

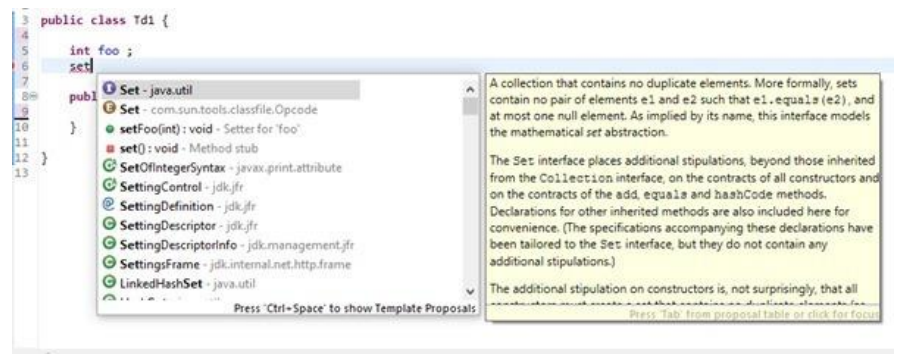
TD1

Exercice 1 - Eclipse

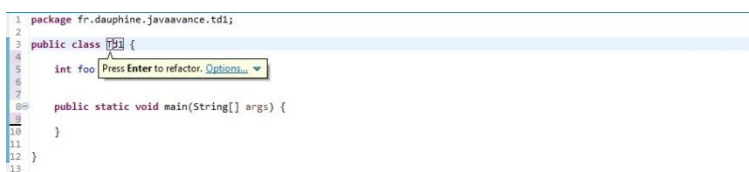
- 1) // réalisé sur Eclipse
- 2) écrire sysout puis presser Ctrl + Space affiche un System.out.println(). (raccourci clavier)
- 3) écrire toStr puis presser Ctrl + Space affiche toString(). (raccourci clavier)
- 4) Ctrl + Space dans une classe ouvre la template de propositions
- 5) Après avoir créé le champ int foo, presser Ctrl + Space ouvre la template de propositions, affichant les méthodes relatives à la classe



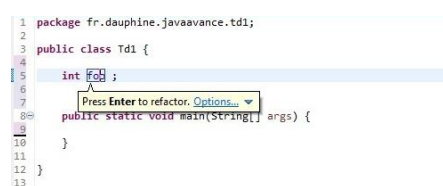
Ecrire « set » puis presser Ctrl+Space affiche tout les champs/méthodes ayant « set » dans leur nom.



- 6) Cela permet de renommer le nom de la classe (fonctionnalité refactor) partout où elle est appelée.



Il en est de même pour le l'attribut foo :



- 7) En général, lorsqu'on ctrl + clic sur une classe, cela nous ramène à sa déclaration.
On voit ici le code source de la déclaration de classe String.

```
String.class  Td1.java  Point.java  TestPoint.java  PolyLine.java  Circle.java
129 * @see      java.nio.charset.Charset
130 * @since     1.0
131 * @jls      15.18.1 String Concatenation Operator +
132 */
133
134 public final class String
135     implements java.io.Serializable, Comparable<String>, CharSequence,
136                Constable, ConstantDesc {
137
138     /**
139      * The value is used for character storage.
140      *
141      * @implNote This field is trusted by the VM, and is a subject to
142      * constant folding if String instance is constant. Overwriting this
143      * field after construction will cause problems.
144      *
145      * Additionally, it is marked with {@link Stable} to trust the contents
146      * of the array. No other facility in JDK provides this functionality (yet).
147      * {@link Stable} is safe here, because value is never null.
148      */
149     @Stable
150     private final byte[] value;
151 }
```

Exercice 2 - Point

- 1) Cela fonctionne car la méthode est définie dans la classe Point où sont définies les variables private, et peut donc y accéder directement.
- 2) On ne peut pas accéder directement aux attributs private d'un objet Point à partir de la classe TestPoint.
Pour pouvoir y accéder, on peut mettre les attributs en public (très peu conseillé), ou alors ajouter des **méthodes « getter » (accessor)** dans la classe Point pour les attributs x et y.
- 3) On doit fixer la visibilité des variables à private pour respecter la **notion d'encapsulation** : chaque classe doit protéger ses attributs afin de toujours respecter ses contrats.
- 4) Comme introduit dans la question 2, un accessor est une méthode qui permet d'accéder aux attributs d'une classe définie comme privés, en effet, elle retourne l'attribut en question.
Oui, il est nécessaire d'en implémenter ici, pour accéder aux attributs privés x et y de la classe Point depuis la classe TestPoint.

