LU2IN002
HÉRITAGE :
PRINCIPE DE SUBSOMPTION

Vincent Guigue & Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ
LIP6
MLIA Machine Learning & Deep Learning for Information Access

Si la classe B hérite de la classe A:

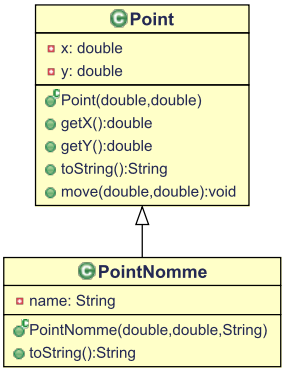
- le type B « EST-UN » le type A
- les méthodes de A peut-être invoquée sur une instance de la classe B (+ transitivité A hérite de SA, SA hérite de SSA, etc.)
- **Subsomption:** Dans toute expression « qui attend » un A (type A), je peux « placer » un B à la place

Polymorphisme = exploiter la subsomption

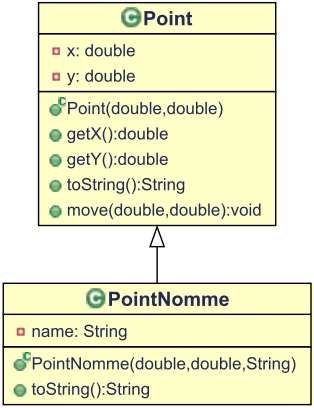
```
1 Point p = new PointNomme(1,2,"toto");
```

Un `PointNomme` EST UN `Point` ⇒

Je peux le traiter comme tel



Polymorphisme: représentation mémoire



- Syntaxe:

```
1 Point p = new PointNomme(1,2,"toto");
2 // PointNomme EST UN Point
```

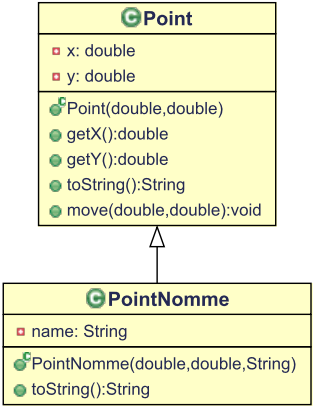
- Limite

```
1 p.getNom(); // -> ERREUR de compilation
```

Role du compilateur:

il vérifie le type des variables et les possibilités offertes par celles-ci.

Polymorphisme: représentation mémoire



- Syntaxe:

```
1 Point p = new PointNomme(1,2,"toto");
2 // PointNomme EST UN Point
```

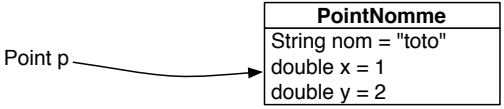
- Limite

```
1 p.getNom(); // -> ERREUR de compilation
```

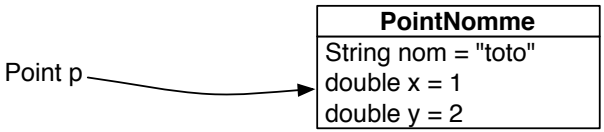
Role du compilateur:

il vérifie le type des variables et les possibilités offertes par celles-ci.

Représentation mémoire:



```java
Point p = new PointNomme(1,2,"toto");
```



Compilateur

La variable p est de type `Point`: seule les méthodes de `Point` sont accessibles:

- \+ `p.getX(); p.getY(); //...`
- − `p.getNom();`
  ⇒ Erreur de compilation (méthode inconnue dans la classe `Point`)

JVM

L'instance référencée par p est de type `PointNomme`

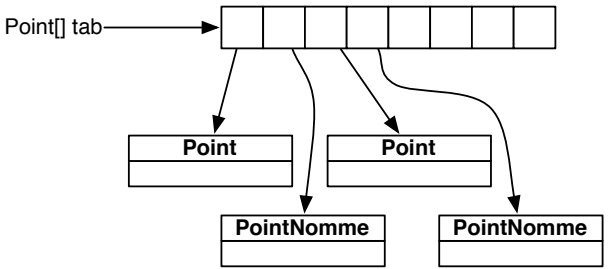
- \+ En cas d'appel à `toString()`, c'est bien la méthode de `PointNomme` qui est invoquée.

