*Java Essentials Samenvatting*

*Het is ten zeerste aangeraden voorals het examen te beginnen, nog een keer door de samenvatting van Programming Basics te lezen.*

Klassen hebben eigenschappen en methoden (voornamelijk getters en setters), Een programma heeft meestal meerdere klassen waarvan meestal slechts 1 een main() heeft.

*Declaratie:*

*Klassen:*

public De klasse kan gebruikt worden door andere klassen. Indien niet vermeld kan deze alleen gebruikt worden binnen de package.

abstract Er kunnen geen concrete objecten van worden gemaakt.

final Er kunnen geen klassen van afgeleid worden.

class # Declareert een klasse met naam #.

extends # De klasse is afgeleid van klasse #.

implements # Gebruikt de interface # (meerdere worden gescheiden met een komma).

Standaard is een klasse niet publiek, niet abstract, niet final, een subklasse van object is en geen interfaces implementeert.

*Eigenschappen:*

static Eigenschap is een klassenvariabele en geen instance-variabele.

final De variabele is onveranderbaar.

# declareert het type #.

# declareert met naam #.

= # initialiseert met waarde # (standaard 0, null of false).

*Methoden:*

static Methode is een klassenmethode en geen instance-methode.

abstract Moet overridden worden door een subklasse (alleen in abstracte klassen).

final De methode kan niet overridden worden door subklassen.

# Het datatype dat wordt teruggegeven of void.

# De naam van de methode.

(#, #) De parameters die meegegeven worden.

throws exceptions Bepaalt de exceptions (zie verder voor uitleg).

*Klassenomschrijving (body):*

*Eigenschappen:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Toegangsniveau | eigen klasse | pakketklassen | subklassen | alle klassen |
| private | X |  |  |  |
| - (package/friendly) | X | X |  |  |
| protected | X | X | X |  |
| public | X | X | X | X |

Eigenschappen kunnen beter private gemaakt worden omdat er anders geen controle op de ingave uitgevoerd kan worden.

*Methoden:*

D.m.v. methoden kunnen andere objecten boodschappen sturen en eventueel een resultaat ontvangen.

this. verwijst naar de variabele in het huidig object en niet naar de binnen de scope gedeclareerde variabele.

Er kunnen meerdere returns voorkomen in 1 methode. Wanneer de compiler 1 van deze returns tegen komt, stopt de code onmiddellijk en wordt de huidige return uitgevoerd.

Method name overloading is verschillende methoden dezelfde naam geven maar een verschillend aantal en/of soort parameters.

*Constructors:*

Een constructor is de code die wordt uitgevoerd als het object gecreëerd wordt of geconstrueerd wordt met de new-operator. Het lijkt op een methode maar heeft geen return-type en de naam komt overeen met de naam van de klasse. Deze kan slechts 1 keer per object opgeroepen worden.

Constructor overloading is verschillende constructors met een verschillend aantal en/of soort parameters.

Wanneer er geen constructors gedeclareerd worden, neemt de klasse de standaard-constructor. Zodra 1 gedeclareerd wordt, vervalt de standaard-constructor.

Indien een constructor een andere constructor aanroept, moet dit altijd in de eerste regel gebeuren.

*Instance:*

Het is telkens nodig te specifiëren over welk object het gaat. Instance-variabelen kunnen geïnitialiseerd worden tijdens de declaratie, in een initialisatieblok of in een constructor.

Methoden kunnen gebruik maken van zowel instance- als klasse-variabelen.

*Klasse:*

Is hetzelfde voor alle objecten van die klasse. De component static zorgt ervoor dat het tot een klasse en niet tot een object behoort. Klasse-variabelen kunnen geïnitialiseerd worden tijdens de declaratie of in een static initialisatieblok.

