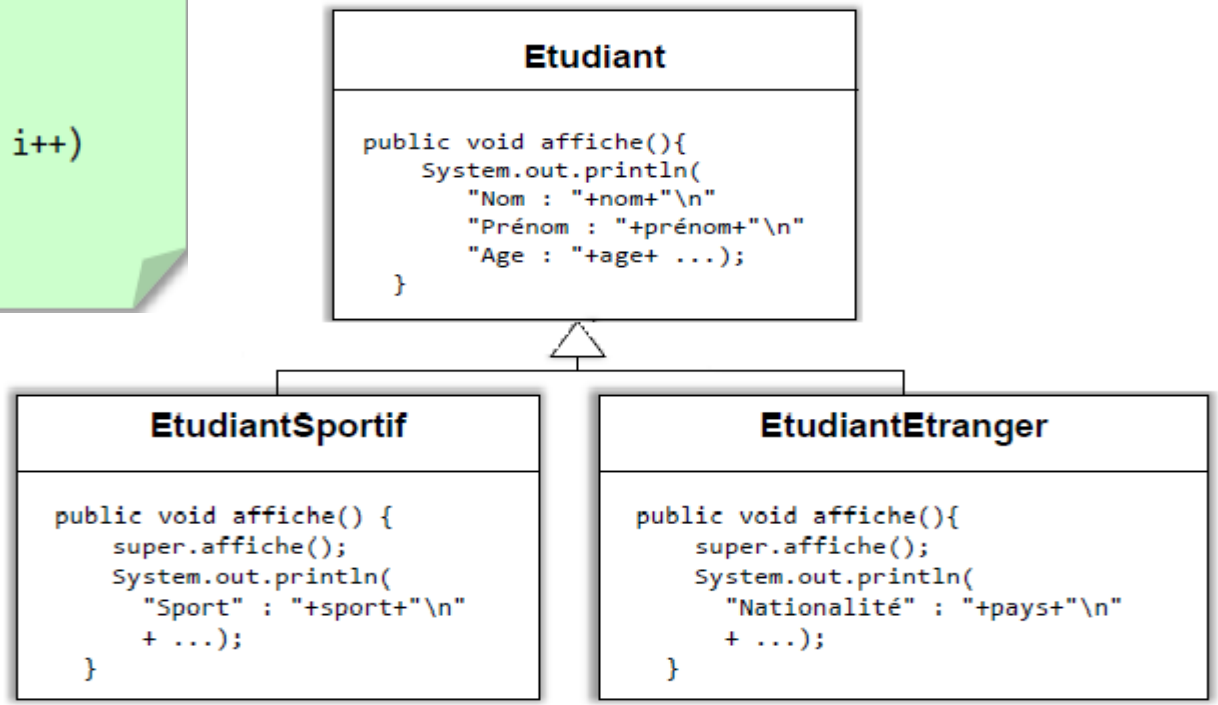


Polymorphisme

```
public class GroupeTD{

   Etudiant[] liste = new Etudiant[30];
   int nbEtudiants = 0;
   ...
   public void ajouter(Etudiant e) {
      if (nbEtudiants < liste.lenght)
         liste[nbEtudiants++] = e;
   }

   public void afficherListe(){
     for (int i=0;i<nbEtudiants; i++)
           liste[i].affiche();
   }
 }
```

Liste peut contenir des étudiants de n'importe quel type

```
GroupeTDtd1 = new GroupeTD();
td1.ajouter(new Etudiant("FARHAN", …));
td1.ajouter(new EtudiantSportif("BILAL",
"Ahmed", … , "ski");
```



