LU2IN002 - Introduction à la programmation objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 7 – 18 octobre 2024

Programme du jour

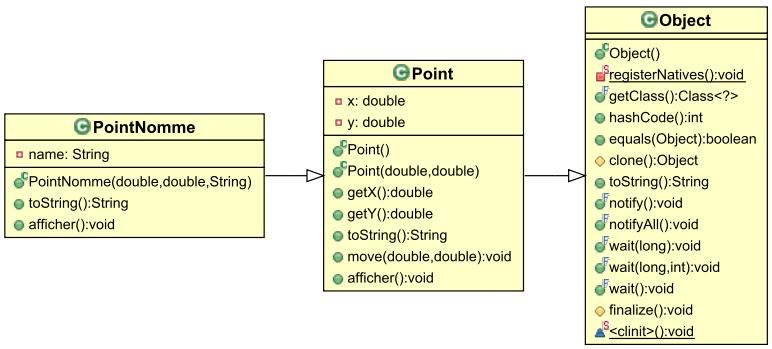
1. Retour sur l'héritage
2. Héritage : surcharge / redéfinitions
3. Héritage : les classes abstraites

Plan du cours

Classe Object

Classe standard

Toutes les classes dérivent de la classe Object de JAVA



Cet héritage est implicite, pas de déclaration dans la signature

Objet (suite)

- Une variable Object peut contenir une référence d'instance quelconque

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

- ... mais on ne peut (presque) rien faire sur o et o2

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

- Création d'un tableau/ArrayList contenant "n'importe quoi"

```
Object[] tab = new Object[10];
tab[0] = "toto";
tab[1] = 10; // --> conversion implicite en Integer
tab[2] = new Point(1,2);
```


Arguments de méthode

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

// invocations possibles: dans une méthode main()
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 : une instance d'une classe héritant de Point sait faire tout ce que sait faire un Point
⇒ aucun problème technique en perspective !

Conclusion et limites

- 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.
- ... **à condition d'avoir bien réfléchi à l'architecture**, aux méthodes communes des différentes classes !
- Attention (rappel) : si `A` EST UN `B` alors `B` n'est pas un `A`
  $\Longrightarrow$ la subsomption ne marche que dans un sens

Plan du cours

Surcharge

Définition

- même nom de méthode MAIS argument(s) différent(s)
- le type de retour n'est pas considéré pour différencier les méthodes

```
1  public class Point {
2  ...
3      public void move(double dx, double dy){
4          x+=dx; y+=dy;
5      }
6      public void move(double dx, double dy, double scale){
7          x+=dx*scale; y+=dy*scale;
8      }
9      public void move(int dx, int dy){
10         x+=dx; y+=dy;
11     }
12     public void move(Point p){
13         x+=p.x; y+=p.y;
14     }
```

Rappel : dans la classe `Point`, accès aux attributs privés des autres instances de `Point`

Surcharge : qui fait quoi

Le compilateur (pré)-sélectionne les méthodes :

- Ces méthodes sont totalement différentes pour le compilateur qui analyse le type des arguments
- Elles peuvent être indifféremment dans la classe ou la super-classe

```
1  public class Point {
2  ...
3      public void move(double dx, double dy){ // 1
4          x+=dx; y+=dy;
5      }
6      public void move(double dx, double dy, double scale){ // 2
7          x+=dx*scale; y+=dy*scale;
8      }
9  ////////////////////////////////////////////
10
11 Point p = new Point(1,2);
12 p.move(3, 1); // présélection de 1
13 p.move(3, 1, 0.5); // présélection de 2
```


Redéfinition :

Définition

Redéfinition d'une méthode de **même signature** dans la classe fille

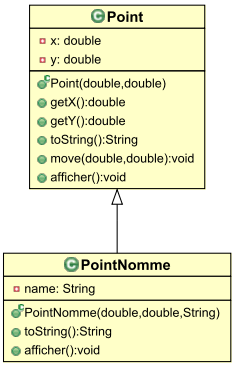
```
1  public class Point {
2  ...
3      public void afficher(){ // 1
4          System.out.println("Je␣suis␣un␣Point");
5      }
6  }
7  ////
8  public class PointNomme extends Point {
9  ...
10     public void afficher(){ // 2
11         System.out.println("Je␣suis␣un␣PointNomme");
12     }
13 ...
14 }
```

