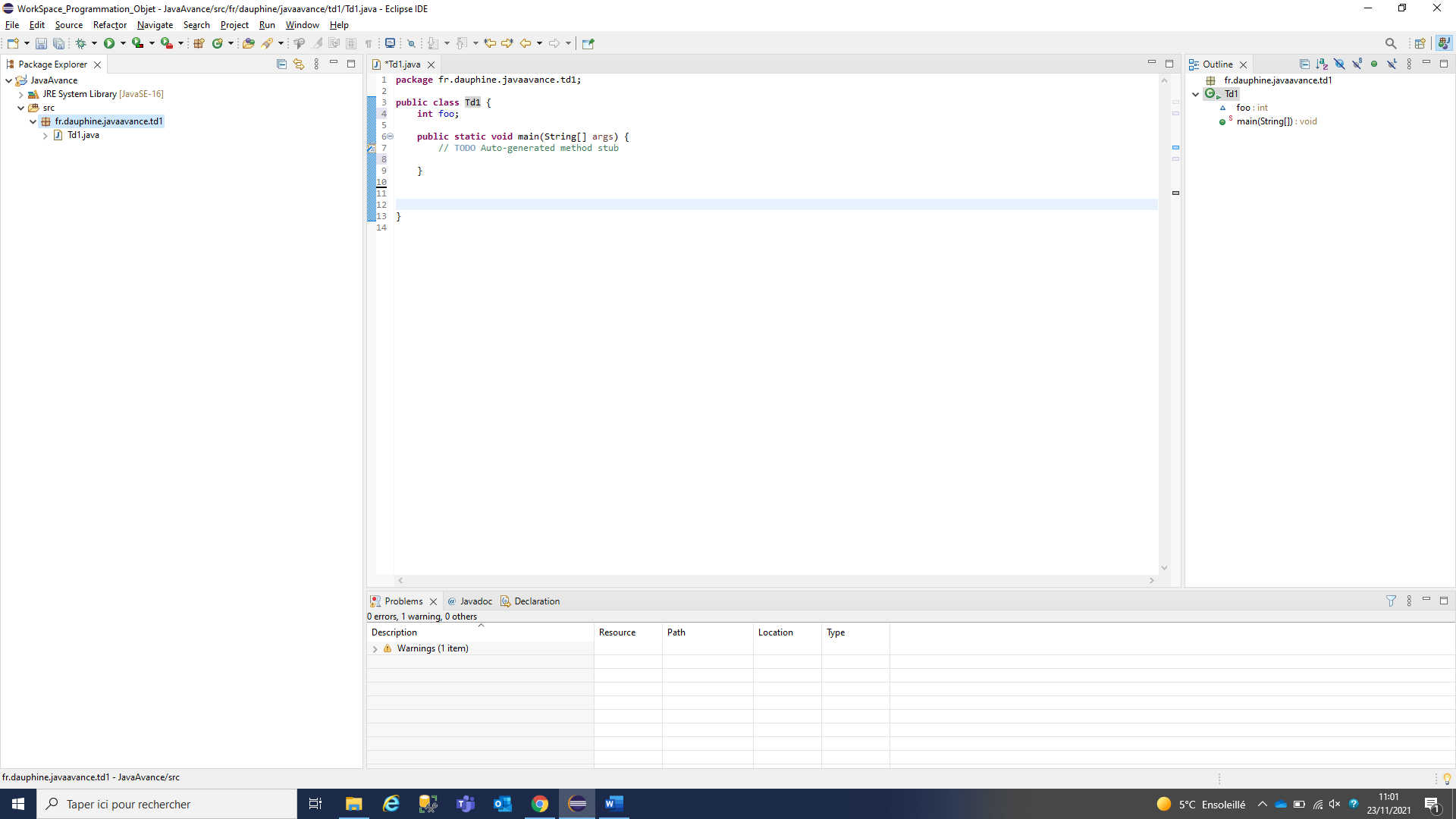
