2i002 - Programmation par objets en Java

Christophe Marsala

(support réalisé par Vincent Guigue)
(et d'autres sources diverses L. Denoyer, F. Peschanski,...)

Cours 4 - 7 février 2017

Programme du jour

1. Egalité entre objets, Clonage d'objet (suite)
2. Cycle de vie des objets
3. Objets complexes, composition d'objets

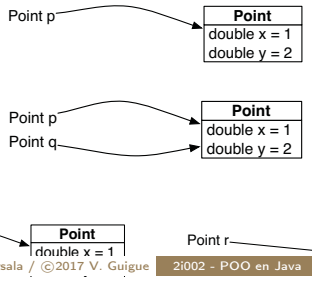
Plan du cours

Problématique

- Le signe = se comporte de manière spécifique avec les objets...
- Le signe == également spécifique avec les objets...

Vocabulaire (uniquement pour les opérations sur objets)

new : instanciation / création d'instance
= : duplication de référence
== : égalité référentielle



Comment dupliquer une instance ?

Idée (assez raisonnable somme toute)

Créer une nouvelle instance dont les valeurs des attributs sont identiques

- Exemple de code dans la classe `Point`

```
1 public class Point{
2 ...
3   public Point clone(){
4     return new Point(x, y);
5   }
6 }
```

- Usage :

```
1 // main
2 Point p = new Point(1,2);
3 Point p2 = p.clone();
```

NB : construction de nouvelle instance sans `new` explicite dans le `main`

Clonage : syntaxe alternative

Avant le JAVA, il y avait le C++

En C++ : la syntaxe standard = **constructeur de copie**

- Exemple de code dans la classe `Point`

```
1 // Constructeur de Point a partir d'un autre Point
2 public Point(Point p){
3   this.x = p.x; // this facultatif
4   this.y = p.y; // this facultatif
5 }
```

- Usage :

```
1 Point p = new Point(1,2);
2 Point p2 = new Point(p);
```

- Résultat ABSOLUMENT identique (depuis JAVA 1.5)
- Avantage du clone en JAVA : il s'agit d'une méthode standard (= + facile à lire)

Comment tester l'égalité structurelle ?

Idée (toujours assez raisonnable)

Créer une méthode qui teste l'égalité des attributs

- **Solution 1** (simple mais pas utilisée)

```java
// Dans la classe Point
public boolean egalite(Point p){
  return p.x == x && p.y == y;
}
```

```java
// Dans le main

Point p1 = new Point(1.,2.);
Point p2 = p1;
Point p3 = new Point(1.,2.);
Point p4 = new Point(1.,3.);

p1.egalite(p2); // true
p1.egalite(p3); // true
p1.egalite(p4); // false
```

- `public boolean egalite(Point p)` produit le résultat attendu
- ⚠ ATTENTION à la signature :
  - la méthode retourne un booléen
  - la méthode ne prend qu'un argument (on teste l'égalité entre l'instance qui invoque la méthode et l'argument)

- **Solution 2 : standard**... mais un peu plus complexe

Méthode standard : `boolean equals(Object o)`

- `equals` existe dans tous les objets (comme `toString`)
  - mais teste l'égalité référentielle... Pas intéressant (comme `toString` en version de base)
- ⇒ Redéfinition : faire en sorte de tester les attributs

Un processus en plusieurs étapes :

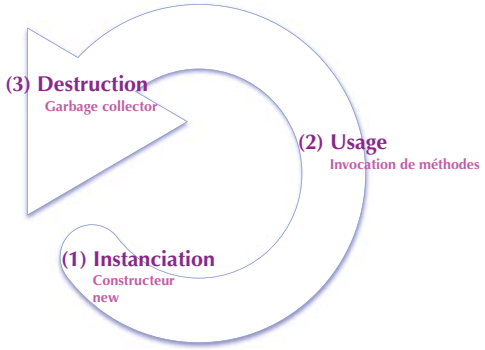
1. Vérifier s'il y a égalité référentielle / référence null
2. Vérifier le type de l'`Object` o (cf cours polymorphisme)
3. Convertir l'`Object` o dans le type de la classe (idem)
4. Vérifier l'égalité entre attributs

```java
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;
}
```


Plan du cours

Cycle de vie : définition

Se placer du point de vue de l'objet :



(1) création d'une instance

(2) évolution / utilisation de l'instance

(3) condition de destruction

(1) Instanciation

Coté fournisseur :

mise en route de l'objet

Instanciation = constructeur = contrat d'initialisation des attributs

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2,double y2){
    x = x2;
    y = y2;
  }
}
```

Coté client :

création d'une instance

Instanciation = création d'une zone mémoire réservée à l'objet

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
  }
}
```

La variable `p1`, de type `Point`, référence un instance de `Point` dont les attributs ont pour valeur 1 et 2.