- Pas de problème à la compilation
- A l'exécution, la JVM décide de la méthode à invoquer en fonction du type de l'instance appelante

Exemples de fonctionnement 1/3

Cas 1 : (facile)

```
1      public static void main(String[] args) {
2          Point p = new Point(1, 2);
3          p.afficher();
4      }
```



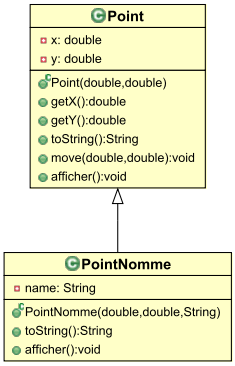
Dans la classe Point, une seule méthode correspond à la signature `afficher()`
Affichage de :

```
1  Je suis un Point
```

Exemples de fonctionnement 2/3

Cas 2 : (résolution d'une ambiguité)

```
1     public static void main(String[] args) {
2         PointNomme p = new PointNomme(1, 2,"toto");
3         p.afficher();
4     }
```



Dans la classe PointNomme, deux méthodes correspondent à la signature `afficher()` (méthodes accessibles : dans la classe et dans les classes parentes)

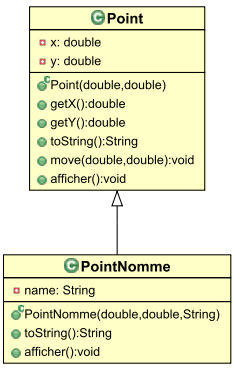
- Une dans `Point`
- Une dans `PointNomme`

La JVM choisit, au moment de l'exécution du programme en fonction du type de l'instance de p, la méthode la plus proche
Affichage de :
`Je suis un PointNomme`

Exemples de fonctionnement 3/3

Cas 3 : Surcharge + **subsomption**

```
1     public static void main(String[] args) {
2         Point p = new PointNomme(1, 2,"toto"); // subsomption
3         p.afficher(); // ???
4     }
```



Compilation : (javac) = type des variables uniquement

- `p` = `Point`
- `void afficher()` existe dans `Point` ⇒ OK

Exécution : JVM regarde le **type de l'instance** référencée par `p`

- `p` référence un `PointNomme`
- recherche de `void afficher()` dans `PointNomme`

Affichage de :
`Je suis un PointNomme`

Subsomption et surcharges (1)

- Attention ! Le mélange des 2 peut être (d)étonnant...

```
public class MaClasseA {
  public void affiche(double x) {
    System.out.println("La classe A affiche un double : "+x);
  }
}
```

```
public class MaClasseB extends MaClasseA{
  public void affiche(int n) {
    System.out.println("La classe B affiche un entier : "+n);
  }
}
```


Subsomption et surcharges (2)

```
public class TestMesClassesAB {
  public static void main(String[] args) {
    MaClasseA a = new MaClasseA();
    a.affiche(11.38);

    MaClasseB b = new MaClasseB();
    b.affiche(42);
    b.affiche(11.38);

    MaClasseA ab = new MaClasseB();
    ab.affiche(11.38);
    ab.affiche(42);
  }
}
```

- Qu'obtient-on ?

Subsomption et surcharges (3)

```
// MaClasseA a = new MaClasseA();
Résultat pour a.affiche(11.38) :
La classe A affiche un double : 11.38
```

```
// MaClasseB b = new MaClasseB();
Résultat pour b.affiche(42) :
La classe B affiche un entier : 42
```

```
Résultat pour b.affiche(11.38) :
La classe A affiche un double : 11.38
```

```
// MaClasseA ab = new MaClasseB();
Résultat pour ab.affiche(11.38) :
La classe A affiche un double : 11.38
```

```
Résultat pour ab.affiche(42) :
La classe A affiche un double : 42.0
```

- Pourquoi ce résultat ?