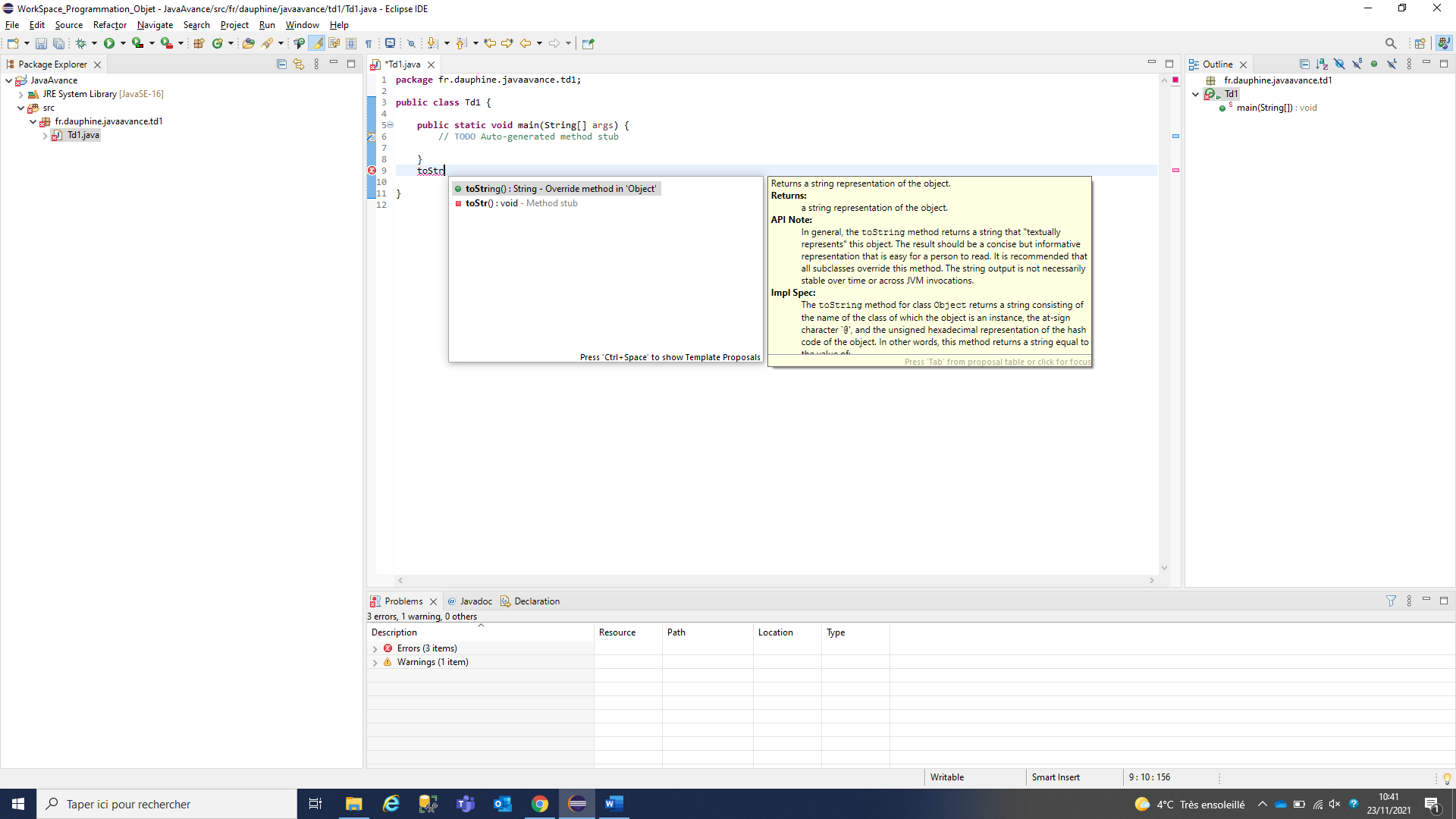
Programmation Objet Avancée – TD1

