hogeschool Windesheim

Techniek - HBO-ICT

Naam: Studentnummer: Klas:

Tentamen JAVA Theorie (119617)

Vakcode	: ICT.P.JAVA1.V21 (ICT.P.JAVA1.V20 , ICT.P.JAVA.V17, 18, 19) (t1)
Datum	: 8-4-2022
Tijd	: 08:30 - 10:30 uur

Klas:	Lokaal:	Aantal:
ICTM2a t/m i, ICTM2m t/m p, ICTM2tt	volgt	387

Opgesteld door	: Moerman, Wilco; Balfaqih, Aminah
Docenten	: WPH01; KNJ24; RWM02; SSM36; LNR08; KEK01; MNC07; VEE02; CNW01; HZJ40; BHA40;
Gecontroleerd door	: Kevin de Korte; Balfaqih, Aminah
Rekenmachine	: alle rekenmachines toegestaan
Literatuur	: alles (boeken, internet, aantekeningen)
Overige hulpmiddelen	: laptop
Opgaven inleveren	: ja

CONTROLEER VOORAF DE VOLGENDE GEGEVENS:

Dit tentamen bevat:

6 opgaves

30 genummerde pagina's

Waarschuw de surveillant als één van deze aantallen niet klopt!

Studentnummer	Naam	Klas	Cijfer
Tijd van inleveren:			

De toets, de punten, etc.

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 kun je vinden op ELO in de folder **`"inleverpunt theorie 8 april"`** in het bestand **`"startsituatie_theorie_8-april-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 losse `main(...)`-methodes die uitgecomment zijn. De `Main`-klasse staat niet in het klassendiagram en wordt ook niet nagekeken.

Wat een regel code *print*, staat 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.

Als je wilt, mag je op de papieren toets aantekeningen maken. Ook mag je het klassendiagram losmaken van de rest zodat je het naast een vraag kunt leggen, etc...

puntenverdeling:

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: verhuizen [30 punten]

In deze vraag modelleer je een verhuiswagen met behulp van een `array`. In een verhuiswagen kunnen dozen geplaatst worden. Hieronder staat de beginsituatie.

```
public class Verhuiswagen {

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

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

	public boolean zetNeer(Doos doos, int plek) {
		// ...todo...
		return false;
	}

	public int zetOpLaatsteVrijePlek(Doos doos) {
		// ...todo...
		return 0;
	}
}
```

```
public class Doos {

	private String inhoud;

	public Doos(String i) { inhoud = i; }

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

a) [5 punten]
Voeg een **Doos**-array (genaamd **dozen**) toe als **attribuut** aan klasse `Verhuiswagen`.

Maak de `Verhuiswagen`-**constructor** af:

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

puntenverdeling:

- 2 punten voor het hebben van atrribuut (`Doos[ ] dozen;`). (hier is niet gelet op public/private)

- 3 punten voor het initialiseren van de array in de constructor met = new `Doos [...gewenste grootte ...];`

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) [15 punten]
Maak de methode **`print()`**, die de array overzichtelijk uitprint. De werking van deze methode zie je in het voorbeeld verderop.

Maak ook de methode **`zetNeer(...)`**. Deze methode:

- voegt een `Doos`-object toe op `plek`, als daar nog **geen** `Doos` staat.
- als op die `plek` **wel** een `Doos` staat, dan wordt de nieuwe `Doos` **niet** toegevoegd. Dan wordt een foutmelding (**`"fout: plek is al bezet"`**) geprint.
- de methode returnt **`alleen`** `true` als het toevoegen van de `Doos` geslaagd is.

Er kan bij `zetNeer(...)` een ***crash*** (`ArrayIndexOutOfBoundsException`) ontstaan.

Zorg ervoor dat de methode **niet** *crasht*:

- in plaats van *crashen*, moet **`"fout: buiten bereik"`** worden geprint.
- je mag **geen** gebruik maken van **try** en **catch**.

print() en zetNeer(...). Output is herkenbaar aan //>.

```
Verhuiswagen v = new Verhuiswagen(4);
v.print();                                          //> Verhuiswagen:
                                                    //> plek 0: null
                                                    //> plek 1: null
                                                    //> plek 2: null
                                                    //> plek 3: null
v.zetNeer(new Doos("Doos 1"), 2);

// vb. van de return van zetNeer(...)
boolean b = v.zetNeer(new Doos("Doos 22"), 2);  //>  fout: plek is al bezet
System.out.println(b);                          //>  false