Usage de super 1/3

Le mot clé super permet :

- de préciser que l'on utilise des informations de la super-classe

```
public class PointNomme extends Point {
  public void afficher(){
    System.out.println("Je suis un PointNomme " +
       "de coordonnées : "+ super.getX()+ " " +
        super.getY());
  }
}
```

- de forcer le programme à aller chercher une méthode dans la super-classe (**obligatoire**)

```
public class PointNomme extends Point {
...
    public void affichageGlobal(){
       afficher(); // -> Je suis un PointNomme
       this.afficher(); // -> Je suis un PointNomme
       super.afficher(); // -> Je suis un Point
    }
}
```

Usage de super 2/3

L'un des usages les plus classique concerne `toString()` :

```
// classe Point
   public String toString() {
      return "("+x+","+y+")";
    }

// classe PointNomme
   public String toString() {
        return "PointNomme [name=" + name + super.toString() + "]";
    }
```

1. Quels sont les affichages en sortie du code suivant :

```
Point p = new Point(1,2);
PointNomme pn = new PointNomme(3,4,"toto");

System.out.println(p.toString());
System.out.println(pn.toString());
```

2. Que se passerait-il si on oublie le `super` ?

Usage de super 3/3

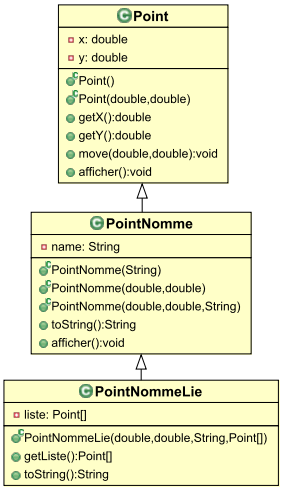
Ajout d'une classe pour un Point lié à d'autres dans l'espace (système de graphe)

```
// dans PointNommeLie

toString(); // OK local
super.toString(); // OK super-classe
super.super.toString(); // syntaxe interdite

getX(); // OK, existe ici par
        // héritage de Point
super.getX(); // OK existe dans PointNom
        // par héritage de Point
```

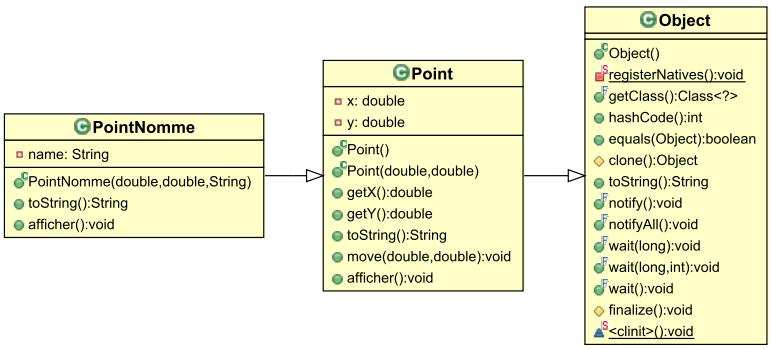
Les méthodes redéfinies dans la super-classes bloquent l'accès aux versions de la génération supérieure.



Classe Object

Classe standard

Toutes les classes dérivent de la classe Object de JAVA



Cet héritage est implicite, pas de déclaration dans la signature

Méthodes standards

Dans le cadre de l'UE, nous nous intéresserons à certaines méthodes dites *standards*

- String toString()
- boolean equals(Object)

Sur la classe basique suivante :

```
public class Point {
    private double x,y;
    public Point(){
        x=0; y=0;
    }
}
```

Les opérations suivantes sont possibles :

```
Point p1 = new Point(1,2); Point p2 = new Point(1,2);
System.out.println(p1.toString());
System.out.println(p1.equals(p2));
System.out.println(p1.equals(p1));
```

Quelles sont les sorties associées ?

Rappels sur `boolean equals(Object o)`

- Par défaut, `equals` existe mais teste l'égalité référentielle, ce qui n'est pas intéressant...
- Redéfinition = faire en sorte de tester les attributs
  Un processus en plusieurs étapes :
  1. Vérifier s'il y a égalité référentielle :
     - si `true` renvoyer `true`
  2. Vérifier le type de l'`Object o` (cf prochain cours)
  3. Convertir l'`Object o` dans le type de la classe (idem)
  4. Vérifier l'égalité entre attributs

```
    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;
    }
```


Ouverture de la redéfinition

Définition

Il est possible d'augmenter la visibilité d'une méthode dans la classe fille mais pas de la réduire