# Exercice 1 :





1. Cela nous propose de généré le code de la fonction suivante : System.***out***.println(); .
2. Cela nous propose de Override la méthode toString ou de créer une méthode toStr de type void. (Voir image ci-dessous)

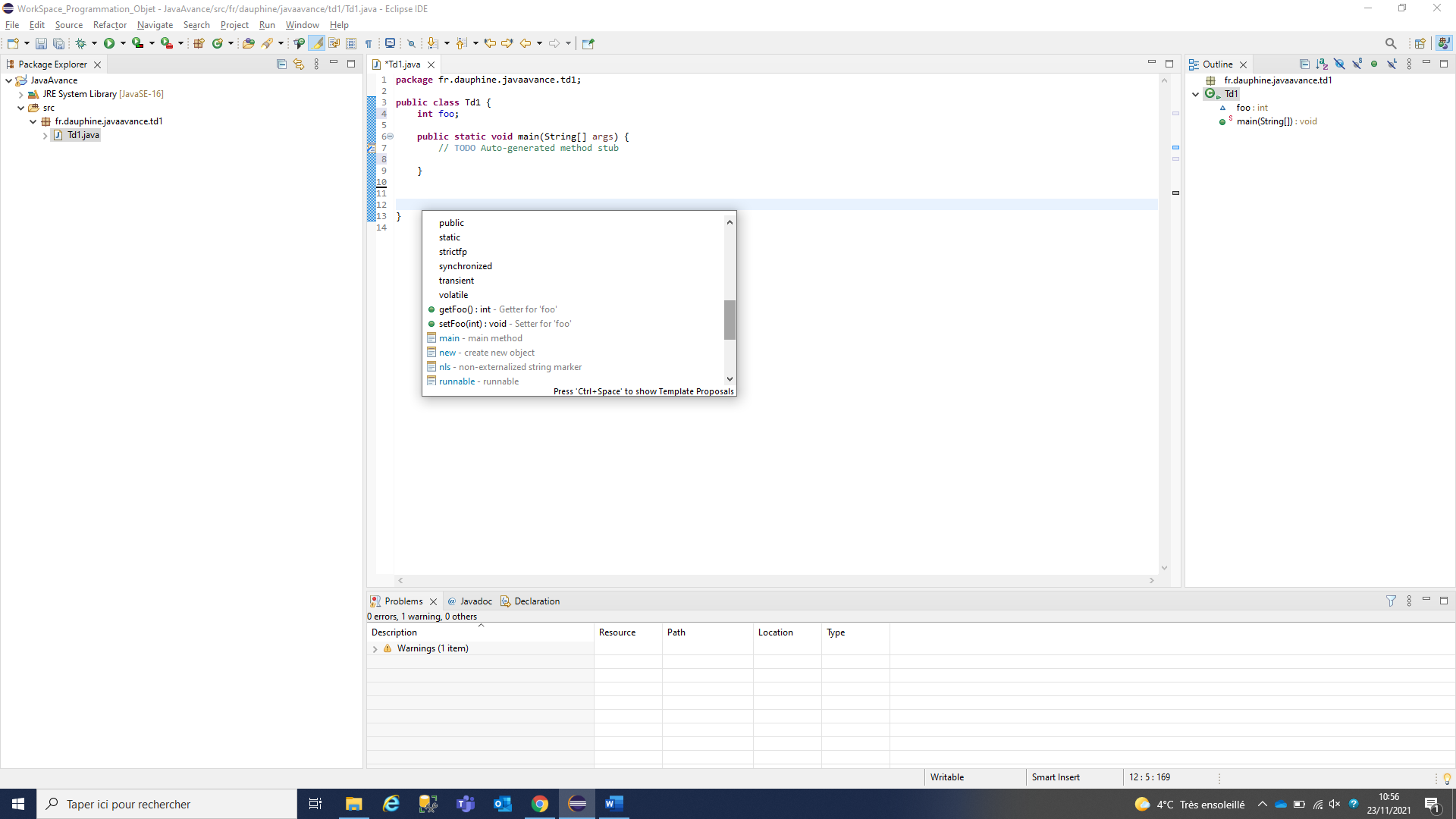


1. Cela nous propose de créer la méthode main.

**public** **static** **void** main(String[] args) {

}

1. Si on tape ctrl + espace, Eclipse ouvre la template de propositions et nous propose de créer un getter et un setter pour la variable int foo :