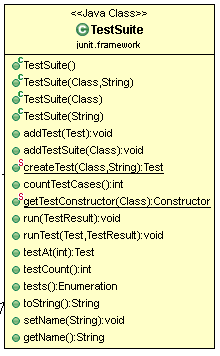
Methoden kunnen enkel gebruik maken van klasse-variabelen en variabelen binnen de scope van de methode.

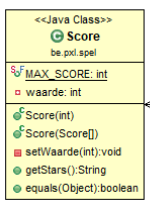
Het is mogelijk static members te importeren wat het vermelden van de klassenaam overbodig maakt.

*Utility-klassen:*

Een voorbeeld hiervan is de Math-klasse. Deze klassen hebben enkel maar static members en men kan geen objecten van deze klasse maken. Dit komt omdat de constructor private is en dit ook niet de bedoeling is of nut heeft.

*UML-diagram:*





*Speciaal if/else statement:*

CONTROLE?RETURNalsTRUE:RETURNalsFALSE;

Voorbeeld zodat men geen negatieve hoogte kan instellen:

this.height = height<0?-height:height;

*Soorten relaties:*

Het is de bedoeling zoveel mogelijk bestaande code te hergebruiken. Dit doen we door relaties te leggen tussen klassen. Het principe High Cohesion zegt ons dat we een klasse maar tot 1 kerntaak moeten beperken zodat deze makkelijker en meer herbruikbaar is.

*Associaties:*

Een object maakt gebruik van een ander object. Er is meestal 1 referentie (unidirectioneel) van de ene klasse naar de andere en niet omgekeerd. Dit is soms bidirectioneel en dus refereren de beide klassen naar elkaar. De mogelijkheid om van de ene klasse met de andere klasse te spreken, noemt met navigeerbaarheid. In het UML-diagram wordt dit weergegeven door een pijltje.

*Aggregaties:*

Het ene object is eigenaar van het andere object. Dit is een bijzondere vorm van een associatie. In het UML-diagram wordt dit weergegeven door een open ruit. Ze kunnen zonder elkaar bestaan.

*Composities:*

Het ene object kan niet zonder het andere object bestaan. Als het ene object stopt met bestaan, stopt het andere object onherroepelijk ook met bestaan.

*Overerving en klassenhiërarchie:*

We kunnen code hergebruiken (code reuse) door overerving tussen super- en subklassen. Subklassen erven alle eigenschappen en methoden, ze kunnen eigen methoden toevoegen of vervangen (@Override). Een subklasse tonen we aan door extends te gebruiken. met super verwijzen we naar de superklasse. Met this verwijzen we naar de huidige klasse.

Er kunnen geen objecten van abstracte klassen gemaakt worden. Ze bestaat om te zeggen hoe een object er abstract, vaag of algemeen uit ziet. Enkel abstracte klassen kunnen abstracte methoden hebben.

Bij een methode Override moet het return-type niet exact hetzelfde zijn maar mag dit een subklasse zijn (covariant return type). Dit geldt alleen voor objecten en niet voor primitieve datatypes. Private methoden kunnen niet vervangen worden.

Annotaties zijn @’s met tekst achter. Ze bieden de mogelijkheid meta-informatie toe te voegen die door allerlei tools of runtime-omgevingen (EDI’s) gebruikt kunnen worden.

Polymorfisme is tegen de compiler zeggen wat het is, maar er stiekem eigenlijk iets anders van maken. Als we het object als Shape definiëren, kan kunnen we ook alleen maar methoden met betrekking tot Shape gebruiken. De concrete koppeling gebeurt dus niet bij de compilatie maar bij de echte uitvoering (late binding). Dit geldt enkel voor methoden en niet bij eigenschappen of statische methoden.

Constructors worden niet mee overgeërfd. Indien geen expliciete constructor wordt gedefinieerd, dan krijgt deze de standaard constructor mee. Dit werkt enkel als de superklasse een constructor zonder parameters heeft.

