Begrippen Object Oriented die elke Java programmeur moet kennen.

Alle voorbeelden staan klaar in het project BegrippenOO.
Alle voorbeelden zijn voorzien van uitgebreide commentaar.
De uitleg staat dus tussen de voorbeeldcode.

Wanneer je op interview gaat voor een stage, zullen opdrachtgevers je kennis van Java willen inschatten. Vaak gebeurt dit aan de hand van een aantal standaardvragen.
Indirect gebruik je al deze begrippen. Maar wanneer een opdrachtgever jou vraagt om specifieke object georiënteerde begrippen uit te leggen, moet je meer theoretische kennis hebben.
Deze kennis zal je trouwens ook een groot stuk vooruithelpen wanneer je documentatie raadpleegt over de klassenbibliotheken.

Inheritance – (overerving) – subclass – superclass – extends – polymorfisme

In Java is het mogelijk dat een klasse alle private variabelen en methodes kan overerven van een andere klasse.
We spreken dan van een **subclass** en een **superclass.**
Om te kunnen erven gebruiken we het keyword **extends.**

In onderstaand voorbeeld maken we een algemene superklasse Dier en laten we de subklasse Hond erven (extends) van Dier.

```
public class Dier {
    private String naam;

    public Dier(String naam) {
        this.naam = naam;
    }

    public void maakGeluid() {
        System.out.println("Het dier maakt geluid");
    }

    public String getNaam() {
        return naam;
    }

    @Override
    public String toString() {
        return "Dier{" +
                "naam='" + naam + '\'' +
                '}';
    }
}
```

```java
class Hond extends Dier {

   /*
        genereer de constructor
        deze constructor roept eerst met super de constructor aan van de
        superklasse Dier
        Aangezien naam een private variabele is van de klasse Dier,
        kan deze constructor naam niet rechtstreeks invullen.
    */
    public Hond(String naam) {
    /*
        roept de constructor van de superklasse op om de private var naam
         in te vullen
    */
        super(naam);

    }

    /*
        overschrijf de inhoud van maakGeluid met gepaste inhoud
        voor de klasse Hond.
        wanneer je letterlijk dezelfde signatuur gebruikt als die van de
        superklasse, dan spreken we van overriding.
    */
    @Override
    public void maakGeluid() {
        System.out.println("Waf waf woef");
    }
    /*
    We zagen reeds les 1 de toString functie waarmee we makkelijk de
    inhoud van een object kunnen bekijken.
    Ook hier spreken we van overriding omdat elke klasse erft van de
    superklasse Object
    Object bevat reeds de toString() functie, die we in elke klasse
    dus kunnen overschrijven.
    het is nuttig om de annotatie @override te laten staan,
    waardoor je onmiddellijk ziet dat het hier overriding betreft.
    */
    @Override
    public String toString() {
        return "Hond: " + super.toString();
    }
    /*
    deze functie roept de toString() functie op van de superklasse en
    plakt de resultaats-string aan de gegeneerde String
    Let op : super.toString()+ wordt niet gegenereerd, maar moet je er
    zelf bij typen.
    */
}
```

```java
public class TestDier {
    public static void main(String[] args) {
        Dier a = new Dier("Helga");
        Dier b = new Hond("Bassie");
        a.maakGeluid();
        b.maakGeluid();
```

```
/*
        a is een object van de klasse Dier en dus wordt de maakGeluid()-
functie
        van de klasse Dier uitgevoerd.
*/

//        Hond h = new Hond("Fluffi");

/*
        h is een object van de klasse Hond en dus wordt eerst gezocht in de
klasse Hond
        of daar een maakGeluid()-functie is, die kan worden uitgevoerd.
        Is dat niet het geval, dan wordt deze functie opgezocht in de
superklasse Dier.
        Dit mechanisme noemt men polymorfisme.
*/

//        h.maakGeluid();
```

Als je een object van de klasse Hond aanmaakt dan kan je dat object op 2 manieren in een variabele bijhouden.

```
Hond h1 = new Hond("Fluffi");
Dier h2 = new Hond("Fluffi");
```

Waarom zou men dit doen?
Een voorbeeld. Stel dat een bepaalde functie een parameter van type Dier verwacht. Zowel h1 als h2 zijn dan geschikte argumenten, hoewel h2 eigenlijk een object van type Hond is. Omwille van overerving is h2 ook een object van type Dier.