Les accessors implémentés dans la classe Point :

```
42 public double getX() {
43     return x;
44 }
45
46 public double getY() {
47     return y;
48 }
49 }
```

Nouveau code de la classe TestPoint :

```
public class TestPoint {

    public static void main(String[] args) {
        Point p=new Point();
        System.out.println(p.getX()+" "+p.getY()) ;
    }
}
```

5) La création du nouveau constructeur paramètre écrase le constructeur par défaut.

Il n'est plus possible d'initialiser un objet Point avec le constructeur par défaut Point().

```
public static void main(String[] args) {  
    Point p=new Point();  
    System.out.println(p.x+" "+p.y) ;  
}
```

```
public Point(double dx, double dy) {  
    this.x = dx ;  
    this.y = dy ;  
}
```

6) En appelant les paramètres x et y, il y a une ambiguïté entre les attributs et les paramètres entrants : il faut donc faire la distinction en appliquant le mot clé **this** aux variables de notre objet courant.

```
public Point(double x, double y) {  
    this.x = x ;  
    this.y = y ;  
}
```

7) Il suffit de créer un attribut static qu'on initialise à 0 et qu'on incrémente à chaque appel au constructeur.

Ici nous avons appelé cet attribut nbPoint.

```
public class Point {  
    private double x,y ;  
  
    private static int nbPoint = 0 ;  
  
    public Point(double x, double y) {  
        this.x = x ;  
        this.y = y ;  
        nbPoint ++ ;  
    }
```

8) Le compilateur reconnaît quel constructeur appeler selon les paramètres d'entrées lors de l'appel au constructeur.

C'est grâce à la **signature** qui est différente que le compilateur fait la distinction entre les deux constructeurs.

```
public class Point {  
    private double x,y ;  
  
    private static int nbPoint = 0 ;  
  
    public Point(double x, double y) {  
        this.x = x ;  
        this.y = y ;  
        nbPoint ++ ;  
    }  
  
    public Point (Point p2) {  
        this(p2.getX(), p2.getY()) ;  
    }
```

9) Pour cela il faut redéfinir la méthode `toString()`. `System.out.println()` applique la méthode `toString()` aux objets saisis dans son appel.

Comme il s'agit d'une redéfinition de la méthode, on ajoute le mot clé **@Override**.

```
@Override
public String toString() {
    return "(" + x + "," + y + ")";
}
```

Exercice 3 – Equality

1) La comparaison avec l'opérateur booléen `==` entre deux objets compare leur adresse mémoire.

Par conséquent, les objets `p1` et `p2` ont la même adresse mémoire : ainsi l'appel à `(p1 == p2)` renvoie `true`.

Cependant, `p3`, même avec les mêmes coordonnées que `p1` et `p2`, a une adresse mémoire différente. Donc l'appel `(p1 == p3)` renvoie `false`.

```
10 public static void main(String[] args) {
11     Point p1=new Point(1,2);
12     Point p2=p1;
13     Point p3=new Point(1,2);
14
15     System.out.println(p1==p2);
16     System.out.println(p1==p3);
17 }
```

Console Properties

<terminated> Point [Java Application] C:\Users\m_ala\.p2\pool\plugins\org.eclipse.justj.openj
true
false

2) La méthode `isSameAs(Point)` permet de retourner `true` lorsque les deux points ont les mêmes coordonnées, ainsi `p1.isSameAs(p3)` renverra `true` (contrairement au booléen `==`).

```
60 public boolean isSameAs(Point p) {
61     return p.getX() == this.x && p.getY() == this.y ;
62 }
63
```

```

10 public static void main(String[] args) {
11     Point p1=new Point(1,2);
12     Point p2=p1;
13     Point p3=new Point(1,2);
14
15     System.out.println(p1==p2);
16     System.out.println(p1==p3);
17     System.out.println(p1.isSameAs(p3));
}

```

// p1.isSameAS(p3) retourne true

```

<terminated> Point [Java Application] C:\Users\m_ala\p2\pool\plugins\org.ecli
true
false
true

```

3) `indexOf(Object)` permet de savoir si une liste contient un objet, en retournant l'index de sa position dans la liste s'il y appartient, ou -1 le cas échéant.

