2i002 - Programmation par objets en Java

Christophe Marsala

(support réalisé par Vincent Guigue)

Cours 2

Programme du jour

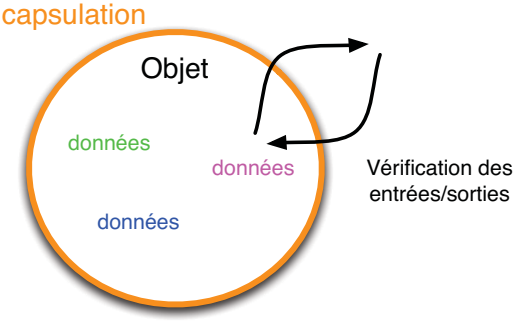
Informations

- Christophe Marsala (email : `Christophe.Marsala@lip6.fr`)
- Transparents de cours sur la page web de l'UE :
- Page web de l'UE sur le site de la licence :

https ://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2016/ue/2I002-2017fev/

Plan du cours

Barrière de sécurisation
=
encapsulation



Barrière de sécurisation
=
encapsulation

Syntaxe : surcharge du constructeur

Comment construire un `Point` ? ... de plusieurs manières !

Ex :
- 2 valeurs à fournir : le plus classique
- 0 valeur : génération aléatoire de x et y
- 1 valeur : affectation de la même valeur pour x et y

⇒ Syntaxe triviale : il suffit de définir plusieurs constructeurs !

CONTRAINTE : signatures toutes différentes

Fournisseur

```
1  //Fichier Point.java
2  public class Point{
3    private double x,y;
4
5    public Point(double x2, double y2){
6      x = x2;
7      y = y2;
8    }
9    public Point(double d){ // surcharge
10     x = d;
11     y = d;
12   }
13   public Point(){ // autre surcharge
14     // aléatoire entre 0 et 10
15     x = Math.random()*10;
16     y = Math.random()*10;
17   }
```

Client

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main
4                      (String[] args){
5
6      // construction d'un point:
7      Point p = new Point(2., 3.1);
8      // construction d'un autre point:
9      Point p2 = new Point();
10     // construction d'un 3e point:
11     Point p3 = new Point(4.2);
12
13   }
14 }
```

Syntaxe : méthodes standards

- Des méthodes standards sont disponibles pour tous les objets (cf cours héritage)...
  - mais avec un comportement pas toujours satisfaisant
- Ex : conversion d'un objet en chaîne de caractères `public String toString()`

Fournisseur

```java
//Fichier Point.java
public class Point{
 ...
 public String toString(){
    return "["+ x +","+ y +"]";
 }
}
```

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p = new Point(2., 3.1);

    String str = p.toString();

    System.out.println("p:" + str);
  }
}
```

```
» p: Point@8764152
```

```
» p: [2, 3.1]
```


Reflexion sur la syntaxe objet

Exemple type : addition de 2 `Point`

Réfléchir à la signature d'une méthode `add` permettant d'additionner 2 instances de `Point` (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des attributs des opérandes)

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p  = new Point(2., 3.1);
    Point p2 = new Point(0.5, 1);

    Point p3 = p.add(p2);
  }
}
```

Fournisseur

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

 public Point add(Point p){
   return new Point(x+p.x, y+p.y);
 }
```

Syntaxe objet = un truc à prendre... Pas évident au début !

Surcharge

Définition

- Même nom de fonction, arguments différents
- Le type de retour ne compte pas

```java
public class Point {
...
    public void move(double dx, double dy){
        x+=dx; y+=dy;
    }
    public void move(double dx, double dy, double scale){
        x+=dx*scale; y+=dy*scale;
    }
    public void move(int dx, int dy){
        x+=dx; y+=dy;
    }
    public void move(Point p){
        x+=p.x; y+=p.y;
    }
```

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

Compilation/exécution

Nous avions vu précédemment comment compiler et executer UNE classe... Comment faire maintenant qu'il y en a plusieurs ?

```
» javac Point.java
» javac TestPoint.java
» java TestPoint
```

Syntaxe réduite :

```
» javac Point.java TestPoint.java OU javac *.java
» java TestPoint
```

Remarque : il peut y avoir plusieurs `main` (mais pas plus de 1 par classe)

- Compilation de tous les `main` d'un coup
- Execution d'un seul (appel à la classe correspondante)

Cas ambigus : this.