v.zetNeer(new Doos("Doos 333"), -1);            //>  fout: buiten bereik

v.print();                                          //> Verhuiswagen:
                                                    //> plek 0: null
                                                    //> plek 1: null
                                                    //> plek 2: Doos 1
                                                    //> plek 3: null
```

puntenverdeling:

`print:` 5 punten voor de correcte werking. Hierbij is niet naar komma's en haakjes gekeken (en ook geprobeerd om typo's niet fout te rekenen, maar dat kan per ongeluk toch gebeurd zijn)

- 2 punten aftrek als niet de juiste indices werden geprint.
- 3 punten aftrek als niet werd geprint wat in de array stond.

`zetNeer(...)`: 5 punten:

- 3 punten aftrek als toevoegen fout gaat op een **vrije** plek,
- 3 aftrek als het fout gaat bij een plek die al **bezet** is.
- 1 aftrek als de **return waarde** bij succesvol toevoegen, fout was.
- 1 aftrek als de **return waarde** bij een poging tot toevoegen op een bezette plek, fout was.

`zetNeer(...)` - foutafhandeling: 5 punten. Voor het voorkomen van **fouten/crashes** bij ongeldige inputs (zonder try-catch te gebruiken. Met try-catch: geen punten):

- 3 aftrek als het mis gaat bij een **`plek < 0`**
- 1 punt aftrek als de returnwaarde bij plek < 0 verkeerd is.
- 3 aftrek als het mis gaat bij **plek = arraylengte** (dus de eerste plek buiten de array).
- 1 punt aftrek als de returnwaarde bij plek = arraylengte verkeerd is.
- 2 punten aftrek als ondanks dat er een verkeerde plek was opgegeven (buiten de array) de array toch veranderd wordt.

c) [10 punten]
Maak de methode **`zetOpLaatsteVrijePlek(...)`** die de meegegeven `Doos` op de **`laatste`** nog vrije plek zet.

werking van zetOpLaatsteVrijePlek(...)

```
Verhuiswagen v = new Verhuiswagen(5);
// zomaar wat vulling:
v.zetNeer(new Doos("doos #1"), 2);
v.zetNeer(new Doos("doos #22"), 3);
v.print();                          //> Verhuiswagen:
                                    //> plek 0: null
                                    //> plek 1: null
                                    //> plek 2: doos #1
                                    //> plek 3: doos #22
                                    //> plek 4: null

// achterste plek zoeken:
int vrijePlek = v.zetOpLaatsteVrijePlek(new Doos("doos #333"));
System.out.println(vrijePlek);  //>  4

vrijePlek = v.zetOpLaatsteVrijePlek(new Doos("doos #4444"));
System.out.println(vrijePlek);  //>  1

v.print();                          //> Verhuiswagen:
                                    //> plek 0: null
                                    //> plek 1: doos #4444
                                    //> plek 2: doos #1
                                    //> plek 3: doos #22
                                    //> plek 4: doos #333
```

puntenverdeling:

10 punten in totaal:

- 3 aftrek als het algoritme het fout doet als de **laatste plek** [arraylengte - 1] in de array vrij is.
- 3 aftrek als het mis gaat bij een plek niet achteraan in de array, maar **ergens in het midden**
- 3 aftrek als het mis gaat bij de **voorste plek** [0]
- 3 aftrek als de hele **array vol** is en het dan verkeerd gaat (bv. toch array veranderen)

- 2 punten aftrek als de methode niet de juiste **returnwaarde** teruggeeft. Uit de voorbeelden in de toets blijkt, dat de methode de plek returnt waarop de doos geplaatst is.

Opgave 2: debuggen en loggen [10 punten]

Hieronder zie je klasse `ZomaarEenKlasse`. Elke keer dat de constructor of de `berekening(...)`-methode wordt aangeroepen, wordt er iets geprint.

In de huidige code zet je *per object* aan of uit of er geprint wordt (met `debug`).

inhoud van klasse ZomaarEenKlasse

```
public class ZomaarEenKlasse {

	public boolean debug;
	private int waarde;

	public ZomaarEenKlasse(int w) {
		waarde = w;
		if(debug) {
			System.out.println("ZomaarEenKlasse(" + waarde + ")");
		}
	}

	public void berekening(int x) {
		waarde = waarde + x;
		if(debug) {
			System.out.println("waarde --> " + waarde);
		}
	}
}
```

a) [10 punten]
Pas `ZomaarEenKlasse` als volgt aan:

- attribuut **debug** bepaalt **voor alle objecten tegelijk** of er geprint wordt.
- in attribuut `teller` wordt **geteld** hoe vaak er iets in `if(debug) { ... }` wordt geprint (het maakt niet uit bij welk `ZomaarEenKlasse`-object dat gebeurt).
- maak de methode **`printDebugging()`** die de status van het debuggen toont.

voorbeeld van goede werking van debug, tellen en printDebugging()

```
// debuggen staat aan. Vanaf nu wordt er geprint en geteld:
ZomaarEenKlasse.printDebugging();              //>  debug: true
                                               //>  0 keer geprint

ZomaarEenKlasse s = new ZomaarEenKlasse(40);   //>   ZomaarEenKlasse(40)
s.berekening(2);                               //>   waarde --> 42
ZomaarEenKlasse s2 = new ZomaarEenKlasse(2);   //>   ZomaarEenKlasse(2)

ZomaarEenKlasse.printDebugging();              //>  debug: true
                                               //>  3 keer geprint

// debuggen uitzetten: er wordt vanaf hier niks geprint of geteld.
ZomaarEenKlasse.debug = false;

// een heleboel constructors en methodes (allemaal niet geprint of geteld)
s.berekening(1);
s.berekening(1111);
s2 = new ZomaarEenKlasse(777);
s2.berekening(22);
ZomaarEenKlasse s3 = new ZomaarEenKlasse(987);
// etc... etc ...

