hogeschool Windesheim

Techniek – HBO-ICT

Naam: Studentnummer: Klas:

Practicum herk. JAVA Theorie (119893)

Vakcode : ICT.P.JAVA1.V21 (ICT.P.JAVA1.V20/V19/V18/V17) (t1)
Datum : dinsdag 14 juni 2022
Tijd : 11:30 - 13:30 uur

Klas:	Lokaal:	Aantal:
ICTM2a t/m p, ICTM2tt	…	226
	…	
	…	

Opgesteld door : Kevin de Korte; Wilco Moerman
Docenten : WPH01; KNJ24; RWM02; SSM36; LNR08; KEK01; MNC07; VEE02; CNW01; HZJ40; BHA40;
Gecontroleerd door : Jairo Hernandez; Aminah Balfaqih

Rekenmachine : alle rekenmachines toegestaan
Literatuur : alles (boeken, internet, aantekeningen)
Overige hulpmiddelen : laptop

Opgaven inleveren : ja

CONTROLEER VOORAF DE VOLGENDE GEGEVENS:

Dit tentamen bevat:

5 opgaves

26 genummerde pagina's

Waarschuw de surveillant als één van deze aantallen niet klopt!

Studentnummer	Naam	Klas	Cijfer
Tijd van inleveren:			

De regels en de punten

Het gebruik van telefoons/social media/forums/dropbox en alles wat je in contact brengt met anderen, is tijdens de toets niet toegestaan.

Het gebruik van internet om informatie op te zoeken is wel toegestaan.

Je mag dus **wel zoeken/googlen.** En je mag bv. *wel* iets lezen op een forum zoals *Stackoverflow,* maar je mag er **geen vragen** stellen.

In totaal zijn **100** punten te behalen. Het cijfer is het aantal behaalde punten gedeeld door 10. Het laagst te behalen cijfer is een 1, het hoogste een 10.

Voorbereiding

Alle in de toets getoonde code en het klassendiagram vind je op **ELO** in de folder **`"inleverpunt theorie 14-06-2022"`** in **`"startcode_theorie_14-06-2022.zip".`**

De codes met voorbeelden zijn ***geen volledige tests.*** Controleer zelf of je code *alles* doet wat de vraag staat. De voorbeelden staan in de klasse `Main` (in de `.zip`) in uitgecommente `main(...)`-methodes. Deze `Main`-klasse wordt *niet* nagekeken.

Als een regel code iets *print,* staat dat erachter *in commentaar,* na **`//>>`**

Voorbeeld: onderstaande System.out.println("Hoi") heeft dus output "Hoi"

```
// dit is gewoon commentaar
System.out.println("Hoi"); //>> Hoi
int x = 10; // ook gewoon commentaar
```

Een spelfoutje of een spatie teveel in een `toString()` of `print` is geen probleem.

Je mag op de papieren toets aantekeningen maken. Ook mag je het klassendiagram losmaken van de rest om het naast een vraag te leggen. **nakijkmodel:**
Over dit nakijkmodel en de puntenverdeling:

let op: in dit document staan alle nakijkregels globaal, zodat het leesbaar blijft. De gerunde tests (en handmatige beoordeling indien nodig) zijn bepalend voor het cijfer.

Omdat de vragen(vaak) onafhankelijk zijn, is ook zoveel mogelijk geprobeerd fouten in de ene vraag niet te laten meetellen bij de volgende vraag. Maar dat kan mis gegaan zijn. **Let hierop** bij controle/inzage van je werk.

waar je op kunt letten bij de inzage:

- doorrekenfouten
- dingen die fout zijn gerekend doordat er een rare **typo** in staat
  (we doen hier niet aan correct Nederlands! De plugin die automatisch je cijfer aanpast, komt pas over een paar jaar ;-)

Zie de voorbeelduitwerking voor een mogelijke goede uitwerking, met her en der wat uitleg.

Opgave 1: koffers [25 punten]

In deze opgave maak je een `Koffer` met beveiliging.

```
public class Koffer {

	public int grootte;
	public String inhoud;
	public int code;

	// voor vraag (c)
	public ArrayList<Koffer> verdeel(int aantal, String naam) {
		int nieuweGrootte = 1 + grootte / aantal;
		ArrayList<Koffer> lijstje = new ArrayList<>();
		for (int i = 0; i < aantal; i++) {
			String nieuweInhoud = inhoud + "_"
					+ naam.toLowerCase() + "#" + (i + 1);
			lijstje.add(new Koffer(nieuweGrootte, nieuweInhoud));
		}
		return lijstje;
	}
}
```

a) [5 punten]
Maak twee constructors in de klasse Koffer met de volgende eisen:

- een constructor met de `grootte` en `inhoud` als inputs
- een andere constructor met alleen de `grootte` als input
- de default waarde voor attribuut `inhoud` is `"leeg"`
- de minimale `grootte` voor een `Koffer` is **`10.`** Indien de opgegeven `grootte` kleiner is dan **`10,`** dan wordt de `grootte` **`10`**
- bij het aanmaken wordt `"nieuw:"` geprint met de `inhoud` en `grootte`
- de constructor met maar 1 input, mag **maar één** Java-statement bevatten, dus maar één puntkomma. (Als dat niet lukt, zorg dan in ieder geval dat de code compileert.)

werking van Koffer-constructors Output staat in comments, herkenbaar aan //>>
`Koffer k1 = new Koffer(555, "JAVA-boek");  //>>  nieuw: JAVA-boek (555)` `Koffer k2 = new Koffer(3);  //>>  nieuw: leeg (10)`

Let op: (b), (c) en **(d)** kunnen los van elkaar gemaakt worden.

nakijkmodel:
5 punten: werking van de constructors van Koffer:

- 2 punten aftrek: goede waardes voor grootte en inhoud worden niet opgeslagen in attributen
- 3 punten aftrek: (veel) te lage waardes worden opgeslagen in grootte
- 2 punten aftrek (per keer): a grootte input rond de 10 mis gaat
- 2 punten aftrek: Als de Koffer(int)-constructor de inhoud niet de default waarde "leeg" geeft
- 2 punten aftrek: goede waardes voor grootte en inhoud worden niet opgeslagen in attributen
- 3 punten aftrek: (veel) te lage waardes worden opgeslagen in grootte

- 2 punten aftrek (per keer): a grootte input rond de 10 mis gaat
- 2 punten aftrek: meer dan 1 Java statement in de construtor (geen this(..) gebruikt)

b) [10 punten]
Pas ***encapsulatie*** toe om attributen te beschermen tegen wijzigingen van buitenaf. Maak daarom ook de setter **`setCode`** voor `code`, die een nieuwe `code` toekent aan de `Koffer`, als aan de volgende eisen wordt voldaan:

- de `code` mag maar **één** keer ingesteld worden
- de nieuwe `code` heeft **4** cijfers (dus **`1000`** t/m **`9999`** zijn geldig)

Print **`"mag niet"`** als niet aan de eisen voldaan wordt.

werking van setCode

```
Koffer k1 = new Koffer(77, "mobieltje");  //>>  nieuw: mobieltje (77)
// te klein of te groot
k1.setCode(1);                            //>>  mag niet
k1.setCode(1000000);                      //>>  mag niet

// goed
k1.setCode(5678);

// mag maar 1x ingesteld worden
k1.setCode(1234);                         //>>  mag niet
```

nakijkmodel:
10 punten: setCode-werking:

- 1 punt aftrek (per keer): attribuut niet private
- 3 punten aftrek: code kan kleiner dan 1000 worden
- 1 punt aftrek (per keer): setCode(int) werkt niet rond ondergrens (1000)
- 1 punt aftrek: foutmelding mist als code te kort is
- 3 punten aftrek: geen foutmelding als code te lang is
- 1 punt aftrek (per keer): setCode(int) werkt niet rond de bovengrens (9999)
- 1 punt aftrek: geen foutmelding als de code te lang is

- 3 punten aftrek: code die goed is, niet wordt geset
- 5 punten aftrek: code kan meer dan 1x gewijzigd worden met setCode
- 2 punten aftrek: geen foutmelding bij 2e keer setCode

c) [5 punten]
De **verdeel**-methode maakt een aantal nieuwe koffers aan om de `inhoud` daarover te verdelen. De nieuwe koffers worden in een `ArrayList` verzameld.

De berekening en de nieuwe inhoud zijn al **goed** (en mogen niet veranderd worden).

De `verdeel`-methode kan ***crashen.*** Vul de methode aan, zodat de crashes **niet** meer kunnen plaatsvinden:

- je mag **geen** gebruik maken van een **`if`**
- er moet **`"dat is wiskundig onmogelijk!"`** uitgeprint worden als het om een **wiskundige** fout gaat
- bij alle **overige** fouten, print de methode **`"er is een fout opgetreden"`**
- bij alle fouten moet **`null`** gereturnd worden

werking van verdeel

```
String input = null;
Koffer k1 = new Koffer(33, "laptop");  //>>  nieuw: laptop (33)

ArrayList<Koffer> lijstje;
lijstje = k1.verdeel(2, "DEEL");       //>>  nieuw: laptop_deel#1 (17)
                                       //>>  nieuw: laptop_deel#2 (17)

// gevulde Arraylist (inhoud niet belangrijk voor deze vraag).
System.out.println(lijstje.size());    //>>  2

lijstje = k1.verdeel(0, "oeps!");      //>>  dat is wiskundig onmogelijk!
System.out.println(lijstje);           //>>  null

lijstje = k1.verdeel(1, input);        //>>  er is een fout opgetreden
System.out.println(lijstje);           //>>  null
```

nakijkmodel:
5 punten: verdeel-methode tegen crashen beschermen m.b.v. try-catch:

- 5 punten aftrek: if gebruikt
- 4 punten aftrek: aanroepen van de verdeel-methode werkt niet bij geldige inputs
- 3 punten aftrek: NullPointerException niet afgehandeld
- 1 punt aftrek: return waarde is niet null in geval van een (gevangen) exception
- 2 punten aftrek: de wiskundige fout (een deling door 0) wordt niet opgevangen
- 1 punt aftrek: return waarde is niet null

d) [5 punten]
Maak de **`getInhoud`**-methode, die de `grootte` en `inhoud` samen als `String` returnt als je de juiste `code` hebt meegegeven.

Als de `code` onjuist is, gaat de `Koffer` in *"lockdown"*. Je kunt de inhoud dan **nooit** meer bekijken, ook niet met de juiste `code`.

Return `"---lockdown---"` als een verkeerde `code` gebruikt wordt.

De `code` veranderen kan uiteraard ook niet meer nadat de koffer in *"lockdown"* is.

werking van de lockdown functionaliteit

```
Koffer k1 = new Koffer(50, "telefoon");  //>>  nieuw: telefoon (50)
k1.setCode(7777);

System.out.println(k1.getInhoud(3400));  //>>  ---lockdown---
System.out.println(k1.getInhoud(5600));  //>>  ---lockdown---

// goede code werkt ook niet meer
System.out.println(k1.getInhoud(7777));  //>>  ---lockdown---
k1.setCode(7777);                        //>>  mag niet

Koffer k2 = new Koffer(33, "boek");      //>>  nieuw: boek (33)
k2.setCode(1234);
System.out.println(k2.getInhoud(1234));  //>>  33: boek
```

nakijkmodel:
5 punten: getInhoud-methode:

- 4 punten aftrek: getInhoud toont de inhoud niet bij de juiste code
- 4 punten aftrek: getInhoud toont de inhoud bij een onjuiste code
- 2 punten aftrek: code nog veranderd kan worden na lockdown
- 1 punt aftrek: er wordt geen lockdown weergegeven

Opgave 2: stations en spoorlijnen [20 punten]

a) [10 punten]
`Spoorlijn`-objecten hebben een `ArrayList` met stations. De eerste is het beginpunt, de laatste het eindpunt. We willen `Spoorlijn`-objecten kunnen vergelijken. Ze zijn hetzelfde als als de begin- en eindstations hetzelfde zijn. De stations die er tussen liggen, en de volgorde, maken niet uit.

```
public class Spoorlijn {

	private String naam;
	private ArrayList<Station> stations;

	public Spoorlijn(String n) {
		naam = n;
		stations = new ArrayList<>();
	}

	public Station getLaatsteStation() {
		return stations.get(stations.size() - 1);
	}

	public int lengte() {
		return stations.size();
	}

	public String toString() {
		return naam + ": " + stations.get(0) + " --> "
				+ getLaatsteStation() + " (" + lengte() + ")";
	}

	public void voegStationToe(String station) {
		Station s = new Station(stations.size(), station);
		stations.add(s);
	}
}
```

```
public class Station {

	private int id;
	private String naam;

	public Station(int i, String n) {
		id = i;
		naam = n;
	}

	public String toString() { return naam; }
}
```

Maak de **equals**-methode voor `Spoorlijn`. Er geldt:

- de methode **overschrijft** *(override)* de `equals`-methode die je van `Object` overerft
- als beide `Spoorlijn`-objecten hetzelfde **begin- en eindstation** hebben, wordt `true` gereturned. Volgorde is niet belangrijk, en tussenstations ook niet:
  `Spoorlijn` **A** `-->` B `-->` **Z** is gelijk aan `Spoorlijn` **A** `-->` X `-->` Y `-->` **Z**
  `Spoorlijn` **A** `-->` ... `-->` **Z** is gelijk aan `Spoorlijn` **Z** `-->` ... `-->` **A**
- stations zijn gelijk als ze dezelfde **`naam`** hebben

Vul de code aan, zodat de `equals`-methode voldoet aan de bovenstaande eisen. Je mag beide klassen aanpassen. Hieronder volgt een voorbeeld van de werking.