Distinguer un argument de méthode et un attribut qui portent le même nom

Exemple le plus classique : le constructeur

```java
public class Point{
 private double x,y; // attributs
 public Point(double x, double y){ // arguments du constructeur
   // distinguer l'attribut et l'argument
   //pour faire l'affectation dans le bon sens

   this.x = x; // utiliser l'argument pour init. l'attribut
   this.y = y;
 }
}
```

Vous avez toujours le droit d'utiliser la syntaxe `this.attr` pour désigner l'attribut `attr` dans la classe (même dans les cas non ambigus)

Vous pouvez utiliser `this.maMethode()` pour invoquer `maMethode` lorsque vous êtes dans une autre méthode de l'objet.

Constructeur multiple : usage du this()

Approche standard pour écrire plusieurs constructeurs dans une classe :

1. Ecrire le constructeur général, prenant le plus d'arguments
2. Appeler le constructeur 1 avec des arguments spécifiques

⇒ éviter les copier-coller, fiabiliser le code

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

 public Point(double x, double y){ // constructeur 1
   this.x = x;
   this.y = y;
 }

 public Point(){ // constructeur 2
 // ATTENTION : aucun code avant this()
   this(Math.random()*10, Math.random()*10); // invocation
                                             // du constructeur 1
 }
}
```

Plan du cours

Types des variables de base en Java

- Entier, réel, booléen, caractère : ces types sont disponibles de base en Java avec les opérateurs les plus courants.
  `int, double, boolean, char, byte, short, long, float`
  ⚠ La plupart des types et syntaxes associées sont comparables au C/C++... Sauf le booléen.

Le booléen vaut true/false et n'est pas convertible en entier

- Déclaration

```
int i; // déclaration de i
System.out.println(i); // => 0
double d = 2.6;
boolean b = true; // ou false
char c = 'a';
```

```
// operations de base: + - / * ...
int j = i+2;
int k = 1/2; //=0 Attention a la division entiere
```


Classe String

Gestion des chaînes de caractères

String n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à Java et son caractère immutable la rapproche très nettement d'un type de base.

```
String s = "Luke"; // création d'une chaîne de caractères
s = s + " est le frère de Leia";
System.out.println(s); // affichage de s dans la console
```

⚠ Ne pas confondre l'**objet** `String` et l'affichage dans la console.

Les **possibilités sont nombreuses** : extraction de sous-chaînes (substring), division en plusieurs chaînes (split), recherche de caractères, construction de nouvelles chaînes à partir d'expressions régulières (replace)... Toute la documentation sur : `http://docs.oracle.com/javase/7/docs/api/java/lang/String.html`

String (suite)

2 choses à retenir sur les `String`

1. **Les chaînes sont immutables** : modifier une chaîne existante est impossible, il faut créer une nouvelle chaîne qui est une modification de l'ancienne. Cela rend la classe peu efficiente dans certain cas... Et il faut alors se tourner vers des objets plus évolués (StringBuffer notamment)
2. **Ne pas utiliser `==` avec les String** mais toujours la méthode .equals. Les deux versions compilent mais la première donnera régulièrement des résultats faux (que nous expliquerons plus tard).

```
String s1 = "Leia";
String s2 = "Luke";
if( s1.equals(s2) )
    System.out.println("les chaînes sont identiques");
else
    System.out.println("les chaînes sont différentes");
```


Classes enveloppes

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