In ons object-model is een Hond ook een Dier.
Je kan een Hond dus gebruiken overal waar je een Dier nodig hebt.

Bekijk hiervoor de klasse DierenShowroom.

Oefeningen

Oefening Dieren nieuw Dier:

Maak een nieuwe subclass Kip van de klasse Dier. Wat voor geluid maakt een Kip?
Maak een nieuwe subclass Eend van de klasse Dier. Wat voor geluid maakt een Eend?

Oefening Dieren array van Dieren:

Maak een array van Dier-objecten.
Loop over deze array en gebruik DierenShowroom om elk dier te show-en.

Oefening Dieren nieuwe functie:

Maak een nieuwe functie verplaats() in de klasse Dier.

Deze functie geeft een String terug die iets zegt over de manier van voortbewegen van deze diersoort.

Bvb: de functie verplaats() voor Hond returnt: dier.getNaam() + " wandelt graag aan de leiband".

Implementeer deze functie voor de verschillende soorten Dieren.
Gebruik deze functie in de DierenShowroom.

Oefening Dieren nieuwe functie:

Maak een nieuwe functie specialiteit() in de klasse Dier.

Deze functie geeft een String terug die iets zegt over een specialiteit van deze diersoort.

Bvb: de functie verplaats() voor Kip returnt: dier.getNaam() + " legt elke dag een ei".

Implementeer deze functie voor de verschillende soorten Dieren.
Gebruik deze functie in de DierenShowroom.

Oefening 1 : Persoon – Bediende - OverzichtPersonen

Maak de klasse Persoon

```
public class Persoon {
    private String naam, voornaam, postcode;
}
```

Maak de klasse Bediende die **erft** van Persoon

```
public class Bediende extends Persoon {
    private double salaris;
}
```

Genereer in alle klassen:
Constructoren, getters en setters, toString-functie.

Maak in de **klasse Bediende** een functie verhoogSal() waardoor je het salaris kan verhogen met een bepaald bedrag.

Verander de gegenereerde toString-functie in de **klasse Bediende** zodat je ook de overgeërfde attributen opneemt in de String.
Je kan de toString functie ook uitwerken door met getters te werken.

Maak de **klasse OverzichtPersonen** aan.
Deze klasse heeft een ArrayList van alle personen en bedienden. Je kan maw objecten van de klasse Persoon en Bediende in dezelfde ArrayList plaatsen.
Denk na hoe je deze dan moet definiëren.

Maak hier 2 functies aan.
1 functie die een object toevoegt aan de ArrayList en de toString-functie die alle objecten van de ArrayList uitprint.

Oefening 2 : De leuke shop

In “De Leuke Shop” in Leuven worden boeken en gezelschapsspelletjes verkocht. Er wordt onderscheid gemaakt tussen gewone klanten en studenten, aangezien Leuven een studentenstad is.

Maak de volgende klassen.

```
public class Klant {
    private String naam, voornaam, email;
}
```

En

```
public class Student extends Klant {
    private String school;
}
```

Maak ook de klasse Artikel.

```
public class Artikel {

    private String omschrijving;
    private double prijs, korting;
}
```

Dit kortingspercentage wordt alleen toegekend aan studenten.

Genereer in alle klassen constructoren, getters en setters en zorg dat je in elke klasse de inhoud van het volledige object kan opvragen met de toString-functie.

Maak een methode ***koop(Artikel a)*** *aan,* zowel in de klasse Klant als in de klasse Student.
Deze functie noemen we ook **veelvormig** of **polymorf**. Deze functie returnt de te betalen prijs, al of niet korting inbegrepen.