(2) Usage

- le **fournisseur** développe et garantit le bon fonctionnement des méthodes pour *utiliser* l'objet correctement,
- le **client** invoque les méthodes sur des objets pour les manipuler.

```java
//Fichier Point.java
public class Point{
  private double x,y;
  public Point(double x2,double y2){
    x = x2;   y = y2;
  }

  public void move(double dx,
                   double dy){
    x += dx; y += dy;
  }
  ...
}
```

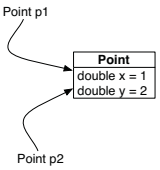
```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
    p1.move(2., 3.);
                // p1 => [x=3, y=5]
  }
}
```

(3) Destruction

1. Un objet est détruit lorsqu'il n'est **plus référencé**
2. La destruction est implicite (contrairement au C++) et traitée en tâche de fond (garbage collector)

- Un objet peut être référencé plusieurs fois...

```
//Fichier TestPoint.java
public class TestPoint{
  public static void main (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
    Point p2 = p1;
  }
}
```

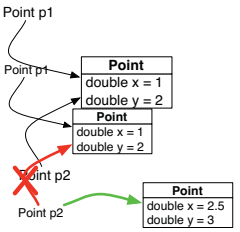


- mais quand est-il dé-référencé ?

Dé-référencement d'un objet

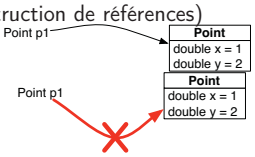
1. Dé-référencement explicite (usage de =

```
//Fichier TestPoint.java
public class TestPoint{
  public static void main (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
    Point p2 = p1;
    p2 = new Point(2.5, 3.);
  }
}
```



2. Dé-référencement implicite (logique de bloc, destruction de variables ⇒ destruction de références)

```
for (int i; i<10;i++) {
      Point p1 = new Point(1,2);
      System.out.println(p1);
  }
  System.out.println(p1);
  // ERREUR DE COMPILATION
  // p1 n'existe plus ici !
```



Retour sur la logique de bloc...

1. le dé-référencement dépend de l'endroit où la variable est **déclarée** (pas de l'endroit où la variable est **initialisée**)
2. ne pas confondre la destruction d'une **variable** et la destruction d'une **instance**

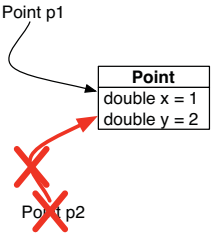
```
public static
    void main(String[] args) {


  {
      Point p1 =
            new Point(1,2);
      System.out.println(p1);
  } // destruction de
    // la variable p1

  System.out.println(p1);
  // ERREUR DE COMPILATION
  // p1 n'existe plus ici !
}
```

```
public static
    void main(String[] args) {
  Point p1; // déclaration
            // avant le bloc
  {
      // initialisation de p1
      p1 = new Point(1,2);
      System.out.println(p1);
  } // pas de destruction de p1


  System.out.println(p1);
  // OK, pas de problème
}
```


Retour sur la logique de bloc (2)

1. le dé-référencement dépend de l'endroit où la variable est **déclarée** (pas de l'endroit où la variable est **initialisée**)
2. ne pas confondre la destruction d'une **variable** et la destruction d'une **instance**

```
public static
    void main(String[] args) {
  Point p1; // déclaration
            // avant le bloc
  {
      Point p2 = new Point(1,2);
      // initialisation de p1
      p1 = p2;
      System.out.println(p1);
  } // destruction de p2

  System.out.println(p1);
  // OK, pas de problème
}
```

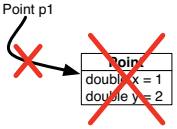


- Fin de bloc = destruction des **variables** déclarées dans le bloc
- Destruction d'instance ⇔ instance plus référencée

Destruction des instances

Destruction d'instance ⇔ instance plus référencée

```
public static void main(String[] args) {
  Point p1 = new Point(1,2);
  p1 = null;
}
```



- Pas besoin d'expliquer comment détruire un objet (≠ C++)
- Le **Garbage Collector** planifie la destruction

```
public static void main(String[] args) {
...
  for(int i=0; i<10; i++){
    // optimisation possible:
    //      réutilisation de la mémoire allouée
    Point p1 = new Point((int)(Math.random()*10),
                         (int)(Math.random()*10));
    ...
  }
}
```

- Appel explicite au garbage collector (pour libérer la mémoire) :

```
  System.gc();
```


Le mot de la fin...

