Hoofdstuk 1

Er bestaan **Klassen** en **Objecten**.

Deze worden opgeslagen in zogenaamde **packages**.

Bovenaan de file geef je aan in welke **package** je document zit. (Max 1 maal)

Met **import** kun je hele packages beschikbaar maken voor de code OF individuele klassen OF **static** fields en methods.

Access Modifiers geven aan vanaf waar je bij het element kunt.

public iedereen
protected package en subclass (=extends)
default package (tegenwoordig ook vaak package-private genoemd)
private alleen de class.

Default wordt java.lang geïmporteerd.

Van de non-access modifiers **abstract**, **final** en **static** moet je hun impact kennen.

abstract class: Kan niet geïnstantieerd worden.
abstract method: Kan geen body hebben en MOET geïmplementeerd worden door subclasses EN de klasse moet abstract zijn.

final class: Kan niet geextend worden.
final method: Kan niet overridden worden.
final field: kan maar 1 keer een waarde hebben: een constant dus.

static field en method: worden met het programma mee geladen en kunnen benaderd worden via KlasseNaam.fieldnaam. fielden kunnen 1 waarde die voor alle objecten gelijk is.

Een .java file is het bestand waar in je als programmeur codeert.
De .java file MOET dezelfde naam hebben als de public class die erin staat. (Max 1 public class per file)
Voor iedere class in de .java file wordt een .class file gemaakt.
De .class file wordt door een computer uitgevoerd.

Als je een .class file runt met cmd, dan wordt er naar de **public static void main(String[] args)** gezocht en uitgevoerd. Let op dat de extra commando's die meegeeft als array in de args eindigen.
static public mag || String args[] mag || String... args mag

