**TD1 JAVA de TABBAKH Mohamed Amine (M1 MIAGE)**

**Exercice 2 :**

1. Cela marche car en Java lorsqu'aucun constructeur n'est déclaré, ce qui est notamment le cas pour notre class Point, par défaut il est possible de créer une instance de la class en appelant un constructeur par défaut avec le nom de la class. Dans notre cas, c'est Point(). De plus, ce constructeur par défaut s'appelle uniquement par le nom de la classe et possède comme signature Nomdelaclass(). Par contre, si l'on définit un constructeur, l'appel du constructeur par défaut n'est plus possible. Ainsi, par exemple dans notre cas, si dans la class Point, l'on définit Point(int a) comme un constructeur, il sera plus possible d'appeler Point(). Cela entrainera l'erreur suivante : The constructor Point() is undefined sauf si l'on redéfinit ce dernier biensûr avec un public Point(){}. De plus, Lorsqu'on donne pas de valeurs aux attributs entiers d'une class par défaut ils prennent la valeur 0 en java.

2. On a les erreurs : The field Point.x is not visible et The field Point.y is not visible car les attributs de Point sont en private. On peut régler ce problème en créant des méthodes accessor qui permettent d'accéder aux attributs par exemple : public int getX() && public int getY().

3. C'est le principe même de la POO et de l'encapsulation. On permet à l'utilisateur d'utiliser un objet sans connaitre son fonctionnement interne. Pour l'utilisateur, il a la possibilité d'utiliser getX() par exemple, sans connaitre les détails de l'implémentation soit le type de X. Et pour le développeur, c'est la possibilité de changer la représentation interne sans changer l'interface. Pou r résumer on contrôle l'accès aux composants critiques de l'object (les variables membres) et ce dernier n'expose que certaines fonctionnalités au Monde extérieur soit l'utilisateur par le bais des méthodes qui sont elles totalement public.

4. Un accessor permet de renvoyer les attributs d'un objet sans forcément connaitre le type de ces dernières. Par exemple, dans notre problème, sachant que Point possède un attribut x en private auquel on ne pas accéder depuis une autre class, on est obligé de passer par un accessor getX() pour pouvoir avoir la valeur de cet attribut. Donc oui dans notre exemple, on est obligé de passer par des accessors à savoir getX() et getY().

5. Le problème est que l'on a déjà les attributs de l'objet Point qui s'appellent px et py. Donc lorsqu'on effectue px=px cela n'a pas de sens tantôt pour un humain et que pour le compilateur.

6. Celà fonctionne d'un coup car cette fois, l'on effectue x=px et c'est donc deux éléments différents, l'une correspond à un paramètre du constructeur alors que l'autre correspond à l'attribut de l'objet. Une autre solution est d'utiliser this, l'on peut donc écrire : this.px=px et cela marcherait aussi si l'on ne veut pas renommer nos paramètres.

7. Il suffit de créer un attribut static number_points et l'incrémenter lors de la création de chaque objet de type Point. Et pour connaitre le nombre de points, il suffira d'effectuer un System.out.println(Point.number_points) qui affichera le nombre de points.

8. Le compilateur utilise la signature du constructeur. Dans notre cas , on a deux constructeur l'un ayant comme signature Point(int,int) et l'autre Point(Point) donc si l'on passe en paramètre un point il va savoir qu'il faut appeler le deuxième constructeur et non pas le premier.

**Exercice 3 :**

1. On affiche True False. L'opérateur "==" en java permet de savoir si deux objets pointent vers le même emplacement mémoire, ici en l'occurence p1 et p2 pointent vers le même emplacement mémoire et donc on a le True. Une autre manière de le voir est que l'opérateur "==" compare si les deux objets ont la même référence sur le tas. Concernant p1 et p3 ils pointent vers deux emplacement mémoire différents car p3 appellent au constructeur Point(int,int) qui alloue un espace mémoire différent de celui alloué par p1 et donc on a le False.