Et lorsque l’on écrit set puis on fait ctrl + espace, Eclipse nous propose de créer la méthode setFoo :

**public** **void** setFoo(**int** foo) {

**this**.foo = foo;

}

1. Cela nous propose d’utiliser l’outil d’Eclipse « Refactor Rename » qui permet de changer le nom de la classe, et cela va modifier le nom de la classe, partout où elle est référencée, dans toutes les méthodes où elle est appelé. Dans le cas où l’on fait cette commande sur l’attribut foo, la même chose se produit, on peut modifier le nom du champs, ce qui va le modifier partout où il est référencé.
2. Lorsque l’on fait la commande « ctrl + clic » sur une classe, tel que la classe String, cela nous emmène directement là où est déclaré la classe, on peut donc y voir toutes les méthodes définis dans la classe String. C’est très utile pour voir le code source de certaines méthodes, afin de mieux comprendre leurs fonctionnemts.

# Exercice 2 :

1. Ça fonctionne car mêmes si les attributs sont privés, on peut y accéder car ils sont dans la même classe. De plus il existe un constructeur par défaut qui va initialiser les variables x et y à zéro, d’où l’affichage en sortie : 0 0
2. Ça ne va pas fonctionner et levé des exceptions car on ne peut pas accéder à des attributs privés d’une autre classe. Pour résoudre ce problème, on peut créer des Getters dans la classe Point, ce qui va permettre d’accéder aux attributs.
3. Pour bien respecter la règle d’encapsulations, on va encapsulé les attributs. Cela va permettre de protégé et conservé l’intégrité des données. Elles seront protéger de l’extérieur, cela permet de contrôler l’accès aux composants critiques de l’objet.
4. C’est une méthode permettant d’accéder à la valeur d’un attribut private tout en le protégeant des modifications extérieures. Oui on doit le faire ici car les attributs x et y sont privés, on va donc y accéder à travers des getters.
5. Le problème est que ça ne compile plus car il n’y a plus de constructeur par défaut (il est écrasé par le nouveau constructeur), en effet, dorénavant le constructeur prend 2 paramètres en entré.
6. Cela peut créer une ambigüité entre les paramètres du constructeur et les attributs de la classe, il faut donc utiliser « this » pour bien différencier les attributs.
7. On peut créer une variable static initialisé à zéro et qui sera incrémenter à chaque appel au constructeur.
8. Le compilateur sait quel constructeur appelé en fonction du nombre d’arguments dans le constructeur et de leurs types.
9. Nous avons redéfinis la méthode toString() :

@Override

**public** String toString() {

**return** "(" + **this**.x + "," + **this**.y + ")";

}

# Exercice 3 :

1. Le premier System.***out***.println va afficher true car les variables p1 et p2 ont la même adresse mémoire. Mais le second System.***out***.println va afficher false car p1 et p3 n’ont pas la même adresse mémoire, même si ces deux variables contiennent la même chose.

