2I002
UML, REPRÉSENTATION DES OBJETS

Vincent Guigue & Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

LIP6

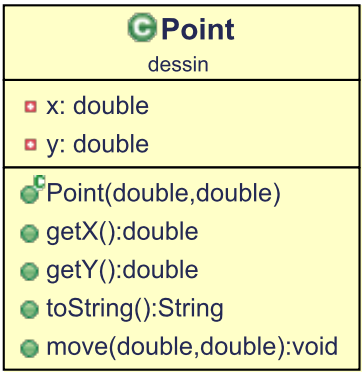
MLIA Machine Learning & Deep Learning for Information Access

On ne programme pas pour soi-même... Mais pour les autres:

- Respecter les codes syntaxiques : majuscules, minuscules...
- Donner des noms explicites (classes, méthodes, attributs)
- Développer une documentation du code (cf cours javadoc)
- ... Et proposer une vision synthétique d'un ensemble de classes: $\Rightarrow$ **UML**

On ne programme pas pour soi-même... Mais pour les autres:

- Respecter les codes syntaxiques : majuscules, minuscules...
- Donner des noms explicites (classes, méthodes, attributs)
- Développer une documentation du code (cf cours javadoc)
- ... Et proposer une vision synthétique d'un ensemble de classes: ⇒ **UML**

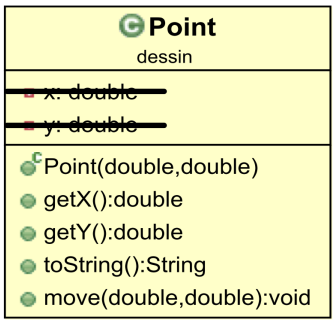


- nom de la classe
- attributs
- méthodes (et constructeurs)

\+ code pour visualiser `public`/`private`
\+ liens entre classes pour les dépendances (cf cours sur la composition)

Plusieurs types de diagrammes pour plusieurs usages:

- Pour les développeurs: représentation complète
- Pour les utilisateurs: représentation `public` uniquement



Le code doit être pensé pour les autres:

- Tous les noms doivent être aussi clairs que possible
- Un diagramme plus limité est plus facile à lire

2 manières de voir l'UML:

1. Outil pour une visualisation globale d'un code complexe
2. Outil de conception / développement indépendant du langage

Dans le cadre de 2i002: seulement l'approche 1

Limites de l'UML:

- Vision architecte...
- Mais pas d'analyse de l'exécution du code

Que se passe-t-il lors de l'exécution du programme:

Nouveau type de représentation:

Diagramme mémoire

```
1 Point p1 = new Point(1,2);
```

Comment décrire cette ligne de code?

```
1 Point p1 = new Point(1,2);
```

Comment décrire cette ligne de code?

La variable p, de type Point, référence une instance dont les attributs x et y ont pour valeur respective 1 et 2.

Comment représenter cette ligne de code?

```
1 Point p1 = new Point(1,2);
```

Comment décrire cette ligne de code?

La variable p, de type Point, référence une instance dont les attributs x et y ont pour valeur respective 1 et 2.

Comment représenter cette ligne de code?

- Représentation des classes sans les méthodes
- Valeur des attributs
- Type & noms des variables
- Lien de référencement

```
1 Point p1 = new Point(1,2);
```

Comment décrire cette ligne de code?

La variable p, de type Point, référence une instance dont les attributs x et y ont pour valeur respective 1 et 2.

Comment représenter cette ligne de code?

- Représentation des classes sans les méthodes
- Valeur des attributs
- Type & noms des variables
- Lien de référencement

$\Rightarrow$ les lignes de code s'enchainent: on raye, on trace des nouveaux liens etc... On représente ce qui se passe au cours de l'exécution du code.

```
1 p1.move(0,1); // en imaginant que la methode existe
```

Les types de base et les objets ne se comportent pas de la même façon avec =

- Liste des types de base:
  `int, double, boolean, char, byte, short, long, float`

```
1 double a, b;
2 a = 1;
3 b = a; // duplication de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

Les types de base et les objets ne se comportent pas de la même façon avec =

- Liste des types de base:

  `int, double, boolean, char, byte, short, long, float`

```
1 double a, b;
2 a = 1;
3 b = a; // duplication de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

- et pour un Objet :

```
4 Point p = new Point(1,2);
5 Point q = p; // duplication de la reference...
6              // 1 seule instance !
```

Les types de base et les objets ne se comportent pas de la même façon avec =

- Liste des types de base:

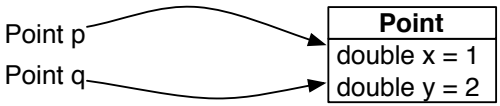
  `int, double, boolean, char, byte, short, long, float`

```
1 double a, b;
2 a = 1;
3 b = a; // duplication de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

- et pour un Objet :

```
4 Point p = new Point(1,2);
5 Point q = p; // duplication de la reference...
6              // 1 seule instance !
```

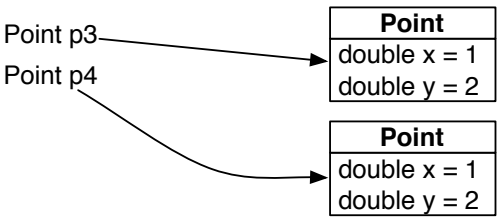


2 variables, 2 références, mais 1 seule instance

Clonage *vs* copie de surface

```java
public static void main(String[] args) {
  Point p1 = new Point(1, 2);
  Point p2 = p1;

  Point p3 = new Point(1, 2);
  Point p4 = new Point(1, 2);
}
```



- Les variables p1 et p2 référencent la même instance
- p3 et p4 référencent des instances différentes

- Passer un argument à une fonction revient à utiliser un signe =
- ... Objets et types de base se comportent différemment !