// teller is onveranderd:
ZomaarEenKlasse.printDebugging();              //>  debug: false
                                               //>  3 keer geprint

// debuggen weer aanzetten. Er wordt weer geprint/geteld:
ZomaarEenKlasse.debug = true;

ZomaarEenKlasse s4 = new ZomaarEenKlasse(1);   //>   ZomaarEenKlasse(1)
s.berekening(-1);                              //>   waarde --> 1153

ZomaarEenKlasse.printDebugging();              //>  debug: true
                                               //>  5 keer geprint
```

puntenverdeling:

In totaal 10 punten, verdeeld over 3 voor 'debug', 4 voor het tellen, en 3 voor het printen.

`debug`, 3 punten voor het static maken van `debug`:

- 1 aftrek als debug **niet op true** staat aan het begin.

`teller`, 4 punten als het tellen helemaal goed werkt:

- alle punten weg, als je **`waarde` static** had gemaakt (daardoor verandert namelijk de werking van de berekeningen etc. omdat nu alle objecten van `ZomaarEenKlasse` dezelfde waarde hebben.
- ook geen punten, als er geen **`static int`** was om in te tellen (er is niet op de naamgeving gelet, alhoewel in de toets 'teller' stond voorgeschreven als naam).
- 2 punten aftrek als het tellen niet helemaal goed ging, bv. omdat je alleen in de mehode of alleen in de constructor telde. (in dat geval wordt er wel iets geteld, maar niet alle 'debug' situaties)

`printDebugging()`, 3 punten voor printen zoals in toets aangegeven in voorbeelden:

- 1 punte aftrek als wel de waardes goed worden geprint, **maar er woorden missen**.

Opgave 3: Temperaturen [10 punten]

Klasse `Meting` wordt gebruikt om temperatuur-metingen op te slaan. Twee ***eenheden*** worden ondersteund: graden ***Celsius*** en graden ***Kelvin***.

Met de methode `waardeInKelvin()` kun je van *Celsius* naar *Kelvin* omrekenen.

```
public class Meting {

	private int waarde;
	private boolean isCelsius;
	private String locatie;

	public Meting(int w, boolean c, String l) {
		waarde = w;
		isCelsius = c;
		locatie = l;
	}

	public int waardeInKelvin() {
		if(isCelsius) { return waarde + 273; }
		else { return waarde; }
	}
}
```

a) [10 punten]
Maak voor klasse `Meting` een geschikte **`equals`**-methode:

- deze **overschrijft** (*override*) de `equals`-methode die je van `Object` overerft
- returnt alleen `true` als de `locatie` **en** de temperatuur hetzelfde zijn.

Twee temperaturen zijn hetzelfde, als je ze naar dezelfde eenheid **omrekent** en ze dan gelijke waardes hebben: 10 graden Celsius is gelijk aan 10 + 273 = 283 Kelvin.

Correcte werking van equals(...)

```
// 10 graden Celsius in Zwolle (= 10 + 273 = 283 Kelvin)
// Hiermee worden alle andere Meting-objecten vergeleken
Meting c10 = new Meting(10, true, "Zwolle");
System.out.println( c10.waardeInKelvin() );              //>  283

// locatie en temperatuur gelijk:
Meting c10_nogEenKeer = new Meting(10, true, "Zwolle");
System.out.println( c10.equals(c10_nogEenKeer) );        //>  true

// locatie en temperatuur gelijk (want 10 Celsius is 283 Kelvin)
Meting k283 = new Meting(283, false, "Zwolle");
System.out.println( c10.equals(k283) );                  //>  true

// locatie ongelijk:
Meting c10_elders = new Meting(10, true, "Rotterdam");
System.out.println( c10.equals(c10_elders) );            //>  false

// temperatuur ongelijk, want 10 graden Celsius is 283 Kelvin
Meting k10 = new Meting(10, false, "Zwolle");
System.out.println( c10.equals(k10) );                   //>  false

// ongelijke temperatuur
Meting c273 = new Meting(273, true, "Zwolle");
System.out.println( c10.equals(c273) );                  //>  false

// geen Meting als input
System.out.println( c10.equals("10 graden Celsius") );   //>  false
```

puntenverdeling:

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(Meting)`. 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 `Meting` met exact **dezelfde waardes**.
- 3 aftrek als het mis gaat bij een meting met dezelfde waarde + meetlocatie maar met **de ene in kelvin en de ander in celsius**
- 3 aftrek als het foutgaat bij metingen waarbij de **locatie verschilt**
- 2 eraf als het fout gaat bij metingen die **dezelfde temperatuur** weergeven, maar de ene in celsius, de ander in kelvin (bv. 0 celsius en 273 kelvin op dezelfde locatie)

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.

Opgave 4: Bananenrepubliek [15 punten]

In opgave **4** en **5** gaat het over een softwarepakket voor gevangenissen, dat gemaakt wordt in het land Bananië, ten zuiden van Verweggistan.

In een `Gevangene`-object worden de gegevens van een veroordeelde opgeslagen.