De compiler weet dus niet op voorhand welke koop()-functie zal worden opgeroepen. Dit wordt op run-time niveau bepaald en is afhankelijk van het soort object.
De compiler genereert uiteraard wel instructies om de juiste functie te kunnen bepalen tijdens run-time.

Je roept de functie koop() op, maar afhankelijk van het desbetreffende object is het gedrag van de functie anders (Klant of Student). Dit noemt met polymorfisme.

Gebruik onderstaande Test-klasse

```
public class TestKlant {
    public static void main(String[] args) {
        Klant k = new Klant("Peeters", "Vera",
"Vera.Peeters@thomasmore.be");
        Student s = new Student("Janssens", "Ben",
                "Ben.Janssens@student.thomasmore.be","Thomas More");
```

```
        Artikel a = new Artikel("Monopoly", 30, 5);
        //klant koopt en betaalt
        System.out.println(k.koop(a));
        //student koopt en betaalt
        System.out.println(s.koop(a));

    }
}
```

En hoe vertalen we dit nu naar het interpreteren van onze klassenbibliotheken?

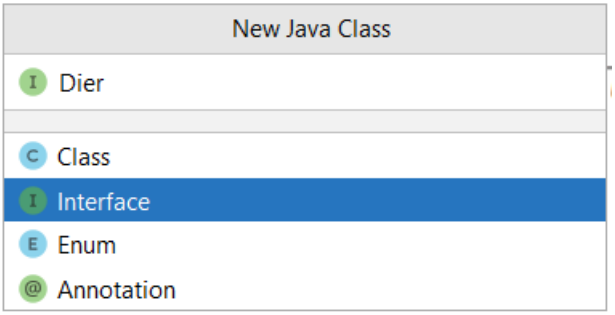
```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

We negeren voorlopig implements en komen hier later op terug.
Wat we wel zien is dat de klasse String erft van de superklasse Object (**extends**)

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Interface – implements

Een interface is eigenlijk een verzameling van functies zonder body die bij elkaar horen, een opsomming van functionaliteiten dus.
Het is een verzameling van hoofdingen of signaturen van functies. Wanneer een klasse een interface implementeert, moeten al deze functies worden uitgewerkt.
Een interface bevat geen private variabelen.




```
public interface Dier {
    void animalSound(); // interface method (does not have a body)
    void sleep(); // interface method (does not have a body)
}
```

De klasse Pig implementeert de interface Dier.
Je kan automatisch de te implementeren functies laten genereren, zodat je onderstaande code krijgt.
Merk de annotatie @override op.

```
public class Pig implements Dier {
    @Override
    public void animalSound() {

    }

    @Override
    public void sleep() {

    }
}
```

Na overriding

```
public class Pig implements Dier {
    @Override
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }

    @Override
    public void sleep() {
        System.out.println("Zzz");
    }
}
```

```
public class TestDier {
    public static void main(String[] args) {
        Pig myPig = new Pig();  // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }

}
```

Waarom gebruiken we dit?

Dit is alleen zinvol als je een class hebt met daarin een functie met parameter Dier (de Interface).

Oefening: Maak een DierenShowroom die elk soort Dier kan show-en. Doordat Dier een Interface is weten gebruikers van de klasse DierenShowroom welke methods een Dier moet implementeren.

Test uit wat er gebeurt indien je **probeert om een instantie te maken van een interface**. Dit zou niet mogen lukken, maar IntelliJ gaat op dat moment automatisch een anonymous inner class aanmaken die de interface implementeert.

Als je dit probeert in de Test-klasse, krijg je de volgende code.