Ainsi, dans cet exemple, `indexOf(p3)` renvoie -1 car la méthode `indexOf(Object)` utilise `Object.equals(Object)` pour comparer l'objet recherché avec ceux de la liste.

La méthode `Object.equals(Object)` compare l'adresse mémoire entre deux objets, comme l'opérateur booléen `==`.

```

ArrayList.class x Td1.java *Point.java TestPoint.java
284 /
285 public int indexOf(Object o) {
286     return indexOfRange(o, 0, size);
287 }
288
289 int indexOfRange(Object o, int start, int end) {
290     Object[] es = elementData;
291     if (o == null) {
292         for (int i = start; i < end; i++) {
293             if (es[i] == null) {
294                 return i;
295             }
296         }
297     } else {
298         for (int i = start; i < end; i++) {
299             if (o.equals(es[i])) {
300                 return i;
301             }
302         }
303     }
304     return -1;
305 }
306

```

Pour résoudre ce problème, on doit redéfinir la méthode `equals(Object)` dans la classe `Point`, en comparant les coordonnées de deux points pour savoir s'ils sont égaux ou non.

Ainsi `indexOf()` utilisera le `equals(object)` redéfini dans la classe `Point` selon le principe de **polymorphisme**.

```

72 @Override
73 public boolean equals(Object obj) {
74     if (this == obj)
75         return true;
76     if (obj == null)
77         return false;
78     if (getClass() != obj.getClass())
79         return false;
80     Point other = (Point) obj;
81     if (x != other.x)
82         return false;
83     if (y != other.y)
84         return false;
85     return true;
86 }
87

```

Exercice 4. Polyline

1)

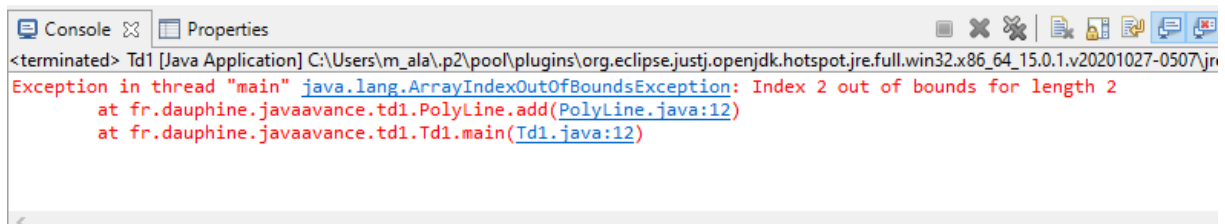
```

public class PolyLine {
    private Point [] points ;
    private int nbPoint ;

    public PolyLine(int maxPoint ) {
        points = new Point [maxPoint] ;
    }
}

```

2) Si on ajoute plus de points dans le tableau que sa capacité maximale, une erreur est générée : « java.lang.ArrayIndexOutOfBoundsException » .



```

<terminated> Td1 [Java Application] C:\Users\m_ala\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_15.0.1.v20201027-0507\jre\bin\java.exe
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds for length 2
    at fr.dauphine.javaavance.td1.PolyLine.add(PolyLine.java:12)
    at fr.dauphine.javaavance.td1.Td1.main(Td1.java:12)

```

Pour éviter ce cas, on ajoute une condition avant l'ajout d'un point dans le tableau : vérifier que le nombre de points dans le tableau (variable nbPoint) est inférieur à la taille du tableau (points.length).

```

11 public void add(Point p) {
12     if(nbPoint < points.length) {
13         points[nbPoint++] = p ;
14     }
15 }
16

```

3)

```

17 public int pointCapacity() {
18     return points.length ;
19 }
20
21 public int nbPoint() {
22     return nbPoint ;
23 }

```

4)

```

25 public boolean contains(Point p) {
26     for(Point x : points) {
27         if (p.equals(x))
28             return true ;
29     }
30     return false ;
31 }
32

```

5) Si on exécute la méthode contains(Point) avec null à la place d'un point existant en paramètre, l'erreur suivante apparaît : « java.lang.NullPointerException » car null ne peut pas appeler de méthode (la méthode equals).