```
public class Gevangene {

	public String naam = "?";
	public int straf = 0;

	public Gevangene(String n) { naam = n; }

	public void geefStraf(int wachtwoord, int straf) {
		if (this.straf < 0) { // straf moet altijd >= 0 zijn.
			straf = 0;
		} else {
			this.straf = straf;
		}
	}

	public void verminder(int wachtwoord) {
		straf--;
	}

	// deze methode mag niet veranderd worden
	public boolean isVrij() {
		return straf <= 0;
	}
}
```

a) [10 punten]
Een `wachtwoord` (1234567) moet **`straf` beschermen**. Pas de klasse als volgt aan:

- alleen als het juiste **`wachtwoord`** wordt gebruikt, kan `straf` veranderd worden.
- er mag **geen** andere manier zijn waarop `straf` veranderd kan worden.
- **`geefStraf(...)`** kan per `Gevangene` maar **1x** gebruikt worden om `straf` een waarde te geven.
- als `geefStraf(...)` vaker gebruikt wordt, dan mag `straf` **niet** veranderen.
- **`verminder(...)`** mag eindeloos vaak gebruikt worden.
- als het `wachtwoord` verkeerd is, of als `geefStraf(...)` al een keer succesvol gebruikt is, dan wordt **`"mag niet!"`** geprint.

Zorg er ook voor dat `Gevangene`-objecten worden **`geprint`** zoals in dit voorbeeld.

gewenste werking van Gevangene

```
int wachtwoord = 1234567;
Gevangene g = new Gevangene("Murdoc");
// g.straf = 1; // uiteraard moet dit ook onmogelijk zijn
System.out.println(g);          //>  Murdoc (straf: 0)

// kan niet, verkeerd wachtwoord
g.geefStraf(111111, 99);        //>  mag niet!
System.out.println(g);          //>  Murdoc (straf: 0)

// geldig wachtwoord:
g.geefStraf(wachtwoord, 10);
System.out.println(g);          //>  Murdoc (straf: 10)

// mag niet, geefStraf(...) is al een keer succesvol uitgevoerd
g.geefStraf(wachtwoord, 5);     //>  mag niet!
System.out.println(g);          //>  Murdoc (straf: 10)

// Aanroepen van verminder(...) kan ook alleen met het juiste wachtwoord.
g.verminder(90000);             //>  mag niet!
g.verminder(wachtwoord);
g.verminder(wachtwoord);
System.out.println(g);          //>  Murdoc (straf: 8)
```

puntenverdeling:

2 punten voor het **private** maken van attribuut `straf` (de rest v/d attributen is genegeerd. Die hoefden in de opgave niet beschermd te worden).

4 punten voor correcte werking van **`geefStraf(...)`**:

- 2 aftrek, als je na een succesvolle aanpassing, daarna **nogmaals** kunt aanpassen met het goede wachtwoord.
- 1 punt aftrek voor een **verkeerde foutmelding**

2 punten voor `verminder(...)`:

- 1 aftrek als het aanpassen als wachtwoord goed is, niet werkt.
- 2 aftrek als het **aanpassen toch gebeurt** als het wachtwoord fout is.

- 1 punt aftrek voor een **verkeerde foutmelding**

2 punten voor de **`toString()`** (dus met naam en straf in de output).

b) [5 punten]
De methodes `geefStraf(...)` en `verminder(...)` bevatten *bugs* waardoor de `straf` kleiner dan 0 kan worden.

voorbeeld van verkeerde werking/bug (straf < 0):

```
int wachtwoord = 1234567;
Gevangene g = new Gevangene("Murdoc");
g.geefStraf(wachtwoord, -7);
System.out.println(g);    //>  Murdoc (straf: -7)

Gevangene g2 = new Gevangene("Harry Houdini");
g2.geefStraf(wachtwoord, 1);
g2.verminder(wachtwoord);
g2.verminder(wachtwoord);
System.out.println(g2);   //>  Harry Houdini (straf: -1)
```

Repareer **`geefStraf(...)`** en **`verminder(...)`**, zodat de `straf` niet meer kleiner dan 0 kan worden.

puntenverdeling:

In beide methoden zit een fout. In het ene geval (`geefStraf`) ging het om een probleem met de **scope**: er werd in de if gekeken naar `if(this.straf < 0)`, maar dat moet uiteraard `if(straf < 0)` zijn. Want this.straf verwijst naar het attribuut, en dat is op dat moment altijd 0 (want dat is de waarde de het krijgt bij declaratieen initialisatie).

In de andere methode ging het erom dat je moet checken of `straf--` niet kleiner dan 0 wordt, wat niks me scope te maken had.

- 2 punten aftrek als de **`verminder(...)`**-methode nog steeds zorgt dat straf < 0 kan worden,
- 4 punten aftrek als **`geefStraf(...)`** niet gerepareerd is/niet werkt.

- 1 punt aftrek als de straf niet meer **0** kan worden (mocht namelijk niet < 0 worden, dus 0 mag wel).

Opgave 5: In de Cel [15 punten]

Hieronder volgen de code van de `Cel`-klasse, waarin 1 `Gevangene` (uit opgave **4**) opgesloten kan worden.