Application classique sur les tableaux de concepts abstraits:

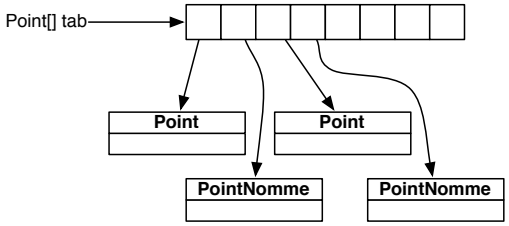
```
1 Point[] tab = new Point[10]; // OK,
2                   // il s'agit de 10 variables de type Point
```

Application classique sur les tableaux de concepts abstraits:

```
1 Point[] tab = new Point[10]; // OK,
2                     // il s'agit de 10 variables de type Point
3 for(int i=0; i<tab.length; i++){
4   if(i%2==0)
5     tab[i] = new Point(Math.random()*10, Math.random()*10);
6   else
7     tab[i] = new PointNomme(Math.random()*10,
8                             Math.random()*10, "toto"+i);
9 }
```



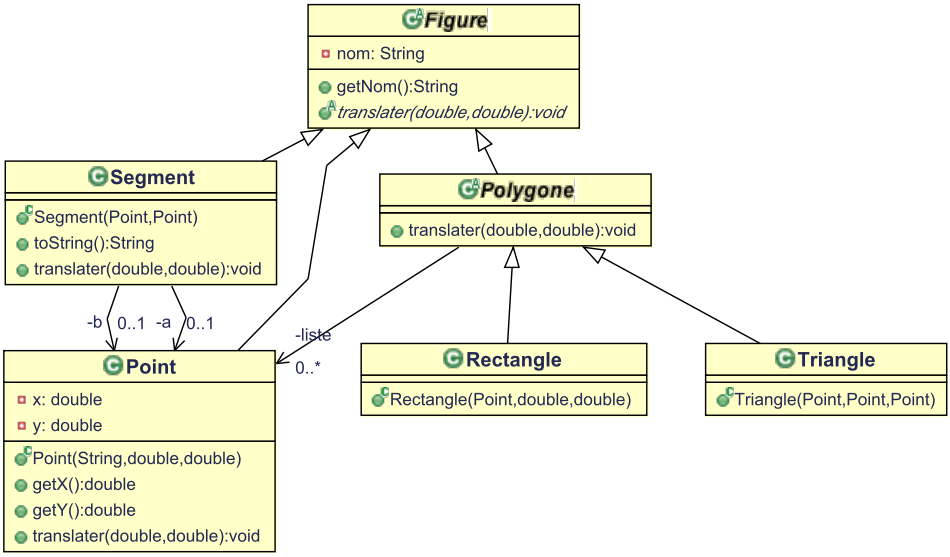
Polymorphisme: application sur un tableau




```java
// par exemple, procedure de Figure (methode de classe)
public static void translaterTout(Point[] pts, double tx, double ty) {
  for(int i=0;i<pts.length;i++)
    pts[i].move(tx,ty);
}

// variante
public static void translaterTout(Point[] pts, double tx, double ty) {
  for(Point p : pts)
    p.move(tx,ty);
}
```

Exemple plus complexe



- On ne peut qu'invoquer les méthodes du super-type
  - ex.: si le type est Figure, on ne peut invoquer que les méthodes de Figure, même si l'instance est un Point, un Carré, etc.