werking van equals van Spoorlijn

```
Spoorlijn een = new Spoorlijn("IC");
een.voegStationToe("Zwolle");
een.voegStationToe("Deventer");

Spoorlijn twee = new Spoorlijn("Sprinter");
twee.voegStationToe("Zwolle");
twee.voegStationToe("Heino");
twee.voegStationToe("Raalte");
twee.voegStationToe("Wierden");
twee.voegStationToe("Deventer");

// begin- en eindpunt hetzelfde
System.out.println(een.equals(twee));  //>>  true

Spoorlijn drie = new Spoorlijn("IC");
drie.voegStationToe("Deventer");
drie.voegStationToe("Zwolle");

// eindpunt van ene is beginpunt van de ander, en andersom.
System.out.println(een.equals(drie));  //>>  true

Spoorlijn vier = new Spoorlijn("stoomtrein");
vier.voegStationToe("Zwolle");
vier.voegStationToe("Middle-of-Nowhere");

// ander begin- of eindpunt
System.out.println(een.equals(vier));  //>>  false
```

nakijkmodel:
4 punten voor `equals(Object)`, dus voor de juiste **signatuur** van de methode. De vraag vroeg naar de equals die je van Object overerft en moet overschrijven. Object weet niks af van `equals(Spoorlijn)`. Object heeft alleen `equals(Object)`.

6 voor de **correcte werking** van de methode:

- 3 eraf als je **Strings niet met equals** vergelijkt. (Dat moet je namelijk **altijd** doen in Java)

- 3 eraf als het fout gaat als je vergelijkt met een `Spoorlijn` met exact **dezelfde waardes.** ...

Deze beoordelingswijze betekent dat je 4 punten krijgt voor het maken van de juiste methode: `public boolean equals(object...)` ook als die alleen maar **`return false;`** of zoiets zou bevatten. In objectgeorienteerde talen (java, C#, etc) is het van groot belang dat je als developer specificeert wanneer iets gelijk is.

Let op: (a) en **(b)** kunnen los van elkaar gemaakt worden.
4 punten: correcte signatuur van methode: public boolean equals(Object)
6 punten: werking v/d equals-methode:

- 6 punten aftrek: traject.equals(traject) returnt false
- 6 punten aftrek: bij compleet verschillende Spoorlijn-objecten true returnen
- 6 punten aftrek: het mis gaat als twee Spoorlijn-objecten identieke inhoud (zelfde Station-objecten) hebben: A --> B en A --> B
- 3 punten aftrek: het mis gaat als twee Spoorlijn-objecten identieke inhoud hebben, behalve naam
- 2 punten aftrek: het mis gaat als twee Station-objecten alleen verschillen in id
- 4 punten aftrek: het mis gaat als twee Spoorlijn-objecten identieke inhoud hebben: A --> B en A --> B
- 3 punten aftrek: het gaat mis als twee Spoorlijn-objecten A --> B en B --> A gelijk zijn.
- 3 punten aftrek: Spoorlijn-objecten zelfde begin- en eindpunt (A --> B en A --> X --> Y --> Z --> B, en dus gelijk), maar hebben andere ArrayList-inhoud

b) [10 punten]
De langste `Spoorlijn` die aangemaakt is, moet onthouden worden. Met de **`printLangste()`**-methode moet deze geprint worden.

Als twee spoorlijnen even lang zijn, moet de *laatst aangemaakte* onthouden worden.

Met de `lengte()`-methode krijg je de lengte van een `Spoorlijn`.

werking van printLangste()

```
Spoorlijn.printLangste();  //>>  null

Spoorlijn een = new Spoorlijn("IC");
een.voegStationToe("Zwolle");
een.voegStationToe("Deventer");

Spoorlijn.printLangste();  //>>  IC: Zwolle --> Deventer (2)

// Spoorlijn twee is langer dan Spoorlijn een, dus twee wordt onthouden
Spoorlijn twee = new Spoorlijn("Sprinter");
twee.voegStationToe("Zwolle");
twee.voegStationToe("Wezep");
twee.voegStationToe("Nunspeet");
twee.voegStationToe("Harderwijk");
twee.voegStationToe("Utrecht");

Spoorlijn.printLangste();  //>>  Sprinter: Zwolle --> Utrecht (5)

// Spoorlijn drie is korter dan de langste
Spoorlijn drie = new Spoorlijn("Sprinter");
drie.voegStationToe("Groningen");
drie.voegStationToe("Assen");
drie.voegStationToe("Zwolle");

Spoorlijn.printLangste();  //>>  Sprinter: Zwolle --> Utrecht (5)
```

nakijkmodel:

6 punten: een static Spoorlijn attribuut om langste bij te houden.:

- 6 punten aftrek: geen static Spoorlijn attribuut (zonder dat werkt het niet)
- 2 punten aftrek: de initiele waarde van langste is niet null
- 2 punten aftrek: als niet de laatste onthouden wordt bij even lange lijnen

4 punten: een static methode maken en aanroepen:

- 4 punten aftrek: static methode niet aanwezig of niet goed aangeroepen

Opgave 3: geheimschrift [25 punten]

In deze opgave ga je een **array** gebruiken voor geheime berichten.

```
public class Geheimschrift {

	public Geheimschrift(int lengte) { /*...todo...*/ }

	public void maakGereed() { /*...todo...*/ }

	public void verberg(int positie, int stap, String[] tekst) {
		/*...todo...*/
	}
}
```

Let op:
In deze hele opgave mag je er vanuit gaan, dat elke `String` in een `array` maar **1** symbool (letter, leesteken, etc) is. De arrays kun je zien als woorden en `Strings` die erin zitten als letters.

