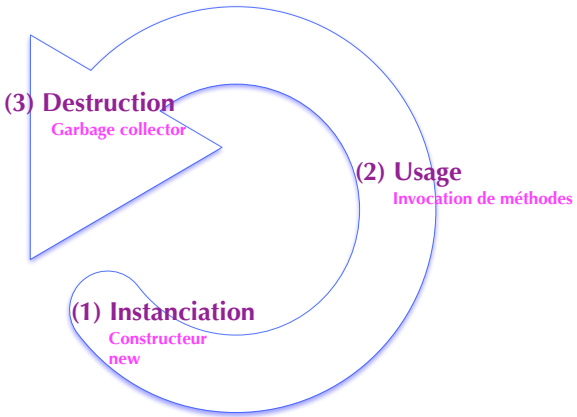
LU2IN002
CYCLE DE VIE DES OBJETS
GABAGE COLLECTOR

Vincent Guigue & Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ
LIP6
MLIA Machine Learning & Deep Learning for Information Access

Se placer du point de vue de l'objet:



(1) création d'une instance

(2) évolution de l'instance

(3) condition de destruction

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.);
  }
}
```

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.

- 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]
  }
}
```

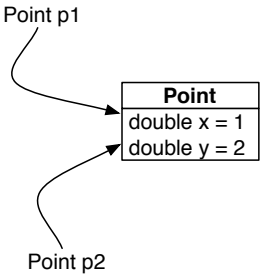
(3) Destruction

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)

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...

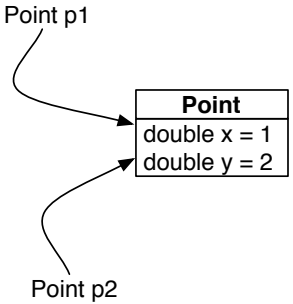
```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
    Point p2 = p1;
  }
}
```



- mais quand est-il dé-référencé?

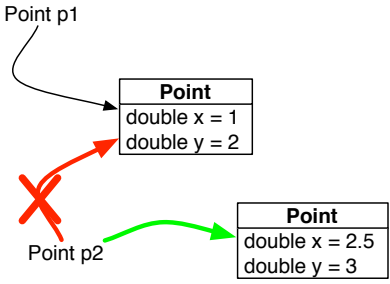
1. Dé-référencement explicite (usage de =)

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



1. Dé-référencement explicite (usage de =)

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main (String[] args){
4      // appel du constructeur avec des valeurs choisies
5      Point p1 = new Point(1., 2.); Point p2 = p1;
6      // Dé-referencement de l'instance [1,2]
7      // et referencement de l'instance [2.5, 3]
8      p2 = new Point(2.5, 3.);
9    }
10 }
```



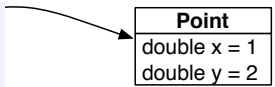
2. Dé-référencement implicite (logique de bloc,
destruction de variables ⇒ destruction de références)

1. Dé-référencement explicite (usage de =)

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main (String[] args){
4      // appel du constructeur avec des valeurs choisies
5      Point p1 = new Point(1., 2.); Point p2 = p1;
6      // Dé-referencement de l'instance [1,2]
7      // et referencement de l'instance [2.5, 3]
8      p2 = new Point(2.5, 3.);
9    }
10 }
```

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

```
1  public static void main(String[] args) {
2    {
3        Point p1 = new Point(1,2);
4        System.out.println(p1);
5    }
```