Final klassen zijn klassen die af zijn en waarvan geen subklasse meer gemaakt kan worden. Dit is voordelig ten gevolge van de beveiliging, ontwerp en snelheid.

Iedere subklasse is een subklasse van de superklasse Object. Dit bepaalt de algemene toestand en gedrag van het object. Het hoogste element in de boomstructuur van de klassen is Object.

instanceof helpt de compiler bepalen of een object een instantie is van een (sub)klasse of interface. Het object moet zich in dezelfde hiërarchische tak bevinden, anders foutcode! Een downcast is het forceren van een referentievariabele naar een anders datatype.

toString() is een methode van Object die een tekstuele voorstelling van het object teruggeeft.

equals() en hashCode() zijn methoden van Object die nagaan of 2 objecten hetzelfde zijn.

equals() moet reflexief (x.equals(x) = true), symmetrisch (x.equals(y) = true && x.equals(x) = true), transitief (x.equals(y) && y.equals(z) → x.equals(z) = true), consistent (inhoud ongewijzigd = zelfde resultaat van equals()), x.equals(null) moet steeds false zijn.

hashCode() moet consistent zijn, als volgens equals() 2 objecten aan elkaar gelijk zijn, moeten ze dezelfde hashcode hebben. als volgens equals() 2 objecten niet aan elkaar gelijk zijn, moeten niet noodzakelijk een andere hashcode hebben.

Associaties hebben een HEEFT EEN relatie terwijl overerving een IS EEN relatie hebben. Vaak hebben we een combinatie van beide technieken om code reuse te maximaliseren.

*Opsomming - Enumeratie:*

Dit is een subklasse van enum van de klasse Enum. Er kan geen verdere subklasse van enum gemaakt worden. Er zijn een vast bepaald aantal instanties beschikbaar, we kunnen zelf geen nieuwe instanties maken. class wordt vervangen door enum.

Iedere opsomming is impliciet een subklasse van de klasse Enum en erft dan ook de methoden van de klasse over. 2 van deze methoden zijn name() en ordinal().

We kunnen de instanties van een enumeratie afgaan door een for each loop:

for (Klasse i : Klasse.values()) {

System.out.println(i);

}

Men kan ook 1 of meerdere elementen toevoegen door:

import static Klasse.\*;

We kunnen private eigenschappen toevoegen en initialiseren door middel van een constructor.

De constructor is altijd impliciet private. Niemand kan nadien nog instanties toevoegen. Als er geen constructor is, zal de default constructor gebruikt worden (dan mogen de ronde haken bij de opsomming weggelaten worden).

Een element uit de opsomming kan ook een bepaalde methode vervangen:

ZWART {

public String toString() {

return “Zwart”:

}

},

WIT,

etc etc

Het is toegelaten een methode als abstract te declareren en deze vervolgens afzonderlijk te implementeren voor ieder object.

*Wrapper-klassen:*

Voor de door ons gekende primitieve datatypes bestaat er telkens een wrapper-klasse die een object van een primitief datatype voorsteld. Voorbeelden zijn:

byte Byte

short Short

int Integer

long Long

float Float

BigInteger

BigDecimal

boolean Boolean

char Character

Met de methode intValue() kunnen we de primitieve waarde uit een wrapper-klasse halen. Het is nodig om met equals() te werken i.p.v. ==.

Autoboxing is de omzetting tussen een primitief datatype en het wrapper-object. (i++ werkt bij een wrapper-klasse door automatische autoboxing. Opgelet! Voor booleans en getallen die in 8 bits passen, werkt == wel!

Widening heeft altijd voorrang op autoboxing. De compiler gaat niet widening en autoboxing doen. Hij doet 1 van de 2. Ze alle 2 is een brug te ver voor de compiler.