```
public class Cel {

	private Gevangene gevangene;

	public String toString() {
		return "Cel: # " + gevangene + " #";
	}

	public void setGevangene(String naam, int straf) {
		gevangene = new Gevangene(naam);
		gevangene.geefStraf(1234567, straf);
	}

	public Gevangene getGevangene() {
		return gevangene;
	}

	public void jaarwisseling() {
		gevangene.verminder(1234567);
	}

	public void laatVrij() {
		if (gevangene.isVrij()) {
			gevangene = null;
		}
	}

	public void omwisselen(Cel andere) {
		andere.gevangene = gevangene;
		gevangene = andere.gevangene;
	}
}
```

Let op: opgaves **4(b)**, **5(a)**, **5(b)** en **5(c)** kunnen los van elkaar gemaakt worden.

a) [5 punten]
Een `Cel` is leeg als het `gevangene`-attribuut **`null`** is. Als dat zo is, treedt er een *crash* op in de methodes `laatVrij()` en `jaarwisseling()`.

Repareer **`laatVrij()`** en **`jaarwisseling()`**:

- je **moet** gebruik maken van **catch**,
- een **zo specifiek mogelijke** `exception` opvangen,
- en dan een foutmelding (**`"Cel is leeg!"`**) printen.

voorbeeld van goede werking: geen crash maar een foutmelding als Cel leeg is.

```
Cel alcatraz = new Cel();
alcatraz.setGevangene("Murdoc", 1);
System.out.println(alcatraz);  //>  Cel: # Murdoc (straf: 1) #

alcatraz.jaarwisseling();
System.out.println(alcatraz);  //>  Cel: # Murdoc (straf: 0) #

alcatraz.laatVrij();
System.out.println(alcatraz);  //>  Cel: # null #

// hier kan een crash ontstaan als de methodes er niet tegen beschermd zijn
alcatraz.jaarwisseling();      //>  Cel is leeg!
alcatraz.laatVrij();           //>  Cel is leeg!
```

puntenverdeling:

per methode geldt:

- Geen punten, als geen gebruik werd gemaakt van **`try...catch`**.
- 1 punt aftrek als het **niet NullPointerException** was, maar een algemenere Exception

b) [5 punten]
Soms moet een `Gevangene` van de ene `Cel` omgewisseld worden met die van een andere `Cel`.

Met de huidige implementatie de methode **`omwisselen(...)`**, raakt men een `Gevangene` kwijt! Dit kun je zien in onderstaand voorbeeld.

voorbeeld van verkeerde werking/bug: omwisselen gaat mis.

```
Cel alcatraz = new Cel();
alcatraz.setGevangene("Murdoc", 1);
Cel bajes = new Cel();
bajes.setGevangene("Harry Houdini", 88);

System.out.println(alcatraz);   //>  Cel: # Murdoc (straf: 1) #
System.out.println(bajes);      //>  Cel: # Harry Houdini (straf: 88) #

alcatraz.omwisselen(bajes);

// de ene Gevangene is nu verdwenen,
// en de andere zit nu in allebei!
System.out.println(alcatraz);   //>  Cel: # Murdoc (straf: 1) #
System.out.println(bajes);      //>  Cel: # Murdoc (straf: 1) #
```

Repareer de **`omwisselen(...)`**-methode, zodat na de wissel de ene `Cel` de andere gevangene heeft, en andersom.

puntenverdeling:

Bij **'swappen'/verwisselen** heb je altijd een **tijdelijke** variabele voor nodig, omdat als je de ene waarde toekent aan de andere variabele, je die andere waarde kwijt bent (tenzij je een tijdelijke variabele hebt om nog naar dat object te verwijzen).

Deze vraag is 'binair'. Werking is fout (geen punten) of goed (alle punten).

c) [5 punten]

De `straf` van het `gevangene`-attribuut mag **niet** veranderen, behalve via de `jaarwisseling()`-methode (van `Cel`). In Bananië doen ze niet aan strafvermindering of gratie!

Helaas zit er nog een *bug* in het systeem: je kunt het `gevangene`-attribuut opvragen via de getter en dan met de `verminder(...)`-methode de `straf` van het attribuut aanpassen en daardoor de `gevangene` vrijlaten.

voorbeeld van verkeerde werking/bug: een bevrijdingsactie!

```
Cel alcatraz = new Cel();
alcatraz.setGevangene("Harry Houdini", 100);
System.out.println(alcatraz);  //>  Cel: # Harry Houdini (straf: 100) #

// iemand vraagt gegevens op ...
Gevangene g = alcatraz.getGevangene();

// ... en gebruikt juiste wachtwoord:
for(int i = 0; i < 1000; i++) { g.verminder(1234567); }
System.out.println(alcatraz);  //>  Cel: # Harry Houdini (straf: 0) #

alcatraz.laatVrij();

// BUG: hij is vrij, maar dat is natuurlijk NIET de bedoeling!
System.out.println(alcatraz);  //>  Cel: # null #
```

Repareer `getGevangene()` zodanig:

- dat je met de getter een `Gevangene`-object met de ***zelfde*** informatie krijgt
- maar dat het **niet** mogelijk is om via `getGevangene()` het attribuut `gevangene` van `Cel` te veranderen

(Je mag voor vraag **(c)** getters en setters voor `straf` en `naam` aan `Gevangene` toevoegen. Je hoeft geen rekening met een `wachtwoord` te houden).

Hieronder volgt een voorbeeld van de gewenste werking. Je kunt de gegevens van de `Gevangene` opvragen met `getGevangene()`. Maar het is niet mogelijk om het attribuut van `Cel` te veranderen.

voorbeeld van de goede werking van getGevangene()

