2i002 - Programmation par objets en Java

Christophe Marsala

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

UPMC SORBONNE UNIVERSITÉS

Cours 5 - 14 février 2017

Programme du jour

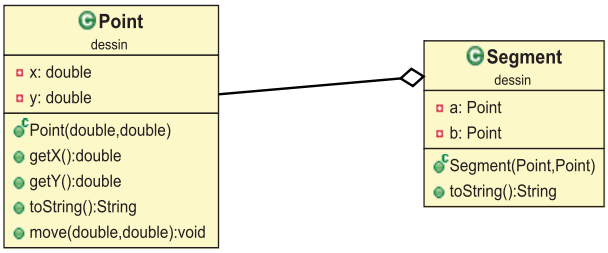
1. Objets complexes, composition d'objets (suite)
2. Tableaux

Plan du cours

Représentation des liens UML

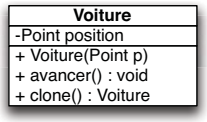
```java
public class Segment{
  private Point a,b;
  ...
```

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



Clonage d'objet composé

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

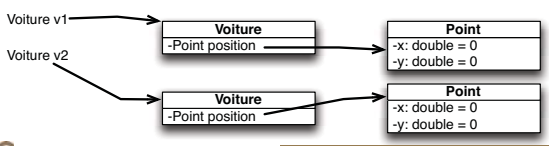


Proposition :

```java
// Dans voiture
Voiture clone(){
  return new Voiture(position);
}
// dans le main
Voiture v1 = new Voiture(new Point(0,0));
Voiture v2 = v1.clone();
```

Solution :

```java
// Dans voiture
Voiture clone(){ // clonage recursif
  return new Voiture(position.clone());
}
```



Egalité structurelle : attention au equals

- Structure standard classique...
- jusqu'au moment du test sur les attributs :
  penser au equals (au lieu de ==)

```java
  public boolean equals(Object obj) {
      if (this == obj) return true;
      if (obj == null) return false;
      if (getClass() != obj.getClass())
        return false;
      Voiture other = (Voiture) obj;
      if (! position.equals(obj.position)) return false;
      return true;
  }
```

Plan du cours

1. Objets complexes, composition d'objets (suite)
2. Tableaux

Structure de données

La **structure de base** de la programmation impérative : disponible sur **les types de base et sur les objets**.

1. Tableau à taille fixe
   - \+ Economie mémoire
   - \+ Rapidité d'accès
   - – Peu flexible (taille fixe !)
2. Tableau à taille variable
   - – Gourmand en mémoire
   - – (Un peu) moins rapide
   - \+ Très flexible

[Taille fixe] Syntaxe

- Déclaration : *type* [] nomVariable
- Instanciation : nomVariable = new *type* [*taille*]
- Accès à la case *i* (lecture ou écriture) : nomVariable[i]
- Accès à la longueur du tableau : nomVariable.length

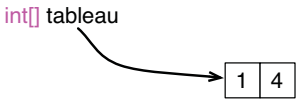
```
1  class TableauA {
2    public static void main(String[] argv) {
3      int[] tableau; // déclaration
4
5      tableau = new int[2]; // instanciation
6      tableau[0] = 1; // utilisation (écriture)
7      tableau[1] = 4;
8
9      int i = tableau[0]; // utilisation en lecture
10     // accès à la longueur du tableau
11     System.out.println("Longueur : "+tableau.length);
12   }
13 }
```

- tableau est une variable de type int[] (ie tableau d'entiers)
- tableau[i] : chaque case de tableau est de type int

Représentation mémoire

```
1  int[] tableau = new int[2];
2  tableau[0] = 1;
3  tableau[1] = 4;
```

tableau = création d'un **ensemble de variables**...
... Facilement accessibles dans les boucles.



Attention : bien différentier variables et instances...

- instanciation d'un tableau = création de **variables**
- créer les **instances** dans un second temps
- ⇒ passage aux objets (un peu) piégeux

Tableau d'objets

Soit la classe Point (vue dans les cours précédent)

Déclaration d'une variable tabP de type Point[] (tableau de points)

⚠ Le tableau n'existe pas ! (il n'est **pas instancié**)

La variable tabP référence un tableau de 10 cases

⚠ **10 cases = 10 variables**...
... 0 instances de Point !

Variantes de syntaxe

Création Syntaxe simplifiée : {value,value,...}
⚠ Ne marche que sur la ligne de déclaration

```
1  boolean[] tableau={true,false,true};
```

Création Syntaxe intermédiaire (marche partout) : new *type* [] {value,value,...}

```
1  boolean[] tableau;
2  tableau = new boolean[]{true,false,true};
```

Boucle Parcours des éléments du tableau (sans référence d'indice) : for(*type* var : nomTableau) ...
var prend successivement toutes les valeurs des éléments du tableau

```
1  for(boolean b: tableau) // affichage de tous les éléments
2    System.out.println(b);
```

⚠ Pas de référence aux indices : usage possible, ou pas, en fonction des algorithmes