Constanten van wrapper-klassen:

MAX\_VALUE maximale waarde

MIN\_VALUE minimale waarde

NaN Not a Number

NEGATIVE\_INFINITY min oneindig

POSITIVE\_INFINITY plus oneindig

Methoden van wrapper-klassen:

parseInt() van String naar int

isNaN() boolean voor Not a Number

isInfinite() boolean voor oneindig

*Datums en tijd in Java:*

Date en Calender zijn verouderd, hebben onvolkomenheden en beperkingen. Java heeft een oplossing gevonden door nieuwe klasse te maken die zich in het pakket java.time bevinden.

De klasse Instant telt het aantal nanoseconden vanaf de EPOCH (01/01/1970 00:00:00).

Methoden van Instant:

getEpochSecond() geeft het aantal seconden sinds EPOCH

getNano geeft het aantal nanoseconden na de laatste seconde

isAfter() boolean voor datum later dan meegegeven parameter

isBefore() boolean voor datum vroeger dan meegegeven parameter

minusSeconds() geeft instantie terug aantal seconden vroeger

plusSeconds() geeft instantie terug aantal seconden later

Als er een bewerking op de tijd wordt gedaan, wordt er een nieuwe instantie aangemaakt. De klasse Instant heeft dus geen setters.

Eigenschappen van Instant:

EPOCH tijdstip van EPOCH

MAX maximaal tijdstip  
MIN minimaal tijdstip

Er is een klasse Month om met maanden te rekenen en een klasse DayOfWeek om met week-dagen te rekenen.

LocalDateTime is een samenstelling van LocalDate en LocalTime.

ZoneId slaagt tijdzones op. ZoneOffset slaagt op hoeveel uren hij van de UTC-tijd zit. Dit kan door ZoneOffset.ofHours(#). De tijdzone van het systeem kan zo benaderd worden door ZoneId.systemDefault().

ZonedDateTime geeft een datum en tijd rekening gehouden met de tijdzones.

Tijdsduur berekenen:

Duration voor instanties van de klasse Instant

ChronoUnit voor dingen zoals weken, dagen, maanden etc (ChronoUnit.DAYS.between())

Period voor menselijke notaties zoals dagen, maanden, jaren etc.

DateTimeFormatter kan de tijd in een bepaald formaat krijgen door parse() of naar een bepaald formaat omzetten door format().

Formatter codes:

tY 1999

ty 99

tB januari

tb jan

tm 01

td 28

tH uren (24-uurs-formaat)

tl uren (12-uurs-formaat)

tM minuten

tS seconden

BEKIJK DE JAVA DOC VOOR MEER INFO

Men kan omzetten van en naar Date en Calender door to-methoden of from-methoden.

*Interfaces:*

Een interface is een soort abstracte klasse die alleen maar abstracte methoden heeft en geen concrete implementaties. Er kunnen nu ook default en static methoden voorkomen. Een interface zegt dat deze methoden geimplementeerd moeten worden. Een klasse kan meerdere interfaces implementeren. Een klasse kan niet meerdere superklassen hebben. Het is gebruikelijk de namen te laten eindigen op able. Door extends kan een interface meerdere interfaces overerven.

public static final mag weggelaten worden (eigenschappen).

public abstract mag weggelaten worden (methoden).

Een klasse implementeert een interface door implements InterfaceName.

Default methoden zijn handig omdat een klasse van niets kan doen, opnieuw de methode abstract declareren of opnieuw implementeren (overridden). Dit is handig om methoden toe te voegen die niet direct maar na een bepaalde tijd pas gebruikt worden in de klasse.

Statische methoden moeten opgeroepen worden d.m.v. de InterfaceNaam en niet de KlasseNaam.

Men kan een array van een interface maken en er objecten in plaatsen die de interface implementeren.

*Exceptions:*

Een exception is een uitzondering of fout die het normale verloop van het programma verstoort. Als zo een uitzondering optreed, wordt er een exception-object gecreëerd. Deze bevat een omschrijving en een toestand. Dit wordt overhandigd aan het runtime system = throwing an exception. Een exception handler is een methode die in staat is de exception af te handelen. Een exception gaat in omgekeerde volgorde door de methodes tot hij een methode vindt, die de exception afhandeld = call stack. Indien niet gevonden, eindigd het programma.