Si un nouveau type d'étudiant est défini, le code de GroupeTD reste inchangé

Polymorphisme

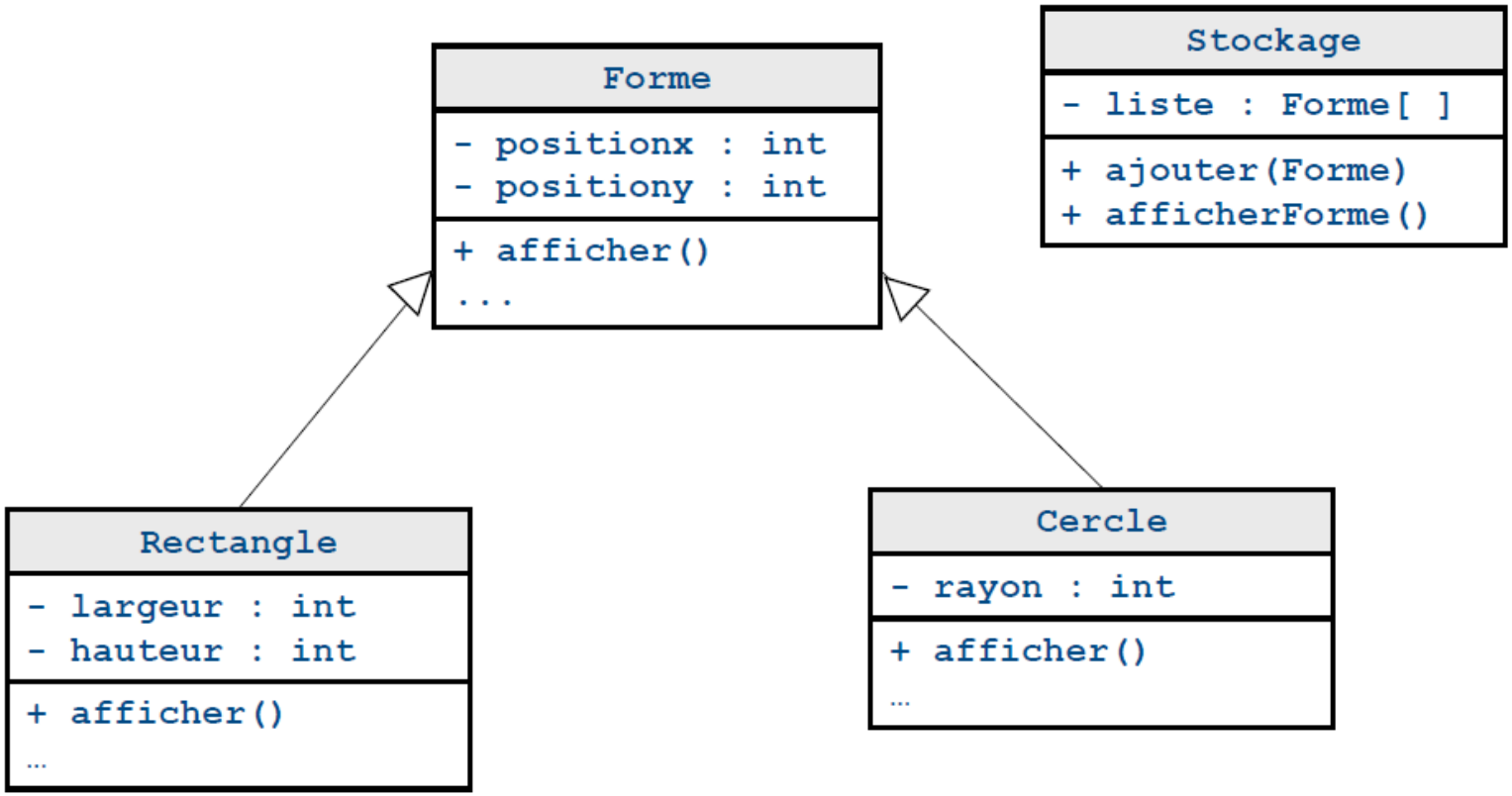
- En utilisant le polymorphisme en association à la liaison dynamique
- plus besoin de distinguer différents cas en fonction de la classe des objets
- possible de définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule l'interface de la classe de base

  - Développement plus rapide
  - Plus grande simplicité et meilleure organisation du code
  - Programmes plus facilement extensibles
  - Maintenance du code plus aisée

Polymorphisme

un exemple typique

- **Exemple :** la géométrie
  - Stocker des objets Forme de n'importe quel type (Rectangle ou Cercle) puis les afficher



Polymorphisme

Downcasting

```
ClasseX obj = …
  ClasseA  a = (ClasseA) obj;
```

- Le downcasting(ou transtypage) permet de « forcer un type » à la compilation
  - C'est une « promesse » que l'on fait au moment de la compilation.
- Pour que le transtypage soit valide, il faut qu'à l'exécution le type effectif de *obj* soit « compatible » avec le type *ClasseA*
  - Compatible : la même classe ou n'importe quelle sous classe de ClasseA (obj instanceof ClasseA)
- Si la promesse n'est pas tenue une erreur d'exécution se produit.
  - *Class CastException est levée et arrêt de l'exécution*

```
java.lang.ClassCastException: ClasseX
        at Test.main(Test.java:52)
```

Polymorphisme

A propos de equals

```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o
 }
 ...
}
```

```
public class Point {

      private double x;
      private double y;

      ...

}
```

```
@Override
public boolean equals(Object o) {
    if (this == o)
         return true;

    if (! (o instanceof Point))
         return false;

    Point pt = (Point) o; // downcasting

    return this.x == pt.x && this.y ==  pt.y;
}
```

redéfinir (overrides) la méthode equals(Object o) héritée de Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2)        --> true
Object o = p2;
p1.equals(o)         --> true

o.equals(p1)         --> true
```