Logboek opdracht 1

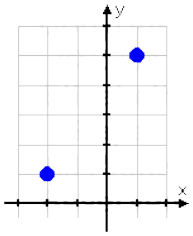
Bevindingen

Packages
In Java heet een namespace een package. Hier zitten dus klassen en/of interfaces in die gerelateerd zijn aan elkaar.

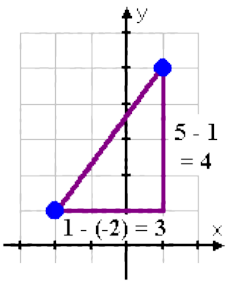
Klasnamen in hoofdletter
String, eerste letter moet uppercase zijn. Objectnamen starten hier altijd met een hoofdletter.

Lists
Ik importeerde java.awt.List om een list te gebruiken, maar dat is niet goed. Dit leverde een aantal foutmeldingen op en daardoor kwam ik erachter dat ik juist java.util.List moest gebruiken. Java.awt.List is een component(bijv. dropdownlist) voor bijv. in een GUI. De util.list versie is onderdeel van de Collections library en is de interface die andere lijsten onder zich heeft: zoals een ArrayList of LinkedList. Op de java.awt.List kun je niet list/collection-specifieke methoden gebruiken.

Comparing
Bij het implementeren van de compare methode van de Comparator interface liep ik tegen een probleem aan. Ik wist niet goed hoe ik de afstand tussen twee punten moest berekenen. Toen ben ik hiernaar gaan googelen en kwam ik uit op de stelling van Pythagoras. Deze heb ik even bestudeerd, want ik was hem allang weer vergeten. Wat deze stelling beweert is eigenlijk het volgende: **In een rechthoekige driehoek is de som van de kwadraten van de lengtes van de rechthoekszijden gelijk aan het kwadraat van de lengte van de schuine zijde.** Nu je dit weet kun je de schuine zijden dus uitrekenen door de volgende formule: $A^2 + B^2 = C^2$.



Het idee voor het probleem in de opdracht is dat je van de twee punten een rechthoekige driehoek maakt door een extra punt toe te voegen. Dit is het punt waar de twee getrokken lijnen vanuit de andere punten elkaar kruisen.



Het is daarna gemakkelijk om de horizontale en verticale zijde te berekenen. Je moet dan gewoon de x-waarden en y-waarden van elkaar aftrekken zoals je ziet in het tweede plaatje. Nu je deze zijdes weet kun je de stelling van Pythagoras erop loslaten. Omdat deze aanpak altijd klopt, hebben ze er een formule van kunnen maken en deze heet de **afstandsformule**. Hierbij is **d** de afstand.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Deze formule heb ik dan ook gebruikt in de compare methode om de afstand van de verschillende tekenitems te vergelijken.

In java heb ik dat vertaald naar de volgende code:

Sqrt is een functie om een wortel te nemen van een getal.

Ik had ook nog wat problemen met het sorteren hierop. In de opdracht stond dat je comparator<drawingItem> moet gebruiken, maar hier kwam ik niet uit. Toen ben ik even een tutorial gaan kijken over het verschil tussen de twee interfaces die er zijn om te sorteren.

Comparator vs Comparable

De comparable interface zorgt ervoor dat je een compareTo methode moet implementeren. Deze methode geeft dan drie mogelijke waardes terug: 1, 0, -1. Deze waardes geven java aan hoe ze om moeten gaan met het sorteren. 1 betekent dat het object waarin de compareTo methode aangeroepen wordt groter is dan het object waarmee je het vergelijkt. 0 betekent dat ze gelijk zijn aan elkaar. -1 betekent dat het andere object groter is. Maar het hoeft niet persé -1 te zijn, dat kan eigenlijk elk negatief nummer zijn en zo ook bij een positief getal.