```
Cel alcatraz = new Cel();
alcatraz.setGevangene("Harry Houdini", 100);
System.out.println(alcatraz);  //>  Cel: # Harry Houdini (straf: 100) #

// iemand vraagt gegevens op
Gevangene g = alcatraz.getGevangene();
System.out.println(g);         //>  Harry Houdini (straf: 100)

// en heeft juiste wachtwoord
for(int i = 0; i < 1000; i++) { g.verminder(1234567); }
// dus g is veranderd
System.out.println(g);         //>  Harry Houdini (straf: 0)

// maar attribuut gevangene is niet veranderd
System.out.println(alcatraz);  //>  Cel: # Harry Houdini (straf: 100) #

// dus vrijlaten gebeurt niet
alcatraz.laatVrij();
System.out.println(alcatraz);  //>  Cel: # Harry Houdini (straf: 100) #
```

puntenverdeling:

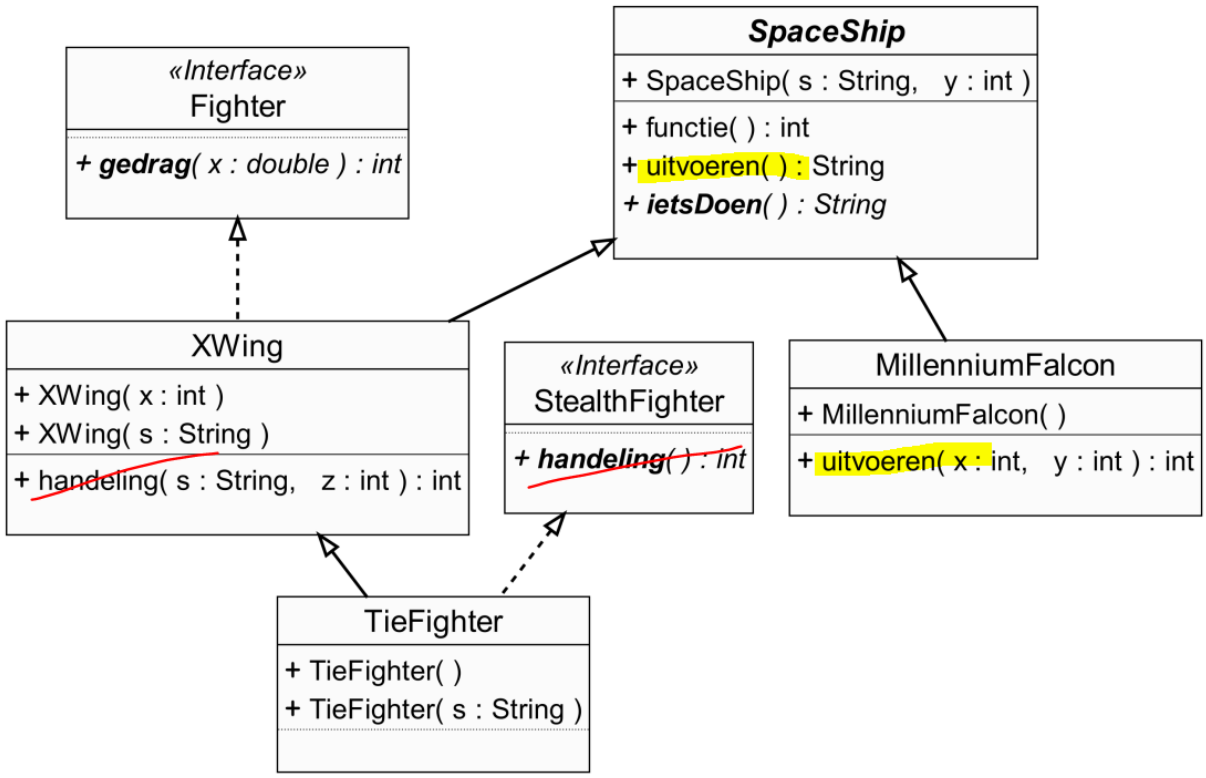
Het probleem is, dat als je de getGevangene() gebruikt, je een **verwijzing** (reference, adres) krijgt naar een stuk geheugen waar de gegevens van de gevangene staan. Als je de opgevraagde gegevens verandert, dan heb je **dus** de gegevens die de Cel heeft, veranderd, want dat attribuut is een verwijzing naar hetzelfde stukje geheugen. Dit is een veel voorkomende bron van bugs.

De meest voor de hand liggende manier om het probleem op te lossen, is om een **`nieuw`** object ter returnen dat wel dezelfde inhoud heeft. Dus **`new Gevangene(...)`**. Hiervoor kon je getters toevoegen aan de klasse Gevangene zodat je de waardes van de attrbuten kunt opvragen en in het nieuw gemaakte Gevangene-object kunt stoppen (dat dus inhoudelijk een kopie is, maar **niet** naar hetzelfde stuk geheugen wijst, want met **`new`** krijg je in Java (en C#, etc) altijd een **nieuw** stuk geheugen.)

Deze vraag is goed (alle punten) of fout (geen punten).

Opgave 6: A long time ago in a galaxy far, far away.... [20 punten]