```
Double d1 = MAX_VALUE; // valeur maximum possible
Double d2 = Double.POSITIVE_INFINITY; // valeur spécifique
                    // gérée dans les opérations
Double d3  = Double.valueof("3.5"); // String => double
// Double.isNaN(double d), Double.isInfinite(double d)...
```

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

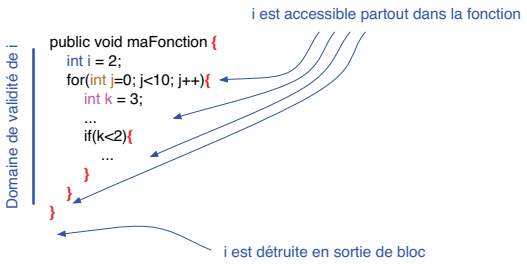
```
// conversions implicites = (un)boxing (depuis JAVA 5)
double d4 = d1;
Double d5 = d4;
```


Durée de vie

Logique de bloc

- une fonction est un bloc,
- une boucle ou une conditionnelle forme également un bloc,
- les blocs sont repérés par des accolades : {...}

Les variables **déclarées** dans un bloc sont détruites en sortant du bloc.



Conversions entre types

Java, un langage typé

Les types sont très importants en Java : le compilateur vérifie toujours les types des différentes variables

- Certaines conversions sont implicites

```
double d = 42; double d2 = i; // avec i un int existant
```

Tout type de base peut se convertir en String

```
String s = "mon message "+1.5+" "+d;
```

- Certaines conversions doivent être données explicitement

```
int i = (int) 2.4;
```

Perte d'information liée à la conversion ; Java ne tolère pas la conversion implicitement, il faut que le programmeur la demande explicitement (pour être sûr que la perte d'information est souhaitée).

- Conversions impossibles

Opérateurs classiques (par ordre de priorité)

opérateurs postfixés	[ ] . expr++ expr--
opérateurs unaires	++expr --expr +expr -expr ~ !
création ou cast	new (type) expr
opérateurs multiplicatifs	* / %
opérateurs additifs	+ -
décalages	<< >> >>>
opérateurs relationnels	< > <= >=
opérateurs d'égalité	== !=
et bit à bit	&
ou exclusif bit à bit	^
ou (inclusif) bit à bit	|
et logique	&&
ou logique	||
opérateur conditionnel	? :
affectations	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Conditionnelles

- Syntaxe de l'alternative

```
int i=11;
if(i > 38){
    // code à effectuer dans ce cas
}
else{ // le else est facultatif
    // Code à effectuer sinon
}
```

- En cas de clauses multiples :

```
switch(i){
case 1:
   // Code à effectuer si i == 1
   break; // sinon le reste du code est AUSSI effectué
case 2: //
    // Code à effectuer si i == 2
   break;
default : // Si on n'est passé nulle part ailleurs
}
```


Structures itératives

Même définition des boucles qu'en C/C++

- Syntaxes : 2 options (principales)

Pour i allant de 0 à 9, faire...

```
int i;
for(i=0; i<10; i++){// i prend les valeurs 0 à 9
                    // ==> 10 itérations
// code a effectuer 10 fois
}
```

Tant que i inférieur à 10, faire...

```
int i = 0;
while(i<10){// i prend les valeurs 0 à 9 =
                    // 10 itérations
    // code a effectuer 10 fois
    i++; // ne pas oublier, sinon boucle infinie !
}
```

- D'autres syntaxes sont possibles : do...while etc...

Interruptions de fonctions/boucles (1/3)

Trois types d'interruptions de boucles

- return : l'interruption **la plus forte**. Retour anticipé de la fonction (**sort de la fonction**, pas seulement de la boucle).

```
// le modulo par 5 peut-il retourner un entier >=5?
public void maFonction(){
    for(int i=0; i<10; i++){
        if(i%5>4){
            System.out.println("C'est très étrange");
            return;
        }
    }
    System.out.println("l'opération modulo 5 retourne"+
       "toujours un entier inférieur à 5");
}
```


Interruptions de fonctions/boucles (2/3)

3 types d'interruptions de boucles

- return
- break : sortie anticipée de la boucle

```
// 6 fait-il parti des multiples de 2?
public void maFonction(){
    boolean found = true;
    for(int i=0; i<10; i++){
        if(i * 2 == 6){
            found  = true;
            break; // pas besoin d'aller plus loin
        }
    }
    if(found)
        System.out.println("6 est un des multiples de 2");
}
```

Interruptions de fonctions/boucles (3/3)

3 types d'interruptions de boucles

- return
- break
- continue : passer à l'itération suivante