```
Dier d = new Dier() {
    @Override
    public void animalSound() {

    }

    @Override
    public void sleep() {

    }
};
```

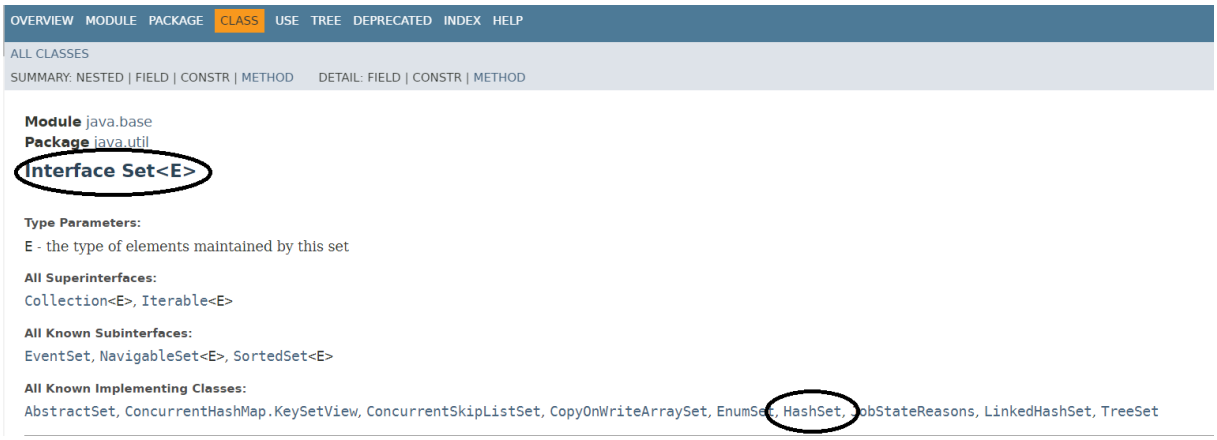
Weet je nog waar we dit gebruiken bij MusicOrganiser?

<u>En hoe vertalen we dit nu naar het interpreteren van onze klassenbibliotheken?</u>

Voorbeeld 1

Voorbeeld van een interface is **Set**.
Je kan dus geen object aanmaken van de interface Set.
Maar zoals je ziet is HashSet wel een klasse die de interface Set implementeert.

HashSet is een andere soort collection die erg lijkt op een ArrayList. Het grote verschil met een ArrayList is dat de elementen geen volgorde hebben en dat er geen dubbele elementen worden opgeslagen.
We hebben te weinig tijd om alles te bespreken, maar met wat experimenteren, kan je zeker aan de slag met een HashSet.




```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Voorbeeld 2
ArrayList implementeert dus een heel aantal interfaces.

Abstract class – abstract method

Zoals we hierboven zagen, is een Interface een beschrijving van functionaliteiten. Deze worden allemaal uitgewerkt in de klassen die de interface implementeren.

Soms kan het zijn dat een gedeelte van de functionaliteiten wordt geïmplementeerd maar een aantal functies niet. .
Dan spreekt men over een **abstracte klasse**. Een abstracte klasse wordt voorafgegaan door het keyword **abstract**.

Je kan van een abstracte klasse ook geen instanties aanmaken, hoewel een abstracte klasse private variabelen kan hebben. (een interface kan geen private variabelen hebben)

Een abstracte klasse kan een interface implementeren, maar dit is niet noodzakelijk. Een abstracte klasse kan ook op zichzelf bestaan.

Bekijk hieronder de klasse PigAbstract waarbij enkel de functie color() is geïmplementeerd. PigAbstract implementeert de interface PigInterface. Dit betekent dat alle functionaliteiten van de interface worden overgenomen, maar slechts 1 functie wordt uitgewerkt.