```java
// classe UnObjet,
// (classe sans importance)
...
public void maFonction1(Point p){
  ...
  p.move(1., 1.);
  ...
}

public void maFonction2(double d){
  ...
  d = 3.; // syntaxe correcte
          // mais tres moche !
  ...
}
```

```java
// dans le main
UnObjet obj = new UnObjet();

Point p = new Point(1.,2.);
double d = 2.;

obj.maFonction1(p);
obj.maFonction2(d);

// p a pour attributs (x=2.,y=3.)
// d vaut 2
```

- Quand un objet est passé en argument:
  il n'y a pas duplication de l'instance (simplement 2 références vers 1 instance)
- Quand un type de base est passé en argument: duplication.

Types de base *vs* Objet : signification de ==

- Opérateur == : prend 2 opérandes de **même type** et retourne un `boolean`
- Type de base : vérification de l'égalité **des valeurs**

```
1    double d1 = 1.;
2    double d2 = 1.;
3    System.out.println(d1==d2); // affichage de true
4                                //dans la console
```

Types de base *vs* Objet : signification de ==

- Opérateur == : prend 2 opérandes de **même type** et retourne un `boolean`
- Type de base : vérification de l'égalité **des valeurs**
- Objet : vérification de l'égalité **des références**

```
1   double d1 = 1.;
2   double d2 = 1.;
3   System.out.println(d1==d2); // affichage de true
4                               //dans la console
5   Point p1 = new Point(1, 2);
6   Point p2 = p1;
7   System.out.println(p1==p2); // affichage de true
8
9   Point p3 = new Point(1, 2);
10  Point p4 = new Point(1, 2);
11  System.out.println(p3==p4); // affichage de false
```

Types de base *vs* Objet : signification de ==

- Opérateur == : prend 2 opérandes de **même type** et retourne un `boolean`
- Type de base : vérification de l'égalité **des valeurs**
- Objet : vérification de l'égalité **des références**
- ⚠ ATTENTION aux classes enveloppes (qui sont des objets)

```
1   double d1 = 1.;
2   double d2 = 1.;
3   System.out.println(d1==d2); // affichage de true
4                               //dans la console
5   Point p1 = new Point(1, 2);
6   Point p2 = p1;
7   System.out.println(p1==p2); // affichage de true
8
9   Point p3 = new Point(1, 2);
10  Point p4 = new Point(1, 2);
11  System.out.println(p3==p4); // affichage de false
12  Double d3  = 1.; // classe enveloppe Double = objet
13  Double d4 = 1.;
14  System.out.println(d3==d4); // affichage de false
```

Les types de base en JAVA sont doublés de *wrappers* ou classes enveloppes pour:

- utiliser les classes génériques (cf cours `ArrayList`)
- fournir quelques outils fort utiles

`int, double, boolean, char, byte, short, long, float` ⇒ `Integer, Double...`

Outils

```
1 Double d1 = Double.MAX_VALUE; // valeur maximum possible
2 Double d2 = Double.POSITIVE_INFINITY; // valeur specifique
3                                       // geree dans les operations
4 Double d3  = Double.valueof("3.5"); // String => double
5 // Double.isNaN(double d), Double.isInfinite(double d)...
```

Documentation: `http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html`

Les types de base en JAVA sont doublés de *wrappers* ou classes enveloppes pour:

- utiliser les classes génériques (cf cours `ArrayList`)
- fournir quelques outils fort utiles

`int, double, boolean, char, byte, short, long, float` ⇒ `Integer, Double...`

Outils

```
1 Double d1 = Double.MAX_VALUE; // valeur maximum possible
2 Double d2 = Double.POSITIVE_INFINITY; // valeur specifique
3                                       // geree dans les operations
4 Double d3  = Double.valueof("3.5"); // String => double
5 // Double.isNaN(double d), Double.isInfinite(double d)...
```

Documentation: `http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html`

```
6 // conversions implicites = (un)boxing (depuis JAVA 5)
7 double d4 = d1;
8 Double d5 = d4;
```

Classe enveloppe: des objets avant tout

Historique:

- *A l'origine:* Philosophie tout objet

  $\Rightarrow$ propre, mais pas pratique

- *Evolution:* plus de dérogation pour faciliter les usages

  $\Rightarrow$ classes enveloppes bcp moins utilisées

$\Rightarrow$ Classe enveloppe = objet !!!