Exemple :

- dans `Object`

```
  protected Object clone(){...}
```

- dans `Point`

```
// pour éviter les cast
  protected Point clone(){return new Point(...);}
```

- dans `PointNomme`

```
// ouverture de la redéfinition
public PointNomme clone(){return new PointNomme(...);}

// MAIS:
//private PointNomme clone(){return new PointNomme(...);}
// Ne compile pas
```

Plan du cours

1. Retour sur l'héritage
2. Héritage : surcharge / redéfinitions
3. Héritage : les classes abstraites
   - abstraction

Nouveaux concepts

- **Classe abstraite**
  - Classe qui ne sera pas instanciable
  - Les classes filles pourront être instanciables
  - Exemple :
    - Animal (abstraite) : définit un comportement général
    - Mouton, Tigre : animaux avec comportements spécifiques
- **Méthode abstraite**
  - Seulement dans les classes abstraites
  - Elle contient une signature mais pas de code
  - Exemple :
    - Animal (abstraite) : `String regimeAlimentaire()`
    - Mouton, Tigre $\Longrightarrow$ `"herbivore"`, `"carnivore"`

Classe abstraite

Définition

- Représente une classe qui ne peut pas être instanciée
- Un concept unificateur qui permet de factoriser du code pour toutes les classes qui hériteront
- Introduction de la notion de contrat : toutes les classes filles devront définir ce qui est déclaré dans la classe mère (signature de méthode abstraite)
  - tous les animaux ont un régime alimentaire...
    → la signature de la méthode regimeAlimentaire() est donnée dans la classe mère Animal
  - ... mais la nature du régime est propre au mouton, tigre...
    → le code de la méthode regimeAlimentaire() est défini dans chaque classe fille Mouton, Tigre,...

Classe abstraite : syntaxe

```java
public abstract class Animal {
  ...
  // signature seulement : PAS D'ACCOLADES !
  public abstract String regimeAlimentaire();
}
```

- Il est impossible de créer une instance de la classe `Animal`

```java
new Animal();  // --> ERREUR compilation: abstract class
```

- Des classes peuvent hériter de Animal, elles doivent alors
  - soit implémenter regimeAlimentaire()

```java
public class Mouton extends Animal {
  ...
  public String regimeAlimentaire() { // code méthode
    return "Herbivore";
  }
}
```

  - soit être elles-même abstraites

```java
public abstract class Poisson extends Animal {
  ...
}
```


Propriétés des classes abstraites

- Les classes abstraites sont des classes, elles peuvent avoir
  - des attributs
  - des constructeurs
  - des méthodes "normales"
- mais en plus, elles peuvent aussi avoir (ou pas)
  - des méthodes abstraites

```java
public abstract class Animal {
  private int age;
  public Animal(int age) {
    this.age = age;
  }
  public int getAge(){
    return age;
  }
  public abstract String regimeAlimentaire();
}
```

Idées

Les classes abstraites sont pensées pour leurs descendantes, les classes filles qui en seront dérivées

(Retour) sur les bonnes pratiques

Développement à long terme

Modification d'un projet existant = ajout d'une classe

- ne pas modifier les classes existantes
- ajouter des classes filles

Idée :

Structurer un projet avec des classes abstraites :

- les classes filles possèdent des fonctionnalités dès leur création
  - factorisation du code
- ajout de contraintes sur les classes filles
  - plus facile à développer (classe fille = canevas à remplir)
  - contrat sur les fonctionnalités (garanties)
  - garanties sur des classes qui n'existent pas encore : facilités d'évolution du code
- usage du polymorphisme
  - ex. : tableau hétérogène $\Rightarrow$ + de possibilités

Classe abstraite : bilan

- une classe abstraite ne peut pas être instanciée
- si une classe contient une méthode abstraite, alors cette classe doit être abstraite
- si une classe hérite d'une méthode abstraite, elle doit
  - soit définir le code de cette méthode
  - soit être déclarée comme une classe abstraite
- une classe abstraite peut ne pas contenir de méthode abstraite
- pourquoi déclarer une classe abstraite ?
  - pour empêcher son instanciation
  - pour représenter un concept abstrait dans l'application
    → par exemple, un instrument de musique (cf. exo 41)