==========

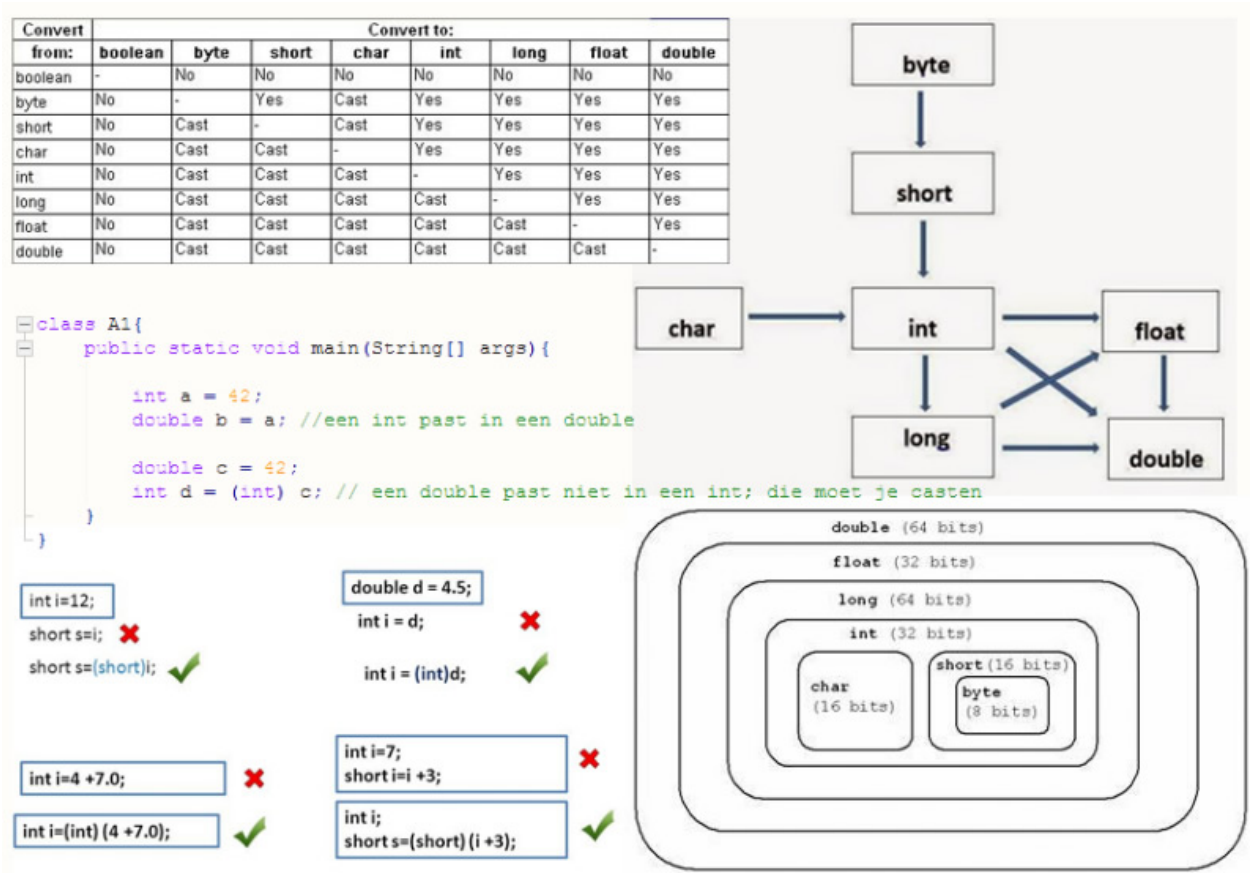
Hoofdstuk 2

Er bestaan 8 primitive types:
char, byte, short, int, long, float, double, boolean
De rest zijn Object references. De enige uitzonderlijke Klasse is String.

De **belangrijkste verschillen** tussen primitives en Object references:
- Met de operator == kun je primitives vergelijken. Met de methode .equals vergelijk je of objecten gelijk zijn. == bij objecten checkt of de leash naar hetzelfde object wijzen.
- De garbage collector werken alleen op Objecten, nooit op primitives.

- Veel operators werken niet met objecten, zoals ++ of -=, alleen met primitives
- Primitives leven op de stack. Zowel hun type, naam als waarde.
Object references hebben hun type en naam op de Stack. Het object op de heap. Objecten zitten met een leash aan hun variabele op de stack.
- Bij methode argumenten worden bij primitives een kopie meegegeven. Object references geven hun leash door in plaats van een kopie.

Het examen toets vaak je kennis op welke primitives in welke primitives past:



Convert from:	Convert to: boolean	byte	short	char	int	long	float	double
boolean	-	No	No	No	No	No	No	No
byte	No	-	Yes	Cast	Yes	Yes	Yes	Yes
short	No	Cast	-	Cast	Yes	Yes	Yes	Yes
char	No	Cast	Cast	-	Yes	Yes	Yes	Yes
int	No	Cast	Cast	Cast	-	Yes	Yes	Yes
long	No	Cast	Cast	Cast	Cast	-	Yes	Yes
float	No	Cast	Cast	Cast	Cast	Cast	-	Yes
double	No	Cast	Cast	Cast	Cast	Cast	Cast	-


```
class A1{
    public static void main(String[] args){

        int a = 42;
        double b = a; //een int past in een double

        double c = 42;
        int d = (int) c; // een double past niet in een int; die moet je casten
    }
}
```


Ook moet je weten dat uit delingen en vermenigvuldigingen default een int of double komen.
Besef je dat een char zowel een character kan zijn als een nummer.
char accepteert alleen 'b', een character met enkele qoutes.

Identifiers mogen alleen 0-9 a-z A-Z een currencyteken of _ hebben. Een identifier mag niet met een cijfer beginnen EN geen reserved word zijn.

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

Om uit een klasse een object te maken gebruik je het woord **new KlasseNaam()**
Met behulp van operators kun je acties uitvoeren.