In de zip met de startcode vind je in klasse `Geheimschrift` ook drie methodes die je kunt gebruiken (maar niet aan mag passen) in **(b)** en **(c).**

a) [5 punten]
Geef de klasse `Geheimschrift` een attribuut dat een String-array is, genaamd **`code`.**

Maak ook de `Geheimschrift`-**constructor** af:

- de input is de gewenste lengte van het `code`-attribuut
- de constructor zorgt er dus voor dat het attribuut `code` die lengte krijgt.

nakijkmodel:
2 punten: declaratie van array-attribuut:

- 2 punten aftrek: array van verkeerd type

3 punten: initialiseren van array-attribuut in constructor:

- 3 punten aftrek: code wordt niet geinitialiseerd in de constructor
- 3 punten aftrek: String[]code niet de juiste lengte krijgt in de constructor

geen punten voor de **hele** opgave, als je een **`ArrayList`** gebruikt ipv een array. Deze opgave test of je met `arrays` kunt omgaan, niet of je er een alternatief voor kunt verzinnen.

b) [5 punten]
programmeer de **`maakGereed`**-methode:

- deze methode gebruikt de **`randomSymbool`**-methode om op elke plek in de array een willekeurig symbool te zetten
- de `maakGereed`-methode wordt aangeroepen in de **constructor**

Maak ook de **`print`**-methode die de inhoud van het `code`-attribuut uitprint.
Met `System.out.`**`print`** (en niet `println`**`)`** kun je tekst op dezelfde regel printen.

print en maakGereed-methodes

```
Geheimschrift g = new Geheimschrift(6);
g.print();  //>>  *-,-..
```

Let op: de *random* symbolen kunnen bij jou anders zijn dan in de voorbeelden.

nakijkmodel:
2 punten: Printen van code-array:

- 2 punten aftrek: niet alle elementen van de array worden uitgeprint
- 1 punt aftrek: niet alle elementen van de array worden uitgeprint

3 punten: juiste werking van maakGereed() van Geheimschrift:

- 3 punten aftrek: de array niet gevuld is met random tekens

c) [15 punten]

maak de **verberg**-methode. Deze krijgt als inputs een tweetal `ints` (`positie` en `stap`) en een `String-array` (`tekst`). De methode "verstopt" de `tekst` in het attribuut `code`. De letters worden met gelijke tussenafstanden in de array geplaatst:

- de eerste `String` in de `tekst`-array wordt op plek **`positie`** in het array-attribuut `code` gezet
- de tweede `String` in de `tekst`-array komt op plek **`positie + stap`** in `code`
- de derde op **`positie + stap + stap`**
- de vierde op **`positie + stap + stap + stap`**

  enz..

Let op: Ter controle kun je gebruik maken van de `cryptischeVerberg`-methode die de boodschap op dezelfde manier verbergt (maar op een veel te ingewikkelde manier). Hiermee kun je andere tests voor je `verberg`-methode maken.

Bij **foute inputs** kan `verberg(...)` ***crashen*** (`ArrayIndexOutOfBoundsException`):

- zorg ervoor dat dat **niet** meer kan gebeuren
- je mag hiervoor **geen** `try` en `catch` gebruiken
- als het niet gaat passen, wordt **`"fout: Array te klein"`** geprint
- het `code`-attribuut mag in zo'n geval **niet** veranderen

werking van verberg

```
Geheimschrift g = new Geheimschrift(12);
g.print();                          //>>  *-,-...+--.-

String[] java = {"J", "A", "V", "A"};
g.verberg(3, 2, java);
g.print();                          //>>  *-,J.A.V-A.-

// ------------ decoderen met de juiste start en stap --------------

String decoded = g.ontcijfer(3, 2, java.length);
System.out.println(decoded);        //>>  JAVA

// ------- decoderen met verkeerde start of stap werkt niet --------

String decodedFout = g.ontcijfer(4, 2, java.length);
System.out.println(decodedFout);    //>>  ..-.

decodedFout = g.ontcijfer(3, 4, java.length);
System.out.println(decodedFout);    //>>  JV-

// ------------------------- past nog net! ---------------------------

Geheimschrift g2 = new Geheimschrift(16);
g2.verberg(1, 4, java);
g2.print();                         //>>  ,J,.,A+.+V*..A--
decoded = g2.ontcijfer(1, 4, java.length);
System.out.println(decoded);        //>>  JAVA

// -------------------------- past niet! ----------------------------

Geheimschrift teKlein = new Geheimschrift(7);
String[] pastNiet = {"P", "A", "S", "T", " ", "N", "I", "E", "T"};
teKlein.verberg(1, 2, pastNiet);    //>>  fout: Array te klein

// de array is niet veranderd:
teKlein.print();                    //>>  ,,*.*+-
```

nakijkmodel:
5 punten: verberg met positie = 0 en stap = 1:

- 5 punten aftrek: de array wordt niet gevuld wordt met gegeven letters

4 punten: verberg met positie > 0 en stap = 1:

- 4 punten aftrek: de input wordt niet goed 'verborgen' bij startpositie > 0 en stap = 1

3 punten: verberg met positie > 0 en stap >= 2:

- 3 punten aftrek: de input wordt niet goed 'verborgen' bij startpositie > 0 en stap > 1

3 punten: methode verberg crasht niet:

- 1 punt aftrek: het gaat mis als de laatste letter heel ver voorbij einde v/d array komt
- 1 punt aftrek: het gaat mis als eerste letter op plek 0 moet komen
- 1 punt aftrek: het gaat mis als er letters 'voor' de eerste arrayplek komen (index < 0)
- 1 punt aftrek: het gaat mis als de laatste letter 1 plek verder komt dan het einde v/d array
- 1 punt aftrek: het gaat mis als de laatste letter op de laatste plek in de array moet komen
- 1 punt aftrek: de array niet hetzelfde is gebleven

Opgave 4: getallen [10 punten]

Een behulpzame wiskundige heeft een wiskunde-gerelateerde klasse gemaakt (genaamd `Getal`). Er zitten helaas nog wat fouten in.

```
public class Getal {

	public int waarde;

	public Getal(int w) {
		waarde = w;
	}

	public String toString() {
		return "Getal: " + waarde;
	}

	// optellen met int
	public Getal plus(int x) {
		waarde = waarde + x;
		return new Getal(waarde);
	}

	// optellen met Getal
	public Getal plus(Getal getal) {
		getal.waarde = waarde + getal.waarde;
		return getal;
	}
}
```

De bedoeling van de **`plus`**-methodes is dat je `int` of een `Getal` bij een ander `Getal` kunt optellen. Maar als je **4** bij **1** optelt, is het natuulijk *niet* de bedoeling dat de 4 of de 1 daarna ineens van waarde veranderd zijn!
Een 4 is een 4 en dat moet zo blijven!

bugs: Getal-objecten veranderen van waarde

```
Getal vier = new Getal(4);
Getal twaalf = vier.plus(8);

System.out.println(vier);              //>>  Getal: 12
System.out.println(twaalf);            //>>  Getal: 12

Getal negenEnNegentig = new Getal(99);
Getal drie = new Getal(3);

Getal honderdTwee = negenEnNegentig.plus(drie);

System.out.println(drie);              //>>  Getal: 102
System.out.println(negenEnNegentig);   //>>  Getal: 99
System.out.println(honderdTwee);       //>>  Getal: 102
```

Repareer alle fouten in klasse `Getal`, zodanig dat de optellingen werken **en** de `Getal` -objecten niet veranderen door de methodes.

verwachte werking (zonder bugs)

```
Getal vier = new Getal(4);
Getal twaalf = vier.plus(8);

System.out.println(vier);              //>>  Getal: 4
System.out.println(twaalf);            //>>  Getal: 12

Getal negenEnNegentig = new Getal(99);
Getal drie = new Getal(3);

Getal honderdTwee = negenEnNegentig.plus(drie);

System.out.println(drie);              //>>  Getal: 3
System.out.println(negenEnNegentig);   //>>  Getal: 99
System.out.println(honderdTwee);       //>>  Getal: 102
```

nakijkmodel:

5 punten: probleem omtrent new vs. reference:

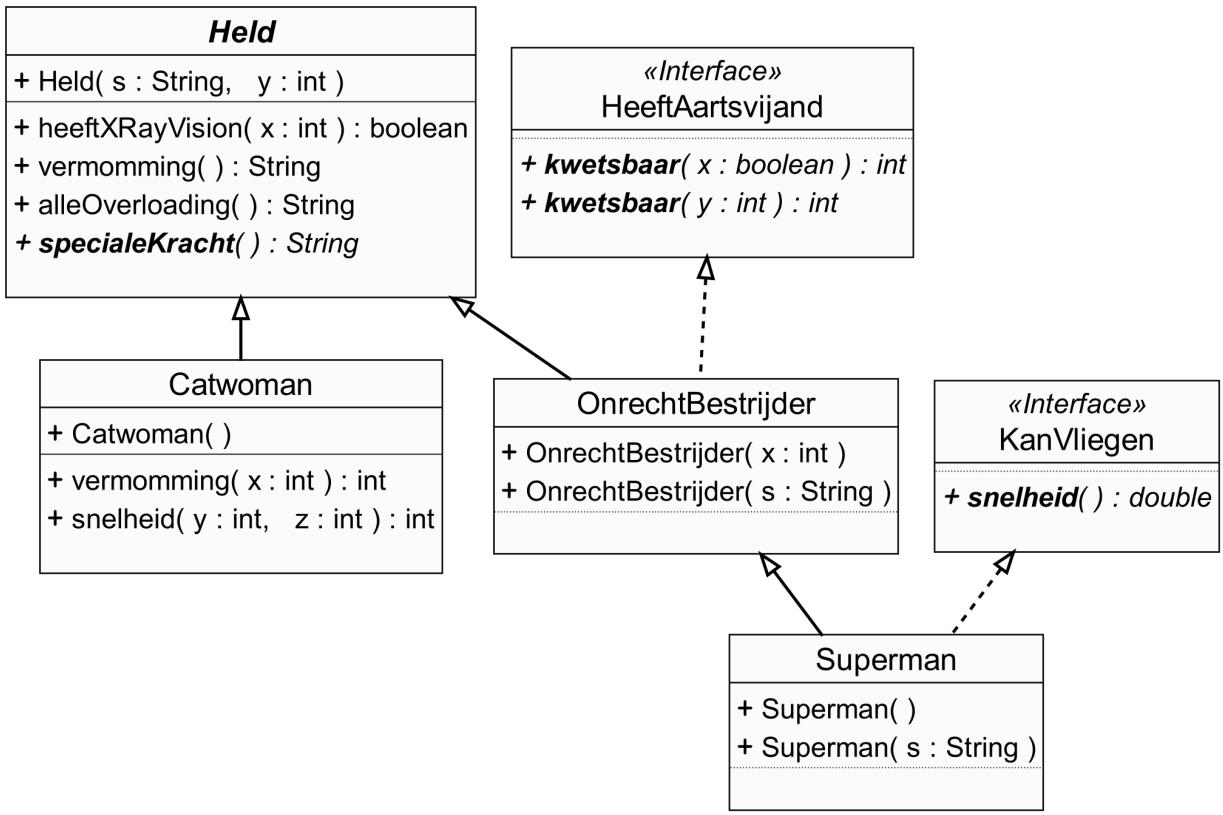
- 5 punten aftrek: probleem omtrent new vs. reference

5 punten: probleem omtrent scoping:

- 5 punten aftrek: probleem omtrent scoping

Opgave 5: Kryptonite [20 punten]

Een fan van *superhelden* heeft dit klassendiagram opgesteld: ***(vetgedrukt + cursief*** geeft "abstract" aan)



Methodes die door klassen moeten worden geimplementeerd vanwege een interface of abstracte klasse, staan **niet** in het diagram vermeld.

a) [10 punten]
Schrijf code op basis van dit diagram, dus met de correcte **klassen, overerving, interfaces** en **methodes** (constructors komen in **(c)):**

- de implementatie moet **zo weinig mogelijk** methodes bevatten

- voor elke overgeërfde methode zul je dus moeten bepalen of deze **overschreven** *(overriding)* moet worden, aan de hand van de regels hieronder

Regels voor wat methodes moeten doen:

- als een methode een *getal* returnt, return dan altijd `42`
- als een methode een `String` returnt, dan **moet** je de naam van de klasse en de methode gebruiken, dus bv.: `return "Superman --> vermomming";`

nakijkmodel:
2 punten aftrek voor **missende klassen/interfaces**

1 punt aftrek voor missende/overbodige functies en missende/foute/overbodige extends/implements.

Let op: een `extends/implements` missen heeft tot gevolg dat 1 of 2 functies missen in de klasse die het had moeten extenden/implementen. Dit is **geen** doorrekenfout, maar bewuste keuze. Deze opgave gaat met name over **overerving/interfaces,** dus het missen van een extends/interface kost je dus de 'prijs' van het niet hebben an extenden/implementen plus het missen van de methodes.

Er is **niet** gecontroleerd op wat de methodes returnen, alleen dat ze aanwezig zijn en de juiste signatuur hebben.

De eis dat er **zo min mogelijk methodes** moesten zijn, heeft het volgende effect: als een methode een `int` moet returnen, dan is die methode alleen nodig in de 'parent/super' klasse. Want elke overerving daarvan krijgt die methode en een int is een int, dus dergelijke methodes moesten in de kind-klassen niet overgeschreven worden want de return waarde is altijd hetzelfde.

Maar een methode die een `String` returnt, **moet** volgens de vraag de naam van de **klasse** + methode returnen. Dat kan uiteraard niet door de 'super' methode afgehandeld worden (behalve als je met `instanceof` zou gaan werken). De makkelijkste oplossing is dus, om de methoden die een String returnen, *wel* in de 'kind' klassen te zetten.

n.b. in het diagram stonden ook methodes die niets met overerving/interface te maken hadden maar "toevallig" dezelfde naam hadden als iets dat in een interface of super klasse stond. Die moeten uiteraard ook gewoon geimplementeerd worden.