De functies die niet werden uitgewerkt krijgen het keyword **abstract**.

```
interface PigInterface {
    public void color();
    public void animalSound();
    public void sleep();
}
```

```
abstract class PigAbstract implements PigInterface {
    private String soort;

    public PigAbstract(String soort) {
        this.soort = soort;
    }
//implementeert de color() method.
    public void color() {
        System.out.println("Pink");
    }

    abstract public void sleep();
    abstract public void animalSound();
}
```

```
public class Pig extends PigAbstract {
    // Erft de color() method van de  PigAbstract Class
    //implementeert de sleep() en de animalSound() method

    private String roepNaam;

    public Pig(String soort, String roepNaam) {
        super(soort);
        this.roepNaam = roepNaam;
    }

    public void sleep() {
        System.out.println("Zzz");
    }

    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
}
public class TestPig {
    public static void main(String[] args) {
        Pig myPig = new Pig("Knobbelzwijn", "Bollie");  // Create a Pig
object
        myPig.color();
        myPig.animalSound();
        myPig.sleep();
        PigAbstract knorrie = new Pig("Wroetzwijn", "Knorrie");

    }

}
```

Van de klasse PigAbstract zelf zal er nooit een object worden aangemaakt omdat het een abstracte klasse is.
color() noemt men **een virtuele functie** omdat deze functie nooit voor een object van klasse PigAbstract wordt aangeroepen. De color-functie kan wel worden opgeroepen voor objecten van klassen die een extends zijn van deze abstracte klasse.

private – public – protected

private :
private variabelen en methods zijn enkel toegankelijk in elke method die deel uitmaakt van **dezelfde** klasse.
Best practice : maak alle eigenschappen / attributen van een klasse private. Maak voor alle private variabelen getters en setters aan om deze te benaderen vanuit andere klassen.

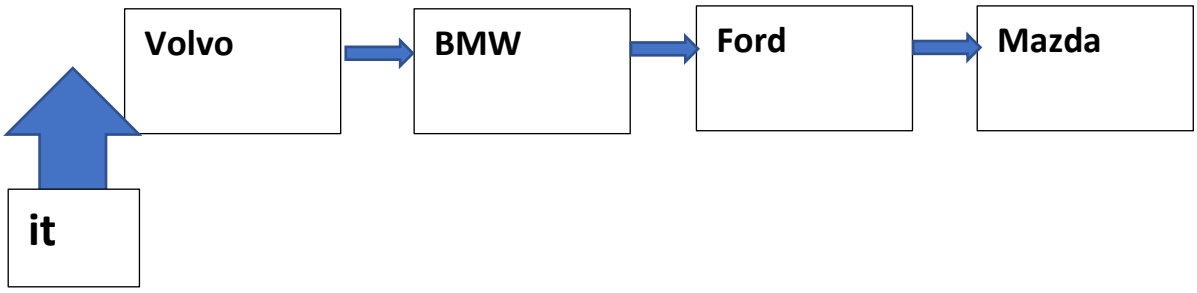
public :
In principe maken we alle methods public. Dit betekent dat de methods kunnen opgeroepen worden vanuit een andere klasse.
Soms wordt een method private gemaakt. Alleen een functie van dezelfde klasse kan deze method dan oproepen.

protected :
Dit wordt gebruikt in het kader van overerving. Je zal op het net en ook elders code tegenkomen met protected variabelen in de basis-klasse. Daardoor zijn deze variabelen rechtstreeks toegankelijk in alle klassen die erven van deze basisklasse.
We geven dit mee, maar het is geen best practice. We gebruiken enkel private voor de eigenschappen.

Iterator

Een Iterator-object is een object waarmee alle elementen van een collectie één voor één kunnen bekeken worden.