Operator type	Operators	Purpose
Assignment	=, +=, -=, *=, /=	Assign value to a variable
Arithmetic	+, -, *, /, %, ++, --	Add, subtract, multiply, divide, and modulus primitives
Relational	<, <=, >, >=, ==, !=	Compare primitives
Logical	!, &&, ||	Apply NOT, AND, and OR logic to primitives

Fields krijgen altijd een default waarde mee, zoals null, 0 of false.

==========

Hoofstuk 3

Locale variabelen krijgen geen default waarde mee. Elke variabele heeft een **scope**, een bereik.
Een locale variabele is alleen in de methode waar hij gemaakt wordt bereikbaar.
Een locale variabele mag dezelfde naam hebben als een **field**.
Shadowing zorgt ervoor dat bij default een variabele de dichtstbijzijnde locale variabele neemt met de kleinste scope. Je kunt het field bereiken met het woord **this.variabeleNaam**.

Methoden kunnen variabelen 'meekrijgen', tussen de haakjes. Dit worden **parameters** aan de definiërende kant genoemd, **argumenten** aan de aanroepende kant. Parameters zijn locale variabelen.

Klasse variabelen zijn variabelen met de modifier **static**. Deze kun je aanroepen door **KlasseNaam.variabele** EN **objectNaam.variabele**, deze laatste is bad practice.
Object variabelen zijn fields en hebben geen modifier static. Deze kun je **ALLEEN** aanpassen met objectNaam.variabele.

Op het moment dat een object **accessible** is (**leash**), ook al is dit via een ander object, dan is het object niet **eligable voor garbage collection**. Wanneer er geen leash meer is, wordt het object aangemerkt voor garbagecollection. LET OP: je weet nooit wanneer het object DAADWERKELIJK word opgeruimd.

Een **methode** heeft altijd een **return type**. Als je niets return is het returntype **void**. Dan is return; toch toegestaan. Na return houdt de verdere executie van de methode op. Hierdoor is natuurlijk makkelijk unreachable code te organiseren. Dit resulteert in een **compile Error.**

Hetgeen dat gereturned wordt MOET een **covariant** van het returntype zijn.
BELANGRIJK: Als een methode met een returntype wordt aangeroepen, veranderd de aanroeping in het resultaat van de methode.

Overloading is een methode met dezelfde naam, met een andere **signature**. (Signature is het aantal en de typen parameters). Overloading methods mogen andere returntypes hebben, daar het returntype **geen onderdeel** is van de signature.

Een field mag dezelfde naam hebben als een methode.

Met **this.methodenaam()** roep je vanuit de ene methode, een andere methode aan in het object.

Constructors zijn speciale methodes.
- Een constructor heeft geen returntype.
- Een constructor heeft dezelfde naam als de Klasse.(Hoofdletter gevoelig).

Constructors bepalen het gedrag bij de creatie van een instantie.
Constructors kunnen overloaded zijn.

In iedere constructor staat impliciet **super()**, als eerste statement. Wat de nul-parameter constructor van zijn ouders aanroept.

Iedere klasse heeft een nul-parameter **default constructor**. Deze verdwijnt zodra de eerste handmatige constructor wordt aangeroepen.

super.field of **super.methode()** kan ook. Maar dan roep je geen constructor aan.

Met **this()** kun je een andere constructor van je eigen klasse aanroepen.

Encapsulation is het private maken van een field, en met behulp van public methods, controle houden over wat er gebeurd wanneer je een field een nieuwe waarde wilt geven.

Methodes met Object types in parameters kunnen natuurlijk ingewikkelde grappen uithalen met deze parameters.

Initialiser blocks komen static en non-static voor. static initialiser blocks worden uitgevoerd bij het laden van het programma. **LET OP** de volgorde. **Declaratie** moet voor **initialisatie** plaatsvinden.
Non static initialiser blocks worden uitgevoerd, iedere keer als er een object wordt gemaakt. NOG VOOR de constructor.
De volgorde is altijd van boven naar beneden. Methodes mogen al eerder aangeroepen worden, dan ze gedeclareerd zijn.