... sur un exemple parlant :

```
Point p = new Point(1,2);
Point p2 = new Point(3,4);
// Point p3 = p; // différence avec et sans cette ligne
p = p2;
```

- Cas 1 : ligne commentée.
  - L'instance `Point(1,2)` **est détruite** à l'issue du re-référencement de p...
  - ... de toutes façons, cette instance était inaccessible.

```
Point p = new Point(1,2);
Point p2 = new Point(3,4);
Point p3 = p; // différence avec et sans cette ligne
p = p2;
```

- Cas 2 : ligne dé-commentée
  - L'instance `Point(1,2)` est **conservée**...
  - On y accède par la variable p3

Plan du cours

Philosophie & syntaxe

Un objet complexe = un objet qui utilise des objets

- Chaque classe reste petite, lisible et facile à débugguer
- Par agrégation, on construit des concepts complexes

Syntaxe : simple et intuitive

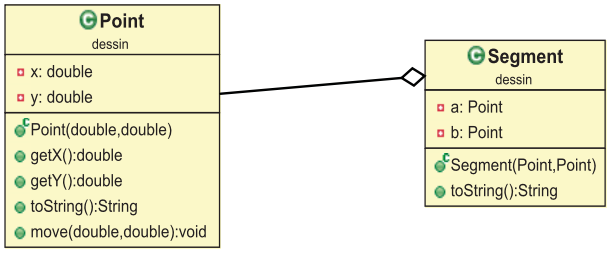
```
1  public class Segment{
2    private Point a, b; // simple déclaration
3
4    public Segment(Point a, Point b) {
5      this.a = a;
6      this.b = b;
7    }
8    public String toString() {
9      return "Segment [a=" + a + ", b=" + b + "]";
10     // note " "+a <=> " "+a.toString() => implicite en JAVA
11   }
12   public void move(double dx, double dy) {
13      a.move(dx, dy); // vision public du Point
14      b.move(dx, dy);
15   }
16 }
```


Représentation des liens UML

```
1 public class Segment{
2   private Point a,b;
3   ...
```

Deux représentations usuelles :

1) Lien d'agrégation : Un segment **est composé de** Point(s)



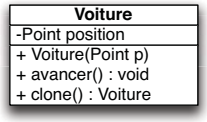
2) Lien d'utilisation :

- Le segment **utilise** un point en attribut privé nommé a
- Le segment **utilise** un point en attribut privé nommé b

Clonage d'objet composé : LE PIEGE

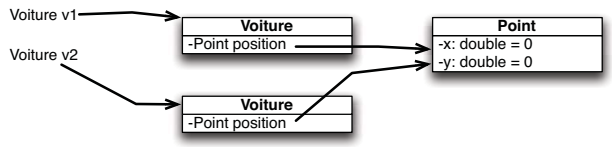
Cas classique : besoin de dupliquer une `Voiture` dont la position est définie par un attribut `Point`

Proposition :




```
1 // Dans voiture
2 Voiture clone(){
3   return new Voiture(position);
4 }
5 // dans le main
6 Voiture v1 = new Voiture(new Point(0,0));
7 Voiture v2 = v1.clone();
```

⚠ GROS PROBLEME !!



Il y a deux instances de Voiture, mais une seule position... Si l'une bouge, l'autre aussi (on va avoir l'impression qu'elle s'est téléporté)

Solution :

Egalité structurelle : attention au equals

- Structure standard classique...
- jusqu'au moment du test sur les attributs :

  penser au `equals` (au lieu de ==)

```
1    public boolean equals(Object obj) {
2        if (this == obj) return true;
3        if (obj == null) return false;
4        if (getClass() != obj.getClass())
5          return false;
6        Voiture other = (Voiture) obj;
7        if (! position.equals(obj.position)) return false;
8        return true;
9    }
```