Na het kijken van teveel *Star Wars* films, heeft iemand 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 **`int`** returnt, return dan het nummer van je favoriete *Star Wars* film (of `42` als je niet kunt kiezen).
- als een methode een **`String`** returnt, dan **moet** je de naam van de klasse en de methode gebruiken, dus bv.: `return "TieFighter --> uitvoeren";`

puntenverdeling:

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.

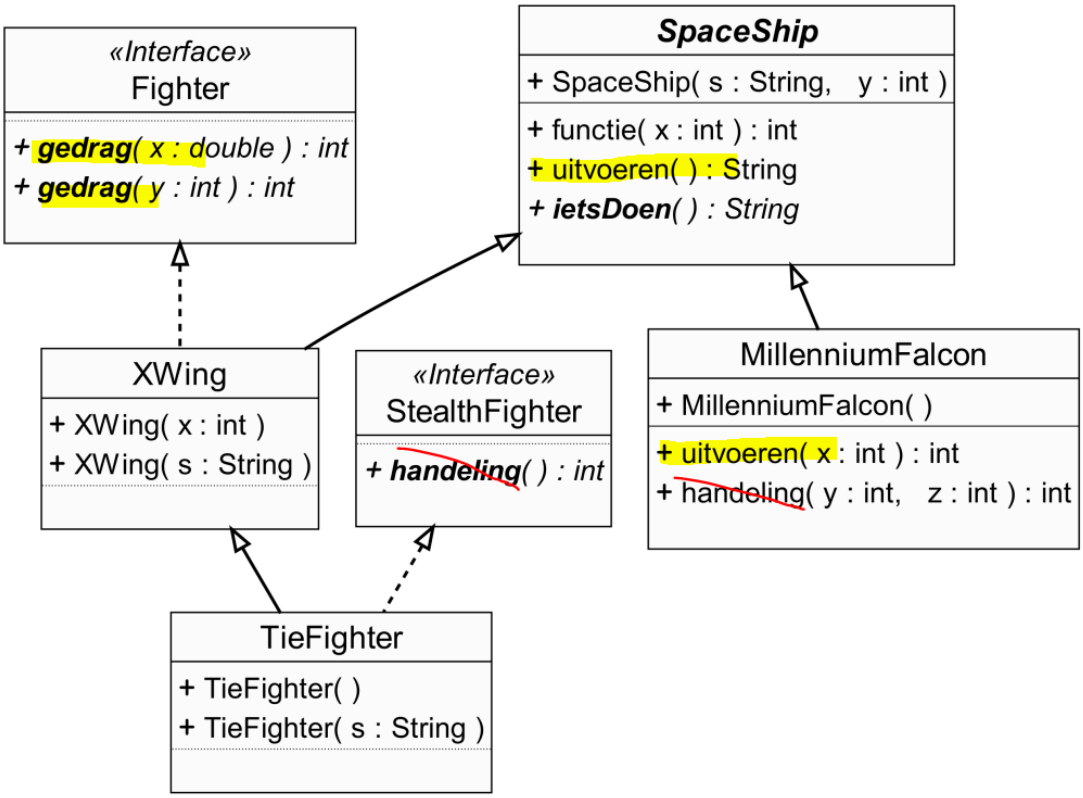
Maar een methode die een **`String`** returnt, **moet** volgens de vraag de naam van de **klasse** + methode returnen. Dat kan uiteraard niet door de 'super' methode afgehandeld worden (behalve als je met **`instanceof`** zou gaan werken). De makkelijkste oplossing is dus, om de methoden die een String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

Door een klein technisch probleem is het commentaar over wat er miste in de implementatie v/h klassendiagram, niet bij iedereen in de nagekeken code terecht gekomen (en het kost teveel tijd om dat commentaar opnieuw te genereren voor ruim 300 toetsen, voordat de inzage is). Uiteraard kun je door vergelijken van je code met de voorbeelduitwerking, zien welke onderdelen er missen (en op verzoek kan ik het ook opnieuw voor je testen/bepalen).

Let op: Door een probleem was een andere versie van de startcode op ELO gekomen dan die hoorde bij de uitgeprinte toets. Daardoor was het klassendiagram dat in de zip stond, anders. Bij het nakijken is hier rekening mee gehouden, en gekeken welke van de twee diagrammen geimplementeerd is (door te kijken naar de verschillen in diverse klassen).

'digitale' versie (in startsituatie zip):



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

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

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