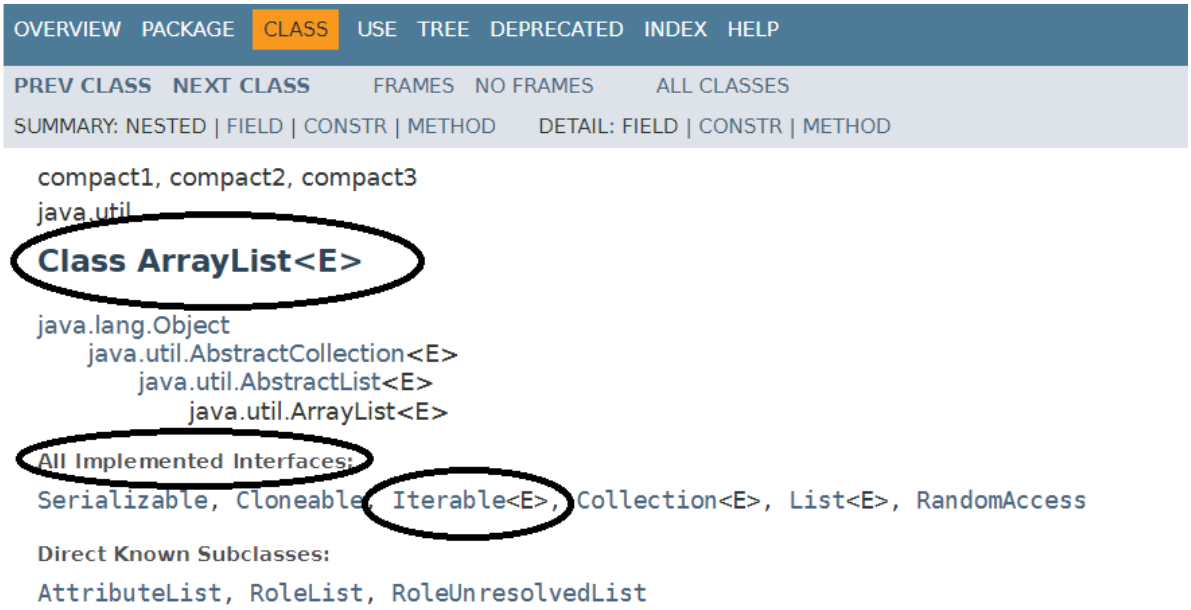
```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorEersteElement {
    public static void main(String[] args) {
       //maak een collectie
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        Iterator<String> it = cars.iterator();
        //de functie iterator() maakt een iterator voor de ArrayList cars
        while (it.hasNext())
        //zolang er nog een volgende element in de ArrayList is
        {
            System.out.println(it.next());
            //iterator haalt het volgende element op. De eerste keer haalt
            //hij het eerste element op.
        }
        //gekende code
        for (String s : cars)
        {
            System.out.println(s);
        }
    }
}
```



Hoe werkt bovenstaande code? Bekijk ook bijgevoegd schema.

STAP1 :

Maak een variabele it van de interface Iterator aan.
Doel : Deze Iterator-variabele kan elk element van de ArrayList apart aanwijzen.
Een Iterator object wordt aangemaakt door de functie iterator().
De aanroep van deze methode iterator() gebeurt 1 keer voor de ArrayList.
Vermits een ArrayList itereerbaar is, maw de interface Iterable implementeert, kan je op deze eenvoudige manier een iterator maken. Hoe weet je nu of een collection itereerbaar is? Zie onderstaande screenshot.
In de volgende module zullen we zien dat dit minder makkelijk is voor bv. een HashMap.



STAP2 :

Van de Iterator klasse gebruiken we 2 functies : **next()** en **hasNext().**

De manier waarop we een Iterator gebruiken, kunnen we als volgt beschrijven **in pseudo-code**.

```
public void gebruikIterator() {
    Iterator it = lijst.iterator();
    //de conditie test of er nog een volgend element is in de collectie
    //waarnaar de iterator wijst
    //deze conditie is onwaar als alle elementen van de collectie overlopen
    //zijn door de iterator.
    while (it.hasNext()) {
        //wijs met de iterator naar het volgende element in de collectie
        //de eerste keer zal de iterator naar het eerste element wijzen,
        //de tweede keer naar het volgende element, enz...
        it.next();
        //doe iets met dat object
    }
}
```

Uitgewerkt voorbeeld

```
public void toonProducten() {
    Iterator<Product> it = lijst.iterator();
    while(it.hasNext())
    {
        Product p = it.next();
        System.out.println(p.getOmschrijving()+" kost "+p.getPrijs());
    }
}
```

Merk op dat alle interactie met de collection gebeurt via de Iterator.
Bovenstaande code kan men net zo goed schrijven met een eenvoudige for- each-loop.
In bepaalde gevallen moet je echter met een iterator werken om een element uit de collection te verwijderen.
Dit illustreren we in onderstaande oefening.