```java
// afficher 3./i pour i variant de -10 à 10
// il faut penser à sauter le cas 0 qui provoque un problème
public void maFonction(){
    for(int i=-10; i<=10; i++){// -10 et 10 inclus
        if(i == 0)
            continue;
        System.out.println("3./"+i+" = "+(3./i) );
    }
}
```

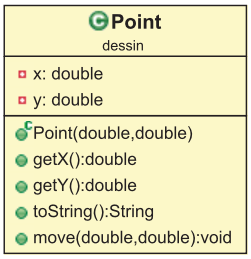
Ces instructions rendent le code plus lisible en limitant notamment le nombre de blocs imbriqués.

Plan du cours

Philosophie

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** : Unified Modeling Language



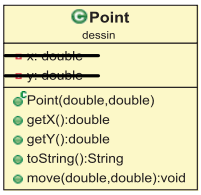
- nom de la classe
- attributs
- méthodes (et constructeurs)

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

UML Client *vs* Développeur

Plusieurs types de diagrammes pour plusieurs usages :

- Vue développeurs : représentation complète
- Vue clients : représentation `public` uniquement



Idée :

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

Philosophie (suite)

Deux 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

Coté JVM : exécution du code

```java
Point p = 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 respectivement 1 et 2.

Comment représenter cette ligne de code ?

- Représentation des classes sans les méthodes
- Valeurs des attributs
- Types & noms des variables
- Liens de référencement

Types de base *vs* Objet : signification de =

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

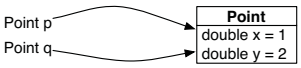
- Liste des types de base :
  `int, double, boolean, char, byte, short, long, float`

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

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

- et pour un Objet :

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

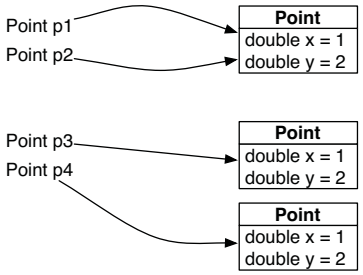


2 variables, 2 références, mais 1 seule instance

Clonage *vs* copie de surface

```
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

Références & arguments de fonctions

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

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

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

```
// 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**
- Objet : vérification de l'égalité **des références**
- ⚠ ATTENTION aux classes enveloppes (qui sont des objets)

```
double d1 = 1.;
double d2 = 1.;
System.out.println(d1==d2); // affichage de true
                            //dans la console
Point p1 = new Point(1, 2);
Point p2 = p1;
System.out.println(p1==p2); // affichage de true

Point p3 = new Point(1, 2);
Point p4 = new Point(1, 2);
System.out.println(p3==p4); // affichage de false
Double d3  = 1.; // classe enveloppe Double = objet
Double d4 = 1.;
System.out.println(d3==d4); // affichage de false
```


Référence null

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

```
Point p;
```

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

```
Point p = null;
```

- On ne peut pas invoquer de méthode

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

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

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

obj.maFonction(null);
```

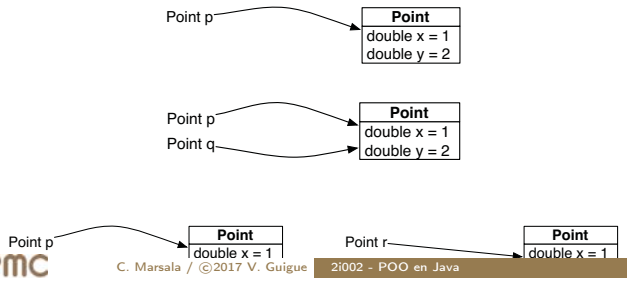

Plan du cours

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)

```
1 // Dans la classe Point
2 public boolean egalite(Point p){
3   return p.x == x && p.y == y;
4 }
5   // equivalent simplifié de
6   // if(p.x == x && p.y == y)
7   //   return true
8   // else
9   //   return false;
```

```
1 // Dans le main
2
3 Point p1 = new Point(1.,2.);
4 Point p2 = p1;
5 Point p3 = new Point(1.,2.);
6 Point p4 = new Point(1.,3.);
7
8 p1.egalite(p2); // true
9 p1.egalite(p3); // true
10 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)`