3. On nous retourne la valeur 0 soit la position de p1 dans la liste alors que c'est faux pour p3 qui n'est pas dans la liste et possède un espace mémoire totalement différent que p1 et p2. Normalement on est censé retourné -1 pour p3. IndexOf appel la méthode equals que l'on a réécrite dans Point et on sait que p1.equals(p3) donne true si l'on s'arrête au code que l'on a établis à la question 1. Donc pour éviter cela, on va rajouter à la méthode equals de Point l'instruction : if(!(p==this){return false} soit que dans le cas ou l'on a deux espaces mémoire différentes l'on renvoie false, ce qui nous permet d'avoir -1 pour list.indexOf(p3) soit que ce dernier ne se trouve pas dans la liste. Quant à p2 cela donne 0 car l'on a vu précédemment que p1 et p2 pointent vers le même espace mémoire donc cela est juste.

**Exercice 4 :**

2. Si l'on arrive à la capacité maximum du tableau, l'on a un grand problème car il est impossible de redimensionner un tableau en java. L'une des solutions possible et de créer un nouveau tableau de dimension supérieur à l'ancien et de copier l'ancien tableau dans ce dernier et d'augmenter la nombre maximum de points tout changeant les attributs de la class Polyline soit que le nouveau tableau soit le tableau de cette class.

5. Si l'on fait add(null), l'on a deux cas de figures si le tableau est full cela ne fait absolument rien et si y'a une place dans le tableau par exemple à l'indice i alors on sait que cette case pointe déjà vers le pointeur nul soit que array[i]=null donc cela ne fait rien. Pour éviter un code redondant, l'on peut utiliser Objects.requireNonNull(o) qui lève une exception si o est nul et on a donc l'erreur à la compilation suivante : java.lang.NullPointerException : On a donné nulle en paramètre !

6. Concernant Pointcapacity à la différence d'un tableau, l'on a une capacité "infini" enfin égale à la taille de la mémoire disponible alloué pour cela et donc selon la RAM de l'ordinateur. Pour les autres, on utilise les méthodes déjà existant des Linkedlists.

**Exercice 5 :**

1. On peut soit effectuer la version mutable ou non-mutable on a donc : translate_mutable et translate_non_mutable.

5. Le problème est lorsqu'on a appelé la méthode translate pour c2 on a modifié les coordonnées du point p1 car la class Point est mutable et donc l'on a que c et c2 ont le même centre se qui est faux ! Pour éviter cela, l'objet référencé à savoir ici Point doit aussi être non mutable et pour cela on doit utiliser la méthode translate_non_mutable de Point et le problème est réglé ! L'on conclut donc que rajouter un final dans Circle ne suffit pas pour rendre Circle non mutable mais il faut bien que Point soit non mutable aussi !

6. On a comme résultat ((2.0,3.0);1.0) soit que le cercle a été translaté alors que non seul le point a translaté pas le cercle pour cela on doit faire appel à notre méthode translate_non_mutable qui rend la class Point non mutable comme expliqué dans les questions précédentes et avec une telle méthode on a bien que le résultat ((1.0,2.0);1.0) souhaité car le class Point est non mutable à l'aide de cette méthode.

7. On calcule la distance euclidéenne entre d(c,p) avec c le centre du cercle et si d(c,p) <= r alors c'est bon.

8. On doit utiliser une déclaration plus précise pour pouvoir vérifier tous les cercles. ??

**Exercice 6 :**

1. L'on peut l'utiliser avec Cercle cela facilite la tache.
4. Cela affiche le toString de Circle et c'est pas se que l'on veut. On doit donc réécrire to String dans la class Ring.

**Exercice 1 :**

2. Un raccourci permettant l'affichage.
3. Un raccourci permettant pour toString.
4. Un raccourci pour créer un main.
5. Raccourci pour créer un setter.
6. Raccourci nous permettant d'avoir accès à toutes les informations nécessaires à savoir package, class …