**public** **boolean** isSameAs(Point p) {

**return** ((p.getX()==**this**.x) && (p.getY()==**this**.y));

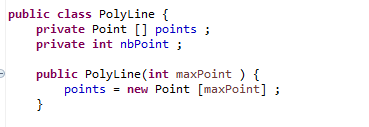
}

1. Pour comprendre quel est le problème dans ce code, il faut comprendre comment fonctionne la méthode indexOf() sur une liste. La méthode indexOf() fait appel à la méthode equals() afin de comparer l’adresse mémoire entre deux objets, elle renvoie true si les adresses mémoires sont identiques et -1 sinon, comme ce qui est fait dans la question 1 avec « == », en d’autre termes, la méthode indexOf() va voir si la liste contient l’objet passé en paramètre de la fonction indexOf(). Ici, lors de l’appel à la méthode indexOf() sur la liste « list » avec comme paramètre p3 (list.indexOf(p3)), cela va renvoyer -1 car il n’existe aucun objets dans la liste ayant la même adresse mémoire que l’objet p3, en effet les objets p1 et p3 n’ont pas la même adresse mémoire même si pourtant ils contiennent les mêmes données. En ce qui concerne l’appel à la méthode indexOf() avec comme paramètre p2 (list.indexOf(p2)), cela va renvoyé true car il existe bien dans la liste un objet ayant la même adresse mémoire que p2, en effet les 2 objets p1 et p2 possèdent (partagent) la même adresse mémoire.

Si l’on veut que l’appel à la méthode indexOf() avec comme paramètre p3 (list.indexOf(p3)) fonctionne, il faudrait implémenter (redéfinir) la méthode equals() dans la classe Point afin de comparer les données dans les 2 objets et non pas les adresses mémoires.

# Exercice 4 :

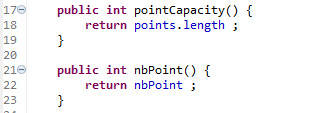




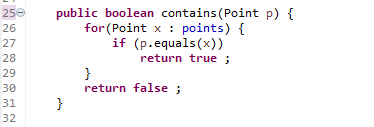
1. Si on ajoute plus de points dans le tableau alors qu’il est déjà à sa capacité maximale, l’exception suivante est levé : « java.lang.ArrayIndexOutOfBoundsException ».

Pour résoudre ce problème, nous pouvons ajouter une condition dès l’entrer dans la méthode add(Point p) qui testera si le nombre de points dans le tableau est bien inférieur à la taille maximale du tableau.









1. Si l’on exécute la méthode contains() en donnant « null » à la place d’un objet « Point », cela va lever une exception :

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Object.equals(Object)" because "p" is null

Si l’on fait add() avant, l’appel à la méthode add() va fonctionné, l’objet null va bien être ajouté dans le tableau points mais l’appel à la méthode contains(null) avec le paramètre null ne va pas fonctionné et va lever la même exception que tout à l’heure.

La méthode Objects.requireNonNull(o) lève une exception si le paramètre o est null.

1. Avec l’utilisation d’une LinkedList, la méthode pointCapacity() ne sert plus à rien et doit donc être supprimé car une LinkedList n’a pas de capacité maximale.

Pour ce qui concerne les méthodes nbPoints() et contains(), voici à quoi elles ressemblent dorénavant, nous utilisons maintenant les fonctions existantes de la LinkedList :

**public** **int** nbPoint() {

**return** points.size();

}

**public** **boolean** contains(Point p) {

**return** points.contains(p);

}

# Exercice 5 :

1. Les deux options pour écrire la méthode translate(dx, dy) sont :
   1. De manière mutable (en modifiant les valeurs des variables d’instances) :

**public** **void** translate(**double** dx, **double** dy) {

x += dx;

y += dy;

}

* 1. De manière non-mutable (sans modifier les valeurs des variables d’instances, donc via un nouvel objet Point) :

**public** Point translate (**double** dx, **double** dy) {

**return** **new** Point(x + dx, y + dy);

}



Une image contenant texte

Description générée automatiquement

@Override

**public** String toString() {

**return** "Circle : center = " + center + ", radius = " + radius;

}



**public** **void** translate(**int** dx, **int** dy) {

**this**.center.translate(dx, dy);

}

1. Le problème avec code est que les deux cerclent c et c2 référencent le même objet Point p, les deux cerclent ont le même centre qui est le point p. Pour résoudre le problème, on pourrait lors de la création du cercle créer une copie du Point passé en paramètre :

**public** Circle(Point center, **double** radius) {

**this**.center = **new** Point(center);

**this**.radius = radius;

}

Ainsi, la translation fonctionnera bien car les centres des 2 cercles auront des adresses mémoires différentes.