==========

Hoofdstuk 4

Dit hoofdstuk is voornamelijk leren.

De **pool** is een extra stukje functionaliteit die de statements als String a = "hallo"; mogelijk maken. Nu verwijst a naar de waarde hallo in de pool.

String a = new String("hallo"); maakt gewoon gebruik van de **heap**.

String is **immutable**.

String kent de methodes:
charAt(), indexOf(), substring(), trim(), replace(), length(), startsWith(), endsWith() als belangrijksten.

immutabiliteit houdt in dat String a = "hallo"; a.replace('a','b');System.out.println(a); nog steeds hallo als output heeft. Alleen wanneer je het tweede statement a = a.replace('a','b'); doet, zal de output hbllo zijn.

String kent de operators += en + om Strings te **concateneren**.

String **==** String in de pool kan true geven. Op de heap is dit false.
String.**equals**(String) geeft in beide gevallen true.

StringBuilder is **mutable**.

StringBuilder kent de methodes:
append(), insert(), delete(), deleteCharAt(), reverse(), replace(), subSequence() als belangrijksten.

LocalDateTime heeft geen constructor. Hij wordt via het factory-pattern geinstantieerd. Je kunt via statische methoden die een LocalDateTime teruggeven een LocalDateTime verkrijgen.

Array's
Arrays hebben een **fixed** lengte. De lengte wordt bepaald in de initialisatie.
Kleinere Arrays kunnen in grotere arrays worden gestopt.
Meerder dimensies kunnen worden gemaakt door het aantal [] te vergroten.
[] = rij
[][] = tabel
[][][] = kubus
int[] mijnArray[][] is een driedimensionale array.

Je vraagt de waarden op door mijnArray[1] aan te roepen.
Het meteen invoeren van waarden bij de initialisatie doe je door int mijnArray[] = new int[]{1,2,3};
of int ma[] = {2,3}; te doen.

Arrayvelden krijgen bij creatie **de default waarde**.

.length moet je als methode kennen.

ArrayList is een verzameling met variabele lengte.
ArrayList<Type> mijnArrayList = new ArrayList<Type>();
ArrayList<Type> mijnArrayList = new ArrayList< >();

met methodes:
add(), listIterator(), set(), remove(), addAll(), clear(), get(), size(), contains(), indexOf(), lastIndexOf() als belangrijksten

LET OP bij arrays en ArrayLists. De lengte wordt bepaald door hoe mensen tellen. De index wordt bepaald door hoe computers tellen. Zij starten bij nul!

LocalDateTime
De LocalDateTime heeft een factory-pattern en daardoor geen publieke constructor. Alleen middels statische methodes die een LocalDateTime returnen, kun je aan dit object komen. LocalDateTime.now() is de bekendste.

Ken het gedrag van de methodes:
of(), now(), minusWeeks(), plusYears() (etc), getMonths(), getYears() (etc), parse(), isBefore(), isAfter(), withMinute() (etc), getDayOfMonth(), getMonthValue()

Weet dat LocalTime en LocalDate ook bestaan, ken de rol van Period binnen verschillende toepassingen.

Wrappers
Elke primitive heeft een eigen wrapper.
Byte, Short, Integer, Long, Character, Boolean, Float, Double

Dat is een objectvariant van de primitive. Dit object faciliteert met vele handige fields en methoden zoals parseFloat() en MAX_VALUE.
De primitive en zijn wrapper zijn dankzij autoboxing en unboxing volledig uitwisselbaar in bv returntype, argumenten of assignments.

```
int kopen(){ return new Integer(5); }    //  unboxing
Integer a = 1;                  // autoboxing
int b = 3; Integer c = b;       // autoboxing
```

Let op de constructor, alle wrappers hebben een String en primitive constructor overload.
Boolean b = new Boolean("true"); en Boolean c = new Boolean(false);

muv Character (heeft GEEN string variant) Float (heeft ook een double overload)