En faisant add(null) avant, la même erreur apparaît en exécutant contains(null) :

```

8 public static void main(String[] args) {
9     PolyLine pl = new PolyLine(2) ;
10    pl.add(null);
11    system.out.println(pl.contains(null) ) ;
12 }

```

Console Properties

<terminated> Td1 [Java Application] C:\Users\m_alal\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_15.0.1.v20201027-
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "fr.dauphine.javaavance.td1.Point.
at fr.dauphine.javaavance.td1.PolyLine.contains(PolyLine.java:27)
at fr.dauphine.javaavance.td1.Td1.main(Td1.java:11)

« java.lang.NullPointerException ». Ici encore, null ne peut pas appeler de méthode (méthode equals).

Objects.requireNonNull(o) lève une exception si l'argument o est null.

6) La méthode pointCapacity ne sert plus à rien, car une LinkedList n'a pas de capacité maximale.

```

6 public class PolyLine {
7     private LinkedList<Point> points ;
8
9     public PolyLine( ) {
10         points = new LinkedList<>();
11     }
12

```

Changements apportés aux variables d'instances et au constructeur.

```

21
22 public int nbPoint() {
23     return points.size() ;
24 }
25
26 public boolean contains(Point p) {
27     return points.contains(p) ;
28 }
29

```

Changements apportés aux méthodes nbPoint et contains (nous utilisons les fonctions de LinkedList).

Exercice 5. Mutability and circle

1) Il y a deux façons d'écrire la méthode translate : une façon **mutable** et une façon **non mutable**.

```
103 public Point translate (double dx, double dy) {  
104     return new Point(x+dx,y+dy) ;  
105 }  
106
```

Façon **non mutable** : on retourne un objet Point avec les nouvelles coordonnées (les variables d'instances restent inchangées)

```
95  
96 public void translate(double dx, double dy) {  
97     x += dx ;  
98     y += dy ;  
99 }  
100
```

Façon **mutable** : on modifie les valeurs des variables d'instances.

2)

```
3 public class Circle {  
4     private final Point center ;  
5     private final double radius ;  
6  
7     public Circle(Point center, double radius) {  
8         this.center = center ;  
9         this.radius = radius ;  
10    }  
11  
12 }  
13
```

3)

```
11  
12 @Override  
13 public String toString() {  
14     return "Circle with center : " + center + ", and radius : " + radius ;  
15 }  
16  
17 public static void main(String[] args) {  
18     Circle c = new Circle(new Point(0.0, 1.5), 4.5) ;  
19     System.out.println(c);  
20 }
```

Console Properties

<terminated> Circle [Java Application] C:\Users\m_ala\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full
Circle with center : (0.0,1.5), and radius : 4.5

4)

```

16
17 public void translate(int dx, int dy) {
18     center.translate(dx,dy) ;
19 }

```

5) Le problème de ce code est que les objets Circle 'c' et 'c2' référencent le même objet Point 'p'.
c2.translate(1,1)

```

7 public Circle(Point center, double radius) {
8     this.center = new Point(center) ;
9     this.radius = radius ;
10 }

```

Console Properties

<terminated> Circle [Java Application] C:\Users\m_ala\.p2\pool\plugins\org.eclipse.jus
 Circle with center : (1.0,2.0), and radius : 1.0
 Circle with center : (2.0,3.0), and radius : 2.0

Comment résoudre ce soucis ?

-> il faut faire une **copie défensive**

6) Une méthode getCenter() qui retourne le center causerait le même problème : on modifierait l'attribut center directement.

Pour résoudre ce soucis, la **copie défensive** doit aussi être réalisée lors de l'envoi de paramètres :

```

20 public static void main(String[] args) {
21     Circle c=new Circle(new Point(1,2), 1);
22     c.getCenter().translate(1,1);
23     System.out.println(c);
24 }
25 }
26 public Point getCenter() {
27     return new Point(center);
28 }

```

Console Properties

<terminated> Circle [Java Application] C:\Users\m_ala\.p2\pool\plugins\org.eclipse
 Circle with center : (1.0,2.0), and radius : 1.0

7) Dans cet exemple on calcule l'aire d'un cercle de rayon 1.

```

21
22 public static void main(String[] args) {
23     Circle c=new Circle(new Point(1,2), 1);
24     System.out.println(c);
25 }
26 public double area() {
27     return Math.PI * Math.pow(radius, 2) ;
28 }
29 @Override
30 public String toString() {
31     return "Circle with an area of : " + area();
32 }

```

Console Properties

<terminated> Circle [Java Application] C:\Users\m_ala\.p2\pool\plugins\org.eclipse
 Circle with an area of : 3.141592653589793