Enkele voordelen zijn dat we fouten kunnen groeperen en gescheiden houden van normale code zodat deze overzichtelijker en makkelijker onderhoudbaar is. Verschillende soorten uitzonderingen samenbrengen onder eenzelfde noemer = klassenhiërarchie.

Exceptions opvangen:

try {

//CODE

}

catch (Throwable exceptionObject) {

//CODE

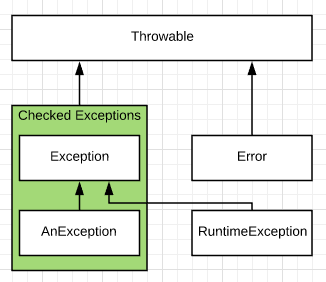
}

Een exception-object moet afgeleid zijn van de klasse Throwable (of een subklasse). De referentie naar het object krijgt meestal de beginletters van de uitzondering. NumberFormatException -> nfe. Men kan meerdere exceptions opvangen door meerdere catch’en onder elkaar te schrijven. Het is belangrijk de meest specifieke uitzondering helemaal bovenaan te zetten. Een catch die alle uitzonderingen opvangt is catch (Exception e) {}, dit is niet aangeraden omdat men dan niet weet welke fout er zich heeft voorgedaan.

Men kan verschillende types opvangen in 1 catch-blok door een | tussen de uitzonderingen te zetten. Een finally-blok wordt ALTIJD uitgevoerd, wat er ook gebeurt.

Exceptions genereren of gooien kan door throw throwableObject; Als we zelf een klasse willen maken die de exception afhandeld, gebruiken we: throw new ExceptionClass();

Soorten exceptions:



Error’s zijn fundamentele fouten die normaal niet morgen voorkomen. Bij checked exceptions controleert de compiler of ze in de loop van het programma worden opgevangen. Bij runtime exception controleert de compiler dit niet.

Voorbeeldcode eigen Exception-klasse:

public class CostumException extends RuntimeException {

public CostumException() {

super();

}

public CostumException(String message) {

super(message);

}

public CostumException(String message, Throwable cause) {

super(message, cause);

}

public CostumException(Throwable cause) {

super(cause);

}

}

Men kan in een exception catchen en in het catch-blok verder gooien naar een andere handler door: throw exceptionNaam of het wrappen in een nieuwe exception door: throw new MyException(exceptionNaam);

*Java Documentatie:*

Dit is handig om later je eigen code te begrijpen of een collega-programmeur in jouw afwezigheid. De JDK bevat een tool JAVADOC om deze automatisch te genereren. We geven commentaar weer door: /\*\* #### \*/. De eerste volledige zin bepaald een samenvatting. De volgende zinnen zijn gedetailleerder. We kunnen ook gebruik maken van HTML-tags zoals <pre> en </pre>.

Tags:

@autor bepaald de auteur

@version bepaald de versie

@see bepaald de verwijzing naar een andere klasse, methode of eigenschap.  
 Syntax: @see package.classname#member

Documentatie gebeurt net voor een eigenschap, methode of constructor.

Bij methoden en constructors kan men de volgende tags gebruiken:

@param beschrijft de parameters

{@link class#member} inline-tag die verwijst naar een andere klasse

@return beschrijft de return-waarde

@throws beschrijft de exception

@see bepaald de verwijzing naar een andere klasse, methode of

eigenschap

Met de JAVADOC-tool kan men parameters meegeven:

-author vermelding van de auteur

-classpath path-list paden waar de tool andere klasse moet zoeken

-d path pad waar de doc gegenereerd wordt

-version vermelding van de versie

-sourcepath pad waar de broncode zich bevindt

-overview overzichtspagina