==========

Hoofdstuk 5

if(evaluatie naar boolean){ }
Zonder {} is alleen **1 statement** geldig.
Tussen if en else mag **niets** staan.
else if() maakt lange ketens mogelijk.

elif of elseif zijn FOUT

KORTE IF NOTITIE voor assignments
String s = i<3? indien true: indien false
bv:
int j = 4;
String s = j<3 ? "kleiner":"groter";

Let op nested if's
else blocken zijn niet **verplicht**.

switch kan gebruikt worden bij **String, char, int, short, byte AND Character, Integer, Short, Byte en enum**.
De default case is NIET verplicht. default mag **overal** staan. default wordt gebruikt als de uitkomst niet bij een case staat.

Als een case wordt gematched, dan zal de switch vanuit daar door **blijven lopen** tot het einde. Alleen het zeer gebruikelijke **break**, stopt de verdere executie van de switch.

De case-waarden moeten **vast staan**, en mogen niet **veranderlijk** zijn. Meerdere cases met **dezelfde waarde** mogen niet. De cases moeten van dezelfde type zijn.

case null mag niet.

for(inititialisation; evaluation ; update){} blijft uitvoeren totdat de evaluation **false** wordt.

initialisation mag meerdere variabelen creëren als ze van hetzelfde typen zijn.
bij initialisation mag je een variabele van buiten het for-statement gebruiken.

De enhanced for loop loopt over een **array, arrayList** heen.

for(Type identifier : Lijst)

Deze loopt altijd over heel de array heen. Je kunt het element direct met de **identifier** aanspreken.

De **while loop** evalueert **eerst** en voert dan eventueel uit.

De **do while loop** loopt sowieso 1 keer en voert **daarna** de evaluatie uit.

break zorgt ervoor dat de loop **eindigt**.

continue zorgt ervoor dat de volgende evaluatie wordt uitgevoerd, met een eventuele uitvoer van de volgende iteratie.

met **labels** kun je een loop een naam geven. **naamloop: for**
Hier kun je uitbreaken door **break naamloop**; of **continue naamloop**; Deze commando's mogen alleen in de loop worden uitgevoerd.

++b heeft meteen de waarde na de ophoging.
b++ heeft nu de oude waarde en verhoogd voor het volgende gebruik.

==========

Hoofdstuk 6

extends zorgt voor een erfrelatie.

overriding is het geven van ander gedrag aan dezelfde methode. Bij overriding is EN de signature van belang EN het returntype.

Een kind kan alles wat zijn ouders kan TENZIJ hij hun gedrag **override**.

Een ouder weet niet wie zijn kinderen zijn.

class Links{}
class Rechts extends Links{}
Links mijnDier = new Rechts();

Links mag **groter** zijn dan rechts.
Links bepaalt bij welke methoden en velden je **kunt**. Rechts bepaalt wat het object uiteindelijk **kan en doet**.

Polymorphisme zorgt ervoor dat de fields van Links worden gebruikt, en de methoden van Rechts.

Als een **methoden** een **field gebruikt,** gebruikt hij het field van Rechts. (Of tussen-level)

Een klasse **kan 1 superklasse** extenden.

Een **abstracte** klasse kan niet **geïnstantieerd** worden.

Als er een **abstracte** methode is, MOET de klasse **abstract** zijn.

Erving zorgt voor een Is-A relatie. Met **instanceof kan de Is-A relatie gechecked** worden.
(Bij klasses moet een Is-A relatie wel tot de opties behoren, anders compilefout)
(Bij interfaces hoeft een Is-A relatie niet tot de opties te behoren, slechts een runtime-error)
Casting werkt volgens dezelfde principes.

Casting wordt ingezet om Links bij de functies van de kleinere Rechts of tussen-level te laten kunnen.
Casting is gebaseerd op vertrouwen. Als het mogelijk zou zijn, compilet het. Blijkt er toch geen Rechts in te zitten, dan krijg je een runtimeError.