```
public String alleOverloading() {
      return "ietsDoen + uitvoeren + functie";
}
```

puntenverdeling:

Bij overloading van **methodes** gaat het om methodes met dezelfde naam in dezelfde klasse.

Afhankelijk van of je de papieren versie van het diagram, of de digitale hebt geimplementeerd, zijn er twee verschillende antwoorden. In het klassendiagram staat/staan de methode(s) die overloading veroorzaken, gehighlight. De methodes die wel dezelfde naam hebben, maar **niet** voor overloading zorgen, zijn rood doorgestreept.

op basis van het **diagram**:

- papier: **`"uitvoeren"`** (want: methode "handeling" in XWing returnt een int, en moest dus niet in TieFigher geimplementeerd worden).
- digitaal: **`"uitvoeren + gedrag"`** (methodes genaamd "handeling" 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 alle 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 laate zien dat je geen antwoord gaf op de vraag naar constructor-overloading.

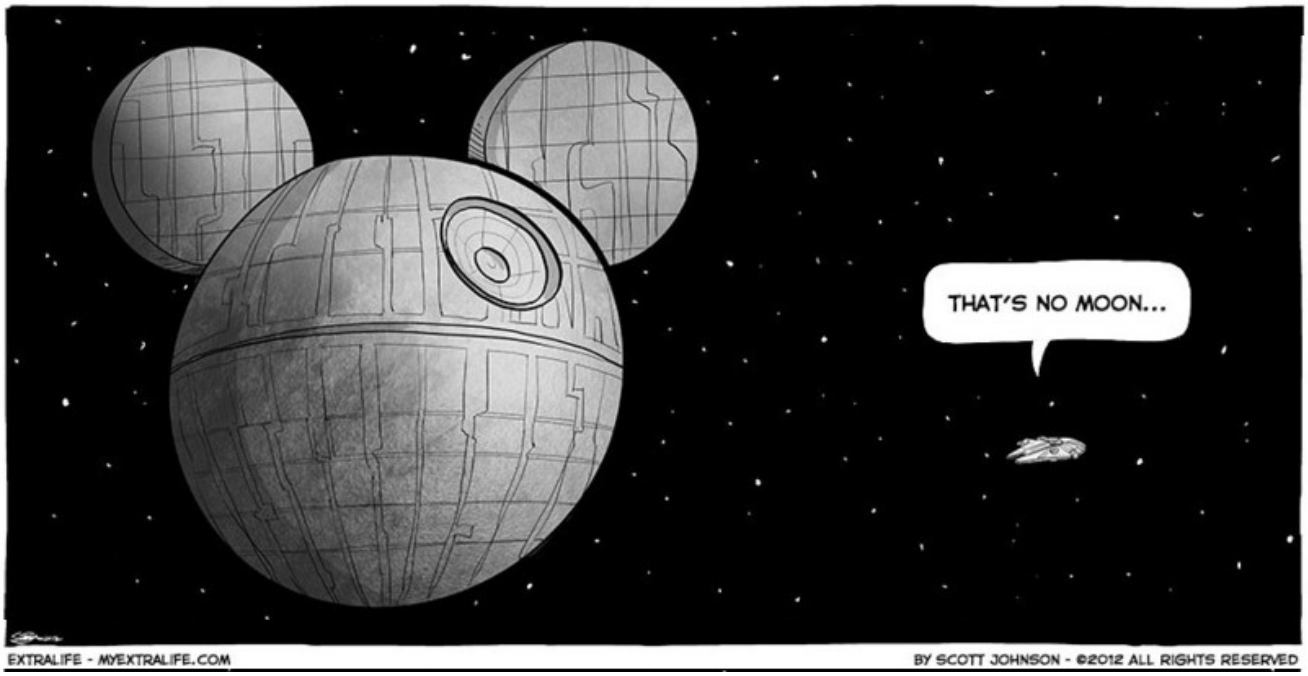
(n.b. als je bv. `return "Spaceship-->doeIets";` had, dan is dat uiteraard geen verwijzing naar een class of constructor, maar gewoon hetzelfde als het antwoord `return "doeIets";` (er werd niet gevraagd om het noemen van de klasse waar de overloade methodes in zaten, alleen de methode-namen, dus die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

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

- de constructors doen niks en slaan niks op. De inputs slaan nergens op.
- je mag zelf weten welke `super(...)` je aanroept, als er keuze is.
- als je een **`int`** nodig hebt als *input* voor een `super(..)`, gebruik dan weer het nummer van je favoriete *Star Wars* film.
- als je een **`String`** nodig hebt als *input* voor een `super(..)`, gebruik dan je favoriete *Star Wars* personage, bv.: `"Jar Jar Binks!"`
- maar **één** constructor ***per klasse*** mag het keyword **`super(...)`** gebruiken.

Let op: Als de eis van **1x** `super(...)` niet lukt, zorg er dan in ieder geval voor dat

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



puntenverdeling:

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)).

- **3 punten voor alle constructors hebben**. (1 aftrek per missende/foute/overbodige constructor.)
- **2 punten voor 1x super per klasse**. Dit was zowel in `XWing` als in `TieFighter` nodig en kon beide keren met `this(...)` opgelost worden, zodat de ene constr. met this(..) naar de andere constr. in dezelfde klasse verwijst. Die andere constr. kan dan super(..) gebruiken om naar de parent te verwijzen.

Zonder klassen waar je twee constructors had, kun je uiteraard ook geen punten krijgen voor het hebben maar 1 super(...).

Einde Tentamen

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

Upload je `zip/rar`-bestand op ELO in het inleverpunt in de folder genaamd: **`"inleverpunt theorie 8 april"`**.

Let op: Kies bij op ELO voor de **inleveren** (of **submit**)-knop!