De Comparable interface wordt dus gebruikt om een eigen invulling te geven aan de vergelijkingsmethode op jouw eigen gemaakte objecten. Maar wat nu als je een String object anders wilt sorteren dan op alfabetische volgorde? Normaal gesproken zouden we dan een public class String aanmaken die Comparable implementeert en dan onze eigen compareTo method maken, maar dat werkt niet want we kunnen geen String class maken. Dit komt doordat het al gebruikt wordt door Java zelf. De volgende stap die je zou kunnen bedenken is misschien een class maken die overerft van String en dan de Comparable interface implementeert, maar de String class is ***final*** en kan niet overerft worden. Dus hoe willen we dan ooit de comparable interface implementeren op zo'n String class? **NIET!**

Een voorbeeld waarin je zo'n eigen sorterings-algoritme wilt maken voor bijv. een string is dat Java strings alfabetisch sorteert, maar als zo'n string een hoofdletter bevat, dan wordt deze string juist voor de string met een kleine letter geplaatst. Een string met een hoofdletter heeft dus een lagere rank/waarde. Dat is eigenlijk raar en dit is zo'n voorbeeld waar je een eigen sorterings-algoritme wilt loslaten op een class die al gereserveerd is door Java. Hiervoor kunnen we **comparator** gebruiken.

Als conclusie wordt er op internet vaak gezegd:
Comparator bied je een manier aan om een eigen sorterings-algoritme te maken voor typen waarover je geen controle hebt.
Comparable staat je toe te specificeren hoe objecten die de Comparable interface implementeren vergeleken worden.

Dus als je geen controle hebt over een class of meerdere sorteringsmogelijkheden wilt implementeren, dan gebruik **Comparator**. Anders **Comparable**.

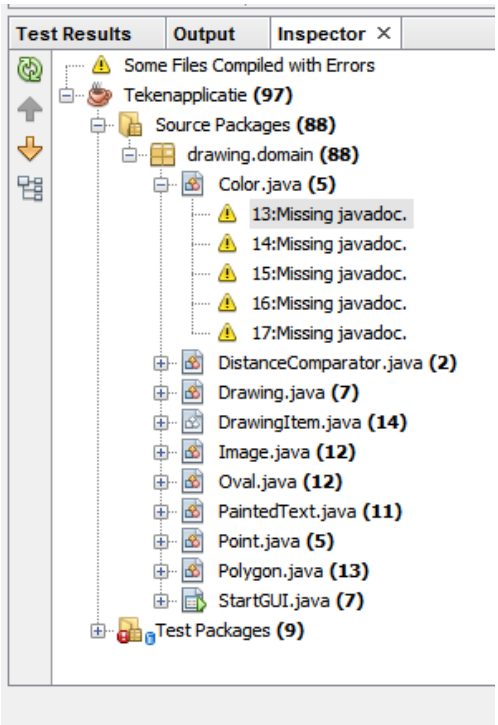
Buiten het feit dat ik nu de belangrijkste verschillen tussen deze twee weet, was dat niet het probleem waar ik tegen aanliep, ben ik achter gekomen. Want ik neigde ernaar om Comparable te gaan gebruiken, maar in de opdracht stond toch echt dat je de Comparator moest gebruiken. Op internet zag ik hoe andere mensen de Comparator implementeerden en toen zag ik dat ik deze fout implementeerde. Ik implementeerde de Comparator op precies dezelfde manier als de Comparable interface en dat is juist het grote verschil tussen deze twee: de manier waarop je ze implementeert.

Als je een lijst sorteert heb je twee overloads voor de .sort() methode. Eentje die maar één parameter bevat en dat is de lijst die je wilt sorteren en deze overload sorteert de lijst dan op basis van de compareTo methodes die in de objecten zitten doordat je de Comparable interface hebt geïmplementeerd in die objecten. De tweede overload heeft twee parameters en de tweede parameter is nu een Comparator. In deze comparator staat dan de sorterings-algoritme en de objecten in de lijst implementeren dan geen interface. Dit was mijn fout: ik liet de objecten de comparator interface implementeren. Maar dan zou je dus gaan sorteren op comparators, maar je moet juist sorteren **met behulp van** een comparator.