Rechts nieuwDier = (Rechts)mijnDier;

Als je een functie wil inzetten moet je extra haken zetten.
(Rechts)mijnDier.mijnMethode(); Cast het resultaat van de methode van mijn Dier. Waarschijnlijk FOUT.
((Rechts)mijnDier).mijnMethode(); Cast het mijnDier naar een Rechts, en kan dan de methode van rechts aanroepen.

Een **interface** is een **superabstracte** klasse. Zo superabstract dat het GEEN **Klasse** is.

interfaces kunnen alleen abstracte methoden hebben. Deze zijn impliciet **public en abstract**.

Dus bij het implementeren van de methoden MOET er public voor de methode staan. EEN EXAMKILLER.

Een klasse kan **meerdere** interfaces **implementeren**.
Een **interface** kan meerdere **interfaces extenden**.

Een klasse kan **geen** interface extenden.
Een interface kan **geen** interface implementeren.
Een interface kan **geen** klasse implementeren.

Een field in een interface is **public static final**, impliciet.

Door het implementeren van een interface ontstaat er een **IS-A** relatie met de interface.

De erflijn van klasses mag **abstract - concreet - abstract - concreet** zijn.

Een methode in een interface kan sinds Java 8 een default implementatie hebben.
interface MetDefault{ default void metDefault(){ System.out.println("met Default"); } }
hiermee VERDWIJNT de verplichting om de methode te implementeren wanneer je interface implements

Een methode in een interface kan sinds Java 8 een static implementatie hebben.
interface MetDefault{ static void metDefault(){ System.out.println("met Default"); } }
Dan is er GEEN verplichting om te implementeren MAAR de methode is NIET via een instantie aan te roepen.
Alleen via InterfaceNaam.methodeNaam(), dus MetDefault.metDefault();

Lambda expressies
De lambda expressie is een argument waarmee functionaliteit kan worden meegegeven aan een toekomstig verzamelingselement, zoals een stream.
Een lambda expressie ziet er als volgt uit

element -> element < 24

Als je een typering wilt weergeven zal dit tussen haken moeten, Ook als je niks met het element wilt doen, en dus geen identifier toekent zul je haakjes voor de pijl moeten hebben.

(Hond deHond) -> deHond.aantalPootjes < 4
() -> 10
Als je meerdere statements wilt uitvoeren moet je achter de pijl accolades gebruiken.
element -> { System.out.println(element); return element < 3; }

==========

Hoofdstuk 7

Exceptions bestaan uit 3 soorten

- Checked Exceptions: Hiervoor gelden alle regels zoals try, catch of throws. Het zijn voorstelbare uitzonderingen die op zouden kunnen treden.
- Unchecked Exceptions (RuntimeExceptions): Deze horen niet voor te komen. Een try, catch of throws kan wel, maar dwingt niets af.
- Errors: Een totaal andere stamboom zorgt voor de vaak voorkomende fout dat hij niet gecatched wordt door catch(Exception e)

Deze tabel zal je vele punten opleveren.

Errors:				
Throwable	Error	LinkageError	**ExceptionInInitializerError**	
Throwable	Error	LinkageError	**NoClassDefFoundError**	
Throwable	Error	VirtualMachineError	**StackOverflowError**	
Throwable	Error	VirtualMachineError	**OutOfMemoryError**	
Runtime exceptions or unchecked exceptions:				
Throwable	Exception	RuntimeException	IndexOutOfBoundsException	**ArrayIndexOutOfBoundsException**
Throwable	Exception	RuntimeException	**IndexOutOfBoundsException**	
Throwable	Exception	RuntimeException	**ClassCastException**	
Throwable	Exception	RuntimeException	IllegalArgumentException	**NumberFormatException**
Throwable	Exception	RuntimeException	**IllegalArgumentException**	
Throwable	Exception	RuntimeException	**IllegalStateException**	
Throwable	Exception	RuntimeException	**NullPointerException**	
Checked Exceptions (uit de API):				
Throwable	Exception	IOException	**FileNotFoundException**	
Throwable	Exception	**IOException**		
Throwable	Exception	(ReflectiveOperationException)	**ClassNotFoundException**	

Het onderstaande geldt voor Checked Exceptions. Het is ook mogelijke met de andere twee soorten ECHTER ze dwingen geen compilefouten af als het niet wordt opgevolgd.