- 8) Contains(Point p) : On calcule la distance euclidienne qui sépare le point 'p' du point 'center'.
Si le rayon est supérieur ou égale à cette distance, c'est que le point p appartient à notre cercle.

```
34  
35 public boolean contains(Point p) {  
36     double distance = Math.sqrt( Math.pow(center.getY() - p.getY(),2) + Math.pow(center.getX() - p.getX(),2)) ;  
37     return radius >= distance ;  
38 }  
39  
40
```

- 9)
- ```
39
40 public boolean contains(Point p, Circle...circles) {
41 for (Circle c : circles) {
42 if(c.contains(p)) {
43 return true ;
44 }
45 }
46 return false ;
47 }
```

Quel changement faire en plus ?

-> Déclarer cette méthode en static car elle ne doit pas dépendre d'une instance.

```
40 public static boolean contains(Point p, Circle...circles) {
41 for (Circle c : circles) {
42 if(c.contains(p)) {
43 return true ;
44 }
45 }
46 return false ;
47 }
```

## Exercice 6. Anneaux

- 1) La classe Ring partagerait des caractéristiques commune à la classe Circle : un centre et un rayon.  
En plus de ces caractéristiques, un anneau possède un deuxième rayon à prendre en compte : il est légitime d'utiliser l'héritage pour que Ring soit une classe fille de Circle.

2)

```
4 public class Ring extends Circle {
5
6 private double innerRadius ;
7
8 public Ring(Point center, double r1, double r2) {
9 super(center, r1);
10 if(r2>r1) {throw new IllegalArgumentException() ;}
11 this.innerRadius = r2 ;
12 }
13
```

3) On redéfinit la méthode equals dans la classe Circle.

```
64 @Override
65 public boolean equals(Object obj) {
66 if (this == obj)
67 return true;
68 if (obj == null)
69 return false;
70 if (getClass() != obj.getClass())
71 return false;
72 Circle other = (Circle) obj;
73 if (center == null) {
74 if (other.center != null)
75 return false;
76 } else if (!center.equals(other.center))
77 return false;
78 if (Double.doubleToLongBits(radius) != Double.doubleToLongBits(other.radius))
79 return false;
80 return true;
81 }
82
83
```

On redéfinit la méthode equals dans la classe Ring, en vérifiant l'égalité des attributs center et r1 de la classe mère Circle, grâce au mot clé **super**.

```
24 @Override
25 public boolean equals(Object obj) {
26 if (this == obj)
27 return true;
28 if (!super.equals(obj))
29 return false;
30 if (getClass() != obj.getClass())
31 return false;
32 Ring other = (Ring) obj;
33 if (Double.doubleToLongBits(innerRadius) != Double.doubleToLongBits(other.innerRadius))
34 return false;
35 return true;
36 }
37
38
```

4) sans modifier le code :

```
38 public static void main(String[] args) {
39 Ring c = new Ring(new Point(1,2), 3, 1);
40 System.out.println(c);
41 }
```

Console Properties

<terminated> Ring [Java Application] C:\Users\m\_ala\.p2\pool\plugins\org.eclipse

Circle with an area of : 28.274333882308138

Sans ajouter une méthode toString() dans la classe Ring, le compilateur utilise la méthode toString() de la classe Circle.

En redéfinissant toString() dans la classe Ring :

```
40 @Override
41 public String toString() {
42 return "Ring with radius : " + super.getRadius() + " and inner radius : " + this.innerRadius;
43 }
44
45 public static void main(String[] args) {
46 Ring c = new Ring(new Point(1,2), 3, 1);
47 System.out.println(c);
48 }
```

Console Properties

<terminated> Ring [Java Application] C:\Users\m\_ala\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_15.0.1.v2020102

Ring with radius : 3.0 and inner radius : 1.0

5)

```
50 @Override
51 public boolean contains(Point p) {
52 double distance = Math.sqrt(Math.pow(super.getCenter().getY() - p.getY(),2)
53 + Math.pow(super.getCenter().getX() - p.getX(),2)) ;
54 return innerRadius >= distance ;
55 }
56
```

6)

```
57 public static boolean contains(Point p, Ring...rings) {
58 for (Ring r : rings) {
59 if(r.contains(p)) {
60 return true ;
61 }
62 }
63 return false ;
64 }
```