```
1        double d1b = 1.;
2        double d2b = d1b;
3        double d3b = 1.;
4        System.out.println(d1b == d2b);
5        System.out.println(d1b == d3b);
```

Classe enveloppe: des objets avant tout

Historique:

- *A l'origine:* Philosophie tout objet

  $\Rightarrow$ propre, mais pas pratique

- *Evolution:* plus de dérogation pour faciliter les usages

  $\Rightarrow$ classes enveloppes bcp moins utilisées

$\Rightarrow$ Classe enveloppe = objet !!!

```
1        double d1b = 1.;
2        double d2b = d1b;
3        double d3b = 1.;
4        System.out.println(d1b == d2b);
5        System.out.println(d1b == d3b);
6        // true & true
```

Historique:

- *A l'origine:* Philosophie tout objet

  $\Rightarrow$ propre, mais pas pratique

- *Evolution:* plus de dérogation pour faciliter les usages

  $\Rightarrow$ classes enveloppes bcp moins utilisées

$\Rightarrow$ Classe enveloppe = objet !!!

```
1         double d1b = 1.;
2         double d2b = d1b;
3         double d3b = 1.;
4         System.out.println(d1b == d2b);
5         System.out.println(d1b == d3b);
6         // true & true
7         Double d1  = 1.;
8         Double d2  = d1;
9         Double d3  = 1.;
10        System.out.println(d1 == d2);
11        System.out.println(d1 == d3);
```

Classe enveloppe: des objets avant tout

Historique:

- *A l'origine:* Philosophie tout objet

  $\Rightarrow$ propre, mais pas pratique

- *Evolution:* plus de dérogation pour faciliter les usages

  $\Rightarrow$ classes enveloppes bcp moins utilisées

$\Rightarrow$ Classe enveloppe = objet !!!

```
1         double d1b = 1.;
2         double d2b = d1b;
3         double d3b = 1.;
4         System.out.println(d1b == d2b);
5         System.out.println(d1b == d3b);
6         // true & true
7         Double d1  = 1.;
8         Double d2  = d1;
9         Double d3  = 1.;
10        System.out.println(d1 == d2);
11        System.out.println(d1 == d3);
12        // true & false
```

Pointeur `null`

Que se passe-t-il quand on déclare une variable (sans l'instancier)?

```
1 Point p;
```

- p vaut `null`.
- On peut écrire de manière équivalente :

```
1 Point p = null;
```

Que se passe-t-il quand on déclare une variable (sans l'instancier)?

```
1 Point p;
```

- p vaut `null`.
- On peut écrire de manière équivalente :

```
1 Point p = null;
```

- On ne peut pas invoquer de méthode

```
1 p.move(1., 2.); // => CRASH de l'execution :
2                 // NullPointerException
```

Pointeur `null`

Que se passe-t-il quand on déclare une variable (sans l'instancier)?

```
1 Point p;
```

- p vaut `null`.
- On peut écrire de manière équivalente :

```
1 Point p = null;
```

- On ne peut pas invoquer de méthode

```
1 p.move(1., 2.); // => CRASH de l'execution :
2                 // NullPointerException
```

- N'importe quel objet peut être `null` et réciproquement, on peut donner `null` à n'importe quel endroit où un objet est attendu... Même si ça provoque parfois des crashs.

```
1 // classe UnObjet,
2 // (classe sans importance)
3 ...
4 public void maFonction(Point p){
5   ...
6   p.move(1., 1.);
7   ...
8 }
```

```
1 // dans le main
2 UnObjet obj = new UnObjet();
3
4 obj.maFonction(null);
```

Un truc classique, mais catastrophique...

```java
public class Vecteur{
  // un peu en avance
  // sur la composition...
  private Point a, b;
  public Vecteur(Point p1, Point p2){
    Point a = p1; Point b=p2;
  }
  public void move(double d){
    a.move(d); b.move(d);
  }
```

```java
public class TestVecteur{
  public static
  void main(String[] args){
    Point p = new Point(1., 2.);
    Point p2 = new Point(3., 5.);
    Vecteur v = new Vecteur(p, p2);

    v.move(3);
  }
}
```

Un truc classique, mais catastrophique...

```
1  public class Vecteur{
2    // un peu en avance
3    // sur la composition...
4    private Point a, b;
5    public Vecteur(Point p1, Point p2){
6      Point a = p1; Point b=p2;
7    }
8    public void move(double d){
9      a.move(d); b.move(d);
10   }
```

```
1  public class TestVecteur{
2    public static
3    void main(String[] args){
4      Point p = new Point(1., 2.);
5      Point p2 = new Point(3., 5.);
6      Vecteur v = new Vecteur(p, p2);
7
8      v.move(3);
9    }
10 }
```

- Crash `NullPointerException` dans Vecteur à la ligne 9
- Venant de `main` à la ligne 8

Une méthode est exécutée sur une instance...

```
1 Point p = new Point(1,2);
2 Point p2 = p;
3 p.move(1.,1.);
4 // methode executee sur l'instance
5 // (qui est referencee par p et p2)
6 System.out.println(p2);
7 // [x=2., y=3.]
```

$\Rightarrow$ Il faut toujours se représenter ce qui se passe dans la mémoire lors de l'exécution d'un programme.