1. Dé-référencement explicite (usage de =)

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main (String[] args){
4      // appel du constructeur avec des valeurs choisies
5      Point p1 = new Point(1., 2.); Point p2 = p1;
6      // Dé-referencement de l'instance [1,2]
7      // et referencement de l'instance [2.5, 3]
8      p2 = new Point(2.5, 3.);
9    }
10 }
```

2. Dé-référencement implicite (logique de bloc, destruction de variables $\Rightarrow$ destruction de références)

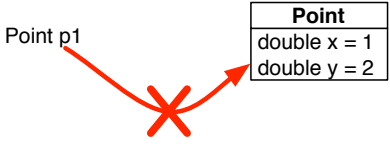
```
1  public static void main(String[] args) {
2    {
3        Point p1 = new Point(1,2);
4        System.out.println(p1);
5    }
6    System.out.println(p1);
```

1 Dé-référencement explicite (usage de =)

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main (String[] args){
4      // appel du constructeur avec des valeurs choisies
5      Point p1 = new Point(1., 2.); Point p2 = p1;
6      // Dé-referencement de l'instance [1,2]
7      // et referencement de l'instance [2.5, 3]
8      p2 = new Point(2.5, 3.);
9    }
10 }
```

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

```
1  public static void main(String[] args) {
2    {
3        Point p1 = new Point(1,2);
4        System.out.println(p1);
5    }
6    System.out.println(p1);
7    // ERREUR DE COMPILATION :p1 n'existe plus ici !
8  }
```



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**

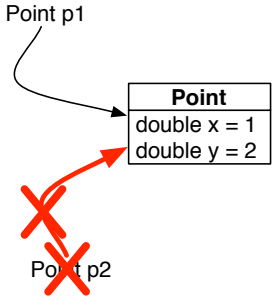
```
1  public static
2      void main(String[] args) {
3
4
5    {
6        Point p1 =
7                new Point(1,2);
8        System.out.println(p1);
9    } // destruction de
10     // la variable p1
11
12   System.out.println(p1);
13   // ERREUR DE COMPILATION
14   // p1 n'existe plus ici !
15 }
```

```
1  public static
2      void main(String[] args) {
3    Point p1; // declaration
4              // avant le bloc
5    {
6        // initialisation de p1
7        p1 = new Point(1,2);
8        System.out.println(p1);
9    } // pas de destruction de p1
10
11
12   System.out.println(p1);
13   // OK, pas de probleme
14
15 }
```

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**

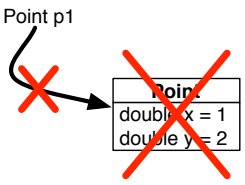
```
1  public static void main(String[] args) {
2    Point p1; // declaration avant le bloc
3    {
4        Point p2 = new Point(1,2);
5        // initialisation de p1
6        p1 = p2;
7        System.out.println(p1);
8    } // destruction de p2
9
10   System.out.println(p1);
11   // OK, pas de probleme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



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

Destruction d'instance ⇔ instance plus référencée

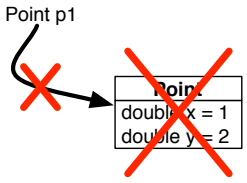
```
1 public static void main(String[] args) {
2   Point p1 = new Point(1,2);
3   p1 = null;
4 }
```



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

Destruction d'instance ⇔ instance plus référencée

```java
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

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

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

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

Exemple complet (1/2)

... sur un exemple parlant:

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

Exemple complet (1/2)

... sur un exemple parlant:

```
1 Point p = new Point(1,2);
2 Point p2 = new Point(3,4);
3 // Point p3 = p; // difference 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.

... sur un exemple parlant:

```
1 Point p = new Point(1,2);
2 Point p2 = new Point(3,4);
3 // Point p3 = p; // difference 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; // difference 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`

Exemple complet (2/2)

```
1    Point p  = new Point(1,1);
2    Point p2 = new Point(2,2);
3    Point p3 = p; Point p4 = p3;
4    p3 = null; p2 = null;
```

- Combien d'instances ont été créées au total?
- Combien d'instances restent-il à la fin de l'exécution du programme?
- Dessiner l'état de la mémoire.

Exemple complet (2/2)

```
1     Point p  = new Point(1,1);
2     Point p2 = new Point(2,2);
3     Point p3 = p; Point p4 = p3;
4     p3 = null; p2 = null;

5   for(int i=0; i<10; i++){
6     p  = p2;
7     p2 = new Point(Math.random()*10, Math.random()*10);
8   }
```

- Combien d'instances ont été créées au total?
- Combien d'instances restent-il à la fin de l'exécution du programme?
- Dessiner l'état de la mémoire.