Tableaux et boucles

Code robuste = pas de duplication de l'information

Attention aux conditions de fin de boucles

```
int[] tab = {2, 3, 4, 5, 6};
```

Besoin de faire une boucle...

```
for(int i=0; i<5; i++) // FAUX dans le cadre de 2i002
    ...
```

length

La taille du tableau `tab` est définie lors de la **création** (implicitement ou explicitement).
Utiliser `tab.length` pour y faire référence

```
// OK : modifier le tableau = modifier la boucle !
for(int i=0; i<tab.length; i++)
    ...
```


[Tableau dynamique] ArrayList

Usage dans 2 cas (imbriqués) :

- **Taille finale inconnue** lorsque l'on commence à utiliser le tableau (e.g. lecture d'un fichier...)
- **Taille variable** en cours d'utilisation (e.g. pile d'objets à traiter de taille variable)

- Objet JAVA à déclarer avant utilisation :

```
import java.util.ArrayList; // en entête
```

- Syntaxe objet classique + approche générique (hors prog.) :
  - la variable sera de type : `ArrayList<type>`
  - *type* est forcément un objet (≠ type de base) : `Integer`, `Double`, `Point`...
- Même représentation mémoire que les tableaux de taille fixe

ArrayList : syntaxe détaillée

- **Exemple** sur un tableau de `Point`
- **Méthodes principales** : constructeur, `add`, `get`, `remove`, `size`
- Plus d'informations sur la javadoc (beaucoup de d'autres méthodes disponibles) :
  `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`

```
// Construction comme un objet classique
ArrayList<Point> tabArr = new ArrayList<Point>();
tabArr.add(new Point(1,2)); // ajout
for(int i=0; i<9; i++) tabArr.add(new Point(i,i));

// accesseur sur le deuxième élément (index = 1)
Point p = tabArr.get(1);
tabArr.remove(0); // suppression du premier élément

// accesseur sur la taille courante
System.out.println(tabArr.size());
```


Sortie de tableau [fixe ou dynamique]

⚠ Tableau... ⇒ possibilité de sortir du tableau

- Cas classique :
  - Mélange entre taille n et dernier indice du tableau $(n-1)$
  - Tentative d'accès à un index négatif
  - Erreur de boucle...
- Symptôme : `ArrayIndexOutOfBoundsException`
  - Echec lors de l'exécution du code (compilation OK)

```
Point[] tab = {new Point(), new Point()};
System.out.println(tab[2]);
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
at test.Point.main(Point.java:118)
```

- Attention aux `NullPointerException` : après instanciation d'un tableau, aucune instance n'est disponible :

```
Point[] tab = new Point[2];
System.out.println(tab[0].getX()); // => NullPointerException
```


Tableau à deux dimensions

Comment gérer les matrices ?
Comme **des tableaux de tableaux**

- Déclaration des variables : type[] []

```
int[][] matrice;
```

- Instanciation

```
matrice = new int[2][3]; // 2 lignes, 3 colonnes
```

- Usage

```
matrice[0][0] = 0; matrice[0][1] = 1; matrice[0][2] = 2;
matrice[1][0] = 3; matrice[1][1] = 4; matrice[1][2] = 5;
```

- Syntaxe alternative d'instanciation/initialisation

```
int[][] matrice = {{0, 1, 2},{3, 4, 5}}
```

- Accès aux dimensions :

```
matrice.length // nb lignes
matrice[0].length // nb de colonnes de la première ligne
```


Tableau à deux dimensions : vision avancée

```
Point[][] matP = new Point[2][3];
// est équivalent à :
//     création d'un tableau de tableau
//     de taille 2
Point[][] matP2 = new Point[2][];
//  création de 2 tableaux de 3 cases
for(int i=0; i<matP2.length; i++)
  matP2[i] = new Point[3];
```

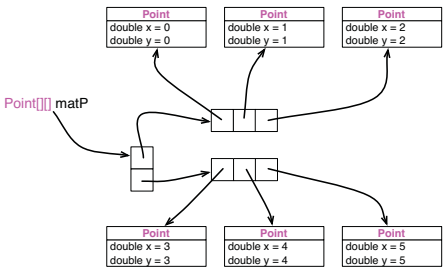
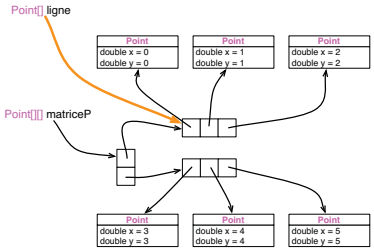
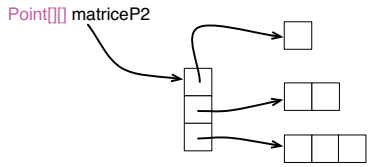


Tableau à deux dimensions : vision avancée (2)





Remarque sur la conversion

- Conversion sur les types de base : OK

```
1 double d = 2.4;
2 int i = (int) d; // conversion explicite
```

- Conversion sur les tableaux : KO, impossible

```
1 double[] tab = {2.5, 5, 8.};
2 // aucune conversion possible
3 // seule option: reconstruction complète:
4 int[] tabInt = new int[tab.length];
5 for(int i=0; i<tab.length; i++)
6   tabInt[i] = (int) tab[i];
```

Même fonctionnement avec les tableaux ou les `ArrayList`

Tableau = ensemble de variables

Pas de flexibilité à ce niveau là...

... A voir avec l'héritage