1. La méthode getCenter() renvoie le Point qui définit le centre du cercle, cela causerait le même problème que tout à l’heure, on modifierait l’attribut center directement sans passer par la méthode translate() du cercle.

Pour éviter ce problème, on doit envoyer une copie, c’est-à-dire que l’on va renvoyer un point ayant le même centre :

**public** Point getCenter() {

**return** **new** Point(center);

}

**public** **static** **void** main(String[] args) {

Circle c = **new** Circle(**new** Point(1, 2), 1);

c.getCenter().translate(1, 1);

System.***out***.println(c);

}



**public** **double** area() {

**return** Math.***PI***\*Math.*pow*(radius, 2);

}

@Override

**public** String toString() {

**return** "Circle : center = " + center + ", radius = " + radius + ", area = " + area();

}



**public** **boolean** contains(Point p) {

**double** distance = Math.*sqrt*(Math.*pow*(center.getY() - p.getY(), 2) + Math.*pow*(center.getX() - p.getX(), 2));

**return** radius >= distance;

}



**public** **static** **boolean** contains(Point p, Circle...circles) {

**for**(Circle c : circles) {

**if**(c.contains(p)){

**return** **true**;

}

}

**return** **false**;

}

L’autre changement que l’on devrait faire est de déclarer la méthode en static afin qu’elle ne dépende pas d’une instance.

# Exercice 6 :

1. Oui, nous devrions utiliser l’héritage car un anneau (ring) possède également un rayon, une aire et un centre, et en plus de cela, un anneau possède un deuxième rayon. Il est donc logique de créer la classe Ring en la faisant hériter de la classe Circle. Il faudra cependant bien penser à prendre en compte les nouvelles caractéristiques de l’anneau et il faudra redéfinir les méthodes toString() et area() car ce ne sont pas les mêmes pour un anneau et pour un cercle.

**public** **class** Ring **extends** Circle{

**private** **double** inRadius;

**public** Ring(Point center, **double** radius1, **double** radius2) {

**super**(center, radius1);

**if**(radius2 > radius1)

**throw** **new** IllegalArgumentException();

**this**.inRadius = radius2;

}

}

1. Il faut redéfinir la méthode equals() dans la classe Circle ainsi que dans la nouvelle classe Ring.

Dans la classe Circle :

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Circle other = (Circle) obj;

**if** (center == **null**) {

**if** (other.center != **null**)

**return** **false**;

} **else** **if** (!center.equals(other.center))

**return** **false**;

**if** (Double.*doubleToLongBits*(radius) != Double.*doubleToLongBits*(other.radius))

**return** **false**;

**return** **true**;

}

Dans la classe Ring :

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (!**super**.equals(obj))

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Ring other = (Ring) obj;

**if** (Double.*doubleToLongBits*(inRadius) != Double.*doubleToLongBits*(other.inRadius))

**return** **false**;

**return** **true**;

}

1. Si on fait System.***out***.println(ring); sans code supplémentaire, le compilateur va utilisé la méthode toString() de la classe mère, c’est-à-dire la classe Circle. Et donc les nouvelles caractéristiques de l’anneau ne seront pas affichés car elles ne sont pas prises en compte dans la méthode toString() de la classe Circle. Il faut donc réimplémenter la méthode toString() dans la classe Ring.

@Override

**public** String toString() {

**return** "Ring : radius = " + **super**.getRadius() + ", inner-radius = " + **this**.getRadius();

}



@Override

**public** **boolean** contains(Point p) {

**double** distance = Math.*sqrt*(Math.*pow*(**super**.getCenter().getY() - p.getY(), 2) + Math.*pow*(**super**.getCenter().getX() - p.getX(), 2));

**return** inRadius >= distance;

}



**public** **static** **boolean** contains(Point p, Ring...rings) {

**for**(Circle c : rings) {

**if**(c.contains(p)){

**return** **true**;

}

}

**return** **false**;

}