Bij een Checked Exception moet je OF:

- Het met een try en catch block afvangen.
- Of het via throws in de methode door moeten geven naar de aanroepende methode

Bij overloading is sprake van compleet verschillende functies dus hier heeft het **geen invloed**.

Bij **overriding van een normale methode** mag het kind alleen een kleinere of gelijke throwsen dan zijn ouder.
Bij **overriding van een Constructor** mag het kind alleen een grotere of gelijke throwsen dan zijn ouder.

Een **try block heeft altijd** OF een catchblok nodig OF een finally block OF allebei.

Het **finally block wordt altijd** uitgevoerd. (behalve bij System.exit)

Als een Exception niet gecatched wordt/afgehandeld wordt dan stopt de executie van het programma **meteen**.

Bij **meerdere catch-blokken** moet je altijd van Klein naar groot opvolgen.

Een Exceptie die niet gecatched wordt geeft een **StackTrace** als output. Dan toont hij welke aanroepende methodes door de fout vastlopen.

Als zowel het try-block een exception throwed, en het catch block deze rethrowed en de finally een andere exception throwed. Wordt de andere exception **gethrowed** daar een methode maar 1 exception kan **uitgooien**. (Ook zal finally eventuele variabelen overschrijven)

LET OP: variabelen die in het try-block gemaakt worden zijn locale variabelen. Na het try-block zijn deze **DOOD**. catch en finally kunnen hier **niet bij**.

In geval van throws in de methode, **verschuif** je het probleem naar de **aanroepende methode**. Die heeft weer de keuze. OF afhandelen met try and catch of door throwsen. Ander compilefout.

Exceptions zijn normale klasses, dus kunnen ook eigen varianten worden gemaakt met behulp van **extends**.

LET DAN GOED OP of ze een unchecked, checked of error extenden! EXAMKILLER

Exception worden ook ingezet om je af te leiden! Dan gaat de vraag helemaal niet over exception-gedrag.

Alles stamt af van **Throwable**.
==========

Tips:
- Concentreer je goed.
- Neem je ID mee en een bankpas met je naam (Dubbele identificatie).
- Eet goed, drink goed, zorg dat je op het moment top bent in concentratie.
- Houd de tijd goed in de gaten, ben niet verrast als de twee uur voorbij zijn. Heb een strategie voor als je ernstig achter komt te liggen.
- Maak notities bij rekenopdrachten.
- Maak tekeningen bij argumenten vragen, garbage collection vragen.
- Zorg dat je 10 minuten voor de tijd-einde kunt checken of je alle vragen wel voldoende antwoorden hebt gegeven. Dit zie je in het overzicht.
- Zorg dat je kunt inloggen na je examen, want je uitslag staat in je Oracle account. Op zeker dat je je wachtwoord niet meer weet als je heel nerveus bent.
- Je kunt naar de WC. (1 voor 1).
- Je mag een nieuw notitieblaadje vragen MAAR OFFICIEEL moet je dan de oude inleveren. Dit verschilt echter per locatie, vraag dit dus vooraf na.
- Je kunt buiten het lokaal drinken nemen.
- Eten, drinken, sleutels, telefoons, jassen, vesten zijn verboden in het lokaal.
- Denk goed na, lees goed, maar probeer niet slimmer te zijn dan de Examencommissie.
- Heb vertrouwen in wat je geleerd hebt, laat je niet van de wijs brengen. Het aantal instinkers blijft beperkt.