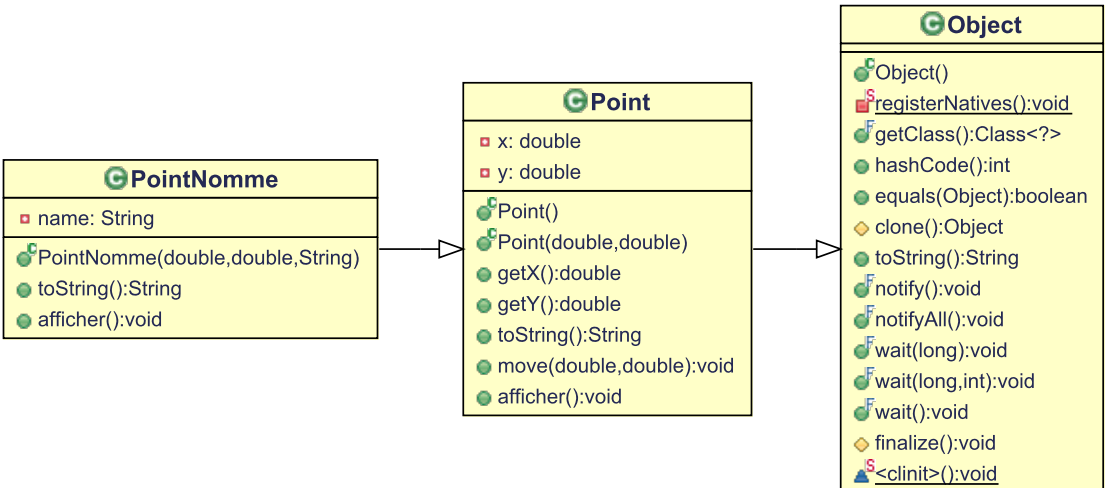
```
1 Figure fig1 = new Carre(2,1,4,5);
2 fig1.translater(2,2);  // OK type Figure
3 double x = fig1.calculerSurface(); // KO type Figure !
4
5 Figure fig2 = new Point(2,1);
6 fig2.translater(2,2);  // OK type Figure
7 fig2.getX(); // KO type Figure !
```

Role du compilateur:

il vérifie le type des variables est les possibilités offertes par celles-ci.

Classe standard

Toutes les classes dérivent de la classe Object de JAVA



- Avec une variable `Object`, je peux référencer n'importe quelle instance:

```
1 Object o = new Point(1,2);
2 Object o2 = new PointNomme(2,3,"toto");
```

- ... Mais je ne peux (presque) rien faire sur `o,o2`

```
3 System.out.println(o.toString()); // OK
4 System.out.println(o.getX()); // KO
5 System.out.println(o.getY()); // KO
6 System.out.println(o2.getNom()); // KO
```

- Je peux donc créer un tableau/`ArrayList` avec n'importe quoi dedans:

```
7 Object[] tab = new Object[10];
8 tab[0] = "toto";
9 tab[1] = 10; // autoboxing
10 tab[2] = new Point(1,2);
```

Arguments de méthode

```java
// dans n'importe quelle classe, par exemple Truc
public void maMethode(Point p){
  ...
}

// invocations possibles:
Truc t = new Truc();
t.maMethode(new Point(1,2));
t.maMethode(new PointNomme(1,2, "toto"));
t.maMethode(new ClasseHeritantDePoint());
```

Idée:

Comme tous les descendants de `Point` sont des `Point`...
⇒Toutes les informations utiles/nécessaires et toutes les méthodes clientes sont disponibles
⇒ aucun problème technique en perspective !

Le cas de la méthode equals

```java
    // Exemple de la classe Point
public boolean equals(Object obj) {
if (this == obj)
    return true;
if (obj == null)
    return false;
if (getClass() != obj.getClass())
    return false;
Point other = (Point) obj;
if (x != other.x)
    return false;
if (y != other.y)
    return false;
return true;
}
```

Tous les objets sont comparables entre eux ! On peut donc rechercher un objet en particulier dans un tableau exploitant le polymorphisme:

```java
ArrayList<Object> tab =
            new ArrayList<Object>();
tab.add( "toto"); tab.add(10);
tab.add(new Point(1,2));
tab.add(new PointNomme(2,3,"titi");

tab.contains(new Point(1,2)); // true
// methode exploitant equals
```

Question suitante: comment passer d'un `Object` (générique) à une classe spécifique (ici : `Point`)?

$\Rightarrow$ réponse dans le cours sur l'opérateur `cast`

- Le principe de subsomption est ce qui fait l'intérêt de l'héritage:
  - stockage d'instances hétérogènes dans des structures de données (type liste),
  - application des méthodes en *batch* sur toutes ces données.
- ... **A condition d'avoir bien réfléchi à l'architecture**, aux méthodes communes des différentes classes !
- Enfin, attention à ne pas s'emmêler: si `A` EST UN `B` alors `B` n'est pas un `A`. **La subsomption ne marche que dans un sens**.