Door deze ontdekking is het me wel gelukt om succesvol te sorteren m.b.v. de comparator interface. Ik heb hiervoor een DistanceComparator aangemaakt waarin de sorterings-algoritme wordt ingevuld.

```
 */
public class DistanceComparator implements Comparator<DrawingItem> {

    @Override
    public int compare(DrawingItem a, DrawingItem b) {
                double aX = a.getAnchor().getX();
                double aY = a.getAnchor().getY();

                double distanceA = Math.sqrt((aX) * (aX) + (aY) * (aY));

                double bX = b.getAnchor().getX();
                double bY = b.getAnchor().getY();

                double distanceB = Math.sqrt((bX) * (bX) + (bY) * (bY));

                return (int) (distanceA - distanceB);
    }

}
```

Resultaten statische codekwaliteit(FindBugs):



Je ziet dat er nogal wat waarschuwingen en zelfs errors zijn. Die errors komen voort uit mijn unit test-project. Ik had geen tijd meer om de unit tests aan te passen op de daarna aangepaste code in de domeinklassen, dus die kloppen simpelweg niet. Verder zie je dat er veel waarschuwingen zijn, maar dat komt ook omdat ik All Analyzers erop losgelaten heb en ik schrijf simpelweg nog niet de efficiëntste code. Dat is natuurlijk wel de bedoeling om dat te leren dit semester en om te kunnen gaan met de test tooling.

Resultaten Code Coverage test(JaCoCo):

Ik heb de plug-in geïnstalleerd, maar heb het niet meer voor elkaar gekregen om uit te zoeken hoe het deze test precies in zijn werking gaat. Dat is mijn doel voor de volgende lessen.

Resultaten voorbeeldscript:

```
initComponents();
drawing = new Drawing();

Point point = new Point(5, 5);
Oval oval = new Oval(Color.BLUE, point, 30, 30, 1);

Point point2 = new Point(15, 15);
Image image = new Image(Color.RED, point2, 100, 100,
        new File("C:\\Users\\Maikkeyy\\Desktop\\School\\ICT\\Semester 3\\JCC\\Tekenapplicatie\\text

Point point3 = new Point(20, 20);
PaintedText text = new PaintedText(Color.GREEN, point3, 100, 20, "Hallo", "Arial");

Point point4 = new Point(2, 2);
Polygon polygon = new Polygon(Color.GREEN, point4, 50, 50, new Point[5], 1);

drawing.addItem(image);
drawing.addItem(oval);
drawing.addItem(text);
drawing.addItem(polygon);

Collections.sort(drawing.getItems(), new DistanceComparator());
```

Hieronder zie je dus dat de tekenitems goed gesorteerd worden op de afstand vanaf de linkerbovenhoek van het scherm. Ik heb hiervoor 0.0 gebruikt.

```
32
33          Point point4 = new Point(2, 2);
34          Polygon polygon = new Polygon(Color.GREEN, point4, 50, 50, new Point[5], 1);
35
36          drawing.addItem(image);
37          drawing.addItem(oval);
38          drawing.addItem(text);
39          drawing.addItem(polygon);
40
41          Collections.sort(drawing.getItems(), new DistanceComparator());
42
            for(DrawingItem item : drawing.getItems()) {
44              System.out.println(item.toString());
45          }
```

drawing.domain.StartGUI › StartGUI

est Results | Output - Tekenapplicatie (run) × | Inspector

```
run:
Polygon, Anchor: 2.0 : 2.0 Color: GREEN Afmeting: 50.0 x 50.0
Oval, Anchor: 5.0 : 5.0 Color: BLUE Afmeting: 30.0 x 30.0
Image, Anchor: 15.0 : 15.0 Color: RED Afmeting: 100.0 x 100.0
PaintedText, Anchor: 20.0 : 20.0 Color: GREEN Afmeting: 100.0 x 20.0
BUILD SUCCESSFUL (total time: 6 seconds)
```