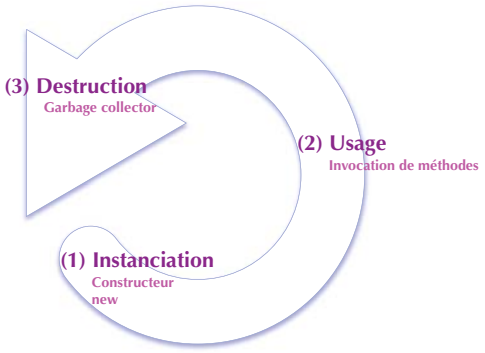
- `equals` existe dans tous les objets (comme `toString`)
  - mais teste l'égalité référentielle... Pas intéressant (comme `toString` en version de base)
- ⇒ Redéfinition : faire en sorte de tester les attributs
  Un processus en plusieurs étapes :
  1. Vérifier s'il y a égalité référentielle / référence 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

```
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         Point other = (Point) obj;
7         if (x != other.x) return false;
8         if (y != other.y) return false;
9         return true;
10    }
```


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;
  }
  ...
}
```

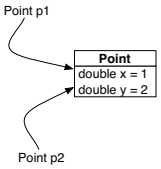
```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
    p1.move(2., 3.);
                // p1 => [x=3, y=5]
  }
}
```


(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...

```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.);
    Point p2 = p1;
  }
}
```

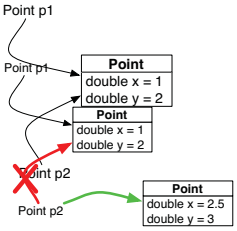


- mais quand est-il dé-référencé ?

Dé-référencement d'un objet

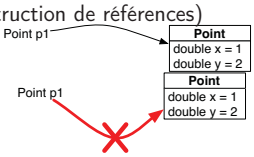
1. Dé-référencement explicite (usage de =

```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.);
    Point p2 = p1;
    p2 = new Point(2.5, 3.);
  }
}
```



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

```java
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**

```java
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 !
}
```

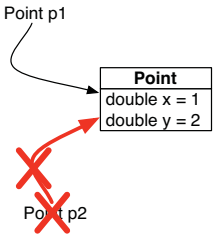
```java
public static
    void main(String[] args) {
  Point p1; // déclaration
            // avant le bloc
  {
      // initialisation de p1
      p1 = new Point(1,2);
      System.out.println(p1);
  } // pas de destruction de p1


  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**

```
1  public static
2      void main(String[] args) {
3    Point p1; // déclaration
4              // avant le bloc
5    {
6        Point p2 = new Point(1,2);
7        // initialisation de p1
8        p1 = p2;
9        System.out.println(p1);
10   } // destruction de p2
11
12   System.out.println(p1);
13   // OK, pas de problème
14 }
```

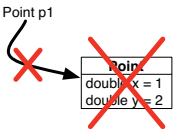


- Fin de bloc = destruction des **variables** déclarées dans le bloc
- Destruction d'instance ⇔ instance plus référencée

Destruction des instances

Destruction d'instance ⇔ instance plus référencée

```
1 public static void main(String[] args) {
2   Point p1 = new Point(1,2);
3   p1 = null;
4 }
```



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

```
1  public static void main(String[] args) {
2  ...
3    for(int i=0; i<10; i++){
4      // optimisation possible:
5      //       réutilisation de la mémoire allouée
6      Point p1 = new Point((int)(Math.random()*10),
7                           (int)(Math.random()*10));
8      ...
9    }
10 }
```

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

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


Le mot de la fin...

... sur un exemple parlant :

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

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

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

Plan du cours

Lapin (interro 2015)

La classe `Lapin` possède :

- deux attributs codant sa position double `posx` et `posy`, et un attribut int `nbEnfants` codant le nombre d'enfants du lapin.
- un constructeur à 2 arguments initialisant les deux attributs de position. `nbEnfants` est toujours initialisé à 0.
- un constructeur sans argument, qui initialise les attributs de position aléatoirement entre -10 et 10 avec `this()`.
- `toString` : génère une chaine de caractère de la forme : [*posx*, *posy*, nbEnfants=...]
- `deplacer` : prend en argument 2 doubles et déplace le lapin en les utilisant,
- `reproduire` : un lapin créé est retourné par la méthode `reproduire` si les lapins sont proches, `null` sinon.
- `estParent` retourne un booléen qui vaut `true` si le lapin a déjà eu des enfants.

Soit les lignes de code suivantes (dans un main) : combien y a-t-il de variables et combien d'instances ?