Let op: opgaves **(b)** en **(c)** kunnen los van elkaar gemaakt worden.

10 punten: klassendiagram: interfaces, overerving, methodes met juiste signatuur

b) [5 punten]
Voeg de methode **`alleOverloading()`** toe aan klasse **`Held.`** Deze returnt een `String` met de **namen** van alle methodes waarbij sprake is van **overloading.**

Als je bv. denkt dat de methodes `heeftXRayVision`, `specialeKracht` en `vermomming` alle gevallen zijn van *overloading,* dan ziet `alleOverloading()` er zo uit (volgorde van de namen maakt niet uit):

```
public String alleOverloading() {
   return "specialeKracht + vermomming + heeftXRayVision";
 }
```

nakijkmodel:
Bij overloading van **methodes** gaat het om methodes met **dezelfde naam** in **dezelfde** klasse. op basis van het **diagram: `vermomming`** en **`kwetsbaar.`** De ene (`vermomming`) komt via overerving in een klasse terecht waar al een andere methode met die naam aanwezig is.

(methodes genaamd **`snelheid`** zaten in **verschillende** klassen, dat is geen overloading).

Uiteraard is niet gelet op of er een **+** stond of iets anders. Spelfouten in de methodes zijn uiteraard ook niet fout gerekend.

Sommigen hebben deze vraag anders opgevat. Gegeven de code die je bij (a) hebt gemaakt (of ie nu goed met het diagram overeenkomt of niet) welke methodes zijn in die **gemaakte** code de voorbeelden van overloading. Bij het nakijken is hier rekening mee gehouden. Er is gekeken of het bij (b) gegeven antwoord beter met de gemaakte code overeenkwam, of dat het antwoord beter met het diagram overeenkwam. Op deze manier kun je bij (b) punten verdienen, ondanks dat je bij (a) veel fouten hebt gemaakt (en andersom kan ook gebeuren: als je bij (a) het diagram perfect hebt geimplementeerd, maar bij (b) geen antwoord of een verkeerd antwoord geeft).

In beide gevallen geldt:

- **antwoord compleet goed** (alle methodes die overloading gevallen zijn, zijn genoemd, maar geen enkele andere methode): alle punten
- **deels goed:** alle overloading gevallen genoemd, maar ook 1 niet-overloaded methode: 3 punten
- **deels goed:** maar 1 overloading genoemd (als er 2 waren, in gemaakte code of digitale versie v/h diagram) maar geen foute gevallen
- **niet goed:** geen gevallen van overloading als antwoord, of meer dan 1 fout antwoord: 0 punten

speciaal geval: als je bij (b) als antwoord naar klassen of constructoren verwijst (en niet naar methodes) dan is dat anwoord **niet goed** gerekend, ook al had je wellicht toevallig code gemaakt waar **geen** overloading inzat. Het verwijzen naar constructors en klassenamen laat zien dat je geen antwoord gaf op de vraag naar constructor-overloading.

(n.b. als je bv. `return "Klasse-->methode";` had, dan is dat uiteraard geen verwijzing naar een class of constructor, maar gewoon een verwijzing naar die methode (er werd niet gevraagd om het noemen van de klasse waar de overloade methodes in zaten, alleen de methode-namen, dus die klasse-namen zijn genegeerd omdat het duidelijk is welke methode bedoeld wordt).

5 punten: methode overloading (vermomming en kwetsbaar)

c) [5 punten]
Voeg de **constructors** uit klassendiagram toe aan je code:

- de constructors doen niks en slaan niks op. De inputs betekenen niks
- je mag zelf weten welke `super(...)` je aanroept, als er keuze is
- als je een *getal* nodig hebt als *input* voor een `super(..)`, gebruik dan 42
- als je een **`String`** nodig hebt als *input,* gebruik dan `"Captain Napalm"`

Let op: Als de code *vanwege de constructors* niet compileert, kun je voor **(c)** geen punten behalen.

nakijkmodel:
Alleen punten als het **constructor-gedeelte compileert** (dus: als er m.b.t. de constructoren niks in de code staat waardoor de code niet compileert. Als de code om andere redenen niet compileert, zoals bv. een *missend return type* bij een methode, dan kun je wel punten krijgen bij (c)).

5 punten voor alle constructors hebben. (2 aftrek per missende/foute/overbodige constructor.)

5 punten: klassendiagram: constructors bij overerving, correcte super(..)

Einde Tentamen

Maak een archief (**.**zip of .rar) van je **java**-bestanden. Geef het bestand de naam: `"java-theorieVoornaam_Achternaam_studentnummer.zip"` (of `rar`)

Upload je `zip/rar`-bestand op ELO in het inleverpunt in de folder genaamd: **`"inleverpunt theorie 14-06-2022"`**