Oefening3

Open de package Oefening3. Voer de main-functie uit.
De volgende exception treedt op bij het uitvoeren van de volgende code.

```
public void remove(double prijs)
{
    for (Oefening3.Product p : lijst )
    {
        if (p.getPrijs() < 5)
        lijst.remove(p);
    }
}
```

```
C:\Users\u0077521\.jdks\openjdk-14.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\In
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
    at Oefening3.Magazijn.remove(Magazijn.java:18)
    at Oefening3.TestMagazijn.main(TestMagazijn.java:14)
```

Wat betekent deze exception?
Het is niet toegelaten om een Collection te wijzigen terwijl je de elementen van de Collection itereert.

Hoe lossen we dit op?
Gebruik een iterator en gebruik de functie remove() van de iterator.
Pas de functie aan.

Oefening4 Transport

Open het package transport.

Bekijk de Test class.

transport oefening 1: maak een nieuw transport-middel TrainTransport en voeg die toe aan de array van transport-middelen

transport oefening 2: maak in main een array met daarin de verschillende transport-middelen.
Maak in RouteCalculator een method calculateAllPossibleRoutes met als parameter deze array
de functie print de route met de verschillende transportmiddelen

transport oefening 3: maak in Transport een nieuwe functie calculateDistance() met dezelfde parameters als calculateRoute
Deze functie returnt een integer - de berekende afstand met dit transportmiddel (test purposes: geef gewoon een getal terug)

transport oefening 4: maak in RouteCalculator een nieuwe functie shortestDistance() met als parameters :
een array van Transport objecten, een startingPoint-string en een destination-string
Deze functie returnt de route-description (String) die de kortste route beschrijft

transport oefening 5: maak in Transport een nieuwe functie calculatedTime() met dezelfde parameters als calculateRoute.
Deze functie returnt een double - de berekende tijd nodig met dit transportmiddel (test purposes: geef gewoon een getal terug)
Maak in RouteCalculator een nieuwe functie fastest() met als parameters :
een array van Transport objecten, een startingPoint-string en een destination-string
Deze functie returnt de route-description (String) die de snelste route beschrijft

Oefening5: Vorm

Maak een **interface Vorm**. Deze interface bevat twee functies:

```
public double berekenOmtrek()
public double berekenOppervlakte()
```

Maak een **klasse Rechthoek** die de interface Vorm implementeert.
De private variabelen van een rechthoek zijn lengte en breedte.

Maak een **klasse Cirkel** die de interface Vorm implementeert.
De private variabelen van een cirkel zijn straal.

Maak een **klasse Opslag**.
Deze klasse bevat een lijst van vormen.
Maak een functie aan waarmee je een vorm kan toevoegen.
Maak een functie berekenen aan die van elke vorm de details van het object toont, de oppervlakte en de omtrek.

Merk op dat je op deze manier verschillende soorten objecten kan opslaan in 1 ArrayList.

Test alles uit.

Merk op dat je de klasse Opslag zonder aanpassing kan blijven hergebruiken als er nieuwe soorten Vormen in je progamma nodig zijn. Het enige dat je moet doen is ervoor zorgen dat de nieuwe Vormen de interface Vorm correct implementeren.

Dit noemt men het **Open-Close Principle**: Open for extension, but Closed for modification.
Open for extension: Je kan je programma uitbreiden met nieuwe Vormen
Closed for modification: zonder dat je Opslag moet aanpassen

Dit is een van de basis-principes van Object Oriented Programming.

Meer weten?

https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/

https://www.javatpoint.com/difference-between-abstract-class-and-interface

https://www.w3schools.com/java/java_abstract.asp