```
1 Lapin l1 = new Lapin();
2 Lapin l2 = new Lapin();
3 Lapin l3 = l2;
4 l3.deplacer(2, 4);
```

Quelles lignes vous semblent correctes ? Justifier les (4) erreurs en indiquant notamment si elles empêchent la compilation ou l'execution du programme. A quelle position se trouvent les lapins à la fin de ce programme ? Quels sont les affichages dans la console ?

```
1  Lapin l4 = new Lapin(1,2);
2  Lapin l5 = l4;
3  Lapin l6;
4  l6.deplacer(1, 2);
5  Lapin l7 = l6;
6  Lapin l7 = new Lapin(1,3);
7  Lapin bebeLapin = l4.reproduire(l7);
8  System.out.println(bebeLapin.toString());
9  l5.deplacer(2, 4);
10 l6.deplacer(1, 2);
11 System.out.printn("nb Enfants de L4 : "+ l4.nbEnfants);
12 System.out.printn("L4 est-il parent ? : "+ l4.estParent());
```


Complexes et composition

La classe `Complexe` possède :

- deux attributs double `reelle` et `imag`,
- un constructeur à 2 arguments initialisant les deux attributs NB : signature obligatoire : `public Complexe(double reelle, double imag)`,
- un constructeur sans argument, qui initialise les arguments aléatoirement entre -2 et 2 (avec `this()`).
- `toString` qui génère une chaine de caractère de la forme : (*reelle* + *imag* `i`)
- `addition` de deux complexes,
- `multiplication` de deux complexes,
- `estReel` qui teste si le complexe est en fait réel (dans le cas où la partie imaginaire est nulle).
- une méthode :
  `public void translateDeUn(){ reelle+= 1; imag += 1; }`

- Donner le code de la classe `SegmentComplexe` qui contient 2 complexes en attributs. Ajouter un constructeur à deux arguments et des accesseurs.

```
1  Complexe c1 = new Complexe(1,0);
2  Complexe c2 = new Complexe(2,0);
3  Complexe c3;
4  SegmentComplexe s1 =
5     new SegmentComplexe(c1, c2);
6
7  c3 = c1.addition(c2);
8  System.out.println(c3);
9  if(c3.estReel())
10   System.out.println("c3 est reel");
11
12 System.out.println(s1);
13 c1.translateDeUn();
14 System.out.println(s1);
15 c1 = c3;
16 System.out.println(s1);
17
18 Complexe c4;
19 if(c4.estReel())
20   System.out.println("c4 est reel");
```

Ce code contient une erreur (qui empêche l'exécution) : trouver là (après la ligne 10).

Donner les affichages produits lors de l'exécution (en imaginant que le problème précédent a été réglé).

Encore des vecteurs

Relever les erreurs et proposer des solutions pour corriger le code. A chaque affichage, donner ce qui s'affiche (ou ce qui devrait s'afficher une fois le code corrigé). Prédire si les clauses des *if* sont satisfaites ou non.

```
1  Vecteur v1 = new Vecteur(1,2);
2  Vecteur v2 = new Vecteur();
3  System.out.println(v1.toString()+"\n"+v2.toString());
4  Vecteur v3;   v3.x = 5;   v3.y = 7;
5  Vecteur v3 = new Vecteur(5,7);
6  double d = scal(v3);
7  double d = v1.scal(v3);
8  v1.translate(3,4);   v1.afficher();
9  Vecteur v4 = v1.addition(v3);
10 double s = v2.scal(v1);
11 if(s<30)  v1.afficher();
12 else    v2.afficher();
13 v1.x = v1.x + 10;     v1.traslate(10,0);
14 if(v1.getX() < 10 || v2.getX() < 10)
15         System.out.println("On passe ici !");
16 if(v2.getY() < 10 && v3.getX() <= 5)
17         System.out.println("On passe ici aussi !");
18 if(v2.getY()+5 < 10 && v3.getX() > 5 || s<30)
19         System.out.println("On passe ici aussi aussi !");
```